IEEE Transactions on Quantum Engineering

Received 19 March 2025; revised 9 October 2025; accepted 10 October 2025; date of publication 16 October 2025; date of current version 20 November 2025.

Digital Object Identifier 10.1109/TQE.2025.3622495

Leveraging Quantum Machine Learning Generalization to Significantly Speed up Quantum Compilation

ALON KUKLIANSKY[1], LUKASZ CINCIO[2], ED YOUNIS[3], AND COSTIN IANCU[3]

[1]Naval Postgraduate School, Monterey, CA 93943 USA
[2]Theoretical Division, Los Alamos National Laboratory, Los Alamos NM 87545 USA
[3]Applied Mathematics and Computational Research Division, Lawrence Berkeley National Laboratory, Berkeley, CA 94720 USA

Corresponding author: Ed Younis (e-mail: edyounis@lbl.gov).

The work of Alon Kukliansky was supported by the U.S. Naval Postgraduate School. The work of Lukasz Cincio was supported by the Laboratory Directed Research and Development program of Los Alamos National Laboratory under Grant 20230049DR and Grant 20230067DR. The work of Ed Younis and Costin Iancu was supported by the U.S. Department of Energy (DOE), Office of Science, Office of Advanced Scientific Computing Research under Contract DE-AC05-00OR22725 through the Accelerated Research in Quantum Computing Program MACH-Q project. This research used resources of the National Energy Research Scientific Computing Center (NERSC), a U.S. DOE Office of Science User Facility located at Lawrence Berkeley National Laboratory, operated under Contract DE-AC02-05CH11231 using NERSC under Award DDR-ERCAPm4141.

ABSTRACT Existing numerical optimizers deployed in quantum compilers use expensive $\mathcal{O}(4^n)$ matrix–matrix operations. Inspired by recent advances in quantum machine learning, QFactor-Sample replaces matrix–matrix operations with simpler $\mathcal{O}(2^n)$ circuit simulations on a set of sample inputs. The simpler the circuit, the lower the number of required input samples. We validate QFactor-Sample on a large set of circuits and discuss its hyperparameter. When incorporated in the BQSKit quantum compiler and compared against a state-of-the-art domain-specific optimizer, we demonstrate improved scalability and reduced compile time, achieving an average speedup factor of 69 for circuits with more than eight qubits. We also discuss how improved numerical optimization affects the dynamics of partitioning-based compilation schemes, which allow a tradeoff between compilation speed and solution quality.

INDEX TERMS Quantum compilation, quantum machine learning.

I. INTRODUCTION

Given a parameterized quantum circuit and a target unitary, a common operation in quantum program development is to solve an optimization problem to determine the parameters that implement the target unitary. Solving for parameters is commonly referred to as *instantiation*, and it is an operation that appears in hybrid algorithms [1], [2], circuit synthesis [3], [4], [5], [6], [7], [8], [9], or within quantum machine learning (QML) [10], [11], [12], [13] algorithms.

In all existing approaches, the objective function in instantiation requires computing process distances [9] between two unitaries, an operation with $\mathcal{O}(4^n)$ complexity. As far as we know, the state of the art is illustrated by the *QFactor* [14] domain-specific optimizer, which uses a tensor network formulation together with analytic methods and an iterative local optimization algorithm to reduce the *effective* number of problem parameters. The improvements over general-purpose optimizers (GPOs) come, among other features, from working at the unitary rather than the parameter level. A given gate may have a very complicated representation in terms of parameters that need to be resolved by GPOs. In contrast, *QFactor* optimizes that gate on each update.

QFactor improves performance and scalability by reducing the number of parameters by a (large) constant. In this article, we show how to further gain $\mathcal{O}(2^n)$ speedup in computational complexity for the *QFactor*'s inner loop, while maintaining the same quality of results (QoRs). This speedup arises from replacing repeated matrix–matrix multiplications with a small number of matrix–vector multiplications, thereby avoiding the dominant $2^n \times 2^n$ operations in favor of lighter $2^n \times 1$ products. When comparing *QFactor-Sample* with *QFactor*, our benchmarks show an average reduction of $17\times$ in runtime and an impressive $69\times$ average reduction for circuits with 9–12 qubits. We have seen a runtime reduction of up to $830\times$ for individual instantiation runs.

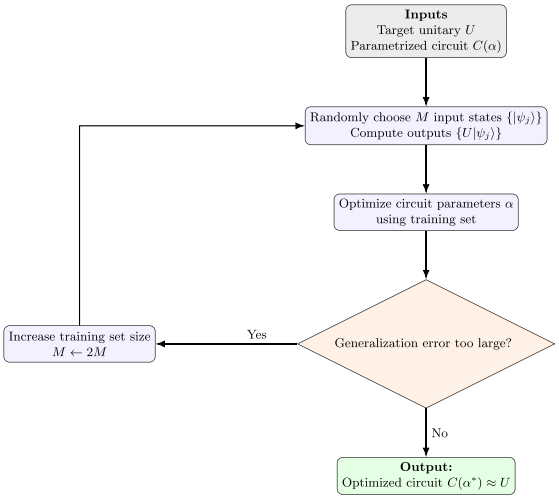

FIGURE 1. High-level flow of the *QFactor-Sample* algorithm. Starting from a target unitary U and a parameterized circuit $C(\alpha)$, the algorithm randomly samples training states and computes their transformed outputs. Circuit parameters are optimized using this training set, with an adaptive increase in the training set if the generalization error is too large. The final result is an optimized circuit $C(\alpha^*)$ instantiating U. The speedup over *QFactor* arises from replacing costly matrix–matrix multiplications ($O(4^n)$) with a small number of vector–matrix multiplications ($O(M2^n)$).

The basic idea is taken from recent advances [15], [16], [17], [18] in QML theory. Modern QML methods involve variationally optimizing a parameterized quantum circuit on a training dataset and subsequently making predictions on unseen data (i.e., generalizing). It has been shown that for a quantum circuit with T parameterized gates that has been trained on M samples, the generalization error is bounded by $\mathcal{O}(\sqrt{\frac{T \log T}{M}})$. We use a reduction from the instantiation problem to a traditional QML flow, linking the generalization error to the instantiation error and taking advantage of this bound to limit the size of the training set, reducing the overall complexity. This approach will yield significant performance improvements, as many unitaries have some inner structure that we can use.

QFactor-Sample is an instantiation algorithm using a QML approach that takes advantage of structure in common unitaries, allowing it to train over a sample of states. Given a target unitary, it randomly draws M orthogonal states and applies the unitary on them to create the training set. It then performs optimization based only on that set. It has an *on-the-fly* mechanism that can increase the size of the training set if it detects that the generalization error is too big. The simpler the unitary, the fewer the training examples required. In contrast, other optimizers use the full target unitary, with 2^n states to perform the optimization. The overall workflow of this process isillustrated in Fig. 1, which shows the adaptive training procedure used to balance accuracy and efficiency.

We integrated *QFactor-Sample* into the BQSKit [6] synthesis infrastructure and evaluated its performance in optimizing large quantum circuits. Compared to the next-best optimizer, our results show an average 4–9× reduction in runtime. Furthermore, by increasing the partition size beyond the capability of other optimizers, we observe an improvement in QoRs, although this comes with the tradeoff of increased runtime.

The rest of this article is organized as follows. Section II provides background on the *QFactor* instantiation algorithm and how to utilize QML generalization error bound to improve instantiation. The *QFactor-Sample* algorithm and its implementation are described in Section III. Our evaluation procedures and results are presented in Section IV. Finally, in Section V, we discuss our results.

II. BACKGROUND

A. NUMERICAL OPTIMIZATION AND INSTANTIATION

Given a $2^n \times 2^n$ unitary U and a parameterized circuit $C : \mathbb{R}^k \mapsto \mathcal{U}(2^n)$, instantiation finds parameters $\boldsymbol{\alpha}$ such that some (e.g., Frobenius) norm between U and $C(\boldsymbol{\alpha})$ is minimized

$$||U - C(\boldsymbol{\alpha})||^2 = 2^{n+1}\left(1 - \frac{\mathrm{ReTr}[U^\dagger C(\boldsymbol{\alpha})]}{2^n}\right). \tag{1}$$

The heaviest part of any existing instantiation algorithm is determined by the $\mathcal{O}(4^n)$ computational complexity of the norm in (1) (or part of it) calculation, where n is the number of qubits in the circuit.

As far as we know, *QFactor* [14] is the state of the art in numerical optimization for instantiation. It is a domain-specific optimizer that reduces circuit parameter complexity by directly updating (possibly multiqubit) unitary gates without parameterizing them internally. By treating each gate as a unitary without parameterization during optimization, it effectively reduces the parameter space, distinguishing it from conventional GPOs.

QFactor utilizes tensor network contractions for efficient circuit manipulations. The algorithm sweeps through a circuit, traversing gate (tensor) by gate (tensor), and at each gate, it executes a local optimization process to update the gate's unitary. The basic step in the sweep is the calculation of $\mathcal{E}$, the gate's environment matrix [14], [19]. The algorithm then performs a singular value decomposition (SVD) on $\mathcal{E} = XDY^\dagger$ resulting in an optimized gate

$$u_{k_{\text{new}}} = YX^\dagger. \tag{2}$$

When compared to GPOs [20], [21], [22], *QFactor* scales better with circuit depth and qubit count, achieving faster runtimes and better success rates.

Numerical optimization can be scaled in two ways: by reducing the number of parameters needed to describe a problem, or by decreasing the computational effort required per parameter instantiation. *QFactor* enhances scalability through the former approach. A key question now is whether we can develop a more efficient optimization procedure, specifically by lowering the cost of the objective function.

B. QUANTUM GENERALIZATION ERROR BOUND

A variational quantum circuit $C(\boldsymbol{\alpha})$ is a circuit with parameters that are adjusted iteratively to optimize a specific

IEEE Transactions on Quantum Engineering

objective function. If its parameters $\boldsymbol{\alpha}$ are trained with respect to some loss function l over a training set $S = \{(|\psi_i\rangle, |\phi_i\rangle)\}_{i=1}^{M}$, then the training loss or training error is usually defined as

$$l_{\text{train}} = \frac{1}{M}\sum_{i=1}^{M} l(C(\boldsymbol{\alpha})|\psi_i\rangle, |\phi_i\rangle). \tag{3}$$

For a given set of parameters $\boldsymbol{\alpha}$, the expected prediction error over some distribution of states ξ is

$$l_{\text{pred}} = \mathop{\mathbb{E}}_{|\psi\rangle,|\phi\rangle\sim\xi}[l\,(C(\boldsymbol{\alpha})|\psi\rangle, |\phi\rangle)]. \tag{4}$$

The generalization error is defined as the difference between the expected prediction error (4) and the training error (3), where ξ is the distribution from which the training set was drawn

$$\text{gen}(C(\boldsymbol{\alpha})) = l_{\text{pred}} - l_{\text{train}}. \tag{5}$$

There are many machine learning algorithms that, in addition to the training set, utilize a validation set for various purposes such as early stopping of the training process when overfitting is detected, finding optimal hyperparameters of the underlying training algorithm, or estimating the generalization error of the model [23].

It has been proven in [16] that for a quantum model with T parameterized gates that has been trained on M samples, the generalization error is bounded by

$$\text{gen}(C(\boldsymbol{\alpha})) \in \mathcal{O}\left(\sqrt{\frac{T\log T}{M}}\right). \tag{6}$$

This means that for circuits with n qubits, if their number of gates is polynomial with respect to n (poly-n), one does not need an exponential number of training states to achieve low generalization error, but rather only poly-n training states will suffice. The class of circuits that can be efficiently implemented on a quantum computer also has poly-n gates; therefore, we can effectively use the generalization bound to limit the number of states used during training, thus improving the training runtime performance. This reduction in runtime is expected to improve as the circuit has more qubits in the same way that $\frac{\text{poly-n}}{\text{exp-n}}$ shrinks as n increases.

C. QML AND INSTANTIATION

Quantum circuit instantiation can be reduced to a conventional QML flow. For a given unitary U, we generate a training set $\{(|\psi_i\rangle, U|\psi_i\rangle)\}_{i=1}^{M}$ by first randomly selecting M mutually orthogonal states $\{|\psi_i\rangle\}_{i=1}^{M}$ as input training states. Then, we apply the unitary U to these states to obtain the output training states. In the following, we define the loss function to be the distance square between the two states:

$$l(|\psi\rangle, |\phi\rangle) = ||\,|\psi\rangle - |\phi\rangle||^2 \tag{7}$$

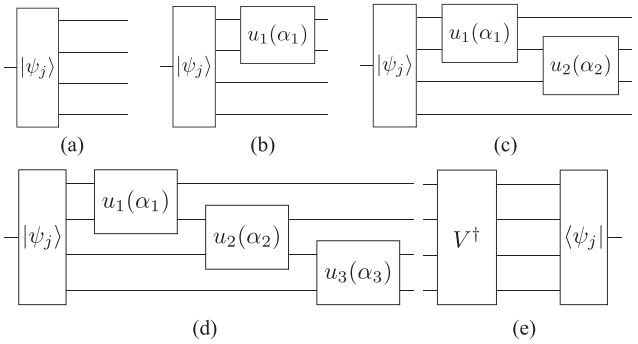

FIGURE 2. Tensors precomputation: Tensors that are precomputed and saved before each algorithm sweep. Recalculating the A and B_0 tensors is done only when the training set $\{|\psi_j\rangle\}_{j=1}^{M}$ is updated. The precomputation is a time–memory tradeoff, where we save intermediate computation results for later use and do not recompute them each time we calculate the environment matrix of a different gate. (a) B_0. (b) B_1. (c) B_2. (d) B_3. (e) A.

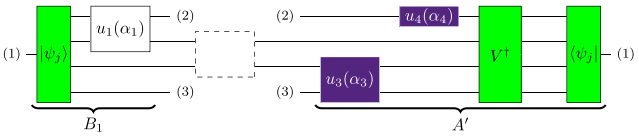

FIGURE 3. The environmental matrix calculation for the second gate in a four-gate circuit. Tensor legs with the same number will be traced together. Gate tensors colored in purple indicate parameters updated in the current sweep. Those in white denote gate tensors with parameters from the preceding sweep, while green represents tensors with consistent values across sweeps. We also note that for the tracing, we use the previously calculated B_1 tensor (see Fig. 2).

and the training loss to be the average loss over the training set

$$l_{\text{train}}(C, \boldsymbol{\alpha}) = \frac{1}{M}\sum_{i=1}^{M} l(C(\boldsymbol{\alpha})|\psi_i\rangle, U|\psi_i\rangle). \tag{8}$$

III. ALGORITHM

QFactor-Sample reduces *QFactor*'s computational complexity from $O(4^n)$ to $O(M2^n)$, where n is the number of qubits and M is the number of states required for training, by making use of known bounds [16] on the generalization error. The algorithm finds the optimal circuit parameters for only a small set of training states, and the expected error on all the other possible input states is bounded by (6). Although *QFactor-Sample* and *QFactor* have different cost functions, their basic optimization step is the same, where they locally optimize each gate at a time, performing an SVD on the gate environment matrix.

The algorithm begins by generating a random orthogonal set of training states, $\{|\psi_j\rangle\}_{j=1}^{M}$, sampled from the Haar random distribution [24]. Then, it sweeps the circuit from right to left. For each gate, it calculates the environment matrix $\mathcal{E}$, performs an SVD on $\mathcal{E}$, and updates the gate using (2). The sweep is repeated several times until a stopping condition has been reached. Fig. 3 presents the $\mathcal{E}$ calculation and the

algorithm state during a sweep. A short discussion about the importance of the training states distribution can be seen in Appendix C.

Since we do not know up front the number of training states required, the algorithm incorporates an "on-the-fly" approach for estimating the generalization error. This is achieved by comparing the cost values for the training states with those of a randomly selected set of validation states. If the observed error surpasses a predefined threshold, the algorithm halts and restarts with double the number of training states. This iterative process terminates upon reaching the desired convergence threshold, detecting a plateau, or reaching the maximum limit of training states. In the latter case, *QFactor-Sample* reduces to the original *QFactor* algorithm since it will use 2^n states for training. We would like to point out that the *double-and-restart* process does not change the asymptotic complexity, as $2 + 4 + 8 + 16 + \cdots + m < 2m$.

In *QFactor*, the cost function (1) is the Frobenius norm between the target unitary and the instantiated circuit, whereas in *QFactor-Sample*, the cost function we use is the average distance between the states generated by applying the target unitary on the training states and the states generated by applying the instantiated circuit on the same training states

$$\frac{1}{M}\sum_{j=1}^{M}||U|\psi_j\rangle - C(\boldsymbol{\alpha})|\psi_j\rangle||^2 = 2 - \frac{2}{M}\mathrm{Re}\left(\sum_{j=1}^{M}\langle\psi_j|U^\dagger C(\boldsymbol{\alpha})|\psi_j\rangle\right). \tag{9}$$

To minimize the cost function, one can also maximize

$$\mathrm{Re}\left(\sum_{j=1}^{M}\langle\psi_j|U^\dagger C(\boldsymbol{\alpha})|\psi_j\rangle\right) \tag{10}$$

such that, for any specific gate u_i in the circuit, this can be written as $\mathrm{Re}(\mathrm{Tr}(\mathcal{E}u_i))$ (see also Fig. 3). This is exactly what *QFactor*'s inner step is maximizing [14, Sect. III-A]; hence, *QFactor-Sample* uses the same updates to the gates as in *QFactor*, and the update optimality proof [14, Sect. III-A1] holds here as well.

To lower the computation overhead of $\mathcal{E}$ calculations, we compute only once $\{\langle\psi_j|U^\dagger\}_{j=1}^{M}$ and denote it as the A tensor. Moreover, when we compute $\{C(\boldsymbol{\alpha})|\psi_j\rangle\}_{j=1}^{M}$, we save the intermediate computation results in a list and denote it as B (see Figs. 2 and 3).

The plateau detection mechanism in *QFactor-Sample* checks if the training cost function has not sufficiently improved over t consecutive iterations. In contrast, *QFactor* halts, indicating a plateau, if the cost function fails to improve enough in just a single iteration. This modification in *QFactor-Sample* can improve the QoRs by sacrificing some runtime efficiency. Given that *QFactor-Sample* is significantly faster than *QFactor*, this tradeoff is advantageous.

We implemented *QFactor-Sample* on graphical processing units (GPUs) using JAX [25], and our source code can be found in [26]. Some implementation details and a list of *QFactor-Sample*'s hyperparameters are provided in Appendix A. We emphasize that the exponential improvement in runtime arises not from hyperparameter tuning, but from a fundamental difference in our core operation: performing vector-to-matrix multiplication instead of matrix-to-matrix multiplication.

We think that the *double-and-restart* approach is a good heuristic to use when the minimal number of training states ($M_{\min}$) is unknown, as in the worse case, we use $2M_{\min}$ states, and the total amount of work is comparable to $4M_{\min}$. An interesting question is how to choose the initial number of training states and the overtrain threshold that controls the *on-the-fly* generalization error detection. For circuits with more than a few qubits and dozens of gates, we observed that the first two to three attempts had the same runtime, although they had different number of training states. This is due to GPU overheads, such as data transfer and code generation. Hence, starting with $M = 2$ is probably suboptimal in real-life circumstances.

Among all *QFactor-Sample*'s hyperparameters, we highlight *overtrain_ratio* and *number_of_training_states*, as they dictate the initial count of training states. If, during the instantiation, the normalized generalization error exceeds *overtrain_ratio*, the algorithm halts, doubles the number of training states, and restarts. The normalized generalization error is calculated by

$$\frac{c_{\mathrm{val}}}{c_{\mathrm{train}}} - 1 \tag{11}$$

where c_{val} and c_{train} represent the validation and training costs, respectively. The *doubling-and-restart* process continues until the number of training states reaches 2^n, where n denotes the number of qubits.

We want to highlight the connection between *overtrain_ratio*, *dist_tol* (the unitary distance threshold parameter), and the resulting distance between the target unitary and the instantiated circuit. When the instantiation is successful, the following two relations hold: $c_{\mathrm{train}} < d_{\mathrm{tol}}$ and $\frac{c_{\mathrm{val}}}{c_{\mathrm{train}}} \leq 1 + otr$, where d_{tol} and otr represent *dist_tol* and *overtrain_ratio*, respectively. Then, for a random state $|\psi\rangle$

$$||U|\psi\rangle - C(\boldsymbol{\alpha})|\psi\rangle||^2 < d_{\mathrm{tol}}(1 + otr) \tag{12}$$

holds with high probability. One might argue that minimizing otr is preferable, yet doing so necessitates employing more training states. A simple remedy to mitigate generalization errors involves marginally reducing d_{tol} while simultaneously increasing otr, ultimately leading to a faster instantiation, since we will need significantly fewer training states.

IEEE Transactions on Quantum Engineering

TABLE 1. Benchmarks Used and Their Gate Counts, Upper Bound of ∼200 000

Circuit	U3	CNOT	Circuit	U3	CNOT
adder9	64	98	grover5	80	48
add17	348	232	hhl8	3288	2421
adder63	2885	1405	shor26	20 896	21072
mult8	210	188	hub4	155	180
mult16	1264	1128	hub18	1992	3541
heis7	490	360	tfim8	428	280
heis8	570	420	tfim16	916	600
heis64	5050	3780	tfim400	88235	87670
qae11	176	110	qpe8	519	372
qae13	247	156	qpe10	1681	1260
qae33	1617	1056	qpe12	3582	2550
qae81	7341	4840			
qaoa5	27	42	vqe5	132	91
qaoa10	40	85	vqe12	4157	7640
qaoa12	90	198	vqe14	10792	20392

Name suffix represents the number of qubits in the circuit up to 400 qubits [14].

IV. EVALUATION

We assess the performance of the instantiation algorithm using two metrics: success rate, which indicates the proportion of circuits from the benchmarks where the algorithm succeeded, and runtime, representing the total time taken by the algorithm to complete, regardless of the reason for termination. We used the same benchmarks and evaluation setup that was used in [14] to enable easy comparison.

The benchmarks used can be seen in Table 1. They represent real circuits with 4–400 qubits, and varying depths of up to ∼200 000 gates. We performed the reinstantiate flow using 1727 random partitions from the aforementioned benchmarks and limited the runtime to 10 min for partitions with fewer than nine qubits and 2 h for the rest. In this flow, we take a partition of a circuit, calculate its unitary, and ask *QFactor-Sample* to instantiate that unitary using the original partition circuit structure.

We compare *QFactor-Sample* performance against the central processing unit (CPU) and GPU versions of *QFactor* and a leading general-purpose numerical optimizer, which we will refer to as CERES [27]. In our comparison, we denote the Rust (CPU) and Python+JAX (GPU) implementations of *QFactor* by QF-R and QF-J, respectively, and we denote *QFactor-Sample* GPU implementation as QFS-J.

We run *QFactor-Sample* evaluation on National Energy Research Scientific Computing Center's Perlmutter [28] supercomputer. We used Perlmutter's hybrid GPU–CPU nodes. Each node has one AMD EPYC 7763 64-core processor, 256-GB DDR4 DRAM, and four NVIDIA A100 GPUs; some have 40 GB of RAM, while others have 80 GB.

QFactor-Sample's hyperparameters used in the evaluation are: $dist\_tol = 10^{-10}$, $diff\_tol\_r = 10^{-3}$, $plateau\_windows\_size = 5$, $number\_of\_training\_states = 2$, $min\_iter = 6$, $overtrain\_ratio = 0.1$, $\beta = 0$, $max\_iter = 10^6$, and $multistarts = 32$.

A. SUCCESS RATE

Fig. 4 holds a comparison of the success rates between the different instantiation algorithms and implementations. It is clear that QFS-J completely outperforms QF-J, while for the smaller partitions, QF-R and CERES outperform QFS-J. For partitions containing more than eight qubits, QFS-J demonstrates the highest success rate, surpassing CERES, QF-J, and QF-R by factors of 41, 2.9, and 2.5, respectively. QFS-J employs more lenient criteria in its plateau detection mechanism. This, coupled with its reduced computational complexity, leads to better optimization performance compared to QF-J.

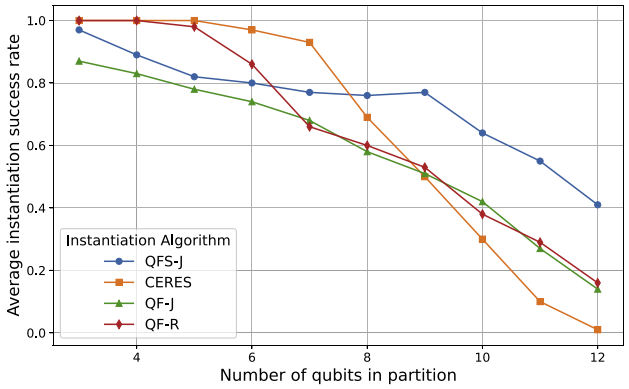

FIGURE 4. Instantiation success rate comparison. The GPU implementation of *QFactor-Sample* significantly surpasses that of *QFactor*. Specifically, for circuits larger than six qubits, the GPU version of *QFactor-Sample* surpasses the CPU version of *QFactor*. Furthermore, for circuits exceeding seven qubits, the GPU version of *QFactor-Sample* outperforms the CPU version of CERES.

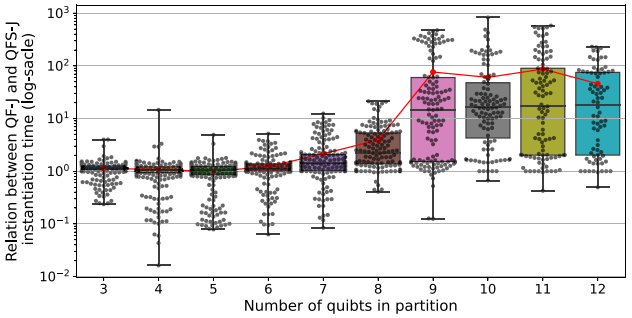

FIGURE 5. Swarm plot superimposed on a box plot of the relation between the instantiation time of *QFactor* and *QFactor-Sample* plotted for different circuits size, shown on a log scale. The whiskers extend to the maximum and minimum values, while the box represents the interquartile range, which contains the middle 50% of the data. A horizontal line inside the box represents the median. Each circle in the swarm plot represents a circuit, and it is plotted in a way that shows the distribution over the *y*-axis. The red markers represent the average runtime relation. From the plots, one can observe the significant improvement in the runtime of *QFactor-Sample* compared to *QFactor*.

B. EXECUTION SPEED

The relation between the runtime of QF-J and QFS-J can be seen in Fig. 5. See its caption for a detailed description of the swarm and box plots. We observe an overall average of 17.7× reduction in instantiation time and a 69× average reduction for partitions with more than eight qubits.

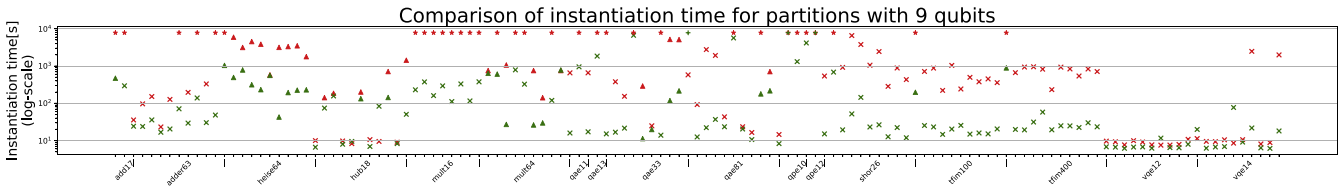

FIGURE 6. Instantiation runtime comparison between (red) *QFactor* and (green) *QFactor-Sample* for partitions with nine qubits. Each mark on the graphs represents the runtime of a single instantiation. "x" is a successful instantiation, "▲" represents a run that finished; however, the desired distance was not achieved, while "*" and "+" are time-outs. The markers are grouped according to the partition's origin circuit, where "|" marks the starting of a new circuit group. On average, *QFactor-Sample* is 76× faster compared to *QFactor* and is able to find a good solution before timing-out.

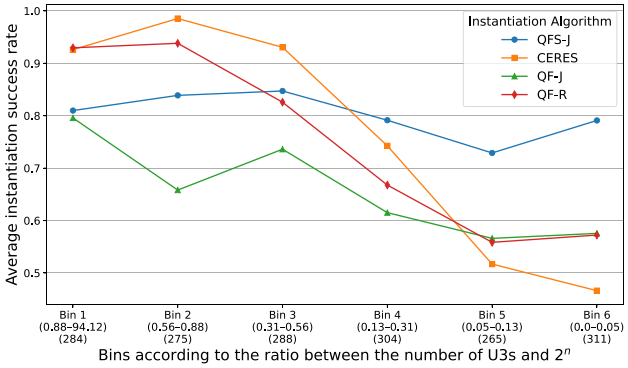

FIGURE 7. Average instantiation success rate plotted for different instantiation algorithms and binned according to the ratio between the number of U3s and 2^n, where n is the number of qubits in the circuit. The x-axis label indicates the bin index, its boundaries, and the number of circuits the bin holds. As expected, QFS-J has a better success rate compared to the other algorithms when the aforementioned ratio is small, as the number of U3s the circuit has is an indicator for the number of parameters, and the fewer parameters it has, the fewer training states QFS-J requires.

Fig. 6 illustrates a detailed comparative analysis of the instantiation performance between QFS-J and QF-J for partitions with nine qubits, showing both the runtime and the instantiation termination reason. We are providing the same type of graph for all of the partition sizes in Appendix B. Each point on the graphs represents an instantiation run, categorized by markers denoting success, failure to achieve the desired distance, and time-outs. Notably, markers are grouped based on the origin circuit of the partition, providing insights into performance trends across different circuit groups. On average, for partitions with nine qubits, QFS-J is 76× times faster compared to QF-J.

C. CIRCUIT SIZE

Fig. 7 shows the average success rate for the different instantiation algorithms binned according to the ratio $\frac{\#U3}{2^n}$, where n is the number of qubits and $\#U3$ is the number of U3 gates in the instantiated circuit. This type of binning was chosen because $\#U3$ is a relatively good proxy for the number of parameters the circuit has, and we know that *QFactor-Sample* will need fewer training states as the circuit has fewer parameters. To further differentiate how the circuit *QFactor-Sample* is superior compared to other instantiation algorithms, we divide $\#U3$ with 2^n, which reflects how compute-intensive the instantiation is going to be for *QFactor* and CERES. It is easy to see that the smaller the ratio is, the more *QFactor-Sample* outperforms the other algorithms.

TABLE 2. Runtime and QoR Comparison for Gate Deletion Flow

Circuit	Instantiator	Partition Size	#U3	#CNOT	Runtime [s]
adder63	QF-J	7	1,724	1,250	31,413
	CERES	7	–	–	time-out
	QFS-J	7	1,685	1,245	5,109
	QFS-J	8	1,653	1,218	7,373
mult16	QF-J	6	573	874	24,660
	CERES	6	569	874	21,463
	QFS-J	6	569	874	5,534
	QFS-J	9	488	836	20,839
shor26	QF-J	6	11,769	19,850	16,197
	CERES	6	–	–	time-out
	QFS-J	6	11,758	19,850	1,730
	QFS-J	7	11,380	19,327	8,954
qae11	QF-J	6	126	110	1,879
	CERES	6	122	110	2,783
	QFS-J	6	126	110	405
	QFS-J	8	114	110	6,386

Each row represents a resynthesis run on eight nodes with a 12-h time-out. Noteworthy findings include a substantial 4–9× synthesis runtime improvement when using *QFactor-Sample* over *QFactor*, and as expected, CERES is struggling on the big partitions and is more likely to time-out. Moreover, the utilization of larger partitions in synthesis leads to improved QoR.

D. USAGE IN SYNTHESIS

To conclude the evaluation, we incorporated *QFactor-Sample* in BQSKit's [6] re-synthesis gate deletion flow [9]. This workflow involves initially partitioning the provided circuit, followed by a unidirectional sweep aimed at deleting one gate at a time while reinstantiating the reduced partition to its original unitary form. This divide-and-conquer approach effectively converts compilation into a task that can be executed in parallel, enhancing efficiency. Our focus lies in examining the scalability of large circuit compilations and assessing *QFactor-Sample*'s effectiveness as a numerical optimizer, gauged by the runtime and the number of deleted gates within a circuit. We compared resynthesis runtime and QoRs when using *QFactor-Sample* against CERES and *QFactor*, utilizing the adder63, mult16, shor26, and qae11 circuits as benchmarks. This evaluation was conducted on eight hybrid nodes.

The results of our evaluation are presented in Table 2, showcasing the clear runtime superiority of *QFactor-Sample* over *QFactor* and CERES. Notably, for the adder63, shor26,

and qae11 circuits, we observe impressive speedups of 6×, 9×, and 4.5×, respectively. Furthermore, it is clear from all circuits that employing larger partitions during resynthesis leads to better QoR, albeit with the tradeoff of longer runtime.

V. DISCUSSION

Leveraging a bound on QML generalization error, *QFactor-Sample* significantly improves the runtime of quantum circuit instantiation when compared to best-in-class optimizers CERES and *QFactor*. The source of the improvement comes from an exponential reduction in the computational complexity of the algorithm compared to the other optimizers. To clarify the source of the reported speedups, we note that the dominant improvement arises from an algorithmic shift that reduces the complexity of the core optimization from matrix–matrix to vector–matrix multiplication, resulting in exponential runtime gains. While hardware acceleration (e.g., GPU use) and more flexible plateau detection policies contribute to improved wall-clock performance, their impact is additive rather than fundamental. All benchmarks were performed under comparable settings, and any differences in stopping criteria were applied thoughtfully to take advantage of the algorithm's inherent speed. Overall, the scalability and practical gains we report are rooted in algorithmic innovation.

The instantiation runtime improvements that we report here are somewhat skewed to the worse, as whenever an instantiation timed out we registered it as 10 or 720 min, depending on the circuit size. Hence, if we were able to run over the time limit, then the runtime gap between *QFactor* and *QFactor-Sample* would increase substantially.

The enhancement in *QFactor-Sample*'s instantiation success rate stems from its increased speed. This not only enables the algorithm to complete before the time-out but also permits the application of a more forgiving policy for plateau detection. Consequently, *QFactor-Sample* can uncover more optimal solutions on challenging optimization planes.

When evaluating *QFactor-Sample*'s impact on circuit resynthesis, we observe a direct decrease in overall runtime. Furthermore, leveraging a more scalable instantiation algorithm enables the utilization of larger partitions, leading to the discovery of additional optimization opportunities and resulting in a reduction in circuit size. During our benchmarks, we noticed that for some circuits, the partitioner we used was able to only cover part of the circuit with large partitions, impeding optimization opportunities. Appendix D provides the coverage statistics. Further investigation into the partitioning algorithm, although beyond this article's scope, is a promising area for future research.

Our benchmarking result analysis can serve as a guide for quantum compilers to choose the appropriate instantiation algorithm according to simple metrics of the instantiated circuit. For example, if there are fewer than five qubits, it is better to use *QFactor*-Rust or CERES, and for the larger qubit count, one should look at the ratio between the number of parameters the circuit has and compare it to 2^n to decide whether to use *QFactor-Sample* or *QFactor*. No single optimizer is perfect for all partitions, but with some hyperparameter tweaking and a decision regarding which instantiator to use on each partition, the overall performance of the compiler can be improved.

This work focuses on improving the runtime of quantum compilation while maintaining—or even improving—the quality of the resulting circuits. An important consequence of our approach is that it produces significantly shorter circuits with fewer entangling gates, such as CNOTs. Since such gates are a primary source of noise on current quantum hardware, our method is expected to yield circuits that execute with higher fidelity and better overall performance when run on real devices.

In this work, we assessed the performance of *QFactor-Sample* on qubit gates. However, it is worth noting that the same algorithm is applicable to qudit gates, where the speedup compared to *QFactor* would be even more significant, reducing the complexity from $O(d^{2n})$ to $O(Md^n)$.

An interesting follow-up research would be to perform unitary instantiation using one of the QML frameworks, such as TensorFlow Quantum [29] or torchquantum [30]. In this article, we have shown a reduction from instantiation to a traditional QML flow. Given the considerable engineering efforts invested in these frameworks, it becomes even more compelling to compare their runtime and QoR with *QFactor-Sample*.

Furthermore, we acknowledge that performing comprehensive sensitivity sweeps across all parameters on many circuits would provide additional insights. While full exhaustive sweeps are beyond the current compute/time resources, such sensitivity testing is a promising direction for future work.

APPENDIX A
QFactor-Sample IMPLEMENTATION DETAILS

In this appendix, we provide more details about *QFactor-Sample* implementation. We list all of *QFactor-Sample*'s hyperparameters and provide some best practices when using our implementation. It is important to note that the exponential runtime improvement is not the result of hyperparameter optimization, but rather stems from a fundamental shift in the underlying computation—from matrix–matrix multiplication to the more efficient vector–matrix multiplication.

We generated the random input test states by extracting columns of a random unitary matrix, which was generated using the `unitary_group` functionality in SciPy. Doubling the size of the training state does not involve randomizing the unitaries of the gates, as we have found that this does not impact the convergence speed. Instead, it only consumes time by generating numerous random unitaries for each gate.

During our testing of the implementation, we encountered GPU out of memory (OOM) exceptions. We did not find a

way to predict our GPU memory usage in JAX; hence, we implemented a recursive trial-and-error when running *QFactor-Sample*. If, while running *QFactor-Sample*, we catch a GPU OOM exception, we would iteratively run two runs of *QFactor-Sample* but with half the number of multistarts. We also want to mention that JAX has an environment variable, XLA_PYTHON_CLIENT_ALLOCATOR, that forces the runtime to always release memory buffers to the GPU when they are no longer needed, avoiding OOM due to fragmentation with the potential runtime cost of reallocating memory.

Following is a list of *QFactor-Sample*'s hyperparameters. They control the termination conditions of the algorithm, generalization error threshold, gate update policy, and randomization.

1) *dist_tol:* When the average distance between the training output states and those generated by the circuit (9) is lower than *dist_tol*, the algorithm stops.
2) *plateau_windows_size* and *diff_tol_r:* Control the plateau-detection mechanism. The algorithm will terminate due to plateauing when, on consecutive *plateau_windows_size* iterations, the cost function did not relatively improve by *diff_tol_r*. The algorithm checks if the relative improvement has been met by calculating the following relation:
$$|c_{i-1}| - |c_i| > diff\_tol\_r * |c_i| \tag{13}$$
where c_i is the cost function value after iteration i.
3) *number_of_training_states* and *overtrain_ratio:* These set the initial number of training states. If during the instantiation the algorithm detects that the normalized generalization error is bigger than *overtrain_ratio*, it stops, doubles the number of training states, and restarts. The normalized generalization error is computed by
$$\frac{c_{\text{val}}}{c_{\text{train}}} - 1 \tag{14}$$
where c_{val} and c_{train} are the validation and training costs, respectively. The *double-and-restart* stops when the number of training states reaches 2^n, where n is the number of qubits.
4) *min_iter:* It sets the minimum number of iterations for *QFactor-Sample* to complete before stopping due to plateauing or overtraining.
5) *max_iter:* It sets the maximum number of iterations for *QFactor-Sample*. The algorithm will always stop when it reaches this limit.
6) *multistarts* and *seed:* To overcome the local minimum problem, one can run *QFactor-Sample* with various initial gate unitaries, in the hope that at least one of the runs will converge to a good solution. The initial random unitaries for each gate are controlled by a *seed* parameter.
7) *Beta:* It serves as a regularization parameter that governs the retention of the previous value during the gate update step. Instead of conducting the SVD operation solely on the environment $\mathcal{E}$, it is performed on the linear combination
$$(1-\beta)\mathcal{E} + \beta * u^{\dagger}. \tag{15}$$
This parameter proves beneficial in addressing slow convergence issues encountered in circuits with intergate dependencies, where local optimization methods may fall short. When $\beta = 0$, the update becomes *full* as described previously, whereas $\beta = 1$ results in no update taking place.

From our experience, the following are some good initial hyperparameter values: $dist\_tol = 10^{-8}$, $plateau\_windows\_size = 5$, $diff\_tol\_r = 10^{-5}$, $number\_of\_training\_states = 2$, $overtrain\_ratio = 0.1$, $min\_iter = 6$, $max\_iter = 10^4$, $multistarts = 16$, and $\beta = 0$. By modifying the aforementioned hyperparameters, one can easily adjust the tradeoff between result quality and execution time. If one wishes to get results faster, decrease *max_iter*, and increase *diff_tol_r* and *diff_tol_a*. Alternately, one might increase β or *multistarts* to find better results with additional execution overhead.

APPENDIX B
INSTANTIATION PERFORMANCE COMPARISON BETWEEN *QFactor-Sample* AND *QFactor* FOR CIRCUITS WITH 3–12 QUBITS

In this appendix, we provide the complete instantiation performance comparison for circuits with 3–12 qubits using *QFactor-Sample* and *QFactor*. The comparison for circuits with 3–12 qubits is shown in Fig. 9, which extends the data presented in Fig. 6.

A notable observation from the analysis is the widening performance gap between *QFactor-Sample* and *QFactor* as the circuit size increases with more qubits. This is expected due to the differences in computational complexity. As the partition size grows, *QFactor-Sample* experiences significantly fewer time-outs compared to *QFactor*, enabling it to either reach a good solution or stop upon detecting a plateau. In the graph, we sorted the instantiations based on the origin circuit of each partition and used different markers to indicate the termination type. This visualization helps illustrate why *QFactor-Sample* has a higher success rate compared to *QFactor* for partitions with three qubits. The key difference lies in the plateau detection mechanism: *QFactor-Sample* employs a more lenient approach, enabling it to find good solutions for all of the Heisenberg partitions (as shown on the right-hand side of the graph), whereas *QFactor* gives up on those.

APPENDIX C
TRAINING STATE DISTRIBUTION IMPACT

As shown in [16], the generalization bounds are valid for arbitrary data-generating distribution. Some distributions will

IEEE Transactions on Quantum Engineering

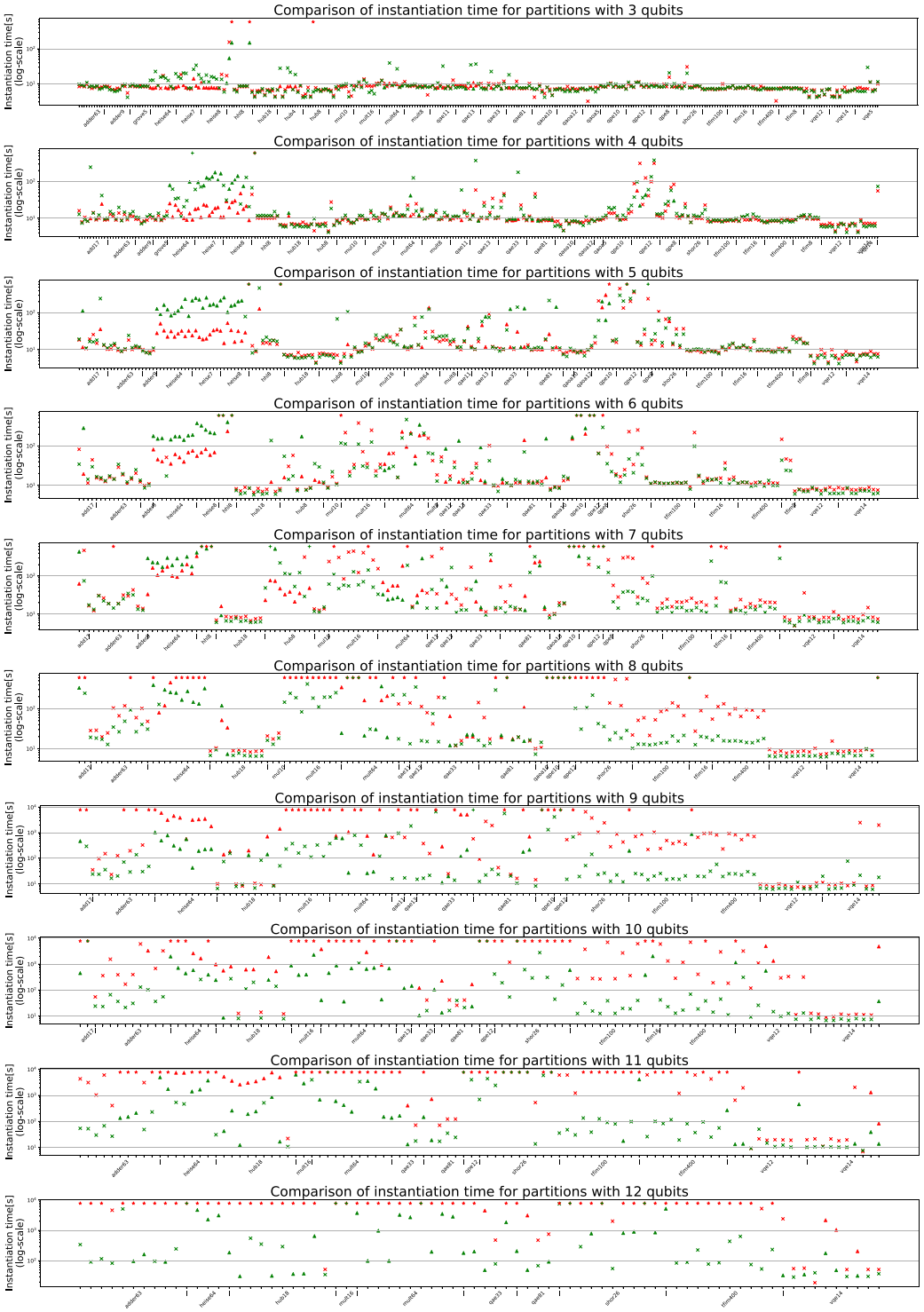

FIGURE 8. **Instantiation runtime comparison between (red) *QFactor* and (green) *QFactor-Sample*. Each graph represents a different partition size varying from 3 to 12. Each mark on the graphs represents the runtime of a single instantiation. "x" is a successful instantiation, "▲" represents a run that finished; however, the desired distance was not achieved, while "*" and "+" are time-outs. The markers are grouped according to the partition's origin circuit, where "|" marks the starting of a new circuit group. We can observe that as the circuit has more qubits, the performance gap between *QFactor-Sample* and *QFactor* increases up to 100×, ultimately allowing *QFactor-Sample* to instantiate bigger circuits.**

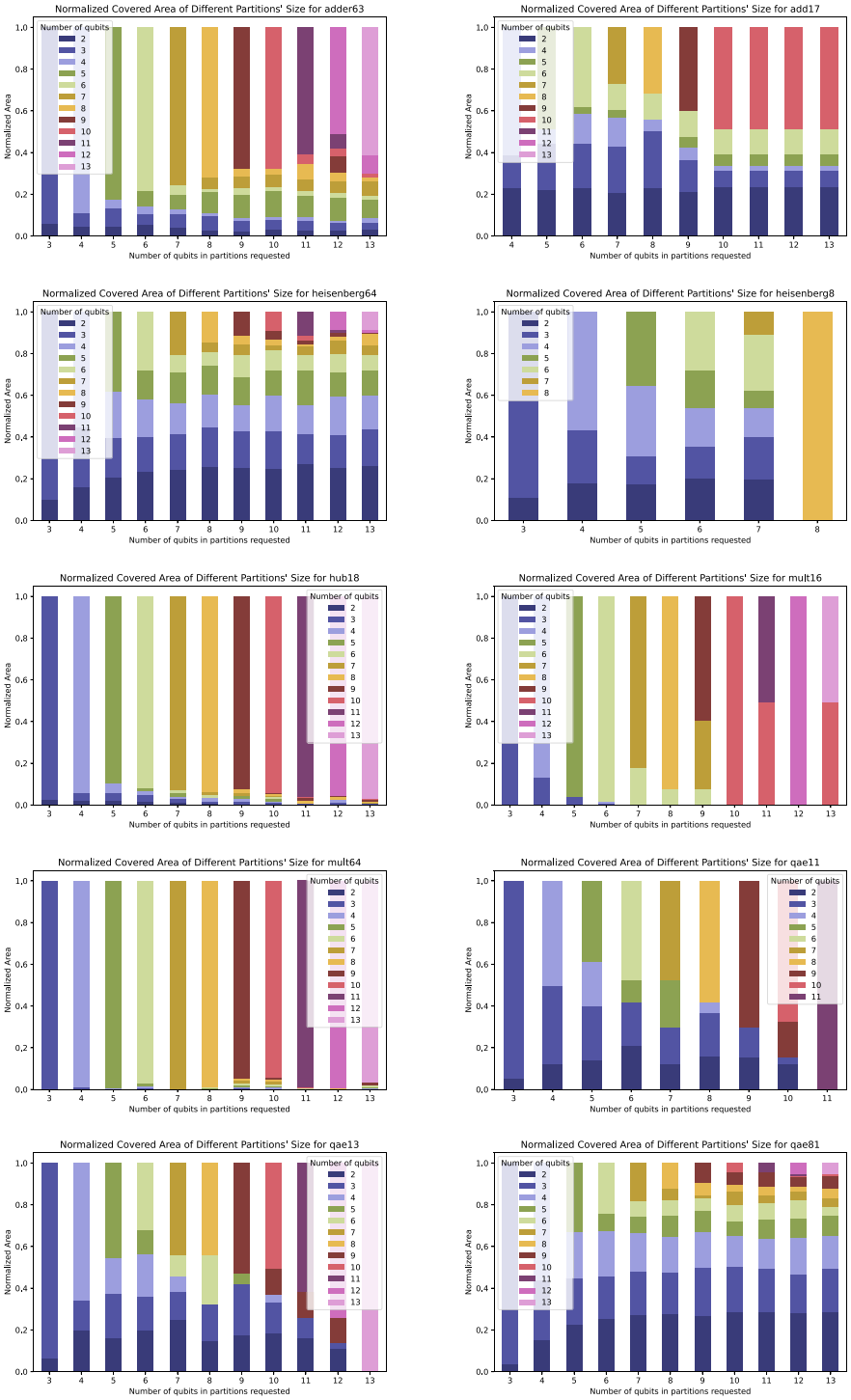

FIGURE 9. **BQSKit's "QuickPartitioner" cover statistics for different circuits (See Table 1) and partition sizes requested.**

saturate that bound but some will perform better. As seen in practice, training based on computational basis states requires much more data than training on Haar random states. It is important to note that training in both cases satisfies the bound in (6). All choices will lead to an exponential reduction in the time required to compute the cost function. This discussion is beyond the scope of this article though. It is because we divide the circuit into parts that can be classically simulated with a state vector simulator. Therefore, we can always afford to choose our data to be generated from Haar random distribution, which performed best in our tests. We have numerically verified that this choice leads to the smallest amount of data needed for good generalization. One might wish to use some quasi-random distribution, and in the worst case, it will double the number of training states needed.

IEEE Transactions on Quantum Engineering

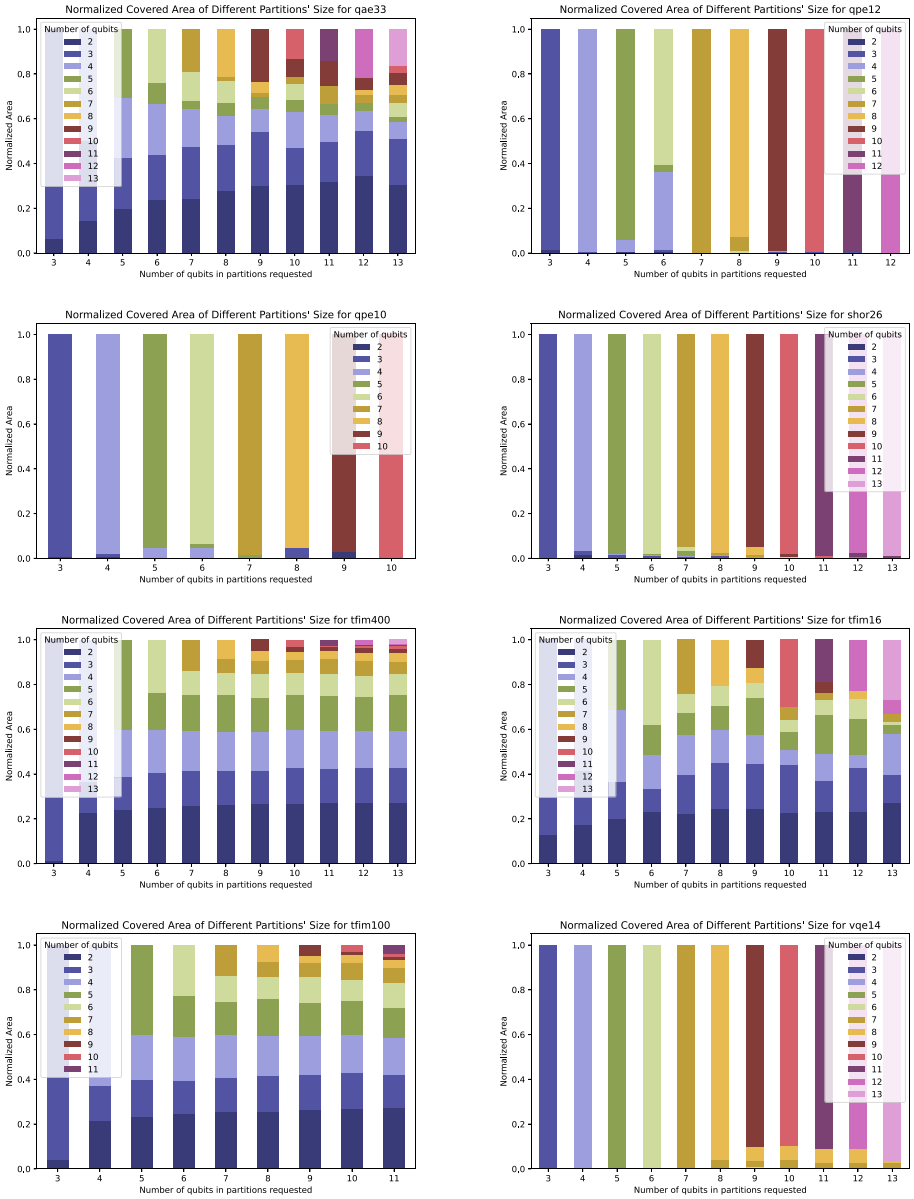

FIGURE 9. ***(Continued.)***

We do not have a formal proof that the Haar distribution is optimal. However, we have numerical evidence that this is indeed the case as well as some intuitive arguments that we describe later. Haar random distribution may not be universally optimal for every circuit, but at least it avoids some problems other distributions face.

Let us say that our task is to compress a circuit U that contains $R_z(\alpha)$ rotation as the first gate acting on the first qubit. Let us also assume that we are using computational basis states as our training data input states. The action of U on any state of the form $|0b_1b_2,\ldots,b_n\rangle$ is insensitive to angle α. That is, unless our training dataset contains at least one state of the form $|0b_1b_2,\ldots,b_n\rangle$, it will be impossible to infer the full action of U with these training data. That problem extends to other gates that act trivially on some computational basis states. Since we draw our training states randomly, this distribution choice will (on average) lead to an increased size of the training dataset needed to achieve generalization. Note that this problem is not present if we choose to work with Haar random input states. Up to a set of measure zero, each Haar random state is sensitive to angle α and will meaningfully contribute to probing the unitary U with training states. One expects that the Haar random input state is likely to be sensitive to all parameters of the circuit U and will lead to a small required size of the training dataset. Random input states do not introduce bias that may be present in other distributions, as described earlier.

APPENDIX D
PARTITIONER COVERAGE STATISTICS

During the evaluation of *QFactor-Sample* as part of a resynthesis flow, we observed that for some circuits, BQSKit's "QuickPartitioner" allocates a significant portion of the circuit area to partitions with a small number of qubits, thus impeding optimization opportunities. In this appendix, we present BQSKit's "QuickPartitioner" performance on different circuits. Fig. 9 summarizes the normalizedcovered area of partitions of different sizes for several representative circuits. Each subplot corresponds toa different circuit, and within each subplot, the stacked bars represent the fraction of the circuit area coveredby partitions of varying sizes as the requested partition size increases.

REFERENCES

[1] J. R. McClean, J. Romero, R. Babbush, and A. Aspuru-Guzik, "The theory of variational hybrid quantum-classical algorithms," *New J. Phys.*, vol. 18, no. 2, Feb. 2016, Art. no. 023023, doi: 10.1088/1367-2630/18/2/023023.

[2] E. Farhi, J. Goldstone, and S. Gutmann, "A quantum approximate optimization algorithm," Nov. 2014, *arXiv:1411.4028*, doi: 10.48550/arXiv.1411.4028.

[3] N. A. Nemkov, E. O. Kiktenko, I. A. Luchnikov, and A. K. Fedorov, "Efficient variational synthesis of quantum circuits with coherent multi-start optimization," *Quantum*, vol. 7, May 2023, Art. no. 993, doi: 10.22331/q-2023-05-04-993.

[4] P. Rakyta and Z. Zimborás, "Approaching the theoretical limit in quantum gate decomposition," 2021, *arXiv:2109.06770*, doi: 10.48550/arXiv.2109.06770.

[5] L. Cincio, K. Rudinger, M. Sarovar, and P. J. Coles, "Machine learning of noise-resilient quantum circuits," *PRX Quantum*, vol. 2, Feb. 2021, Art. no. 010324, doi: 10.1103/PRXQuantum.2.010324.

[6] E. Younis, C. C. Iancu, W. Lavrijsen, M. Davis, and E. Smith, "Berkeley quantum synthesis toolkit (BQSKit) v1," Computer Software, Apr. 2021, doi: 10.11578/dc.20210603.2.

[7] E. Younis, K. Sen, K. Yelick, and C. Iancu, "QFAST: Conflating search and numerical optimization for scalable quantum circuit synthesis," in *Proc. IEEE Int. Conf. Quantum Comput. Eng.*, 2021, pp. 232–243, doi: 10.1109/QCE52317.2021.00041.

[8] M. G. Davis, E. Smith, A. Tudor, K. Sen, I. Siddiqi, and C. Iancu, "Towards optimal topology aware quantum circuit synthesis," in *Proc. IEEE Int. Conf. Quantum Comput. Eng.*, 2020, pp. 223–234, doi: 10.1109/QCE49297.2020.00036.

[9] E. Younis and C. Iancu, "Quantum circuit optimization and transpilation via parameterized circuit instantiation," in *Proc. IEEE Int. Conf. Quantum Comput. Eng.*, 2022, pp. 465–475, doi: 10.1109/QCE53715.2022.00068.

[10] M. Cerezo et al., "Variational quantum algorithms," *Nat. Rev. Phys.*, vol. 3, no. 9, pp. 625–644, Sep. 2021, doi: 10.1038/s42254-021-00348-9.

[11] M. Benedetti, E. Lloyd, S. Sack, and M. Fiorentini, "Parameterized quantum circuits as machine learning models," *Quantum Sci. Technol.*, vol. 4, no. 4, Nov. 2019, Art. no. 043001, doi: 10.1088/2058-9565/ab4eb5.

[12] J. Biamonte, P. Wittek, N. Pancotti, P. Rebentrost, N. Wiebe, and S. Lloyd, "Quantum machine learning," *Nature*, vol. 549, no. 7671, pp. 195–202, Sep. 2017, doi: 10.1038/nature23474.

[13] K. Beer et al., "Training deep quantum neural networks," *Nat. Commun.*, vol. 11, no. 1, Feb. 2020, Art. no. 808, doi: 10.1038/s41467-020-14454-2.

[14] A. Kukliansky, E. Younis, L. Cincio, and C. Iancu, "QFactor: A domain-specific optimizer for quantum circuit instantiation," in *Proc IEEE Int. Conf. Quantum Comput. Eng.*, Sep. 2023, pp. 814–824, doi: 10.1109/QCE57702.2023.00096.

[15] A. Abbas, D. Sutter, C. Zoufal, A. Lucchi, A. Figalli, and S. Woerner, "The power of quantum neural networks," *Nat. Comput. Sci.*, vol. 1, no. 6, pp. 403–409, Jun. 2021, doi: 10.1038/s43588-021-00084-1.

[16] M. C. Caro et al., "Generalization in quantum machine learning from few training data," *Nat. Commun.*, vol. 13, no. 1, 2022, Art. no. 4919, doi: 10.1038/s41467-022-32550-3.

[17] L. Banchi, J. Pereira, and S. Pirandola, "Generalization in quantum machine learning: A quantum information standpoint," *PRX Quantum*, vol. 2, no. 4, Nov. 2021, Art. no. 040321, doi: 10.1103/PRXQuantum.2.040321.

[18] Y. Du, Z. Tu, X. Yuan, and D. Tao, "Efficient measure for the expressivity of variational quantum algorithms," *Phys. Rev. Lett.*, vol. 128, no. 8, Feb. 2022, Art. no. 080506, doi: 10.1103/PhysRevLett.128.080506.

[19] R. Orús, "A practical introduction to tensor networks: Matrix product states and projected entangled pair states," *Ann. Phys.*, vol. 349, pp. 117–158, 2014, doi: 10.1016/j.aop.2014.06.013.

[20] J. Nocedal, "Updating quasi-Newton matrices with limited storage," *Math. Comput.*, vol. 35, no. 151, pp. 773–782, 1980, doi: 10.1090/S0025-5718-1980-0572855-7.

[21] D. C. Liu and J. Nocedal, "On the limited memory BFGS method for large scale optimization," *Math. Program.*, vol. 45, nos. 1–3, pp. 503–528, 1989, doi: 10.1007/BF01589116.

[22] A. Ranganathan, "The Levenberg-Marquardt algorithm," 2004. [Online]. Available: https://sites.cs.ucsb.edu/yfwang/courses/cs290i_mvg/pdf/LMA.pdf

[23] A. Géron, *Hands-On Machine Learning With Scikit-Learn, Keras, and Tensor-Flow*, 2nd ed. Sebastopol, CA, USA: O'Reilly Media, 2019. [Online]. Available: https://www.oreilly.com/library/view/hands-on-machine-learning/9781492032632/

[24] F. Mezzadri, "How to generate random matrices from the classical compact groups," *Notices Amer. Math. Soc.*, vol. 54, no. 5, pp. 592–604, 2007. [Online]. Available: https://www.ams.org/journals/notices/200705/fea-mezzadri-web.pdf

[25] J. Bradbury et al., "JAX: Composable transformations of Python NumPy programs." Accessed: Nov. 6, 2025. [Online]. Available: http://github.com/google/jax

[26] A. Kukliansky, "GPU implementation of QFactor circuit instantiation using JAX." Accessed: Nov. 6, 2025. [Online]. Available: https://github.com/BQSKit/bqskit-qfactor-jax/

[27] S. Agarwal, K. Mierle, and T. C. S. Team, "Ceres Solver," 2022. [Online]. Available: https://github.com/ceres-solver/ceres-solver

[28] Perlmutter Architecture, 2023. [Online]. Available: https://docs.nersc.gov/systems/perlmutter/architecture/

[29] M. Broughton et al., "Tensorflow quantum: A software framework for quantum machine learning," 2020, *arXiv:2003.02989*, doi: 10.48550/arXiv.2003.02989.

[30] H. Wang et al., "QuantumNAS: Noise-adaptive search for robust quantum circuits," in *Proc. 28th IEEE Int. Symp. High-Perform. Comput. Archit.*, 2022, pp. 692–708, doi: 10.1109/HPCA53966.2022.00057.