Leveraging Machine Learning for Quantum Circuit Optimization

Mathias Weiden
John D. Kubiatowicz, Ed.

Electrical Engineering and Computer Sciences
University of California, Berkeley

Technical Report No. UCB/EECS-2023-84
http://www2.eecs.berkeley.edu/Pubs/TechRpts/2023/EECS-2023-84.html

May 10, 2023

Leveraging Machine Learning for Quantum Circuit Optimization

Mathias Weiden

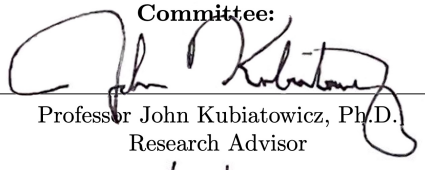
Spring 2023

Research Project

Submitted to the Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, in partial satisfaction of the requirements for the degree of **Master of Science, Plan II**.

Approval for the Report and Comprehensive Examination:

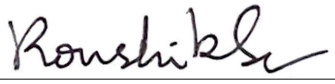
Committee:



Professor John Kubiatowicz, Ph.D.
Research Advisor

5/9/2023

Date



Professor Koushik Sen, Ph.D.
Second Reader

5/10/2023

Date

Abstract

In the era of noisy, non-fault tolerant, quantum hardware, reducing the number of operations in quantum programs is essential to ensure computations produce meaningful results. Unitary synthesis is a particularly promising technique which uses non-linear numerical optimization tools paired with heuristic-guided tree search algorithms to produce resource efficient implementations of quantum algorithms. This report illustrates how unitary synthesis can be used to optimize quantum circuits with many qubits. The TopAS algorithm demonstrates how minimizing both operation count and interaction complexity before mapping can produce better circuit implementations than the state of the art. This report also presents several discoveries that enable the use of machine learning in accelerating unitary synthesis. Namely, a canonical representation of unitary matrices, and an argument for the learnability of mappings between a specific class of unitary matrices and parameterized quantum circuits are presented. The QSeed algorithm shows how machine learning can be leveraged to accelerate the optimization of wide quantum circuits without sacrificing solution quality.

I sincerely thank all my family, friends, and teachers.
Your compassion and support have been essential in all my pursuits.

Leveraging Machine Learning for Quantum Circuit Optimization

Mathias Weiden

Spring 2023

1 Introduction

The idea to employ the seemingly unnatural powers of quantum mechanics for computation originated in Richard Feynman's 1981 lecture titled "Simulating Physics with Computers" [11]. Sixteen years later, in 1997, Simon's problem demonstrated the first oracle separation between quantum computers and probabilistic classical computers [25]. In that same year, Peter Shor showed that quantum computers can achieve super-polynomial speedups over classical computers for the prime number factoring problem [23]. Yet, the execution of these algorithms remains out of reach for modern quantum machines. The current Noisy Intermediate Scale Quantum (NISQ) era is defined by limited hardware that is extremely susceptible to environmental noise and decoherence [21]. Quantum error correction promises to elevate the field into an era of fault tolerance, but questions relating to the scaling of hardware capable of carrying out these complex error correction schemes remain unanswered [1]. Error mitigation, an ever widening suite of techniques used to reduce the effects of environment interference, serve as a NISQ era aid to these problems. Google's quantum supremacy experiments have confirmed the accuracy of a simple digital noise model, meaning just reducing the number of operations in quantum programs can dramatically improve the likelihood that computations are correct [3].

In this report, I will discuss common techniques used to optimize the operation count and depth of quantum circuits. I will focus on the use of unitary synthesis, a promising optimization technique. I will also present research efforts I have made in applying and improving unitary synthesis algorithms with machine learning.

This report is divided as follows: Section 2 surveys common techniques used for quantum circuit optimization and introduces unitary synthesis. Section 3 details the efforts I have made in applying machine learning to circuit optimization. Finally, Section 4 discusses promising future directions for the field of machine learning in circuit optimization.

2 Quantum Circuit Optimization

In this section, I will introduce preliminary concepts in quantum computing and the circuit optimization problem. I will also detail several widely used optimization techniques, and algorithms used for a specific style of optimization called unitary synthesis.

2.1 Preliminaries

Definition 2.1 (Qubits)**.** A qubit is the base unit of quantum information. The qubit model assumes that a quantum system can be in any arbitrary superposition of two orthogonal states $|0\rangle$ and $|1\rangle$, so long as it satisfies

$$|\psi\rangle = \alpha|0\rangle + \beta|1\rangle \quad |\alpha|^2 + |\beta|^2 = 1$$

where $\alpha, \beta \in \mathbb{C}$. Given a system of n qubits and defining $N = 2^n$, the quantum state lies in a Hilbert space of size $\mathbb{C}^{N\times N}$. The composite state $|\phi\rangle$ of this n qubit system can be found by taking the n-fold tensor product

$$|\phi\rangle = \bigotimes_{i=1}^{n} |\psi_i\rangle.$$

Definition 2.2 (Quantum Gates)**.** Operations on qubits must preserve the unit L2 norm of the quantum state. For this reason, operations on qubits must be unitary. Gates may act on one or multiple qubits, and they can be functions of real parameters. Two noteworthy quantum gates are the single qubit U3 gate

$$U3(a,b,c) = \begin{bmatrix} \cos(\frac{a}{2}) & -e^{ic}\sin(\frac{a}{2}) \\ e^{ib}\sin(\frac{a}{2}) & e^{i(b+c)}\cos(\frac{a}{2}) \end{bmatrix}$$

and the two qubit CNOT or CX gate

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

The gate set of $\{CNOT, U3\}$ is both universal for quantum computation and capable of creating any unitary matrix [19].

Quantum gates can be composed in parallel or sequentially. Parallel composition corresponds to a tensor product while sequential composition corresponds to a matrix multiplication.

Definition 2.3 (Quantum Circuits)**.** A quantum circuit is a data structure that describes a quantum program. Qubits are represented as horizontal wires. Quantum gates are composed in sequences where time is assumed to flow from left to right. The single-qubit U3 gate appears as a squares, the multi-qubit CNOT gate spans the qubits on which it is applied. The width of a circuit

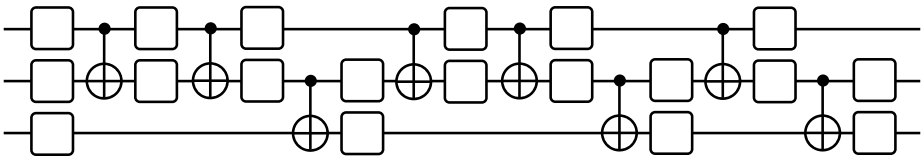
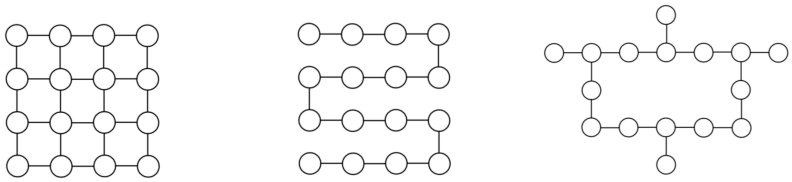


Figure 1: An example of a quantum circuit with U3 and CNOT gates.



(a) Mesh (b) Linear (c) Falcon

Figure 2: Examples of three common qubit topologies. The heavy hexagonal "falcon" style of topology is used in IBM's superconducting machines. Meshes and lines are also common topologies.

refers to the number of qubits in the circuit, the depth refers to the number of time steps along the circuit's critical path. Circuits that contain one or more parameterized gates (such as the U3 gate) are called Parameterized Quantum Circuits (PQCs). The unitary operation implemented by quantum circuit C is U_C. If C is a PQC parameterized by a vector $\theta \in \mathbb{R}^m$, then the unitary associated with C is denoted $U_C(\theta)$.

Although quantum circuits allow interactions between any pair of qubits, hardware is typically more limited. To account for this, a *qubit topology* can be used to describe more restrictive connectivities.

Definition 2.4 (Qubit Topology)**.** A qubit topology is a graph $G = (V, E)$ where V is a set of vertices and E is a set of edges. Vertices $v \in V$ represent qubits, while edges $(u, v) \in E$ represent (hardware) allowed interactions between qubits.

The goal of quantum circuit optimization is to reduce the operation count and depth of quantum programs. This is necessary as NISQ era qubits are extremely sensitive to noise and are not error corrected. Certain quantum gates are more likely to introduce error than others. Multi-qubit gates, such as the CNOT, take far more time to execute and are far more error prone than single-qubit gates. Depending on the specific machine, some single-qubit gates may be more error prone and more difficult to calibrate than others [7]. Critically, the optimized version of a circuit must implement the same functionality as the original circuit.

2.2 Rule Based Circuit Optimization

Rule based circuit optimizations are a class of optimization techniques that make local gate transformations based on a pre-compiled series of transformation rules. Often, these sorts of algorithms are implemented with pattern recognition and replacement schemes [6]. Peephole optimization techniques (which are heavily used in classical compilation), where a window scans across the quantum circuit seeking opportunities for pattern replacement, are commonly used in practice [26, 2, 13].

This sort of optimization is fast, and is often very effective at reducing both circuit depth and gate counts, but, it has two inherent limitations. First, patterns for replacement must be identified beforehand, a process which is typically carried out by human observation. The space of possible circuit pattern-replacement pairs is vast, and the greatest opportunities for optimization have almost certainly not been fully explored. These replacement rules are human constructed and guided by heuristic and intuition. This restriction of peephole optimization has also been pointed out in classical compilers [4]. In response, tools such has CompilerGym [8] have been developed to use Reinforcement Learning (RL) to automate compiler optimization flows in classical compilers. The use of RL in peephole optimization for quantum computers has also been proposed.

Second, many of these rule based optimization techniques only consider very small width windows for optimization. Although fast, much opportunity for optimization is passed by because a more global perspective is not taken. Ultimately, the greatest issue with this particular aspect of rule based optimization is that typically small tweaks to imperfectly programmed algorithms are made. It is often the case that human written algorithm implementations are written in a way that is more understandable to humans, but this often results in non-optimal code. In these cases, it may instead be advantageous to rewrite instructions completely rather than make small changes to the existing code.

Both of these concerns are addressed in unitary synthesis optimization.

2.3 Unitary Synthesis

Unitary synthesis is a bottom-up method of implementing unitaries as quantum circuits. More formally,

Definition 2.5 (Unitary Synthesis)**.** Given a unitary matrix $U \in \mathbb{C}^{N\times N}$, a unitary synthesis algorithm constructs a quantum circuit C that implements a unitary U_C satisfying

$$\|U - U_C\|_{HS} \leq \epsilon \tag{1}$$

where $\|\cdot\|_{HS}$ is the Hilbert-Schmidt norm defined as

$$\|A\|_{HS} = Tr(A^\dagger A). \tag{2}$$

Unitary synthesis describes the process of implementing a unitary as a quantum circuit in a way that allows for approximation. In the NISQ era, this is

a particularly interesting feature. Increasing the amount of approximation (or increasing the constant ϵ) typically reduces the number of gates in a circuit. So one can find a balance between error introduced by approximation, and error "saved" by gate reduction.

The Hilbert-Schmidt norm is one of the many metrics that can be used for this task, the most natural of which is the diamond norm [5]. However, the diamond norm is typically not used because of its extreme computational demands.

Synthesis can be performed with both parameterized and non-parameterized gate sets. This fact is known due to the Solovay-Kitaev theorem, which proves that discrete non-parameterized gate sets are capable of implementing any unitary to within some approximation factor [18]. For practicality, we typically assume a parameterized gate set such as $\{CNOT, U3\}$. Circuits expressed in this gate set are typically far less deep. Given a PQC, the process of solving for parameters to approximate a unitary is called *instantiation*. Expressed concretely,

Definition 2.6 (Instantiation)**.** Given a target unitary U and a unitary $U(\theta)$ parameterized by $\theta \in \mathbb{R}^m$, instantiation solves the problem

$$\arg\min_{\theta} \|U - U(\theta)\|_{HS}. \tag{3}$$

The instantiation problem becomes exponentially more difficult as qubits are added to the circuit. This is because the size of the unitaries grow with $O(2^n)$. The parameterized unitary $U(\theta)$ is typically implemented with a PQC. Adding more gates to this circuit increases the number of parameters θ, which also adds difficulty.

2.3.1 QSearch

QSearch was among the first unitary synthesis algorithms [10]. QSearch approaches unitary synthesis by decomposing the problem into two operations: synthesis tree search and instantiation.

As input, QSearch accepts a qubit topology $G = (V, E)$ and a target unitary U. The edges $(u, v) \in E$ describe allowed operations in the synthesized circuit. Synthesis begins with a base circuit containing only U3 gates applied to each qubit. The circuit is instantiated to determine if it is a suitable approximation of the target unitary. At this point, the synthesis tree search process begins. A unit of gates is appended to the base circuit, one along each edge in the qubit topology. Circuits derived this way can be arranged in a tree, called the *synthesis circuit tree*. A node in the tree acts as a circuit prefix to its children; its children will only differ by gates appended to the end of the circuit. The process of synthesis can then be considered a search through the synthesis circuit tree. An example of synthesis circuit tree is depicted in Figure 3.

QSearch uses an A^* heuristic that considers the Hilbert-Schmidt distance to the target unitary as well as the current circuit's gate count to guide the

synthesis tree search. Given a node in the tree, QSearch calls instantiation on each of the node's children, and ranks them according to the heuristic.

Synthesis begins at the root node, then traverses through the synthesis circuit tree until a target circuit is reached. The number of instantiation calls made is at least as large as the number of circuits visited in this traversal. Other than wall-clock time, one can also measure the time needed to synthesize a circuit by measuring this path distance in the synthesis tree search.

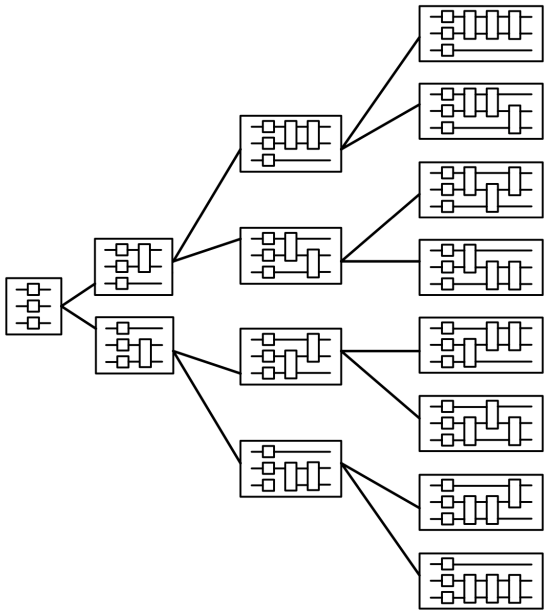


Figure 3: The synthesis circuit tree for a three qubit circuit with linear connectivity.

The details of QSearch's CNOT gate count reduction compared to other rule-based methods can be found in [10]. QSearch's bottom-up approach and allowed tunable approximation threshold result in unmatched gate reductions. This degree of optimization is possible because QSearch is capable of taking a broader view of the unitary being implemented. Redundancies and inefficiencies present in human written circuits are eschewed, and a globally efficient path to the target unitary is found. This can lead to rather non-intuitive results: Transverse Field Ising Model (TFIM) circuits, which simulate a material's magnetization, can be compiled into constant depth circuits. These circuits iteratively append gates that compute the magnetization for new time steps in the simulation, but synthesis reliably finds a circuit with the same structure that summarizes the entire simulation.

Because it only applies gates along allowed edges in some qubit topology, circuits synthesized by QSearch can also be mapped to restrictive hardware.

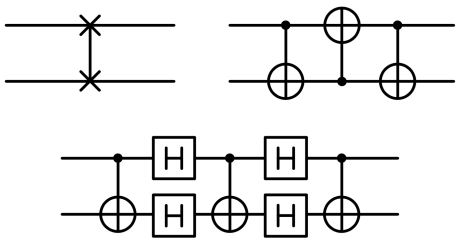


Figure 4: Three equivalent implementations of a quantum SWAP gate, which can be decomposed into three CNOT gates.

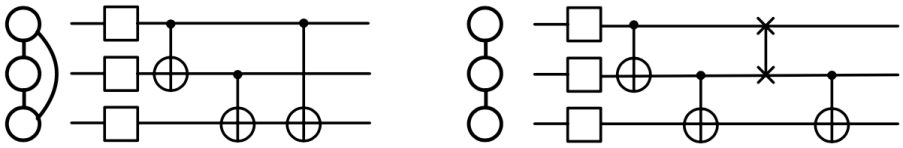


Figure 5: A quantum circuit that has been mapped to a restrictive qubit topology

Definition 2.7 (Mapped Quantum Circuits)**.** A quantum circuit C is said to be mapped to a qubit topology $G = (V, E)$ if multi-qubit gates in C only occur along edges in E. A quantum circuit can be mapped to a qubit topology by the use of SWAP operations, which exchange the quantum state held by two qubits. The process of injecting SWAP gates into a circuit is called *routing*.

2.3.2 Synthesis Time Scaling

The run time of unitary synthesis algorithms such as QSearch grows exponentially as the number of qubits increases. Both stages, instantiation and synthesis tree search, contribute to this scaling. The time complexity of instantiation grows as $O(2^n)$ because the width of the target and parameterized unitaries grows exponentially. The run time of the tree search component also scales exponentially, as the minimum number of CNOT gates needed to implement an arbitrary unitary is $\frac{1}{4}\lceil 4^n - 3n - 1 \rceil$ [22]. Both components of QSearch's approach to synthesis result in synthesis times quickly becoming untenable.

2.3.3 Circuit Partitioning

In order to handle the run time scaling issues of bottom-up synthesis, wide quantum circuits must be broken into smaller subcircuits. This process is referred to as *circuit partitioning*.

Definition 2.8 (Circuit Partitioning)**.** A partitioned quantum circuit is a quantum circuit C along with a sequence of sets S. Each set $B \in S$ is a set of gates

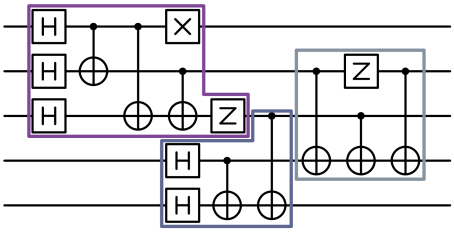


Figure 6: A five qubit quantum circuit that has been partitioned into three qubit subcircuits.

which appear in C. B is called a block, partition, or subcircuit. Blocks satisfy the properties

$$\forall B \in S \quad \bigcap B = 0 \quad \text{and} \quad \bigcup B = C \tag{4}$$

meaning each gate is a member of only one block, and all gates are a member of a block. To be a valid circuit partitioning, gates in block B must precede or succeed other gates in B. Valid partitionings only contain gates that act on interacting qubits. This means that the subgraph induced by some set of gates in B is connected. Typically, the width of partitions are limited to a value $k \in \mathbb{Z}$ called the *partition or block width*.

There are several strategies for partitioning quantum circuits. Their run times vary from growing with $\binom{n}{k}$, to linear in the number of gates. Should partitioning algorithms form blocks containing a similar number of gates? Should blocks greedily be formed so there are a few blocks with many gates? It remains unclear what the goal of a partitioning algorithm should be. The criteria for circuit partitioning is likely dependent on what optimization will be run, but this requires further investigation. Typically, it is the case that synthesis performs better when there are more CNOT gates in the circuit being synthesized. However, the time required to partition circuits is in practice far more important than this, so simple partitioning strategies are preferred.

To apply synthesis optimization to circuits wider than four (possibly five) qubits, circuit partitioning must be used. In a quantum compilation flow, the end goal is typically a series of instructions that can be run on real hardware. Optimized circuits must therefore be mapped to hardware. But, should optimization, such as synthesis, be applied before or after mapping?

2.3.4 QGo

QGo was the first wide quantum circuit synthesis algorithm [28]. QGo optimizes quantum circuits that have already been mapped to restrictive qubit topologies. This style of optimization is called post-mapping optimization, and the programmatic flow of QGo is illustrated in Figure 8. The guiding philosophy behind QGo is: use synthesis to combine redundant operations introduced by mapping with operations necessary for computation.

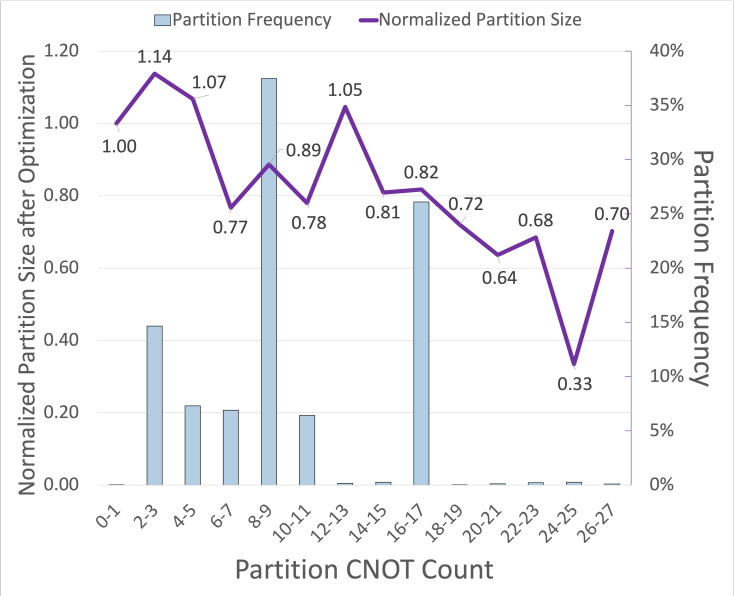


Figure 7: Synthesis performance vs. partition CNOT count. Also pictured is the distribution or frequency of blocks with that number of CNOTs in a set of partitioned benchmark circuits. A partition size of less than one indicates optimization occurred, lower is better.

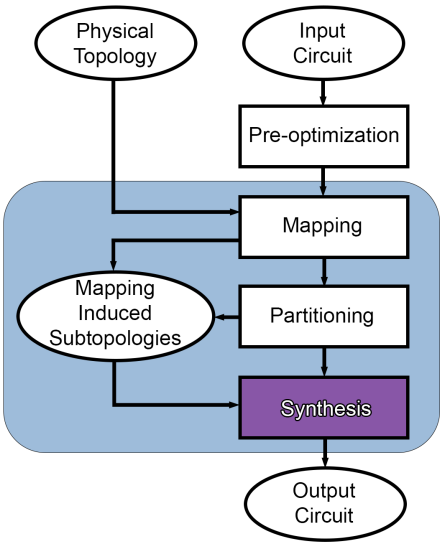


Figure 8: The program flow of a post-mapping synthesis algorithm such as QGo.

2.3.5 TopAS

The program flow of QGo begs the question, can further reductions in depth and gate count be achieved by applying synthesis before mapping? This "pre-mapping" style of optimization is illustrated in Figure 9. TopAS, a Topology Aware Synthesis algorithm for wide circuit optimization, is a pre-mapping synthesis algorithm [27].

The TopAS algorithm has one guiding principle: use synthesis to make the circuit mapping problem easier. Synthesis can be used to achieve this in two ways. First, reducing the number of operations in a circuit before mapping means fewer gates must be mapped, which typically results in better mapping algorithm performance. Second, restricting the complexity of operations in a circuit before mapping helps mapping performance, as simpler communication patterns require fewer mapping operations.

The first observation, that mapping algorithms do better when there are fewer gates, came by observing the performance of Qiskit's implementation of the SABRE Swap algorithm on TFIM circuits mapped to mesh style topologies. The interactions in TFIM circuits are linear, meaning the circuit can be mapped to a linear qubit topology without the use of any SWAP operations. Because linear graphs are embedded within meshes, this implies that TFIM circuits can also be mapped to mesh topologies without the use of SWAP operations. However, our experiments showed that the SABRE mapping algorithm was unable to find a zero SWAP mapping of TFIM circuits, unless synthesis was performed beforehand.

The second observation, that mapping prefers simpler interactions, was

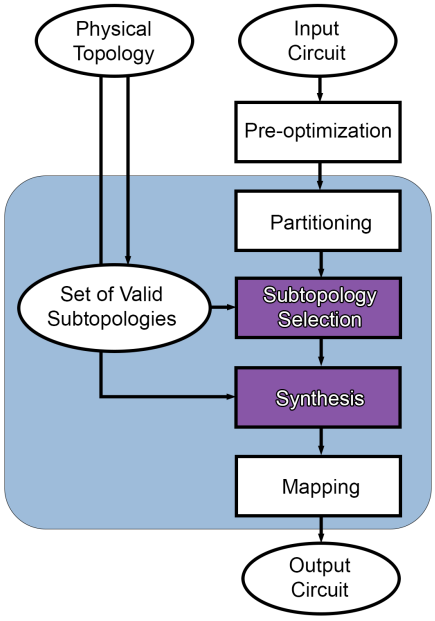


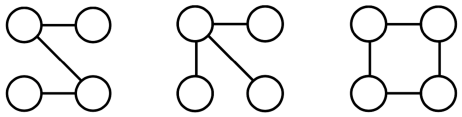
Figure 9: The program flow of a pre-mapping synthesis algorithm such as TopAS.

guided by a shift in perspective regarding mapping algorithms. Once partitions are formed, SWAPs can be divided into two categories: intra-partition SWAPs that exchange state within a partition, and inter-partition SWAPs that exchange qubits between partitions. Inter-partitions SWAPs are unavoidable. They describe movements of quantum information that are necessary for the correct sequence of interactions to take place. Intra-partitions SWAPs, however, describe a partition's suitability for execution on that region of the qubit topology. If that partition were instead mapped to the region of the topology (called a subtopology) on which is it being executed, then no intra-partition SWAPs would be needed.

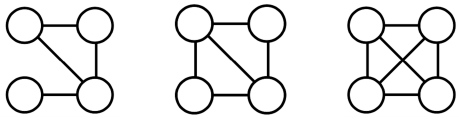
As mentioned in Section 2.3.1, QSearch accepts as input both a target unitary and a graph $G = (V, E)$ which describes the allowed interactions between qubits. This graph G is often denoted G_S and is called a *synthesis subtopology*. Subcircuits synthesized given a synthesis subtopology G_S are also mapped to that synthesis subtopology. Synthesis subtopologies differ from physical subtopologies in that physical subtopologies describe connected subgraphs in a qubit topology, whereas synthesis subtopologies need not have any connection to a real machine. Superconducting qubit machines are rarely fully connected, but synthesis subtopologies can be complete graphs. All order four synthesis subtopologies are illustrated in Figure 10.

The multi-qubit gates in a circuit also define a graph, called the *logical*

connectivity. Each edge in the logical connectivity corresponds to a multi-qubit gate between two qubits in a circuit. The weight of an edge in the logical connectivity corresponds to the number of times that interaction occurs. Logical connectivity graphs are often given the notation $G_L = (V, E)$, where E consists of weighted edges (w, u, v) for $w \in \mathbb{Z}$ and $u, v \in V$.



Embedded Subgraphs a) Line b) Star c) Ring



Unembedded Subgraphs d) Kite e) Theta f) Complete

Figure 10: Possible order 4 synthesis subtopologies. The graphs a, b, and c are embedded within the mesh physical topology. Graphs d, e, and f are not. Only a and b are embedded in the falcon topology, while a is the only embedded subgraph of the linear topology.

When optimization is carried out before mapping, there is no a priori way of determining what synthesis subtopology each partition should be mapped to. Instead, TopAS uses a heuristic approach to select subtopologies for each partition. We observed that partitions synthesized to subtopologies that were similar to their unoptimized logical connectivity graphs typically saw greater reductions in gate counts. In order to quantify the similarity between a partition and a subtopology, we devised a graph similarity function inspired by machine learning kernel functions [15].

Definition 2.9 (Similarity Score)**.** The similarity score function examines each edge in a synthesis subtopology G_S a logical connectivity G_L. From the edges, two vectors $v_L, v_S \in R^{k(k-1)/2}$ are computed. The vector v_S has a 1 at each location that corresponds to a present edge in G_S. Element i of v_L contains the weight w associated with edge i in G_L. The normalized inner product

$$\text{similarity}(v_S, v_L) = \frac{v_P^T v_L}{\sum_i v_L(i)} \tag{5}$$

quantifies the similarity between graphs G_S and G_L.

We also observed that using more restrictive synthesis subtopologies typically resulted in fewer routing operations in the final mapped and optimized circuits. Furthermore, selecting subtopologies embedded in the underlying qubit topologies also resulted in fewer total routing operations. These results are illustrated in Figure 11.

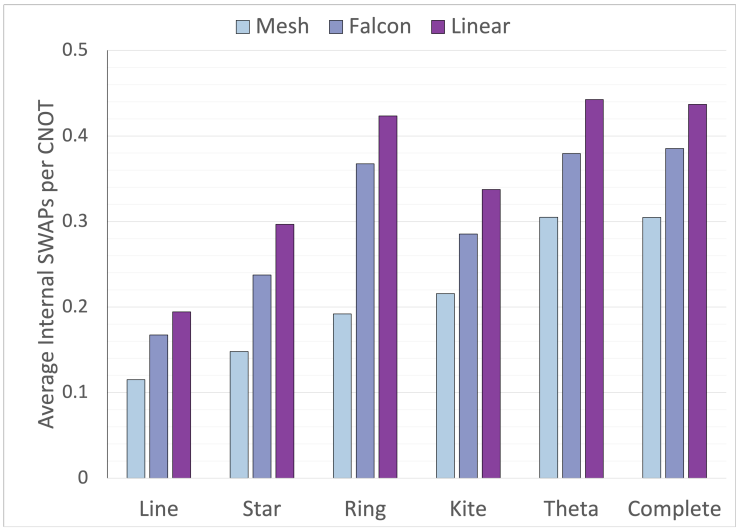


Figure 11: Simpler synthesis subtopologies result in fewer routing operations. Synthesis subtopologies embedded in the underlying qubit topology also result in fewer average routing operations. Internal SWAPs are intra-partition SWAPs.

So, the process of synthesis favors more densely connected synthesis subtopologies, as they are more similar to the logical connectivities of the circuits being synthesized, but the mapping algorithm performs better when more restrictive subtopologies are used. We devised a subtopology selection algorithm that combines and balances these two opposing demands. First, the set of allowed synthesis subtopologies is limited to those that are embedded in the underlying qubit topology. Then, the most restrictive of these subtopologies with the highest similarity score is used.

This subtopology selection policy allowed for TopAS to reach gate count reductions similar too, or just below those of QGo. This policy was able to reduce the number of intra-partition SWAPs, but something had to be done to reduce the number of inter-partition SWAP operations. The development of a *neighbor-aware* subtopology selection scheme was devised for this reason. Although partitioned subcircuits are synthesized independently, they are not executed in isolation. If a given partition can peek into a neighboring block to see what edges it uses, then the likelihood that a SWAP is inserted between blocks can be decreased by using edges that appear in both blocks. To accommodate this change, the weighted logical connectivity graph for each partition was modified to included shared edges in neighboring partitions. The inclusion of neighbor-awareness was the last piece of TopAS' pre-mapping synthesis flow (see Figure 12). The subtopology selection algorithm is detailed in Algorithm 1.

Figures 13 and 14 show the CNOT gate count reduction and depth reduction of TopAS optimized circuits compared to Qiskit, Pytket, and QGo. Benchmarks

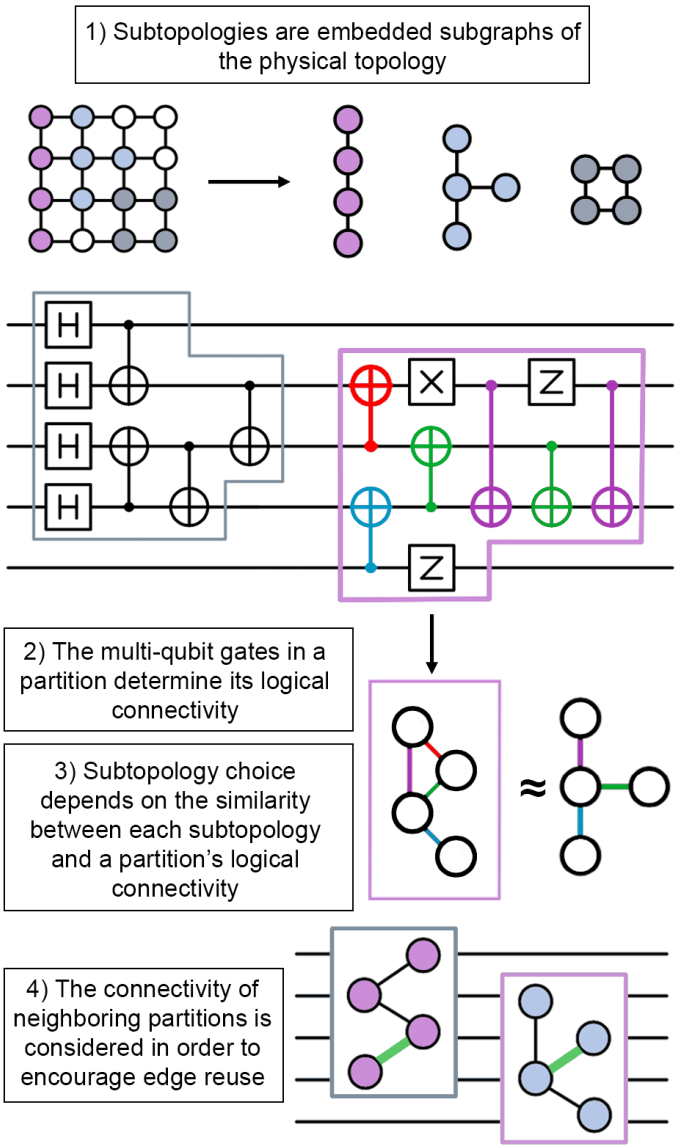


Figure 12: A high level view of the TopAS style implementation of pre-mapping synthesis optimization.

Algorithm 1 Subtopology Selection

Input: graph G_L, set of graphs $G_{neighbors}$, *bias* function
Output: graph G_S

```
for all graphs G_n in G_neighbors do
  if shared_edge in G_n and in G_L then
    G_overlap ← shared_edge
  end if
end for
v_L ← edge weight vector for G_overlap ∪ G_L
candidates ← order ≤ 4 subgraphs in physical topology
for all graphs G_P in candidates do
  v_P ← indicator vector for G_P
  score =similarity(v_P, v_L)
  if bias(G_P) × score > high_score then
    high_score ← bias(G_P) × score
    G_S ← G_P
  end if
end for
return G_S
```

are labeled so that the number following the benchmark name indicates the number of qubits in the circuit. The CNOT gate count and depth results reported are normalized by the values obtained for circuits optimized then mapped with Qiskit's SABRE SWAP mapping algorithm.

TopAS is able to, on average, reduce CNOT count compared to Qiskit by 35.9% and Pytket by 24.3% when targeting a mesh qubit topology. For the falcon/heavy hexagonal qubit topology, TopAS outperforms Qiskit by 33.4% and Pytket by 19.1% on average. TopAS shows even greater reductions in circuit depth, averaging 38.6% and 37.6% shallower circuit compared to Qiskit and Pytket across all benchmarks.

TopAS is able to produce circuits with lower CNOT counts and depth than QGo. On average, TopAS outperforms QGo by 11.5% in CNOT count and 34.3% in depth when compiling to a mesh qubit topology. For the heavy hexagonal topology, TopAS does as well as QGo (only an average of 0.8% better) in CNOT count, but maintains an advantage of 37.3% in average depth reduction.

The TopAS project highlights an interesting fact: unitary synthesis is still able to outperform rule based optimization flows such as Qiskit and Pytket, even when a global view of the unitary being optimized is too large to do direct synthesis on. Synthesizing partitioned circuits may not produce the same results as direct, complete, synthesis, but by operating in a bottom-up manner, results are still impressive.

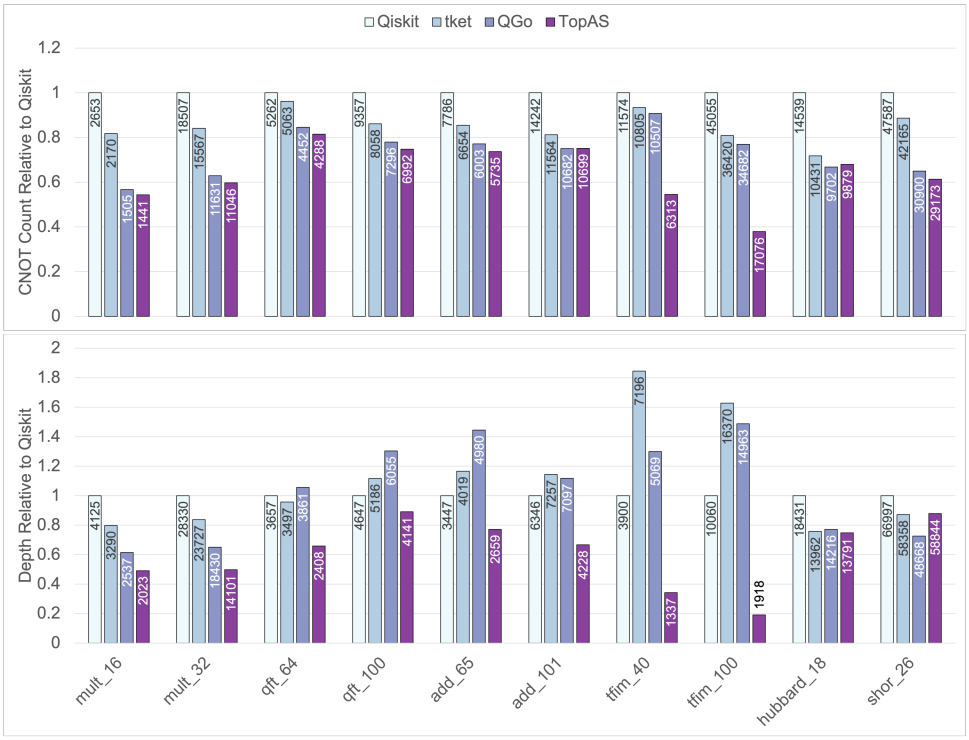


Figure 13: Comparison of relative CNOT count (top) and depth (bottom) for circuits mapped to the mesh qubit topology. CNOT count and depth is shown relative to optimized circuits mapped using Qiskit's SABRE Swap algorithm. Both lower depth and CNOT count correspond to improved output fidelity on noisy machines.

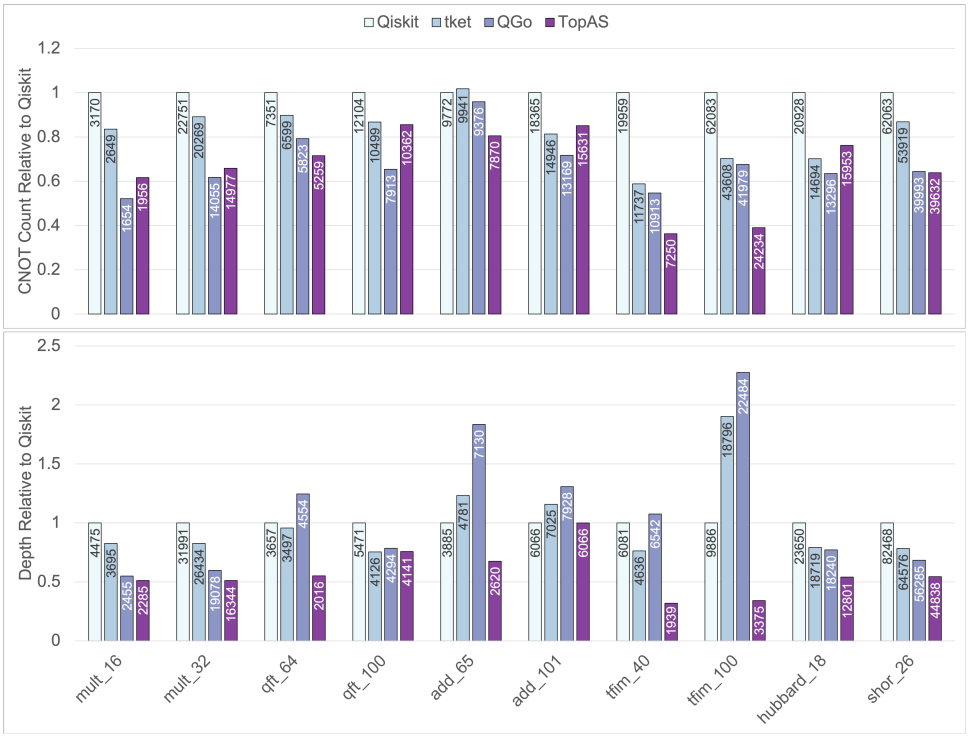


Figure 14: Comparison of relative CNOT count (top) and depth (bottom) for circuits mapped to the IBM style falcon/heavy hexagonal qubit topology. CNOT count and depth is shown relative to optimized circuits mapped using Qiskit's SABRE Swap algorithm. Both lower depth and CNOT count correspond to improved output fidelity on noisy machines.

3 Using Machine Learning for Circuit Optimization

In this section, I will justify the use of machine learning in circuit optimization and present recent work in this field. I will discuss my findings relating to the dimensionality of unitary datasets of interest. Finally, I will discuss QSeed, a seeded synthesis algorithm that leverages machine learning to accelerate unitary synthesis optimization.

3.1 Related Work in Reinforcement Learning

Reinforcement Learning (RL), is a machine learning paradigm where agents learn policies based off rewards received from interacting with an environment. In classical compiling, RL is becoming a popular technique for discovering optimal policies for applying peephole optimizations. The CompilerGym and CompilerZoo tools both provide methods of training RL agents to guide compilation flows in order to minimize the number of machine instructions in programs [8]. RL gyms for quantum compilation do not exist, but they almost certainly will in the future, as this style of optimization has proven to be very powerful in the classical compilation space.

The work of [12] demonstrated how RL can be used to learn good peephole style replacement rules for randomly generated circuits. To my knowledge, this is the first project to have applied machine learning to quantum circuit optimization. I believe that RL policies trained to perform peephole optimization may ultimately become the de facto method of performing circuit optimization. However, I believe a weakness of this particular project is that these agents were trained to discover optimizations in random circuits. As I will show, random circuits, especially random unitaries, do not necessarily correspond to cases that we can or desire to optimize.

3.2 Dimensionality of Unitary Datasets

The manifold hypothesis is a conjecture of deep learning that posits that neural networks operate by learning low dimensional manifolds in high dimensional spaces. Although far from the first to propose the manifold hypothesis, the authors of [24] showed how a digit that is rotated, translated, or stretched still lies along the same manifold. Actually learning the shape of these manifolds is hard, but we can begin to identify some of their properties by looking at high level linear approximations of data. The Principle Component Analysis (PCA) specifically is a widely used technique for doing this [17]. PCA changes the basis of a set of data so that the new directions, or Principle Components (PCs), point in the directions of maximum variance in the dataset. When the dataset can be largely explained by a few PCs, neural networks typically perform well, as patterns of interest in the data are easier to learn in these cases [14].

For a fixed number of qubits, the space of unitaries is large. To present support for this claim, we can conduct an experiment to measure the dimensionality

of two sets of unitaries. First we prepare a set of random unitaries generated by PQCs. These unitaries are generated by taking one of twelve candidate PQCs with parameters $\theta \in \mathbb{R}^m$, and uniformly sampling parameters $\theta_i \sim U[-\pi, \pi]$. Each of the twelve PQCs generate 10,000 unitaries. Next, we prepare a dataset that consists of about 203,000 unitaries taken from partitioned subcircuits. For each unitary, random and partitioned, we extract a *Pauli coefficient vector*.

Definition 3.1 (Pauli Coefficient Vector)**.** Given a unitary $U \in \mathbb{C}^{2^n \times 2^n}$, there exists a Hermitian matrix $H \in \mathbb{C}^{2^n \times 2^n}$ such that

$$U = e^{iH}. \tag{6}$$

This matrix H can be decomposed into the linear combinations of all 4^n Pauli matrices. Concretely, we can construct a vector of Pauli matrices

$$\sigma_1 = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix},\ \sigma_2 = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix},\ \sigma_3 = \begin{bmatrix} 0 & -i \\ i & 0 \end{bmatrix},\ \sigma_4 = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}$$

$$\vec{\sigma} = \begin{bmatrix} \sigma_1 \otimes \sigma_1 \cdots \otimes \sigma_1 & \sigma_2 \otimes \sigma_1 \cdots \otimes \sigma_1 & \ldots & \sigma_4 \otimes \sigma_4 \cdots \otimes \sigma_4 \end{bmatrix}$$

such that,

$$H = \alpha \cdot \vec{\sigma} \tag{7}$$

where $\alpha \in \mathbb{R}^{4^n}$ and $\alpha_i \in (-\pi, \pi]$. The vector α is called the Pauli coefficient vector. The Pauli coefficient vector is a convenient way to represent matrices as representing a unitary matrix requires 4^n *complex* values, while the Pauli coefficient vector requires only 4^n *real* values.

After the Pauli coefficient vectors have been collected for each unitary, we can do two PCAs; one for the random and one for the partitioned unitaries. Figure 15 shows the cumulative variance explained as a function of the number of principle components considered. Because a small number of PCs describe the majority of the variance in the partitioned dataset (the first 16 explain 72.2%, the first 32 explain 90.7%), we expect patterns of interest to be learnable by a neural network. The random dataset's variance is spread even across its PCs. We would not expect that learning patterns of interest in this dataset is easy to do.

A natural way to think of what a PCA shows is how compressible a dataset is. For the partitioned unitary case, we could project our dataset into the space spanned by the first 32 principle components, then reconstruct the original dataset with 90.7% accuracy. To achieve this same level of accuracy on the random unitary dataset, the first 57 principle component directions must be kept. In this sense, the random unitary data is much higher dimensional than the partitioned unitary case.

This analysis shines a light on an important insight, random unitaries are very different from unitaries that come from circuits that do real computation. Random circuits (that implement random unitaries) are not representative of what we expect to see in real benchmarks.

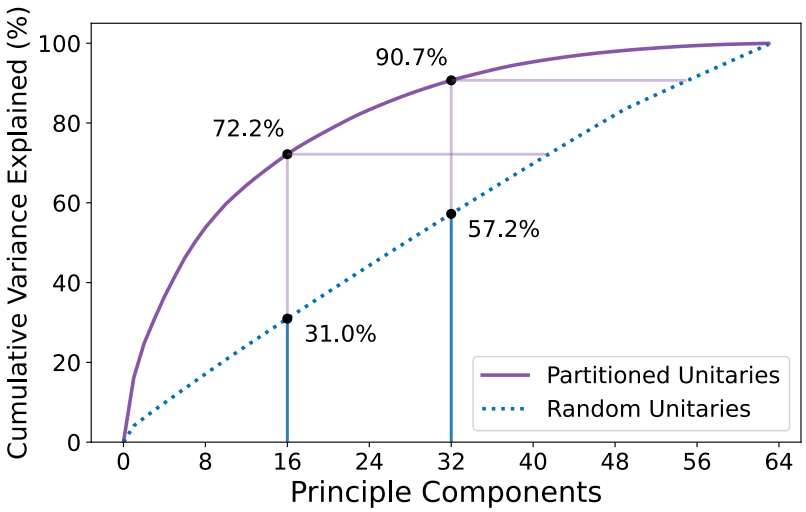


Figure 15: Cumulative variance explained by Principle Components of two datasets of three qubit unitaries. One is randomly generated, the other is made of partitioned unitaries from circuits in Table 1. The PCA of the partitioned unitaries implies these unitaries lie on a low dimensional manifold; there are patterns that can be identified in this dataset. The random unitaries do not exhibit identifiable low dimensional patterns. Learning on random unitaries is difficult, learning on partitioned unitaries is easier.

3.3 Learning a Mapping from Unitaries to Quantum Circuits

Now that I have established that we expect to be able to learn patterns of interest in partitioned unitaries, what can we do with this information?

Imagine that we have a series of n qubits, and a set of unitaries that these n qubits can implement $\mathcal{U}(2^n)$. We can also imagine a set $\mathcal{PQC}(n, K)$, which represents all PQCs with n qubits and K or fewer CNOT gates. We want to find a mapping

$$\phi : \mathcal{U}(2^n) \longrightarrow \mathcal{PQC}(n, K) \tag{8}$$

such that all $U \in \mathcal{U}(2^n)$ are mapped to the $PQC \in \mathcal{PQC}(n, K)$ that implements U with the fewest CNOTs. We assume that the set $\mathcal{PQC}(n, K)$ is large enough so that some PQC implements U with error at most ϵ.

Why might learning this mapping be useful? Because if we can, then we can replace the expensive synthesis tree search process described in Section 2.3.1 with much less expensive ML inference! Before this is possible, there are several things that must be addressed.

3.3.1 Expressivity of Parameterized Quantum Circuits

Much work has been done to quantify the expressivity of parameterized quantum circuits [16]. Many unitaries can be implemented by any given PQC, and many PQCs can implement any given unitary. It is important to note this because it makes the problem of finding the mapping $\phi : \mathcal{U}(2^n) \to \mathcal{PQC}(n, K)$ difficult. This explains why the PCA of random unitaries (Figure 15) shows that the variance is spread evenly across the principle components.

Another added difficulty of working with such expressive circuits is that is makes the problem of instantiation (Equation (3)) more difficult. To explain how we know this, we can again return to the example of the random unitary dataset. Here, we directly generate unitaries from a given PQC by randomly sampling parameters. But, if we have the unitary and the PQC, asking the instantiation function to solve for the parameters of that circuit that yield that unitary often fails. This means that even when we know the correct mapping of random unitaries to the PQCs that generated them, it is difficult to verify that this mapping is correct.

3.3.2 Canonical Representation of Unitaries

If two unitaries differ only by a global phase, states transformed by the two unitaries will always yield the same distribution of measurement results [19]. This means that for all intents and purposes, unitaries that differ only by a global phase are the same. However, if we want to apply numerical methods to unitaries, it is important that there is a common, canonical way of representing unitaries. Formally, we need a function f such that for unitaries U, U^*,

$$\forall \theta \in \mathbb{R} \quad U = e^{i\theta} U^* \implies f(U) = U^*.$$

If $f(U) = U^*$ then the matrix U^* is the canonical representation of U. The algorithm for computing U^* is given by Algorithm 2. Performing these steps

Algorithm 2 Canonical Unitary Transformation

Input: unitary $U \in \mathbb{C}^{N \times N}$,
Output: unitary $U^* \in \mathbb{C}^{N \times N}$

```
Find e^{iθ} = |U|
Compute V = e^{iθ} U = e^{i2πk/N} U* ∈ 𝕊𝕌(N)
e^{i2πk/N} is the complex phase of first non-zero element in V
U* = e^{-i2πk/N} V
return U*
```

ensures that each unitary is mapped to a canonical special unitary. To my knowledge, this is the first algorithm that finds a canonical form for unitaries.

Previous works stop at the special unitary conversion step. Just doing this part is not enough to build global phase resilience into methods that do numerical computation on unitaries. I noticed by looking at many random examples of

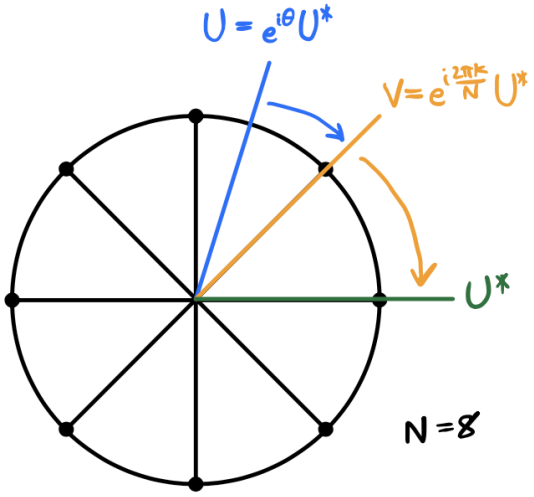


Figure 16: An illustration of the two steps in the canonical unitary transformation process.

this process, that for an $N \times N$ unitary, there were N different special unitaries that appeared. This lead me to notice that converting to a special unitary only removes the small phase differences between the different roots of unity (the blue arrow in Figure 16). In order to find a real canonical form, we need to always go to the same root of unity. A unitary will always have at least one non-zero element in each row and column. If we ensure that the first non-zero element of the first row has zero complex phase, we will always arrive at the same root of unity, no matter which root the special unitary conversion step took us to (orange arrow in Figure 16).

Even though this process is simple, I believe that it is a pretty significant tool to have. Unitary synthesis algorithms, and other numerical methods that work on unitary matrices, must choose cost functions that are resilient to global phase, otherwise intricacies of the numerical optimization process may skip over valid solutions to problems. Having a canonical representation is therefore significant because it broadens the scope of what can be done with unitary matrices. Cost functions need not be phase resilient anymore.

3.4 QSeed

Equipped with a canonical representation of unitary matrices and the knowledge that patterns of interest could be learned for partitioned unitary matrices, I began work on using machine learning to learn a mapping between unitaries and the PQCs that implement them. This work culminated in an algorithm called QSeed, a seeded synthesis algorithm that uses ML to predict seed circuits from which to begin synthesis. QSeed is able to achieve dramatic speedups compared

to QSearch because it decreases the time spent searching the synthesis circuit tree.

3.4.1 Why use ML for this task?

Before detailing the development behind QSeed, first I want to address why using ML might be advantageous for this task. If we want to learn a mapping from unitaries to PQCs, could we instead just observe many instances of unitaries that can implement PQCs and memorize those results? This approach would construct a database of previously seen unitary-PQC pairs that we could quickly interrogate. The problem with this approach is that it only allows us to do this mapping for unitary-PQC pairs that have already been observed.

Why might ML work any better than this database? A learned mapping is more robust than this database approach because the point of the learned mapping is that it can generalize to cases that have not yet been seen. This is useful because the number of possible unitaries we can see is vast (even when limiting ourselves to unitaries from partitioned circuits), and surely we cannot capture everything we might expect to see in a training dataset.

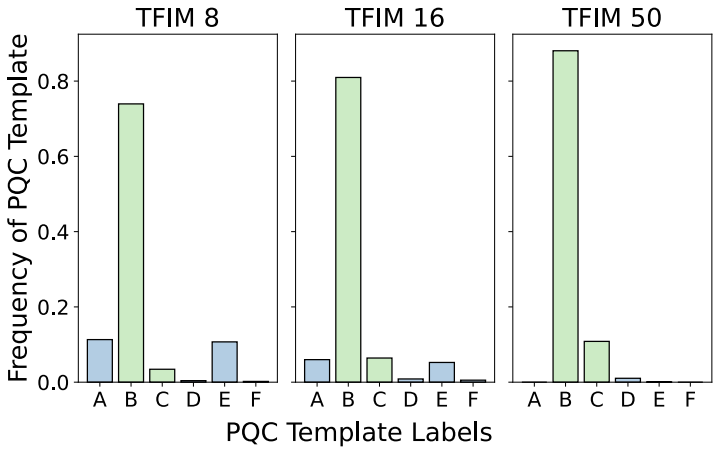
Why might we expect that learning this mapping is possible? We expect this is possible because there are common patterns that appear in similar quantum circuits. When synthesizing generated circuits from the same "family" but of different widths, the distribution of PQC templates that appear has similarities. This is illustrated in Figure 17. This figure shows how synthesized TFIM circuits of various widths all have similar looking distributions of used PQC templates, which are also illustrated.

Furthermore, circuits not from the same family, but that are still in some sense related, share common structure as well. Figure 18 shows how both QFT and Shor style modular exponentiation circuits have distributions of selected PQCs that look similar. The modular exponentiation (Shor) distribution look like the QFT distribution, with a extra mode around the template labeled "P". The existence of these commonalities means that there is in fact some underlying structure or pattern to the mapping of unitary to PQCs. Now we can rest assured that there are in fact patterns of interest in our data that we can train a model to recognize.

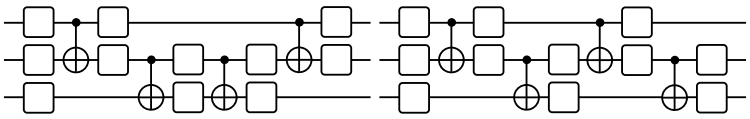
3.4.2 Seeded Synthesis

If this mapping between unitaries and PQCs is learned, then the synthesis circuit tree search process could be almost entirely replaced. In actuality, it is unlikely that this mapping will be learned perfectly, so instead of immediately using the first PQC proposed, we simply begin synthesis from that circuit.

Definition 3.2 (Seeded Synthesis)**.** Seeded synthesis accepts three inputs, a target unitary U, a qubit topology $G = (V, E)$, and a seed circuit C_{seed}. Instead of beginning the synthesis tree search process at the root node of the tree (which contains only single-qubit gates), synthesis begins at the circuit C_{seed}. CNOT

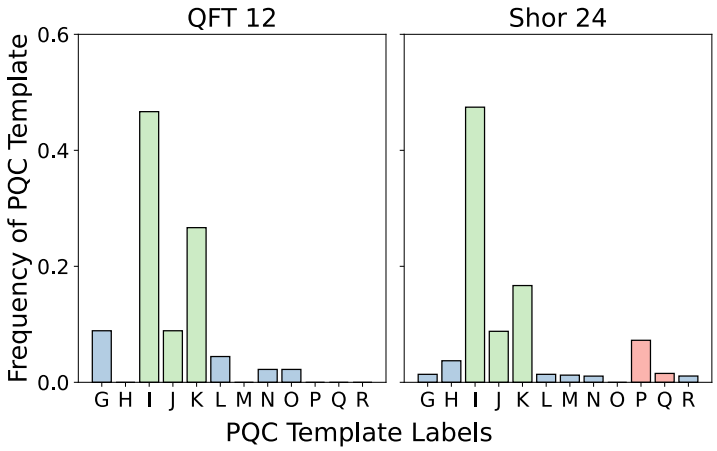


(a) PQC templates that appear in synthesized TFIM circuits of various widths. Circuits of different widths share characteristic modes around PQCs B and C.

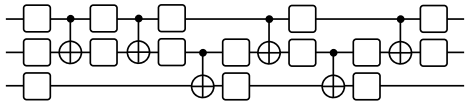


(b) Template B (c) Template C

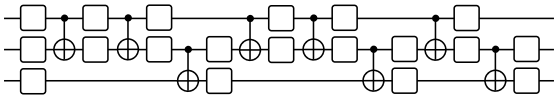
Figure 17: Histograms and examples of commonly repeated PQC templates that appear in optimized TFIM circuits.



(a) PQC templates that appear in synthesized QFT and Shor modular exponentiation circuits.



(b) Template I



(c) Template P

Figure 18: Histograms and examples of commonly repeated PQC templates that appear in optimized QFT and modular exponentiation circuits.

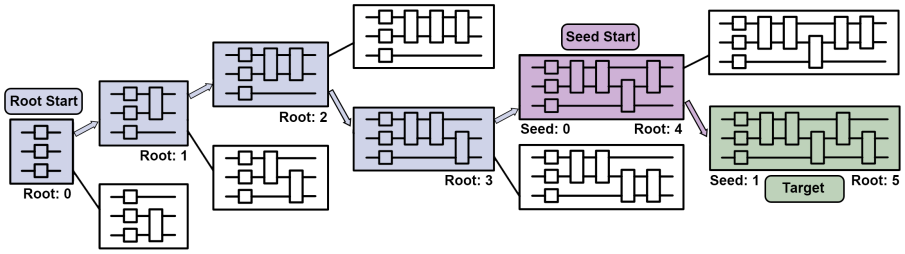


Figure 19: Seeded synthesis allows for the synthesis tree search process to begin at circuits closer to the target circuit node.

gates can now be both added and removed, which means that the tree search progresses both towards leaf nodes and the root node.

The purpose of the learned mapping between unitaries and PQCs is now simply to propose seed circuits from which the synthesis process can begin. Figure 19 shows why seeded synthesis can provide speedups compared to normal synthesis. QSearch always begins synthesis at the node marked "Root Start." In this case, the path between the root and the "Target" has distance five. This means if the shortest path is taken, then at least six instantiation calls must be made. If we instead begin synthesis at the node marked "Seed Start," because the path to the target has distance one, only two instantiation calls must be made. This example illustrates why we might expect to see large speedups in a seeded synthesis algorithm compared to QSearch.

3.4.3 Dataset Preparation

I took a supervised learning approached to learn the mapping in Equation (8). After collecting unitaries, computing their canonical forms (Section 3.3.2), and extracting their Pauli coefficient vectors (Definition 3.1), these datapoints still lacked labels. I approached this problem by enumerating all PQC templates with three qubits and up to eight CNOTs, then synthesizing each unitary. Each unitary was then paired with the integer label that corresponded to the PQC that implemented it. Because some unitaries appeared multiple times and some PQCs were more frequent than others, I normalized this dataset by removing repeated unitaries and randomly discarding examples from over represented PQCs. This ensured that whatever model I trained would not simply learn that some PQCs were more prevalent than others, but actually learn to propose PQCs based off the unitaries themselves.

The process of producing these labels is synthesis, and therefore labeling these unitaries is expensive.

3.4.4 The Recommender Model

A recommender is any function (learned or not) that proposes seed circuits for seeded synthesis. I chose to implement a recommender model with a neural network. More specifically, I chose an autoencoder style architecture. The results of the PCA of partitioned unitaries in Figure 15 motivated this choice. A purely linear projection of only half the total dimensionality of the original space captured the vast majority of the variance in the partitioned unitary dataset. I therefore believed that the underlying structure of these unitaries was simple, and that compressing their representation could add a degree of robustness to noise [14]. As the process of labeling the data for this task was expensive, I also wanted to ensure I could use unsupervised techniques to pre-train the model. Autoencoders can easily be pre-trained on the task of input reconstruction, so the recommender model was pre-trained to reconstruct input unitaries.

The output of the recommender model is a large vector where element i corresponds to how likely the model believes the PQC with label i is able to implement the input unitary. There were a total of 1199 possible output PQCs to select. To state this concretely, the output of the model is a vector $y \in \mathbb{R}^{1199}$, where

$$softmax(y_i) \in [0, 1] \tag{9}$$

represents the model's confidence that PQC i can implement the input.

As this was meant primarily as a proof of concept, I elected to limit the width of unitaries and circuits to three qubits. I began by training a model which took as input the Pauli coefficient vector associated with each unitary, which is a vector in $\mathbb{R}^{64}$. Later on, I discovered that the process of converting unitaries into Pauli coefficient vectors was extremely expensive, and so the architecture was changed to instead operate directly on the canonical form of unitaries. As elements of the unitaries are complex, each element of the input unitary was split into real and imaginary parts. These new vectors $x \in \mathbb{R}^{128}$ were used as the new input.

The training and test datasets consisted of circuits from twelve different families of generated circuits. Each family contained multiple instances of that algorithm implemented at various circuit widths. The training and test data splits are shown in Table 1. Note that no Grover's algorithm circuits of any size appear in the training dataset. The whole flow of the QSeed algorithm is depicted in Figure 20.

3.4.5 Comparing QSeed to QSearch

Recall that the goal of the QSeed algorithm is to accelerate unitary synthesis by avoiding spending too much time traversing the synthesis circuit tree. We can measure this by directly measuring the time it takes to synthesize circuits with both QSearch and QSeed. Importantly, we do not want to degrade the quality of our results for the sake of speed. Therefore, we must also ensure that

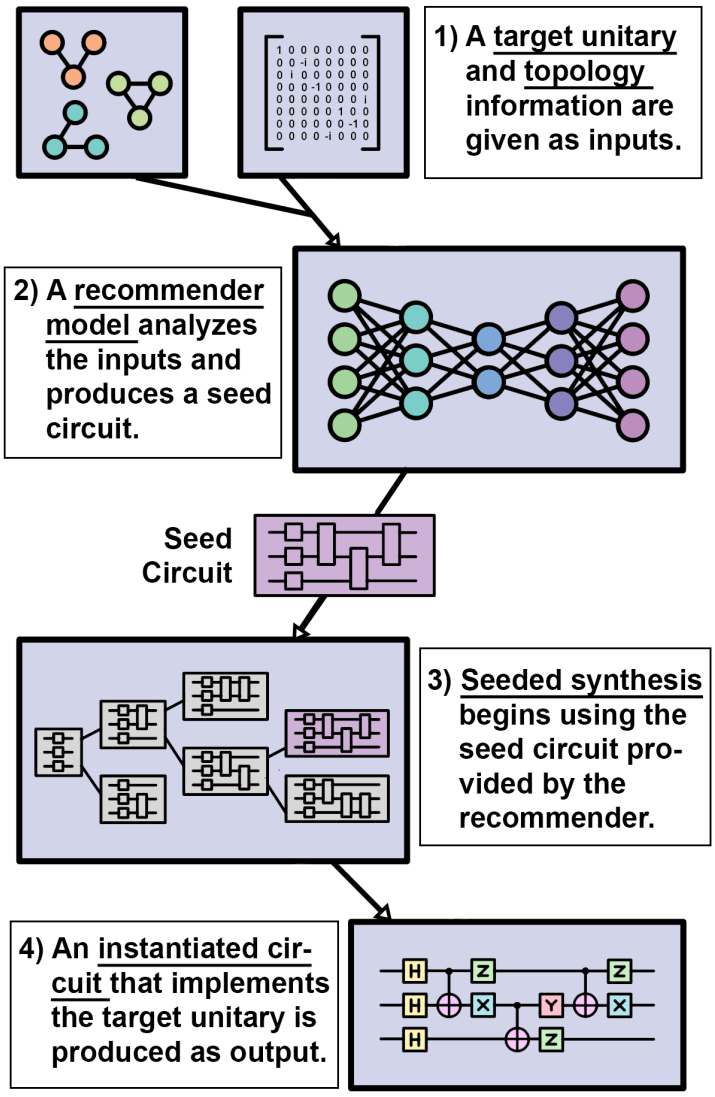


Figure 20: A high level depiction of the QSeed algorithm.

Circuit Name	Training Widths	Test Widths
add	17, 65	41
grover	-	10
heisenberg	4, 6, 7, 8, 16, 32, 64	5
hhl	8	6
hubbard	4, 18, 50	8
mult	8, 32, 64	16
qae	11, 33, 101	65
qft	3, 4, 8, 16, 32	64
qml	4, 25, 60, 108	128
qpe	6, 10, 14	18
shor	16, 32	64
tfim	3, 4, 5, 6, 7, 8, 16, 32,	64
vqe	12, 14	18

Table 1: Split of circuits withheld from recommender training for testing.

the number of CNOTs in the QSeed synthesized circuits is close to the number of CNOTs in the QSearch synthesized circuits.

Figure 21 summarizes the results of experiments run to compare the time to- and quality of- solutions for circuits optimized with QSearch and QSeed. Each of these circuits is first mapped with Qiskit's SABRE SWAP mapping algorithm, then optimized with QGo style post-mapping synthesis (Figure 8). QSeed achieves an average speedup of 2.4× compared to QSearch, while only suffering an average of 1.2% increase in CNOT count. Note that these test circuits contain unitaries not included in the training of the recommender model. The 64 qubit modular exponentiation circuit, a core component of Shor's algorithm, has a speedup of 3.7× (nearly 8000 seconds). The 18 qubit VQE circuit achieves a speedup of 2.3×. This particular benchmark is worth noting as classical-quantum hybrid algorithms have strict compilation time demands [20]. It is also worth noting that QSeed's synthesis times include the time required to do inference/make seed recommendations.

Most impressive are the results for the compilation of the 10 qubit Grover's algorithm circuit. No unitaries for the family of Grover's algorithm circuits were included in training, so this benchmark is entirely alien to the recommender model. Still, a speedup of 3.2× is achieved, and there are actually fewer CNOTs than in the QSearch optimized version.

3.4.6 Evaluating the Recommender Model

The results presented in Figure 21 demonstrate that using the recommender model and seeded synthesis provides substantial speedups over QSearch. However, one can imagine other recommender schemes. To ensure that our recommender model is performing as expected, we can do a deeper analysis of what is happening each time a partition is synthesized. To do this, we can observe the

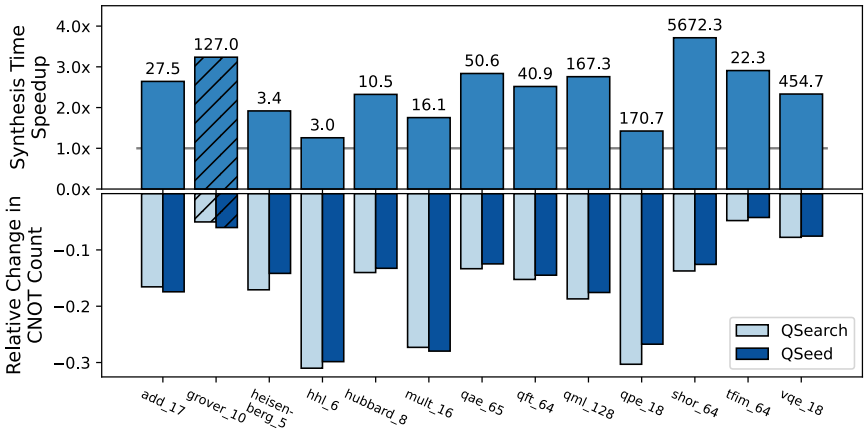


Figure 21: Comparing QSeed and QSearch. The top graph wall-clock synthesis time speedups. Larger speedups are better. The bottom graph shows relative changes in CNOT gate count relative to Qiskit level 3 optimized circuits. Lower relative change in CNOT counts are better.

number of instantiation calls made and the number of CNOTs in the optimized version of each block.

Recall that the number of instantiation calls is a proxy of how long synthesis will take, because each circuit along a path through the synthesis circuit tree must be instantiated. Figure 22 shows the distribution of the number of instantiation calls for the QSeed, randomly seeded, and QSearch approaches. The randomly seeded scheme simply randomly selects one of the 1199 possible seeds. QSearch acts like a seeded synthesis algorithm that always proposes the root as a seed. QSeed's recommender scheme is able to propose seeds that require only a single instantiation call almost 90% of the time. The randomly seed case also typically requires very few instantiation calls. By measuring the expected number of instantiation calls, we can see that on average, QSeed and the randomly seeded scheme require almost 6× fewer instantiations on average. This is why QSeed is able to synthesize unitaries faster than QSearch.

Figure 23 shows the number of CNOTs in the optimized blocks divided by the number of CNOTs in the original blocks. A value of 1 here indicates that no reduction in CNOTs has occurred, whereas a value less than 1 indicates the optimized subcircuit has fewer CNOTs. Lower values are better. Values higher than 1 indicate growth, which is undesirable. The expected reduction for QSeed closely matches that of QSearch. This is why we can expect that QSeed produces circuits with a similar number of CNOTs as QSearch. The randomly seeded scheme may offer speedups, but it averages an increase of 19% in CNOTs.

The randomly seeded results for these two experiments indicate that a randomly proposed PQC is likely to be capable of implementing a unitary, but

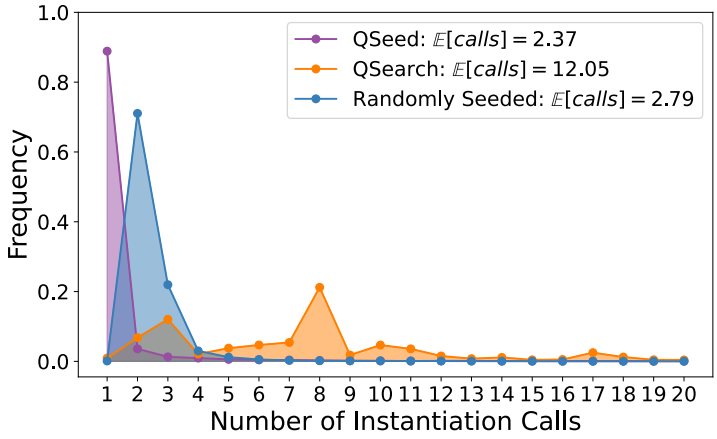


Figure 22: Distribution of instantiation calls made for various recommender schemes. QSeed uses the autoencoder style recommender model while the Randomly Seeded strategy randomly picks a seed circuit. Both are faster than QSearch as seen by the lower expected number of instantiation calls. The X-axis only shows up to 20 instantiation calls, but QSearch requires as many as 1704.

it is likely to be far deeper than is necessary. These experiments also show why QSeed is able to achieve speedups over QSearch while maintaining solution quality.

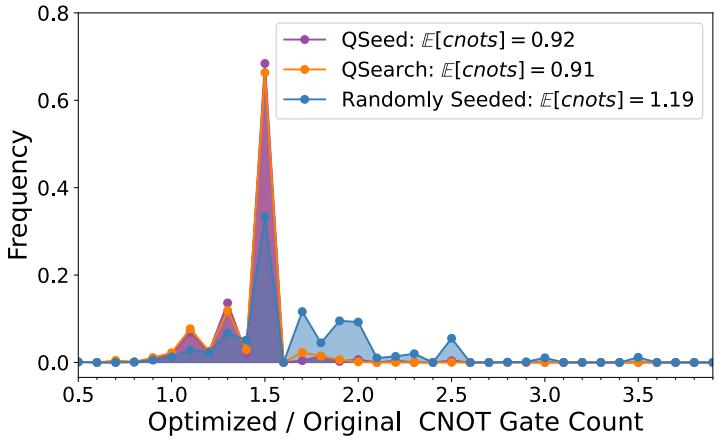


Figure 23: Distribution of relative CNOT gate counts for various recommender schemes. A value less than one corresponds to there being fewer gates in optimized block compared to the original block. Lower is better. The relative CNOT counts of QSeed closely match those of QSearch. The Randomly Seeded scheme results in circuit growth, as seen by the expected value being larger than 1.

4 Discussion and Future Directions

In this section, I will offer further discussion and describe future research directions that could further leverage machine learning for quantum circuit optimization.

4.1 Increasing Unitary Widths

Although promising, the results presented in Section 3 are limited to three qubit unitaries. A PCA of partitioned unitaries with four and five qubit widths show similar structure; therefore we expect that learning is still possible in these cases. Increasing the width of the unitaries operated on by the QSeed algorithm is limited by the lack of a labeled dataset of larger unitaries. Compiling such a dataset for the three qubit case was possible, but synthesis is much slower in these wider cases, so labeling becomes much more difficult.

When the width of the unitaries in the QSeed algorithm increase, so does the number of possible output PQCs. This is a problem because it means the enumeration process used to exhaustively label PQCs is no longer viable. Luckily, it appears that not all PQCs are equally important. In the dataset of 203,000 partitioned unitaries. Only 415 appear more than 10 times. Of these, only 220 appear 100 or more times. I therefore expect that the number of PQCs that are widely used in the larger width cases to be similarly small.

The results presented in Figure 22 show that proposing any seed in the three qubit case often works, though not optimally. This is because the number of circuits capable of implementing a given unitary is high; there are many possible solution circuits. It seems that as widths increase, the complexity of both unitaries and circuits also increases substantially. This means that solution circuits become more sparse, and this random seed strategy is expected to perform far worse. This also means that a good recommender model is likely capable of demonstrating even larger speedups in these wider unitary cases.

4.2 Network Architectures for Learning on Unitaries

The selection of an autoencoder style neural network for the recommender model was motivated by the PCA results of Figure 15. There are likely architectures that are better suited to this task. This is especially true for cases where wider unitaries are considered, as the number of input parameters grows by 4 with the addition of each new qubit.

Convolutional Neural Networks (CNNs) are popular in computer vision tasks as they are translationally equivarient and particularly adept at working with data which exhibits spatial locality. CNNs are likely not well suited to the task of learning on unitaries, as the placement of a gate on the first of last qubit is in some sense very similar, but results in very differently structured unitaries.

Graph Neural Networks (GNNs) are a generalization of CNNs that have gained much popularity recently. It could be possible that using a GNN to learn a dense representation of a quantum circuit expressed as a Directed Acyclic

Graph could address the issues associated with the number of input parameters scaling with $O(4^n)$. Because unitaries of interest (non-random unitaries) are lower dimensional than their general counterparts (which use all 4^n parameters), it is worth investigating if one can learn a dense representation directly from a quantum circuit, without explicitly computing the full unitary.

4.3 Further Exploring the use of Reinforcement Learning

The work presented in [12] show that reinforcement learning can be used to learn good peephole style optimization techniques. I am interested to see whether these results can be further improved by exposing the agent exclusively to non-randomly generated circuits. I expect that this was not done originally because the number of non-random quantum circuits that do useful computation is rather limited.

I also wonder if a reinforcement learning agent can be trained to learn good synthesis tree search strategies. Paired with seeded synthesis, this could provide additional benefits over QSearch's static A^* heuristic approach.

4.4 Learning to Incorporate Noise Resilience

How machine learning can be used to incorporate noise resilience is a research direction that also requires more attention. In [9], ML is used to learn a noise model for a machine. By executing a circuit consisting of only Clifford gates, and simulating this same circuit (which can be done efficiently as all gates are Cliffords), deviations in the expected and actual outcomes offer the necessary signals to train a neural network. I believe it is worth exploring other methods of predicting the amount of noise in or output fidelity of circuits executed on hardware.

I believe the role of machine learning in quantum circuit optimization, and quantum compilation as a whole, will surely grow over time.

References

[1] D. Aharonov and M. Ben-Or, "Fault-tolerant quantum computation with constant error rate," 1999.

[2] G. Aleksandrowicz, T. Alexander, P. Barkoutsos, L. Bello, Y. Ben-Haim, D. Bucher, F. Cabrera-Hernández, J. Carballo-Franquis, A. Chen, C. Chen *et al.*, "Qiskit: An open-source framework for quantum computing," *Accessed on: Mar*, vol. 16, 2019.

[3] F. Arute, K. Arya, R. Babbush, D. Bacon, J. C. Bardin, R. Barends, R. Biswas, S. Boixo, F. G. Brandao, D. A. Buell *et al.*, "Quantum supremacy using a programmable superconducting processor," *Nature*, vol. 574, no. 7779, pp. 505–510, 2019.

[4] S. Bansal and A. Aiken, "Automatic generation of peephole superoptimizers," vol. 41, 10 2006, pp. 394–403.

[5] A. Ben-Aroya and A. Ta-Shma, "On the complexity of approximating the diamond norm," 2009.

[6] S. Bravyi, R. Shaydulin, S. Hu, and D. Maslov, "Clifford circuit optimization with templates and symbolic pauli gates," *Quantum*, vol. 5, p. 580, nov 2021. [Online]. Available: https://doi.org/10.22331%2Fq-2021-11-16-580

[7] S. Bravyi, S. Sheldon, A. Kandala, D. C. Mckay, and J. M. Gambetta, "Mitigating measurement errors in multiqubit experiments," *Physical Review A*, vol. 103, no. 4, apr 2021. [Online]. Available: https://doi.org/10.1103%2Fphysreva.103.042605

[8] C. Cummins, B. Wasti, J. Guo, B. Cui, J. Ansel, S. Gomez, S. Jain, J. Liu, O. Teytaud, B. Steiner, Y. Tian, and H. Leather, "CompilerGym: Robust, Performant Compiler Optimization Environments for AI Research," in *CGO*, 2022.

[9] P. Czarnik, A. Arrasmith, P. J. Coles, and L. Cincio, "Error mitigation with clifford quantum-circuit data," *Quantum*, vol. 5, p. 592, nov 2021. [Online]. Available: https://doi.org/10.22331%2Fq-2021-11-26-592

[10] M. G. Davis, E. Smith, A. Tudor, K. Sen, I. Siddiqi, and C. Iancu, "Heuristics for quantum compiling with a continuous gate set," 2019. [Online]. Available: http://arxiv.org/abs/1912.02727

[11] R. P. Feynman, "Simulating physics with computers," *International journal of theoretical physics*, vol. 21, no. 6/7, pp. 467–488, 1982.

[12] T. Fosel, M. Niu, F. Marquardt, and L. Li, "Quantum circuit optimization with deep reinforcement learning," *null*, 2021.

[13] C. Gidney and D. Bacon, "The cirq developers, quantumlib/-cirq: A python framework for creating, editing, and invoking noisy intermediate scale quantum (nisq) circuits."

[14] G. Hinton and R. Salakhutdinov, "Reducing the dimensionality of data with neural netowrks," *Science*, pp. 504–7, 2006. [Online]. Available: https://pubmed.ncbi.nlm.nih.gov/16873662/

[15] T. Hofmann, B. Schölkopf, and A. J. Smola, "Kernel methods in machine learning," vol. 36, no. 3, 2008. [Online]. Available: http://arxiv.org/abs/math/0701907

[16] Z. Holmes, K. Sharma, M. Cerezo, and P. J. Coles, "Connecting ansatz expressibility to gradient magnitudes and barren plateaus," *Quantum*, vol. 3, no. 1, p. 010313, 2022. [Online]. Available: http://arxiv.org/abs/2101.02138

[17] I. T. Jolliffe and J. Cadima, "Principal component analysis: a review and recent developments," *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, vol. 374, no. 2065, p. 20150202, 2016. [Online]. Available: https://royalsocietypublishing.org/doi/abs/10.1098/rsta.2015.0202

[18] A. Y. Kitaev, "Quantum computations: algorithms and error correction," *Russian Mathematical Surveys*, vol. 52, no. 6, p. 1191, dec 1997. [Online]. Available: https://dx.doi.org/10.1070/RM1997v052n06ABEH002155

[19] Michael Nielsen and Isaac Chuang, *Quantum Computation and Quantum Information*, 2nd ed. Cambridge University Press, 2010.

[20] A. Peruzzo, J. McClean, P. Shadbolt, M.-H. Yung, X.-Q. Zhou, P. J. Love, A. Aspuru-Guzik, and J. L. O'Brien, "A variational eigenvalue solver on a photonic quantum processor," *Nature Communications*, vol. 5, no. 1, p. 4213, 2014. [Online]. Available: https://doi.org/10.1038/ncomms5213

[21] J. Preskill, "Quantum computing in the NISQ era and beyond," *Quantum*, vol. 2, p. 79, 2018.

[22] V. V. Shende, S. S. Bullock, and I. L. Markov, "Synthesis of quantum-logic circuits," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 25, no. 6, pp. 1000–1010, 2006.

[23] P. W. Shor, "Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer," *SIAM Journal on Computing*, vol. 26, no. 5, pp. 1484–1509, oct 1997. [Online]. Available: https://doi.org/10.1137%2Fs0097539795293172

[24] P. Simard, Y. LeCun, J. S. Denker, and B. Victorri, "Transformation invariance in pattern recognition-tangent distance and tangent propagation," in *Neural Networks: Tricks of the Trade, This Book is an Outgrowth of a 1996 NIPS Workshop*. Berlin, Heidelberg: Springer-Verlag, 1998, p. 239–27.

[25] D. R. Simon, "On the power of quantum computation," *SIAM Journal on Computing*, vol. 26, no. 5, pp. 1474–1483, 1997. [Online]. Available: https://doi.org/10.1137/S0097539796298637

[26] S. Sivarajah, S. Dilkes, A. Cowtan, W. Simmons, A. Edgington, and R. Duncan, "t— ket¿: A retargetable compiler for nisq devices," *Quantum Science and Technology*, 2020.

[27] M. Weiden, J. Kalloor, J. Kubiatowicz, E. Younis, and C. Iancu, "Wide quantum circuit optimization with topology aware synthesis," *2022 IEEE/ACM Third International Workshop on Quantum Computing Software (QCS)*, 2022.

[28] X.-C. Wu, M. G. Davis, F. T. Chong, and C. Iancu, "Reoptimization of quantum circuits via hierarchical synthesis," in *2021 International Conference on Rebooting Computing (ICRC)*, 2021, pp. 35–46.