POPQC: Parallel Optimization for Quantum Circuits

Pengyu Liu
pengyuliu@cmu.edu
Carnegie Mellon University
Pittsburgh, PA, USA

Jatin Arora*
jatina29@gmail.com
Carnegie Mellon University
Pittsburgh, PA, USA

Mingkuan Xu
mingkuan@cmu.edu
Carnegie Mellon University
Pittsburgh, PA, USA

Umut A. Acar
umut@cmu.edu
Carnegie Mellon University
Pittsburgh, PA, USA

Abstract

Optimization of quantum programs or circuits is a fundamental problem in quantum computing and remains a major challenge. State-of-the-art quantum circuit optimizers rely on heuristics and typically require superlinear, and even exponential, time. Recent work [8] proposed a new approach that pursues a weaker form of optimality called local optimality. Parameterized by a natural number Ω, local optimality insists that each and every Ω-segment of the circuit is optimal with respect to an external optimizer, called the *oracle*. Local optimization can be performed using only a linear number of calls to the oracle but still incurs quadratic computational overheads in addition to oracle calls. Perhaps most importantly, the algorithm is sequential.

In this paper, we present a parallel algorithm for local optimization of quantum circuits. To ensure efficiency, the algorithm operates by keeping a set of *fingers* into the circuit and maintains the invariant that a Ω-deep circuit needs to be optimized only if it contains a finger. Operating in rounds, the algorithm selects a set of fingers, optimizes in parallel the segments containing the fingers, and updates the finger set to ensure the invariant. For constant Ω, we prove that the algorithm requires $O(n \lg n)$ work and $O(r \lg n)$ span, where n is the circuit size and r is the number of rounds. We prove that the optimized circuit returned by the algorithm is locally optimal in the sense that any Ω-segment of the circuit is optimal with respect to the oracle.

To assess the algorithm's effectiveness in practice, we implement it in the Rust programming language and evaluate it by considering a range of quantum benchmarks and several state-of-the-art optimizers. The evaluation shows that the algorithm is work efficient and scales well as the number of processors (cores) increases. On our benchmarks, the algorithm outperforms existing optimizers, which are all sequential, by as much as several orders of magnitude, without degrading the quality. Our code is available on GitHub at https://github.com/UmutAcarLab/popqc.

*Currently at Amazon Web Services. This work was completed while at Carnegie Mellon University.

SPAA '25, July 28-August 1, 2025, Portland, OR, USA

ACM ISBN 979-8-4007-1258-6/25/07
https://doi.org/10.1145/3694906.3743325

CCS Concepts

• **Theory of computation** → **Parallel algorithms**; • **Hardware** → **Quantum computation**.

Keywords

Quantum circuit optimization, parallel algorithms, quantum computation, local optimization

ACM Reference Format:
Pengyu Liu, Jatin Arora, Mingkuan Xu, and Umut A. Acar. 2025. POPQC: Parallel Optimization for Quantum Circuits. In *37th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '25), July 28–August 1, 2025, Portland, OR, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3694906.3743325

1 Introduction

Quantum computing is an emerging field at the intersection of computer science, physics, and mathematics. By using quantum bits or qubits that offer exponential advantage by operating in a superposition of states, quantum computers hold the promise of major breakthroughs in numerous fields including quantum simulation[10, 16], optimization[13, 43], cryptography[49], and machine learning[12, 47]. Although quantum computer hardware has made significant advances in recent years, including superconducting[28], trapped ions[36, 37], and Rydberg atom arrays[15, 46], they still suffer from numerous limitations, including a natural tendency to decohere, or imperfections in the quantum gates, which can lead to accumulating errors or "noise". Due to the modest scale and noisy nature of quantum computers, the modern and near-term quantum era is sometimes referred to as the NISQ (Noisy Intermediate-Scale Quantum) era[44].

In this context, optimization of quantum programs or circuits has emerged as an important area of research. By optimizing the quantum circuit, circuit optimizers can reduce the number of gates and thus the number of operations in the circuit. In addition to increasing efficiency, quantum optimizers can improve circuit fidelity by reducing errors due to decoherence, and even make computations possible that are otherwise impossible (by bringing the computation within the decoherence envelope of the quantum computer).

Optimization of quantum circuits, however, is a challenging problem: global optimization is QMA-hard[39], which is believed to be beyond the reach of even quantum computers. Practical optimizers therefore typically rely on heuristics. For example, Nam et al.'s rule-based optimization[39] and Hietala's verified implementation[22]

lack quality guarantees and require quadratic time in the size of the circuit. Xu et al.'s automated search-based optimizations[58, 60] face severe scalability issues because they rely on an exponential-time search algorithm to apply optimization rules. For these reasons, state-of-the-art optimizers can struggle to optimize larger quantum circuits within a reasonable amount of time (e.g., within hours). As quantum computing continues to move from the NISQ era to the FASQ (Fault-Tolerant Application-Scale Quantum) era[45], the efficiency and quality limitations of quantum optimizers become even more important, because quantum circuits are expected to contain millions of gates. Even as the scale of the circuit optimization challenge increases, quantum circuit optimization approaches remain sequential. We don't know of any provably and practically efficient parallel algorithms for this task.

Recently, Arora et al. proposed an approach to quantum circuit optimization that is able to offer both quality guarantees and efficiency[8] with respect to a given *oracle* optimizer. Their algorithm assumes that the oracle optimizer works well for small to moderate circuits of up to a few thousand gates and is parameterized by a natural number Ω. The algorithm cuts the circuit into Ω-segments, a block containing Ω gates, optimizes each segment by using the oracle, and melds the optimized segments by optimizing along the seams of the cuts. The algorithm then compresses the circuit by moving all gates to the beginning of the circuit as much as possible, thus minimizing the gaps in the circuit, which can reduce the effectiveness of the optimizer. By repeating this cut-optimize-meld-compress process until convergence, Arora et al.[8] prove that their algorithm can guarantee that the output circuit has the local optimality property, which ensures that any Ω-wide segment in the circuit is optimal with respect to the oracle. The key to this property is the meld algorithm that "propagates" optimizations in one segment to adjacent segments. They also prove, under some reasonable assumptions, that the algorithm makes a linear number of calls to the oracle and show that the algorithm can improve performance significantly, leading to about an order of magnitude improvement without degrading quality. Although Arora et al.'s work has made significant progress by showing a path to improving the efficiency of quantum circuit optimization, their algorithm is sequential. Notably, the meld operations are inherently sequential, as they propagate optimizations from one segment to the other by sequentially optimizing segments in a sliding-window style. Furthermore, their algorithm incurs quadratic overheads to implement the cut, meld, and compress operations on circuits.

In this paper, we present an efficient parallel algorithm for local optimization. As with Arora et al.'s algorithm, our algorithm relies on an external oracle to optimize circuit segments and takes a parameter Ω. Unlike their algorithm, our algorithm avoids cut and meld operations for reasons of efficiency and parallelism. The algorithm instead keeps a set of *fingers* into the circuit and maintains the invariant that all unoptimized Ω-segments of the circuit contain a finger. Operating in rounds, the algorithm selects a set of non-interfering fingers, optimizes in parallel the segments containing the fingers, and updates the fingers to ensure the invariant. To expose parallelism, the algorithm only uses a parallel-map construct, which may be implemented in many ways, e.g., by using fork-join primitives.

To support efficient parallel access and updates to the quantum circuit, we represent the circuit with a sparse array of gates that allows gates to be removed, and pair it with an *index tree* that allows finding all non-deleted gates. Using this representation, we prove that the total work (uniprocessor time) of the algorithm is $O(n(\Omega \lg n + W))$, where n is the number of gates in the input circuit and W is the work of the oracle on 2Ω-segments. In practice, Ω is a moderate constant (in hundreds), leading to constant work W for the oracle, and $O(n \lg n)$ work for the optimizer. For span (parallel time), we show a bound $O(r(\lg n + S))$, where r is the number of rounds required by the algorithm, n is the number of gates in the input circuit, and S is the span of the oracle optimizer (equals to W for sequential oracles). This bound shows that the algorithm delivers significant parallelism in each round, with the caveat that total parallelism may be limited if the number of rounds is large. Fortunately, in practice, we observe that the number of rounds is a modest number (fewer than 100 in most of our experiments) and the algorithm exhibits significant parallelism.

To establish a quality guarantee on the optimized circuits, we prove that the optimized circuit returned by the algorithm is locally optimal, meaning any Ω-segment of the circuit is optimal with respect to the oracle. This shows that our approach can guarantee some degree of quality of the output while also ensuring efficiency and parallelism.

To assess the practicality of the algorithm, we present an implementation in the Rust language and evaluate the algorithm by using a number of quantum benchmarks. The evaluation shows that the number of rounds required by the algorithm is small relative to the circuit size, and therefore the algorithm scales well to multiple cores in practice, especially as the input circuits grow larger. The evaluation also shows that the constant factors hidden in our asymptotic analysis are small and that the algorithm is fast in practice: single-processor runs of our algorithm outperform state-of-the-art sequential optimizers. As a combined effect of the small constant factors and parallelism, our optimizer delivers orders of magnitude speedups over existing optimizers, especially as circuit sizes grow, while incurring no noticeable degradation in the quality of optimization. Notably, on a 64-core computer, our parallel optimizer can optimize circuits in seconds that existing optimizers are unable to optimize within 24 hours of compute time. This result shows that parallel optimization of quantum circuits can be effective, both in terms of performance and quality.

The specific contributions of the paper include the following:

- A parallel algorithm for optimizing quantum circuits.
- Bounds on the work and span of the algorithm.
- Proof that the algorithm returns a locally optimal circuit.
- An implementation of the algorithm in the Rust language.
- An evaluation of the algorithm by considering multiple oracle optimizers and challenging quantum benchmarks.

2 Background

In this section, we introduce basic concepts of quantum computing and establish the notation used throughout this paper.

2.1 Quantum States

A quantum bit or a ***qubit*** is the basic unit of quantum computation. The state of a qubit $|\psi\rangle$ can be represented as a linear superposition of basis states: $|\psi\rangle = \alpha|0\rangle + \beta|1\rangle$, where $\alpha, \beta \in \mathbb{C}$ satisfy the normalization condition $|\alpha|^2 + |\beta|^2 = 1$. We can also write it as a vector for mathematical manipulation: $\begin{bmatrix}\alpha & \beta\end{bmatrix}$. For an n-qubit system, the quantum state becomes a superposition of 2^n basis states: $|\psi\rangle = \sum_{i\in\{0,1\}^n} \alpha_i |i\rangle$, and can be written as a 2^n-dimensional vector.

A quantum algorithm realizes a unitary transformation U, a $2^n \times 2^n$ matrix with $UU^\dagger = I$. When applied to the initial state $|0\rangle$, the state becomes $U|0\rangle = |\psi\rangle$, which contains the result of the algorithm.

2.2 Gates and Circuits

A unitary is neither implementable on a quantum computer nor representable due to its exponential size. So, we use a quantum circuit C to describe a quantum algorithm, which can be represented as a sequence of gates. A gate only acts on a small number of qubits (usually one or two), and the other qubits are left unchanged. We use $[g]$ and $[C]$ to denote the matrix representation of a gate g and a circuit C, respectively. For a single-qubit gate g that acts on qubit i, the matrix representation is $[g] = I^{\otimes i} \otimes U \otimes I^{\otimes n-i-1}$, where n is the total number of qubits and U is a 2×2 unitary matrix. The matrix representation of the whole circuit $C : g_1, g_2, \ldots, g_k$ is the product of the matrix representations of all the gates: $[C] = [g_k][g_{k-1}] \ldots [g_1]$. From the matrix representation, we can see that a fundamental property of quantum circuits is that any subcircuit is interchangeable with any equivalent subcircuit (those implementing the same unitary transformation) because matrix multiplication is associative.

The most straightforward representation of a quantum circuit is the gate sequence representation, where a circuit is represented as a sequence of gates. Other representations also exist: for example, the layered representation, where circuits are organized into layers of independent gates. Two gates are defined as ***independent*** if they act on disjoint sets of qubits. This layered representation is valuable because it directly maps to parallel execution schedules on quantum hardware, with the number of layers or ***depth*** serving as a natural indicator of the running time of the circuit, which is a quantum analogue of span in classical computing.

2.3 Quantum Circuit Optimization

Circuit optimization transforms an input quantum circuit into an equivalent circuit (same unitary) that minimizes some cost function, with some examples being the number of gates, the circuit depth, the number of non-Clifford gates, the number of two-qubit gates, etc.

The optimization of large quantum circuits presents significant challenges due to the exponential growth in the dimensionality of the underlying unitary transformations with increasing qubit count. Indeed, global optimization of quantum circuits has been proven to be QMA-hard[25]. As a result, most existing quantum circuit optimization methods are local, meaning that they only optimize a small part of the circuit at a time.

2.4 Parallelism Model

We specify the parallel algorithms using traditional algorithmic style, written in pseudocode, and use traditional work and span (depth) analysis (e.g.,[1]). We use a parallel-map (**parmap**) primitive as the only means of exposing parallelism. This primitive maps over the elements of a collection in parallel and computes some value for each element, returning the collection of result values. When analyzing work and span, we assume that this primitive adds a logarithmic (in the number of iterations) cost to the span of each iteration. This is a conservative assumption and it is realistic for the multicore architecture, where parallel maps can be implemented with a logarithmic-depth fork-join tree. We note that for our specific algorithm and analysis, a stronger assumption such as constant-span parallel-map would not improve our bounds, because the iterations all have at least logarithmic span.

3 A Parallel Data Structure for Quantum Circuits

To optimize a quantum circuit in parallel, we propose a specialized data structure that enables efficient parallel access and manipulation of gates. Our data structure addresses a key challenge in quantum circuit optimization: as the optimization process progresses, the circuit becomes increasingly sparse due to gate removals. This sparsity requires efficient mechanisms for locating neighboring gates without scanning the entire circuit. Algorithm 1 shows the interface for our circuit data structure along with the cost bounds for each operation. To support the operations with the given bounds, we use an array to store the gates. This enables constant-time access to each gate given its index in the array. To remove a gate (e.g., during optimization), we replace it with a "tombstone", which indicates an absent gate.

Our algorithm operates by optimizing circuit ***segments*** using the oracle optimizer. A segment is defined as a contiguous sequence of non-tombstone gates between indices i and j in the circuit's gate array. We call a segment an Ω-***segment*** if it contains Ω gates. These segments form the basic units of optimization in our approach. For this to work efficiently, the algorithm must disregard the tombstones when partitioning the circuit into segments, which are then optimized independently. This can be challenging because as the optimization proceeds, the number of tombstones increases. We therefore augment the array with a binary tree data structure, which we call the ***index tree***, that helps locate the gates efficiently.

The index tree takes the form of a complete binary tree, where each leaf of the tree corresponds to a gate in the circuit and is labeled with a weight of 1 if there is a gate at that position, and 0 if there is a tombstone. We tag each internal node with a weight, which is equal to the sum of the weights of its children, indicating the number of gates in the subtree rooted at that node. Figure 1 shows an example index tree for a circuit with 5 gates. Initially, as shown in Figure 1a, the index tree has 5 leaves, all with a weight of 1, representing the state before any optimization. We realize that we can optimize the circuit by removing the two `X` gates separated by a `CNOT` gate and replacing them with tombstones. To update the index tree, we change the weights of the leaves corresponding to the removed `X` gates to 0 and update all the weights to reflect this change. Figure 1b shows the index tree after optimization.

Algorithm 1: The interface for the quantum circuit data structure. For cost bounds, n is the number of gates in the circuit.

```
Interface Circuit
    type circuit
    // Create a circuit from a gate array.
    // Cost: O(n) work and O(lg n) span
    def create(g: gate array): circuit;
    // Return the number of gates, excluding
       tombstones, before an index i.
    // Cost: O(lg n) work and span
    def before(c: circuit, i: int): int;
    // Return the i^th gate, excluding all
       tombstones.
    // Cost: O(lg n) work and span
    def get(c: circuit, i: int): gate;
    // Replace each gate at the specified index
       with the specified gate.
    // Use a tombstone to indicate a removed gate.
    // Cost: For l gates, O(l lg n) work and O(lg n)
       span
    def substitute(c: circuit, g: (int×gate) array): void;
    // Return the gate array of the circuit,
       excluding all tombstones.
    // Cost: O(n) work and O(lg n) span
    def gates(c: circuit): gate array;
```

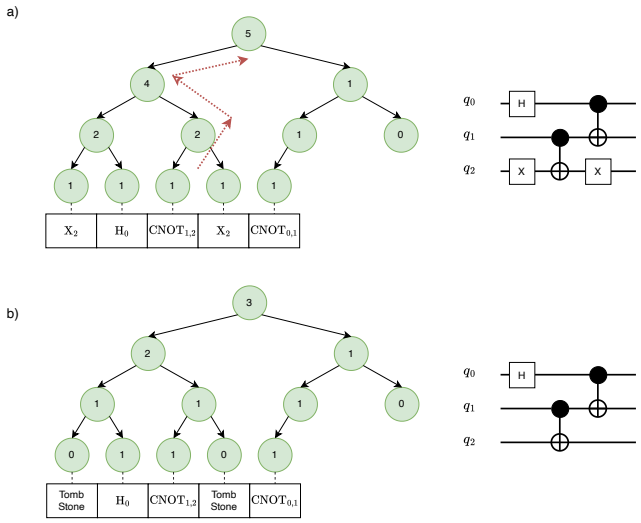

Figure 1: The index tree data structure. a) A circuit with 5 gates and a corresponding index tree. b) A circuit after optimization with 3 gates remaining and the updated index tree.

To **create** a circuit from a given array of gates, we construct the index tree and tag each internal node layer by layer; this requires linear work and logarithmic span.

Algorithm 2: POPQC Algorithm for parallel optimization of quantum circuits. The algorithm takes as input 1) an oracle optimizer denoted by "oracle", 2) a gate array $\mathcal{A}$, and 3) a segment size Ω, and outputs an optimized circuit.

```
def POPQC(oracle: gate array → gate array, A: gate
  array, Ω: int) : gate array
    // Initialize fingers
    F ← { 0, Ω, 2·Ω, ..., ⌊|A|/Ω⌋ · Ω }
    // Create circuit
    C ← Circuit.create(A)
    // Optimize in rounds
    while F ≠ ∅ do
        F ← optimizeSegments(C, F, oracle, Ω)
    end
    // Return the gates of the optimized circuit
    return Circuit.gates(C)
end
```

For the **before** operation, we start at the leaf corresponding to the specified index and walk up the index tree to the root, summing the weights of left siblings along the path. Figure 1a illustrates this process with an example that finds the number of non-tombstone gates before $\texttt{CNOT}_{1,2}$. We trace the unique path from the leaf corresponding to $\texttt{CNOT}_{1,2}$ to the root (shown as the red path). Along this path, the first node has weight 1 with no left sibling, while the second node has weight 2 with a left sibling of weight 2. Since only the second node contributes a left sibling weight, we conclude that there are two non-tombstone gates before $\texttt{CNOT}_{1,2}$.

Because the index tree is balanced, this requires $O(\lg n)$ work and span. The **get** operation takes an integer i and fetches the i^{th} gate, ignoring all tombstones. This can be implemented in logarithmic work and span by starting at the root of the index tree and tracing a path down to a leaf. The **substitute** operation takes an array of index-gate pairs, substitutes the gates at the specified indices with the specified gates, and updates the index tree. The **gates** operation returns the gate array of the circuit, excluding all tombstones.

The index tree data structure naturally generalizes to the layered representation of circuits (discussed in Section 2.2): we think of each layer as a "big" gate and perform all operations at the granularity of layers accordingly. For our results, we primarily use the gate sequence representation, but we also use the layered representation for an additional experiment on optimizing depth (Section 7.8).

4 Algorithm

We present the POPQC (Parallel Optimizer for Quantum Circuits) algorithm in this section. At a high level, the algorithm works by tracking a set of ***fingers*** to be optimized and optimizing in parallel the segments around the fingers until no further optimization is possible. The fingers are a set of indices that track the indices of the circuit near which further optimization is needed.

We say that two fingers are ***non-interfering*** if there are at least 2Ω gates between them. This property allows us to optimize the segments around these fingers in parallel without conflicts.

Algorithm 3: optimizeSegments Algorithm

```
1  def optimizeSegments(C: circuit, F: int array, oracle:
     gate array → gate array, Ω: int) : int array
       // Select non-interfering fingers to optimize
2      F_selected, F_remaining ← selectFingers(F, C, Ω)
       // Optimize segments around selected fingers
          independently
3      (F_new, C_updates) ← parmap f ∈ F_selected
4          segment ← {Circuit.get(C, Circuit.before(C, f) +
             i) for i ∈ [−Ω, Ω]}
5          optSegment ← oracle(segment)
6          if |optSegment| < |segment| then
               // Update fingers
7              F_new ← {Circuit.before(C, f) −
                 Ω, Circuit.before(C, f) + Ω}
8              optSegment ←
                 padWithTombstone(optSegment, |segment|)
               // Collect updates to circuit
9              C_updates ← {(Circuit.before(C, f) +
                 i, optSegment[i + Ω]) for i ∈ [−Ω, Ω]}
10             return (F_new, C_updates)
11         else
12             return (∅, ∅)
13         end
14     end
15     F_new ← ⋃_{f∈F_new} f
16     C_updates ← ⋃_{c∈C_updates} c
       // Apply updates to circuit
17     Circuit.substitute(C, C_updates)
       // Merge two sorted lists, removing duplicate
          items and maintaining sorted order
18     F ← mergeAndDeduplicate(F_remaining, F_new)
19     return F
20 end
```

Algorithm 4: selectFingers Algorithm

```
1  def selectFingers(F: int array, C: circuit, Ω: int) : (int
     array, int array)
2      (F_even, F_odd) ← parmap i ∈ {0, ··· , |F|}
3          groupIndex ← ⌊Circuit.before(C, F[i])/2Ω⌋
4          groupIndexPrev ← if i >
             0 then ⌊Circuit.before(C, F[i − 1])/2Ω⌋ else −1
           // Determine if f is the first finger in
              its group by comparing its group index
              and the group index of the previous
              finger
5          if groupIndex > groupIndexPrev then
6              if groupIndex mod 2 = 0 then
7                  return ({i}, ∅)
8              else
9                  return (∅, {i})
10             end
11         else
12             return (∅, ∅)
13         end
14     end
15     F_even ← ⋃_{f∈F_even} f
16     F_odd ← ⋃_{f∈F_odd} f
       // Return the larger of the two sets
17     if |F_even| > |F_odd| then
18         return F_even, F \ F_even
19     else
20         return F_odd, F \ F_odd
21     end
22 end
```

Algorithm 2 shows the pseudo-code for the POPQC algorithm. The algorithm is parameterized by a *local* oracle optimizer that can optimize a small circuit segment. We represent the oracle as a function and make no assumptions about its inner workings. In addition to the oracle, the algorithm takes as input the gate array $\mathcal{A}$ representing the circuit to optimize, and a segment size Ω.

The algorithm maintains a set of fingers ($\mathcal{F}$) and optimizes segments around them. More precisely, given a finger at some position in the circuit, the algorithm presumes that any Ω-segment ***containing*** the finger needs to be optimized. We say that a segment from index i to j contains a finger at f, if $i \le f < j$.

To optimize the input circuit, the algorithm proceeds in rounds. Each round of optimization uses the **optimizeSegments** algorithm, which starts by partitioning the fingers into two sets: $\mathcal{F}_{selected}$, which contains non-interfering fingers that will be optimized, and $\mathcal{F}_{remaining}$, which contains the remaining fingers that will not be optimized in the current round. This is done using the **selectFingers** algorithm (Algorithm 4). Then, for each finger $f \in \mathcal{F}_{selected}$, the **optimizeSegments** algorithm finds a 2Ω-segment centered at f using the index tree and optimizes the segment using the oracle. Because the selected fingers are non-interfering, the algorithm can optimize around each finger in parallel. As the algorithm optimizes the segments around each selected finger, it creates a new set of fingers $\mathcal{F}_{new}$ around the segments that are optimized by the oracle. Specifically, if the oracle does not optimize the segment (determined by comparing the size or the gate count of the optimized segment to the unoptimized one, denoted as $|segment|$), then the algorithm removes the finger. Otherwise, it adds new fingers at the boundaries of the optimized segment. As we will show in Lemma 6, the **optimizeSegments** algorithm preserves the invariant that each unoptimized Ω-segment contains a finger.

The **selectFingers** algorithm partitions the set of fingers into two sets, where the first set contains non-interfering fingers and the second set contains the remaining fingers, while trying to maximize the size of the first set. To do so, the **selectFingers** algorithm first partitions the circuit into groups of 2Ω gates each, except possibly for the last group. It then selects the first finger from each even-numbered group to construct the set $\mathcal{F}_{even}$. Similarly, the algorithm selects the first finger from each odd-numbered group to construct the set $\mathcal{F}_{odd}$. The algorithm then selects the larger of $\mathcal{F}_{odd}$ and $\mathcal{F}_{even}$, and returns the corresponding partition of the fingers

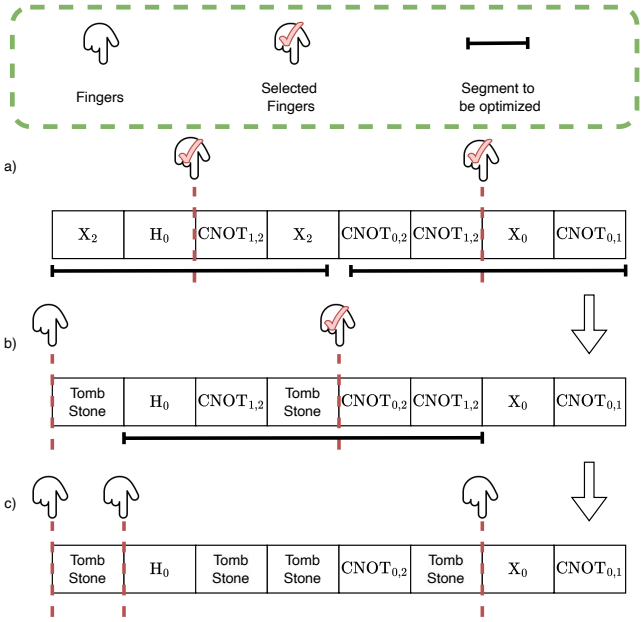

Figure 2: An illustration of a run of the POPQC algorithm. We assume $\Omega = 2$, and thus the segments being optimized consist of $2\Omega = 4$ gates.

to ensure maximum progress. Actually, it is guaranteed that this **selectFingers** algorithm selects a constant fraction of the fingers as we will show in Lemma 1.

Figure 2 illustrates an example run of the POPQC algorithm. As shown in Figure 2a, we initially have two fingers at indices 2 and 6. These two fingers are non-interfering, and thus, in the first round, the **optimizeSegments** algorithm optimizes the segments centered at these fingers in parallel. This optimization removes the two `X` gates from the left segment, while the right segment remains unaffected (no optimizations are needed). The algorithm then removes the finger within the second, unaffected segment and adds the new fingers at the boundaries of the first, optimized segment, as shown in Figure 2b. In the second round, the two fingers interfere, so only one of them is selected for optimization. Assuming the right finger is selected, the algorithm optimizes the segment containing it, which deletes two `CNOT` gates as shown in Figure 2c. This algorithm continues optimization in rounds until no more fingers remain.

5 Efficiency Analysis

In this section, we analyze the termination and efficiency of the POPQC algorithm. Our analysis relies on two key lemmas that characterize the efficiency of finger selection and the total number of oracle calls required.

Lemma 1 (Efficiency of **selectFingers**). *The **selectFingers** algorithm has work $O(|\mathcal{F}| \lg n)$ and span $O(\lg n)$. Furthermore, the returned set of non-interfering fingers contains at least a $\frac{1}{4\Omega}$ fraction of all fingers.*

Proof. The **selectFingers** algorithm processes all fingers in parallel. For each finger $f \in \mathcal{F}$, the algorithm performs two **before** operations involving the index tree, which take $O(\lg n)$ time. Therefore, the work is $O(|\mathcal{F}| \lg n)$ and the span is $O(\lg n)$.

To prove that the number of selected fingers is a constant fraction of all the fingers, note that

$$|\mathcal{F}_{even}| + |\mathcal{F}_{odd}| \geq \frac{|\mathcal{F}|}{2\Omega},$$

because each group has at most 2Ω fingers and for each group with at least one finger, one of the fingers is selected. Thus, we have $\max(|\mathcal{F}_{even}|, |\mathcal{F}_{odd}|) \geq \frac{|\mathcal{F}|}{4\Omega}$, which means at least a $\frac{1}{4\Omega}$ fraction of all fingers are selected. □

Lemma 2 (Oracle Call Efficiency). *The POPQC algorithm requires $O(n)$ oracle calls.*

Proof. We prove this lemma by defining a potential function $L = |\mathcal{F}| + 2|C|$, then showing that it is bounded by $O(n)$ and decreases by at least 1 for each oracle call.

For each oracle call, there are two cases:

a) The oracle does not optimize the segment. Only the selected finger is removed, and $|\mathcal{F}|$ decreases by 1.
b) The oracle makes changes to the circuit. This means that the size of the optimized circuit is reduced by at least 1, so $|C|$ decreases by at least 1. The original finger is removed and two new fingers are added, so $|\mathcal{F}|$ increases by 1.

Therefore, L decreases by at least 1 for each oracle call in either case. Initially, we have $L = \lceil \frac{n}{\Omega} \rceil + 2n$. Hence, the number of oracle calls is $O(n)$. □

We have the following lemma by combining the two lemmas above.

Lemma 3 (Number of Fingers). *Let $\mathcal{F}^i$ be the set of fingers in the i-th round. The total number of fingers across all rounds, $\sum_i |\mathcal{F}^i|$, is $O(\Omega n)$.*

Proof. From Lemma 1, we know that in the i-th round, the number of selected fingers satisfies $|\mathcal{F}^i_{selected}| \geq \frac{|\mathcal{F}^i|}{4\Omega}$. By Lemma 2, the total number of oracle calls across all rounds is $O(n)$. Since each selected finger results in exactly one oracle call, we have $\sum_i |\mathcal{F}^i_{selected}| = O(n)$. Therefore, $\sum_i |\mathcal{F}^i| \leq 4\Omega \cdot \sum_i |\mathcal{F}^i_{selected}| = 4\Omega \cdot O(n) = O(\Omega n)$. □

Finally, we prove the work and span bounds on the POPQC algorithm.

Theorem 4 (Work and Span of POPQC). *Suppose the upper bound of the work of an oracle call on a 2Ω-segment is W and the span is S. The POPQC algorithm has almost linear work $O(n(\Omega \lg n + W))$ in total and logarithmic span $O(r(\lg n + S))$, where r is the number of rounds (iterations of the outer loop) and n is the size of the circuit.*

Proof. We analyze the work and span of the **optimizeSegments** function by examining each component:

- **selectFingers**: As established in Lemma 1, this has work $O(|\mathcal{F}| \lg n)$ and span $O(\lg n)$.
- **Parallel optimization**: For each finger $f \in \mathcal{F}_{selected}$, we perform:

  - Segment extraction: $O(\Omega \lg n)$ work and $O(\lg n)$ span
  - Oracle call: $O(W)$ work and $O(S)$ span
  - Collecting updates: $O(\Omega)$ work and $O(\lg \Omega)$ span

  This gives a total work of $O(|\mathcal{F}_{selected}|\,(\Omega \lg n + W))$ and span $O(\lg n + S)$ for this phase. Here, we use the fact that $\Omega < n$.
- **Circuit substitution**: The **substitute** function processes input of length $O(\Omega \cdot |\mathcal{F}_{selected}|)$, requiring $O(\Omega \cdot |\mathcal{F}_{selected}| \cdot \lg n)$ work with $O(\lg n)$ span.
- **Finger merging**: The **mergeAndDeduplicate** operation has work $O(|\mathcal{F}|)$ and span $O(\lg |\mathcal{F}|)$.

For each round, the total work is $O(|\mathcal{F}| \lg n + |\mathcal{F}_{selected}|\,(\Omega \lg n + W))$ and the span per round is $O(\lg n + S)$.

By Lemma 3, the sum of $|\mathcal{F}|$ across all rounds is $O(\Omega n)$. By Lemma 2, the sum of $|\mathcal{F}_{selected}|$ across all rounds is $O(n)$. Therefore, the total work is $O(n(\Omega \lg n + W))$, and the span is $O(r(\lg n + S))$, where r is the number of rounds. While r could theoretically be as large as $\Theta(n)$ in the worst case, our empirical results in the next section demonstrate that r is typically a small constant in practice. □

In practice, we choose Ω to be a constant factor that depends on the oracle. As we increase Ω, the oracle and thus the run-time will increase, but the quality may not increase due to natural locality of circuits. The goal will be to find a setting for Ω that delivers good quality at reasonable cost.

Assuming that Ω is constant, we can conclude that the work and span of the oracle is also constant. As a corollary to Theorem 4, we therefore conclude that the work of the POPQC algorithm is $O(n \lg n)$ and the span is $O(r \lg n)$.

6 Local Optimality of POPQC

We define local optimality with respect to a given oracle function "oracle" and segment size Ω as follows: for a circuit represented by a gate array $\mathcal{A}$, we say $\mathcal{A}$ is ***locally optimal*** if for any Ω-segment $\mathcal{A}'$, applying the oracle optimizer does not reduce its size, i.e., $|\mathcal{A}'| \le |\text{oracle}(\mathcal{A}')|$.

For this definition to be meaningful, we require the oracle function to be well-behaved. An oracle is ***well-behaved*** if, after it has optimized a circuit, any segment of its output is optimal with respect to the oracle. Formally, if $\mathcal{A}' = \text{oracle}(\mathcal{A})$, then for any segment $\mathcal{A}''$ of $\mathcal{A}'$, we have $|\mathcal{A}''| \le |\text{oracle}(\mathcal{A}'')|$.

This property ensures that when we call the oracle with a 2Ω-segment, the output is locally optimal with respect to the oracle and segment size Ω.

Before proving the local optimality of POPQC, we first establish two correctness lemmas for the **selectFingers** and **optimizeSegments** algorithms.

Lemma 5 (Correctness of **selectFingers**). *The **selectFingers** algorithm returns a partition of the finger set such that the first part contains non-interfering fingers (for any two selected fingers, there are at least 2Ω gates between them).*

Proof. The algorithm first computes two groups of fingers, even-numbered ($\mathcal{F}_{even}$) and odd-numbered ($\mathcal{F}_{odd}$). Because it selects only one finger from each group, any pair of fingers in $\mathcal{F}_{even}$ are from different groups and there is always an odd group between them. Therefore, any pair of fingers in $\mathcal{F}_{even}$ is separated by at least a 2Ω-segment, and is non-interfering. Similarly, any pair of fingers in $\mathcal{F}_{odd}$ is also non-interfering. □

We now prove the correctness of **optimizeSegments** by showing that it preserves the invariant that every unoptimized Ω-segment contains a finger.

Lemma 6 (Correctness of **optimizeSegments**). *For any well-behaved oracle, if the input circuit C and the fingers $\mathcal{F}$ satisfy the invariant that every unoptimized Ω-segment contains a finger, then **optimizeSegments** returns an updated circuit C' and updated fingers $\mathcal{F}'$ that maintain this invariant: each unoptimized Ω-segment in C' contains a finger from $\mathcal{F}'$. Besides, the updated fingers $\mathcal{F}'$ are sorted.*

Proof. During the execution of **optimizeSegments**, for each selected finger $f \in \mathcal{F}_{selected}$, one of the following two cases occurs:

(1) The oracle makes no changes to the 2Ω-length segment centered at f. This implies that all Ω-length subsegments within this region are locally optimal by the well-behaved property of the oracle. The finger f can be safely removed.
(2) The oracle optimizes the 2Ω-length segment centered at f. After optimization, any new Ω-length segments fall into two categories: 1) Segments fully contained within the optimized region, which are optimal by the well-behaved property of the oracle. 2) Segments that cross the boundaries of the optimized region, which contain the boundary fingers we placed at the start and end of the optimized region. Therefore, all potentially unoptimized Ω-length segments are properly tracked with the updated fingers.

Thus, the invariant is preserved: each unoptimized Ω-segment in the updated circuit C' contains a finger from the updated set $\mathcal{F}'$.

We note that the set $\mathcal{F}_{new}$ is sorted because the selected fingers are non-interfering and $\mathcal{F}_{remaining}$ is also sorted. So a simple merge operation is enough to maintain the sorted property. □

Based on these lemmas, we now prove that the function POPQC is correct.

Theorem 7 (Local Optimality of POPQC). *The POPQC algorithm produces a locally optimal circuit with respect to a given oracle function* oracle *and segment size* Ω.

Proof. The algorithm starts by initializing the fingers $\mathcal{F} = [0, \Omega, 2\Omega, \cdots]$, placing a finger at the start of each Ω-segment. This ensures our initial invariant holds: every unoptimized Ω-segment contains a finger. We note that the choice of the initial fingers is not unique, but this specific choice minimizes the initial number of fingers. By Lemma 6, each call to **optimizeSegments** preserves the invariant that every unoptimized Ω-segment contains a finger. Furthermore, by Lemma 1, in each round, at least one finger is selected, and by Lemma 2, the total number of oracle calls is bounded by $O(n)$. The algorithm terminates in at most $O(n)$ rounds.

Since our invariant guarantees that every unoptimized Ω-segment contains a finger, the absence of fingers implies that no unoptimized Ω-segments remain in the circuit. Therefore, the final circuit is locally optimal with respect to the oracle and segment size Ω. □

7 Implementation and Evaluation

Thus far in the paper, we have presented an algorithm that guarantees $O(n \lg n)$ work and $O(r \lg n)$ span (assuming that Ω is a constant). Is this algorithm practical and does it perform well in practice? Specifically, can it take advantage of parallelism effectively to optimize large circuits? In this section, we describe an implementation and present an evaluation that answers these questions affirmatively.

7.1 Implementation

We implement the described algorithm in the Rust language, using the Rayon library which provides fork-join parallelism. [1] Our implementation matches closely with the algorithm description. In our implementation, we did not attempt to optimize manually the overheads of parallelism (e.g., via granularity control data [2, 3]) but instead left it to Rust+Rayon's adaptive loop-splitting strategy, which attempts to avoid the overheads of parallelism when the loop iterations are computationally insignificant. As we discuss, this works well in most cases, except perhaps for the smallest circuits, when parallelism overheads can be proportionally more significant.

In our implementation, we primarily use the VOQC[22] optimizer, because at the circuit sizes we consider, it is the fastest optimizer and produces the best quality outputs within a reasonable amount of time (e.g., 24 hours). (See Section 8 for a discussion of other optimizers.) Our implementation calls the oracle via a system call and is therefore relatively easy to adapt to use other optimizers as oracles. As an additional oracle, we use the Quartz[60] optimizer, which is significantly slower than VOQC but allows us to optimize other cost metrics, such as the depth of the circuit. We note that both oracle optimizers are fast for small to moderate size circuits consisting of several thousand gates, but perform poorly on larger circuits, which are necessary to realize the benefits of quantum computing.

For our experiments, we choose $\Omega = 200$ because it provides a good balance between speed and quality. The results are not particularly sensitive to the exact setting of Ω within the range from 100 to 800 and in principle, Ω can be set to a circuit-dependent value. (see extended version[33] for details).

7.2 Benchmarks

To evaluate the performance and effectiveness of our techniques, we select benchmarks from existing benchmarking suites, including PennyLane[11], Qiskit[55], and NWQBench[30]. The benchmarks include boolean satisfaction problems (`BoolSat`), the Binary Welded Tree (`BWT`), Grover's searching algorithm (`Grover`)[20], the HHL algorithm for solving linear equations (`HHL`)[21], Shor's algorithm for factoring integers (`Shor`)[49], square-root algorithm (`Sqrt`), state vector preparation (`StateVec`), and Variational Quantum Eigensolver (`VQE`)[43]. We choose the specific benchmarks because their size scales rapidly with qubit counts, and they remain challenging for state-of-the-art optimizers. Our benchmarks and oracles all use a gate set consisting of Hadamard (`H`), Pauli-X (`X`), controlled-not (`CNOT`), and rotation-Z (`RZ`), which is the gate set used by VOQC.

[1] We implemented an earlier version of the algorithm in Parallel ML [6, 7, 54] but later changed to Rust to make the implementation more accessible.

7.3 Evaluation Setup

We conduct experiments on a machine with 64 cores (AMD EPYC 7763) and 256GB RAM. To evaluate our optimizer POPQC, we first compare it to the VOQC optimizer (Section 7.4). Because VOQC is sequential, this comparison includes two advantages of our optimizer, locality and parallelism. We then separately analyze the impact of local optimality (Section 7.5) and parallelism (Section 7.6) on the overall performance.

Subsequently, we examine the work efficiency of our optimizer (Section 7.7 and extended version[33] for details), demonstrate its flexibility by integrating Quartz as an alternative oracle optimizer with a layered circuit representation (Section 7.8), and analyze the sensitivity of our method to different values of Ω (see extended version[33] for details).

7.4 Our Optimizer is Orders of Magnitude Faster than VOQC

We compare the speed and quality of our POPQC optimizer with the VOQC optimizer, which is sequential (Table 1). We note that in some cases, our baseline optimizer VOQC does not terminate in our timeout of 24 hours. We indicate non-terminating runs with an "N.A." in Table 1 and exclude their missing data from gate reduction averages. Given the large timeout, it may seem surprising that optimizers can take such a long time, but it is common indeed in quantum circuit optimizers. Such large run times are one of the motivations behind this work. Our parallel optimizer terminates on all benchmarks and does so relatively quickly, achieving more than $10^3\times$ speedup on average across all benchmarks. As the table shows, the speedups increase as the circuit sizes increase, e.g., for `Shor` with 16 qubits, the speedup is more than $10^4\times$. We also observe that the performance improvements come at little to no deterioration in optimization quality: for most benchmarks, we see a slight decrease in quality, e.g., 0.5%, while in one benchmark `HHL`, our optimizer POPQC improves quality by more than 10% over VOQC.

It's not intuitive why POPQC outperforms its base oracle VOQC in some cases. This occurs because VOQC applies optimization passes sequentially. A later pass might create opportunities for an earlier pass, which VOQC would miss in a single execution. In contrast, POPQC applies the oracle multiple times to nearby segments and can capture these opportunities, effectively behaving like "running VOQC until convergence" rather than a single pass.

The fact that we do not see significant degradation in quality may come across as surprising, because our optimizer only performs local optimization. We attribute this outcome to two factors. First, our locality requirement applies at each and every Ω-segment; it therefore is a reasonably strong guarantee in practice, especially for moderate values of Ω (e.g., $\Omega \geq 100$). Second, much like classical algorithms, quantum circuits naturally possess some degree of locality, because each segment of the circuit performs a specific function, leading to few, if any, optimizations across distant gates.

7.5 Local Optimality Unlocks Significant Efficiency

To measure the efficiency improvement due to local optimality (compared to global optimality), we compare the single-core run time of our POPQC optimizer with the VOQC optimizer. As can be

benchmark	#qubits	#gates	VOQC(1 thread)		POPQC(64 threads)		
			gate reduction	time(s)	gate reduction	time(s)	speedup
BoolSat	28	75818	83.2%	145.5	83.7%	3.6	40.2
	30	138443	83.3%	722.5	83.6%	4.6	155.8
	32	262724	83.3%	3055.0	83.4%	5.6	544.2
	34	510137	83.3%	15952.6	83.3%	7.6	2091.1
BWT	17	361603	44.7%	12165.6	44.7%	6.9	1770.0
	21	553603	51.4%	32549.3	51.4%	12.0	2712.5
	25	946801	N.A.	≥86400.0	52.9%	17.2	≥5027.9
	29	1298801	N.A.	≥86400.0	53.9%	20.0	≥4326.3
Grover	9	8968	29.4%	5.8	29.3%	1.1	5.1
	11	27136	29.9%	63.8	29.6%	1.5	42.9
	13	72646	29.7%	565.3	29.3%	2.1	264.2
	15	180497	29.5%	3911.3	28.9%	3.4	1151.2
HHL	7	5796	44.5%	0.3	58.9%	0.8	0.4
	9	68558	44.7%	151.1	59.5%	1.7	89.2
	11	680376	41.9%	33483.9	56.5%	6.0	5600.6
	13	5954308	N.A.	≥86400.0	55.9%	35.1	≥2464.3
Shor	10	8476	11.1%	5.4	10.9%	0.5	10.7
	12	31267	3.2%	106.6	3.2%	0.6	173.5
	14	136320	11.3%	2276.9	11.1%	1.6	1451.7
	16	545008	11.3%	53486.1	11.1%	4.1	13110.7
Sqrt	42	111956	42.2%	442.8	41.3%	3.4	132.0
	48	258725	42.2%	3154.9	40.0%	5.4	585.3
	54	585234	42.2%	17854.0	38.8%	9.5	1875.2
	60	1306507	N.A.	≥86400.0	37.9%	17.7	≥4879.6
StateVec	5	32147	79.6%	12.9	79.6%	1.5	8.7
	6	134632	79.2%	605.1	79.1%	2.6	230.4
	7	546035	78.9%	15272.9	78.8%	3.7	4084.1
	8	2175747	N.A.	≥86400.0	78.7%	9.6	≥9027.5
VQE	18	29800	64.4%	8.8	64.8%	0.9	9.8
	22	48448	61.6%	37.3	62.0%	1.1	33.6
	26	72600	59.0%	122.7	59.3%	1.4	85.3
	30	102768	56.5%	308.8	56.9%	1.4	228.5
average			48.94%		51.20%		≥1944.14

Table 1: Optimization quality (represented as gate reduction) and running time comparison of POPQC and VOQC on a set of quantum benchmarks. VOQC is sequential and POPQC is executed on 64 threads.

seen in Table 2, POPQC achieves more than 70× speedup on average. This shows that significant speedup is due to local optimality. We note that we do not include the output circuit sizes in this table, but they are the same as in the parallel experiments (Table 1), which show that (as discussed above) these improvements come without noticeable degradation in quality of optimization.

7.6 Our Optimizer Scales Well as Number of Cores Increases

To evaluate the scalability of our optimizer POPQC as we increase the number of cores, we run our optimizer with different numbers of cores up to the 64 cores of our experiment machine and calculate the self-speedup with respect to the single-core run. We use self-speedups, rather than calculating speedups with respect to another oracle, for two reasons. First, our oracle is the fastest (on a uniprocessor) of all other sequential oracles that we have experimented with. Second, for scalability analysis, self-speedups are more revealing as they factor out other concerns, such as algorithmic and implementation differences.

Figure 3 shows the speedups. We can discern two patterns. A majority of the benchmarks HHL, Sqrt, BWT, Shor, StateVec and

benchmark	#qubits	VOQC(1 thread) time	POPQC(1 thread) time	speedup
BoolSat	28	145.47	20.76	7.0
	30	722.50	38.25	18.9
	32	3055.01	72.32	42.2
	34	15952.59	142.03	112.3
BWT	17	12165.58	164.48	74.0
	21	32549.34	325.93	99.9
	25	≥86400.00	527.38	≥163.8
	29	≥86400.00	672.45	≥128.5
Grover	9	5.78	3.52	1.6
	11	63.76	10.97	5.8
	13	565.35	31.22	18.1
	15	3911.32	77.78	50.3
HHL	7	0.32	1.23	0.3
	9	151.12	14.38	10.5
	11	33483.88	154.47	216.8
	13	≥86400.00	1338.62	≥64.5
Shor	10	5.43	2.14	2.5
	12	106.58	7.20	14.8
	14	2276.87	34.07	66.8
	16	53486.13	135.62	394.4
Sqrt	42	442.84	53.56	8.3
	48	3154.85	125.29	25.2
	54	17854.00	280.18	63.7
	60	≥86400.00	632.01	≥136.7
StateVec	5	12.95	4.47	2.9
	6	605.14	18.90	32.0
	7	15272.95	69.92	218.4
	8	≥86400.00	272.07	≥317.6
VQE	18	8.79	3.57	2.5
	22	37.27	5.83	6.4
	26	122.73	9.01	13.6
	30	308.75	12.67	24.4
average				≥73.3

Table 2: Running time comparison (in seconds) between POPQC and VOQC, both executed on a single thread. The speedup column shows how many times faster POPQC is compared to VOQC.

Grover scale well, achieving speedups of 20 fold or more. The remaining benchmarks, VQE and BoolSat, scale less well.

To understand the scalability of VQE and BoolSat, we perform two additional experiments that measure the number of rounds and speedup with respect to input size.

Figure 4 shows the number of rounds for each benchmark; for each benchmark, the first bar is the number of rounds for the smallest instance and the second bar is the number of rounds for the largest instance. We observe that the number of rounds is between 20 and 130. We also observe that for a given benchmark, the number of rounds increases slightly compared to the input sizes, e.g., for BoolSat, the input size difference between the two bars is approximately 8-fold, but the number of rounds increases by about 40%. This is important, because our theoretical analysis shows that our algorithm has span linear in the number of rounds, which can be as large as the number of gates, but in practice, we see that the number of rounds is smaller. This is because practical quantum programs/circuits exhibit a great degree of locality much like classical programs, and consist of independent sections that do not interact deeply. Coming back to the analysis of the scalability, we attribute the relatively low scalability of BoolSat to the fact that it requires a large number of rounds.

Figure 5 shows the self-speedup with 64 cores for varying input circuit sizes. The figure shows that speedups increase with circuit size and thus larger circuits provide more opportunities for parallel optimization, where parallelism is most needed and beneficial. We explain the poor scalability of VQE partly by the fact that its circuit sizes are small (as can be seen in Figure 5). This does not explain,

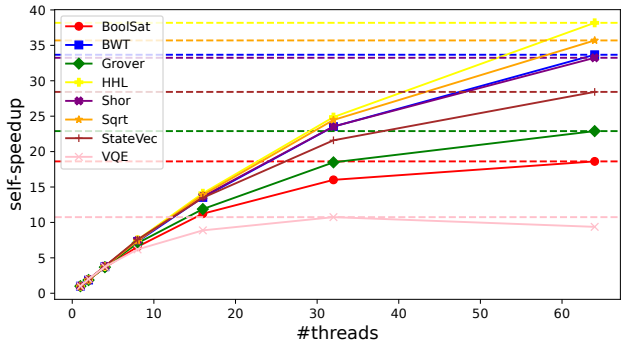

Figure 3: Self-speedup using different numbers of threads with respect to the single-thread case. The data points correspond to the largest instances in each circuit family.

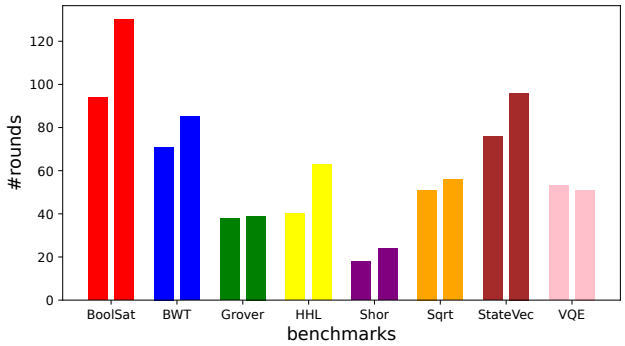

Figure 4: Number of rounds for different benchmarks. For each benchmark, the first bar represents the number of rounds for the smallest instance and the second bar represents the number of rounds for the largest instance.

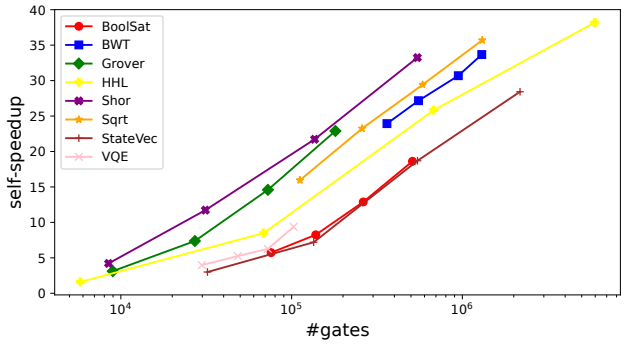

Figure 5: Self-speedup with respect to the number of gates using 64 threads. Each data point corresponds to a benchmark circuit. Log-scale on the x-axis.

however, the slight degradation in the scalability of VQE after 32 cores. Poor scalability due to difficulties in controlling the grain of parallelism is a common problem when problem sizes are small. It is possible to engineer around this problem by carefully optimizing the code on the target machine[2]. In our experiments, we have instead allowed the Rust language to manage parallelism automatically; the experiment with the VQE benchmark shows that Rust does mostly a good job but leaves some room for improvement.

7.7 Our Optimizer is Work Efficient

The term work efficiency of a parallel algorithm refers to its efficiency with respect to the work of an optimized sequential algorithm solving the same problem. Theoretically, our algorithm performs $O(n \lg n)$ work (assuming constant Ω) and is therefore reasonably work efficient, with respect to the $\Omega(n)$ lower bound that would be needed for circuit optimization. To assess practical work efficiency, we compare our optimizer to the OAC[8] optimizer, which is the fastest sequential optimizer available. OAC also ensures local optimality (as our algorithm does), making it an appropriate baseline for this comparison. In this comparison, both optimizers use the same oracle (VOQC) and use approximately the same Ω value to ensure the optimization quality differences are within 0.1% of each other.

As shown in Table 3, in a vast majority of the benchmarks our optimizer outperforms OAC, usually significantly. For example, for the HHL benchmark with 13 qubits, POPQC is approximately 5× faster than OAC. In the case of a few benchmarks, HHL, StateVec, and VQE, our optimizer performs slightly slower with the smallest circuits but outperforms OAC with all other circuits. We attribute our performance advantage over OAC to the OAC's quadratic overhead from cutting and melding circuits during optimization, which our index-tree data structure efficiently avoids. As a result of this asymptotic gap, the performance advantage of our optimizer widens over OAC as circuit size increases. We note that it is interesting that our parallel algorithm when run on a uniprocessor outperforms the best sequential optimizer. This shows that with the right algorithm, the (perceived) overheads of parallelism can be compensated to reap its benefits.

As another measure of work efficiency, we have also measured the fraction of the time our optimizer spends within the oracle, doing actual optimizations, as opposed to "administrative" work, including selecting and updating fingers. As discussed in the extended version[33], our optimizer spends over 90% of its time in the oracle, showing that very little time is spent for administrative purposes.

7.8 Our Optimizer is Flexible

To demonstrate the flexibility of our method, we evaluate it using Quartz[60] as another oracle optimizer and a layered circuit representation. Quartz is a search-based optimizer that supports customizable cost functions to guide optimization. We define a cost function that balances circuit depth and gate count, with a stronger emphasis on depth reduction: cost = 10 × depth + gates. This cost function is used both by Quartz and by our algorithm when deciding whether to accept the oracle's optimizations. For efficient cost computation, we represent circuits in layers and optimize at the layer granularity with $\Omega = 100$. The results are shown in Figure 6.

Our experiments show that using this depth-aware cost function achieves significantly better depth reduction compared to optimizing purely for gate count, with only modest increases in gate count. Two benchmarks demonstrate particularly interesting results: For Shor, while Quartz with a pure gate count objective finds no optimizations, our depth-aware approach reduces circuit depth by 20% with only a small gate count increase. For VQE, the depth-aware cost function improves both depth and gate count compared to

	n_qubits	Time(s)		Gate Reduction	
		OAC	POPQC(1-thread)	OAC	POPQC(1-thread)
BoolSat	28	51.68	44.32	83.8%	83.7%
	30	94.00	75.97	83.7%	83.6%
	32	260.95	151.01	83.5%	83.4%
	34	571.49	293.09	83.4%	83.4%
BWT	17	734.48	298.10	45.1%	45.0%
	21	1392.15	577.99	52.2%	52.2%
	25	2398.93	1083.17	55.4%	55.2%
	29	4632.67	1684.60	56.5%	56.2%
Grover	9	5.73	5.51	29.4%	29.4%
	11	23.54	20.13	30.0%	30.0%
	13	89.72	52.49	29.8%	29.8%
	15	311.93	152.74	29.6%	29.5%
HHL	7	1.39	1.55	59.0%	58.9%
	9	29.47	27.82	59.5%	59.5%
	11	785.73	316.79	56.6%	56.6%
	13	17968.66	2692.69	56.1%	56.0%
Shor	10	6.01	5.33	11.0%	11.0%
	12	25.41	14.48	3.2%	3.2%
	14	154.12	70.52	11.2%	11.2%
	16	968.38	274.75	11.2%	11.2%
Sqrt	42	156.25	83.26	42.1%	41.9%
	48	440.49	215.53	42.2%	41.8%
	54	1306.68	507.52	42.2%	41.8%
	60	3592.70	1243.65	42.2%	41.8%
StateVec	5	4.28	6.12	79.6%	79.6%
	6	33.23	31.35	79.2%	79.2%
	7	207.16	133.82	78.8%	78.8%
	8	1393.07	488.68	78.7%	78.7%
VQE	18	4.60	4.89	64.8%	64.8%
	22	8.99	8.66	61.9%	62.0%
	26	20.67	12.72	59.3%	59.3%
	30	31.97	18.78	56.9%	56.9%

Table 3: Optimization quality (represented as gate reduction) and running time comparison of POPQC and OAC. For fairness, we execute POPQC on a single thread and increase Ω to 400.

gate-only optimization. We hypothesize that by encouraging more compact gate arrangements, the depth-aware cost function helps create additional optimization opportunities that can be exploited in subsequent rounds.

8 Related Work

In this section, we provide a comprehensive overview of quantum circuit optimization techniques. Our approach is orthogonal to the techniques discussed in this section, and can be combined with them to achieve both the benefits of local optimality and the benefits of the optimization techniques.

8.1 Optimization Objectives

In addition to gate count, other optimization metrics have been proposed. In the NISQ era, circuit fidelity, which is directly related to the success probability of quantum computation, is one of the most important metrics. Notable examples of these cost functions include noise-resilient circuit optimization to maximize fidelity[38, 51]. Also, NISQ devices have various hardware constraints, and thus the optimization should be tailored to specific device architectures, like topology, gateset, and pulse. Some examples include Ref.[14, 19, 24, 31, 34, 35, 41, 48, 57].

In the fault-tolerant quantum computing era, it is widely believed that the T gate is the most expensive gate, and thus the T count becomes the most important metric in the fault-tolerant era. For

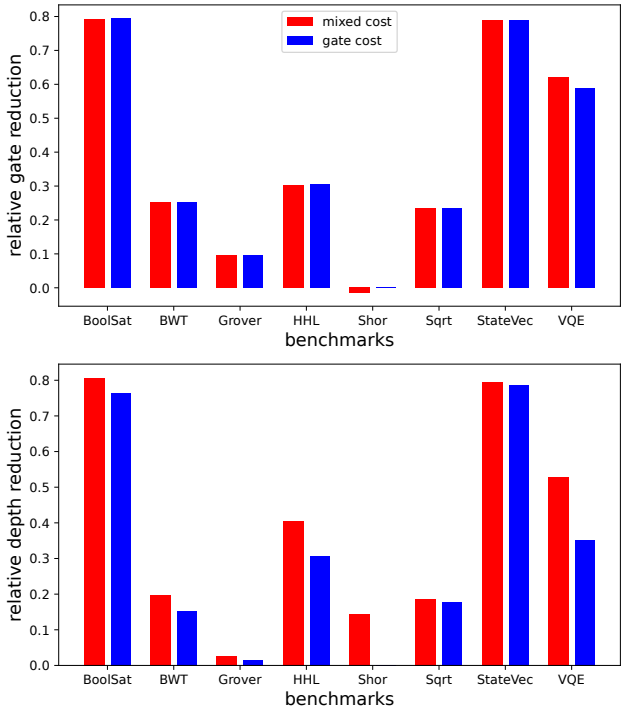

Figure 6: Gate and depth reduction using Quartz with different cost functions. Each bar corresponds to the average of the four instances in each circuit family.

small circuits, algorithms for generating asymptotically optimal T count circuits[18] exist but are not efficient for large circuits. There are also optimizers targeted at minimizing the T count with more efficient algorithms[4, 5].

8.2 Optimization Techniques

To achieve the aforementioned optimization objectives, many optimization techniques have been proposed. We classify them into rule-based, search-based, resynthesis-based, and approximated methods.

Rule-based methods. Rule-based methods encode heuristic rules into the optimizer and apply them to the circuit to optimize the circuit[9, 22, 23, 60]. The optimizer proposed by Nam et al.[39] and the following formally-verified implementation[22] are great examples of rule-based optimization. These rules take quadratic time or even cubic time in circuit size[39], thus they are not suitable for large circuits. As another example, PyZX[27] converts the circuit into ZX-diagrams and applies ZX-calculus rules to optimize the circuit which is another example of rule-based optimization.

Resynthesis-based methods. Resynthesis methods compute the unitary matrix of a small sequence of gates, then use mathematical techniques to decompose the unitary matrix into some gateset and hope the decomposed circuit is better than the original one. Examples include KAK decomposition[52] and Cartan decomposition[26]. However, these techniques can't scale to larger circuits: before decomposing, even computing the unitary takes exponential time in circuit size. QGo[56] proposes a hierarchical approach that partitions the circuit into blocks and resynthesizes each block to address the scalability issue.

Search-based methods. Rule-based optimizers often contain manually designed rules that have limited optimization capability and are not flexible enough for gateset and cost function. To address this, search-based optimizers have been proposed[29, 58, 60, 61] that automatically synthesize rules to optimize the circuit. A greedy application of the rules might lead to a local minimum. To avoid this, Quartz[60] and Queso[58] search for all possible rules that can be applied to the circuit, even if they increase the gate count, hoping to find better optimizations in future iterations. As a result, this approach has delivered excellent reductions in gate count for relatively small benchmarks but struggled to scale to large circuits. Further improvements in this direction include reinforcement learning-based methods[17, 32] that guide the optimization of quantum circuits, and combining rule-based, search-based, and resynthesis methods[59] to deliver a better trade-off between the circuit quality and the optimization time.

Approximated optimization methods. The above optimization methods focus on finding a better circuit with the exact same unitary. However, in practice, quantum computers can tolerate a small error in the unitary[40], and algorithms like quantum machine learning and quantum variational algorithms can tolerate even more. As a result, researchers have also developed approximated optimization methods. QFast[61] and QSearch[29] apply numerical optimizations to search for circuit decompositions that are close to the desired unitary. Researchers have also developed machine learning-based methods for optimizing specific quantum circuits[42, 50, 53] for variational or quantum machine learning applications.

9 Discussion and Conclusion

This paper presents a parallel algorithm, called POPQC, for circuit optimization and an implementation of the algorithm. The algorithm is reasonably work efficient and incurs only a logarithmic factor overhead over the lower bound and works well in practice. The algorithm guarantees a notion of local optimality with respect to the oracle used for optimizing small segments of the circuit. Due to its efficiency, performance, and quality guarantees, our implementation delivers significant speedups over existing optimizers without degrading optimization quality.

Acknowledgments

This research was supported by the following NSF grants CCF-1901381, CCF-2115104, CCF-2119352, CCF-2107241. We are grateful to Chameleon Cloud for providing the compute cycles needed for the experiments.

References

[1] Umut A. Acar and Guy E. Blelloch. *Algorithms: Parallel and Sequential.* 2022. http:www.algorithms-book.com.

[2] Umut A. Acar, Arthur Charguéraud, Adrien Guatto, Mike Rainey, and Filip Sieczkowski. Heartbeat scheduling: Provable efficiency for nested parallelism. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2018, pages 769–782, 2018.

[3] Umut A. Acar, Arthur Charguéraud, and Mike Rainey. Oracle scheduling: Controlling granularity in implicitly parallel languages. In *ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 499–518, 2011.

[4] Matthew Amy. Formal methods in quantum circuit design. 2019.

[5] Matthew Amy, Dmitri Maslov, and Michele Mosca. Polynomial-time t-depth optimization of clifford+ t circuits via matroid partitioning. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 33(10):1476–1489, 2014.

[6] Jatin Arora, Sam Westrick, and Umut A. Acar. Provably space efficient parallel functional programming. In *Proceedings of the 48th Annual ACM Symposium on Principles of Programming Languages (POPL)*, 2021.

[7] Jatin Arora, Sam Westrick, and Umut A. Acar. Efficient parallel functional programming with effects. *Proc. ACM Program. Lang.*, 7(PLDI):1558–1583, 2023.

[8] Jatin Arora, Mingkuan Xu, Sam Westrick, Pengyu Liu, Dantong Li, Yongshan Ding, and Umut A Acar. Local optimization of quantum circuits (extended version). *arXiv preprint arXiv:2502.19526*, 2025.

[9] Chandan Bandyopadhyay, Robert Wille, Rolf Drechsler, and Hafizur Rahaman. Post synthesis-optimization of reversible circuit using template matching. In *2020 24th International Symposium on VLSI Design and Test (VDAT)*, pages 1–4. IEEE, 2020.

[10] Paul Benioff. The computer as a physical system: A microscopic quantum mechanical hamiltonian model of computers as represented by turing machines. *Journal of Statistical Physics*, 22:563–591, 05 1980.

[11] Ville Bergholm, Josh Izaac, Maria Schuld, Christian Gogolin, Shahnawaz Ahmed, Vishnu Ajith, M Sohaib Alam, Guillermo Alonso-Linaje, B AkashNarayanan, Ali Asadi, et al. Pennylane: Automatic differentiation of hybrid quantum-classical computations. *arXiv preprint arXiv:1811.04968*, 2018.

[12] Jacob Biamonte, Peter Wittek, Nicola Pancotti, Patrick Rebentrost, Nathan Wiebe, and Seth Lloyd. Quantum machine learning. *Nature*, 549(7671):195–202, 2017.

[13] Andrew M Childs, Robin Kothari, and Rolando D Somma. Quantum algorithm for systems of linear equations with exponentially improved dependence on precision. *SIAM Journal on Computing*, 46(6):1920–1950, 2017.

[14] Marc G Davis, Ethan Smith, Ana Tudor, Koushik Sen, Irfan Siddiqi, and Costin Iancu. Towards optimal topology aware quantum circuit synthesis. In *2020 IEEE International Conference on Quantum Computing and Engineering (QCE)*, pages 223–234. IEEE, 2020.

[15] Sepehr Ebadi, Tout T Wang, Harry Levine, Alexander Keesling, Giulia Semeghini, Ahmed Omran, Dolev Bluvstein, Rhine Samajdar, Hannes Pichler, Wen Wei Ho, et al. Quantum phases of matter on a 256-atom programmable quantum simulator. *Nature*, 595(7866):227–232, 2021.

[16] Richard P Feynman. Simulating physics with computers. In *Feynman and computation*, pages 133–153. CRC Press, 2018.

[17] Thomas Fösel, Murphy Yuezhen Niu, Florian Marquardt, and Li Li. Quantum circuit optimization with deep reinforcement learning. *arXiv preprint arXiv:2103.07585*, 2021.

[18] Brett Giles and Peter Selinger. Remarks on matsumoto and amano's normal form for single-qubit clifford+ t operators. *arXiv preprint arXiv:1312.6584*, 2013.

[19] Pranav Gokhale, Ali Javadi-Abhari, Nathan Earnest, Yunong Shi, and Frederic T Chong. Optimized quantum compilation for near-term algorithms with openpulse. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 186–200. IEEE, 2020.

[20] Lov K Grover. A fast quantum mechanical algorithm for database search. *arXiv preprint quant-ph/9605043*, 1996.

[21] Aram W Harrow, Avinatan Hassidim, and Seth Lloyd. Quantum algorithm for linear systems of equations. *Physical review letters*, 103(15):150502, 2009.

[22] Kesha Hietala, Robert Rand, Shih-Han Hung, Xiaodi Wu, and Michael Hicks. A verified optimizer for quantum circuits. *Proceedings of the ACM on Programming Languages*, 5(POPL):1–29, 2021.

[23] Raban Iten, Romain Moyard, Tony Metger, David Sutter, and Stefan Woerner. Exact and practical pattern matching for quantum circuit optimization. *ACM Transactions on Quantum Computing*, 3(1):1–41, 2022.

[24] Toshinari Itoko, Rudy Raymond, Takashi Imamichi, and Atsushi Matsuo. Optimization of quantum circuit mapping using gate transformation and commutation. *Integration*, 70:43–50, 2020.

[25] Dominik Janzing, Pawel Wocjan, and Thomas Beth. Identity check is qma-complete, 2003.

[26] Navin Khaneja and Steffen J Glaser. Cartan decomposition of su (2n) and control of spin systems. *Chemical Physics*, 267(1-3):11–23, 2001.

[27] Aleks Kissinger and John van de Wetering. PyZX: Large Scale Automated Diagrammatic Reasoning. In Bob Coecke and Matthew Leifer, editors, Proceedings 16th International Conference on *Quantum Physics and Logic*, Chapman University, Orange, CA, USA., 10-14 June 2019, volume 318 of *Electronic Proceedings in Theoretical Computer Science*, pages 229–241. Open Publishing Association, 2020.

[28] Morten Kjaergaard, Mollie E Schwartz, Jochen Braumüller, Philip Krantz, Joel I-J Wang, Simon Gustavsson, and William D Oliver. Superconducting qubits: Current state of play. *Annual Review of Condensed Matter Physics*, 11(1):369–395, 2020.

[29] Costin Lancu, Marc Davis, Ethan Smith, and USDOE. Quantum search compiler (qsearch) v2.0, version v2.0, 10 2020.

[30] Ang Li, Samuel Stein, Sriram Krishnamoorthy, and James Ang. Qasmbench: A low-level qasm benchmark suite for nisq evaluation and simulation. *arXiv preprint arXiv:2005.13018*, 2021.

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

[32] Zikun Li, Jinjun Peng, Yixuan Mei, Sina Lin, Yi Wu, Oded Padon, and Zhihao Jia. Quarl: A learning-based quantum circuit optimizer. *Proceedings of the ACM on Programming Languages*, 8(OOPSLA1):555–582, 2024.

[33] Pengyu Liu, Jatin Arora, Mingkuan Xu, and Umut A. Acar. Popqc: Parallel optimization for quantum circuits (extended version). *arXiv preprint arXiv:2506.13720*, 2025.

[34] Aaron Lye, Robert Wille, and Rolf Drechsler. Determining the minimal number of swap gates for multi-dimensional nearest neighbor quantum circuits. In *The 20th Asia and South Pacific Design Automation Conference*, pages 178–183. IEEE, 2015.

[35] Abtin Molavi, Amanda Xu, Martin Diges, Lauren Pick, Swamit Tannu, and Aws Albarghouthi. Qubit mapping and routing via maxsat. In *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1078–1091. IEEE, 2022.

[36] Christopher Monroe, Wes C Campbell, L-M Duan, Z-X Gong, Alexey V Gorshkov, Paul W Hess, Rajibul Islam, Kihwan Kim, Norbert M Linke, Guido Pagano, et al. Programmable quantum simulations of spin systems with trapped ions. *Reviews of Modern Physics*, 93(2):025001, 2021.

[37] S. A. Moses, C. H. Baldwin, M. S. Allman, R. Ancona, L. Ascarrunz, C. Barnes, J. Bartolotta, B. Bjork, P. Blanchard, M. Bohn, J. G. Bohnet, N. C. Brown, N. Q. Burdick, W. C. Burton, S. L. Campbell, J. P. Campora III au2, C. Carron, J. Chambers, J. W. Chan, Y. H. Chen, A. Chernoguzov, E. Chertkov, J. Colina, J. P. Curtis, R. Daniel, M. DeCross, D. Deen, C. Delaney, J. M. Dreiling, C. T. Ertsgaard, J. Esposito, B. Estey, M. Fabrikant, C. Figgatt, C. Foltz, M. Foss-Feig, D. Francois, J. P. Gaebler, T. M. Gatterman, C. N. Gilbreth, J. Giles, E. Glynn, A. Hall, A. M. Hankin, A. Hansen, D. Hayes, B. Higashi, I. M. Hoffman, B. Horning, J. J. Hout, R. Jacobs, J. Johansen, L. Jones, J. Karcz, T. Klein, P. Lauria, P. Lee, D. Liefer, C. Lytle, S. T. Lu, D. Lucchetti, A. Malm, M. Matheny, B. Mathewson, K. Mayer, D. B. Miller, M. Mills, B. Neyenhuis, L. Nugent, S. Olson, J. Parks, G. N. Price, Z. Price, M. Pugh, A. Ransford, A. P. Reed, C. Roman, M. Rowe, C. Ryan-Anderson, S. Sanders, J. Sedlacek, P. Shevchuk, P. Siegfried, T. Skripka, B. Spaun, R. T. Sprenkle, R. P. Stutz, M. Swallows, R. I. Tobey, A. Tran, T. Tran, E. Vogt, C. Volin, J. Walker, A. M. Zolot, and J. M. Pino. A race track trapped-ion quantum processor, 2023.

[38] Prakash Murali, Jonathan M Baker, Ali Javadi-Abhari, Frederic T Chong, and Margaret Martonosi. Noise-adaptive compiler mappings for noisy intermediate-scale quantum computers. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 1015–1029. ACM, 2019.

[39] Yunseong Nam, Neil J. Ross, Yuan Su, Andrew M. Childs, and Dmitri Maslov. Automated optimization of large quantum circuits with continuous parameters. *npj Quantum Information*, 4(1), may 2018.

[40] Michael A Nielsen and Isaac Chuang. Quantum computation and quantum information, 2002.

[41] Natalia Nottingham, Michael A Perlin, Ryan White, Hannes Bernien, Frederic T Chong, and Jonathan M Baker. Decomposing and routing quantum circuits under constraints for neutral atom architectures. *arXiv preprint arXiv:2307.14996*, 2023.

[42] Mateusz Ostaszewski, Lea M Trenkwalder, Wojciech Masarczyk, Eleanor Scerri, and Vedran Dunjko. Reinforcement learning for optimization of variational quantum circuit architectures. *Advances in Neural Information Processing Systems*, 34:18182–18194, 2021.

[43] Alberto Peruzzo, Jarrod McClean, Peter Shadbolt, Man-Hong Yung, Xiao-Qi Zhou, Peter J Love, Alán Aspuru-Guzik, and Jeremy L O'brien. A variational eigenvalue solver on a photonic quantum processor. *Nature communications*, 5:4213, 2014.

[44] John Preskill. Quantum computing in the nisq era and beyond. *Quantum*, 2:79, 2018.

[45] John Preskill. Beyond nisq: The megaquop machine, 2024.

[46] Pascal Scholl, Michael Schuler, Hannah J Williams, Alexander A Eberharter, Daniel Barredo, Kai-Niklas Schymik, Vincent Lienhard, Louis-Paul Henry, Thomas C Lang, Thierry Lahaye, et al. Quantum simulation of 2d antiferromagnets with hundreds of rydberg atoms. *Nature*, 595(7866):233–238, 2021.

[47] Maria Schuld, Ilya Sinayskiy, and Francesco Petruccione. An introduction to quantum machine learning. *Contemporary Physics*, 56(2):172–185, 2015.

[48] Yunong Shi, Nelson Leung, Pranav Gokhale, Zane Rossi, David I Schuster, Henry Hoffmann, and Frederic T Chong. Optimized compilation of aggregated instructions for realistic quantum computers. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 1031–1044. ACM, 2019.

[49] Peter W Shor. Algorithms for quantum computation: Discrete logarithms and factoring. In *Proceedings 35th annual symposium on foundations of computer*

science, pages 124–134. Ieee, 1994.

[50] Sukin Sim, Jonathan Romero, Jérôme F Gonthier, and Alexander A Kunitsa. Adaptive pruning-based optimization of parameterized quantum circuits. *Quantum Science and Technology*, 6(2):025019, 2021.

[51] Swamit S Tannu and Moinuddin K Qureshi. Not all qubits are created equal: a case for variability-aware policies for nisq-era quantum computers. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 987–999. ACM, 2019.

[52] Robert R Tucci. An introduction to cartan's kak decomposition for qc programmers. *arXiv preprint quant-ph/0507171*, 2005.

[53] Hanrui Wang, Yongshan Ding, Jiaqi Gu, Yujun Lin, David Z Pan, Frederic T Chong, and Song Han. Quantumnas: Noise-adaptive search for robust quantum circuits. In *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 692–708. IEEE, 2022.

[54] Sam Westrick, Rohan Yadav, Matthew Fluet, and Umut A. Acar. Disentanglement in nested-parallel programs. In *Proceedings of the 47th Annual ACM Symposium on Principles of Programming Languages (POPL)*, 2020.

[55] Robert Wille, Rod Van Meter, and Yehuda Naveh. Ibm's qiskit tool chain: Working with and developing for real quantum computers. In *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1234–1240. IEEE, 2019.

[56] Xin-Chuan Wu, Marc Grau Davis, Frederic T Chong, and Costin Iancu. Qgo: Scalable quantum circuit optimization using automated synthesis. *arXiv preprint arXiv:2012.09835*, 2020.

[57] Xin-Chuan Wu, Dripto M Debroy, Yongshan Ding, Jonathan M Baker, Yuri Alexeev, Kenneth R Brown, and Frederic T Chong. Tilt: Achieving higher fidelity on a trapped-ion linear-tape quantum computing architecture. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 153–166. IEEE, 2021.

[58] Amanda Xu, Abtin Molavi, Lauren Pick, Swamit Tannu, and Aws Albarghouthi. Synthesizing quantum-circuit optimizers. *Proc. ACM Program. Lang.*, 7(PLDI), jun 2023.

[59] Amanda Xu, Abtin Molavi, Swamit Tannu, and Aws Albarghouthi. Optimizing quantum circuits, fast and slow. *arXiv preprint arXiv:2411.04104*, 2024.

[60] Mingkuan Xu, Zikun Li, Oded Padon, Sina Lin, Jessica Pointing, Auguste Hirth, Henry Ma, Jens Palsberg, Alex Aiken, Umut A. Acar, and Zhihao Jia. Quartz: Superoptimization of quantum circuits. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2022, page 625–640, New York, NY, USA, 2022. Association for Computing Machinery.

[61] Ed Younis, Koushik Sen, Katherine Yelick, and Costin Iancu. Qfast: Conflating search and numerical optimization for scalable quantum circuit synthesis, 2021.