Local Optimization of Quantum Circuits

Jatin Arora*
Carnegie Mellon University
jatina29@gmail.com

Mingkuan Xu
Carnegie Mellon University
mingkuan@cmu.edu

Sam Westrick
New York University
shw8119@nyu.edu

Pengyu Liu
Carnegie Mellon University
pengyuliu@cmu.edu

Dantong Li
Yale University
dantong.li@yale.edu

Yongshan Ding
Yale University
yongshan.ding@yale.edu

Umut A. Acar
Carnegie Mellon University
umut@cmu.edu

***Abstract*—Recent advances in quantum architectures and computing have motivated the development of new optimizing compilers for quantum programs or circuits. Even though steady progress has been made, existing quantum optimization techniques remain asymptotically and practically inefficient. Because many global quantum circuit optimization problems belong to the complexity class QMA (the quantum analog of NP) [51], it is not clear whether efficiency guarantees can be achieved.**

In this paper, we present an algorithm that achieves efficiency by aiming at a form of local optimality rather than global optimality. Our local-optimization algorithm cuts a circuit into subcircuits, optimizes each subcircuit with the specified oracle optimizer, and melds the subcircuits by optimizing across the cuts lazily as needed. To prove efficiency, we show that, under some assumptions, the main optimization phase of the algorithm requires a linear number of calls to the oracle optimizer. Our empirical results show that our local-optimization algorithm can outperform existing optimizers significantly, by more than an order of magnitude on average. Perhaps surprisingly, the algorithm achieves these improvements without noticeable degradation in optimization quality. These results show that local optimization can be effective for optimizing quantum circuits.

***Index Terms*—Optimization methods, partitioning algorithms, quantum circuit.**

I. Introduction

Quantum computing holds the potential to solve problems in fields such as chemistry simulation [20], [8], optimization [14], [56], cryptography [67], and machine learning [9], [64] that can be very challenging for classical computing techniques. Key to realizing the advantage of quantum computing in these and similar fields is achieving the scale of thousands of qubits and millions of quantum operations (a.k.a., gates), often with high fidelity (minimal error) [32], [23], [1]. Over the past decade, the potential of quantum computing and the challenges of scaling it have motivated much work on both hardware and software. On the hardware front, quantum computers based on superconducting circuits [39], trapped ions [48], [49], and Rydberg atom arrays [19], [63] have advanced rapidly, scaling to hundreds of qubits and achieving entanglement fidelity over 99%. On the software front, a plethora of programming languages, optimizing compilers, and run-time environments have been proposed, both in industry and in academia (e.g., [65], [27], [54], [51], [69], [10], [31], [81], [82], [79], [58], [73]).

Due to the limitations of modern quantum hardware and the need for scaling the hardware to a larger number of gates, optimization of quantum programs or circuits remain key to realizing the potential of quantum computing. The problem, therefore, has attracted significant research. Starting with the fact that global optimization of circuits is QMA hard and therefore unlikely to succeed, Nam et al. developed a set of heuristics for optimizing quantum programs or circuits [51]. Their approach takes at least quadratic time in the number of gates in the circuit, making it difficult to scale to larger circuits, consisting for example hundreds of thousands of gates. In followup work Hietala et al. [31] presented a verified implementation of Nam et al.'s approach. In more recent work Xu et al. [79] presented techniques for automatically discovering peephole optimizations (instead of human-generated heuristics) and applying them to optimize a circuit. Xu et al.'s optimization algorithm, however, requires exponential time in the number of the optimization rules and make no quality guarantees due to pruning techniques used for controlling space and time consumption. In follow-up work Xu et al. [78] and Li et al. [43] improve on Quartz's run-time. All of these optimizers can take hours to optimize moderately large circuits (Section V) and cannot make any quality guarantees.

Given this state of the art and the fact that global optimality is unlikely to be efficiently attainable due to its QMA hardness [51], we ask: ***is it possible to improve efficiency without harming quality?***

In this work, we answer this question affirmatively and thus bridge quality and efficiency. To ensure generality, we formulate local optimality ("segment optimality") in an "unopinionated" fashion in the sense that we do not make any assumptions about which optimizations may be performed by the local rewrites. Instead, we defer all optimization decisions to an abstract ***oracle*** that can be instantiated with an available optimizer as desired. Intuitively, each segment of the circuit should be optimal with respect to the oracle.

We present a local-optimization algorithm, called OAC (Optimize-and-Compact) that takes a circuit and optimizes it in rounds, each of which consists of an optimization and compaction phase. The optimization phase takes the

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

circuit and outputs a segment-optimal version of it, and the compaction phase compacts the circuit by eliminating "gaps" left by the optimization, potentially enabling new optimizations. The OAC algorithm repeats the optimization and compaction phases until convergence, where no more optimizations may be found.

To ensure efficiency, the optimization phase of OAC partitions the circuit hierarchically into smaller subcircuits, optimizes each subcircuit independently, and combines the optimized subcircuits into a locally optimal circuit. To optimize small segments, the algorithm uses any chosen oracle and does not make any restrictions on the optimizations that may be performed by the oracle. The approach can therefore be used in conjunction with many existing optimizers that support different gate sets and cost functions. By partitioning the circuit into smaller circuits, the algorithm ensures that most of the optimizations take place in the context of small circuits, which then helps reduce the total optimization cost. But optimizing subcircuits independently can miss crucial optimizations. We therefore propose a ***melding*** technique to "meld" the optimized subcircuits by optimizing over the cuts. To ensure efficiency, our melding technique starts at the cut, optimizes over the cut, and proceeds deeper into the circuit only as needed.

The correctness and efficiency properties of our cut-and-meld algorithm are far from obvious. In particular, it may appear possible that 1) the algorithm misses optimizations and 2) the cost of meld operations grows large. We show that none of these are possible and establish that the optimization algorithm guarantees segment optimality and accepts a linear time cost bound in terms of the call to oracle. For the efficiency bound, we use an "output-sensitive" analysis technique that charges costs not only to input size but also to the cost improvement, i.e., reduction in the cost (e.g., number of gates) between the input and the output. Even though the algorithm can in principle take a linear number of rounds, this appears unlikely, and we observe in practice that it requires very few rounds (e.g., less than four on average).

We evaluate the effectiveness of the OAC algorithm on a variety of quantum circuits. Our experiments show that our OAC algorithm improves efficiency, by more than one order of magnitude (on average), and closely matches or improves optimization quality. These results show that local optimality is a reasonably strong optimization criterion and our cut-and-meld algorithm can be an efficient approach to optimizing circuits. Because our approach is generic, and can be tooled to use existing optimizers, it can be used to amplify the effectiveness of existing optimizers to optimize large circuits.

Specific contributions of the paper include the following.

- An algorithm OAC for optimizing quantum circuits locally.
- Proof of correctness of OAC.
- Run-time complexity bounds and their proofs for the OAC algorithm.
- Implementation and a comprehensive empirical evaluation of OAC, demonstrating the benefits of local optimality and giving experimental evidence for the practicality of the approach.

We note that due to space restrictions, we have omitted proofs of correctness and efficiency; we provide these proofs and additional experiments in the extended version on arXiv [5].

II. Background

In this section, we provide some quantum computing background that is relevant for the paper.

a) Quantum States, Gates, and Circuits: The state of a quantum bit (or *qubit*) is represented as a linear superposition, $|\psi\rangle = \alpha\,|0\rangle + \beta\,|1\rangle$, of the single-qubit basis vectors $|0\rangle = [1\ \ 0]^{\mathrm{T}}$ and $|1\rangle = [0\ \ 1]^{T}$, for $\alpha, \beta \in \mathbb{C}$ with normalization constraint $|\alpha|^2 + |\beta|^2 = 1$. A valid transformation from one quantum state to another is described as a 2×2 complex unitary matrix, U, where $U^{\dagger}U = I$. An n-qubit quantum state is a superposition of 2^n basis vectors, $|\psi\rangle = \sum_{i\in\{0,1\}^n} \alpha_i\,|i\rangle$, and its transformation is a $2^n \times 2^n$ unitary matrix.

A *quantum circuit* is an ordered sequence of quantum logic gates selected from a predefined gate set. Each *quantum gate* represents a unitary matrix that transforms the state of one, two, or a few qubits. Given a circuit C, the *size* ($|C|$) is the total number of gates used, while the *width* (n) represents the number of qubits. The *depth* (d) is the number of circuit layers, wherein each qubit participates in at most one gate.

b) Circuit Representation: Quantum circuits can be represented with many data structures such as graphs, matrices, text, and layer diagrams. In this paper, we represent circuits with a sequence of layers, where each layer contains gates that may act at the same time step on their respective qubits. We use the layer representation to define and prove the circuit quality guaranteed by our optimization algorithm. In addition to the layer representation, our implementation uses the QASM (quantum assembly language) representation. The QASM is a standard format which orders all gates of the circuit in a way that respects the sequential dependencies between gates. It is supported by almost all quantum computing frameworks and enables our implementation to interact with off-the-shelf tools.

c) Quantum Circuit Synthesis and Optimization: The goal of *circuit synthesis* is to decompose the desired unitary transformation into a sequence of basic gates that are physically realizable within the constraints of the underlying quantum hardware architecture. Quantum circuits for the same unitary transformation can be represented in multiple ways, and their efficiency can vary when executed on real quantum devices. *Circuit optimization* aims to take a given quantum circuit as input and produce another quantum circuit that is logically equivalent but requires fewer resources or shorter execution time, such as a reduced number of gates or a reduced circuit depth. Synthesizing and optimizing large circuits are known to be challenging due to their high dimensionality. For example, as the number of qubits in a quantum circuit increases, the degree of freedom in the unitary transformation grows exponentially, leading to

higher synthesis and optimization complexity. In particular, global optimization of quantum circuits is QMA-hard [35].

III. Circuit Syntax and Semantics

We present our circuit language called Laqe (Layered Quantum Representation) which represents a quantum circuit as a sequence of layers. Figure 1 shows the abstract syntax of the language. We let the variable q denote a qubit, and G denote a gate. For simplicity, we consider only unary gates $g(q)$ and binary gates $g(q_1, q_2)$, where g is a gate name in the desired gate set. These definitions can be easily extended to support gates of any arity.

A Laqe circuit C consists of a sequence of layers $\langle L_0, \ldots, L_{n-1}\rangle$, where each layer L_i is a set of gates that are applied to qubits in parallel. The circuit is ***well formed*** if the gates of every layer act on disjoint qubits, i.e., no layer can apply multiple gates to the same qubit. As a shorthand, we write $G_1 \diamond G_2$ to denote that gates G_1 and G_2 act on disjoint qubits, i.e., $\mathsf{qubits}(G_1) \cap \mathsf{qubits}(G_2) = \emptyset$. We similarly write $L_1 \diamond L_2$ for the same condition on layers. Note that we implicitly assume well-formedness throughout the section because it is preserved by all our rewriting rules.

We define the ***length*** of a Laqe circuit as the number of layers, and the ***size*** of a circuit C, denoted $|C|$, as the total number of gates. A ***segment*** is a contiguous subsequence of layers of the circuit, and a ***k-segment*** is a segment of length k. We use the Python-style notation $C[i : j]$ to represent a segment containing layers $\langle L_i, \ldots, L_{j-1}\rangle$ from circuit C. In the case of overflow (where either $i < 0$ or $j > \mathsf{length}(C)$), we define $C[i : j] = C[\max(0, i) : \min(j, \mathsf{length}(C))]$.

Two circuits C and C' can be concatenated together as $C; C'$, creating a circuit containing the layers of C followed by layers of C'. Formally, if $C = \langle L_0 \ldots L_{n-1}\rangle$ and $C' = \langle L'_0 \ldots L'_{m-1}\rangle$, then $C; C' = \langle L_0 \ldots L_{n-1}, L'_0 \ldots L'_{m-1}\rangle$.

IV. Local Optimization Algorithm

Given an oracle to optimize small circuits, it might seem straightforward to implement an algorithm to scale it to larger circuits by simply applying the oracle on all small segments until it converges. However, such an algorithm is not efficient, because searching for a segment to optimize requires linear time in the size of the circuit (both in worst and the average case), yielding a quadratic bound for optimization. For improved efficiency, it is crucial to reduce the search time needed to find a segment that would benefit from optimization.

Our algorithm, called OAC, controls search time by using a circuit cutting-and-melding technique. The algorithm cuts the circuit hierarchically into smaller subcircuits, optimizes each subcircuit independently. The hierarchical cutting naturally reduces the search time for the optimizations by ensuring that most of the optimizations take place in the context of small circuits. Because the algorithm optimizes each subcircuit independently, it can miss crucial optimizations. To compensate for this, the algorithm melds the optimized subcircuits and optimizes further the melded subcircuits starting with the ***seam***, or the boundary between the two subcircuits. The meld operation guarantees local optimality and does so efficiently by first optimizing the seam and further optimizing into each subcircuit only if necessary. By melding locally optimal subcircuits, the algorithm can guarantee that the subcircuits or any of their "untouched" portions (what it means to be "untouched" is relatively complex) remain optimal. We make this intuitive explanation precise by proving that the algorithm yields a locally optimal circuit. We note that circuit cutting techniques have been studied for the purposes of simulating quantum circuits on classical hardware [55], [70], [12]. We are not aware of prior work on circuit melding techniques that can lazily optimize across circuit cuts.

A. The algorithm

Figure 2 shows the pseudocode for our algorithm. The algorithm (OAC) organizes the computation into rounds, where each round corresponds to a recursive invocation of OAC. A round consists of a compaction phase (via the function compact) and a segment-optimization phase, via the function segopt. The rounds repeat until convergence, i.e., until no more optimization is possible (at which point the final circuit is guaranteed to be *locally optimal*). As the terminology suggests, the segment optimization phase always yields a segment-optimal circuit, where each and every segment is optimal (as defined by our rewriting semantics). Compaction rounds ensure that the algorithm does not miss optimization opportunities that arise due to compaction. We also present a relaxed version of our algorithm that stops early when a user-specified ***convergence threshold*** $0 \leq \epsilon \leq 1$, is reached.

Function segopt. The function segopt takes a circuit C and produces a segment optimal output. A circuit is ***segment-optimal*** if each and every Ω-segment of the circuit is optimal for the given **oracle** and **cost** function. To achieve this, it uses a divide-and-conquer strategy to cut the circuit hierarchically into smaller and smaller circuits: it splits the circuit into the subcircuits C_1 and C_2, optimizes each recursively, and then calls meld on the resulting circuits to join them back together without losing segment optimality. This recursive splitting continues until the circuit has been partitioned into sufficiently small segments, specifically where each piece is at most 2Ω in length. For such small segments, the function directly uses the oracle and obtains optimal segments.

Function meld. Figure 2 (right) presents the pseudocode of the meld function. The function takes segment-optimal inputs C_1 and C_2 and returns a segment-optimal circuit that is functionally equivalent to the concatenation of inputs.

Given that the inputs C_1 and C_2 are segment optimal, all Ω-segments that lie completely within C_1 or C_2 are already optimal. Therefore, the function only considers and optimizes "boundary segments" which have some layers from circuit C_1 and other layers from circuit C_2.

To optimize segments at the boundary, the function creates a "super segment", named W, by concatenating the last Ω layers of circuit C_1 with the first Ω layers of circuit C_2. The

$$
\begin{array}{lll}
\textit{Qubit} & q & \\
\textit{Gate Name} & g & \\
\textit{Gate} & G & ::= \quad g\ (q) \mid g\ (q_1, q_2) \\
\textit{Layer} & L & ::= \quad \{G_0, G_1, \ldots, G_{t-1}\} \\
\textit{Circuit} & C & ::= \quad \langle L_0, L_1, \ldots L_{n-1} \rangle
\end{array}
$$

$$
\mathsf{qubits}(G) \triangleq \begin{cases} \{q\}, & G = g(q) \\ \{q_1, q_2\}, & G = g(q_1, q_2) \end{cases}
$$

$$
\mathsf{qubits}(L) \triangleq \bigcup_{G \in L} \mathsf{qubits}(G)
$$

$$
G_1 \diamond G_2 \Leftrightarrow \mathsf{qubits}(G_1) \cap \mathsf{qubits}(G_2) = \emptyset
$$

$$
L_1 \diamond L_2 \Leftrightarrow \mathsf{qubits}(L_1) \cap \mathsf{qubits}(L_2) = \emptyset
$$

$$
C \text{ well-formed} \Leftrightarrow \forall L \in C.\ \forall G_1, G_2 \in L.\ G_1 \diamond G_2
$$

Fig. 1: Syntax of LAQE and well-formed circuits.

```
1  Ω: int
2  oracle: circuit → circuit
3  cost: circuit → int
4  compact: circuit → circuit
5
6  def OAC(C):
7    C' = segopt(compact(C))
8    if C' = C:
9      return C
10   else:
11     return OAC(C')
```

```
12 def segopt(C):
13   d = length(C)
14   if d ≤ 2Ω:
15     C' = oracle(C)
16     if cost(C') < cost(C):
17       return C'
18     else:
19       return C
20   else:
21     m = ⌊d/2⌋
22     C1 = C[0 : m]
23     C2 = C[m : d]
24     C1' = segopt(C1)
25     C2' = segopt(C2)
26     return meld(C1', C2')
```

```
27 def meld(C1, C2):
28   d1 = length(C1)
29   d2 = length(C2)
30   W = C1[d1 − Ω : d1] + C2[0 : Ω]
31   W' = oracle(W)
32   if cost(W') = cost(W):
33     return (C1; C2)  // concatenate
34   else:
35     M = meld(C1[0 : d1 − Ω], W')
36     return meld(M, C2[Ω : d2])
```

Fig. 2: Algorithm OAC produces locally optimal circuits with respect to a given **oracle**, **cost**, and segment length Ω. To achieve local optimality, OAC only uses the oracle on small segments of length 2Ω. The algorithm repeatedly optimizes and compacts the circuit until convergence. The function segopt() implements our optimization algorithm and uses meld() to efficiently produce segment optimal circuits.

function denotes this concatenation as $C_1[d_1-\Omega : d_1]+C_2[0 : \Omega]$ (see line 30). The meld function calls the oracle on W and retrieves the W', which is guaranteed to be segment-optimal because it is returned by the oracle. The meld function then considers the costs of W and W'.

If the costs of W and W' are identical, then W is already segment optimal. Consequently, all Ω-segments at the boundary of C_1 and C_2 are also optimal. The key point is that the "super segment" W encompasses all possible Ω-segments at the boundary of C_1 and C_2. To see this, let's choose an Ω-segment at boundary, which takes the last $i > 0$ layers of circuit C_1 and the first $j > 0$ layers from of circuit C_2; we can write this as $C_1[d_1 - i : d_1] + C_2[0 : j]$, where d_1 is the number of layers in C_1. Given that this is an Ω−segment and has $i + j$ layers, we get that $i + j = \Omega$ and $i < \Omega$ and $j < \Omega$. Now observe that our chosen segment $C_1[d_1 - i : d_1] + C_2[0 : j]$ is contained within the super segment $W = C_1[d_1 - \Omega : d_1] + C_2[0 : \Omega]$ (line 30), because $i < \Omega$ and $j < \Omega$. Given that W is segment optimal, our chosen segment is also optimal (relative to the oracle).

Returning to the meld algorithm, consider the case where the segment W' improves upon the segment W. In this case, meld incorporates W' into the circuit and propagates this change to the neighboring layers. To do this, meld works with three segment optimal circuits: circuit $C_1[0 : d_1 - \Omega]$, which contains the first $d_1-\Omega$ layers of circuit C_1, is segment optimal because C_1 is segment optimal; the circuit W' is segment optimal because it was returned by the oracle; and the circuit $C_2[\Omega : d_2]$, which contains the last $d_2 - \Omega$ layers of circuit C_2, is segment optimal because C_2 is segment optimal. Thus, we propagate the changes of window W', by recursively melding these segment optimal circuits.

In Figure 2, the function meld first melds the remaining layers of circuit C_1 with the segment W', obtaining circuit M (see line 35), and then melds the circuit M with the remaining layers of C_2.

B. Meld Example

We present an example of how meld joins two circuits by optimizing the "seam", and does so "lazily", as needed.

Figure 3 shows a three-step meld operation that identifies optimizations at the boundary of two circuits. All the circuits in the figure are expressed using the H gate (Hadamard gate), the R_Z gate (rotation around Z), and the two-qubit CNOT gate (Controlled Not gate), which is represented using a dot and an XOR symbol. We first provide background on the optimizations used by meld and label them "Optimization 1" and "Optimization 2". Optimization 1 shows that when a CNOT gate is surrounded by four H gates, all of these gates can replaced by a single CNOT gate whose qubits are flipped. Optimization 2 shows that when two CNOT gates are separated by a R_Z gate as shown, they may be removed.

The steps in the figure describe a meld operation on the two circuits separated by a dashed line, which represents their seam. To join the circuits, the meld operation proceeds outwards in both directions and optimizes the boundary segment, represented as a green box with solid lines. The

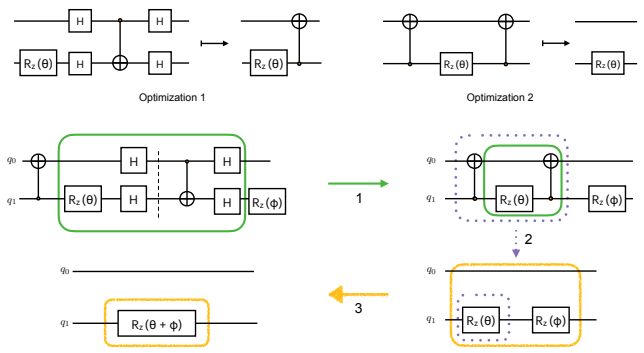

Fig. 3: The figure shows a three-step meld operation and illustrates how it propagates optimizations at the boundary of two optimal circuits. The figure shows specific optimizations "1" and "2" at the top, and the optimization steps of meld towards the bottom. Before the first step, two individual circuit segments that are optimal are separated by a dashed line. Each meld step considers a segment, represented by a box with solid/dotted/shaded lines, and applies an optimization to it, reducing the gate count. The first step focuses on the boundary segment within the solid green box, overlapping with both circuits, and applies "Optimization 1". This step introduces a flipped CNOT gate, which interacts with a neighboring CNOT gate and triggers "Optimization 2" in the purple dotted box. The third step merges two neighboring rotation gates in the yellow shaded box.

meld applies Optimization 1 to the green segment and this introduces a flipped CNOT gate.

The meld propagates this change in step 2, by considering a new segment, which includes a neighboring layer. We represent this segment by a purple box with dotted lines, and it contains two CNOT gates, one of which was introduced by the first optimization. The meld then applies Optimization 2, removing the CNOT gates and bringing the two rotation gates next to each other. Note that Optimization 2 became possible only because of Optimization 1, which introduced the flipped the CNOT gate. In the final step, the meld considers the segment represented by a yellow box with shaded lines and performs a third optimization, merging the two rotation gates. Overall, this sequence of optimizations, at the boundary of two circuits, reduces the gate count by 7.

C. Correctness and Efficiency

Because our algorithm cuts the input circuit into subcircuits and optimizes them independently, it is far from obvious that its output is segment optimal. We prove that this is indeed the case with Theorem 2 below. The reason for this is the meld operation that is able to optimize circuit cuts. We also prove, with Corollary 4, that even though meld behaves dynamically and its cost varies from one circuit to another, it remains efficient, in the sense that the number of calls to oracle is always linear in the size of the circuit plus the improvement in the cost. The proofs are on arXiv [5].

Lemma 1 (Segment optimality of meld)**.** *Given any additive* **cost** *function and any segment optimal circuits C_1 and C_2, the result of meld(C_1, C_2) is a segment optimal circuit C and* $\mathbf{cost}(C) \leq \mathbf{cost}(C_1) + \mathbf{cost}(C_2)$.

Theorem 2 (Segment optimality algorithm)**.** *For any circuit C, the function* $\mathsf{segopt}(C)$ *outputs a segment optimal circuit.*

Theorem 3 (Efficiency of segment optimization)**.** *The function* $\mathsf{segopt}(C)$ *calls the oracle at most* $\mathsf{length}(C) + 2\Delta$ *times on segments of length at most 2Ω, where Δ is the improvement in the cost of the output.*

Corollary 4 (Linear calls to the oracle)**.** *When optimizing for gate count, our* $\mathsf{segopt}(C)$ *makes a linear, $O(\mathsf{length}(C)+|C|)$, number of calls to the oracle.*

We experimentally validate this corollary in Section V-E, where we study the number of oracle calls made by our algorithm for many circuits.

D. OAC*: controlling convergence

We observe that, except for the very last round, each round of our OAC algorithm improves the cost of the circuit. This raises a practical question: how does the improvement in cost vary across rounds? For the vast majority of our evaluation, we observed that nearly all optimization (> 99%) occur in the first round itself (see Section V-F); the subsequent rounds have a small impact on the quality. Based on this observation, we propose OAC* which uses a convergence threshold $0 \leq \epsilon \leq 1$ to provide control over how quickly the algorithm converges.

```
def OAC*(f, C):
  C' = segopt(compact(C))
  if 1 − cost(C')/cost(C) ≤ f:
    return C'
  else:
    return OAC*(f, C')
```

Fig. 4: OAC* terminates when the cost improvement ratio falls below the *convergence threshold*, f.

The OAC* algorithm, in Figure 4, terminates as soon as the cost is reduced by a smaller fraction than f. For example, if we measure cost as the number of gates and $\epsilon = 0.01$, then OAC* will terminate as soon as an optimization round removes fewer than 1% of the remaining gates. Note that OAC* gives two guarantees: 1) the output circuit is still segment optimal, and 2) the fractional cost improvement in the last round is less than ϵ.

V. Evaluation

We perform an empirical evaluation of the effectiveness of the local optimality approach for quantum circuits. Specifically, we consider the following research questions (RQ).

RQ I: Is the OAC algorithm efficient and scalable?

RQ II: Is there any degradation in optimization quality? How important is the meld operation?

RQ III: Is empirical performance consistent with the asymptotic bound?

RQ IV: What is the impact of segment size Ω and compaction on the local optimality and performance of OAC algorithm?

To answer these questions, we implement the OAC algorithm and integrate it with VOQC [31] as an oracle. We chose VOQC because it is overall the best optimizer both in terms of efficiency and quality of optimization among all the optimizers that we have experimented with. We evaluate the effectiveness of local optimality and OAC, on the Nam gate set [51] and compare it with three state-of-the-art optimizers Quartz, Queso, and VOQC [79], [78], [31].

In brief, these experiments show that our cut-and-meld algorithm delivers fast optimization while closely matching (within 0.1%) or improving the optimization quality for all circuits. These results show that the local optimality approach can be effective in optimizing large quantum circuits, and can help scale existing optimizers.

In the extended version on arXiv [5], we present results for the Clifford+T gate set by using the FeynOpt [2], as an oracle. We omitted these from the main body of the paper due to space reasons but note that they are similar to the results presented here in terms of efficiency and quality.

A. Implementation

To evaluate whether the OAC algorithm (Section IV) is practically feasible, we implemented OAC in SML (Standard ML), which comes with an optimizing compiler, MLton, that can generate fast executables. Our implementation closely follows the algorithm description. It uses the layered circuit representation and represents circuits as an array of arrays, where each array denotes a "layer" of the circuit. The implementation splits and joins circuit segments by splitting and joining the corresponding arrays, performing rounds of optimization and compaction.

As described in Section IV-D, our implementation allows user control over convergence through a specified convergence ratio ϵ, where $0 \leq \epsilon \leq 1$. In the evaluation, we choose $\epsilon = 0.01$, and analyze this choice in Section V-F1. Note that regardless of ϵ, the implementation always guarantees that output is segment optimal.

Our implementation is parametric in the oracle being used. To allow calls to existing optimizers, we use MLton's foreign function interface, which supports cross-language calls to C++. Specifically, to use an existing optimizer as an oracle, we only need to provide a C++ wrapper that takes a circuit in QASM format as input and returns an optimized circuit as output.

B. Benchmarks and gate set

To evaluate our OAC algorithm, we consider a benchmark suite of eight circuit families that include both near-term and future fault-tolerant quantum algorithms. For each family, we select circuits with different sizes by changing the number of qubits. Our benchmark suite includes advanced quantum algorithm such as Grover's algorithm for unstructured search [28], the HHL algorithm for solving linear systems of equations [29], Shor's algorithm for factoring large integers [67], and the Binary Welded Tree (bwt) quantum walk algorithm [13]. In addition, our benchmarks include near-term algorithms like Variational Quantum Eigensolver (vqe) [56] and reversible arithmetic algorithms [4], [51] such as boolean satisfaction problems (boolsat) and square-root algorithm (sqrt).

The benchmark suite is written in the Nam gate set [51], which consists of the Hadamard (H), Pauli-X (X), controlled-NOT (CNOT), and Z-rotation (R_Z) gates [51]. We preprocess all our benchmarks with the Quartz preprocessor, which merges rotation gates [79].

C. RQ I: Efficiency of OAC

To evaluate the efficiency of OAC, we use our OAC implementation with VOQC as the oracle on segments of size $\Omega = 40$ and compare it to optimizers Quartz, Queso, and VOQC. The approach works for many different settings of Ω and we analyze the impact of Ω in Section V-F2 in detail. We give each optimizer a 12-hour cut-off time (excluding time for parsing and printing), to allow completion of the experiments within a reasonable amount of time. Throughout, we omit circuit-parsing time for timings of VOQC, whose parser appears to scale superlinearly and can take significant time (sometimes more than the optimization itself). This approach is consistent with prior work on VOQC, which also excludes parse time. When we use VOQC as an oracle of OAC, however, we do include the parse time. This makes the comparison somewhat unfair for OAC.

We evaluate the running times of these optimizers on benchmarks from the Nam gate set with sizes ranging from thousands to hundreds of thousands of gates.

Time Performance. Figure 5 show the time for our OAC implementation (with VOQC oracle) compared against Quartz, Queso, and VOQC. The figure includes eight families of circuits, where horizontal lines separate families and circuits within each family arranged body increasing qubit and gate counts. The optimizers Quartz and Queso use the maximum allotted time of 12 hours in all circuits, because they explore a very large search space of all optimizations. In a few cases, Queso throws an error or runs out of memory (denoted "OOM"). The VOQC optimizer and our OAC optimizer terminate much faster. Specifically, OAC optimizes all circuits between 0.2 seconds and 3 hours depending on the size, and VOQC finishes for all but six benchmarks within 12 hours. In the figure, we highlight in bold the fastest optimizer(s) for each circuit.

Comparing between VOQC and our OAC, we observe the following:

- **Performance:** OAC is the fastest across the board except for `vqe` and except perhaps for the smallest circuits in some families.
- **Scalability:** the gap between OAC and VOQC increases as the circuit size increases, with OAC performing as much as 100× faster in some cases.
- **Overall:** OAC is over an order of magnitude faster than VOQC on average.

In the case of the `vqe` family, VOQC is consistently faster, but as we discuss next, this comes at the cost of poorer

Benchmark	Qubits	Input Size	Time (s)				OAC speedup
			Quartz	Queso	VOQC	OAC	
boolsat	28	75670	12h	12h	68.6	**45.2**	1.52
	30	138293	12h	12h	307.3	**98.7**	3.11
	32	262548	12h	12h	1266.2	**213.6**	5.93
	34	509907	12h	12h	6151.0	**462.8**	13.29
bwt	17	262514	12h	12h	8303.1	**524.9**	15.82
	21	402022	12h	12h	23236.8	**1062.1**	21.88
	25	687356	12h	12h	>12h	**2341.7**	> 18.45
	29	941438	12h	12h	>12h	**3982.6**	> 10.85
grover	9	8968	12h	12h	9.3	**4.8**	1.94
	11	27136	12h	12h	106.5	**20.8**	5.12
	13	72646	12h	12h	815.7	**68.1**	11.97
	15	180497	12h	12h	5743.2	**223.9**	25.65
hhl	7	5319	12h	12h	**0.3**	0.9	0.27
	9	63392	12h	12h	74.1	**22.9**	3.24
	11	629247	12h	12h	14868.8	**434.3**	34.24
	13	5522186	12h	Parsing Error	>12h	**8243.1**	> 5.24
shor	10	8476	12h	OOM	8.8	**5.2**	1.70
	12	34084	12h	12h	179.9	**26.3**	6.84
	14	136320	12h	12h	3638.4	**126.0**	28.88
	16	545008	12h	12h	70475.2	**648.9**	108.60
sqrt	42	79574	12h	12h	**30.0**	81.4	0.37
	48	186101	12h	12h	**191.2**	268.0	0.71
	54	424994	12h	12h	3946.5	**679.8**	5.81
	60	895253	12h	12h	>12h	**1653.5**	> 26.13
statevec	5	31000	12h	OOM	**1.6**	4.3	0.38
	6	129827	12h	12h	45.9	**27.1**	1.70
	7	526541	12h	12h	1812.2	**164.7**	11.00
	8	2175747	12h	12h	>12h	**1345.1**	> 32.12
vqe	12	11022	12h	12h	**0.2**	1.2	0.13
	16	22374	12h	12h	**0.6**	3.4	0.18
	20	38462	12h	12h	**2.0**	7.0	0.29
	24	59798	12h	12h	**5.4**	13.4	0.41
avg							> 12.62

Fig. 5: The figure shows the running time in seconds of the four optimizers, using gate count as the cost metric. The column "OAC Speedup" is the speed of our OAC with respect to VOQC, calculated as VOQC time divided by OAC time. These measurements show that our optimizer OAC can be significantly faster, especially for large circuits (more than one order of magnitude on average). These time improvements come without any loss in optimization quality. These results suggest that local optimality approach to optimization of quantum circuits can be effective in practice.

optimization quality. For the small circuits of families `hhl`, `statevec`, and `sqrt`, our optimizer is slower than VOQC. This is due to the overheads that our implementation incurs for (1) splitting and joining circuits, (2) serialization/deserialization of input/output circuits for each oracle call, and (3) various system-level calls needed to support calls to an external oracle. For example, for the 7-qubit `hhl` benchmark and the 42-qubit `sqrt` benchmark, we have measured that at least 30% of the running time is spent parsing and serializing/deserializing circuits.

D. RQ II: Optimization quality and importance of meld

We used VOQC as the oracle because it scales better and achieves a better optimization quality than all other previous works on all circuits presented in Figure 5. Figure 6 shows the output quality (measured by gate count) of VOQC and OAC for eight families of circuits.

For almost all families, we observe that the output quality of OAC matches that of VOQC within 0.1% or improves it, sometimes significantly. This shows that OAC scales better without sacrificing optimization quality. Specifically, for the `vqe` family, OAC optimizes better than VOQC. For example, on the 24-qubit `vqe` circuit, OAC improves the gate count by 60.6%, and VOQC improves the gate count by 54.9%.

To better understand what contributes to the optimization quality, we perform an ablation experiment of the meld operation. Specifically, we implement an ablating version of our OAC, called OACMinus, that simply concatenates the optimized subcircuits instead of the meld operation.

Figure 6 shows that the meld operation contributes an improvement of 2.1% in absolute and 4.4% or relative terms. Even though the percentage degradation due to ablation may seem modest, it is significant, because

- each and every gate has a significant runtime and fidelity cost on modern and near-term quantum computers, and
- the optimization quality does not decrease uniformly: in some circuit families, such as `vqe`, the ablating version is more than 5% worse in absolute and 12% in relative terms.

Due to these differences the ablating version (without the

Benchmark	Qubits	Input Size	Optimizer		
			VOQC	OAC	OACMinus
boolsat	28	75670	-83.2%	**-83.7%**	-83.0%
	30	138293	-83.3%	**-83.7%**	-83.1%
	32	262548	-83.3%	**-83.5%**	-83.1%
	34	509907	-83.3%	**-83.4%**	-83.1%
bwt	17	262514	-30.0%	**-31.1%**	-28.3%
	21	402022	-38.4%	**-40.0%**	-36.3%
	25	687356	N.A.	**-43.8%**	-40.0%
	29	941438	N.A.	**-44.5%**	-41.0%
grover	9	8968	**-29.4%**	**-29.4%**	-26.1%
	11	27136	-29.9%	**-30.0%**	-26.3%
	13	72646	**-29.7%**	**-29.7%**	-26.0%
	15	180497	**-29.5%**	**-29.5%**	-26.0%
hhl	7	5319	**-55.4%**	-55.3%	-54.4%
	9	63392	-56.3%	**-56.5%**	-55.6%
	11	629247	**-53.7%**	**-53.7%**	-53.1%
	13	5522186	N.A.	**-52.6%**	-52.1%
shor	10	8476	**-11.1%**	-11.0%	-10.3%
	12	34084	**-11.2%**	**-11.2%**	-10.7%
	14	136320	**-11.3%**	-11.2%	-10.8%
	16	545008	**-11.3%**	**-11.3%**	-10.9%
sqrt	42	79574	**-33.0%**	**-33.0%**	-31.5%
	48	186101	**-32.7%**	-32.6%	-31.2%
	54	424994	**-32.4%**	-32.3%	-30.9%
	60	895253	N.A.	**-34.3%**	-32.9%
statevec	5	31000	-78.8%	**-78.9%**	-78.1%
	6	129827	**-78.4%**	**-78.4%**	-77.9%
	7	526541	**-78.1%**	**-78.1%**	-77.8%
	8	2175747	N.A.	**-78.7%**	-78.6%
vqe	12	11022	-63.0%	**-69.5%**	-62.7%
	16	22374	-60.1%	**-66.3%**	-59.7%
	20	38462	-57.4%	**-63.4%**	-57.2%
	24	59798	-54.9%	**-60.6%**	-54.8%
avg			-48.1%	**-49.4%**	-47.3%

Fig. 6: Optimization quality of VOQC, OAC, and OACMinus, measured by gate count reduction percentage. OACMinus is a modified version of OAC that uses simply concatenation instead of the meld algorithm. The measurements show that with the meld algorithm, our OAC optimizer reduces gate counts better than using VOQC alone. The relative improvement of OAC over OACMinus is 4.4%.

meld operation) performs consistently worse than VOQC, whereas the non-ablating version (OAC) outperforms VOQC in all but 5 of the 32 circuits. In the 5 circuits, where VOQC outperforms our local-optimization, the difference is 0.1%.

E. RQ III: Empirical versus asymptotic performance

In Section IV, we established bounds on the number of oracle calls performed by our OAC algorithm. In this section, we check that our implementation is consistent with these bounds by analyzing the number of oracle calls with respect to the circuit size. Figure 7 plots the number of oracle calls made by our OAC optimizer for a subset of circuit families (the other circuit families behave similarly). The Y-axis represents the number of oracle calls and the X-axis represents the input circuit size. The plot shows that the number of oracle calls increases linearly with circuit size, for all circuit families.

We note that it would be more desirable to establish that the total run-time, rather than the number of oracle calls, is linear, but this is not the case because the oracle optimizers

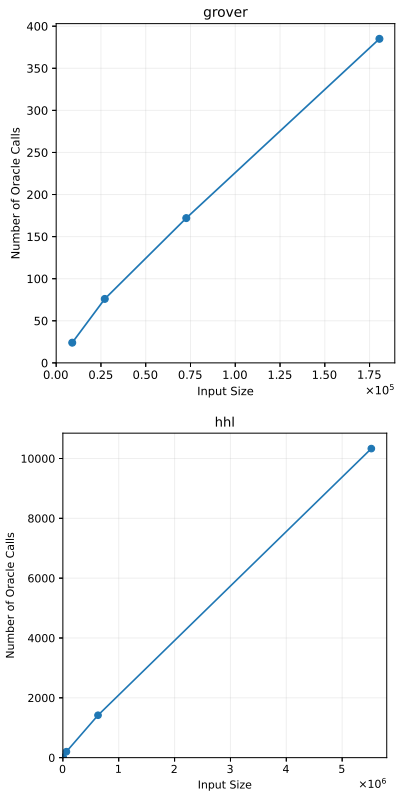

Fig. 7: The number of oracle calls versus input circuit size for selected circuits. The plots show that the number of calls scales linearly with the number of gates.

can take asymptotically non-linear time. For example, the oracle VOQC can require at least quadratic time in the number of gates in the circuit being optimized, which varies as we increase the qubit counts.

F. RQ IV: Impact of compaction and Ω

1) Impact of compaction: Figure 8 shows the number of rounds and percentage optimizations for each round of our OAC algorithm, using the convergence ratio $\epsilon = 0.01$. The results show that OAC converges very quickly, always terminating after 2 rounds of optimization, and that it consistently finds over 99% of the optimizations in the first round. This is because our OAC ensures segment optimality after the first round of optimization (see Section IV), that is, all segments are optimal, though there may be gaps. The experiment shows that although compaction can enable some optimizations by removing the gaps, its impact on these benchmarks is minimal. We separately ran the same experiments with $\epsilon = 0$, which forces the algorithm to run up to perfect convergence, and observed that OAC requires 4 rounds of optimization on average over all circuits. These results show that in practice a small number of compaction rounds suffice to obtain results that are within a very small fraction of the local optimal.

2) Impact of varying segment size Ω: Figure 9 shows the running time and the gate count reduction of OAC with

Family	Qubits	#Rounds	Optimizations	
			Round 1	Round 2
boolsat	28	2	100.00%	0.00%
	30	2	100.00%	0.00%
	32	2	100.00%	0.00%
	34	2	100.00%	0.00%
bwt	17	2	99.59%	0.41%
	21	2	99.79%	0.21%
	25	2	99.83%	0.17%
	29	2	99.90%	0.10%
grover	9	2	99.96%	0.04%
	11	2	99.90%	0.10%
	13	2	99.99%	0.01%
	15	2	99.97%	0.03%
hhl	7	2	99.76%	0.24%
	9	2	99.95%	0.05%
	11	2	99.96%	0.04%
	13	2	99.95%	0.05%
shor	10	2	100.00%	0.00%
	12	2	99.97%	0.03%
	14	2	99.96%	0.04%
	16	2	99.99%	0.01%
sqrt	42	2	99.90%	0.10%
	48	2	99.81%	0.19%
	54	2	99.97%	0.03%
	60	2	100.00%	0.00%
statevec	5	2	100.00%	0.00%
	6	2	99.92%	0.08%
	7	2	99.92%	0.08%
	8	2	99.99%	0.01%
vqe	12	2	99.95%	0.05%
	16	2	99.99%	0.01%
	20	2	99.98%	0.02%
	24	2	99.99%	0.01%

Fig. 8: The number of rounds and the percentage of optimizations in each optimization round of OAC.

Ω	Time (s)	Output gate count (reduction)
2	17.4	27600 (-56.46%)
5	13.9	27620 (-56.43%)
10	17.7	27609 (-56.45%)
20	18.9	27590 (-56.48%)
40	21.8	27570 (-56.51%)
80	35.3	27564 (-56.52%)
160	66.6	27559 (-56.53%)
320	122.8	27551 (-56.54%)
VOQC	74.1	27673 (-56.35%)

Fig. 9: Choice of Ω: performance of OAC with different Ω on the `hhl` circuit with 9 qubits, which initially contains 63392 gates. For reference, we also present the performance of VOQC at the end.

different values of Ω on the `hhl` circuit with 9 qubits, which initially contains 63392 gates. The results show that for a wide range of Ω values, our optimizer produces a circuit of similar quality to the baseline VOQC, and typically does so in significantly less time. When Ω is large, OAC's running time scales linearly with Ω, and the output gate count reduces marginally when Ω increases. We choose $\Omega = 40$ in our evaluation to achieve a balance between running time and output quality but note that many different values work similarly well.

VI. Related Work

We discussed most closely related work in the body of the paper. In this section, we present a broader overview of the work on quantum circuit optimization.

a) Cost Functions: Gate count is a widely used metric for optimizing quantum circuits. In the NISQ era, reducing gate count improves circuit performance by minimizing noise from operations and decoherence. It also reduces resources in fault-tolerant architectures like the Clifford+T gate set. Researchers have developed techniques to reduce gate count by either directly optimizing circuits or resynthesizing parts using efficient synthesis algorithms. We cover optimization techniques later in the section.

In addition to reducing gate counts, compilers like Qiskit and t|ket⟩, implement circuit transformations that optimize cost specific to NISQ architectures. Examples include maximizing circuit fidelity in the presence of noise [50], [71], and reducing qubit mapping and routing overhead (SWAP gates) for specific device topologies [47], [45], [34], [42], or hardware-native gates and pulses [52], [77], [66], [26]. Techniques also exist to optimize/synthesis circuits for specific unitary types, such as classical reversible gates [57], [18], [6], [75], Clifford+T [3], [40], [61], [38], Clifford-cyclotomic [21], V-basis [11], [60], and Clifford-CS [25] circuits. While algorithms for small unitaries produce Clifford+T circuits with an asymptotically optimal number of T gates [24], efficiently generating optimal large Clifford+T circuits remains a challenge. The FeynOpt optimizer is used for optimizing the T count of quantum circuits. It uses an efficient (polynomial-time) algorithm called phase folding [4], to reduce phase gates, such as the T gate, by merging them. More generally, the Feynman toolkit combines phase folding with synthesis techniques to optimize other metrics like the CNOT count [2]. We demonstrate that our OAC algorithm, which guarantees local optimality, can use FeynOpt as an oracle for optimizing T count in Clifford+T circuits in the extended version [5]. These experiments show that our OAC algorithm scales to large circuits without reducing optimization quality.

b) Resynthesis methods: Resynthesis methods focus on decomposing unitaries into sequences of smaller unitaries using algebraic structures of matrices. Examples include the Cartan decomposition [72], the Cosine-Sine Decomposition (CSD), the Khaneja Glaser Decomposition (KGD) [36], and the Quantum Shannon Decomposition (QSD). Some synthesis methods demonstrate optimality for arbitrary unitaries of small size (typically for fewer than five qubits), particularly in terms of gate counts like CNOT gates [59]. However, their efficiency degrades significantly when dealing with larger unitaries; furthermore, they require the time-consuming step of turning the circuit into a unitary. QGo [76] addresses this limitation with a hierarchical approach that partitions and resynthesizes circuits block-by-block. However, due to the lack of optimization across blocks, the performance of QGo depends heavily on how circuits are partitioned. Our local optimality technique, and specifically melding, could be used to address this limitation of QGo.

c) Rule-based and peephole optimization methods: Rule-based methods find and substitute rules in quantum circuits to optimize the circuit [33], [6], [31], [79]. VOQC [31] is

a formally verified optimizer that uses rules to optimize circuits. VOQC implements several optimization passes inspired by state-of-the-art unverified optimizer proposed by Nam et al. [51]. These passes include rules that perform NOT gate propagation, Hadamard gate reduction, single- and two-qubit gate cancellation, and rotation merging. Most of these passes take quadratic time in circuit size, while some can take as much as cubic time [51]. Our experiments show that our local optimization algorithm OAC effectively uses VOQC as an oracle for gate count optimization.

The notion of local optimality proposed in this paper is related peephole optimization techniques from the classical compilers literature [15], [30], [7]. Peephole optimizers typically optimize a small number of instructions, e.g., rewriting a sequence of three addition operations into a single multiplication operation. Our notion of local optimality applies to segments of quantum circuits, without making any assumption about segment sizes (in our experiments, our segments typically contained over a thousand gates). Because peephole optimizers typically operate on small instructions at a time and because they traditionally consider the non-quantum programs, efficiency concerns are less important. In our case, efficiency is crucial, because our segments can be large, and optimizing quantum programs is expensive. To ensure efficiency and quality, we devise a circuit cutting-and-melding technique.

Prior work use peephole optimizers [57], [44] to improve the circuit one group of gates at a time, and repeat the process from the start until they reach a fixed point. The Quartz optimizer also uses a peephole optimization technique but cannot make any quality guarantees [79]. Our algorithm differs from this prior work, in several aspects. First, it ensures efficiency, while also providing a quality guarantee based on local optimality. Key to attaining efficiency and quality is its use of circuit cutting and melding techniques. Second, our algorithm is generic: it can work on large segments (far larger than a peephole) and optimizes each segment with an oracle, which can optimize the circuit in any way it desires, e.g., it can use any of the techniques described above.

PyZX [37] is another rule-based optimizer that optimizes T count. It uses ZX-diagrams to optimize circuits and then extracts the circuit. Circuit extraction for ZX-diagrams is #P-hard [17], and can take up much more time than optimization itself. Because OAC invokes the optimizer many times, circuit extraction for ZX-diagrams can become a bottleneck. In addition, PyZX only minimizes T count and does not explicitly optimize gate count. We therefore did not use PyZX in our evaluation.

d) Search-based methods: Rule-based optimizers may be limited by a small set of rules and are not exhaustive. To address this, researchers have developed search-based optimizers [78], [79], [80], [41] including Quartz [79] and Queso [78] that automatically synthesize exhaustive circuit equivalence rules. Although their rule-synthesis approach differs, both use similar algorithms for circuit optimization. They iteratively operate on a search queue of candidate circuits. In each iteration, they pop a circuit from the queue, rewrite parts of the circuit using equivalence rules, and insert the new circuits back into the queue. To manage the exponential growth of candidate circuits, both tools use a "beam search" approach that limits the search queue size by dropping circuits appropriately. By limiting the size of the search queue, Quartz and Queso ensure that the space usage is linear relative to the size of the circuit. Their running time remains exponential, and they offer a timeout functionality, allowing users to halt optimization after a set time. This approach has delivered excellent reductions in gate count for relatively small benchmarks [78], [79]. However, for large circuits, the optimizers do not scale well because they attempt to search an exponentially large search space.

QFast and QSearch apply numerical optimizations to search for circuit decompositions that are close to the desired unitary [80], [41]. Although faster than search-based methods [16], these numerical methods tend to produce longer circuits, and their running time is difficult to analyze.

e) Learning-based methods: Researchers have also developed machine learning models [22], [62] for optimizing quantum circuits with variational/continuous parameters, which reduce gate count by tuning parameters of shallow circuit ansatze [46], [53], or by iteratively pruning gates [68], [74]. These approaches, however, are associated with substantial training costs [74].

VII. Conclusion

Quantum circuit optimization is a fundamental problem in quantum computing. State-of-the-art optimizers require at least quadratic time in the size of the circuit, which does not scale to larger circuits that are necessary for obtaining quantum advantage, and are unable to make strong quality guarantees. This paper shows that it is possible to optimize circuits for local optimality efficiently by proposing a circuit cutting-and-melding technique. With this cut-and-meld technique, the algorithm cuts a circuit into subcircuits, optimizes them independently, and melds them efficiently, while also guaranteeing optimization quality. Our implementation and experiments show that the algorithm is practical and performs well, leading to more than an order of magnitude performance improvement (on average) while also improving optimization quality. These results show that local optimizations can be effective in large quantum circuits, which are necessary for quantum advantage. These results, however, do not suggest stopping to develop global optimizers, which remains to be an important goal. It is likely, however, due to inherent complexity of the problem (it is QMA hard), global optimizers may struggle to scale to larger circuits efficiently. Because our approach to local optimality is generic, it can scale global optimizers to larger circuits by employing them as oracles for local optimization.

Acknowledgments

This research was supported by the following NSF grants CCF-1901381, CCF-2115104, CCF-2119352, CCF-2107241.

References

[1] Yuri Alexeev, Dave Bacon, Kenneth R Brown, Robert Calderbank, Lincoln D Carr, Frederic T Chong, Brian DeMarco, Dirk Englund, Edward Farhi, Bill Fefferman, et al. Quantum computer systems for scientific discovery. *PRX quantum*, 2(1):017001, 2021.

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

[3] Matthew Amy, Andrew N Glaudell, and Neil J Ross. Number-theoretic characterizations of some restricted clifford+ t circuits. *Quantum*, 4:252, 2020.

[4] 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.

[5] Jatin Arora, Mingkuan Xu, Sam Westrick, Pengyu Liu, Dantong Li, Yongshan Ding, and Umut A. Acar. Local optimization of quantum circuits (extended version), 2025. https://arxiv.org/abs/2502.19526.

[6] 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.

[7] Sorav Bansal and Alex Aiken. Automatic generation of peephole superoptimizers. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XII, page 394–403, New York, NY, USA, 2006. Association for Computing Machinery.

[8] 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.

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

[10] Benjamin Bichsel, Maximilian Baader, Timon Gehr, and Martin Vechev. Silq: A high-level quantum language with safe uncomputation and intuitive semantics. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 286–300, 2020.

[11] Alex Bocharov, Yuri Gurevich, and Krysta M Svore. Efficient decomposition of single-qubit gates into v basis circuits. *Physical Review A*, 88(1):012313, 2013.

[12] Sergey Bravyi, Oliver Dial, Jay M Gambetta, Darío Gil, and Zaira Nazario. The future of quantum computing with superconducting qubits. *Journal of Applied Physics*, 132(16), 2022.

[13] Andrew M Childs, Richard Cleve, Enrico Deotto, Edward Farhi, Sam Gutmann, and Daniel A Spielman. Exponential algorithmic speedup by a quantum walk. In *Proceedings of the thirty-fifth annual ACM symposium on Theory of computing*, pages 59–68, 2003.

[14] 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.

[15] Keith D Cooper and Linda Torczon. *Engineering a compiler*. Morgan Kaufmann, 2022.

[16] 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.

[17] Niel de Beaudrap, Aleks Kissinger, and John van de Wetering. Circuit extraction for zx-diagrams can be# p-hard. *arXiv preprint arXiv:2202.09194*, 2022.

[18] Yongshan Ding, Xin-Chuan Wu, Adam Holmes, Ash Wiseth, Diana Franklin, Margaret Martonosi, and Frederic T Chong. Square: Strategic quantum ancilla reuse for modular quantum programs via cost-effective uncomputation. *arXiv preprint arXiv:2004.08539*, 2020.

[19] 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.

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

[21] Simon Forest, David Gosset, Vadym Kliuchnikov, and David McKinnon. Exact synthesis of single-qubit unitaries over clifford-cyclotomic gate sets. *Journal of Mathematical Physics*, 56(8):082201, 2015.

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

[23] Craig Gidney and Martin Ekerρa. How to factor 2048 bit rsa integers in 8 hours using 20 million noisy qubits. *Quantum*, 5:433, 2021.

[24] 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.

[25] Andrew N Glaudell, Neil J Ross, and Jacob M Taylor. Optimal two-qubit circuits for universal fault-tolerant quantum computation. *arXiv preprint arXiv:2001.05997*, 2020.

[26] 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.

[27] Alexander S Green, Peter LeFanu Lumsdaine, Neil J Ross, Peter Selinger, and Benoît Valiron. Quipper: a scalable quantum programming language. In *Proceedings of the 34th ACM SIGPLAN conference on Programming language design and implementation*, pages 333–342, 2013.

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

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

[30] Stefan Heule, Eric Schkufza, Rahul Sharma, and Alex Aiken. Stratified synthesis: Automatically learning the x86-64 instruction set. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '16, page 237–250, New York, NY, USA, 2016. Association for Computing Machinery.

[31] 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.

[32] Torsten Hoefler, Thomas Häner, and Matthias Troyer. Disentangling hype from practicality: On realistically achieving quantum advantage. *Communications of the ACM*, 66(5):82–87, 2023.

[33] 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.

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

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

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

[37] 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.

[38] Aleks Kissinger and John van de Wetering. Reducing the number of non-clifford gates in quantum circuits. *Physical Review A*, 102(2):022406, 2020.

[39] 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.

[40] Vadym Kliuchnikov, Alex Bocharov, and Krysta M Svore. Asymptotically optimal topological quantum compiling. *Physical review letters*, 112(14):140504, 2014.

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

[42] 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.

[43] 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.

[44] Ji Liu, Luciano Bello, and Huiyang Zhou. Relaxed peephole optimization: A novel compiler optimization for quantum circuits. In *2021*

IEEE/ACM International Symposium on Code Generation and Optimization (CGO), pages 301–314. IEEE, 2021.

[45] 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.

[46] Kosuke Mitarai, Makoto Negoro, Masahiro Kitagawa, and Keisuke Fujii. Quantum circuit learning. *Physical Review A*, 98(3):032309, 2018.

[47] 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.

[48] 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.

[49] Steven A Moses, Charles H Baldwin, Michael S Allman, R Ancona, L Ascarrunz, C Barnes, J Bartolotta, B Bjork, P Blanchard, M Bohn, et al. A race-track trapped-ion quantum processor. *Physical Review X*, 13(4):041052, 2023.

[50] 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.

[51] 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.

[52] 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.

[53] 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.

[54] Jennifer Paykin, Robert Rand, and Steve Zdancewic. Qwire: a core language for quantum circuits. *ACM SIGPLAN Notices*, 52(1):846–858, 2017.

[55] Tianyi Peng, Aram W Harrow, Maris Ozols, and Xiaodi Wu. Simulating large quantum circuits on a small quantum computer. *Physical review letters*, 125(15):150504, 2020.

[56] 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.

[57] Aditya K Prasad, Vivek V Shende, Igor L Markov, John P Hayes, and Ketan N Patel. Data structures and algorithms for simplifying reversible circuits. *ACM Journal on Emerging Technologies in Computing Systems (JETC)*, 2(4):277–293, 2006.

[58] Qiskit contributors. Qiskit: An open-source framework for quantum computing, 2023.

[59] Péter Rakyta and Zoltán Zimborás. Approaching the theoretical limit in quantum gate decomposition. *Quantum*, 6:710, 2022.

[60] Neil J Ross. Optimal ancilla-free clifford+ v approximation of z-rotations. *Quantum Information & Computation*, 15(11-12):932–950, 2015.

[61] Neil J Ross and Peter Selinger. Optimal ancilla-free clifford+ t approximation of z-rotations. *arXiv preprint arXiv:1403.2975*, 2014.

[62] Francisco J. R. Ruiz, Tuomas Laakkonen, Johannes Bausch, Matej Balog, Mohammadamin Barekatain, Francisco J. H. Heras, Alexander Novikov, Nathan Fitzpatrick, Bernardino Romera-Paredes, John van de Wetering, Alhussein Fawzi, Konstantinos Meichanetzidis, and Pushmeet Kohli. Quantum circuit optimization with AlphaTensor. 7(3):374–385. Publisher: Nature Publishing Group.

[63] 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.

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

[65] Peter Selinger. Towards a quantum programming language. *Mathematical Structures in Computer Science*, 14(4):527–586, 2004.

[66] 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.

[67] 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.

[68] 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.

[69] Seyon Sivarajah, Silas Dilkes, Alexander Cowtan, Will Simmons, Alec Edgington, and Ross Duncan. t| ket>: a retargetable compiler for nisq devices. *Quantum Science and Technology*, 6(1):014003, 2020.

[70] Wei Tang, Teague Tomesh, Martin Suchara, Jeffrey Larson, and Margaret Martonosi. Cutqc: using small quantum computers for large quantum circuit evaluations. In *Proceedings of the 26th ACM International conference on architectural support for programming languages and operating systems*, pages 473–486, 2021.

[71] 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.

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

[73] Finn Voichick, Liyi Li, Robert Rand, and Michael Hicks. Qunity: A unified language for quantum and classical computing. *Proceedings of the ACM on Programming Languages*, 7(POPL):921–951, 2023.

[74] 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.

[75] Robert Wille, Majid Haghparast, Smaran Adarsh, and M Tanmay. Towards hdl-based synthesis of reversible circuits with no additional lines. In *2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 1–7. IEEE, 2019.

[76] 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.

[77] 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.

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

[79] 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.

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

[81] Charles Yuan and Michael Carbin. Tower: data structures in quantum superposition. *Proceedings of the ACM on Programming Languages*, 6(OOPSLA2):259–288, 2022.

[82] Charles Yuan, Christopher McNally, and Michael Carbin. Twist: Sound reasoning for purity and entanglement in quantum programs. *Proceedings of the ACM on Programming Languages*, 6(POPL):1–32, 2022.