Reducing the CNOT Count for Clifford+$T$ Circuits on NISQ Architectures

Vlad Gheorghiu, Jiaxin Huang, Sarah Meng Li, Member, IEEE, Michele Mosca, and Priyanka Mukhopadhyay

Abstract—While mapping a quantum circuit to the physical layer one has to consider the numerous constraints imposed by the underlying hardware architecture. Connectivity of the physical qubits is one such constraint that restricts two-qubit operations, such as CNOT, to “connected” qubits. SWAP gates can be used to place the logical qubits on admissible physical qubits, but they entail a significant increase in CNOT-count. In this article, we consider the problem of reducing the CNOT-count in Clifford+$T$ circuits on connectivity-constrained architectures, like noisy intermediate-scale quantum (NISQ) computing devices. We “slice” the circuit at the position of Hadamard gates and “build” the intermediate $[\text{CNOT}, T]$ subcircuits using Steiner trees, significantly improving on previous methods. We compared the performance of our algorithms while mapping different benchmark and random circuits to some well-known architectures, such as 9-qubit square grid, 16-qubit square grid, Rigetti 16-qubit Aspen, 16-qubit IBM QX5, and 20-qubit IBM Tokyo. Our methods give less CNOT-count compared to Qiskit and TKET transpiler as well as using SW AP gates. Assuming most of the errors in an NISQ circuit implementation are due to CNOT errors, then our method would allow circuits with a few times more CNOT gates to be reliably implemented than the previous methods would permit.

Index Terms—CNOT-count, connectivity constraints, noisy intermediate-scale quantum (NISQ) architectures, Steiner tree.

I. INTRODUCTION

Quantum computing is a computational paradigm which is predicted to provide significant speedups for problems, including, but not limited to, large number factorization [1], simulation of quantum systems [2], and unstructured search [3], all of which are believed to be intractable or significantly slower on a classical computer. Somewhat similar to its classical counterpart, a quantum circuit consisting of elementary unitary operations remains the most popular model for quantum computation. Thus, we need efficient quantum compilers that map a high-level algorithm into a lower-level form, that is, a quantum circuit consisting of quantum gates that are admissible by the hardware constraints.

Presently, we do not have large-scale quantum computers. Rather the devices available today are referred to as noisy intermediate-scale quantum (NISQ) computers [4]. The current technologies that realize these devices, such as superconducting quantum circuits [5], [6] and ion traps [7], [8], [9], [10], impose certain connectivity constraints by which two-qubit operations are possible only among certain pairs of physical qubits. Naively we can insert SWAP operators to move a pair of logical qubits to physical positions admissible for two-qubit operations. However, this increases the number of two-qubit operations, each of which again introduces non-negligible noise. Hence, it is important to optimize the number of two-qubit operators while respecting the connectivity constraints.

In this article, we consider the problem of resynthesizing a circuit over the universal fault-tolerant Clifford+$T$ gate set. We designed and implemented an algorithm that reduces the number of CNOT gates required to meet the connectivity constraints imposed by the physical hardware architectures. The connectivity constraints are represented in the form of a graph $G$ (called connectivity graph) in which the vertices represent (physical) qubits and a two-qubit operation can be applied if and only if the corresponding vertices are connected by an edge in $G$. We assume without loss of generality that the desired circuit is connected (that is, we cannot break it up into noninteracting collections of qubits).

The Clifford+$T$ gate set is one of the most studied fault-tolerant universal gate set used to realize a quantum operator. We consider the following gates in this set: $G_{\text{univ}} = \{\text{CNOT}, H, T, T^\dagger, S, S^\dagger, X, Y, Z\}$, among which CNOT is the only multiqubit operator. If the shortest path length between vertices corresponding to $c$ and $t$ in $G$ is $\ell$, then the naive way of using SWAP gates (equivalent to three CNOT gates) would require $6(\ell - 1)$ CNOT gates (Fig. 1).
Thus, we devise algorithms using Steiner trees that reduce the number of CNOT gates. Steiner trees were also used in [11] and [12] for the similar goal of reducing CNOT gates. Our algorithms differ from these works, which we have pointed out in the following paragraphs, and give much less CNOT-count.

A. Our Techniques

Our approach can be described as slice-and-build. Given a circuit $C_1$ as a sequence of gates we slice it at “suitable” points and resynthesize or build the intermediate sliced portions in a manner such that connectivity constraints are respected and at the same time we aim to reduce the number of CNOT gates. We have described two methods for slice-and-build.

The first procedure, CNOT-OPT-A (Algorithm 6) described in Section IV-A, has a simple slicing technique. We partition the circuit $C_1$ at the position of the Hadamard ($H$) gates. Each intermediate subcircuit composed of the gates $G_{ph} = G_{univ} \setminus \{H\}$ is resynthesized using algorithms PHASE-NW-SYNTH (Algorithm 5) and LINEAR-TF-SYNTH (Algorithm 2). For each subcircuit, we first calculate the phase polynomial $P$ and overall linear transformation $A_{slice}$. We synthesize a phase polynomial network circuit $C_{ph}$ with the gates in $G_{ph}$, using PHASE-NW-SYNTH (Algorithm 5) described in Section III-B. The algorithm draws inspiration from the parity network synthesis algorithm in [13]. We calculate the parity network matrix in which each column stores a parity term. The aim is to apply a series of transformations (CNOT gates) such that each parity term occurs at least once in the circuit. Then, depending on the coefficients of the parity terms we place the gates in $G_{ph} \setminus \{CNOT, X\}$. To impose connectivity constraints we construct Steiner trees (Section II-C) with terminals being the set of qubits (or vertices) satisfying certain conditions. Then depending on the edge information, we perform a series of CNOT operations to get the desired result. We emphasize that our way of placing CNOT gates according to the Steiner tree edges is different from that described in [11]. Kissinger and van de Griend [12] remarked that Steiner trees can be used for synthesizing a circuit from its phase polynomial, but no detail was given.

The phase polynomial network corresponding to $P$ has some overall transformation $A_{ph}$. We synthesize a circuit $C_{lin}$ that implements the “residual” linear transformation $A = A_{ph}^{-1}A_{slice}$ using LINEAR-TF-SYNTH (Algorithm 2), described in Section III-A. The main motivation of this algorithm comes from the work in [14] that synthesizes a linear reversible circuit using CNOT gates. We follow the same reverse-engineering procedure where we 1) reduce $A$ first to an upper triangular form, 2) transpose the result and then 3) reduce it to a lower triangular form so that we get the identity matrix $I$. Each linear operation corresponds to a CNOT gate and connectivity constraints are imposed by constructing series of Steiner trees. Our procedures at steps 1) and 3) differ from the approach in [11] and [12], and we also get less CNOT-count.

In our second procedure CNOT-OPT-B (Algorithm 7) described in Section IV-B, the slicing points remain the $H$ gates but the set that is partitioned is the phase polynomial $P_I$ of the entire circuit $C_1$. Between two $H$ gates (including the ends) we synthesize a phase polynomial network circuit using gates in $G_{ph}$ that realizes the partial phase polynomial $P_{sub}$, comprising of terms in $P$ that become uncomputable after the $H$ gate being placed at the end of the current slice. Here, it must be noted that by the sum-over-paths formulation (Section II-B) new path variables are introduced after application of each $H$ gate. This renders some terms of the phase polynomial uncomputable after certain points in the circuit. The synthesis of the phase polynomial network is done using PHASE-NW-SYNTH (Algorithm 5). Let $A_{slice}$ be the transformation that maps the state of the qubits in $C_1$ after the $H$ gate at the beginning of a slice to the state of the qubits in $C_1$ before the $H$ gate at the end of the slice. We synthesize a circuit implementing $A = A_{ph}^{-1}A_{slice}$ using LINEAR-TF-SYNTH (Algorithm 2) such that between any two $H$ gates (as well as at the ends) the linear transformation $A_{slice}$ remain unchanged. A similar kind of partitioning of the circuit according to the phase polynomial was used in [15] where the goal was to reduce T-depth of the input circuit.

B. Our Results

We have resynthesized some benchmark as well as randomly generated circuits after taking into account connectivity constraints imposed by the architectures (which have been implemented in practice) shown in Figs. 1 and 2. Here, we
emphasize that we have studied the performance of our procedures as baseline algorithms. The results will likely improve if coupled with some other procedures that handle the problem of the optimal initial mapping of qubits. To be precise, we have considered only one mapping where qubit \( i \) is mapped to vertex \( i \) of the given connectivity graphs. Considerable amount of work has been done, which considers the optimal mapping that reduces the resources required. So if such a procedure is done as preprocessing, then the CNOT-count will likely reduce further. We have compared the CNOT-count overhead, that is, the increase in CNOT-count obtained from our algorithms with the overhead obtained using SWAP-template (Fig. 1), Qiskit [17], and TKET [18] transpiler. We have observed that both our algorithms give remarkable improvement in the case of random circuits. In each case of random circuits, we have found that the simple way of slicing in CNOT-OPT-A gives much less overhead (Table I in Section IV-C). CNOT-OPT-B, however, fares poorly in many cases.

C. Related Work

There have been quite a number of works that deal with the problem of CNOT optimization without taking into account the connectivity constraints imposed by the underlying hardware architecture, for example, [14], [19], [20], [21], [22], [23], and [13].

Some authors [24], [25], [26], [27], [28] use SWAP gates along with some gate commutation and transformation rules to obtain a circuit that respect connectivity constraints. There are algorithms that take advantage of the restricted topology, such as 1-D linear nearest neighbor ([29], [30], [31], [32], [33], [34]), hypercubic [35] which rely on classical sorting networks and 2-D grid ([26], [36], [37], [38]). Some algorithms that work on general topology for NISQ devices are [24], [39], [40], [41], [42], [43]. Broadly, these algorithms use the qubit mapping technique to search for the optimal placement of SWAP gates and qubits. The search space scales exponentially for exact algorithms, such as [44] and [45], making them impractical for large NISQ devices. Thus, some authors [24], [42], [46] use heuristics to reduce the search space. Some of these heuristics algorithms, e.g., [39], which is based on depth partitioning and \( A^* \) search, are developed for specialized architectures such as IBM devices. Ferrari and Amoretti [47] gave an approach for realizing arbitrary parity-function oracles, while taking care of the underlying topology. It has been shown in [48] that the size of the resulting circuit is very sensitive to the original placement of the logical qubits on the device.

To reduce CNOT-count, Steiner trees have been used in [11], [12], [49], and [50], while in [16] the problem is reduced to a well-known cryptographic problem—the syndrome decoding problem.

D. Organization

After giving some preliminaries in Section II we describe our algorithms in Sections III and IV. The algorithms LINEAR-TF-SYNTH and PHASE-NW-SYNTH that synthesize linear reversible circuits and phase polynomial network circuits are given in Sections III-A and III-B, respectively. The algorithms CNOT-OPT-A and CNOT-OPT-B that synthesize the complete circuit over the Clifford+T gate set is described in Sections IV-A and IV-B, respectively. Finally, we conclude in Section V.

II. Preliminaries

We write \( N = 2^n \) and \([K] = \{1, 2, \ldots, K\}\). The \((i, j)\)th entry of any matrix \( M \) is denoted by \( M_{ij} \) or \( M_{i,j} \). We denote the \( i \)th row of \( M \) by \( M[i, \_] \) and the \( j \)th column by \( M[\_j] \). We denote the \( n \times n \) identity matrix by \( I_n \) or \( I \) if the dimension is clear from the context. The set of \( n \)-qubit unitaries of size \( 2^n \times 2^n \) is denoted by \( U(2^n) \) or \( U_n \).

A. Cliffords and Paulis

The single qubit Pauli matrices are as follows:

\[
X = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}, \quad Y = \begin{bmatrix} 0 & -i \\ i & 0 \end{bmatrix}, \quad Z = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}.
\]

The \( n \)-qubit Pauli operators are

\[ P_n = \{Q_1 \otimes Q_2 \otimes \ldots \otimes Q_n, Q_i \in \{I, X, Y, Z\}\}. \]

The single-qubit Clifford group \( C_1 \) is generated by the Hadamard and phase gate

\[ C_1 = (H, S), \quad H = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}, \quad S = \begin{bmatrix} 1 & 0 \\ 0 & i \end{bmatrix}. \]

When \( n > 1 \) the \( n \)-qubit Clifford group \( C_n \) is generated by these two gates (acting on any of the \( n \) qubits) along with the two-qubit CNOT = \(|0\rangle \langle 0| \otimes I + |1\rangle \langle 1| \otimes X \) gate (acting on any pair of qubits). We write CNOT\(_{ct} \) to denote the CNOT gate applied between qubit \( c \) (control) and \( t \) (target) and CNOT\(_c t \) = \(|c, c \oplus t\rangle \).

The Clifford+T gate set consists of the \( n \)-qubit Clifford group gates along with the T gate, where

\[ T = \begin{bmatrix} 1 & 0 \\ 0 & e^{\frac{i\pi}{4}} \end{bmatrix}. \]

It is easy to verify that this set is a group since the \( H \) and CNOT gates are their own inverses and \( T^{-1} = T^7 \). Here, we note \( S = T^2 \). For multiple qubits, a minimal generating set for this group is \( \{H, T, \text{CNOT}\} \).

B. Circuit-Polynomial Correspondence

The circuit-polynomial correspondence [51] associates a phase polynomial and a linear Boolean transformation with every quantum circuit generated by \{CNOT, H, T\}. More precisely,

**Lemma 1 [15]:** A unitary \( U \in U(2^n) \) is exactly implementable by an \( n \)-qubit circuit over \{CNOT, T\} if and only if

\[ U \mid x_1, x_2, \ldots, x_n \rangle = e^{i\omega(x_1, x_2, \ldots, x_n)} \mid g(x_1, x_2, \ldots, x_n) \rangle \]

Authorized licensed use limited to the terms of the applicable license agreement with IEEE. Restrictions apply.
where \( \omega = e^{ix/4} \), \( x_1, x_2, \ldots, x_n \in \mathbb{F}_2 \) and
\[
p(x_1, x_2, \ldots, x_n) = \sum_{i=1}^{\ell} c_i \cdot f_i(x_1, x_2, \ldots, x_n)
\]
for some linear reversible function \( g : \mathbb{F}_2^n \to \mathbb{F}_2^n \) and linear Boolean functions \( f_1, f_2, \ldots, f_\ell \in (\mathbb{F}_2^n)^* \) with coefficients \( c_1, c_2, \ldots, c_\ell \in \mathbb{Z}_8 \).

For convenience, the set of \( n \)-ary linear Boolean functions \( \mathbb{F}_2^n \to \mathbb{F}_2^n \) is referred to as the dual vector space \((\mathbb{F}_2^n)^*\) of \( \mathbb{F}_2^n \).

This is called the sum-over-paths form of a circuit [51], [52], [53] and the variables \( x_1, x_2, \ldots, x_n \) are called the path variables. \( p(x_1, x_2, \ldots, x_n) \) is referred to as the phase polynomial. Each \( f_i(x_1, \ldots, x_n) \) is a parity term.

Thus, we can fully characterize a unitary \( U \in U(2^n) \) implemented by a \([\text{CNOT}, T]\)-generated circuit with a set \( \mathcal{P} \subseteq \mathbb{Z}_8 \times (\mathbb{F}_2^n)^* \) of linear Boolean functions together with coefficients in \( \mathbb{Z}_8 \) and a linear reversible output function \( g : \mathbb{F}_2^n \to \mathbb{F}_2^n \), with the interpretation
\[
U(\mathcal{P}, g) : (x_1, x_2, \ldots, x_n) \to y, \quad \text{where} \quad y = \omega \sum_{(c, f) \in \mathcal{P}} c \cdot f(x_1, x_2, \ldots, x_n) \cdot g(x_1, x_2, \ldots, x_n).
\]

The set \( \mathcal{P} \) (phase polynomial set) and \( g \) are efficiently computable given a circuit over \([\text{CNOT}, T]\), taking time linear in the number of qubits and gates.

The \( H \) gate is a “branching gate” and has the following effect on a basis state \( x_1 \in \mathbb{F}_2^n \):
\[
H : x_1 \to \frac{1}{\sqrt{2}} \sum_{x_2 \in \mathbb{F}_2^n} \omega^{x_1 \cdot x_2} |x_2\rangle.
\]

Here, \( x_2 \) is the new path variable and the variable \( x_1 \) ceases to exist after \( H \) is applied. Similar to Lemma 1 we can have the following result.

Lemma 2 [15]: If a unitary \( U \in U(2^n) \) is exactly implementable by an \( n \)-qubit circuit over \([\text{CNOT}, H, T]\) with \( k \) \( H \) gates, then for \( x_1, x_2, \ldots, x_n \in \mathbb{F}_2^n \)
\[
U |x_1, x_2, \ldots, x_n\rangle = \frac{1}{\sqrt{2^n}} y, \quad \text{where} \quad y = \sum_{x_{n+1}, \ldots, x_{n+k} \in \mathbb{F}_2^n} \omega^{x_1 \cdot x_{n+k}} |y_1 y_2 \ldots y_n\rangle
\]

\[
y_i = h_i(x_1, x_2, \ldots, x_{n+k}), p(x_1, x_2, \ldots, x_{n+k})
\]

\[
= \sum_{i=1}^{\ell} c_i \cdot f_i(x_1, \ldots, x_{n+k}) + 4 \cdot \sum_{i=1}^{\ell} x_{n+i} \cdot g_i(x_1, \ldots, x_{n+k})
\]

for some linear Boolean functions \( h_i, f_i, \) and \( g_i \) and coefficients \( c_i \in \mathbb{Z}_8 \). The \( k \) path variables \( x_{n+1}, \ldots, x_{n+k} \) result from the application of Hadamard gates.

But, unlike Lemma 1, the converse is not true.

C. Steiner Tree

A graph is a pair \( G = (V_G, E_G) \) where \( V_G \) is a set of vertices and \( E_G \) is a set of pairs \( e = (u, v) \) such that \( u, v \in V_G \). Each such pair is called an edge. One may define a function \( w_{E_G} : E_G \to \mathbb{R} \) that assigns a weight to each edge. The graphs we consider are simple (with at most one edge between two distinct vertices and no self-loops, i.e., \( (u, u) \notin E_G \), undirected [edges have no direction, i.e., \( (u, v) \equiv (v, u) \)] with edge-weight 1, that is, \( w_{E_G}(e) = 1 \) for every \( e \in E_G \). A graph \( G' = (V_{G'}, E_{G'}) \) is a subgraph of \( G \) such that \( V_{G'} \subseteq V_G \) and \( E_{G'} \subseteq E_G \). A tree is an undirected graph in which any two vertices are connected by exactly one path, or equivalently a connected, acyclic, and undirected graph.

Definition 1 (Steiner Tree): Given a graph \( G = (V_G, E_G) \) with a weight function \( w_E \) and a set of vertices \( S \subseteq V_G \), a Steiner tree \( T = (V_T, E_T) \) is a minimum weight tree that is a subgraph of \( G \) such that \( S \subseteq V_T \). The set of vertices in \( S \) are called terminals while those in \( V_T \setminus S \) are called Steiner nodes.

Computing Steiner trees is NP-hard and the related decision problem is NP-complete [54]. There are a number of heuristic algorithms that compute approximate Steiner trees [55], [56], [57]. The choice of algorithm is usually determined by the application, and typically involves a tradeoff between quality (approximation factor) and running time. We use the algorithm given in [58] (Algorithm 1). This helps us achieve a better running time compared to the best (with respect to quality) approximation algorithms in the literature, without sacrificing the approximation factor much.

The size of the constructed Steiner tree is at most \( 2(1 - (1/\ell)) \) times the size of the minimal Steiner tree, where \( \ell \) is the number of leaves in the minimal Steiner tree. The running time is \( O(|S|^2(|V_G| + |E_G|)) \) [58].

III. SYNTHESIS ALGORITHMS WITH CONNECTIVITY CONSTRAINT

In this section first we describe a synthesis algorithm that generates a circuit implementing a linear transformation using gates in the set \( \mathcal{G}_{\text{lin}} = \{\text{CNOT}, X\} \) (Section III-A). Then, we describe an algorithm that synthesizes a circuit implementing a phase polynomial network using gates \( \mathcal{G}_{\text{ph}} \) generated by the set \( \{\text{CNOT}, X, T\} \).

A. Synthesis of Circuits Over \([\text{CNOT}, X]\)

Consider an \( n \)-qubit circuit built with gates in the set \( \mathcal{G}_{\text{lin}} = \{\text{CNOT}, X\} \). We represent the overall linear transformation by an \( n \times n + 1 \) “augmented” matrix \( A = [A'_{\ell \times n}|b_{\ell \times 1}] \), whose rows represent or are indexed by qubits. If we label the initial states of the qubits by variables \( x_1, \ldots, x_n \) then

\[\begin{algorithm}
\textbf{Algorithm 1: Steiner Tree Algorithm}
\begin{enumerate}
\item \textbf{Input:} (i) A graph \( G = (V, E) \), (ii) Terminal set \( S = \{s_1, \ldots, s_k\} \subseteq V \)
\item \textbf{Output:} A Steiner tree constructed from \( G \)
\item Construct a forest \( F \) of \( k \) sub-graphs \( f_1, \ldots, f_k \) consisting of one terminal each;
\item while does not exist a \( f_i \in F \) such that all terminals \( s_1, \ldots, s_k \) inf \( f_i \) do
\item \quad for all \( i \neq j \) do
\item \quad \quad Determine the shortest path between all nodes in \( f_i \) to all those in \( f_j \);
\item \quad end
\item \quad Find the minimum length path \( P \) among all computed paths;
\item \quad Construct \( f_n = f_i \cap f_j \cap P \) and add it to forest \( F \);
\item end
\item Construct a minimum spanning tree \( T \) on \( f_i \);
\item Remove non-terminal nodes of degree 1 from \( T \);
\item return \( T \);
\end{enumerate}
\end{algorithm}\]
the first \( n \) columns represent these variables and the last column represents the variable \( b \) indicating bit flips. Each variable \( x_1, \ldots, x_r, b \) takes values from the set \( \{0, 1\} \). The initial state of \( A \) is \( [b_0 | 0_{n \times 1}]_{n \times n+1} \). This represents the initial state of all the qubits. When CNOT gate \( i \), is applied row \( j \) is added (mod 2) to row \( i \) (row \( j \) remains same). The parity at qubit \( i \) is \( x_i \otimes x_j \). When an X gate is applied on qubit \( i \) then \( A_{i,n+1} \leftarrow 1 \oplus A_{i,n+1} \).

Now suppose we are given a linear transformation \( A = [A_{i,j}]_{n \times n+1} \) of a circuit and we want to synthesize a circuit implementing this transformation. We do reverse engineering (as in [14]), using a procedure similar to the Gaussian elimination. 1) First we make \( b = 0 \) by flipping the entries with 1. This corresponds to applying X on the respective qubit. 2) We apply a series of elementary row operations (bit-wise addition) on \( A \) such that \( A' \) is in upper triangular form. Each row operation represents the application of a CNOT gate. 3) Then, we transpose the matrix and perform elementary row operations on \( A^T \) such that \( A' \) is \( I \). The output circuit is constructed as follows: first, the CNOT gates obtained in 3) with the control-target flipped but preserving the order, then the CNOT gates obtained in 2) with the control-target preserved but reversing the order in which they were performed, and lastly the X gates obtained in 1).

To incorporate connectivity constraints we use Steiner trees as described in LINEAR-TF-SYNTH (Algorithm 2). We first make \( b = 0 \) by placing X gates (step 2), as described before. Then, we convert \( A' \) into an upper triangular form (step 8) by row-operations “permitted” by the input connectivity graph \( G \). For each column of \( A' \) (starting from the first one) we compute a minimal Steiner tree approximation with: 1) connectivity graph \( G \) and terminals \( S \) which are the rows below the diagonal and having a 1 in the diagonal is “propagated” via intermediate Steiner nodes to have 1 in the diagonal and 0 in the rest and 2) set of terminals \( S \) which are the rows below the diagonal and having a 1. Then we invoke the procedure ROW-OP, as described in Algorithm 3.

The idea of ROW-OP is to use a set of operations such that if the diagonal is “propagated” via intermediate Steiner nodes to cancel the 1 in the terminal nodes and then use another set of operations to cancel any 1s in the Steiner nodes. We assume the diagonal has 1, else it is adjusted by a set of operations to propagate a 1 to the diagonal node (step 4 of Algorithm 2). The diagonal node (let’s call it \( c \)) becomes the “pivot” node. The input Steiner tree approximation \( T_{c,S} \) is separated into a set of subtrees (step 1 of Algorithm 4) by calling the procedure SEPARATE (Algorithm 4). The root and leaves in each such subtree are terminal nodes (from \( S \)) and the rest are Steiner nodes. Then the 1 from the root of each subtree cancels the 1 at the leaves via operations performed in steps 11 and 13 of Algorithm 3 and the 1 s at Steiner nodes get canceled by the operations performed in steps 15 and 17 of the same procedure. If in a subtree the root node is \( r \) and the leaves are \( \ell_1, \ldots, \ell_m \) then the parity at the root node and each Steiner node remains unchanged but the parity at leaf \( \ell_i \) becomes \( x_j \oplus x_i \oplus \bigoplus_{j \in P} x_j \) where \( P \) is the set of Steiner nodes in the path from \( r \) to \( \ell_i \). The resultant matrix \( A' \) is in upper triangular form.

Next we transpose \( A' \). Our goal is now to convert \( A'^T \) into upper triangular form without destroying the 0 s in the upper triangle. This in turn implies that for each nondiagonal node \( j \) we want the parity to be \( x_j \oplus x_{\ell_j} \), where \( k < j \) and \( x_j, x_{\ell_j} \) are the parities at node \( j \) and \( k \), respectively, before the transpose step 11 in Algorithm 2. Similarly as before, we invoke the procedure ROW-OP (Algorithm 3) but this time we include steps 6, 8, 20, and 22 in it, so that for each subtree constructed the parity at root \( r \) and Steiner nodes remain unchanged, but the parity at each leaf node \( \ell \) becomes \( x_{\ell_j} \oplus x_{\ell_j} \). Now if \( r > \ell \) then we perform some correction procedures (steps 21–29). Our goal is still to “cancel” each parity at \( \ell \). This in turn implies that for each nondiagonal node \( j \) we want the parity to be \( x_j \oplus x_{\ell_j} \), where \( k < j \) and \( x_j, x_{\ell_j} \) are the parities at node \( j \) and \( k \), respectively, before the transpose step 11 in Algorithm 2. Similarly as before, we invoke the procedure ROW-OP (Algorithm 3) but this time we include steps 6, 8, 20, and 22 in it, so that for each subtree constructed the parity at root \( r \) and Steiner nodes remain unchanged, but the parity at each leaf node \( \ell \) becomes \( x_{\ell_j} \oplus x_{\ell_j} \). Now if \( r > \ell \) then we perform some correction procedures (steps 21–29). Note the parity at \( r \) is \( x_{\ell_j} \oplus x_{\ell_j} \) where \( r \) is the root of the subtree in which \( r \) is a leaf. Then, if we invoke ROW-OP with the shortest path from \( r \) to \( \ell \) as a tree, then the parity at \( \ell \) becomes \( x_{\ell_j} \oplus x_{\ell_j} \). Every other parity remains unaffected. If \( r > \ell \), then we again invoke ROW-OP with the shortest path from \( r \) to \( \ell \) as a tree. We continue doing this till the parity at \( \ell \) is “corrected,” i.e., it becomes \( x_{\ell_j} \oplus x_{\ell_j} \) for some \( k < \ell \). We start these correction procedures from the

---

**Algorithm 2: LINEAR-TF-SYNTH**

**Input:** (i) Linear transformation matrix \( A_{n 	imes n+1} = [A_{i,j}]_{n 	imes n+1} \).

**Output:** A circuit over \([\text{CNOT}, X]\) realizing \( A \).

1. \( Y_{1,2}, X \leftarrow \emptyset, G' \leftarrow G \) // Make a copy of \( G \).
2. If \( A_{2+1,i} = 1 \) then \( X \cdot \text{append} \left(X_i\right) \) and \( A_{2+1,i} \leftarrow 0 \).
3. for columns \( i = 1 \ldots n \) do
   4. if \( A_{i} = 0 \), find all rows \( j > i \) such that \( A_{ij} = 1 \). Choose \( j \) with the shortest path (in \( G' \)) to \( i \). Suppose \( i \) is the root and \( j \) is the leaf. \( Y_{1} \cdot \text{append} \left(\text{CNOT}_{ij} \right) \) if \( i \) is the child of \( v \).
   5. \( A[v] \cdot \text{append} \left(A[v] \cdot \text{append} \left(\text{CNOT}_{ij} \right) \right) \).
   6. \( S' = \{j > i \text{ and } A_{ij} = 1\}, S \leftarrow S' \cup \{i\} \). // \( S \) is the set of terminals;
   7. Find a minimum Steiner tree approximation with connectivity graph \( G' \) and terminals \( S \).
   8. \( Y_{1} \cdot \text{append} \left(Y_{1} \right), G' \leftarrow G' \setminus \{i\} \) // Remove vertex \( i \) and its edges from \( G' \);

end

end

10. Transpose \( A' \) and \( G' \).
11. for columns \( i = 1 \ldots n \) do
   12. Repeat steps 3-6;
   13. \( Y_{1} \cdot \text{append} \left(Y_{1} \right), G' \leftarrow G' \setminus \{i\} \).

end

end

19. \( (r, \ell_1, \ldots, \ell_m) \leftarrow T_r \{\ell_i \} \);
20. while \( \exists k \) such that \( \ell_k < r \) do
21. \( \ell \leftarrow \ell_k \);
22. while \( r > \ell \) do
   23. \( S' = \{r \cdot \ell_i \}; T_r, S \leftarrow \text{The shortest path between } r \text{ and } \ell \text{ stored as tree with root } r \text{ and leaf } \ell \); 
   24. \( Y_{1} \cdot \text{append} \left(Y_{1} \right), G' \leftarrow G' \setminus \{i\} \);
25. \( B[r] = B[r] \cup \{r \} \);

end

end

32. \( Y_{2} \leftarrow Y_{2} \) with control and target flipped for each CNOT gate;
33. return \( Y_{2} \cup \text{reverse}(Y_{1}) \) as the circuit.
Algorithm 3: ROW-OP

Input: (i) Linear transformation matrix \( \mathbf{A} \), (ii) Set of terminals \( S \), (iii) Pivot node \( c \), (iv) A minimal Steiner tree approximation \( T_{c,S} \), (v) \( \text{alg} \in \mathbb{Z} \).

Output: (i) List \( \mathcal{Y} \) of CNOT operations, (ii) \( \mathbf{A} \) (after the row operations), (iii) \( T_1 = (r, \ell_1, \ldots, \ell_m) \): \( r \) is the root and \( \ell_1, \ldots, \ell_m \) are the leaves in a sub-tree; and leaves \( S_k \setminus \{k\} \subset S \).

1. \( \mathcal{T} \leftarrow \text{SEPARATE}(T_{c,S}, \mathcal{Y}, \text{alg}) \) // Separate \( T_{c,S} \) into a set \( \mathcal{T} \) of sub-trees \( T_i \) with root \( k \in S \) and leaves \( S_k \setminus \{k\} \subset S \).

2. \( T_1 = (r, \ell_1, \ldots, \ell_m) \): \( r \) is the root and \( \ell_1, \ldots, \ell_m \) are the leaves in a sub-tree;

3. \( t \leftarrow |T| \); \( \mathcal{Y} \leftarrow \emptyset; \)

4. for \( i = 1, 1, 2, \ldots, 0 \) // Starting from the last sub-tree do
   
   if \( \text{alg} \neq 1 \) then
      
      (Bottom-Up-1): Starting from the last layer till layer 1,
      \( \mathcal{Y} \leftarrow \text{append}(\text{CNOT}) \) if \( u \) is a non-root parent node and \( v \) is a child of \( u \);
      
      if \( \text{alg} \neq 4 \) then
         
         \( [A[v_1] \leftarrow A[v_1] \oplus A[u_v]; \)
      
   end
   
   end

   (Top-Down-1): Starting from top layer till last layer,
   \( \mathcal{Y} \leftarrow \text{append}(\text{CNOT}) \) if \( u \) is a parent of \( v \);

   if \( \text{alg} \neq 4 \) then
      
      \( [A[v_1] \leftarrow A[v_1] \oplus A[u_v]; \)
   
   end

   (Bottom-Up-2): Starting from the last layer till layer 0,
   \( \mathcal{Y} \leftarrow \text{append}(\text{CNOT}) \) if \( u \) is a parent node and \( v \) is a non-root child node of \( u \);

   if \( \text{alg} \neq 4 \) then
      
      \( [A[v_1] \leftarrow A[v_1] \oplus A[u_v]; \)
   
   end

   (Top-Down-2): Starting from the second layer till last layer,
   \( \mathcal{Y} \leftarrow \text{append}(\text{CNOT}) \) if \( u \) is a parent node (that is not a child of the root) and \( v \) is a non-root leaf child of \( u \);

   if \( \text{alg} \neq 4 \) then
      
      \( [A[v_1] \leftarrow A[v_1] \oplus A[u_v]; \)
   
   end

   if \( \text{alg} = 0 \) then
      
      \( [A[v_1] \leftarrow A[v_1] \oplus A[c_v]; \) where \( r, \ell \) are the root and leaf of the current sub-tree respectively; \)
   
   end

if \( \text{alg} \neq 2 \) then

   return \( (\mathcal{Y}, \mathbf{A}) \);

else

   return \( (\mathcal{Y}, \mathbf{A}, T_1) \);

end

Algorithm 4: SEPARATE

Input: (i) Steiner tree \( T_{c,S} \), (ii) Pivot \( c \), (iii) Set of terminals \( S \).

Output: \( \mathcal{T} = \{T_k : S_k \} \) Edge-disjoint sub-trees with root \( k \in S \) and leaves \( S_k \setminus \{k\} \subset S \).

1. \( T \leftarrow \emptyset \); \( \text{root} \leftarrow c \); \( R \leftarrow \{\text{root}\}; \) \text{delete}(c);

2. while \( S \neq \emptyset \) do

   3. \( S_{\text{root}} \leftarrow \{\text{root}\}; \) \( T_{\text{root}} \leftarrow \emptyset \) // Initialize the data-structure to store a sub-tree;

   4. Starting from root, traverse \( T_{c,S} \) in breadth first search order.
      Store the vertices and edges in \( T_{\text{root}} \).

   5. When arriving at a non-leaf terminal \( u \), then \( S_{\text{root}} \leftarrow \text{append}(u) \) and store \( u \) as a leaf in \( T_{\text{root}} \).

   6. if \( \text{alg} \neq 4 \) then

      7. \( T \leftarrow \text{append}(T_{\text{root}} \backslash \text{root}) \) // Store the tree-information depth-wise;

   else

      8. for \( u \in S_{\text{root}} \) \( \backslash \{\text{root}\} \) do

         9. \( S_u \leftarrow \{u, \text{root}\}; \)

         10. \( T_u \leftarrow \text{Path from } u \text{ to root} \) // Store the tree as if \( u \) is root and root is leaf;

         11. \( T \leftarrow \text{append}(T_u \backslash \text{root}) \);

      end

   end

12. return \( T \);

first subtree, so we can guarantee that the parity at each node gets corrected as desired.

Remark 1: The use of Steiner trees to take care of connectivity constraints was also done in [11] and [12]. Our procedures are different from both of them. While calling the procedure ROW-OP during the reduction to upper-triangular form (before transpose in step 11 in Algorithm 2) we skipped some steps (steps 6, 8, 20, and 22 in Algorithm 3) because it was not necessary and this reduced the CNOT count. We traverse each Steiner tree twice, so the number of CNOT gates required is approximately \( 2e \) where \( e \) is the number of edges in the tree. In contrast, the algorithm in [11] in this phase consumes approximately \( 4e \) CNOT gates. After transposing in step 11 in Algorithm 2 our procedure is markedly different from the approach taken in [11]. Even our “correction procedure” is different from the recursive approach taken in [12] for general graphs. Asymptotically the complexity of LINEAR-TF-SYNTH is similar to the corresponding algorithms in [11] and [12]. There are \( n \) Steiner trees constructed for each of the \( n \) columns. Each Steiner tree approximation will always be of size \( O(n) \). The number of CNOT gates applied is \( O(n) \). So overall complexity is \( O(n^2) \). An illustration of LINEAR-TF-SYNTH has been shown in [59], using an example given in [11]. We resynthesize a given linear transformation circuit using 26 CNOTs, while [11] used 43 CNOTs for resynthesizing the same circuit. We are not reproducing it here due to space constraints.

B. Synthesis of Circuits Over \( \{\text{CNOT, X, T}\} \)

We consider the circuits implemented with the set of gates \( G_{ph} = \{\text{CNOT, X, T, T*, S, S*, Y, Z}\} \). Lemma 1 can be applied and such a unitary can be characterized by a set \( \mathcal{P} = \{(c, f) : c \in \mathbb{Z}_2 \land f \in \mathcal{F}_{\mathbb{Z}_2}^+\} \) and linear reversible output function \( g \). Given such an \( n \)-qubit circuit with input path variables \( x_1, x_2, \ldots, x_n \), we can compute each \( P \) as follows: For each gate \( U \in \{T, T^*, S, S^*, Y, Z\} \) consider the parity, \( \bigoplus_{x_i \in [0, 1]} f_i \oplus b \), of the qubit just before \( U \) acts. Here, \( b \in \{0, 1\} \) is the bit variable that takes the value 1 only after an \( X \) or \( Y \) gate acts. This is represented by the function \( f \). The coefficient \( c \) is given by \{1, 7, 2, 6, 4, 4, 4, 4\}, respectively. For \((c_1, f_1), (c_2, f_2) \in \mathcal{P} \) if \( f_1 \neq f_2 \) then we can merge them into a single pair \((c_1 + c_2 \mod 8, f)\).

The linear reversible output function \( g : \mathbb{F}_2^n \times \mathbb{F}_2 \rightarrow \mathbb{F}_2^n \times \mathbb{F}_2 \) (one of the variables is the bit flip variable \( b \)) is represented by a matrix and procedures to synthesize circuits over \( \{\text{CNOT, X}\} \) that realize \( g \) has been given in Section III-A.

We follow the connectivity-oblivious approach taken in [13] while resynthesizing circuits over \( \{\text{CNOT, X, T}\} \). Given a phase polynomial set \( \mathcal{P} \) and matrix \( \mathbf{A} \) corresponding to the
linear reversible output function $g$, first a parity network (defined below) is synthesized that realizes the parity terms ($f$ where $(c, f) \in \mathcal{P}$) in $\mathcal{P}$. Then, single-qubit gates are applied depending on the coefficients ($c$) in $\mathcal{P}$. After that, a circuit is synthesized such that the overall linear transformation is $A$.

Definition 2 (Parity Network): A parity network for a set $\mathcal{P} = \{(c, f) : c \in \mathbb{Z}_2$ and $f \in \mathbb{F}_2^{m \times n}\}$ is an $n$-qubit circuit over $\{\text{CNOT}, X, T, H\}$ gates in which each parity term $f$ such that $(c, f) \in \mathcal{P}$ appears at least once.

Definition 3 (Phase Polynomial Network): A polynomial network for a set $\mathcal{P} = \{(c, f) : c \in \mathbb{Z}_2$ and $f \in \mathbb{F}_2^{m \times n}\}$ is an $n$-qubit circuit over $\{\text{CNOT}, X, T, H\}$ such that for each element $(c, f) \in \mathcal{P}$ the parity $f$ appears before a gate in $\{T, T^+, S, S^+, Z, Y\}$ when $c \in \{1, 2, 6, 4, 4\}$, respectively.

We now describe our algorithm PHASE-NW-SYNTH (Algorithm 5) that synthesizes a phase polynomial network given by $\mathcal{P}$. We construct the parity network matrix $\mathbf{P}$, which has $n$ rows corresponding to each qubit and where each column corresponds to a parity term $f$ in $\mathcal{P}$. Similar to [13], the optimization procedure to synthesize the parity network represented by $\mathbf{P}$ is inspired by Gray codes [60], which cycle through the set of $n$-bit strings using the exact minimum number of bit flips. Given a set $B$ of binary strings (step 4), we synthesize a parity network for $B$ by repeatedly choosing an index $j$ (step 25) to expand and then effectively recurring on the co-factors $B_0$ and $B_1$ (step 26), consisting of the strings $\mathbf{p} \in B$ with $p_j = 0$ or 1, respectively. As a subset $B$ is recursively expanded, CNOT gates are applied so that a designated target qubit $i$ contains the partial parity $\bigoplus_{k \in S^i} x_k$ where $S^i$ is the set of qubits (or row indices) such that $p_k = 1$ for $i$), for all $\mathbf{p} \in B$ (step 11). Whenever a column has a single 1, it implies that the corresponding parity has been realized. So we can remove these columns from the set $S$ (step 14) to find a minimum Steiner tree approximation with connectivity graph $G$ and terminals $S$. Let $i$ be the root of this tree, $T_i S$; $\mathbf{A} = \text{Matrix}$ with columns $\mathbf{p} \in B'$ such that $(b', i', j') \in \mathbf{A} \cup (B, i, j)$; $\mathbf{Y} = \emptyset$; $(\mathbf{Y}, \mathbf{A}) \leftarrow \text{ROW-OP}(\mathbf{A}, S, i, T_i S, 4)$; $\mathbf{Y}' = \emptyset$; $\mathbf{p}_k = 0$ for $k \in [n]$ and $k \neq i$ then At the place in the circuit where the parity given by $b'^{th}$ column of $\mathbf{P}$ is realized, place (append in $\mathbf{Y}'$) $U$ or $U$ (accordingly); $\mathbf{B}'$. delete($\mathbf{p}$).

To incorporate connectivity constraints we find a minimal Steiner tree $T_i S$ with connectivity graph $G$ and terminals $S = S^i \cup \{i\}$ (step 14). We call the procedure ROW-OP (Algorithm 3) with the matrix $\mathbf{A}$ such that its columns are the set of unrealized parities $\mathbf{Z}$. ROW-OP calls the subroutine SEPARATE 4 which as before separates $T_i S$ into edge-disjoint subtrees such that in each tree the root and leaves belong to set of terminals $S$. However, unlike the previous methods, this time each subtree with multiple leaves is further subdivided such that each tree has a single leaf. Each such tree is stored in reverse depth-first order such that the leaf becomes root and vice-versa (steps 9–11 in Algorithm 4). Now, when we perform steps 4–27 of Algorithm 3 then for each subtree the parity at root is $x_r \oplus x_{r'}$ where $r$ and $\ell$ are the root and leaf of the subtree, respectively (before flipping). Now since we process the trees from last subtree to first, so the net parity at node (or qubit $i$) is $\bigoplus_{k \in S^i} x_k$. To maintain the invariant that the remaining parities are expressed over the current state of the qubits, we modify the matrix $\mathbf{A}$ as given in step 26 of Algorithm 3.

Remark 2: In [11] an algorithm to synthesize parity networks over $\{\text{CNOT}, R_z\}$ was described and a somewhat similar scheme was sketched in [12]. Both used Steiner trees and the sum-over-path formulation of such circuits. Our algorithm is significantly different from both, especially considering the way we assigned CNOT gates according to the constructed Steiner trees. An illustration of PHASE-NW-SYNTH has been given in [59].

IV. SYNTHESIS OF CIRCUITS OVER $\{\text{CNOT, X, T, H}\}$

Finally, in this section, we are in a position to describe our resynthesis algorithms that takes as input a circuit $\mathcal{C}_I$ over a universal fault-tolerant gate set $\mathcal{G}_{\text{univ}} = \{\text{CNOT, T, T}', S, S^+, X, Y, Z, H\}$ and it outputs a circuit $\mathcal{C}_O$
with gates in the same set, but the position of the CNOT gates are restricted by some connectivity constraints imposed by an input connectivity graph $G$. The basic format of our resynthesis algorithms include slicing the given circuit and building the sliced portions. We partition the given circuit at the position of the $H$ gates and then sequentially resynthesize subcircuits in each portion such that the transformation within each portion and the overall circuit transformation remains unchanged. We investigate two methods of slicing—the first one is a simple slice-and-build, where we partition the input circuit according to the position of the $H$ gate and the second one is motivated by the Tpar algorithm given in [15], where the phase polynomial terms are partitioned. Unlike Tpar we are not interested in reducing the T-depth of the input circuit. So we partition the phase polynomial terms depending only on the path variables, which indicate the parities that can be synthesized before or after a certain $H$ gate.

A. Simply Slice-and-Build

In our first algorithm CNOT-OPT-A (Algorithm 6) we first partition the given circuit at the position of $H$ gates. Within each partition we initialize the state of the qubits $Q$ by the path variables $(x_1, x_2, \ldots, x_n)$ and the phase polynomial set $\mathcal{P}$ as empty set (step 4). Then with the application of each gate $U_i \in \mathcal{G}_{\text{init}}$ or $U_{(ij)} \in \mathcal{G}_{\text{init}}$, we update $\mathcal{P}$ and $Q$ (steps 5–7) by the function $\tilde{U}$; $(\mathcal{P}, Q) \rightarrow (\mathcal{P}', Q')$ as follows:

\[
\tilde{X}_i(\mathcal{P}, Q) = (\mathcal{P}, \langle q_1, \ldots, q_{i-1}, 1 \oplus q_i, \ldots, q_n \rangle),
\]

\[
\text{CNOT}_{\langle c, f \rangle}(\mathcal{P}, Q) = (\mathcal{P}', \langle q_1, \ldots, q_{i-1}, q_i \oplus q_{f, i}, q_{c, i}, q_{i+1}, \ldots, q_n \rangle),
\]

\[
\tilde{S}_i(\mathcal{P}, Q) = (\mathcal{P} \cup \{q_i, q_{i+1}, q_{i+2} \}), Q
\]

\[
\tilde{T}_i(\mathcal{P}, Q) = (\mathcal{P} \cup \{q_i, 1 \}), Q
\]

\[
\tilde{T}'_i(\mathcal{P}, Q) = (\mathcal{P} \cup \{q_{i+1}, 1 \}), Q
\]

\[
\tilde{Z}_i(\mathcal{P}, Q) = (\mathcal{P} \cup \{q_{i+1}, 1 \}), Q
\]

\[
\tilde{Y}_i(\mathcal{P}, Q) = (\mathcal{P} \cup \{q_{i+1}, 1 \}), (q_1, \ldots, q_{i-1}, 1 \oplus q_i, \ldots, q_n) \).
\]

In the above set of equations, for two sets $\mathcal{P}'$ and $\mathcal{P}''$, $\mathcal{P}' \cup \mathcal{P}'' = \{(c, f) : (c_1, f) \in \mathcal{P}'$, $(c_2, f) \in \mathcal{P}'', c = c_1 + c_2 \text{ mod } 8\}$.

Here, we assume that if $\exists k$ such that $(c, f) \notin \mathcal{P}$ for any $c = \{1, \ldots, 7\}$ and any set $\mathcal{P}$, then we say $(0, f) \in \mathcal{P}$.

Then, we synthesize the phase polynomial network $(C_{ph})$ for $\mathcal{P}$ (step 10) by invoking the procedure PHASE-NW-SYNT (Algorithm 5). We calculate the linear transformation $A$ (step 12) mapping $C_{ph}$ (state of the qubits after $C_{ph}$) to $Q$, which after steps 5–7 stores the state of the qubits at the end of the present slice. We synthesize the circuit $C_{lin}$ for $A$ (step 13) using the procedure LINEAR-TF-SYNT (Algorithm 2). We append the gates from $C_{ph}, C_{lin}$ followed by the $H$ gate (step 14). Then, we repeat the same steps for the next slice (till the next $H$ gate or the end of the given circuit).

B. Second Type of Slice-and-Build

In this section, we describe another way of slicing the given circuit, as described in procedure CNOT-OPT-B (Algorithm 7). Unlike CNOT-OPT-A, here we first compute some necessary information about the whole circuit and then between two $H$ gates we try to synthesize a circuit that computes part of the information. Similar to CNOT-OPT-A the transformations between two $H$ gates as well as the overall transformation remain unchanged. That is, given $C_I$, we first compute a triple $D = (\mathcal{P}, Q, H)$, where $\mathcal{P}$ is the phase polynomial set, $Q = \{q_1, q_2, \ldots, q_n\}$ represents the state of each qubit given

\[
15 \text{ end}
\]

\[
13 \text{ end}
\]

\[
11 \text{ end}
\]

\[
10 \text{ end}
\]
TABLE I

| Architecture | #Qubits | Initial Count | SWAP-template Overhead | CNOT-OPT-A Overhead | CNOT-OPT-B Overhead | Qiskit Time | TKET Overhead |
|--------------|--------|--------------|-------------------------|---------------------|---------------------|-----------|-------------|
| 9q-square    | 9      | 550%         | 0.00%                   | 0.18%               | 345%                | 310%      | 180%        |
|              |        | 612%         | 146%                    | 0.14%               | 0.00%               | 300%      | 218%        |
|              |        | 594%         | 105%                    | 0.16%               | 0.10%               | 303%      | 177%        |
|              |        | 546%         | 176%                    | 0.2%                | 0.15%               | 265.5%    | 154%        |
|              |        | 596%         | 184.6%                  | 0.23%               | 0.18%               | 310%      | 165.33%     |
| 16q-square   | 16     | 4050%        | 238%                    | 0.23%               | 768%                | 562.50%   | 465%        |
|              |        | 840%         | 146.25%                 | 0.27%               | 660%                | 461.25%   | 321.25%     |
|              |        | 817.50%      | 158.13%                 | 0.43%               | 864%                | 470.63%   | 306.88%     |
|              |        | 853%         | 340.63%                 | 0.41%               | 1213%               | 491.33%   | 309.38%     |
|              |        | 892.50%      | 220.78%                 | 0.49%               | 1259%               | 476.25%   | 276.09%     |
|              |        | 858.75%      | 210.63%                 | 0.57%               | 1156%               | 475.31%   | 276.8%      |
|              |        | 897.42%      | 237.5%                  | 0.72%               | 1306%               | 497.46%   | 276.6%      |
| rigetti-16q-aspen | 16 | 4       | 1680%                   | 355%                | 1278%               | 75%       | 645%        |
|              |        | 1740%        | 235%                    | 0.39%               | 1313%               | 738.75%   | 578.75%     |
|              |        | 1619.90%     | 351%                    | 0.47%               | 1304%               | 701.25%   | 454.38%     |
|              |        | 1794%        | 469.46%                 | 0.48%               | 1852%               | 732.19%   | 495.63%     |
|              |        | 1755%        | 399%                    | 0.66%               | 1900%               | 728.91%   | 476.25%     |
|              |        | 1760.63%     | 368.13%                 | 0.58%               | 1953%               | 731.02%   | 471.17%     |
|              |        | 1757.11%     | 410.9%                  | 0.61%               | 1962%               | 725.74%   | 463.44%     |
| ibm-qx5      | 16     | 4       | 1260%                   | 173%                | 988%                | 690%      | 547.5%      |
|              |        | 1035%        | 295%                    | 0.36%               | 1065%               | 712.50%   | 517.5%      |
|              |        | 1042.50%     | 283%                    | 0.41%               | 1226%               | 710.63%   | 469.38%     |
|              |        | 1179.38%     | 398.44%                 | 0.42%               | 1677%               | 670.31%   | 447.5%      |
|              |        | 1130.63%     | 339.06%                 | 0.45%               | 1733%               | 646.88%   | 441.56%     |
|              |        | 1120.94%     | 344.69%                 | 0.57%               | 1675%               | 689.30%   | 443.45%     |
|              |        | 1141.17%     | 379.88%                 | 0.73%               | 1792%               | 678.52%   | 434.1%      |
| ibm-q20-tokyo | 20 | 4       | 525%                    | 128%                | 418%                | 435%      | 315%        |
|              |        | 555%         | 275%                    | 0.29%               | 690%                | 506.25%   | 293.75%     |
|              |        | 570%         | 188%                    | 0.37%               | 663%                | 472.50%   | 257.5%     |
|              |        | 500.63%      | 154.38%                 | 0.55%               | 972%                | 420%      | 235.63%     |
|              |        | 542.81%      | 136.88%                 | 0.54%               | 1094%               | 463.50%   | 229.22%     |
|              |        | 539.53%      | 141.02%                 | 0.64%               | 1028%               | 468.05%   | 228.52%     |
|              |        | 534.61%      | 125.27%                 | 0.72%               | 1030%               | 457.50%   | 226.37%     |

as a function of the path variables and the bit flip variable $b \in \{0, 1\}$, and $H$ is an array of structures where the $i$th structure stores the state of the qubits before and after the application of the $i$th $H$ gate. The initial state of the qubits is $Q = (x_1, x_2, \ldots, x_n)$ (i.e., $q_i = x_i \ \forall i$). Both $P$ and $H$ are initialized as empty sets. With the application of each gate $U_i \in G_{\text{univ}}$ or $U_{i,j} \in G_{\text{univ}}$ (subscripts denote the qubit on which the gate acts) the triple $D$ gets updated by a function $\tilde{U}_i : D \rightarrow D$. Except for the $H$ gate, this function is similar to the function $\tilde{U}_i$ defined in Section IV-A. The array $H$ remains unchanged after the application of $U_i'$ for each gate except $H$. For the $H$ gate the function is defined as follows:

$$\tilde{H}_i(P, Q, H) = (P, Q', H'),$$

where $Q' = (q_1, \ldots, q_{i-1}, x_{n+j+1}, \ldots, q_n)$ and $H' = H \cup \{h_{j+1}\}$ such that $h_{j+1}.\text{Pos} = i$, $h_{j+1}.Q_I = Q$, $h_{j+1}.Q_O = Q'$, [Here $|H| = j$]. $h_{j+1}.\text{Pos}$ stores the qubit (which in the above equation is the $i$th one) on which the $(j+1)$th $H$ gate is applied. $h_{j+1}.Q_I$ and $h_{j+1}.Q_O$ store the state of the qubits before and after the application of the $(j+1)$th $H$ gate, respectively. We have introduced new path variables after application of each $H$ gate. We actually slice the sets $P, Q,$ and $H$ according to some conditions and synthesize circuits according to these slices.

For each $h \in H$ we calculate the set $P' = \{(c, f) \in P : f \in \text{span}(h.Q_I) \text{ but } f \notin \text{span}(h.Q_O)\} \subseteq P$ of parity terms that become uncomputable after placement of $H$ at qubit $h.\text{Pos}$ (step 6 in Algorithm 7). We express these parities in the basis given by the state of the qubits at the beginning of the current time slice, which is $Q_{\text{init}} = (x_1, \ldots, x_n)$ if $h$ is the first Hadamard gate, else it is $h'.Q_O$, the state of the qubits after the previous $H$ gate (step 9). We then calculate the phase polynomial network ($C_{\text{ph}}$) for the set $P_{\text{init}} (P')$ in the new basis, i.e., parity terms in $P'$ as function of $Q_{\text{init}}$ by invoking the procedure PHASE-NW-SYNTH (Algorithm 5). Let $Q_{\text{ph}}$ is the state of the qubits after $C_{\text{ph}}$ and $A$ is the linear...
C. Implementation and Results

We have considered the connectivity constraints imposed by some popular architectures, such as 9-qubit square grid (Fig. 1), 16-qubit square grid, Rigetti 16-qubit Aspen, 16-qubit IBM QX5, and 20-qubit IBM Tokyo (Fig. 2). We have worked with some benchmark circuits (Table II) and some randomly generated circuits on 9, 16, and 20 qubits (Fig. 1). The 9-qubit random input circuits have CNOT-count 3, 5, 10, 20, or 30, while both the 16 and 20-qubit random input circuits have CNOT-count 4, 8, 16, 32, 64, 128, or 256. For each of these groups we generated ten random circuits. We have compared the CNOT-count overhead obtained by using SWAP-template (Fig. 1), IBM’s Qiskit [17] and TKET [18] transpiler with the CNOT-count obtained from procedures CNOT-OPT-A (Algorithm 6) and CNOT-OPT-B (Algorithm 7). By overhead we mean the percentage increase in CNOT-count after taking into consideration connectivity constraints, i.e., (Final count - Initial count)/Initial count × 100%. The results for benchmark circuits and the random circuits have been tabulated in Tables I and II, respectively. All the simulations have been done in Java on a 3.1-GHz Dual-Core Intel Core i7 machine with 8-GB RAM and running MacOS Catalina 10.15.2. In Qiskit we ran each circuit with optimization level 10.15.2. In Qiskit we ran each circuit with optimization level

### Table II

| Architecture      | #Qubits | Benchmark | SWAP-template | CNOT-OPT-A | CNOT-OPT-B | Qiskit | TKET |
|-------------------|---------|-----------|---------------|------------|------------|--------|------|
|                   |         |           |               | Overhead   | Overhead   | Time   | Time |
| 9q-square         | 9       | bareco-tof-5 | 457.14%      | 245.24%    | 0.365s     | 140.48% | 0.52s | 232.14% | 110.71% |
|                   |         | grover-5   | 685.71%      | 116.67%    | 0.502s     | 105.36% | 0.84s | 219.64% | 94.05%  |
|                   |         | mod-ncr-55 | 752.73%      | 321.82%    | 0.31s      | 203.64% | 0.26s | 256.36% | 147.27% |
|                   |         | tof-5      | 465.31%      | 140.82%    | 0.27s      | 138.78% | 0.24s | 226.53% | 110.20% |
| 16q-square        | 16      | hw10       | 977.95%      | −63%       | 8.77s      | −57.67% | 7.25s | 284.76% | 133.10% |
|                   |         | rigetti-16a-aspen | 1508.63%   | −36.13%    | 8.8s       | −34.29% | 6.64s | 356.70% | 187.02% |
|                   |         | ibm-qx5    | 1099.84%     | −54.32%    | 6.18s      | −50.75% | 8.61s | 321.71% | 160.32% |
|                   |         | ibm-q20-tokyo | 571.44%    | −52.52%    | 0.72s      | −63.21% | 0.66s | 242.76% | 72.92%  |
| 20q-square        | 20      | ham15-high | 619.42%      | −77.58%    | 177.23s    | −74.92% | 231.14s | 277.82% | 87.81%  |

transform mapping $Q_{ph}$ to $h.Q_i$. To realize this transformation in this portion of the circuit we call the procedure LINEAR-TF-SYNTH (Algorithm 2). We append $(C_{ph}, C_{lin}, H[k])$ to the set of circuit gates, where $C_{lin}$ is the circuit returned by LINEAR-TF-SYNTH and $k = h.Pos$. After we process all the partitions till the last Hadamard gate we ensure that the complete phase polynomial set $P$ has been synthesized and the overall linear transformation of the output circuit maps $(x_1, x_2, \ldots, x_n)$ to $Q_{out}$, the final output of the circuit (which was calculated at the beginning while calculating $D$). For this we first synthesize the phase polynomial network $C_{ph}$ of any residual parity terms (step 21 in Algorithm 7). Then we calculate the residual transformation $A$ (step 26) that maps $Q_{ph}$, state of the qubits after $C_{ph}$, to $Q_{out}$. And, synthesize the circuit $C_{lin}$ (step 27) for $A$.

V. Conclusion

While implementing a quantum algorithm on an actual hardware, one needs to consider the different constraints imposed by the underlying physical architecture. One such constraint is the qubit connectivity, which is more concerning for multi-qubit gates such as CNOT. In a universal fault-tolerant gate set such as CNOT+T, although the T-gate is the most costly to implement fault-tolerantly, the CNOT-count is also important, especially in the NISQ era. In this work we have considered the problem of resynthesizing Clifford+T circuits with reduced CNOT-count compared to the SWAP-template, while perform quite well in the case of benchmark circuits. CNOT-OPT-A performs much better than the other algorithms in the case of random circuits. In CNOT-OPT-B, we compute the phase polynomial of the entire circuit and then do slicing. If we resynthesize a circuit from its phase polynomial then usually the gates which change the phase, like T-gate, gets reduced. This comes at the cost of increase in other gates like CNOT. Such observations have also been made in [15], though they do not consider connectivity constraints. In contrast, CNOT-OPT-A slices the entire circuit and then computes the phase polynomial of the segments. Although there is still some optimization of T-gates that may increase the CNOT-count, but this time it happens in a “more localized scale.” This can be one reason for a less significant overall increment of CNOTs, compared to CNOT-OPT-B. In fact, we think that any algorithm that synthesizes or resynthesizes using phase terms will tend to favor the reduction of gates like T. Hence, in CNOT-OPT-A this effect is mitigated to some extent by confining this reduction in shorter segments, leading to a better performance compared to CNOT-OPT-B.

Software packages like Qiskit and TKET have other built-in features like efficient initial qubit mapping, that aims to improve their performance. We have considered only one mapping, that is, logical qubit $i$ is mapped to vertex $i$ in the connectivity graph. If we change this mapping, it may be possible to get even better results. Hence, it is not completely fair for us to compare our results with these software packages. Despite such difference, our CNOT-OPT-A algorithm gives much less overhead compared to Qiskit and TKET in most cases.
respecting the connectivity constraint. Broadly, we have taken recourse to a slice-and-build approach, where we slice or partition the input circuit and resynthesize the slices with algorithms that use Steiner trees to place the CNOT gates. We have simulated benchmarks as well as random circuits on popular architectures, such as 9-qubit square grid, 16-qubit square grid, Rigetti 16-qubit Aspen, 16-qubit IBM QX5, and 20-qubit IBM Tokyo. Our results show that for both benchmarks and random circuits the simpler way of slicing the circuit (and not the phase polynomial) results in much less overhead in terms of increase in CNOT-count, compared to the overhead obtained by using SWAP-template, Qiskit, and TKET tranpsiler.

**ACKNOWLEDGMENT**

The ideas were given by P. Mukhopadhyay. The implementations were done by S. M. Li and J. Huang. All the authors contributed in preparation of the this article. The results obtained in this article are available at https://github.com/SarahMLI/cnotcount. The code is available from the corresponding author on reasonable request.

**REFERENCES**

[1] P. W. Shor, “Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer,” *SIAM Rev.*, vol. 41, no. 2, pp. 303–332, 1999.

[2] R. P. Feynman, “Simulating physics with computers,” *Int. J. Theor. Phys.*, vol. 21, no. 6–7, pp. 467–488, 1982.

[3] L. K. Grover, “Quantum mechanics helps in searching for a needle in a haystack,” *Phys. Rev. Lett.*, vol. 79, pp. 325–328, Jul. 1997.

[4] J. Preskill, “Quantum computing in the NISQ era and beyond,” *Quantum*, vol. 2, no. 7, Aug. 2018.

[5] M. Reagor et al., “Demonstration of universal parametric entangling gates on a multi-qubit lattice,” *Sci. Adv.*, vol. 4, no. 2, p. eaao3603, 2018.

[6] R. Versluis et al., “Scalable quantum circuit and control for a superconducting surface code,” *Phys. Rev. Appl.*, vol. 8, no. 3, 2017, Art. no. 34021.

[7] J. W. Britton et al., “Engineered two-dimensional ising interactions in a trapped-ion quantum simulator with hundreds of spins,” *Nature*, vol. 484, no. 7395, pp. 489–492, 2012.

[8] C. J. Ballance, T. P. Harty, N. M. Linke, M. A. Sepiol, and D. M. Lucas, “High-fidelity quantum logic gates using trapped-ion hyperfine qubits,” *Phys. Rev. Lett.*, vol. 117, no. 6, 2016, Art. no. 60504.

[9] J. P. Gaebler et al., “High-fidelity universal gate set for $^{13}$Be$^+$ ion qubits,” *Phys. Rev. Lett.*, vol. 117, no. 6, 2016, Art. no. 60505.

[10] W. K. Hensinger et al., “T-junction ion trap array for two-dimensional ion shuffling, storage, and manipulation,” *Appl. Phys. Lett.*, vol. 88, no. 3, 2006, Art. no. 34101.

[11] B. Nash, V. Gheorghiu, and M. Mosca, “Quantum circuit optimizations for NISQ architectures,” *Quantum Sci. Technol.*, vol. 5, no. 2, 2020, Art. no. 25010.

[12] A. Kissinger and A. M. van de Griend, “CNOT circuit extraction for topologically-constrained quantum memories,” *Quantum Inf. Comput.*, vol. 20, nos. 7–8, pp. 581–596, 2020.

[13] M. Amy, P. Azimzadeh, and M. Mosca, “On the controlled-NOT complexity of controlled-NOT-phase circuits,” *Quantum Sci. Technol.*, vol. 4, no. 1, 2018, Art. no. 15002.

[14] K. N. Patel, I. L. Markov, and J. P. Hayes, “Optimal synthesis of linear reversible circuits,” *Quantum Inf. Comput.*, vol. 8, no. 3, pp. 282–294, 2008.

[15] M. Amy, D. Maslov, and M. Mosca, “Polynomial-time T-depth optimization of Clifford+$T$ circuits via matroid partitioning,” *IEEE Trans. Computer-Aided Design Integr. Circuits Syst.*, vol. 33, no. 10, pp. 1476–1489, Oct. 2014.

[16] T. G. de Bruijère, M. Baboulin, B. Valiron, S. Martiel, and C. Allauche, “Quantum CNOT circuits synthesis for NISQ architectures using the syndrome decoding problem,” in *Proc. Int. Conf. Reversible Comput.*, 2020, pp. 189–205.
[44] D. Venturelli, M. Do, E. Rieffel, and J. Frank, “Compiling quantum circuits to realistic hardware architectures using temporal planners,” *Quantum Sci. Technol.*, vol. 3, no. 2, 2018, Art. no. 25004.

[45] P. Murali, A. Javadi-Abhari, F. T. Chong, and M. Martonosi, “Formal constraint-based compilation for noisy intermediate-scale quantum systems,” *Microprocess. Microsyst.*, vol. 66, pp. 102–112, Apr. 2019.

[46] A. Pater, A. Zaleleh, and R. Wille, “NISQ circuit compilation is the travelling salesman problem on a torus,” *Quantum Sci. Technol.*, vol. 6, no. 2, 2021, Art. no. 25016.

[47] D. Ferrari and M. Amoretti, “Efficient and effective quantum compiling for entanglement-based machine learning on IBM Q devices,” 2018, arXiv:1801.02363.

[48] Y. Tang, “Efficient CNOT synthesis for NISQ devices,” 2020, arXiv:2011.06760.

[51] A. Montanaro, “Quantum circuits and low-degree polynomials over \( \mathbb{F}_2 \),” *J. Phys. A, Math. Theor.*, vol. 50, no. 8, 2017, Art. no. 84002.

[52] C. M. Dawson, A. P. Hines, D. Mortimer, H. L. Haselgrove, M. A. Nielsen, and T. J. Osborne, “Quantum computing and polynomial equations over the finite field \( \mathbb{F}_2 \),” *Quantum Inf. Comput.*, vol. 5, no. 2, pp. 102–112, 2005.

[53] D. E. Koh, M. D. Penney, and R. W. Spekkens, “Quantum circuits and low-degree polynomials over \( \mathbb{F}_2 \),” *Quantum Inf. Comput.*, vol. 17, nos. 13–14, pp. 1081–1095, 2017.

[54] R. M. Karp, “Reducibility among combinatorial problems,” in *Complexity of Computer Computations*. Boston, MA, USA: Springer, 1972, pp. 85–103.

[55] K. Hwang and D. S. Richards, “Steiner tree problems,” *Networks*, vol. 22, no. 1, pp. 55–89, 1992.

[56] G. Robins and A. Zelikovsky, “Tighten bounds for graph Steiner tree approximation,” *SIAM J. Discrete Math.*, vol. 19, no. 1, pp. 122–134, 2005.

[57] J. Byrka, F. Grandoni, T. Rothvoss, and L. Sanità, “Steiner tree approximation via iterative randomized rounding,” *J. ACM*, vol. 60, no. 1, pp. 1–33, 2013.

[58] D. Sadeghi and H. Fröhlich, “Steiner tree methods for optimal subnetwork identification: An empirical study,” *BMC Bioinform.*, vol. 14, no. 1, p. 144, 2013.

[59] V. Gheorghiu, J. Huang, S. M. Li, M. Mosca, and P. Mukhopadhyay, “Reducing the CNOT count for Clifford+T circuits on NISQ architectures,” 2020, arXiv:2011.12191.

[60] G. Frank, “Pulse code communication,” U.S. Patent 2 632 058, Mar. 17, 1953.

Jiaxin Huang is currently pursuing the B.Sc. degree in pure mathematics, with a minor degree in physics in computer science with the Hong Kong University of Science and Technology, Hong Kong.

He was an exchange student with the University of Waterloo, Waterloo, ON, Canada, in 2022. His current research interests include graphical quantum processes, quantum circuit optimization, and variational quantum algorithms.

Mr. Huang was awarded the HKSAR Government Reaching Out Award in 2022.

Sarah Meng Li (Member, IEEE) received the B.Sc. degree (First Class Hons.) in mathematics and computer science with a minor in statistics from Dalhousie University, Halifax, NS, Canada, in 2021. She is currently pursuing the master’s degree in mathematics with the Institute for Quantum Computing, University of Waterloo, Waterloo, ON, Canada.

In her undergraduate research, she studied families of Clifford+T circuits from a number-theoretic perspective. Her graduate study focuses on quantum error correction and fault-tolerant quantum computing. Beyond that, she is interested in quantum circuit synthesis and optimization.

Michele Mosca received the Doctoral degree in mathematics, on the topic of quantum computer algorithms from the University of Oxford, Oxford, U.K., in 1999.

He started working in cryptography during his undergraduate studies. He is the Co-Founder of the Institute for Quantum Computing, University of Waterloo, Waterloo, ON, Canada, a Professor with the Department of Combinatorics and Optimization, Faculty of Mathematics, and a Founding Member of Waterloo’s Perimeter Institute for Theoretical Physics. He is the Co-Founder and the CEO of the quantum-safe cybersecurity company, evolutionQ, Kitchener, ON, Canada, and the Co-Founder of the quantum software and applications company, softwareQ Inc., Waterloo.

His research interests include algorithms and software for quantum computers, and cryptographic tools designed to be safe against quantum technologies.

Prof. Mosca’s awards and honours include 2010 Canada’s Top 40 Under 40, the Queen Elizabeth II Diamond Jubilee Medal in 2013, the SIU Fr. Norm Chatte Lifetime Achievement Award in 2017, and the Knighthood (Cavaliere) in the Order of Merit of the Italian Republic in 2018. He serves as Chair of the board of Quantum Industry Canada. He co-founded the not-for-profit Quantum-Safe Canada, and the ETSI-IQC workshop series in quantum-safe cryptography and is globally recognized for his drive to help academia, industry and government prepare our cyber systems to be safe in an era with quantum computers.

Priyanka Mukhopadhyay received the Ph.D. degree from the Centre for Quantum Technologies, National University of Singapore, Singapore, in 2018.

She is a Postdoctoral Fellow with the Institute for Quantum Computing and Department of Combinatorics and Optimization, University of Waterloo, Waterloo, ON, Canada. Her current research interests include quantum circuit synthesis and optimization, quantum algorithms, Hamiltonian simulation, algorithms for quantum chemistry and quantum error correction and fault tolerance.