Architecture-aware synthesis of phase polynomials for NISQ devices

Arianne Meijer-van de Griend[1,*]   Ross Duncan[1,2,†]

[1] *Cambridge Quantum Computing Ltd*
9a Bridge Street, Cambridge, United Kingdom
[2] *Department of Computer and Information Sciences*
University of Strathclyde
26 Richmond Street, Glasgow, United Kingdom

We propose a new algorithm to synthesise quantum circuits for phase polynomials, which takes into account the qubit connectivity of the quantum computer. We focus on the architectures of currently available NISQ devices. Our algorithm generates circuits with a smaller CNOT depth than the algorithms currently used in Staq and t|ket⟩, while improving the runtime with respect the former.

1   Introduction

Many current quantum computing architectures have restricted qubit connectivity, meaning that interactions between qubits are only possible when the physical qubits are adjacent in a certain graph, henceforth called the *architecture*, defined by the design of the quantum hardware. Traditional compiling techniques for quantum circuits work around this limitation by inserting additional SWAP gates into the circuit to move the logical qubits into a location where the desired interaction is physically possible, a process called *routing* or *mapping* [6, 16, 19, 17]. This typically increases the depth and gate count of the circuit by a multiplicative factor between 1.5 and 3 [6]. However, recent work by Kissinger and Meijer-van de Griend [11] has shown that for pure CNOT circuits it is possible to compile a circuit directly to an architecture without dramatically increasing the number of CNOT gates. Their approach was to use a higher-level representation of the desired unitary transform and (re)synthesise the corresponding circuit in an architecture-aware manner.

In this paper, we consider another class of high-level constructs called *phase polynomials*, which give rise to circuits containing only CNOT and $R_Z(\theta)$ gates. The current state-of-the-art algorithm for phase polynomial synthesis is the GraySynth algorithm [1]. Unlike other algorithms for phase polynomial synthesis [3], GraySynth attempts to minimise the number of 2-qubit gates. Unfortunately, GraySynth assumes unrestricted qubit connectivity. This limitation was removed by Nash et al. [12], by adding qubit permutation subcircuits whenever a sequence of CNOTs required by GraySynth is not permitted by the architecture. Nevertheless, the algorithm still relies on the same recursive strategy as GraySynth, which might be suboptimal for sparse architectures.

---

[*]ariannemeijer@gmail.com

[†]ross.duncan@cambridgequantum.com

In this paper we propose a new algorithm for the architecture-aware synthesis of phase polynomial circuits. The algorithm has been tuned for the relatively sparse connectivity graphs of current quantum computers.

We compare our algorithm against two compilers that are able to natively synthesise phase polynomials: $\mathsf{t|ket\rangle}$ [15] and Staq [2]. We compare the different methods based on the final CNOT count, final CNOT depth, and their runtime. These figures of merit are appropriate for noisy-intermediate scale quantum (NISQ) devices [14], since the single-qubit gates of such devices typically have error rates an order of magnitude less than that of the two qubit gates. By minimising the CNOT count we are minimising the exposure of our computation to gate error, including crosstalk; by minimising depth we reduce its exposure to decoherence.

We show that for sufficiently sparse quantum computer architectures and sufficiently large phase polynomials, our algorithm outperforms the algorithm from Nash et al. [12] that is used in Staq [2] as well as the decomposition and routing strategies from $\mathsf{t|ket\rangle}$ [6]. Our algorithm relies on finding non-cutting vertices in the connectivity graph, and does not require computing any Steiner trees; we find that in most cases our algorithm has reduced runtime compared to that of Nash et al.

In section 2, we introduce phase polynomials and existing methods for their synthesis, both with and without architecture-awareness. Our new algorithm is described in section 3 and our experimental results can be found in section 4. Throughout the paper we will assume some familiarity with the zx-calculus [4], which we use as notation. For the uninitiated, Cowtan et al. [7] give a short introduction to the calculus, including the phase gadget notation; Coecke and Kissinger provide a complete treatment [5].

Notation   We use bold face letters $\boldsymbol{x}$, $\boldsymbol{y}$, to denote vectors, and the corresponding regular weight letters x_i, y_j to denote their components.

2   Phase polynomial synthesis

Following Amy et al. [1], we define the phase polynomial via the sum-over-paths formalism [8].

Definition 2.1. Let C be a circuit consisting of only CNOT and $R_Z(\theta)$ gates; then its corresponding unitary matrix U_C has a *sum-over-paths* form,

$$U_C = \sum_{\boldsymbol{x} \in \mathbb{F}_2^n} e^{2\pi i f(\boldsymbol{x})} |A\boldsymbol{x}\rangle \langle \boldsymbol{x}| \tag{1}$$

consisting of a *phase polynomial*

$$f(\boldsymbol{x}) = \sum_{\boldsymbol{y} \in \mathbb{F}_2^n} \hat{f}(\boldsymbol{y}) \cdot (x_1 y_1 \oplus x_2 y_2 \oplus \cdots \oplus x_n y_n) \tag{2}$$

with *Fourier coefficients* $\hat{f}(\boldsymbol{y}) \in \mathbb{R}$, and a *basis transform* $A \in GL(n, \mathbb{F}_2)$. When no confusion will arise we refer to the pair (f, A) as the phase polynomial of C.

Note that phase functions – henceforth just called *parities* – of the form $\boldsymbol{x} \mapsto (x_1 y_1 \oplus \cdots \oplus x_n y_n)$ as in Equation 2, can be identified with the bit string $\boldsymbol{y}$; these are the basis of the space of phase polynomials. Those parities for which $\hat{f}(\boldsymbol{y}) \neq 0$ are called the *support* of f.

Every circuit over $\{\text{CNOT}, R_Z(\theta)\}$ has a canonical sum-over-paths form, which we now sketch. First, we associate a parity to each "wire segment" of the circuit as follows: the inputs of the circuit are labelled $x_1, \ldots, x_n$ respectively; the output of an R_Z gate has the same parity as its input; and a CNOT gate with parities p_1 and p_2 on its control and target inputs has output parities p_1 and $p_1 \oplus p_2$, respectively. Second, the coefficients $\hat{f}(\boldsymbol{y})$ are computed by summing all the angles θ occurring in R_Z gates labelled by the parity $\boldsymbol{y}$. Finally, the linear transform A is defined by the mapping $\boldsymbol{x} \mapsto \boldsymbol{x'}$ where $\boldsymbol{x'}$ are the final labels of circuit outputs. We refer the reader to Amy et al. [1] for more details.

The task of *phase polynomial synthesis* is the reverse: given (f, A) we must find the circuit C. This amounts to constructing a parity labelled CNOT circuit such that every $\boldsymbol{y}$ in the support of f occurs as a label on some wire, adding an $R_Z(\hat{f}(\boldsymbol{y}))$ gate on that wire, and extending that circuit so that the desired output parities for A are achieved. Since $f(\boldsymbol{x})$ is a sum, and addition is commutative, the order in which the parities are achieved is irrelevant; neither does it matter on which qubits these parities occur. To obtain the required final parities, additional CNOTs are added to the circuit. Since the new parity is the sum of the parities of both the control and the target qubit, applying a CNOT gate can therefore be seen as an elementary row operation on the matrix $\boldsymbol{x} \mapsto \boldsymbol{x'}$. If the desired parities for each qubit are known, Gaussian elimination can produce a CNOT sequence to achieve those parities [13, 11, 12]. This method suffices to synthesise the matrix A of the phase polynomial [3, 1, 12]; note, however, that this second phase is totally independent of the earlier synthesis of the parities required for $f(\boldsymbol{x})$.

Architecture agnostic synthesis. Phase polynomials may be synthesised via the *phase gadget* construct of the zx-calculus [7]. Since our algorithm can be intuitively described using phase gadgets, we will briefly explain this method.

Definition 2.2. In zx-calculus notation we denote the R_Z gate with phase α and CNOT gate as:

In a phase polynomial (f, A), each term in $f(\boldsymbol{x})$ defines an operator $e^{-i\frac{\alpha}{2}Z^{\otimes n}}$, which we represent by the *phase gadget* $\Phi_n(\alpha)$:

where $\alpha = \hat{f}(\boldsymbol{y})$ and the gadget is connected to qubit i iff $x_i y_i = 1$.

Lemma 2.3. *We have the following law for decomposition of phase gadgets [7].*

Lemma 2.3 serves as a recursive definition of the phase gadget, and demonstrates how the gadget may be realised as two ladders of CNOTs and an R_Z gate. Cowtan et al. [7] showed how to synthesise phase gadgets in reduced depth using a balanced tree of CNOTs, however if the

gadgets are synthesised singly, and their ordering is not taken into account, the circuit may still be suboptimal even after local optimisation.

A consequence of Lemma 2.3 is that phase gadgets stabilise CNOT circuits in the following sense. Let C_{ij} be a CNOT gate with control qubit i and target qubit j; then for all phase gadgets $\Phi(\alpha)$ there exists $\Phi'(\alpha)$ such that $C_{ij}\Phi(\alpha) = \Phi'(\alpha)C_{ij}$. Φ' is identical to Φ except that Φ' is connected qubit i iff Φ is connected to exactly one of i and j.

This observation leads to an improvement in the algorithm. If we view the sequence of phase gadgets as a binary matrix whose rows are the qubits and whose columns are the corresponding parities $\boldsymbol{y}$ in the support of f, then commuting C_{ij} through the entire circuit is an elementary row operation, namely adding row j to row i. Therefore, by conjugating the circuit with CNOTs, we may obtain a column containing a single 1. At that point, the desired parity (corresponding to the column in the matrix) is achieved on the qubit corresponding to the row with the 1. The R_Z gate can then be placed, and the column can be removed from the matrix.

For example, the 3 qubit phase polynomial, $(f(\boldsymbol{x}),\, I)$, specified by $f(\boldsymbol{x}) = \alpha_1(x_2 \oplus x_3) + \alpha_2(x_1 \oplus x_2) + \alpha_3(x_1 \oplus x_3) + \alpha_4 x_3$, can be represented in a zx-diagram and corresponding binary matrix as:

$$\sim \begin{pmatrix} 0 & 1 & 1 & 0 \\ 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 \end{pmatrix}$$

Conjugating the first and second qubits with two CNOTs, and applying Eq. 3 we obtain the following rewrite sequence and final matrix:

$$\sim \begin{pmatrix} 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 \end{pmatrix}$$

The second and last columns of the matrix contain only a single 1, so we can use Equation 4 to place a R_Z gate:

$$\sim \begin{pmatrix} 1 & 1 \\ 1 & 0 \\ 1 & 1 \end{pmatrix}$$

Note the equation relies on the fact that R_Z gates commute with phase gadgets.

The matrix representation reduces the task of phase polynomial synthesis to finding the order in which to reduce the columns, and which qubit should remain a 1 in the matrix for each column. Amy et al. [1] proposed a heuristic algorithm called *GraySynth* based on Gray codes. The main idea is to pick the qubit q participating in most parities and then achieving all parities containing q in order of Gray codes [9] on qubit q. As a result, many CNOTs will have the same target qubit. This algorithm has been implemented as part of Staq [2] in combination with SWAP-based routing.

Unfortunately, GraySynth does not accommodate qubit connectivity restrictions, making it less useful for NISQ devices. A naive solution is to apply a generic qubit routing routine to the synthesised circuit, however this will almost always increase the size of the circuit. Luckily, there is no need to be so naive.

Architecture-aware synthesis. It is possible to define synthesis algorithms which produce circuits that immediately satisfy the constraints imposed by the quantum computer. Several algorithms such *architecture-aware synthesis* algorithms for CNOT circuits and phase polynomials have recently been proposed [11, 12]. While SWAP-based methods respect the original structure of the circuit at the level of individual gates, architecture-aware synthesis preserves only the overall unitary, and this additional freedom allows the architectural constraints to inform the choice of which gates to generate. This concept has also been used in the Staq compiler [2], which uses the algorithms described in this section.

Kissinger et al. [11] and Nash et al. [12] independently modified the Gaussian elimination algorithm sketched above to synthesise routed CNOT circuits. They used Steiner trees to determine paths on the connectivity graph across which to simulate one or more CNOT gates. Nash et al. [12] showed that their method scales well with respect to the size and the density of the connectivity graph of the quantum computer. Kissinger et al. [11] showed that for circuits consisting only of CNOT gates their method outperformed current state-of-the-art SWAP-based methods. Wu et al. [18] have recently improved these methods with an adaptation relying on Steiner trees and non-cutting vertices.

This constrained version of Gaussian elimination, called *Steiner-Gauss*, can be used in any synthesis algorithm by replacing the original Gaussian elimination such that it routes (part of) the synthesised circuit. In particular, this can be used in the T-par algorithm [3] and in GraySynth it can be used to synthesise the matrix *A*.

Nash et al. [12] also proposed an adaptation of the GraySynth algorithm we called *Steiner-GraySynth*. They replaced the step in the original GraySynth algorithm that generates a small sequence of CNOTs with a step that emulates this sequence with routed CNOTs. This emulation is created using a Steiner tree over the connectivity graph with the phase qubit as root and the other qubits participating in the sequence of CNOTs as nodes. Then, a CNOT is placed for every Steiner-node in the tree and one for every edge in the Steiner tree.

For phase polynomial synthesis, this algorithm performs better than naive routing [12]. However, following GraySynth, it will place many CNOT gates with the same target qubit. If this qubit is poorly connected in the architecture, a large CNOT overhead will result. Furthermore, it requires the construction of a Steiner tree in order to route the CNOT gates. The minimal Steiner tree problem is NP-hard[10], so finding the true optimum is not feasible, but it can be approximated in polynomial time using the all-pairs shortest paths and building a spanning tree between them.

3 New natively routed heuristic algorithm

In this section, we describe a natively routed algorithm that attempts to take the architecture into account. It uses a novel heuristic which works well for sparse architecture graphs.

Pseudo-code for the algorithm is shown in Figure 1 and its sub-procedures are listed in Appendix A. A full worked example is presented in Appendix B; for ease of comparison this

global variables
  G, the architecture graph
  Circuit, An initially empty circuit with $|G.\text{vertices}|$ qubits
  A, The basis transform of the phase polynomial
  P, The matrix describing the support of f
  ZPhases, The list of Z phases $\hat{f}(\boldsymbol{y})$ belonging to each parity $\boldsymbol{y}$ in f
end global variables

function BASERECURSIONSTEP(*Cols*, *Qubits*)
  if *Qubits* non-empty and *Cols* non-empty **then**
    $H \leftarrow \text{InducedSubgraph}(G, Qubits)$
    $Rows \leftarrow \text{NonCuttingVertices}(H)$
    $ChosenRow \leftarrow \text{argmax}_{r \in Rows} \max_{x \in \mathbb{F}_2} |\{c \in Cols \text{ where } P_{r,c} = x\}|$
    $Cols0, Cols1 \leftarrow \text{SplitColsOnRow}(Cols, ChosenRow)$
    BaseRecursionStep($Cols0$, $Qubits \setminus \{ChosenRow\}$)
    OnesRecursionStep($Cols1$, $Qubits$, $ChosenRow$)
  end if
end function

function ONESRECURSIONSTEP(*Cols*, *Qubits*, *ChosenRow*)
  if *Cols* non-empty **then**
    $Neighbours \leftarrow \{q \in Qubits \text{ where } q \sim ChosenRow \text{ in } G\}$
    $n \leftarrow \text{argmax}_{q \in Neighbours} |\{c \in Cols \text{ where } P_{q,c} = 1\}|$
    if $|\{c \in Cols \text{ where } P_{n,c} = 1\}| > 0$ **then**
      PlaceCNOT ($ChosenRow$, n)
      $Cols \leftarrow \text{ReduceColumns}(Cols)$
    else
      PlaceCNOT (n, $ChosenRow$)
      PlaceCNOT ($ChosenRow$, n)
    end if
    $Cols0, Cols1 \leftarrow \text{SplitColsOnRow}(Cols, ChosenRow)$
    BaseRecursionStep($Cols0$, $Qubits \setminus \{ChosenRow\}$)
    OnesRecursionStep($Cols1$, $Qubits$, $ChosenRow$)
  end if
end function

algorithm ROUTEDPHASEPOLYSYNTH
  $Columns \leftarrow \text{ReduceColumns}(\{0, \ldots, |P.\text{columns}|\})$
  BaseRecursionStep($Columns$, $G.\text{vertices}$)
  $Circuit.\text{AddGates}(\text{SteinerGauss}(A * P'^{-1}))$
end algorithm

Figure 1: Algorithm for synthesising phase polynomials in an architecture aware manner. The subroutines not defined here are in listed in Appendix A.

example is the same one treated by Amy et al. [1] using the GraySynth algorithm.

In the following, the architecture graph – that is, the connectivity map of the physical qubits – is denoted G. The phase polynomial to be synthesised, (f, A), is represented as two binary matrices, P and A, where the columns of P are the corresponding parities $\boldsymbol{y}$ in the support of f, as explained in Section 2. By construction, the columns in P are unique and no column $\boldsymbol{y}$ has all values set to 0.

Preprocessing. The algorithm starts by synthesising phase gadgets of the form specified by Equation 4. This will remove trivial columns in P and placing their corresponding R_Z phase gates. A column $\boldsymbol{y}$ is trivial if it has exactly one index j such that $y_j = 1$. The phase gate R_Z is placed on the qubit corresponding to j and its phase α is equal to $\hat{f}(\boldsymbol{y})$. This makes sure that every column $\boldsymbol{y}$ in P contains at least two elements with value 1. Hence, each column requires at least one CNOT in order to be synthesised by Lemma 2.3.

For example, consider the phase polynomial from Section 2, we can use Equation 4 to remove the fourth column (corresponding to α_4) and synthesise the phase gate $R_Z(\alpha_4)$ on qubit 3:

$$\begin{pmatrix} 0 & 1 & 1 & 0 \\ 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 \end{pmatrix} \sim \quad = \quad \sim \begin{pmatrix} 0 & 1 & 1 \\ 1 & 1 & 0 \\ 1 & 0 & 1 \end{pmatrix}$$

Base recursion step. Similar to GraySynth, we want to synthesise the phase gadgets in the phase polynomial in an order that requires the least amount of CNOT gates. However, we do not want to synthesise the phase gadgets such that many phase gates are placed on the same qubit. Instead, we pick one qubit and attempt to remove its row from P. However, we cannot pick just any row to remove from P because it might still be needed to synthesise other phase gadgets due to the connectivity constraints. Thus, we pick a non-cutting vertex $i \in G$ such that row P_i has either the most ones or the most zeroes. A *non-cutting vertex* is a vertex in G that can be removed from G without disconnecting the remaining graph. Like GraySynth, we split P into two matrices, P^0 and P^1, such that column P_j is a column in P^0 iff $P_{i,j} = 0$ and P_j is a column in P^1 otherwise. Since all entries in row P_i^0 are equal to 0, we do not need this row any more and we can remove it from P^0, and because i is non-cutting, its removal leaves the graph connected. Then, we use the base recursion step on the sub-matrix P^0 (excluding row i) with the sub-graph of G where vertex i has been removed. The matrix P^1 is treated by a different recursive procedure using the full graph G, described below.

Continuing the example above, suppose we are targeting the architecture $G : x_1 \Leftrightarrow x_2 \Leftrightarrow x_3$. We can pick either x_1 or x_3 as they are both non-cutting and have the same number of ones and zeroes; we will make the arbitrary choice of x_1. This choice yields our new P^0 and P^1:

$$P = \begin{pmatrix} 0 & 1 & 1 \\ 1 & 1 & 0 \\ 1 & 0 & 1 \end{pmatrix} \qquad P^0 = \begin{pmatrix} 1 \\ 1 \end{pmatrix} \qquad P^1 = \begin{pmatrix} 1 & 1 \\ 1 & 0 \\ 0 & 1 \end{pmatrix}$$

Note that P^0 corresponds to the phase gadget α_1, and P^1 corresponds to the phase gadgets α_2 and α_3. Recursing on P^0 will eventually place the CNOT $C_{3,2}$ and $R_Z(\alpha_1)$ gate on qubit x_2, as

shown below.

$$\begin{pmatrix} 0 & 1 & 1 \\ 1 & 1 & 0 \\ 1 & 0 & 1 \end{pmatrix} \sim \quad = \quad = \quad \sim \begin{pmatrix} 1 & 1 \\ 1 & 0 \\ 1 & 1 \end{pmatrix}$$

Bear in mind that the recursion on P^0 may add CNOTs to the circuit, performing a row operation on the global P matrix. For our recursion scheme to be valid we require that the row P_i^1 remains equal to 1. Initially, this holds by the construction of P^1. Since row i has been removed from P^0, no gate involving qubit i will be added by recursion on P^0, and hence the ith row of P^1 will be unchanged. Moreover, P^1 does not contain any trivial columns because for every column j in P^1 there will always be another row $k \neq i$ such that $P_{k,j}^1 = 1$.

Ones recursion step. The recursion step for P^1 attempts to remove as many ones from row i as possible such that it can be removed. This can be achieved by placing CNOTs in the circuit, however we are restricted by the connectivity graph. Therefore, we pick a neighbour vertex $n \in G$ such that row P_n^1 has most ones. Picking the row P_n^1 with most ones will ensure that most ones are removed. Then, we can conjugate with CNOT $C_{i,n}$, and update P by adding row n to row i, as explained in Section 2. This might introduce trivial columns in P^1 (note that $P^0 = \emptyset$ by the recursion), which are removed like in the preprocessing step. Thus, in the example circuit:

$$\begin{pmatrix} 1 & 1 \\ 1 & 0 \\ 1 & 1 \end{pmatrix} \sim \quad = \quad \sim \begin{pmatrix} 0 & 1 \\ 1 & 0 \\ 1 & 1 \end{pmatrix}$$

However, if every entry of row P_n^1 is 0, conjugation with $C_{i,n}$ will have no effect. In this situation, we first apply the opposite CNOT, $C_{n,i}$ and then $C_{i,n}$ as before. This effectively swaps the rows i and n, so there is no need to reduce the circuit. Nevertheless, this ensures that every entry in row P_i^1 is 0.

After placing the CNOT gate(s), we have modified row P_i^1 and we can split P^1 into two matrices, $P^{1,0}$ and $P^{1,1}$, and recurse upon these two as in the base recursion step.

In our example $P^{1,0} \sim \{\alpha_2\}$ and $P^{1,1} \sim \{\alpha_3\}$. We use the base recursion on $P^{1,0}$ and pick x_3 arbitrarily. Note that we only consider the sub-matrix $P^{1,0}$ to count the number of ones and zeroes. Then, we split $P^{1,0}$ into $\emptyset$ and $\{\alpha_2\}$, respectively. In the ones recursion step, we pick neighbour x_2 and place CNOT $C_{3,2}$ and $R_Z(\alpha_2)$ on qubit x_2.

$$= \quad = \quad \sim \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix}$$

Afterwards, we use the ones recursion step twice on the remaining row, placing two CNOTs, $C_{2,1}$ and $C_{3,2}$, and placing the final phase gate $R_Z(\alpha_3)$ on qubit x_3

$$= \quad =$$

Post-processing. Lastly, we need to synthesise the basis transform A. Because the CNOT gates in the circuit, obtained by synthesising the phase gadgets, change the parities on each qubit, we need to undo these changes. Let P' be the basis transform corresponding to the final parities of the synthesised circuit, then we can undo this transform and apply the desired transform A by synthesising $A \cdot P'^{-1}$ using Steiner-Gauss as explained in Section 2.

In our synthesis example, P'^{-1} is equivalent to the CNOTs that were commuted to the end of the zx-diagram. Incidentally, $P'^{-1} = A \cdot P'^{-1}$ because of our choice $A = I$, thus the desired linear transformation is already achieved. Moreover, these trailing CNOTs are already routed, however resynthesising them might remove a few redundant CNOTs for the final circuit.

Termination and correctness. Our algorithm terminates and is correct if the recursion converges and it synthesises the desired phase polynomial (f, A) while satisfying the connectivity constraints imposed by the architecture.

At each recursion step, the matrix P is split into P^0 and P^1. In the case of P^0, the base recursion step will effectively remove a row from P^0. In the case of P^1, the ones recursion step will place one or two CNOT gates. This will either remove a column from P^1 or, when splitting P^1 into $P^{1,0}$ and $P^{1,1}$, make sure that $P^{1,0} \neq \emptyset$. The recursion finishes when P is empty. Hence, the recursion converges and the algorithm terminates.

By construction, the matrix P describes the remaining phase gadgets to be synthesised (initially the parities $\boldsymbol{y}$ in the support of f). This remains the case while synthesising because placing a CNOT updates P with an elementary row addition as explained in Section 2. Moreover, a column is only removed from P iff the phase gadget is trivial, i.e. it is of the form described by Equation 4. Consequently, the phase gates are placed at the right parity by Lemma 2.3. Lastly, the basis transform A is obtained as described in the previous paragraph. Thus, the algorithm has synthesised the desired phase polynomial once it has terminated.

Additionally, all CNOT gates that are added have the property that the control and target qubits are neighbours in the connectivity graph G, thus satisfying the connectivity constraints imposed by the architecture.

Hence, our algorithm terminates and when it does the desired phase polynomial has been synthesised in an architecture-aware manner.

4 Results and discussion

To verify the quality of our algorithm, we generated random phase polynomials and synthesised them for two different real quantum computers: Rigetti's 16 qubit Aspen device and IBM's 20 qubit Singapore device [1]. We compare the average CNOT count, CNOT depth and runtime (in seconds) of our proposed algorithm with Staq [2] and t|ket⟩ [15]. To the best of our knowledge, t|ket⟩ and Staq are the only compilers that can synthesise and route phase polynomials from an abstract representation[2].

For each architecture, we randomly generated phase polynomials until we had 20 distinct ones with 1, 5, 10, 50, 100, 500, and 1000 phase gadgets in each. The phase gadgets were sampled uniformly across the parameter space. Figure 2 shows how each algorithm scales with respect to

---

[1]Qubit-scaling and gadget-scaling results for synthetic architectures can be found in Appendix C

[2]The source code to replicate our results, including the raw experimental data, can be found on
`https://github.com/CQCL/architecture-aware-phasepoly-synth`
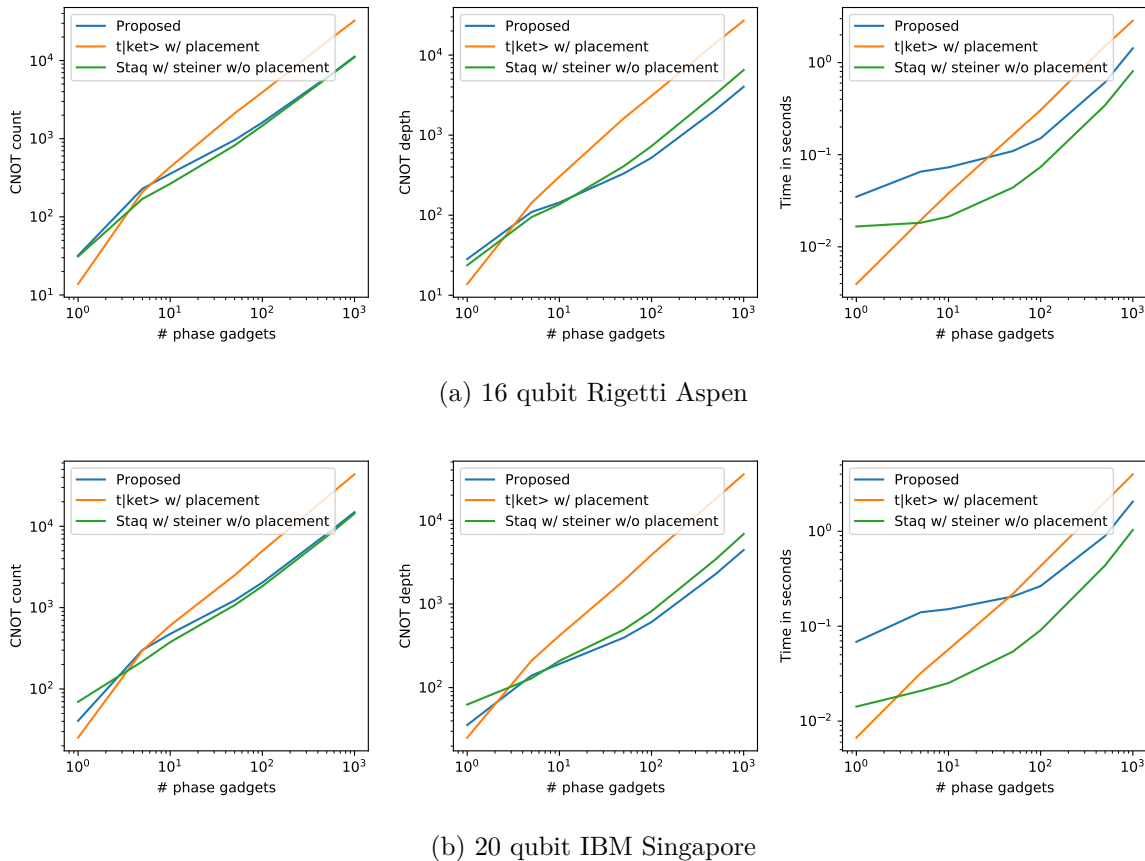
(a) 16 qubit Rigetti Aspen

(b) 20 qubit IBM Singapore

Figure 2: Plots showing the scaling of the CNOT count, CNOT depth and runtime with respect to the number of phase gadgets on the 16 qubit Rigetti Aspen architecture and the 20 qubit IBMQ Singapore device. The exact data can be found in Table 1 in Appendix C.

the number of phase gadgets on the two quantum computer architectures. Each point in the chart is the average of the 20 phase polynomials of that size.

We used pytket version 0.4.3[3]. We described our phase polynomials in terms of t|ket⟩'s abstract representation for phase gadgets (PauliExpBox) which t|ket⟩ synthesises and then routes using swaps [15]. While routing, we allowed t|ket⟩ to also find an optimal qubit placement.

For Staq, we used version 1.0. We chose to use the Steiner tree option because this results in a much lower CNOT count and depth. Unfortunately, we were unable to use this option in combination with optimal qubit placement because this took too long for large phase polynomials (≥ 50 phase gadgets) [4].

Note that both t|ket⟩ and Staq are implemented in C++, while our algorithm was written in python 3.6, putting it at a significant runtime disadvantage. All experiments were run on a 2017 MacBook Pro with an Intel Core i5 2.3 GHz and 8 GB 2133 MHz RAM. We used pytket to

---

[3]This pytket version will be released for the general public soon

[4]Staq results with placement for small phase polynomials can be found in Figure 5 of Appendix C

calculate the CNOT count and CNOT depth of all circuits (including Staq).

We observe that for very small phase polynomials (1 phase gadget), t|ket⟩ is the best method, but it does not scale well in CNOT count and depth for larger, more realistic phase polynomials (see Figure 2). This shows that naive synthesis combined with clever routing is not competitive with architecture-aware synthesis methods.

Between five and 100 phase gadgets, Staq has the lowest average CNOT count. For larger phase polynomials, Staq's CNOT count performance is equal to the proposed algorithm. However, the CNOT depth is consistently better when synthesised with the proposed algorithm for phase polynomials with more than 10 gadgets. This means that it is better at parallelising CNOT gates than Staq.

With respect to runtime, we observe that for phase polynomials with 5-1000 phase gadgets, Staq is the fastest synthesis algorithm. The proposed algorithm is faster at synthesising than t|ket⟩ for phase polynomials 50-1000 gadgets on both architectures. We do note that both Staq and the proposed algorithm does not scale linearly with respect to the number of phase gadgets, thus it might not be faster than t|ket⟩ for phase polynomials with more phase gadgets than we have tested.

5   Conclusion and Future Work

In this paper, we introduced one of the first successful algorithms for architecture-aware synthesis of phase polynomials. We showed that this algorithm performs comparable or better than current state-of-the-art compilers for current NISQ devices without compromising the runtime of the algorithm.

Although our algorithm is very promising, it should still be adjusted to better fit the specification of the device that it is synthesising for. For example, the choice of placing the qubits affects the size of the synthesised circuit because the connectivity graph of a quantum computer is generally not regular. Similarly, the current algorithm improves CNOT depth, but it might do so in a way that increases the crosstalk between parallel gates.

And, lastly, our algorithm can only synthesise phase polynomials. This means that circuits containing rotations over X and Y need to be split into subcircuits to use our algorithm. It will be much more beneficial if our algorithm can be extended to also synthesise the generalised version of phase gadgets, called *Pauli exponentials*.

Acknowledgements

References

[1] Matthew Amy, Parsiad Azimzadeh, and Michele Mosca. On the controlled-NOT complexity of controlled-NOT–phase circuits. *Quantum Science and Technology*, 4(1):015002, sep 2018, arXiv:1712.01859.

[2] Matthew Amy and Vlad Gheorghiu. staq–A full-stack quantum processing toolkit. *arXiv preprint arXiv:1912.06070*, 2019.

[3] 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.

[4] Bob Coecke and Ross Duncan. Interacting Quantum Observables: Categorical Algebra and Diagrammatics. *New J. Phys*, 13(043016), 2011, arXiv:0906.4725.

[5] Bob Coecke and Aleks Kissinger. *Picturing Quantum Processes: A First Course in Quantum Theory and Diagrammatic Reasoning.* Cambridge University Press, 2017.

[6] Alexander Cowtan, Silas Dilkes, Ross Duncan, Alexandre Krajenbrink, Will Simmons, and Seyon Sivarajah. On the qubit routing problem. In Wim van Dam and Laura Mancinska, editors, *14th Conference on the Theory of Quantum Computation, Communication and Cryptography (TQC 2019)*, volume 135 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 5:1–5:32, 2019.

[7] Alexander Cowtan, Silas Dilkes, Ross Duncan, Will Simmons, and Seyon Sivarajah. Phase Gadget Synthesis for Shallow Circuits. In *Proceedings of QPL2019 (to appear)*, 2019, arXiv:1906.01734.

[8] Christopher M. Dawson, Henry L. Haselgrove, Andrew P. Hines, Duncan Mortimer, Michael A. Nielsen, and Tobias J. Osborne. Quantum computing and polynomial equations over the finite field Z_2. *Quantum Information and Computation*, 5(2):102–112, 2005, arXiv:quant-ph/0408129.

[9] Frank Gray. Pulse code communication. US Patent 2632058, 1953.

[10] Richard M Karp. Reducibility among combinatorial problems. In *Complexity of computer computations*, pages 85–103. Springer, 1972.

[11] Aleks Kissinger and Arianne Meijer van de Griend. CNOT circuit extraction for topologically-constrained quantum memories. *Quantum Information and Computation*, 2020, arXiv:1904.00633. To appear.

[12] Beatrice Nash, Vlad Gheorghiu, and Michele Mosca. Quantum circuit optimizations for NISQ architectures. *arXiv.org*, 2019, arXiv:1904.01972.

[13] Ketan N Patel, Igor L Markov, and John P Hayes. Optimal synthesis of linear reversible circuits. *Quantum Information & Computation*, 8(3):282–294, 2008.

[14] John Preskill. Quantum Computing in the NISQ era and beyond. *Quantum*, 2:79, August 2018.

[15] Seyon Sivarajah, Silas Dilkes, Alexander Cowtan, Will Simmons, Alec Edgington, and Ross Duncan. t|ket⟩: A Retargetable Compiler for NISQ Devices. *ArXiv.org*, 2020, arXiv:2003.10611.

[16] Mathias Soeken. Using ZDDs in the mapping of quantum circuits. *arXiv.org*, 2019, arXiv:1901.02406.

[17] Robert Wille, Lukas Burgholzer, and Alwin Zulehner. Mapping Quantum Circuits to IBM QX Architectures Using the Minimal Number of SWAP and H Operations. *arXiv.org*, 2019, arXiv:1907.02026.

[18] Bujiao Wu, Xiaoyu He, Shuai Yang, Lifu Shou, Guojing Tian, Jialin Zhang, and Xiaoming Sun. Optimization of CNOT circuits on topological superconducting processors. *arXiv preprint arXiv:1910.14478*, 2019.

[19] Alwin Zulehner, Alexandru Paler, and Robert Wille. An Efficient Methodology for Mapping Quantum Circuits to the IBM QX Architectures. *arXiv.org*, 2017, arXiv:1712.04722.

A   Subfunctions for the proposed synthesis algorithm

The subfunctions that we used in the pseudocode for our algorithm (Figure 1) are listed below.

> **function** REDUCECOLUMNS($Columns$)
>   **for all** $c \in Columns$ **do**
>   **if** $|\{q \in P.\text{rows where } P_{q,c} = 1\}| = 1$ **then**
> $Qubit \leftarrow \text{argmax}_{q \in P.\text{rows}} P_{q,c}$
> $Circuit.\text{AddGate}(R_Z(ZPhases[c], Qubit))$
> $Columns \leftarrow Columns \setminus \{c\}$
>   **end if**
>   **end for**
>   **return** $Columns$
> **end function**
> **function** PLACECNOT($Control$, $Target$)
> $Circuit.\text{AddGate}(\text{CNOT } (Control, Target))$
> $P[Control] \leftarrow P[Control] + P[Target]$
> **end function**
> **function** SPLITCOLSONROW($Columns$, Row)
> $Cols0 \leftarrow \{c \in Columns \text{ where } P_{Row,c} = 0\}$
> $Cols1 \leftarrow \{c \in Columns \text{ where } P_{Row,c} = 1\}$
>   **return** $Cols0$, $Cols1$
> **end function**

B   Example synthesis

To get a better idea of the inner workings of the algorithm, we synthesise the following phase polynomial:

$$f(\boldsymbol{x}) = \alpha_1(x_2 \oplus x_3) + \alpha_2(x_1) + \alpha_3(x_1 \oplus x_4) + \alpha_4(x_1 \oplus x_2 \oplus x_4) + \alpha_5(x_1 \oplus x_2) + \alpha_6(x_1 \oplus x_2 \oplus x_3)$$
$$A = I$$

Note that this is the parameterised version of the example phase polynomial given by Amy et al.[1]. The connectivity graph we use for synthesis is a simple line architecture: $G : x_1 \Leftrightarrow x_2 \Leftrightarrow x_3 \Leftrightarrow x_4$.

This phase polynomial corresponds to the following ZX-diagram C and matrix representation P.


Note that the matrix P has a column for each phase gadget in the diagram and each row has a 1 iff the corresponding qubit is participating in the corresponding phase gadget (i.e. it has a green spider). We have added a red vertical line to the ZX-diagram to represent the *frontier*. This indicates the progress of our synthesis. The diagram on the left of the frontier has

been synthesised, the diagram on the right of the frontier contains the phase polynomial to be synthesised. Additionally, while synthesising, we will rewrite the diagram C by adding gates to the frontier without changing the semantics of C.

Preprocessing. The first step in the algorithm is to check if any columns can be removed from the matrix. This is possible if the column contains exactly a single entry with the value 1. If this is the case, the phase gadget is only acting on a single qubit and it is equivalent to a Z phase gate which we can move to the other side of the frontier.
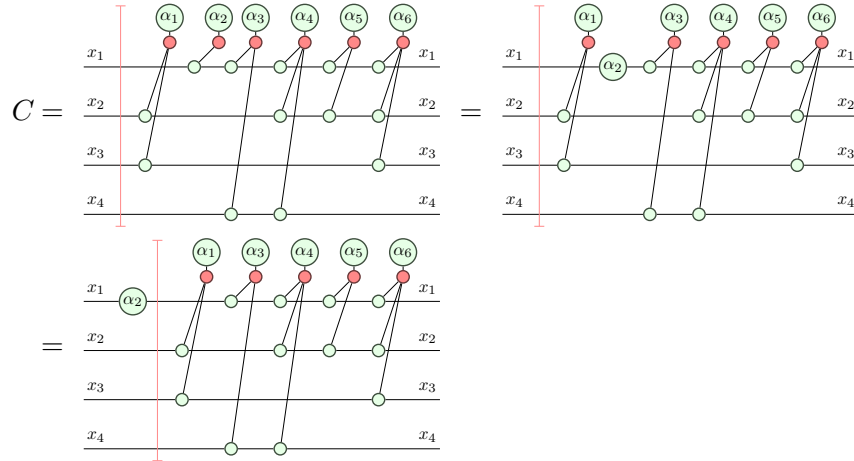
We describe this process as *placing a phase gate*.

Once the phase gate $R_Z(\alpha_2)$ is placed on qubit x_1, we have a phase gadget less, so we can remove the corresponding column from the matrix P.

$$\sim \begin{pmatrix} 0 & 1 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 \end{pmatrix}$$
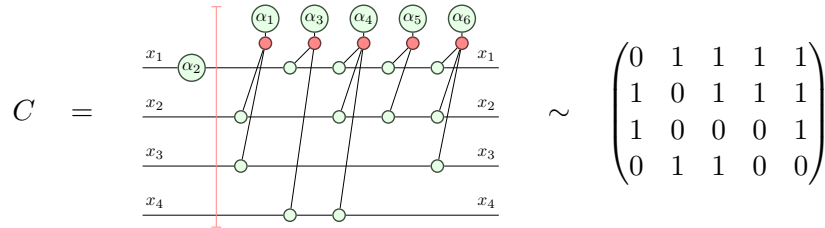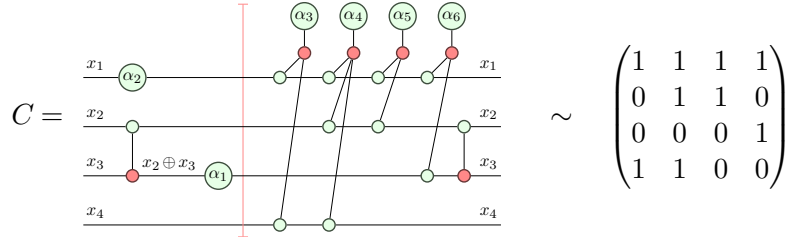
Main recursion. Now we can start the main recursion loop. We start with the base recursion step and calculate all non-cutting vertices of our graph G, which are $\{x_1, x_4\}$. We pick the row in P with either most ones or most zeroes, which is x_1. We split the row in to columns with zeroes $P^0 \sim \{\alpha_1\}$, and columns with ones $P^1 \sim \{\alpha_3, \alpha_4, \alpha_5, \alpha_6\}$. We recurse using the base recursion step on P^0 and the ones recursion step on P^1.

In the base recursion step on P^0, we have subgraph $G : x_2 \Leftrightarrow x_3 \Leftrightarrow x_4$, with non-cutting vertices $\{x_2, x_4\}$. We pick x_2 arbitrarily and split the matrix once more into $P^{0,0} \sim \emptyset$ and $P^{0,1} \sim \{\alpha_1\}$. This time, there are no columns with zeroes, so the base recursion step is trivial. Then, in the ones recursion step on $P^{0,1}$, we pick a neighbour of x_2 with the most ones, this is x_3, and we place a CNOT gate, C_{x_2,x_3}, in front of the frontier. To keep the diagram equivalent to the previous diagrams, we add a second CNOT gate, C_{x_2,x_3}, after the frontier and commute it through the phase gadgets. By commuting the second CNOT gate through the gadgets, each control qubit will participate in the phase gadget iff either the control or the target qubit

(exclusive) was participating before commuting the CNOT through, see Section 2 for a detailed explanation. This is the same as adding the target row to the control row in the matrix P (modulo 2). Observe that this also changes the columns in P^1.

$$C = \quad = \quad \sim \begin{pmatrix} 0 & 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 \end{pmatrix}$$
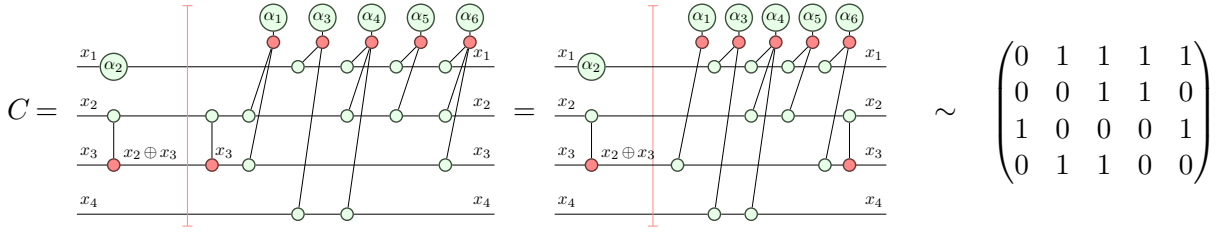
As a result, we can place a phase gate, $R_Z(\alpha_1)$, corresponding to α_1 on qubit x_3.

$$C = \quad \sim \begin{pmatrix} 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 1 & 0 & 0 \end{pmatrix}$$

Note that placing the phase gate causes that $P^0 = \emptyset$ so splitting the row X_2 and recursing on $P^{0,0} \sim \emptyset$ and $P^{0,1} \sim \emptyset$ is trivial.
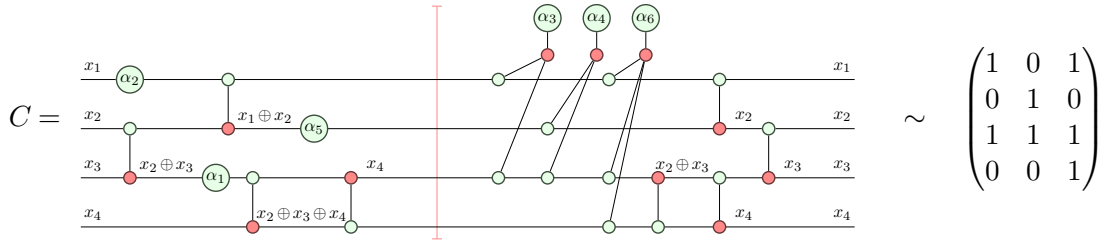
Now we are finished with the base recursion step on P^0 and continue with the ones recursion step on P^1 and the full graph G. We had chosen x_1 earlier, now we pick a neighbour, x_2, and place the CNOT gate, C_{x_1,x_2}. This allows us to place a phase gate, $R_Z(\alpha_5)$, on qubit x_2.

$$C = \quad = \quad \sim \begin{pmatrix} 1 & 0 & 1 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 1 & 0 \end{pmatrix}$$
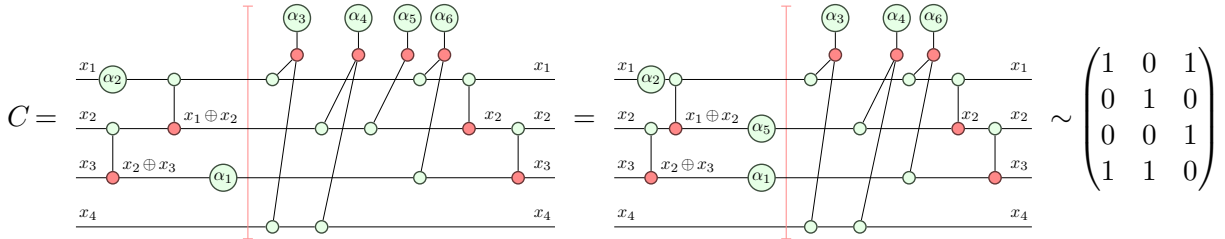
Again, we split row x_1 into columns with zeroes $P^{1,0} \sim \{\alpha_4\}$ and with ones $P^{1,1} \sim \{\alpha_3, \alpha_6\}$. We use the base recursion step on $P^{1,0}$ with the subgraph $G : x_2 \Leftrightarrow x_3 \Leftrightarrow x_4$ and we use the ones recursion step on $P^{1,1}$

The subgraph $G : x_2 \Leftrightarrow x_3 \Leftrightarrow x_4$ has non-cutting vertices x_2 and x_4. We pick x_4 arbitrarily and split the row into $P^{1,0,0} \sim \emptyset$ and $P^{1,0,1} \sim \{\alpha_4\}$. The base recursion step on $P^{1,0,0}$ is trivial. In the ones recursion step, we pick neighbour x_3 and place two CNOT gates, C_{x_3,x_4}, and C_{x_4,x_3}, because x_3 only has zeroes in $P^{1,0,1}$.

$$C = \quad \sim \begin{pmatrix} 1 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 1 & 1 \\ 1 & 1 & 0 \end{pmatrix}$$
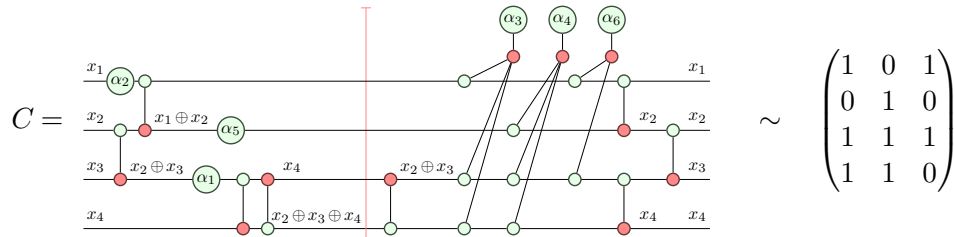
Now we split $P^{1,0,1}$ on row x_4 into $P^{1,0,1,0} \sim \{\alpha_4\}$ and $P^{1,0,1,1} \sim \emptyset$ and recurse as before, note that the latter case in trivial.

In the base recursion step on $P^{1,0,1,0}$, we are left with the subgraph $G : x_2 \Leftrightarrow x_3$. We pick row $x_2$ arbitrarily and split it into $P^{1,0,1,0,0} \sim \emptyset$ and $P^{1,0,1,0,1} \sim \{\alpha_4\}$. The base recursion step on $P^{1,0,1,0,0}$ is trivial and in the ones recursion step, we pick neighbour x_3. Hence we can place a CNOT gate, C_{x_2,x_3}, and a phase gate, $R_Z(\alpha_4)$ on qubit x_3.
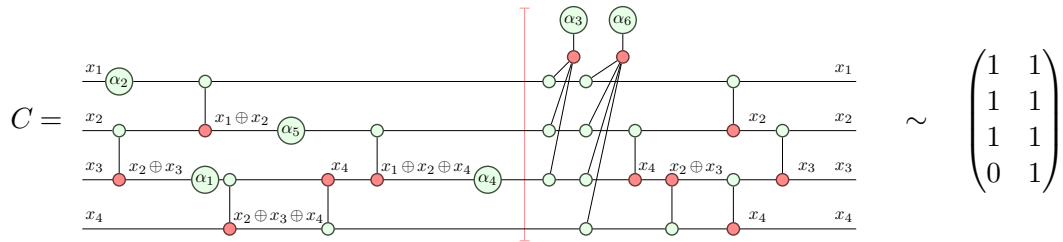
This finishes the recursion on $P^{1,0}$ and we can continue with the ones recursion step on $P^{1,1} \sim \{\alpha_3, \alpha_6\}$. Once more, we are back at the original graph $G : x_1 \Leftrightarrow x_2 \Leftrightarrow x_3 \Leftrightarrow x_4$. We previously picked row x_1 and so we now pick neighbour x_2. We place a CNOT gate, C_{x_1,x_2}, and split on row x_1 into $P^{1,1,0} \sim \{\alpha_3, \alpha_6\}$, and $P^{1,1,1} \sim \emptyset$.
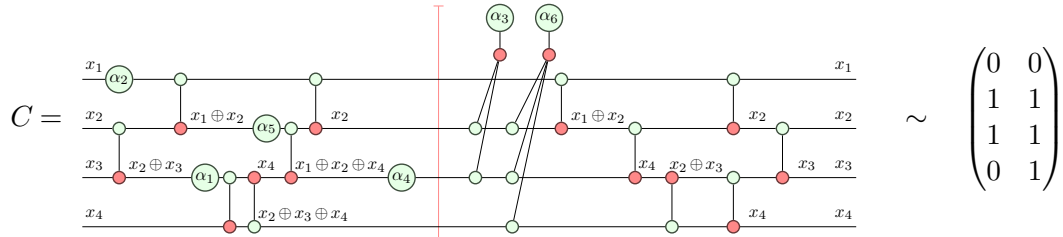
In the base recursion step on $P^{1,1,0}$, we pick row x_2 and split $P^{1,1,0}$ into $P^{1,1,0,0} \sim \emptyset$, and $P^{1,1,0,1} \sim \{\alpha_3, \alpha_6\}$. The base recursion step on $P^{1,1,0,0}$ is trivial.

In the ones recursion step on $P^{1,1,0,1}$, we pick neighbour x_3, and place a CNOT gate, C_{x_2,x_3}, and a phase gate, $R_Z(\alpha_3)$, on qubit x_3.
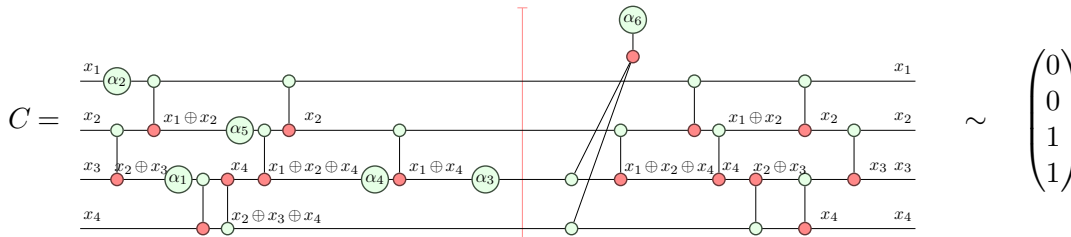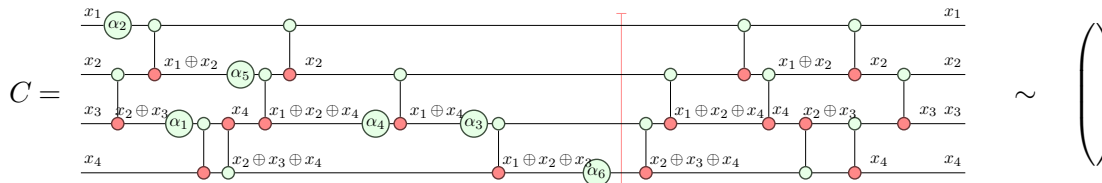
We split $P^{1,1,0,1}$ on row x_2, resulting in $P^{1,1,0,1,0} \sim \{\alpha_6\}$, and $P^{1,1,0,1,1} \sim \emptyset$ and we recurse as before.

In the base recursion on $P^{1,1,0,1,0}$, we are left with subgraph $G : x_3 \Leftrightarrow x_4$. We pick x_3 and split on it resulting in $P^{1,1,0,1,0,0} \sim \emptyset$ and $P^{1,1,0,1,0,1} \sim \{\alpha_6\}$. The base recursion step is trivial.

Finally, in the ones recursion step on $P^{1,1,0,1,0,1}$, we pick neighbour x_4 and place a CNOT gate, C_{x_3,x_4} and a phase gate, $R_Z(\alpha_6)$, on qubit x_4.

Now we have synthesised every phase gadget in the support of f.

Post-processing. What remains is synthesising the basis transform $A = I$. At the frontier, the basis transform of the qubits is equal to the matrix P',

$$P' = \begin{pmatrix} 1 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

as can be seen in the parity annotation of each qubit of the final circuit. This transform needs to be undone before the basis transform A can be applied.

As explained at the end of Section 3, this is transformation is undone by the trailing CNOTs on the right of the frontier. I.e. the CNOTs on the right of the frontier apply the basis transform P'^{-1}. Although these CNOTs are already mapped, they could be optimised using an architecture-aware CNOT circuit synthesis technique, such as Steiner-Gauss. In case the matrix $A \neq I$, we can calculate the full transformation A' by undoing the existing linear transformation and then applying the desired transformation: $A' = A \cdot P'^{-1}$.

C   Additional results

This appendix contains additional figures and tables to show the performance of the proposed algorithm with respect to the existing algorithms.

To show the scaling of our algorithm with respect the number of qubits, the number of phase gadgets and the density of the device connectivity graph, we have run several experiments, generating 20 random phase polynomials per experimental setting. Since Staq only supports a small selection of quantum computer architectures, we compare the proposed algorithm against an in-house implementation of Steiner-GraySynth for all synthetic architectures.

Figure 3 shows how our algorithm and the two baselines perform on a line, square and fully connected connectivity of various sizes given a phase polynomial with 100 phase gadgets. Similarly, Figure 4 shows how our algorithm and the two baselines perform on phase polynomials of various sizes given a 36 qubit line, square and unconstrained connectivity graph. In Figure 5, we show that, if Staq is used with qubit placement optimisation, it can synthesise slightly smaller

circuits than without qubit placement. However, this comes at an extreme runtime cost. The runtime of this option was long enough that it was not feasible for to run experiments with more than 50 and 100 gadgets (IBMQ Singapore and Rigetti Aspen, respectively) because Staq would take more than two hours to synthesise a single circuit with 500 gadgets on Rigetti Aspen.

Lastly, the exact data that was visualised in each figure, Figure 2, 3, 4, and 5, is given in Table 1, 2, 3, and 4, respectively.
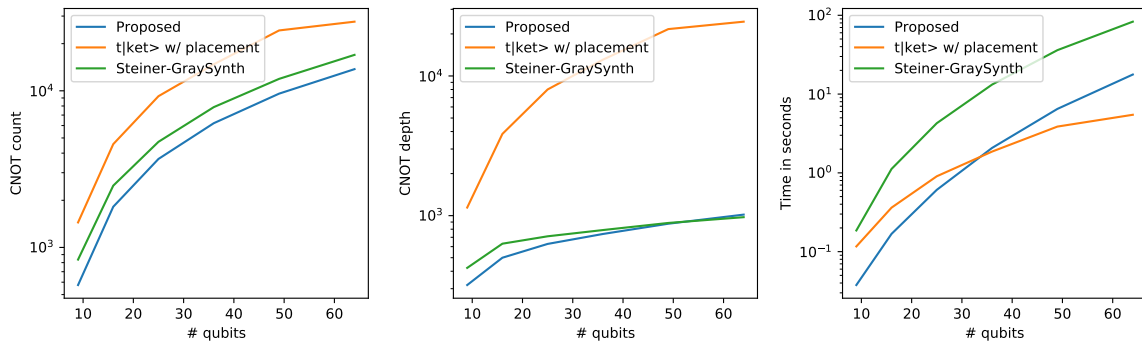
(a) line
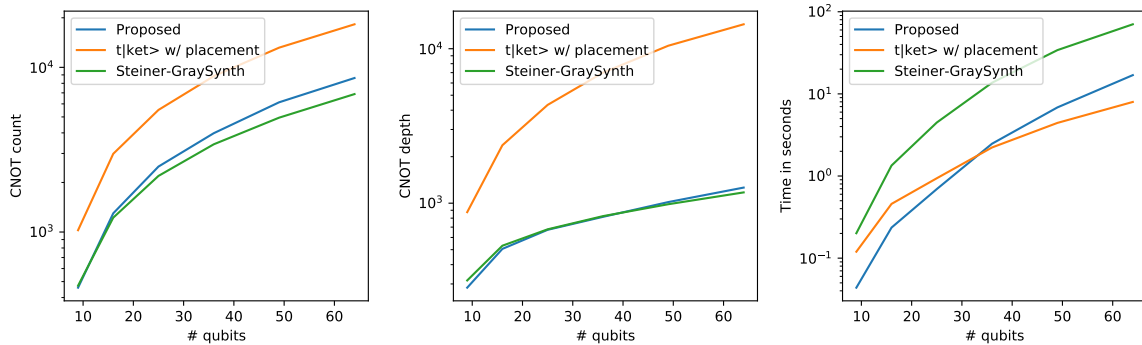
(b) square grid
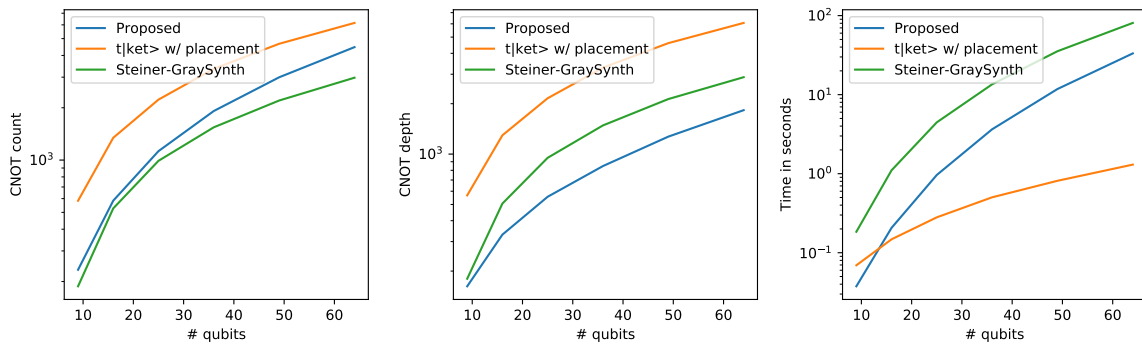
(c) fully connected

Figure 3: The influence of the number of qubits on the CNOT count, CNOT depth and runtime for architectures with different regular structures: line, square grid and fully connected. The exact data can be found in Table 2.
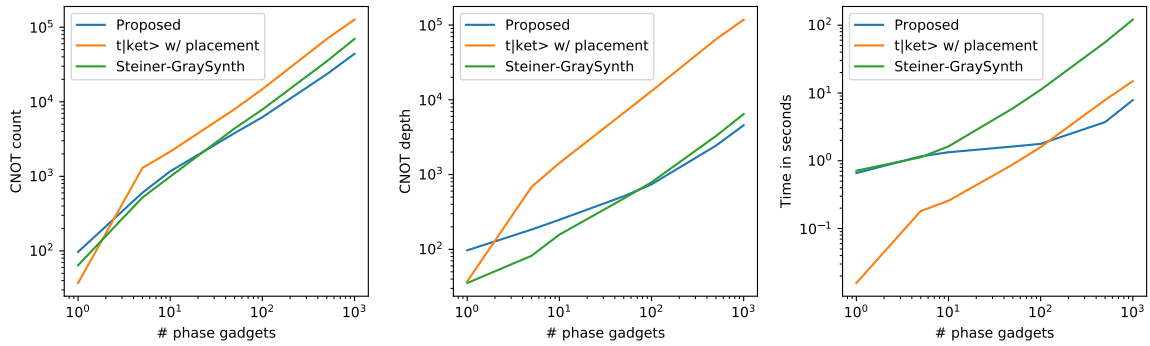
(a) line
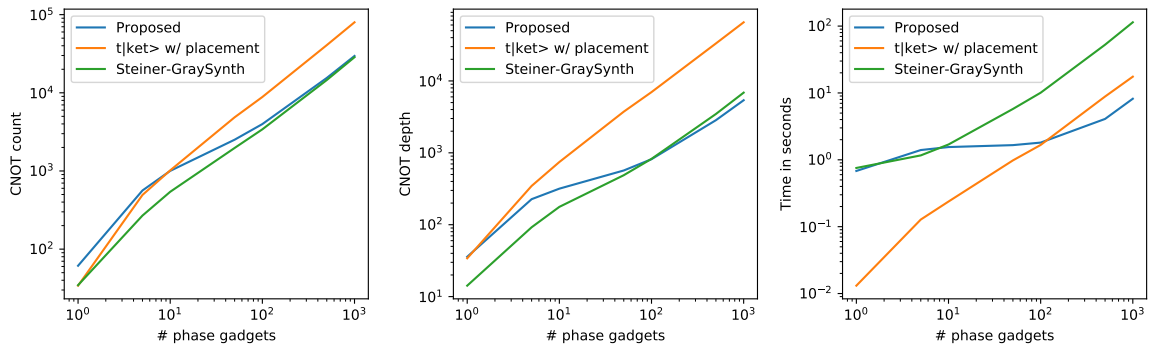
(b) square grid
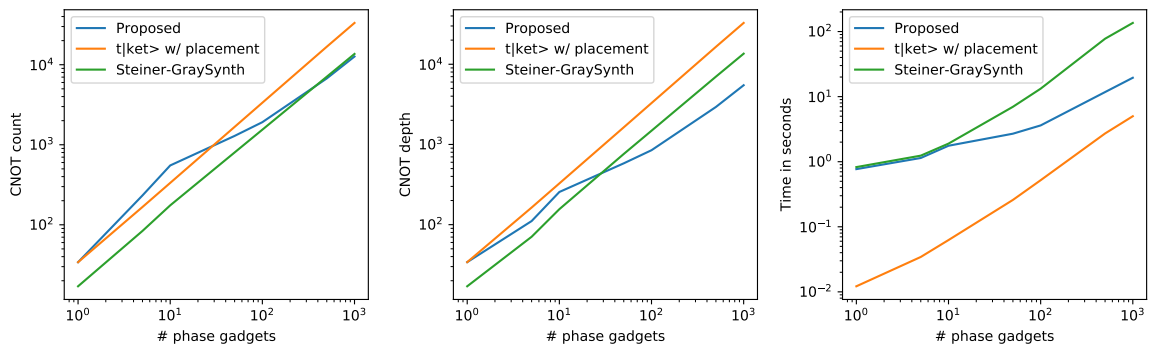
(c) fully connected

Figure 4: Plots showing the scaling of the CNOT count, CNOT depth and runtime with respect to the number of phase gadgets on a 36 qubit line, square grid, and unconstrained architecture. The exact data can be found in Table 3.
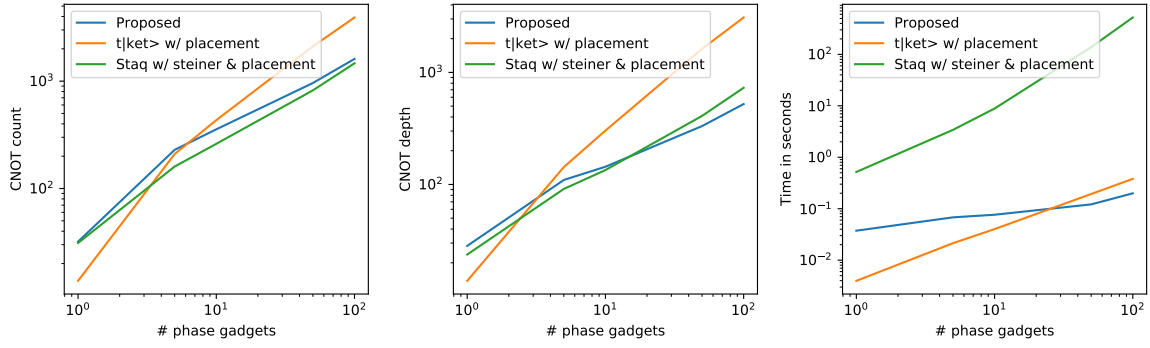
(a) 16 qubit Rigetti Aspen
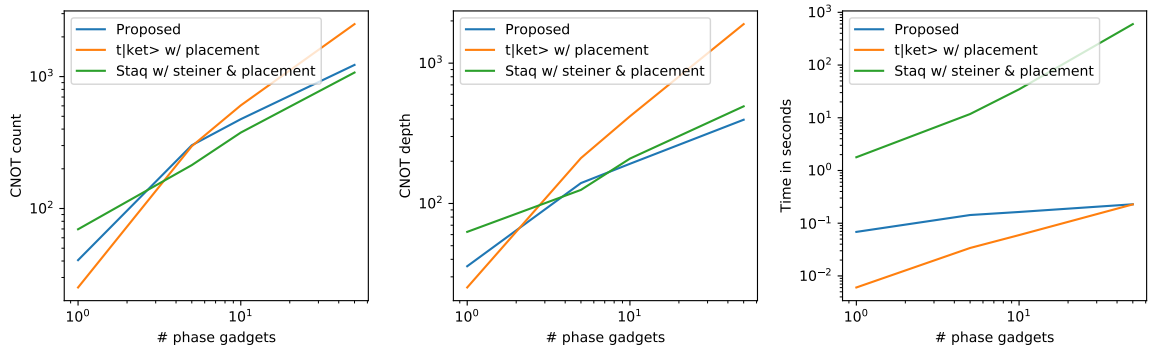
(b) 20 qubit IBM Singapore

Figure 5: Plots showing the scaling of the CNOT count, CNOT depth and runtime with respect to the number of phase gadgets on the 16 qubit Rigetti Aspen architecture and the 20 qubit IBMQ Singapore device. The exact data can be found in Table 4.

$\#R_Z$	t|ket⟩			Staq			Proposed		
	count	depth	time	count	depth	time	count	depth	time
1	13.80	13.80	0.004s	31.10	23.70	0.017s	32.00	28.30	0.035s
5	209.55	142.95	0.020s	169.85	94.70	0.018s	229.30	109.70	0.066s
10	433.25	302.65	0.038s	264.80	136.45	0.021s	355.80	143.95	0.073s
50	2109.35	1622.45	0.164s	820.80	409.00	0.044s	961.05	331.90	0.110s
100	3917.60	3090.60	0.306s	1466.85	728.80	0.074s	1611.90	522.05	0.151s
500	17416.70	14274.65	1.506s	5928.60	3293.90	0.345s	6081.30	2082.25	0.611s
1000	32357.05	26896.25	2.851s	11043.40	6500.05	0.807s	11238.30	4018.20	1.431s

(a) Rigetti 16Q Aspen

| $\#R_Z$	t|ket⟩			Staq			Proposed		
	count	depth	time	count	depth	time	count	depth	time
1	25.20	25.20	0.007s	69.60	62.75	0.014s	40.60	35.70	0.068s
5	296.70	210.60	0.032s	218.60	129.15	0.021s	301.60	139.75	0.140s
10	604.95	417.90	0.057s	376.50	208.60	0.025s	475.50	191.75	0.151s
50	2499.30	1897.70	0.219s	1073.25	492.40	0.054s	1226.00	395.20	0.206s
100	4969.60	3863.90	0.431s	1834.75	819.65	0.091s	2035.10	607.95	0.265s
500	22710.00	18205.90	2.033s	7498.85	3440.55	0.437s	8054.35	2293.45	0.893s
1000	43538.10	35533.30	3.983s	14309.70	6867.05	1.031s	14908.55	4422.70	2.054s

(b) IBMQ Singapore

Table 1: The average number of CNOT, CNOT depth and runtime for 20 circuits for synthesising phase polynomials with various sizes using t\|ket⟩, Staq (without qubit placement) and our proposed algorithm on Rigetti Aspen (Table 1a) and IBMQ Singapore (Table 1b). This data was visualised in Figure 2.

Qubits	t|ket⟩			Nash			Proposed		
	count	depth	time	count	depth	time	count	depth	time
9	1444.30	1142.75	0.117s	836.45	422.75	0.186s	575.40	318.65	0.038s
16	4565.50	3846.85	0.362s	2480.10	629.95	1.121s	1818.75	499.85	0.168s
25	9240.80	8004.30	0.905s	4724.60	711.55	4.269s	3673.35	627.50	0.609s
36	14761.20	13074.70	1.863s	7866.50	788.75	13.198s	6211.55	739.30	2.072s
49	24291.45	21651.35	3.867s	11920.00	886.60	36.099s	9592.70	875.40	6.484s
64	27632.10	24473.65	5.462s	16960.45	975.30	82.994s	13750.00	1017.30	17.706s

(a) Line

Qubits	t|ket⟩			Nash			Proposed		
	count	depth	time	count	depth	time	count	depth	time
9	1025.65	874.75	0.120s	472.15	316.10	0.201s	459.35	283.85	0.044s
16	2986.20	2372.30	0.458s	1222.50	530.50	1.342s	1299.00	505.95	0.236s
25	5514.35	4327.40	0.939s	2191.35	677.30	4.479s	2497.65	672.35	0.693s
36	8842.65	6993.00	2.227s	3409.85	824.35	13.679s	3982.25	815.60	2.475s
49	13171.00	10457.15	4.432s	4948.05	984.55	34.106s	6135.10	1016.65	6.843s
64	18259.70	14393.05	7.977s	6881.55	1174.30	70.358s	8615.05	1261.85	16.918s

(b) Square

Qubits	t|ket⟩			Nash			Proposed		
	count	depth	time	count	depth	time	count	depth	time
9	583.30	566.90	0.069s	187.75	180.05	0.184s	233.75	162.25	0.038s
16	1343.40	1294.60	0.148s	527.05	506.25	1.103s	583.85	330.15	0.206s
25	2227.70	2155.15	0.280s	990.10	949.85	4.463s	1125.55	555.80	0.963s
36	3351.10	3281.25	0.502s	1543.40	1483.15	13.501s	1915.45	847.85	3.642s
49	4668.50	4599.60	0.813s	2200.75	2131.25	35.417s	2994.10	1271.05	11.777s
64	6155.90	6076.20	1.303s	2978.95	2880.10	80.856s	4466.60	1829.50	33.257s

(c) Unconstrained

Table 2: The average number of CNOT, CNOT depth and runtime for 20 circuits for synthesising phase polynomials with 100 phase gadgets using t\|ket⟩, Nash and our proposed algorithm on synthetic qubit architectures of various sizes connected in a line (Table 2a), square (Table 2b) and fully connected (Table 2c). This data was visualised in Figure 3.

$\#R_Z$	t|ket⟩ count	depth	time	Nash count	depth	time	Proposed count	depth	time
1	37.10	37.10	0.016s	64.15	35.40	0.714s	96.90	96.90	0.658s
5	1298.70	684.85	0.181s	521.10	82.00	1.126s	603.80	183.95	1.163s
10	2142.80	1424.30	0.257s	991.85	157.15	1.631s	1167.45	248.40	1.335s
50	7960.45	6700.05	0.884s	4391.70	477.30	5.943s	3800.50	512.50	1.624s
100	14761.20	13074.70	1.591s	7866.50	788.75	11.137s	6211.55	739.30	1.776s
500	69417.10	64493.35	7.885s	34883.95	3272.30	55.913s	23425.10	2435.75	3.718s
1000	126095.50	118035.55	14.913s	69584.50	6446.50	120.575s	43934.05	4564.45	7.853s

(a) 36 qubit line

$\#R_Z$	t|ket⟩ count	depth	time	Nash count	depth	time	Proposed count	depth	time
1	34.10	34.10	0.013s	34.60	14.20	0.755s	61.30	35.80	0.681s
5	493.45	346.30	0.128s	268.80	92.15	1.164s	562.95	226.65	1.400s
10	1017.65	745.20	0.238s	541.70	177.15	1.695s	1002.75	317.45	1.552s
50	4859.05	3737.90	0.980s	1969.35	488.35	5.762s	2512.10	568.20	1.658s
100	8842.65	6993.00	1.669s	3409.85	824.35	10.080s	3982.25	815.60	1.809s
500	40734.55	33232.65	8.840s	14560.45	3469.65	52.656s	15492.85	2844.85	4.087s
1000	79821.90	65423.70	17.518s	28530.30	6876.90	113.956s	29621.30	5390.25	8.196s

(b) 36 qubit square

$\#R_Z$	t|ket⟩ count	depth	time	Nash count	depth	time	Proposed count	depth	time
1	34.10	34.10	0.012s	17.05	17.05	0.826s	34.10	34.10	0.769s
5	166.90	163.50	0.034s	83.80	70.65	1.239s	232.70	110.90	1.141s
10	333.80	326.95	0.063s	174.95	156.00	1.897s	551.40	255.60	1.759s
50	1673.00	1636.20	0.259s	801.60	761.30	6.981s	1290.25	581.80	2.695s
100	3351.10	3281.25	0.520s	1543.40	1483.15	13.252s	1915.45	847.85	3.611s
500	16781.80	16489.55	2.715s	7081.75	7006.85	77.596s	6741.75	2935.00	11.749s
1000	33291.10	32759.35	4.999s	13642.20	13550.90	135.732s	12660.40	5479.05	19.504s

(c) 36 qubit unconstrained

Table 3: The average number of CNOT, CNOT depth and runtime for 20 circuits for synthesising phase polynomials with various sizes using t\|ket⟩, Nash and our proposed algorithm on synthetic 36 qubit architectures connected in a line (Table 3a), square (Table 3b) and fully connected (Table 3c). This data was visualised in Figure 4.

$\#R_Z$	t|ket⟩			Staq			Proposed		
	count	depth	time	count	depth	time	count	depth	time
1	13.80	13.80	0.004s	31.10	23.70	0.518s	32.00	28.30	0.037s
5	209.55	142.95	0.021s	160.20	91.20	3.407s	229.30	109.70	0.068s
10	433.25	302.65	0.040s	260.65	134.45	8.854s	355.80	143.95	0.076s
50	2109.35	1622.45	0.193s	820.80	409.00	138.448s	961.05	331.90	0.121s
100	3917.60	3090.60	0.380s	1466.85	728.80	521.102s	1611.90	522.05	0.199s

(a) Rigetti 16Q Aspen

$\#R_Z$	t|ket⟩			Staq			Proposed		
	count	depth	time	count	depth	time	count	depth	time
1	25.20	25.20	0.006s	69.60	62.75	1.784s	40.60	35.70	0.068s
5	296.70	210.60	0.034s	213.30	124.95	11.770s	301.60	139.75	0.143s
10	604.95	417.90	0.060s	376.30	208.80	34.636s	475.50	191.75	0.163s
50	2499.30	1897.70	0.227s	1073.25	492.40	597.870s	1226.00	395.20	0.229s

(b) IBMQ Singapore

Table 4: The average number of CNOT, CNOT depth and runtime for 20 circuits for synthesising phase polynomials with various sizes using t\|ket⟩, Staq (with qubit placement) and our proposed algorithm on Rigetti Aspen (Table 4a) and IBMQ Singapore (Table 4b). This data was visualised in Figure 5.