
Article:
Campbell, E. and Heyfron, L. (2018) An efficient quantum compiler that reduces T count. Quantum Science and Technology, 4 (1). 015004. ISSN 2058-9565

https://doi.org/10.1088/2058-9565/aad604

An Efficient Quantum Compiler that Reduces T Count

Luke E. Heyfron[1,*] and Earl T. Campbell[1,†]

[1] *Department of Physics and Astronomy, University of Sheffield, Sheffield, UK*

(Dated: 25th May 2018)

Before executing a quantum algorithm, one must first decompose the algorithm into machine-level instructions compatible with the architecture of the quantum computer, a process known as quantum compiling. There are many different quantum circuit decompositions for the same algorithm but it is desirable to compile leaner circuits. A fundamentally important cost metric is the T count – the number of T gates in a circuit. For the single qubit case, optimal compiling is essentially a solved problem. However, multi-qubit compiling is a harder problem with optimal algorithms requiring classical runtime exponential in the number of qubits. Here, we present and compare several efficient quantum compilers for multi-qubit Clifford + T circuits. We implemented our compilers in C++ and benchmarked them on random circuits, from which we determine that our TODD compiler yields the lowest T counts on average. We also benchmarked TODD on a library of reversible logic circuits that appear in quantum algorithms and found that it reduced the T count for 97% of the circuits with an average T-count saving of 20% when compared against the best of all previous circuit decompositions.

Compiling is the conversion of an algorithm into a series of hardware level commands or elementary gates. Better compilers can implement the same algorithm using fewer hardware level instructions, reducing runtime and other resources. Quantum compiling or gate-synthesis is the analogous task for a quantum computer and is especially important given the current expense of quantum hardware. Early in the field, Solovay and Kitaev proposed a general purpose compiler for any universal set of elementary gates [1–3]. Newer compilers exploit the specific structure of the Clifford+T gate set and have reduced quantum circuit depths by several orders of magnitude [4–7], often improving the classical compile time. The Clifford+T gate set is natural since it is the fault-tolerant logical gate set in almost every computing architecture [8]. Moreover, fault-tolerance protocols have been proposed such as magic state distillation [9] that lead to a cost per T gate which is several hundred times larger than that of Clifford gates [10–12], which suggests T count as the key metric of compiler performance. Furthermore, the T count is an important metric beyond the standard compiling problem because it relates to the classical overhead of simulating quantum circuits [16, 40, 41] as well as the distillation cost of synthillation [24]. For these reasons, it is clear that developing methods for minimizing the T count is crucial for a variety of applications in quantum computation.

Significant progress has been made on synthesis of single-qubit unitaries from Clifford+T gates. For purely unitary synthesis, the problem is essentially solved since we have a compiler that is asymptotically optimal and efficient [4, 7]. Although further improvements are possible beyond unitary circuits, by making use of ancilla qubits and measurements [13–16] or adding an element of randomness to compiling [17, 18]. On the other hand, the multi-qubit problem is much more challenging. An algorithm for multi-qubit unitary synthesis over the Clifford+T gate set is known that is provably optimal in terms of the T count but the compile runtime is exponential in the number of qubits [6, 19]. Compilers with efficient runtimes have been proposed but with no promise of T count optimality [20, 21]. We seek a compiler that runs efficiently and yields circuits with T counts that are as low as practically achievable.

A useful strategy is to take an initial Clifford+T circuit and split it into subcircuits containing Hadamards and subcircuits containing CNOT, S and T gates. One can then attempt to reduce the number of T gates within just the latter type of subcircuit. Amy and Mosca recently showed that this restricted problem is formally equivalent to error decoding on a class of Reed-Muller codes [22], which is in turn equivalent to finding the symmetric tensor rank of a 3-tensor [23]. Unfortunately, even this easier sub-problem is difficult to solve optimally. Nevertheless, it is more amenable to efficient solvers that offer reductions in T count. Amy and Mosca proved that an n-qubit subcircuit (containing CNOT, S and T gates) has an optimal decomposition into $n^2/2 + O(n)$ T gates. At the time, known efficient compilers could only promise an output circuit with no more than $O(n^3)$ T gates. Later, Campbell and Howard [24] sketched a compiler that is efficient and promises an output circuit with at most $n^2/2 + O(n)$ T gates. This shows efficient compilers can in this sense be "near-optimal" with respect to worst case scaling. On the mathematical level, Campbell and Howard exploited a previously known efficient and optimal solver for a related 2-tensor problem [25] but suitably modified so that it nearly-optimally solves the required 3-tensor problem.

This paper develops several different compilers that have polynomial runtime in n and are near-optimal in the above sense when restricted to CNOT+T circuits. We modify the compiler to also accommodate Hadamard gates using a gadgetisation trick that requires additional resources (measurements, feed-forward and ancillas) and find that it performs well in practice. We provide the first implementations of such compilers (the source code is available here [44]) and compare performance against: a family of random circuits; and a library of benchmark circuits that implement actual quantum algorithms. For random circuits, we observe $O(n^2)$ scaling in T count for all variants of our compiling approach

* leheyfron1@sheffield.ac.uk

† earltcampbell@gmail.com

compared with $O(n^3)$ scaling for compilers based on earlier work. Quantum algorithms are highly structured and far from random, so the number of T gates can not be meaningfully compared with the worst case scaling. Instead, we benchmark against the best previously known results and found on average a 20% T count reduction. In one instance, our compiler gave a 51% T count reduction and it performed better than previous results for all but one of the benchmarked circuits. Of course, the T count is not the only metric relevant to gate synthesis. We discuss the limitations of the T count, as well as other metrics in section IV C.

All of the near-optimal compilers described in this paper look for inspiration in algorithms for the related 2-tensor problem, which we call Lempel's algorithm. We give specific details for a compiler here called TOOL (Target Optimal by Order Lowering) that comes in two different flavours (with and without feedback). The TOOL compilers can be considered concrete versions of the approach outlined by Campbell and Howard [24]. Also described in this paper is the TODD (Third Order Duplicate and Destroy) compiler, which is again inspired by Lempel but in a more direct and elegant way than TOOL. In benchmarking, we find that TODD often achieved even lower T count than TOOL.

I. PRELIMINARIES

The Pauli group on n qubits $\mathcal{P}^n$ is the set of all n-fold tensor products of the single qubit Pauli operators $\{X, Y, Z, \mathbb{I}\}$ with allowed coefficients $\in \{\pm 1, \pm i\}$. The k^{th} level of the *Clifford hierarchy* $\mathcal{C}_k^n$ is defined as follows,

$$\mathcal{C}_k^n = \{U \mid U\mathcal{P}^n U^\dagger \subseteq \mathcal{C}_{k-1}^n\}, \tag{1}$$

with recursion terminated by $\mathcal{C}_1^n = \mathcal{P}^n$. The Clifford group on n qubits $\mathcal{C}^n$ is the normalizer of $\mathcal{P}^n$. We define $\mathcal{D}_k^n$ to be the diagonal elements of $\langle CNOT, T\rangle$. We will omit the superscript n when the number of qubits is obvious from context. We define Clifford to be any generating set for the Clifford group on n qubits such as $\{CNOT, H, S\}$. We define the CNOT + T gate set to be $\{\text{CNOT}, S, T\}$, where we include the phase gate $S = T^2$ as a separate gate due to the magic states cost model for gate synthesis [9]. A quantum circuit decomposition for a unitary U is denoted $\mathcal{U}$; conversely we say that $\mathcal{U}$ implements U. Similarly, a circuit $\mathcal{E}$ implements non-unitary channel $\rho \to \varepsilon(\rho)$. We refer to a circuit $\mathcal{U}$ that implements a $U \in \mathcal{D}_3$ as a *diagonal* CNOT + T *circuit*.

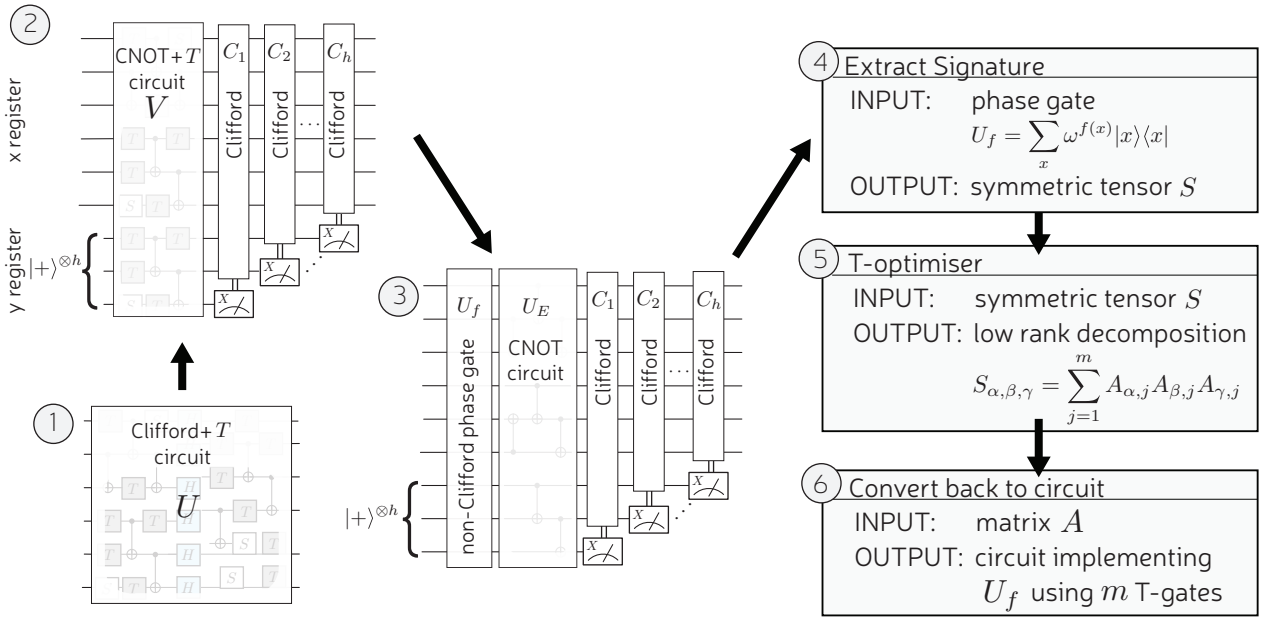

FIG. 1. The high level work-flow of the T gate optimization protocol is shown. A Clifford + T circuit is converted to the CNOT+T gate set by introducing ancillas and performing classically controlled Clifford gates. A non-Clifford phase gate is extracted, which maps to a signature tensor upon which the core optimization algorithm is performed. The optimized symmetric tensor decomposition is then converted back into a circuit of the form in panel 2) yielding an implementation of the original Clifford + T circuit with reduced T count.

II. WORK-FLOW OVERVIEW

In this section, we give a high level work-flow of our approach to compiling as sketched in Fig. 1. In stages 1-3, some simple circuit preprocessing is performed so that a Clifford+T circuit is converted into a form where the only non-Clifford part is a diagonal CNOT+T gate (an element of $\mathcal{D}_3$). Subsection II A describes this preprocessing. In stages 4-6, the technically difficult aspect of compiling is addressed using a series of different algebraic representations of the circuit and these stages are described in Subsection II B.

A. Circuit preprocessing

The input circuit $\mathcal{U}_{\text{in}} \in \langle \text{Clifford}, T \rangle$ implements some unitary U. It acts on a register we denote x, which is composed of n qubits and spans the Hilbert space $\mathcal{H}_{\text{x}}$. The output of our compiler is a circuit $\mathcal{E}_{\text{out}}$ composed of Clifford and T gates but additionally allows: the preparation of $|+\rangle$ states; measurement in the Pauli-X basis, and classical feedforward. To account for ancilla $|+\rangle$ qubits, we include a register labelled y that is composed of h qubits and spans the Hilbert space $\mathcal{H}_{\text{y}}$. The circuit $\mathcal{E}_{\text{out}}$ will realise the input unitary after the y register is traced out

$$\text{Tr}_{\text{y}}[\varepsilon_{\text{out}}(\rho_{\text{x}})] = \text{Tr}_{\text{y}}[\varepsilon_{\text{post}}(V(\rho_{\text{x}} \otimes |+\rangle \langle +|^{\otimes h}))V^\dagger)], \tag{2}$$

$$= U\rho_{\text{x}}U^\dagger, \tag{3}$$

where ρ_{x} is the density matrix for an arbitrary input pure state on $\mathcal{H}_{\text{x}}$. Furthermore, $V \in \mathcal{C}_3$ is the unitary portion of $\mathcal{E}_{\text{out}}$, and $\varepsilon_{\text{post}}$ is a quantum channel that is associated with the sequence of Pauli-X measurements and subsequent classically controlled Clifford gates, $C_1, C_2, \ldots, C_h$, seen in Fig. 1.

We emphasize that later stages of compiling will make use of a framework valid only for CNOT + T circuits, which makes Hadamard gates an obstacle. There are two commonly used methods for dealing with Hadamard gates: first, we can partition the quantum circuit into alternating $\langle CNOT, T \rangle$ and $\langle H \rangle$ subcircuits and optimize each CNOT + T subcircuit independently [20]. The second way is to replace each Hadamard gate with a gadget (see for example references [26, 27]) that makes use of extra resources (ancillas, measurements and feedforward). The central portion of the gadget contains all of the non-Clifford behaviour and is in the CNOT + T gate set, so is directly compatible with our T-optimizers. The remainder of this section focusses on the second method (Hadamard gadgetization), but we discuss the Hadamard-bounded partitioning method in more detail in appendix A.

Each[43] of the h Hadamard gates is replaced by a *Hadamard-gadget* (as shown in panel 1) of Fig. 2. A Hadamard-gadget consists of a CNOT + T block followed by a Pauli-X gate conditioned on the outcome of measuring a *Hadamard-ancilla* (a qubit in the y register initialized in the $|+\rangle$ state) in the Pauli-X basis, so the size of the y register is h. After Hadamard-gadgetisation, we commute the classically controlled Pauli-X gates to the end of the circuit, starting with the right-most and iteratively working our way left (see panel 3 of Fig. 2). The end result is a circuit composed of a single CNOT+T block on $n+h$ qubits, followed by a sequence of classically controlled Clifford operators conditioned on Pauli-X measurements. The latter sequence of non-unitary gates constitutes the circuit $\mathcal{E}_{\text{post}}$. This method of circumventing Hadamards is preferred over forming Hadamard-bounded partitions as in previous works [20] because it allows us to convert most of the input circuit into the optimization-compatible gate set, which we find leads to better performance of the *T-Optimiser* subroutine (see appendix A for numerical evidence of this).

Once the internal Hadamards are removed, we are left with a CNOT + T circuit that implements unitary V, whose action on the computational basis is fully described [20, 22, 24, 28] by two mathematical objects: a *phase function*, $f : \mathbb{Z}_2^n \mapsto \mathbb{Z}_8$, and an invertible matrix $E \in \mathbb{Z}_2^{(n,n)}$, such that

$$V |\mathbf{x}\rangle = \omega^{f(\mathbf{x})} |E\mathbf{x}\rangle \tag{4}$$

where $\omega = e^{i\frac{\pi}{4}}$. It has been shown [22, 24] that $V = U_E U_f$ where $U_f \in \mathcal{D}_3$ can be implemented with a diagonal CNOT + T circuit and gives the phase

$$U_f |\mathbf{x}\rangle = \omega^{f(\mathbf{x})} |\mathbf{x}\rangle , \tag{5}$$

and U_E can be implemented with CNOTs.

B. Diagonal CNOT+T Framework

In section II A, we isolated all the non-Clifford behaviour of a Clifford + T circuit within a diagonal CNOT + T circuit defined on a larger qubit register. This method allows us to map the T gate optimization problem for any Clifford + T circuit to the following.

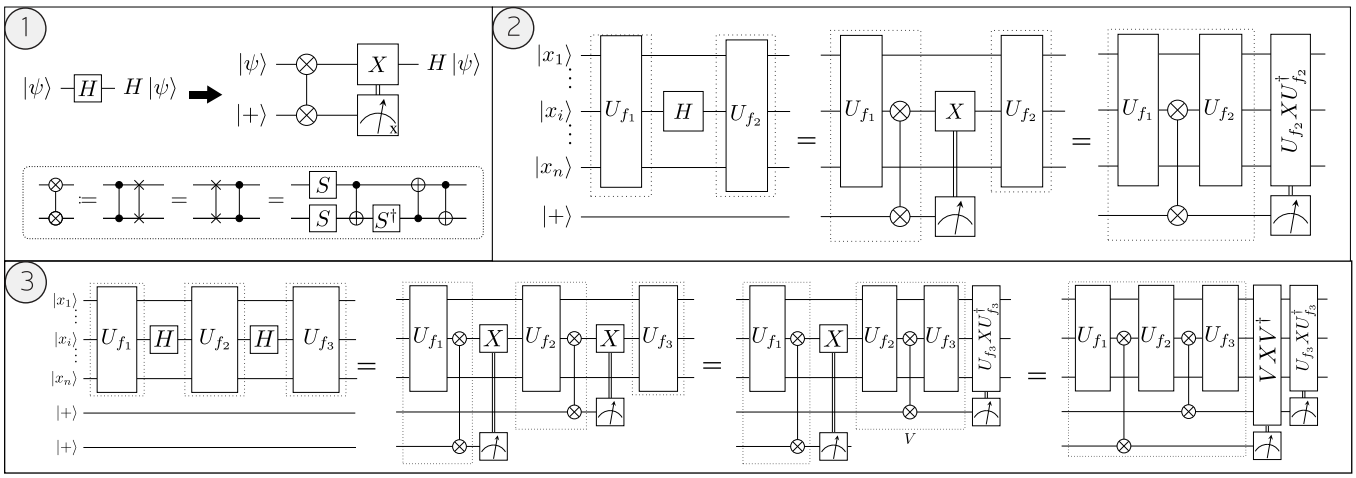

FIG. 2. Hadamard gates are replaced by Hadamard-gadgets according to the rewrite in the upper part of panel 1). In the lower part, we define notation for the phase-swap gate and provide an example decomposition into the CNOT + T gate set. Panel 2) shows an example of a Hadamard gate swapped for a Hadamard-gadget where the classically controlled Pauli-X gate is commuted through U_{f_2} to the end. The CNOT + T -only region increases as shown by the dotted lines. As $U_{f_2} \in \mathcal{C}_3$, it follows that $U_{f_2} X U_{f_2}^\dagger \in \mathcal{C}_2$ as per equation (1), so has a T-count of 0. The example in panel 3) shows the same process as 2) but for 2 internal Hadamards. As $\mathcal{D}_3$ is a group, the operator $V \in \mathcal{D}_3$ and the second Pauli-X gate can also commute to the end to form a Clifford. This leads to a decomposition of the form in panel 2) of Fig. 1.

Problem II.1. ***(T-OPT)*** *Given a unitary $U_f \in \mathcal{D}_3$, find a circuit decomposition $\mathcal{U}_f \in \langle CNOT, T, S\rangle$ that implements U_f with minimal uses of the T gate.*

This section describes how we map the T-OPT problem from the quantum circuit picture to an algebraic problem following stages 4-6 of Fig. 1. Throughout this section we use the framework for diagonal CNOT+T circuits (also called *linear phase operators* [22]) introduced in reference [28] and built upon in [20, 22, 24]. We proceed by recalling from equation (5) that the action of any $U_f \in \mathcal{D}_3$ on the computational basis is given by $U_f |\mathbf{x}\rangle = \omega^{f(\mathbf{x})} |\mathbf{x}\rangle$ and that U_f is completely characterized by the phase function, f. A phase function can be decomposed into a sum of linear, quadratic and cubic monomials on the Boolean variables x_i. Each monomial of order r has a coefficient in $\mathbb{Z}_8$ and is weighted by a factor 2^{r-1}, as in the following:

$$f(\mathbf{x}) = \sum_{\alpha=1}^{n} l_\alpha x_\alpha + 2\sum_{\alpha<\beta}^{n} q_{\alpha,\beta} x_\alpha x_\beta + 4\sum_{\alpha<\beta<\gamma}^{n} c_{\alpha,\beta,\gamma} x_\alpha x_\beta x_\gamma \pmod 8, \tag{6}$$

where $l_\alpha, q_{\alpha,\beta}, c_{\alpha,\beta,\gamma} \in \mathbb{Z}_8$. We refer to decompositions of f that take the form of equation (6) as *weighted polynomials* as in reference [24], in which it was shown that $U_{2f} = U_f^2 \in \mathcal{C}_2$ for any weighted polynomial, f. This implies that any two unitaries with weighted polynomials whose coefficients all have the same parity are Clifford equivalent. Note that the weighted polynomial can be lifted directly from the circuit definition of U_f if we work in the $\{T, CS, CCZ\}$ basis, as each kind of gate corresponds to the linear, quadratic and cubic terms, respectively.

In stage 4 of Fig. 1, we define the *signature tensor*, $S^{(U_f)} \in \mathbb{Z}_2^{(n,n,n)}$, to be a symmetric tensor of order 3 whose elements are equal to the parity of the weighted polynomial coefficients of U_f according to the following relations:

$$S_{\sigma(\alpha,\alpha,\alpha)} = S_{a,a,a} = l_\alpha \pmod 2 \tag{7a}$$

$$S_{\sigma(\alpha,\beta,\beta)} = S_{\sigma(\alpha,\alpha,\beta)} = q_{\alpha,\beta} \pmod 2 \tag{7b}$$

$$S_{\sigma(\alpha,\beta,\gamma)} = c_{\alpha,\beta,\gamma} \pmod 2 \tag{7c}$$

for all permutations of the indices, denoted σ. It follows that any two unitaries with the same signature tensor are Clifford equivalent.

We recall the definition of gate synthesis matrices from reference [24], where a matrix, A in $\mathbb{Z}_2^{(n,m)}$, is a gate synthesis matrix for a unitary U_f if it satisfies,

$$f(\mathbf{x}) = |A^T \mathbf{x}| \pmod 8 = \sum_j \left[\bigoplus_i A_{i,j} x_i \right] \pmod 8 \tag{8}$$

where $|.|$ is the Hamming weight of a binary vector. Notice that inside the square brackets is evaluated modulo 2 and outside is evaluated modulo 8.

Obtaining a gate synthesis matrix from a quantum circuit is best understood via the phase polynomial representation. A phase polynomial of a phase function, f, is a set, $P_f = \{\{\lambda_1, a_1\}, \{\lambda_2, a_2\}, \ldots, \{\lambda_p, a_{|P|}\}\}$, of linear boolean functions $\lambda_k(\mathbf{x})$, together with coefficients $a_k \in \mathbb{Z}_8$ such that

$$f(\mathbf{x}) = \sum_{k=1}^{|P_f|} a_k \lambda_k(\mathbf{x}) \pmod 8. \tag{9}$$

A phase polynomial can be extracted from a diagonal CNOT + T circuit by tracking the action of each gate on the computational basis states through the circuit [20, 28]. We then map P_f to an A matrix with a procedure such as the following. Start with an empty A matrix. Then for each $\{\lambda_k, a_k\} \in P_f$,

1. Define column vector, $\mathbf{v} \in \mathbb{Z}_2^n$, such that $\lambda_k(\mathbf{x}) = v_1 x_1 \oplus v_2 x_2 \oplus \cdots \oplus v_n x_n$.
2. Add a_k copies of $\mathbf{v}$ to the right-hand end of A.

We define a *proper* gate synthesis matrix to be an A matrix with no all-zero or repeated columns, and we define the function PROPER such that $A' = \text{PROPER}(A)$ is the proper gate synthesis matrix formed by removing all all-zero columns and pairs of repeated columns from A. The purpose of this function is to strip away the Clifford behaviour from the gate synthesis matrix.

We will exploit the key property of A matrices described in the following lemma, which is a corollary of lemma 2 of reference [28].

Lemma II.1. *Let $U_f \in \mathcal{D}_3$ be a unitary with phase function $f(\mathbf{x}) = |A^T\mathbf{x}|$ and $A' = \text{PROPER}(A) \in \mathbb{Z}_2^{(n,m)}$. It follows that one can generate a circuit that implements U_f with $m = \text{col}(A')$ uses of the T gate.*

Proof. First, we note from the definition of A in equation (8) that the j^{th} column of A leads to a factor of $\omega^{\lambda_j(\mathbf{x})}$ appearing in the diagonal elements of U_f as written in equation (5), where λ_j is a reversible linear Boolean function given by,

$$\lambda_j(\mathbf{x}) = A_{1,j}x_1 \oplus A_{2,j}x_2 \oplus \cdots \oplus A_{n,j}x_n. \tag{10}$$

The action of a circuit generated by CNOT gates on computational basis state $|\mathbf{x}\rangle$ is to replace the value of each qubit with a reversible linear Boolean function on $x_1, x_2, \ldots, x_n$. Next, we show how to add the phase $\omega^{\lambda_j(\mathbf{x})}$. We define B_j to be a CNOT unitary such that after applying B_j the first qubit is mapped $|x_1\rangle \to |\lambda_j(\mathbf{x})\rangle$. A T gate subsequently applied to this qubit will now produce the desired phase. We then uncompute B_j by reversing the order of the CNOT gates. This procedure is repeated for every j until all columns of A have been implemented in this way. Only the columns of A that also appear in A' require the use of a T gate as all other columns have duplicates, where any pair of duplicates can be implemented by replacing the T gate with an S gate in the above procedure. Therefore the T count is equal to $m = \text{col}(A')$. □

The signature tensor of U_f can be determined from an A matrix of U_f using the following relation,

$$S^{(A)}_{\alpha,\beta,\gamma} = \sum_{j=1}^{m} A_{\alpha,j}A_{\beta,j}A_{\gamma,j} \pmod 2. \tag{11}$$

Therefore, the gate synthesis problem T-OPT reduces to the following tensor rank problem.

Problem II.2. ***(3-STR)*** *Given a symmetric tensor of order 3, $S \in \mathbb{Z}_2^{(n,n,n)}$, find a matrix $A \in \mathbb{Z}_2^{(n,m)}$ that satisfies equation* (11) *with minimal m.*

Any algorithm attempting to solve 3-STR can be used in stage 5 of Fig. 1. The observation that T-OPT reduces to 3-STR is not new as it follows directly from earlier work. Amy and Mosca [22] proved that T-OPT is equivalent to minimum distance decoding of the punctured Reed-Muller code of order $n-4$ and length n (often written as $RM^*(n-4, n)$). Furthermore, in 1980 Seroussi and Lempel [23] recognised that this Reed-Muller decoding problem is equivalent to 3-STR and conjectured that this is a hard computational task. A non-symmetric generalisation of 3-STR has been proved to be NP-complete [29], giving further weight to the conjecture. This imposes a practical upper bound on the number of qubits, n_{RM}, over which circuits can be optimally synthesized.

The problem 3-STR is closely related to

Problem II.3. ***(2-STR)*** *Given a symmetric tensor of order 2, $S \in \mathbb{Z}_2^{(n,n)}$, find a matrix $A \in \mathbb{Z}_2^{(n,m)}$ that satisfies*

$$S^{(A)}_{\alpha,\beta} = \sum_{j=1}^{m} A_{\alpha,j}A_{\beta,j} \pmod 2. \tag{12}$$

with minimal m.

This could also be stated as a matrix factorisation $S = AA^T$ problem. As such, we say any A satisfying $S = AA^T$ is a factor of S and a minimal factor is one with the minimum possible number of columns. As is often the case in complexity theory, the matrix variant of the problem is considerably simpler than the higher order tensor variant. Lempel gave an algorithm that finds an optimal solution to 2-STR in polynomial time [25]. We call this Lempel's factoring algorithm and for completeness describe it in App. B. Our main strategy to T count optimisation is to take insights from Lempel's algorithm for 2-STR and apply them to 3-STR. In doing so, our compilers will be efficient but lose the promise of optimality, instead providing approximate solutions to 3-STR and T-OPT.

In the final stage (see 6 of Fig. 1), we map the output matrix of stage 5 back to a diagonal CNOT + T circuit, $\mathcal{U}_{f'}$, that comprises m instances of the T gate using lemma II.1. The circuit $\mathcal{U}_{f'}$ implements a unitary $U_{f'} = U_f U_{\text{Clifford}}$, where U_{Clifford} is a diagonal Clifford factor. The input weighted polynomial stored since step 4 contains sufficient information to generate a circuit for $U^\dagger_{\text{Clifford}}$ (see appendix D), hence we recover the original unitary, $U_f = U_{f'} U^\dagger_{\text{Clifford}}$. The final part of step 6 constitutes replacing $\mathcal{U}_f$ with $(\mathcal{U}^\dagger_{\text{Clifford}} \circ \mathcal{U}_{f'})$. At this stage, the protocol terminates returning the final output, $\mathcal{E}_{\text{out}} = (\mathcal{U}^\dagger_{\text{Clifford}} \circ \mathcal{U}_{f'} \circ \mathcal{U}_E \circ \mathcal{E}_{\text{post}})$.

III. *T-OPTIMISER*

Until now the *T-optimiser* subroutine of our protocol has been treated as a black box whose input is a signature tensor S and the output is a gate synthesis matrix A with few columns. In this section, we describe the inner workings of the various *T-optimiser*s we have implemented in this work.

A. Reed-Muller decoder (RM)

Although Reed-Muller decoding is believed to be hard, a brute force solver can be implemented for a small number of qubits. We implement such a brute force decoder and found its limit to be $n_{\text{RM}} = 6$. To gain some intuition for the complexity of the problem, consider the following. The number of codespace generators for $RM^*(n-4, n)$ is equal to $N_G = \sum_{r=1}^{n-4} \binom{n}{r}$. Therefore, the size of the search space is $N_{\text{search}} = 2^{N_G}$. On a processor with a clock speed of 3.20GHz, generously assuming we can check one codeword per clock cycle, it would take over 91 years to exhaustively search this space for $n = 7$. Performing the same back-of-the-envelope calculation for $n = 6$, it would take $\approx 7 \times 10^{-4}$ seconds. In practice, we find the brute force decoder executes in around 10 minutes for $n = 6$, so the time for $n = 7$ would be significantly worse. Clearly, we need to develop heuristics for this problem.

B. Recursive Expansion (RE)

The simplest means of efficiently obtaining an A matrix for a given signature tensor S is to make use of the modulo identity $2ab = a + b - a \oplus b$. More concretely, for each non-zero coefficient in the weighted polynomial l_α, $q_{\alpha,\beta}$, $c_{\alpha,\beta,\gamma}$, make the following substitutions to the corresponding monomials:

$$x_\alpha \to x_\alpha, \tag{13}$$

$$2x_\alpha x_\beta \to x_\alpha + x_\beta - (x_\alpha \oplus x_\beta), \tag{14}$$

$$4x_\alpha x_\beta x_\gamma \to x_\alpha + x_\beta + x_\gamma - (x_\alpha \oplus x_\beta) - (x_\alpha \oplus x_\gamma) - (x_\beta \oplus x_\gamma) + (x_\alpha \oplus x_\beta \oplus x_\gamma), \tag{15}$$

from which the corresponding A matrix can be easily extracted. We call this the *recursive expansion* (RE) algorithm, which has been shown to yield worst-case T counts of $O(n^3)$. It is straightforward to understand this cubic scaling because any proper gate synthesis matrix resulting from the RE algorithm may include any column of Hamming weight 3 or less. There are $\sum_{k=1}^{3} \binom{n}{k} = O(n^3)$ such columns so from lemma II.1 there can be at most $O(n^3)$ T gates in the corresponding circuit decomposition.

C. Target Optimal by Order Lowering (TOOL)

Campbell and Howard [24] proposed an efficient heuristic for T-OPT that requires at most $O(n^2)$ T gates compared to $O(n^3)$ of the best previous (RE) optimizer. In the quantum circuit picture, the algorithm involves decomposing the input CNOT + T circuit into a cascade of control-$U_{2\tilde{f}}$ operators where $\tilde{f}$ is quadratic rather than cubic. Lowering the order in this way means that each control-$U_{2\tilde{f}}$ can be synthesized both efficiently and optimally using Lempel's factoring algorithm. For this reason we call it the *Target Optimal by Order Lowering* (TOOL) algorithm. Fig. 3 shows a single step of how TOOL pulls out a single control-$U_{2\tilde{f}}$ operator, reducing the number of qubits non-trivially affected by the remaining unitary. The process is repeated until the circuit is small enough to be solved using the RM algorithm. The core

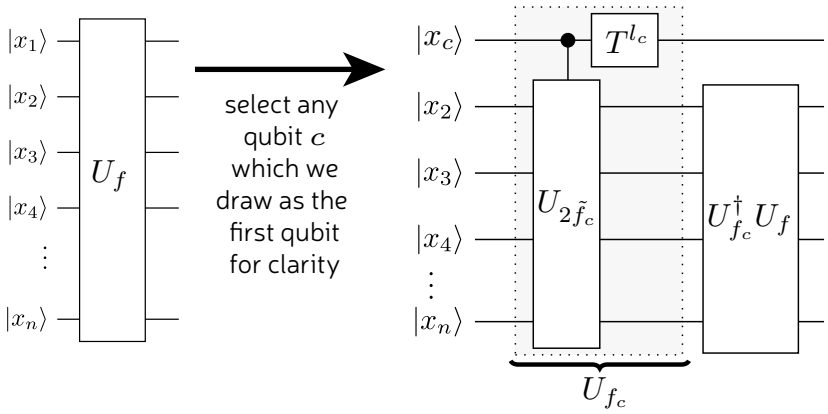

FIG. 3. A sketch of one round of TOOL (without feedback). We identify a sub-circuit U_{f_c} with a single control qubit and then use that such a subcirciut can be efficiently and optimally compiled using Lempel's algorithm. The remaining circuit $U_{f_c}^\dagger U_f$ contains one fewer qubit and so the process can be iterated until the circuit is down to 6 qubits when it can be optimally compiled by brute force.

of the algorithm was already outlined in previous work [24] but for completeness App. C describes both plain TOOL and a variant called TOOL (with feedback). This paper presents the first numerical results obtained from an implementation of TOOL.

D. Third Order Duplicate and Destroy (TODD)

In this section, we present an algorithm based on Lempel's factoring algorithm [25] that is extended to work for order 3 tensors. Since this algorithm does not appear in any previous work, we will provide an extended explanation here. This algorithm requires some initial A matrix to be generated by another algorithm such as RE or TOOL, then it reduces the number of columns of the initial gate synthesis matrix iteratively until exit. In section IV, we present numerical evidence that it is the best efficient solver of the T-OPT problem developed so far. We call this the *Third Order Duplicate and Destroy* (TODD) algorithm because, much like the villainous Victorian barber, it shaves away at the columns of the input A matrix iteratively until the algorithm finishes execution. Pseudo-code is provided in App. E.

We begin by introducing the key mechanism through which TODD reduces the T count of quantum circuits: by *destroying* pairs of duplicate columns of a gate synthesis matrix, a process through which the signature tensor is unchanged, as shown in the following lemma.

Lemma III.1. *Let $A \in \mathbb{Z}^{(n,m)}$ be a gate synthesis matrix whose a^{th} and b^{th} columns are duplicates. Let $A_{\text{des}} \in \mathbb{Z}^{(n,m-2)}$ be a gate synthesis matrix formed by removing the a^{th} and b^{th} columns of A. It follows that $S^{(A)} = S^{(A_{\text{des}})}$ for any such A and A_{des}.*

Proof. We start by writing the signature tensor in terms of the elements of A according to equation (11),

$$S^{(A)}_{\alpha,\beta,\gamma} = \sum_{k=1}^{m} A_{\alpha,k} A_{\beta,k} A_{\gamma,k} \pmod 2, \tag{16}$$

and separating the terms associated with a, b from the rest of the summation,

$$S^{(A)}_{\alpha,\beta,\gamma} = \left(\sum_{j \in \mathcal{J}} A_{\alpha,j} A_{\beta,j} A_{\gamma,j} \right) + A_{\alpha,a} A_{\beta,a} A_{\gamma,a} + A_{\alpha,b} A_{\beta,b} A_{\gamma,b} \pmod 2, \tag{17}$$

where $\mathcal{J} = [1, m] \setminus \{a, b\}$, so that

$$S^{(A)}_{\alpha,\beta,\gamma} = S^{(A_{\text{des}})}_{\alpha,\beta,\gamma} + A_{\alpha,a} A_{\beta,a} A_{\gamma,a} + A_{\alpha,b} A_{\beta,b} A_{\gamma,b} \pmod 2, \tag{18}$$

As stated in the lemma, the a^{th} and b^{th} columns of A are duplicates and so

$$A_{i,a} = A_{i,b} \; \forall \; i \in [1, n] \, . \tag{19}$$

Now substitute equation (19) into equation (18),

$$S^{(A)}_{\alpha,\beta,\gamma} = S^{(A_{\text{des}})}_{\alpha,\beta,\gamma} + 2A_{\alpha,a}A_{\beta,a}A_{\gamma,a} \pmod 2 \tag{20}$$

$$= S^{(A_{\text{des}})}_{\alpha,\beta,\gamma} \pmod 2 \tag{21}$$

where the last step follows from modulo 2 addition. □

Lemma III.1 gives us a simple means to remove columns from a gate synthesis matrix by destroying pairs of duplicates columns and thereby reducing the T count of a CNOT + T circuit by 2. However, it is often the case that the A matrix does not already contain any duplicate columns. Therefore, we wish to perform some transformation: $A \to A'$ such that

(a) A' has duplicate columns;

(b) the transformation preserves the signature tensor of A.

In the following lemma we introduce a class of transformations that *duplicate* a particular column of an A matrix such that property (a) is met. We then use lemma III.3 to establish what conditions must be satisfied for the duplication transformation to have property (b).

Lemma III.2. *Let $A \in \mathbb{Z}_2^{(n,m)}$ be a proper gate synthesis matrix. For some choice of a and b, let $\mathbf{c}_a(A)$ and $\mathbf{c}_b(A)$ denote the a^{th} and b^{th} columns of A and define $\mathbf{z} = \mathbf{c}_a(A) \oplus \mathbf{c}_b(A)$. Let $\mathbf{y} \in \mathbb{Z}_2^m$ be any vector such that $y_a \oplus y_b = 1$. We consider duplication transformations of the form $A \to A' = A \oplus \mathbf{z}\mathbf{y}^T$. It follows that the a^{th} and b^{th} columns of A' are duplicates and so property (a) holds.*

Proof. We begin by finding expressions for the matrix elements of A' in terms of A, $\mathbf{z}$ and $\mathbf{y}$,

$$A'_{i,j} = A_{i,j} \oplus z_i y_j, \tag{22}$$

and substitute the definition of $\mathbf{z}$,

$$A'_{i,j} = A_{i,j} \oplus (A_{i,a} \oplus A_{i,b}) y_j. \tag{23}$$

Now we can find the elements of the columns a and b of A',

$$A'_{i,a} = A_{i,a} \oplus (A_{i,a} \oplus A_{i,b}) y_a, \tag{24}$$

$$A'_{i,b} = A_{i,b} \oplus (A_{i,a} \oplus A_{i,b}) y_b. \tag{25}$$

We substitute in the condition $y_b = y_a \oplus 1$ into equation (25),

$$\begin{aligned} A'_{i,b} &= A_{i,b} \oplus (A_{i,a} \oplus A_{i,b})(y_a \oplus 1) \\ &= A_{i,b} \oplus (A_{i,a} \oplus A_{i,b}) y_a \oplus A_{i,a} \oplus A_{i,b} \\ &= A_{i,a} \oplus (A_{i,a} \oplus A_{i,b}) y_a \\ &= A'_{i,a}, \end{aligned} \tag{26}$$

where the two $A_{i,b}$ terms cancel in the second step of equation (26). □

Lemma III.3. *Consider a duplication transformation of the form $A \to A' = A \oplus \mathbf{z}\mathbf{y}^T$ where $\mathbf{z}$, $\mathbf{y}$ are vectors of appropriate length. It follows that $S^{(A)} = S^{(A')}$ (satisfying property (b)) if the following conditions hold true:*

C1: $|\mathbf{y}| = 0 \pmod 2$

C2: $A\mathbf{y} = \mathbf{0}$

C3: $\chi(A, \mathbf{z})\, \mathbf{y} = \mathbf{0}$.

where we define $\chi(A, \mathbf{z})$ as follows. Given some gate synthesis matrix, A, and a column vector $\mathbf{z} \in \mathbb{Z}_2^n$ let χ be a matrix with rows labelled by (α, β, γ) and of the form

$$\mathbf{R}_{\alpha,\beta,\gamma} = (z_\alpha \mathbf{r}_\beta \wedge \mathbf{r}_\gamma) \oplus (z_\beta \mathbf{r}_\gamma \wedge \mathbf{r}_\alpha) \oplus (z_\gamma \mathbf{r}_\alpha \wedge \mathbf{r}_\beta) \tag{27}$$

where $\mathbf{r}_\alpha$ is the α^{th} row of A, and $\mathbf{x} \wedge \mathbf{y}$ is the element-wise product of vectors $\mathbf{x}$ and $\mathbf{y}$. The order of the rows in χ is unimportant, but must include every choice of $\alpha, \beta, \gamma \in \mathbb{Z}_n$ with no pair of indices being equal.

Proof. We begin by finding an expression for $S(A')$ using equation (11),

$$S^{(A')}_{\alpha,\beta,\gamma} = \sum_{j=1}^{m} (A_{\alpha,j} \oplus z_\alpha y_j)\,(A_{\beta,j} \oplus z_\beta y_j)\,(A_{\gamma,j} \oplus z_\gamma y_j) \pmod 2, \tag{28}$$

and expanding the brackets,

$$\begin{aligned} S^{(A')}_{\alpha,\beta,\gamma} = \sum_{j=1}^{m} (&A_{\alpha,j}A_{\beta,j}A_{\gamma,j} \oplus z_\alpha z_\beta z_\gamma y_j \\ &\oplus z_\alpha z_\beta A_{\gamma,j} y_j \oplus z_\beta z_\gamma A_{\alpha,j} y_j \oplus z_\gamma z_\alpha A_{\beta,j} y_j \\ &\oplus z_\alpha A_{\beta,j}A_{\gamma,j} y_j \oplus z_\beta A_{\gamma,j}A_{\alpha,j} y_j \oplus z_\gamma A_{\alpha,j}A_{\beta,j} y_j) \pmod 2. \end{aligned} \tag{29}$$

We can see that the first term of equation (29) summed over all j is equal to $S^{(A)}$, by definition. The task is to show that the remaining terms sum to zero under the specified conditions. Next, we sum over all j and substitute in the definitions of $|\mathbf{y}|$, $A\mathbf{y}$ and $\chi(A, \mathbf{z})\,\mathbf{y}$,

$$S^{(A')}_{\alpha,\beta,\gamma} = S^{(A)}_{\alpha,\beta,\gamma} \oplus z_\alpha z_\beta z_\gamma |\mathbf{y}| \oplus z_\alpha z_\beta \left[A\mathbf{y}\right]_\gamma \oplus z_\beta z_\gamma \left[A\mathbf{y}\right]_\alpha \oplus z_\gamma z_\alpha \left[A\mathbf{y}\right]_\beta \oplus (\mathbf{R}_{\alpha,\beta,\gamma} \cdot \mathbf{y}). \tag{30}$$

By applying condition *C1*, the second term is eliminated; by applying condition *C2*, the next three terms are eliminated, and by applying condition *C3*, the final term is eliminated. □

Having shown how to duplicate and destroy columns of a gate synthesis matrix, we are ready to describe the TODD algorithm, presented as pseudo-code in algorithm 1. Given an input gate synthesis matrix A with signature tensor S, we begin by iterating through all column pairs of A given by indices a, b. We construct the vector $\mathbf{z} = \mathbf{c}_a \oplus \mathbf{c}_b$ where $\mathbf{c}_j$ is the j^{th} column of A, as in lemma III.2. We check to see if the conditions in lemma III.3 are satisfied for $\mathbf{z}$ by forming the matrix,

$$\tilde{A} = \begin{pmatrix} A \\ \chi(A, \mathbf{z}) \end{pmatrix}. \tag{31}$$

Any vector, $\mathbf{y}$, in the null space of $\tilde{A}$ simultaneously satisfies *C2* and *C3* of lemma III.3. We scan through the null space basis until we find a $\mathbf{y}$ such that $y_a \oplus y_b = 1$. At this stage we know that we can remove at least one column from A, depending on the following cases

i: If $|\mathbf{y}| = 0 \pmod 2$ then condition *C1* is satisfied and we can perform the duplication transformation from lemma III.3;

ii: If $|\mathbf{y}| = 1 \pmod 2$ then we force *C1* to be satisfied by appending a 1 to $\mathbf{y}$ and an all-zero column to A before applying the duplication transformation.

Finally, we use the function PROPER as in App. E to destroy all duplicate pairs to maximize efficiency. In case *i*, at least two columns have been removed and in case *ii* at least one column has been removed [45]. This reduces the number of columns of A and therefore the T count of U_f. We now start again from the beginning, iterating over columns of the new A matrix. The algorithm terminates if every column pair has been exhausted without success.

IV. RESULTS & DISCUSSION

We implemented our compiler, which we call *TOpt*, in C++ including each variant of *T-Optimiser* described in section III, and tested it on two types of benchmark. First, we performed a random benchmark, in which we randomly sampled signature tensors from a uniform probability distribution for a range of n and used them as input for the four versions of *T-optimiser*: RE, TOOL (feedback), TOOL (without feedback) and TODD. The results for the random benchmark are shown in Fig. 4. Second, we tested the compiler on a library of benchmark circuits taken from Dmitri Maslov's Reversible Logic Synthesis Benchmarks Page [30], Matthew Amy's GitHub repository for T-par [42] and Nam et al's GitHub repository [32] for reference [21]. These circuits implement useful quantum algorithms including Galois Field multipliers, integer addition, n^{th} prime, Hamming coding functions and the hidden weighted bit functions. The results for the quantum algorithm benchmark are listed in Table I. For all benchmarks, the results were obtained on the University of Sheffield's Iceberg HPC cluster[31].

A. Random Circuit Benchmark

We performed the random benchmark in order to determine the average case scaling of the T-count with respect to n for each computationally efficient version of *T-optimiser* with results shown in Fig. 4. For both versions of TOOL, we find that the numerical results for the T count follow the expected analytical scaling of $O(n^2)$ and correspondingly the results for RE scales as $O(n^3)$. We see that TODD slightly outperforms the next best algorithm, TOOL (without feedback) and is therefore the preferred algorithm in settings where classical runtime is not an issue. Furthermore, for all compilers the distribution of T-counts (for fixed n) concentrates around the mean value. Fig. 4 includes error bars showing the distribution but they are too small to be clearly visible, so for one data point we highlight this with an inset histogram. Therefore, TODD performs better, not just on average, but on the vast majority of random circuits so far tested. While both have a polynomial runtime, we found TOOL runs faster than TODD. Therefore, TOOL may have some advantage for larger circuits that are impractically large for TODD. However, TODD can always partition a very large circuit into several smaller circuits at the cost of being slightly less effective at reducing T count. Consequently, for very large circuits, it is unclear which compiler will work best and running both is recommended.

The random benchmark effectively uses diagonal CNOT + T circuits. This gate set is not universal and therefore is computationally limited. However, these circuits are generated by $\{T, CS, CCZ\}$, which all commute. This means such circuits lie in the computational complexity class IQP (which stands for *instantaneous quantum polynomial-time*) that feature in proposals for quantum supremacy experiments [26, 34, 35]. Low cost designs of IQP circuits provided by our compiler would therefore be an asset for achieving quantum supremacy.

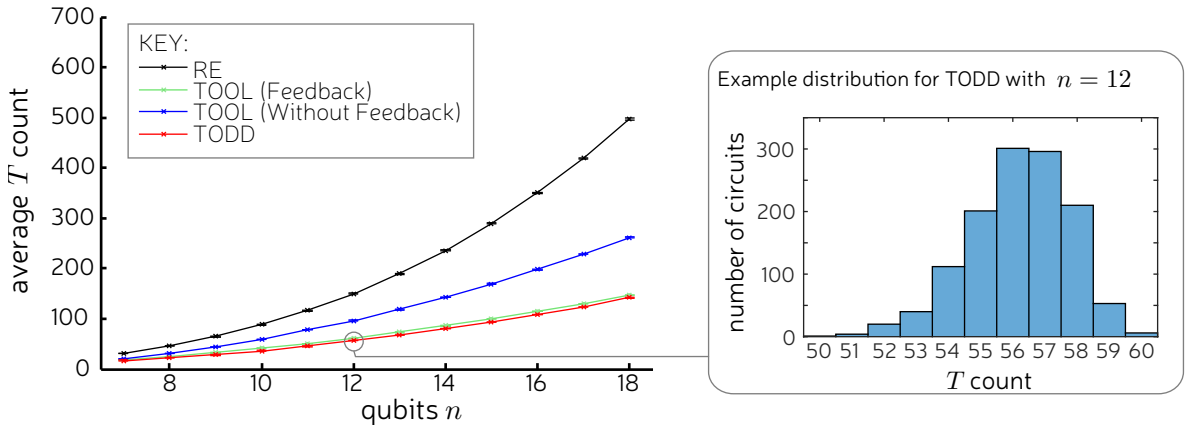

FIG. 4. Circuits generated by the CNOT and T gate were randomly generated for varying number of qubits n then optimized by our implementations of RE, TOOL and TODD. The average T-count for each n over many random circuits are shown on the vertical axis. TODD produces circuit decompositions with the smallest T-counts on average but scales the same as the next best algorithm, TOOL (Feedback). Both of these algorithms are better than RE by a factor n. The difference between the T-counts for TODD and TOOL (Feedback) seem to converge to a constant 5.5 ± 0.7 for large n.

B. Quantum Algorithms Benchmark

The results in Table I show that the TODD algorithm reduced or preserved the T count for every input quantum circuit upon which it was tested, as expected. Additionally, TODD yields a positive saving over the best previous algorithm for all benchmarks except Mod 5_4 with an average and maximum saving of 20% and 51%, respectively. This is immediately useful due to the lower cost associated with solving these problems.

Crucially, the output circuits of our protocol often require a considerable number of ancilla qubits due to our use of Hadamard gadgets. This space-time trade-off is justifiable when the cost of introducing an additional qubit is small in comparison to that of performing an additional T gate [39]. Furthermore, our compilers can be executed with a cap, h_{cap}, on the size of the ancilla register by dividing the circuit into subcircuits containing no more than h_{cap} Hadamard gates. A larger number of Hadamard gates generally leads to an increased classical compilation time for TODD as well as an increased T count for TODD-part (see appendix A), which naturally motivates future investigation into Hadamard gate optimization as a pre-processing step of TOpt-like compilers. Finally, further reductions in the space (and other) resource requirements may be possible by back-substituting the Hadamard gadget identity from Fig. 2 post-optimization.

The TOOL algorithms (with and without feedback) reduced T counts below those of the best previous result for 18% and 30% of the benchmark circuits, respectively. But for the majority, we find that TOOL actually results in negative savings. This seems to contradict the result for the random benchmark (see Fig. 4) in which TOOL (feedback) nearly performs as well as TODD. We offer the following explanation for this apparent contradiction. The circuits generated as

input for the random benchmark typically have optimal T counts close to the worst-case bound of $O(n^2)$. TODD yields T counts very close to optimal because it only terminates when nearly all avenues for T count reduction have been exhausted. The TOOL algorithm outputs T counts below $O(n^2)$, so closely competes with TODD for random circuits. However, for the Clifford + T benchmark, the optimal T count is typically much less than the worst-case $O(n^2)$ bound. It is important to recall at this stage that TOOL is optimal for the special case where the circuit implements a control-Clifford. But even for this special case, TOOL needs to know which qubit is the control qubit in order to take advantage of this special case behaviour. Consequently, a general-purpose automated compiler without prior knowledge about the input quantum circuit must have access to an additional subroutine which determines the control qubit. For general quantum circuits, the task is especially challenging because the circuit must also be optimally partitioned into a sequence of control-Cliffords. As such, we have left this task as an avenue of future work. Our implementation of TOOL uses a naive random control-qubit selection subroutine, so regardless of the low optimal T count, TOOL will often output T counts that remain close to the worst-case of $O(n^2)$. We suggest that this is the principle cause for the relatively poor performance seen in Table I, which has lead to negative savings not only over the best previous result and TODD, but sometimes also over the input circuit, and conclude that a better control-qubit selector would unlock more of TOOL's T-optimizing potential.

C. The T Count and Other Metrics

We acknowledge that the T count does not account for the full space-time cost of quantum computation. Recall that we justified neglecting the cost of Clifford gates due to the high ratio between the cost of the T gate and that of Clifford gates. The full space-time cost is highly sensitive to the architecture of the quantum computer, but for the surface code, this ratio is estimated to be between 50 and 1000 [12, 36–38], depending on architectural assumptions.

Note that while our protocol leads to circuits with low T count, the final output often has an *increased* CNOT count. This is largely due to step 6 of our protocol where we map the phase polynomial back to a quantum circuit using a naive approach. Although T gates cost significantly more than CNOTs individually, the lower bound on number of CNOT gates required to implement high complexity reversible functions exceeds the upper bound on the number of T gates required by an amount that grows exponentially in n [39]. So for large n, our focus should turn instead to CNOT optimization. In this paper, we focus exclusively on T count optimization, which is relevant not just to circuit optimization but also to classical simulation runtime [16, 40, 41] and distillation of magic states [24]. For this reason, we omit the CNOT count from our benchmark tables and leave the problem of optimizing CNOT count as an avenue for future work.

V. CONCLUSIONS & ACKNOWLEDGEMENTS

In this work, we have developed a framework for compiling and optimizing Clifford + T quantum circuits that reduces the T count. This scheme maps the quantum circuit problem to an algebraic problem involving order 3 symmetric tensors, for which we have presented an efficient near-optimal solver, and we have reviewed previous methods. We implemented our protocol in C++ and used it to obtain T count data for quantum circuit benchmarks. Each variant of the compiler has managed to produce quantum circuits for quantum algorithms with lower T-counts than any previous attempts known to us. However, we find that the TODD compiler with Hadamard gadgets performs the best in practice. This lowers the cost of quantum computation and takes us closer to achieving practical universal fault-tolerant quantum computation.

We acknowledge support by the Engineering and Physical Sciences Research Council (EPSRC) through grant EP/M024261/1. We thank Mark Howard and Matthew Amy for valuable discussions, and Dmitri Maslov for comments on the manuscript. We thank Quanlong Wang for spotting an error in an earlier draft of the manuscript.

TABLE I. T-counts of Clifford + T benchmark circuits for the TODD, TOOL(F) (with feedback) and TOOL(NF) (without feedback) variants of the TOpt compiler are shown. Results for other variants can be seen in Table II of appendix A. Columns **n** and $\mathbf{n_h}$ show the number of qubits for the input circuit and the number of Hadamard ancillas, respectively. The T-count for the circuit is given: before optimization (Pre-Opt.); after optimization using the best previous algorithm (Best prev.); and post-optimization using our implementation of TODD, TOOL(F) and TOOL(NF). The best previous algorithm is given in the **Alg.** column where: T-par is from [20]; $\mathrm{RM_m}$ and $\mathrm{RM_r}$ are the majority and recursive Reed-Muller optimizers, respectively, both from [22]; and $\mathrm{Auto_H}$ is the heavy version of the algorithm from [21]. We show the T-count saving for each TOpt variant over the best previous algorithm in the **s** columns and the execution time as run on the Iceberg HPC cluster in the **t** columns. Results where the execution time is marked with † were obtained using an alternative implementation of TODD that is faster but less stable. The row **Positive saving** shows the proportion of the benchmark circuits, as a percentage, for which the corresponding compiler yields a positive saving over the best previous result.

	Pre-Opt.		**Best prev.**		**TOpt**	**TODD**			**TOOL(F)**			**TOOL(NF)**		
Circuit	**n**	**T**	**T**	**Alg.**	$\mathbf{n_h}$	**T**	**t (s)**	**s(%)**	**T**	**t (s)**	**s(%)**	**T**	**t (s)**	**s(%)**
Mod 5_4[42]	5	28	16	T-par	6	16	0.04	0	19	0.38	−18.75	19	0.37	−18.75
8-bit adder[42]	24	399	213	$\mathrm{RM_m}$	71	129	40914.1	39.44	279	71886.1	−30.99	284	55574.6	−33.33
CSLA-$\mathrm{MUX_3}$[32]	16	70	58	$\mathrm{RM_r}$	17	52	30.41	10.34	84	122.95	−44.83	73	84.54	−25.86
CSUM-$\mathrm{MUX_9}$[32]	30	196	76	$\mathrm{RM_r}$	12	72	587.21	5.26	83	2081.13	−9.21	104	340.19	−36.84
GF(2^4)-mult[42]	12	112	68	T-par	7	54	8.88	20.59	75	5.96	−10.29	75	2.38	−10.29
GF(2^5)-mult[42]	15	175	101	$\mathrm{RM_r}$	9	87	66.83	13.86	109	17.6	−7.92	107	28.27	−5.94
GF(2^6)-mult[42]	18	252	144	$\mathrm{RM_r}$	11	126	521.86	12.50	165	82.52	−14.58	157	60.16	−9.03
GF(2^7)-mult[42]	21	343	208	$\mathrm{RM_r}$	13	189	2541.4	9.13	277	226.4	−33.17	209	122.17	−0.48
GF(2^8)-mult[42]	24	448	237	$\mathrm{RM_r}$	15	230	36335.7	2.95	370	379.97	−56.12	281	322.83	−18.57
GF(2^9)-mult[42]	27	567	301	$\mathrm{RM_r}$	17	295	50671.1	1.99	454	1463.02	−50.83	351	816.04	−16.61
GF(2^{10})-mult[42]	30	700	410	T-par	19	350	15860.3†	14.63	550	7074.29	−34.15	434	988.04	−5.85
GF(2^{16})-mult[42]	48	1792	1040	T-par	31		-		1723	75204.8	−65.67	1089	30061.1	−4.71
$\mathrm{Grover_5}$[42]	9	52	52	T-par	23	44	17.07	15.38	106	110.29	−103.85	83	117.39	−59.62
$\mathrm{Hamming_{15}}$ (low)[42]	17	161	97	T-par	34	75	902.69	22.68	161	2787	−65.98	132	1041.22	−36.08
$\mathrm{Hamming_{15}}$ (med)[42]	17	574	230	T-par	85	162	12410.8†	29.57	727	176275	−216.09	277	59112.2	−20.43
$\mathrm{HWB_6}$[30]	7	105	71	T-par	24	51	55.66	28.17	189	140.79	−166.20	149	59.24	−109.86
$\mathrm{Mod\text{-}Mult_{55}}$[42]	9	49	35	$\mathrm{RM_{m\&r}}$	10	17	0.26	51.43	35	5.45	0	19	0.92	45.71
$\mathrm{Mod\text{-}Red_{21}}$[42]	11	119	73	T-par	17	55	25.78	24.66	68	40.82	6.85	71	19.76	2.74
$\mathrm{n^{th}\text{-}prime_6}$[30]	9	567	400	$\mathrm{RM_{m\&r}}$	97	208	37348†	48	830	205869	−107.50	344	135165	14
$\mathrm{QCLA\text{-}Adder_{10}}$[42]	36	238	162	T-par	28	116	5496.66	28.40	167	7544.58	−3.09	180	4560.78	−11.11
$\mathrm{QCLA\text{-}Com_7}$[42]	24	203	94	$\mathrm{RM_m}$	19	59	198.55	37.23	79	420.95	15.96	125	465.41	−32.98
$\mathrm{QCLA\text{-}Mod_7}$[42]	26	413	235	$\mathrm{Auto_H}$	58	165	46574.3	29.79	295	35249.2	−25.53	310	22355.4	−31.91
$\mathrm{QFT_4}$[42]	5	69	67	T-par	39	55	93.65	17.91	67	1602.91	0	59	2756.34	11.94
$\mathrm{RC\text{-}Adder_6}$[42]	14	77	47	$\mathrm{RM_{m\&r}}$	21	37	18.72	21.28	48	1238.12	−2.13	44	81.77	6.38
NC $\mathrm{Toff_4}$[42]	5	21	15	T-par	2	13	$< 10^{-2}$	13.33	14	0.02	6.67	14	0.01	6.67
NC $\mathrm{Toff_5}$[42]	7	35	23	T-par	4	19	0.06	17.39	22	0.24	4.35	22	0.12	4.35
NC $\mathrm{Toff_6}$[42]	9	49	31	T-par	6	25	0.4	19.35	31	1146.04	0	29	0.67	6.45
NC $\mathrm{Toff_{10}}$[42]	19	119	71	T-par	16	55	44.78	22.54	65	1357.98	8.45	67	110.44	5.63
Barenco $\mathrm{Toff_4}$[42]	5	28	16	T-par	3	14	$< 10^{-2}$	12.50	16	0.02	0	16	0.03	0
Barenco $\mathrm{Toff_5}$[42]	7	56	28	T-par	7	24	0.45	14.29	26	0.88	7.14	27	0.56	3.57
Barenco $\mathrm{Toff_6}$[42]	9	84	40	T-par	11	34	1.94	15	42	12.6	−5	42	2.59	−5
Barenco $\mathrm{Toff_{10}}$[42]	19	224	100	T-par	31	84	460.33	16	120	1938.01	−20	122	1269.03	−22
$\mathrm{VBE\text{-}Adder_3}$[42]	10	70	24	T-par	4	20	0.15	16.67	24	1639.76	0	38	1.93	−58.33
Mean								**19.76**			−31.59			−14.13
Standard error								2.12			8.87			4.69
Min								0			−216.09			−109.86
Max								**51.43**			15.96			45.71
Positive saving (%)							**96.88**			**18.18**			**30.30**	

[1] A. Y. Kitaev, A. Shen, and M. N. Vyalyi, *Classical and quantum computation*, Vol. 47 (American Mathematical Society Providence, 2002).
[2] C. M. Dawson and M. A. Nielsen, arXiv:quant-ph/0505030 (2005).
[3] A. G. Fowler, Quantum Information & Computation **11**, 867 (2011).
[4] V. Kliuchnikov, D. Maslov, and M. Mosca, Physical review letters **110**, 190502 (2013).
[5] P. Selinger, Physical Review A **87**, 042302 (2013).
[6] D. Gosset, V. Kliuchnikov, M. Mosca, and V. Russo, Quantum Information & Computation **14**, 1261 (2014).

[7] N. J. Ross and P. Selinger, Quant. Inf. and Comp. **16**, 901 (2016).
[8] E. T. Campbell, B. M. Terhal, and C. Vuillot, arXiv:1612.07330 (2016).
[9] S. Bravyi and A. Kitaev, Phys. Rev. A **71**, 022316 (2005).
[10] R. Raussendorf, J. Harrington, and K. Goyal, New Journal of Physics **9**, 199 (2007), arXiv:quant-ph/0703143.
[11] A. G. Fowler, M. Mariantoni, J. M. Martinis, and A. N. Cleland, Phys. Rev. A **86**, 032324 (2012).
[12] J. O'Gorman and E. T. Campbell, Physical Review A **95**, 032338 (2017).
[13] A. Paetznick and K. M. Svore, Quantum Information & Computation **14**, 1277 (2014).
[14] A. Bocharov, M. Roetteler, and K. M. Svore, Physical Review A **91**, 052317 (2015).
[15] A. Bocharov, M. Roetteler, and K. M. Svore, Phys. Rev. Lett. **114**, 080502 (2015).
[16] M. Howard and E. Campbell, Phys. Rev. Lett. **118**, 090501 (2017).
[17] E. Campbell, Physical Review A **95**, 042306 (2017).
[18] M. B. Hastings, arXiv:1612.01011 (2016).
[19] M. Amy, D. Maslov, M. Mosca, and M. Roetteler, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems **32**, 818 (2013).
[20] M. Amy, D. Maslov, and M. Mosca, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems **33**, 1476 (2014).
[21] Y. Nam, N. J. Ross, Y. Su, A. M. Childs, and D. Maslov, arXiv:1710.07345 (2017).
[22] M. Amy and M. Mosca, arXiv:1601.07363 (2016).
[23] G. Seroussi and A. Lempel, SIAM Journal on Computing **9**, 758 (1980).
[24] E. T. Campbell and M. Howard, Phys. Rev. A **95** (2017).
[25] A. Lempel, SIAM J. Comput. **4** (1975).
[26] M. J. Bremner, R. Jozsa, and D. J. Shepherd, Proceedings of the Royal Society of London A: Mathematical, Physical and Engineering Sciences (2010), 10.1098/rspa.2010.0301, http://rspa.royalsocietypublishing.org/content/early/2010/08/05/rspa.2010.0301.full.pdf.
[27] A. Montanaro, Journal of Physics A: Mathematical and Theoretical **50**, 084002 (2017).
[28] M. Amy, D. Maslov, M. Mosca, and M. Roetteler, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems **32**, 818 (2013).
[29] J. Håstad, Journal of Algorithms **11**, 644 (1990).
[30] D. Maslov, "Reversible logic synthesis benchmarks page," `http://webhome.cs.uvic.ca/~dmaslov/`, 2011.
[31] Information on the Iceberg HPC Cluster can be found here: , `https://www.sheffield.ac.uk/wrgrid/iceberg`.
[32] Y. Nam, N. J. Ross, Y. Su, A. M. Childs, and D. Maslov, "GitHub for reference [21]," `https://github.com/njross/optimizer`.
[33] M. A. Nielsen and I. L. Chuang, *Quantum Computation and Quantum Information* (Cambridge, 2000).
[34] A. W. Harrow and A. Montanaro, Nature **549**, 203 (2017).
[35] D. Shepherd and M. J. Bremner, Proceedings of the Royal Society of London A: Mathematical, Physical and Engineering Sciences **465**, 1413 (2009), http://rspa.royalsocietypublishing.org/content/465/2105/1413.full.pdf.
[36] R. Raussendorf, J. Harrington, and K. Goyal, New Journal of Physics **9**, 199 (2007).
[37] A. G. Fowler, S. J. Devitt, and C. Jones, Scientific reports **3**, 1939 (2013).
[38] A. G. Fowler and S. J. Devitt, arXiv:1209.0510 (2012).
[39] D. Maslov, arXiv:1602.02627 (2016).
[40] S. Bravyi, G. Smith, and J. A. Smolin, Phys. Rev. X **6**, 021043 (2016).
[41] S. Bravyi and D. Gosset, Phys. Rev. Lett. **116**, 250501 (2016).
[42] M. Amy, "T-par GitHub," `https://github.com/meamy/t-par`.
[43] To be precise, gadgets are only need for internal Hadamards. The external Hadamards that appear at the beginning and end of the circuit do not need to be replaced with Hadamard gadgets.
[44] Source code available at `https://github.com/Luke-Heyfron/TOpt`.
[45] Other column pairs may be destroyed after the duplication transformation in addition to the a^{th} and b^{th} columns but only for the latter pair is destruction guaranteed.

Appendix A CLIFFORD + T BENCHMARKS FOR TODD-PART AND TODD-h_{cap}

TABLE II. T-counts of Clifford + T benchmark circuits for the TODD-part and TODD-h_{cap} variants of TOpt are shown. TODD-part uses Hadamard-bounded partitions rather than Hadamard gadgets and ancillas and TODD-h_{cap} sets a fixed cap, h_{cap}, on the number of Hadamard ancillas available to the compiler. Starting at $h_{\mathrm{cap}} = 1$, we iteratively incremented the value of h_{cap} by 1 until obtaining the first result with a positive T-count saving over the best previous algorithm. The value of h_{cap} for which this occured is reported in the h_{cap} column, and the number of partitions, T-count, execution time and percentage saving for this result are detailed by column group TODD-h_{cap}. TODD-h_{cap} results that yield a positive saving for $h_{\mathrm{cap}} = 0$ correspond to results for TODD-part and results that require $h_{\mathrm{cap}} = n_h$ Hadamard ancillas correspond to results for TODD. As we are strictly interested in intermediate values of h_{cap}, we omit these data and refer the reader to the appropriate result. The number of Hadamard partitions is given by the $\mathbf{N_p}$ columns. As in Table I, **n** is the number of qubits for the input circuit; **T** are T-counts: for the circuit before optimization (Pre-Opt.); due to the best previous algorithm (Best prev.); and post-optimization using variants of our compiler. The best previous algorithm is given in the **Alg.** column where: T-par is from [20]; $\mathrm{RM_m}$ and $\mathrm{RM_r}$ are the majority and recursive Reed-Muller optimizers, respectively, both from [22]; and $\mathrm{Auto_H}$ is the heavy version of the algorithm from [21]. We show the T-count saving for each TOpt variant over the best previous algorithm in the **s** columns and the execution time as run on the Iceberg HPC cluster in the **t** columns. Results where the execution time is marked with † were obtained using an alternative implementation of TODD that is faster but less stable. **Positive saving** shows the proportion of the benchmark circuits, as a percentage, for which the corresponding compiler yields a positive saving over the best previous result.

	Pre-Opt.		**Best prev.**		**TODD-part**				**TODD-h_{cap}**				
Circuit	**n**	**T**	**T**	**Alg.**	$\mathbf{N_p}$	**T**	**t (s)**	**s(%)**	h_{cap}	$\mathbf{N_p}$	**T**	**t (s)**	**s(%)**
Mod 5_4[42]	5	28	16	T-par	7	18	$< 10^{-2}$	−12.50	1	4	16	$< 10^{-2}$	0
8-bit adder[42]	24	399	213	$\mathrm{RM_m}$	20	283	12.63	−32.86	13	5	212	227.81	0.47
CSLA-MUX_3[32]	16	70	58	$\mathrm{RM_r}$	7	62	0.38	−6.90	5	3	54	3.73	6.90
CSUM-MUX_9[32]	30	196	76	$\mathrm{RM_r}$	3	76	20.31	0	4	2	74	36.57	2.63
Cycle 17_3[42]	35	4739	1944	$\mathrm{RM_m}$	573	2625	1001.11	−35.03	43	15	1939	25507.5†	0.26
GF(2^4)-mult[42]	12	112	68	T-par	3	56	0.55	17.65	*0*	*See result for TODD-part*			
GF(2^5)-mult[42]	15	175	101	$\mathrm{RM_r}$	3	90	6.96	10.89	*0*	*See result for TODD-part*			
GF(2^6)-mult[42]	18	252	144	$\mathrm{RM_r}$	3	132	121.16	8.33	*0*	*See result for TODD-part*			
GF(2^7)-mult[42]	21	343	208	$\mathrm{RM_r}$	3	185	153.75	11.06	*0*	*See result for TODD-part*			
GF(2^8)-mult[42]	24	448	237	$\mathrm{RM_r}$	3	216	517.63	8.86	*0*	*See result for TODD-part*			
GF(2^9)-mult[42]	27	567	301	$\mathrm{RM_r}$	3	301	2840.56	0	8	2	295	3212.53	1.99
GF(2^{10})-mult[42]	30	700	410	T-par	3	351	23969.1	14.39	*0*	*See result for TODD-part*			
GF(2^{16})-mult[42]	48	1792	1040	T-par	3	922	76312.5†	11.35			-		
Grover_5[42]	9	52	52	T-par	18	52	0.02	0	5	4	50	0.3	3.85
$\mathrm{Hamming}_{15}$ (low)[42]	17	161	97	T-par	22	113	0.53	−16.49	5	6	93	2.93	4.12
$\mathrm{Hamming}_{15}$ (med)[42]	17	574	230	T-par	59	322	1.57	−40	11	7	226	58.08	1.74
$\mathrm{Hamming}_{15}$ (high)[42]	20	2457	1019	T-par	256	1505	16.84	−47.69	13	24	1010	595.8	0.88
HWB_6[30]	7	105	71	T-par	15	82	0.01	−15.49	3	6	68	0.13	4.23
HWB_8[30]	12	5887	3531	$\mathrm{RM_{m\&r}}$	709	4187	6.53	−18.58	9	110	3517	259.14	0.40
$\mathrm{Mod\text{-}Adder}_{1024}$ [42]	28	1995	1011	T-par	234	1165	98.8	−15.23	10	27	978	665.5	3.26
$\mathrm{Mod\text{-}Adder}_{1048576}$[42]	0	0	7298	T-par	2030	9480	89486.5†	−29.90			-		
$\mathrm{Mod\text{-}Mult}_{55}$[42]	9	49	35	$\mathrm{RM_{m\&r}}$	6	28	0.02	20	*0*	*See result for TODD-part*			
$\mathrm{Mod\text{-}Red}_{21}$[42]	11	119	73	T-par	15	85	0.06	−16.44	4	5	69	0.59	5.48
$\mathrm{n^{th}\text{-}prime}_6$[30]	6	567	400	$\mathrm{RM_{m\&r}}$	63	402	0.17	−0.50	2	29	384	0.98	4
$\mathrm{n^{th}\text{-}prime}_8$[30]	12	6671	4045	$\mathrm{RM_{m\&r}}$	774	5034	8.4	−24.45	12	105	4043	898.98	0.05
$\mathrm{QCLA\text{-}Adder}_{10}$[42]	36	238	162	T-par	6	184	223.25	−13.58	5	3	157	366.1	3.09
$\mathrm{QCLA\text{-}Com}_7$[42]	24	203	94	$\mathrm{RM_m}$	7	135	11.62	−43.62	16	2	81	170.77	13.83
$\mathrm{QCLA\text{-}Mod}_7$[42]	26	413	235	$\mathrm{Auto_H}$	15	305	34.76	−29.79	23	3	221	289.77†	5.96
QFT_4[42]	5	69	67	T-par	38	67	$< 10^{-2}$	0	2	13	63	0.02	5.97
$\mathrm{RC\text{-}Adder}_6$[42]	14	77	47	$\mathrm{RM_{m\&r}}$	13	59	0.11	−25.53	6	3	45	0.97	4.26
NC Toff_3[42]	5	21	15	T-par	3	15	$< 10^{-2}$	0	*2*	$= n_h$	*See result for TODD*		
NC Toff_4[42]	7	35	23	T-par	5	23	$< 10^{-2}$	0	*4*	$= n_h$	*See result for TODD*		
NC Toff_5[42]	9	49	31	T-par	7	31	0.01	0	5	2	29	0.2	6.45
NC Toff_{10}[42]	19	119	71	T-par	17	71	0.74	0	10	3	69	12.48	2.82
Barenco Toff_3[42]	5	28	16	T-par	4	22	$< 10^{-2}$	−37.50	2	2	14	$< 10^{-2}$	12.50
Barenco Toff_4[42]	7	56	28	T-par	8	38	0.01	−35.71	4	2	26	0.06	7.14
Barenco Toff_5[42]	9	84	40	T-par	12	54	0.03	−35	6	2	38	0.35	5
Barenco Toff_{10}[42]	19	224	100	T-par	32	134	2.27	−34	16	2	98	54.75	2
VBE-Adder_3[42]	10	70	24	T-par	5	36	0.04	−50	*4*	$= n_h$	*See result for TODD*		
Mean								−13.19	**9**				4.05
Standard error								3.15	1.65				0.64
Min								−50	1				0
Max								20	43				13.83
Positive saving (%)						**20.51**				**96.30**			

In order to investigate the relative effectiveness of the Hadamard gadget and Hadamard-bounded partition methods for dealing with Hadamard gates, we repeated the benchmarks from Table I but for the latter method. The results are shown in the TODD-part column group of Table II. For the Hadamard partition method, we found that the compiler runtime is significantly decreased, making the optimization of larger quantum circuits feasible. However, the performance is worse in terms of raw T count reductions, often leading to higher T counts than the best previous result. It is important to note that for a given input circuit, the T count is highly sensitive on the choice of Hadamard partitioning, of which, in general, there are many. Our implementation does not optimize over Hadamard partitioning choices, so there is potential for developing a more powerful version of TODD-part that makes use of an advanced Hadamard partitioning algorithm, which may lead to greater T count reductions.

The TODD compiler completely gadgetizes each Hadamard gate, whereas the TODD-part compiler completely partitions the circuit into Hadamard-bounded partitions. It is possible to interpolate between these two approaches using a parameter h_{cap} that enforces a cap on the number of available Hadamard ancillas. Upon reaching this cap, the compiler synthesises the circuit encountered so far, freeing up the Hadamard ancillas for the subsequent Hadamard partition. We have implemented this feature, and in order to quantify the overhead required to see a T count reduction, we ran each benchmark repeatedly, incrementing the value of h_{cap} until we saw a reduction over the best previous result. The results for this experiment are presented in Table II. We found that the relationship between h_{cap} and T count savings is favourable: relatively few Hadamard gadgets are required to see a reduction over the best previous result. Over all the benchmark circuits, where the number of qubits and the T count ranges up to $n = 36$ and $T = 6671$, respectively, we found that on average 9 Hadamard ancillas are required to see positive saving and at most 23 ancillas are needed for all but one exceptional result (Cycle 17_3), which requires 43. This suggests that, while TODD combined with full Hadamard gadgetization is clearly the forerunner amongst our compilers for reducing the T count, a modest improvement in the Hadamard partitioning scheme, or adding a pre-processing step that looks for Hadamard gate reductions may lead to a better version of TODD that requires no non-unitary gadgets, has feasible compiler runtimes for large circuits, and yields positive T count savings.

Appendix B LEMPEL'S FACTORING ALGORITHM

We describe Lempel's factoring algorithm (originally from reference [25]) using conventions consistent with our description of the TODD algorithm to more easily see how TODD generalizes Lempel's algorithm for order 3 tensors. Lempel's factoring algorithm takes as input a symmetric tensor of order 2 (a matrix), which we denote $S \in \mathbb{Z}_2^{(n,n)}$ and outputs a matrix $A \in \mathbb{Z}_2^{(n,m)}$ where the elements of A and S are related as follows:

$$S_{\alpha,\beta} = \sum_{k=1}^{m} A_{\alpha,k} A_{\beta,k} \pmod 2. \tag{32}$$

Lempel proved that the minimal value of m is equal to

$$\mu(S) = \rho(S) + \delta(S), \tag{33}$$

where $\rho(S)$ is the rank of matrix S and

$$\delta(S) = \begin{cases} 1 & \text{if } S_{\alpha,\alpha} = 0 \ \forall \ \alpha \in [1,n] \\ 0 & \text{otherwise} \end{cases}. \tag{34}$$

Lempel's algorithm solves the problem of finding an A matrix that obeys equation (32) for a given S matrix such that $m = \mu(S)$. Such an A matrix is referred to as a minimal factor of S.

In the following, we denote the number of columns of A as $c(A)$ and the j^{th} column of A as $\mathbf{c}_j(A)$. Lempel's algorithm is the following:

1. Generate an initial (necessarily suboptimal) A matrix for S.
2. Check if $c(A) = \mu(S)$. If true, exit and output A. Otherwise, perform steps 3 to 7.
3. Find a $\mathbf{y} \in \mathbb{Z}_2^m$ such that $A\mathbf{y} = \mathbf{0}$ and $0 < |y| < c(A)$.
4. If $|y| = 1 \pmod 2$ then update $\mathbf{y} \to (\mathbf{y}^T, 1)^T$ and $A = (A \quad \mathbf{0})$.
5. Find a pair of indices $a, b \in [1, m],\ a \neq b$ such that $y_a \oplus y_b = 1$.
6. Apply transformation $A \to A \oplus \mathbf{z}\mathbf{y}^T$, where $\mathbf{z} = \mathbf{c}_a(A) \oplus \mathbf{c}_b(A)$.
7. Remove the a^{th} and b^{th} columns from A, then go to step 2.

Note that the key difference between the Lempel and TODD algorithm is that TODD additionally requires condition *C3* from lemma III.3 to be satisfied.

Appendix C TOOL ALGORITHM

Here we give a detailed description of TOOL, with the main idea illustrated by Fig. 3. TOOL is best explained in terms of weighted polynomials (recall equation (6)). The algorithm is iterative, where each round consists of the five steps detailed below. Before the first round, we initialize an 'empty' output gate synthesis matrix, $A_{\text{out}} \in \mathbb{Z}_2^{(n,0)}$.

1. Choose an integer $c \in [1, n]$ such that there is at least one term in f with x_c as a factor. If no such c exists, the algorithm terminates and outputs A_{out}.
2. Find $\tilde{f}_c$, the *target polynomial* of f with respect to x_c (see equation 35 below).
3. Determine the order 2 signature tensor, $\tilde{S}$, of $\tilde{f}_c$.
4. Find $\tilde{A}$, a minimal factor of $\tilde{S}$, using Lempel's factoring algorithm.
5. Recover an order 3 gate synthesis matrix, A, for $\tilde{A}$, and append it to A_{out}. Replace f with $f - |A^T\mathbf{x}|$.

Each round of TOOL gives a new f that depends on fewer x variables. When f depends on only n_{RM} or fewer variables, we switch to the optimal brute force optimizer, RM.

We will now explain each step of the above description in detail, unpacking the contained definitions. In step 1, we select an index c, which corresponds to the control qubit of the control-$U_{2\tilde{f}_c}$ operator shown in Fig. 3. The order that we choose c for each round can affect the output and therefore is a parameter of TOOL. For all results, we randomly selected c with uniform probability from the set of all indices $\{c\}$ for which x_c is a factor of at least one term in f.

Next, we observe that any f can be decomposed into $f = f_c + f_c'$, where we define f_c as a weighted polynomial containing all terms of f with x_c as a factor. The former part, f_c, can be further decomposed as follows,

$$f_c = 2x_c\tilde{f}_c + l_c x_c \tag{35}$$

where $\tilde{f}_c$ is quadratic and so can be optimally synthesized efficiently. In step 2, we extract $\tilde{f}_c$, which is implicitly fixed by the above equations. We refer to $\tilde{f}_c$ as a *target polynomial* because it corresponds to the target of a control-U_{2f} operator, where $f = \tilde{f}_c$ and $|x_c\rangle$ is the control qubit.

As an aside, we remark that the target polynomial is related to Shannon cofactors that appear in Boole's expansion theorem. Specifically, we have

$$\tilde{f}_c = \frac{f_c^+ - f_c^- - l_c}{2}, \tag{36}$$

where f_c^+ and f_c^- are the positive and negative Shannon cofactors, respectively, of f with respect to x_c, and l_c is the linear coefficient of f associated with x_c.

In step 3, we map $\tilde{f}_c$ to a signature tensor of order 2 (a matrix) for use with Lempel's factoring algorithm. Let $\tilde{l}_\alpha, \tilde{q}_{\alpha,\beta}$ be the linear and quadratic coefficients of $\tilde{f}_c$, respectively. For each $\alpha, \beta \neq c$, the elements of $\tilde{S}$ are obtained as follows.

$$\tilde{S}_{\alpha,\beta} = \begin{cases} \tilde{l}_\alpha \pmod 2 & \text{if } \alpha = \beta \\ \tilde{q}_{\alpha,\beta} \pmod 2 & \text{if } \alpha \neq \beta \end{cases}. \tag{37}$$

Finding a minimal factor of $\tilde{S}_{\alpha,\beta}$ is the problem 2-STR. Therefore, we can use Lempel's algorithm (see appendix B) to find a matrix $\tilde{A} \in \mathbb{Z}_2^{(n,\tilde{m})}$, which is a minimal factor of $\tilde{S}$ such that

$$\tilde{f}_c = |\tilde{A}^T\mathbf{x}| = \sum_{j=1}^{\tilde{m}} \left[\bigoplus_{i=1}^{n} \tilde{A}_{i,j} x_i \right] \pmod 8. \tag{38}$$

By substituting equation (38) into equation (35) we obtain

$$f_c = 2x_c|\tilde{A}^T\mathbf{x}| + l_c x_c, \tag{39}$$

$$= \sum_{j=1}^{\tilde{m}} 2x_c \left[\bigoplus_{i=1}^{n} \tilde{A}_{i,j} x_i \right] + l_c x_c \pmod 8, \tag{40}$$

where we have taken the factor $2x_c$ within the Hamming weight summation. Next, we use the modular identity $2ab = a + b - a \oplus b$ with $a = x_c$ and b as the contents of the square brackets. This gives

$$f_c = \sum_{j=1}^{\tilde{m}} \left(x_c + \left[\bigoplus_{i=1}^{n} \tilde{A}_{i,j} x_i \right] - x_c \oplus \left[\bigoplus_{i=1}^{n} \tilde{A}_{i,j} x_i \right] \right) + l_c x_c \pmod 8, \tag{41}$$

$$= x_c(\tilde{m} + l_c) + \sum_{j=1}^{\tilde{m}} \left[\bigoplus_{i=1}^{n} \tilde{A}_{i,j} x_i \right] - \sum_{j=1}^{\tilde{m}} x_c \oplus \left[\bigoplus_{i=1}^{n} \tilde{A}_{i,j} x_i \right] \pmod 8, \tag{42}$$

$$= x_c(\tilde{m} + l_c) + |\tilde{A}^T \mathbf{x}| - |(\tilde{A} \oplus B_c)^T \mathbf{x}| \pmod 8, \tag{43}$$

where $B_c \in \mathbb{Z}_2^{(n,m)}$ is a matrix with elements

$$[B_c]_{i,j} = \begin{cases} 1 & \text{if } i = c \\ 0 & \text{otherwise} \end{cases}. \tag{44}$$

This is now in the form of a phase polynomial (e.g. see equation (8)) with no more than $1 + 2\tilde{m}$ terms, where $\tilde{m}$ was the optimal size of the factorisation found using Lempel's algorithm.

There are two versions of TOOL: with and without feedback. The difference between these versions determines whether all of equation (43) is put into A_{out} or whether parts are 'fed back' into f for subsequent rounds. This leads to two distinct definitions of the A matrix referred to in step 5 of TOOL:

$$|A^T \mathbf{x}| = \begin{cases} (\tilde{m} + l_c)x_c - |(\tilde{A} \oplus B_c)^T \mathbf{x}| & \text{feedback} \\ (\tilde{m} + l_c)x_c - |(\tilde{A} \oplus B_c)^T \mathbf{x}| + |\tilde{A}^T \mathbf{x}| & \text{without feedback} \end{cases}. \tag{45}$$

Notice that both $(\tilde{m} + l_c)x_c$ and $|(\tilde{A} \oplus B_c)^T \mathbf{x}|$ depend on x_c, so must be sent to output. Furthermore, they comprise *all* the terms that depend on x_c, which is why sending $|\tilde{A}^T \mathbf{x}|$ to output is optional, and why the number of dependent variables is reduced by at least 1 each round. For the *feedback* version, $|\tilde{A}^T \mathbf{x}|$ is kept within f during step 5, whereas it is sent to output A_{out} in the *without feedback* version.

Appendix D CALCULATING CLIFFORD CORRECTION

We will now describe how to determine the Clifford correction required to restore the output of *T-Optimiser* to the input unitary. Let the input of *T-Optimiser* be a weighted polynomial f that implements unitary $U_f \in \mathcal{D}_3$, and let the output be a weighted polynomial g. Any f can be split into the sum

$$f = f_1 + f_2, \tag{46}$$

where the coefficients of f_1 are in $\mathbb{Z}_2$ and those of f_2 are even. From the definition of *T-Optimiser*, we know the coefficients of f and g have the same parity i.e.

$$g = g_1 + g_2 = f_1 + g_2, \tag{47}$$

where g_1, g_2 are similarly defined for g. Using equations (46) and (47) we find,

$$g = f + (g_2 - f_2). \tag{48}$$

Equation (48) implies that $U_{\text{Clifford}} = U_{(g_2 - f_2)} \in \mathcal{D}_2$. Therefore, the Clifford correction is $U^\dagger_{\text{Clifford}} = U^\dagger_{(g_2 - f_2)} = U_{(f_2 - g_2)}$. We can map $(f_2 - g_2)$ to a phase polynomial and subsequently to a quantum circuit, $\mathcal{U}^\dagger_{\text{Clifford}}$.

Appendix E TODD PSEUDOCODE

Algorithm 1 Third Order Duplicate-then-Destroy (TODD) Algorithm

Input: Gate synthesis matrix $A \in \mathbb{Z}_2^{(n,m)}$.

Output: Gate synthesis matrix $A' \in \mathbb{Z}_2^{(n,m')}$ such that $m' \leq m$ and $S^{(A')} = S^{(A)}$.

- Let $\mathrm{col}_j(A)$ be a function that returns the j^{th} column of A.
- Let $\mathrm{cols}(A)$ be a function that returns the number of columns of A.
- Let nullspace(A) be a function that returns a matrix whose columns generate the right null space of A.
- Let proper(A) be a function that returns matrix A with every pair of identical columns and every all-zero column removed.

procedure TODD
  Initialize $A' \leftarrow A$
start:
  for all $1 \leq a < b \leq \mathrm{cols}(A')$ **do**
    $\mathbf{z} \leftarrow \mathrm{col}_a(A') + \mathrm{col}_b(A')$
    $\tilde{A} \leftarrow \begin{pmatrix} A' \\ \chi(A', \mathbf{z}) \end{pmatrix}$
    $N \leftarrow \mathrm{nullspace}(\tilde{A})$
    for all $1 \leq k \leq \mathrm{cols}(N)$ **do**
      $\mathbf{y} \leftarrow \mathrm{col}_k(N)$
      if $y_a \oplus y_b = 1$ **then**
        if $|\mathbf{y}| = 1 \pmod 2$ **then**
          $A' \leftarrow \begin{pmatrix} A' & \mathbf{0} \end{pmatrix}$
          $\mathbf{y} \leftarrow \begin{pmatrix} \mathbf{y} \\ 1 \end{pmatrix}$
        $A' \leftarrow A' + \mathbf{z}\mathbf{y}^T$
        $A' \leftarrow \mathrm{proper(A')}$
        goto *start*

Appendix F COMPUTATIONAL EFFICIENCY OF TODD

In this appendix, we calculate an upper-bound on the worst-case computational efficiency of the TODD algorithm as described in appendix E, in terms of the number of arithmetic operations on $GF(2)$ required.

Let A be a gate synthesis matrix with n rows and m columns that is used as input for the TODD algorithm. The loop, L_1, over each column pair (a, b) requires at most $\binom{m}{2} = O(m^2)$ iterations to complete. Inside L_1, there are four lines of pseudocode: a column addition, requiring no more than n operations; a matrix concatenation and calculation of $\chi(A, \mathbf{z})$, requiring E_1 operations; a nullspace calculation, requiring $O(n^3) + O(m^2 n)$ operations using Gaussian elimination; and finally a nested loop L_2, requiring E_2 operations.

From equation (27), we see that each row of $\chi(A, \mathbf{z})$ can be calculated with $O(m)$ operations. There are a maximum of $\binom{n}{3}$ rows in $\chi(A, \mathbf{z})$ so the total number of operations required to calculate χ is $O(n^3 m)$. Combining this with the matrix concatenation, we find that $E_1 = O(n^3 m) + nm = O(n^3 m)$.

The loop L_2 executes in at most

$$\textsc{cols}(\textsc{nullspace}(\tilde{A})) := \textsc{colrank}(\textsc{nullspace}(\tilde{A})) = m - \textsc{rank}(\tilde{A}) \leq m - \textsc{rank}(A) \leq m - n \tag{49}$$

iterations. The identity between the column rank and the number of columns follows from the assertion that the nullspace function outputs a matrix whose columns are a linearly independent basis for the nullspace of A.

The loop L_2 is composed of a conditional that requires 1 addition (by merging the first line of L_2 and the conditional). The content of the conditional is only evaluated once, so can be considered as part of L_1 for this calculation. Therefore, the number of operations performed in L_2 is $E_2 = m - n$.

The nested conditional requires at most $m + n + 1$ operations, where the terms are due to the Hamming weight of $|\mathbf{y}|$, concatenating an all-zero column to A' and concatenating a one to $\mathbf{y}$, respectively. The line $A' \leftarrow A' + \mathbf{z}\mathbf{y}^T$ requires at most $n(m + 1)$ operations and the proper function can be computed using at most m operations by keeping track of all-zero columns with a Boolean array, for a small physical overhead of m.

The outermost loop (between *start* and **goto** *start*) by definition executes in no more than $m - m'$ iterations where m' is the number of columns of the output. In this worst-case calculation, we assume $m' = 0$.

So the TODD algorithm can be executed using

$$O(m\left[n + O(n^3m) + O(n^3) + O(m^2n) + (m-n) + (m+n+1) + n(m+1) + m\right]) \tag{50}$$
$$= O(m\left[O(n^3m) + O(n^3) + O(m^2n)\right]) \tag{51}$$
$$= O(n^3m^2) + O(nm^3) \tag{52}$$

operations.

Therefore, given a family of Clifford + T circuits with n qubits, h Hadamard gates and t T gates, we would expect our compiler to execute in time asymptotically upper-bounded by a function of the following form

$$O((n+h)^3t^2) + O((n+h)t^3) \tag{53}$$
$$= O(n^3t^2) + O(h^3t^2) + O(nt^3) + O(ht^3), \tag{54}$$

where we have made the reasonable assumption that the computational bottleneck is due to the TODD algorithm, rather than the circuit preprocessing stages or mapping between different circuit representations, for instance.

In practice, the actual runtimes for the benchmark quantum circuits seen in Table I are much lower than this worst-case upper-bound. Furthermore, the compiler runtime is dependent on the structure of the input quantum circuit, rather than simply the number of qubits and gates from which it is composed. Consequently, we do not see a simple relation between circuit parameters n, t, h and the runtime for the benchmarks in Table I.

Note that in our calculation of the complexity, we assumed that we must calculate *every* row of $\chi(A, \mathbf{z})$. In practice, we find that many of the rows are identical. An algorithm that calculates only the unique rows may lead to improved computational efficiency.