Automated optimization of large quantum circuits with continuous parameters

Yunseong Nam,[1,2,3] Neil J. Ross,[1,2,4] Yuan Su,[1,2,5]
Andrew M. Childs,[1,2,5] and Dmitri Maslov[1,2,6]

[1] Institute for Advanced Computer Studies, University of Maryland, College Park, MD 20742, USA

[2] Joint Center for Quantum Information and Computer Science, University of Maryland, College Park, MD 20742, USA

[3] IonQ Inc., College Park, MD 20740, USA

[4] Department of Mathematics and Statistics, Dalhousie University, Halifax, NS B3H 4R2, Canada

[5] Department of Computer Science, University of Maryland, College Park, MD 20742, USA

[6] National Science Foundation, Alexandria, VA 22314, USA

Abstract

We develop and implement automated methods for optimizing quantum circuits of the size and type expected in quantum computations that outperform classical computers. We show how to handle continuous gate parameters and report a collection of fast algorithms capable of optimizing large-scale quantum circuits. For the suite of benchmarks considered, we obtain substantial reductions in gate counts. In particular, we provide better optimization in significantly less time than previous approaches, while making minimal structural changes so as to preserve the basic layout of the underlying quantum algorithms. Our results help bridge the gap between the computations that can be run on existing hardware and those that are expected to outperform classical computers.

1 Introduction

Quantum computers have the potential to dramatically outperform classical computers at solving certain problems. Perhaps their best-known application is to the task of factoring integers: whereas the fastest known classical algorithm is superpolynomial [22], Shor's algorithm solves this problem in polynomial time [34], providing an attack on the widely-used RSA cryptosystem.

Even before the discovery of Shor's algorithm, quantum computers were proposed for simulating quantum mechanics [11]. By simulating Hamiltonian dynamics, quantum computers can study phenomena in condensed-matter and high-energy physics, quantum chemistry, and materials science. Useful instances of quantum simulation are likely accessible to smaller-scale quantum computers than classically-hard instances of the factoring problem.

These and other potential applications [19] have helped motivate significant efforts toward building a scalable quantum computer. Two quantum computing technologies, superconducting circuits [16] and trapped ions [8], have matured sufficiently to enable fully programmable universal devices, albeit currently of modest size. Several groups are actively developing these platforms into larger-scale devices, backed by significant investments from both industry [15, 17, 24, 35] and government [10, 12, 28]. Thus, it is plausible that quantum computations involving tens or even hundreds of qubits will be carried out in the not-too-distant future [14, 20].

Experimental quantum information processing remains a difficult technical challenge, and the resources available for quantum computation will likely continue to be expensive and severely limited for some time.

To make the most out of the available hardware, it is essential to develop implementations of quantum algorithms that are as efficient as possible.

Quantum algorithms are typically expressed in terms of *quantum circuits*, which describe a computation as a sequence of elementary quantum logic gates acting on qubits (see Section 2 for more details). There are many ways of implementing a given algorithm with an available set of elementary operations, and it is advantageous to find an implementation that uses the fewest resources. While it is imperative to develop algorithms that are efficient in an abstract sense and to implement them with an eye toward practical efficiency, large-scale quantum circuits are likely to have sufficient complexity to benefit from automated optimization.

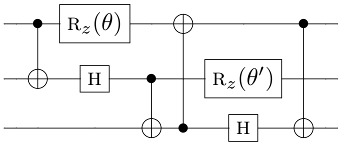
In this work, we develop software tools for reducing the size of quantum circuits, aiming to improve their performance as much as possible at a scale where manual gate-level optimization is no longer practical. Since global optimization of arbitrary quantum circuits is QMA-hard [18], our goal is more modest: we apply a set of carefully chosen heuristics to reduce the gate counts, often resulting in substantial savings.

We apply our optimization techniques to several types of quantum circuits. Our benchmark circuits include components of quantum algorithms for factoring and computing discrete logarithms, such as the quantum Fourier transform, integer adders, and Galois field multipliers. We also consider circuits for the product formula approach to Hamiltonian simulation [4, 23]. In all cases, we focus on circuit sizes likely to be useful in applications that outperform classical computation. Our techniques can help practitioners understand which implementation of an algorithm is most efficient in a given application. We detail our methods in Section 3 and discuss our results in Section 4, before concluding in Section 5.

While there has been considerable previous work on quantum circuit optimization (as detailed in Section 4.3), we are not aware of prior work on automated optimization that has targeted large-scale circuits such as the ones considered here. Moreover, extrapolation of previously-reported runtimes suggests it is unlikely that existing quantum circuit optimizers would perform well for such large circuits. We perform direct comparisons by running our software on the same circuits optimized in Ref. [1], showing that our approach typically finds smaller circuits in less time. In addition, to the best of our knowledge, our work is the first to focus on automated optimization of quantum circuits with continuous gate parameters.

2 Background

A *quantum circuit* is a sequence of quantum gates acting on a collection of qubits. Quantum circuits are conveniently represented by diagrams in which horizontal wires denote time evolution of qubits, with time propagating from left to right, and boxes (or other symbols joining the wires) represent quantum gates. For example, the diagram



(1)

describes a simple three-qubit quantum circuit.

We consider a simple set of elementary gates for quantum circuits consisting of the two-qubit controlled-NOT gate (abbreviated CNOT, the leftmost gate in the above circuit), together with the single-qubit NOT gate, Hadamard gate H, and z-rotation gate $\mathrm{R}_z(\theta)$. Unitary matrices for these gates take the form

$$\text{NOT} := \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}, \quad \text{H} := \frac{1}{\sqrt{2}}\begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}, \quad \mathrm{R}_z(\theta) := \begin{pmatrix} e^{-i\theta/2} & 0 \\ 0 & e^{i\theta/2} \end{pmatrix}, \quad \text{and} \quad \text{CNOT} := \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix}, \tag{2}$$

where $\theta \in (0, 2\pi]$ is the rotation angle. The gates $\text{S} := \mathrm{R}_z(\pi/2)$ and $\text{T} := \mathrm{R}_z(\pi/4)$ are known as the *Phase* and T gates, respectively. When the rotation angle is irrelevant, we denote a generic z-rotation by R_z.

While we aim to produce quantum circuits over the set of NOT, H, R_z, and CNOT gates, we consider input circuits that may also include Toffoli gates. The Toffoli gate (the top gate in Figure 5) is described by the mapping $|x, y, z\rangle \mapsto |x, y, z \oplus (x \wedge y)\rangle$ of computational basis states. We also allow Toffoli gates to have negated controls. For example, the Toffoli gate with its top control negated (the middle gate in Figure 5) acts as $|x, y, z\rangle \mapsto |x, y, z \oplus (\bar{x} \wedge y)\rangle$, and the Toffoli gate with both controls negated (the bottom gate in Figure 5) acts as $|x, y, z\rangle \mapsto |x, y, z \oplus (\bar{x} \wedge \bar{y})\rangle$.

The cost of performing a given quantum circuit depends on the physical system used to implement it. The cost can also vary significantly between a physical-level (unprotected) implementation and a logical-level (fault-tolerant) implementation. At the physical level, a two-qubit gate is typically more expensive to implement than a single-qubit gate [8, 16]. We accommodate this by considering the CNOT gate count and optimizing the number of the CNOT gates in our algorithms.

For logical-level fault-tolerant circuits, the so-called Clifford operations (generated by the Hadamard, Phase, and CNOT gates) are often relatively easy to implement, whereas non-Clifford operations incur significant overhead [5, 30]. Thus we also consider the number of R_z gates in our algorithms and try to optimize their count. In fault-tolerant implementations, R_z gates are approximated over a discrete gate set, typically consisting of Clifford and T gates. Optimal algorithms for producing such approximations are known [21, 32]. The number of Clifford+T gates required to approximate a generic R_z gate depends primarily on the desired accuracy rather than the specific angle of rotation, so it is preferable to optimize a circuit before approximating its R_z gates with Clifford+T fault-tolerant circuits.

By minimizing both the CNOT and R_z counts, we perform optimizations targeting both physical- and logical-level implementations. One might expect a trade-off between these two goals, and in fact we know of instances where such trade-offs do occur. However, in this paper we only consider optimizations aimed at reducing both the R_z and CNOT counts.

3 Algorithms and implementation

In this section, we describe our optimization algorithms and their implementation. Throughout, we use g to denote the number of gates appearing in a circuit. We begin in Section 3.1 by describing three distinct representations of quantum circuits that we employ. In Section 3.2, we describe a preprocessing step used in all versions of our algorithm. Then, in Section 3.3, we describe several subroutines that form the basic building blocks of our approach. Section 3.4 explains how these subroutines are combined to form our main algorithms. Finally, in Section 3.5, we present two special-purpose optimization techniques that we use to handle particular types of circuits.

3.1 Representations of quantum circuits

We use the following three representations of quantum circuits:

- First, we store a circuit as a list of gates to be applied sequentially (a *netlist*). It is sometimes convenient to specify the circuit in terms of subroutines, which we call *blocks*. Each block can be iterated any number of times and applied to any subset of the qubits present in the circuit. A representation using blocks can be especially concise since many quantum circuits exhibit a significant amount of repetition. A block is specified as a list of gates and qubit addresses.

  We input and output the netlists using the .qc format of [1] and the format produced by the quantum programming language Quipper [13]. Both include the ability to handle blocks.

- Second, we use a *directed acyclic graph* (DAG) representation. The vertices of the DAG are the gates of the circuit and the edges encode their input/output relationships. The DAG representation has the advantage of making adjacency between gates easy to access.

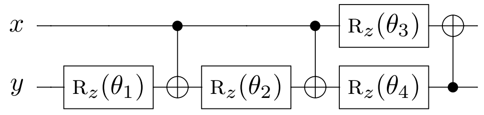
- Third, we use a generalization of the *phase polynomial* representation of {CNOT,T} circuits [2]. Unlike the netlist and DAG representations, this last representation applies only to circuits consisting entirely of NOT, CNOT, and R_z gates. Such circuits can be concisely expressed as the composition of an affine reversible transformation and a diagonal phase transformation. Let C be a circuit consisting only of NOT gates, CNOT gates, and the gates $\mathrm{R}_z(\theta_1), \mathrm{R}_z(\theta_2), \ldots, \mathrm{R}_z(\theta_\ell)$. Then the action of C on the n-qubit basis state $|x_1, x_2, \ldots, x_n\rangle$ has the form

$$|x_1, x_2, \ldots, x_n\rangle \mapsto e^{ip(x_1,x_2,\ldots,x_n)}|h(x_1, x_2, \ldots, x_n)\rangle, \tag{3}$$

where $h\colon \{0,1\}^n \to \{0,1\}^n$ is an affine reversible function and

$$p(x_1, x_2, \ldots, x_n) = \sum_{i=1}^{\ell}(\theta_i \bmod 2\pi) \cdot f_i(x_1, x_2, \ldots, x_n) \tag{4}$$

is a linear combination of affine Boolean functions $f_i\colon \{0,1\}^n \to \{0,1\}$ with the coefficients reduced modulo 2π. We call $p(x_1, x_2, \ldots, x_n)$ the *phase polynomial* associated with the circuit C. For example, the circuit



(5)

can be represented by the mapping

$$|x, y\rangle \mapsto e^{ip(x,y)}|x \oplus y, y\rangle \tag{6}$$

where $p(x, y) = \theta_1 y + \theta_2(x \oplus y) + \theta_3 x + \theta_4 y$. (In Ref. [2], the phase polynomial representation is only considered for $\{\mathrm{CNOT}, \mathrm{T}\}$ circuits, so all θ_i in the expression (4) are integer multiples of $\pi/4$ and the functions f_i are linear.)

We can convert between any two of the above three circuit representations in time linear in the number of gates in the circuit. Given a netlist, we can build the corresponding DAG gate-by-gate. Conversely, we can convert a DAG to a netlist by standard topological sorting. To convert between the netlist and phase polynomial representations of $\{\mathrm{NOT}, \mathrm{CNOT}, \mathrm{R}_z\}$ circuits, we use a straightforward generalization of the algorithm of [2].

3.2 Preprocessing

Before running our main optimization procedures, we preprocess the circuit to make it more amenable to further optimization. Specifically, the preprocessing applies provided the input circuit consists only of NOT, CNOT, and Toffoli gates (as is the case for the Quipper adders described in Section 4.1 and the T-par circuit benchmarks described in Section 4.3). In this case, we push the NOT gates as far to the right as possible by commuting them through the controls of Toffoli gates and the targets of Toffoli and CNOT gates. When pushing a NOT gate through a Toffoli gate control, we negate that control (or remove the negation if it was initially negated). If this procedure leads to a pair of adjacent NOT gates, we remove them from the circuit. If no such cancellation is found, we revert the control negation changes and move the NOT gate back to its original position.

This NOT gate propagation leverages two aspects of our optimizer. First, we accept Toffoli gates that may have negated controls and optimize their decomposition into Clifford+T circuits by exploiting freedom in the choice of $\mathrm{T}/\mathrm{T}^\dagger$ polarities (see Section 3.5). Second, since cancellations of NOT gates simplify the phase polynomial representation (by making some of the functions f_i in the phase polynomial representation (4) linear instead of merely affine), such cancellations make it more likely that Routine 4 and Routine 5 in Section 3.3 will find optimizations (since those routines rely on finding matching terms in the phase polynomial representation).

The complexity of this preprocessing step is $O(g)$ since we simply make a single pass through the circuit.

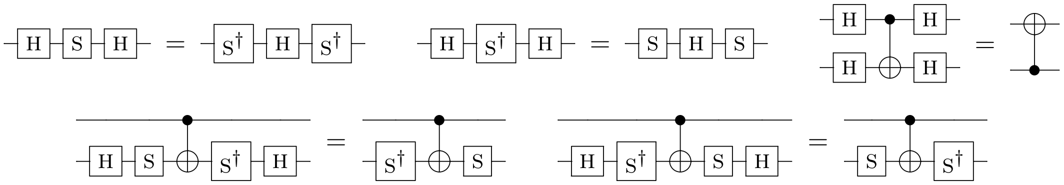


Figure 1: Hadamard gate reductions. The two rules illustrated on the bottom can be applied even if the middle CNOT gate is replaced by a circuit with any number of CNOT gates, provided they all share the target of the original CNOT.

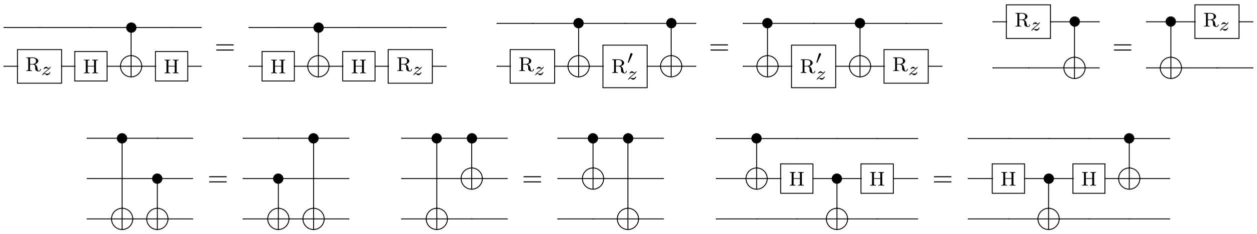


Figure 2: Commutation rules. Top: Commuting an R$_z$ gate to the right. Bottom: Commuting a CNOT gate to the right.

3.3 Optimization subroutines

Our optimization algorithms rely on a variety of subroutines that we now describe. For each of them, we report the worst-case time complexity as a function of the number of gates g in the circuit (for simplicity, we neglect the dependence on the number of qubits and other parameters). We optimize practical performance by carefully ordering and restricting the subroutines, as we discuss further below.

1. *Hadamard gate reduction*

   Hadamard gates do not participate in phase polynomial optimization (Routine 4 and Routine 5 below) and also tend to hinder gate commutation. Thus, we use the circuit identities pictured in Figure 1 to reduce the Hadamard gate count. Each application of these rules reduces the H count by up to 4. For a given Hadamard gate, we can use the DAG representation to check in constant time whether it is involved in one of these circuit identities. Thus, we can implement this subroutine with complexity $O(g)$ by making a single pass through all Hadamard gates in the circuit.

2. *Single-qubit gate cancellation*

   Using the DAG representation of a quantum circuit, it is straightforward to determine whether a gate and its inverse are adjacent. If so, both gates can be removed to reduce the gate count. More generally, we can cancel two single-qubit gates U and $U^\dagger$ that are separated by a subcircuit A that commutes with U. In general, deciding whether a gate U commutes with a circuit A may be computationally demanding. Instead, we apply a specific set of rules that provide sufficient (but not necessary) conditions for commutation. This approach is fast and appears to discover many commutations that can be exploited to simplify quantum circuits.

   Specifically, for each gate U in the circuit, the optimizer searches for possible cancellations with some instance of $U^\dagger$. To do this, we repeatedly check whether U commutes through a set of consecutive gates, as evidenced by one of the patterns in Figure 2. If at some stage we cannot move U to the right by some allowed commutation pattern, then we fail to cancel U with a matched $U^\dagger$, so we restore the initial configuration. Otherwise, we successfully cancel U with some instance of $U^\dagger$.

   For each of the g gates U, we check whether it commutes through $O(g)$ subsequent positions. Thus the complexity of the overall gate cancellation rule is $O(g^2)$. We could make the complexity linear in

g by only considering commutations through a constant number of subsequent gates, but we do not find this to be necessary in practice.

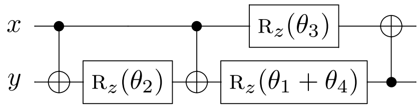
We also use a slight variation of this subroutine to merge rotation gates, rather than cancel inverses. Specifically, two rotations $\mathrm{R}_z(\theta_1)$ and $\mathrm{R}_z(\theta_2)$ can be combined into a single rotation $\mathrm{R}_z(\theta_1 + \theta_2)$ to eliminate one R_z gate.

3. *Two-qubit gate cancellation*

   This routine is analogous to Routine 2, except that U is a two-qubit gate, which is always CNOT in the circuits we consider. Again its complexity is $O(g^2)$, but may be reduced to $O(g)$ by imposing a maximal size for the subcircuit A.

4. *Rotation merging using phase polynomials*

   Consider a subcircuit consisting of NOT, CNOT, and R_z gates. Observe that if two individual terms of its phase polynomial expression satisfy $f_i(x_1, x_2, \ldots, x_n) = f_j(x_1, x_2, \ldots, x_n)$ for some $i \neq j$, then the corresponding rotations $\mathrm{R}_z(\theta_i)$ and $\mathrm{R}_z(\theta_j)$ can be merged. For example, in the circuit (5), the first and fourth rotations are both applied to the qubit carrying the value y, as evidenced by its phase polynomial representation. Thus (5) is equivalent to the circuit

   (7)

   in which the two rotations are combined. In other words, the phase polynomial representation of circuits reveals when two rotations—in this case, $\mathrm{R}_z(\theta_1)$ and $\mathrm{R}_z(\theta_4)$—are applied to the same affine function of the inputs, even if they appear in different parts of the circuit. Then we may combine these rotations into a single rotation, improving the circuit.[1] We have the flexibility to place the combined rotation at any point in the circuit where the relevant affine function appears. For concreteness, we place it at the first (leftmost) such location.

We next discuss some implementation details for Routine 4. To apply this routine, we must identify a subcircuit consisting only of $\{\text{NOT}, \text{CNOT}, \mathrm{R}_z\}$ gates. We build this subcircuit one qubit at a time, starting from a designated CNOT gate. For the first qubit of this gate, we scan through all preceding and subsequent NOT, CNOT, and R_z gates that act on this qubit, adding them to the subcircuit. When we encounter a Hadamard gate or the beginning or end of the circuit, we mark a *termination point* and stop exploring in that direction (so that each qubit has one beginning termination point and one ending termination point). For each CNOT gate between this qubit and some qubit that has not yet been encountered, we mark an *anchor point* where the gate acts on the newly-encountered qubit. We then carry out this process with the second qubit acted on by the initial CNOT gate, and repeat the process starting from every anchor point until no new qubits are encountered.

While the resulting subcircuit consists only of NOT, CNOT, and R_z gates, its qubit wires may not be continuous. To apply the phase polynomial formalism, we must ensure that there are no intermediate changes to the values on any wires that leave and re-enter the subcircuit. To achieve this, we perform the following pruning procedure. Starting with the designated initial CNOT gate, we successively consider gates both before and after it in the netlist until we encounter a termination point. Note that we only need to consider CNOT gates, since every NOT and R_z gate reached by this process can be included. If both the control and target qubits of an encountered CNOT gate are within the termination border, we continue. If the control qubit is outside the termination border but the target qubit is inside, we move the termination point of the target qubit so that the CNOT gate being inspected falls outside the border, excluding it and any subsequent gates acting on its target qubit from the subcircuit. However, when the control is inside the border and the target is outside, we make an exception and do not move the termination point (although we

[1] Note that in this particular example, the simplification could have alternatively been obtained using the commutation method described above. However, this is not the case in general.

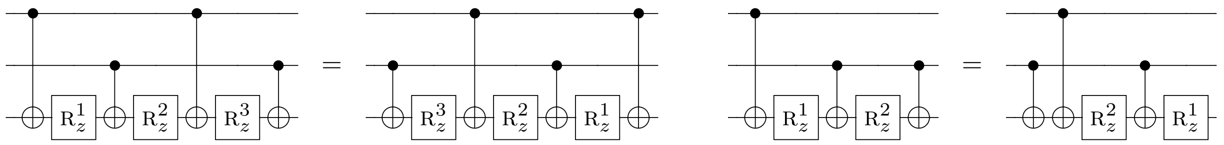
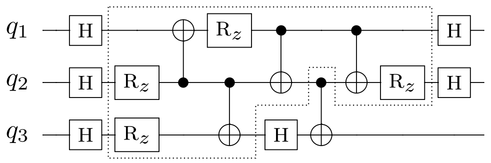


Figure 3: Gate count preserving rewriting rules employed in Routine 5.

do not include the CNOT gate in the subcircuit). This exception gives a larger $\{\mathrm{NOT}, \mathrm{CNOT}, \mathrm{R}_z\}$ subcircuit that remains amenable to phase polynomial representation, as in the following example:



(8)

In the example circuit (8), suppose we start our search from the first CNOT gate acting on the top (q_1) and middle (q_2) qubits. Traversing q_1 to the left, we find an H gate, where we mark a termination point. Traversing q_1 to the right, we find two CNOT gates, one R_z gate, and then an H gate, where we mark a termination point. Observe that neither of the encountered CNOT gates joins q_1 or q_2 to the remaining qubit q_3. Next, we repeat the same procedure on q_2 from the original CNOT gate. To the left we find an R_z gate and then an H gate, where we mark a termination point. Traversing to the right, we find a CNOT acting on q_2 and q_3. This CNOT reveals additional connectivity, so we mark an anchor point at the target of this CNOT gate. Further to the right on the q_2 wire, we have three more CNOT gates (none of which reveals additional connectivity), an R_z gate, and finally an H gate, where we mark a termination point. Next we examine q_3. We start from the aforementioned anchor point. To the left, we find an H gate with no further connections to other qubits, where we mark a termination point. To the right, we immediately find an H gate and mark a termination point.

Having built the subcircuit, we go through the netlist representation and prune it. In this pass, we encounter the fourth CNOT gate acting on q_2 and q_3, where we find that the control is within the border but the target is not. In this case we continue according to the exception handling scheme described in the pruning procedure. This ensures that we include the last CNOT gate in the $\{\mathrm{NOT}, \mathrm{CNOT}, \mathrm{R}_z\}$ region, while excluding the fourth CNOT gate (as indicated by the dotted border in (8)). Thus we discover that the last R_z gate appearing in the circuit can be relocated to the very beginning of the circuit on the q_2 line, to the right of the leftmost H, enabling a phase-polynomial based R_z merge (see below for details).

Once a valid $\{\mathrm{NOT}, \mathrm{CNOT}, \mathrm{R}_z\}$ subcircuit is identified, we generate its phase polynomial. For each R_z gate, we determine the associated affine function its phase is applied to and the location in the circuit where it is applied. We then sort the list of recorded affine functions. Finally, we find and merge all R_z gate repetitions, placing the merged R_z at the first location in the subcircuit that computes the desired affine function.

This procedure considers $O(g)$ subcircuits, and the cost of processing each of these is dominated by sorting, with complexity $O(g \log g)$, giving an overall complexity of $O(g^2 \log g)$ for Routine 4. However, in practice the subcircuits are typically smaller when there are more of them to consider, so the true complexity is lower. In addition, when identifying a $\{\mathrm{NOT}, \mathrm{CNOT}, \mathrm{R}_z\}$ subcircuit, we choose to start with a CNOT gate that has not yet been included in any of the previously-identified $\{\mathrm{NOT}, \mathrm{CNOT}, \mathrm{R}_z\}$ subcircuits, so the number of subcircuits can be much smaller than g in practice. If desired, the overall complexity can be lowered to $O(g)$ by limiting the maximal size of the subcircuit.

We now return to the description of optimization subroutines.

5. *Floating* R_z *gates*

In Routine 4, we keep track of the affine functions associated with R_z gates. More generally, we can record all affine functions that occur in the subcircuit and their respective locations, regardless of the presence of R_z gates. Thus we can identify all possible locations where an R_z gate could be placed, not just those locations where R_z gates already appear in the circuit. In this "floating" R_z gate placement picture, we employ three optimization subroutines: two-qubit gate cancellations, gate count preserving rewriting rules, and gate count reducing rewriting rules.

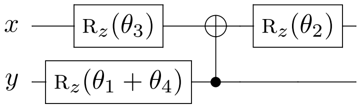
The first of these subroutines is essentially identical to Routine 3, except that R_z gates are now floatable and we focus on a specific identified subcircuit. This approach allows us to place R_z gates to facilitate cancellations by keeping track of all possible R_z gate locations along the way. In particular, if not placing an R_z gate at a particular location will allow two CNOT gates to cancel, we simply remove that location from the list of possible locations for the R_z gate while ensuring that the reduced list remains non-empty, and perform the CNOT cancellation.

We next apply rewriting rules that preserve the gate count (see Figure 3) in an attempt to find further optimizations. While these replacements do not eliminate gates, they modify the circuit in ways that can enable optimizations elsewhere. The rewriting rules are provided by an external library file, and we identify subcircuits to which they can be applied using the DAG representation. The replacements are applied only if they lead to a reduction in the two-qubit gate count through one more round of the aforementioned two-qubit cancellation subroutine with floatable R_z gates. Note that the rewriting rules are applicable only with certain floating R_z gates at particular locations in a circuit. This subroutine uses floating R_z gates to choose those combinations of R_z gate locations that lead to reduction in the gate count.

The last subroutine applies rewriting rules that reduce the gate count (see Figure 4). These rules are also provided via an external library file. Since these rules reduce the gate count on their own, we always perform the rewriting whenever a suitable pattern is found.

The complexity of this three-step routine is upper bounded by $O(g^3)$ since the number of subcircuits is $O(g)$, and within each subcircuit, the two-qubit cancellation (Routine 3) has complexity $O(g^2)$. The rewriting rules can be applied with complexity $O(g)$ since, as in Routine 1, a single pass through the gates in the circuit suffices. Again, in practice, the number of subcircuits and the subcircuit sizes are typically inversely related, which lowers the observed complexity by about a factor of g. The complexity can also be lowered to $O(g^2)$ by limiting the maximal size of the subcircuit. The complexity can be further lowered to $O(g \log g)$ by limiting the maximal size of the subcircuit A in the two-qubit gate cancellation (the sorting could still have complexity $O(g \log g)$).

To illustrate how this optimization works, consider the circuit in equation (7). Observe that $\mathrm{R}_z(\theta_2)$ may be executed on the top qubit at the end of the circuit, allowing the first two CNOTs to cancel, leading to the circuit



(9)

which is simplified even further.

3.4 General-purpose optimization algorithms

Our optimization algorithms simply apply the subroutines from Section 3.3 in a carefully chosen order. We consider two versions of the optimizer that we call *Light* and *Heavy*. The Heavy version applies more subroutines, yielding better optimization results at the cost of a higher runtime. The preprocessing step (see Section 3.2) is used in both Light and Heavy versions of the optimizer.

The Light version of the optimizer applies the optimization subroutines in the order

1, 3, 2, 3, 1, 2, 4, 3, 2.

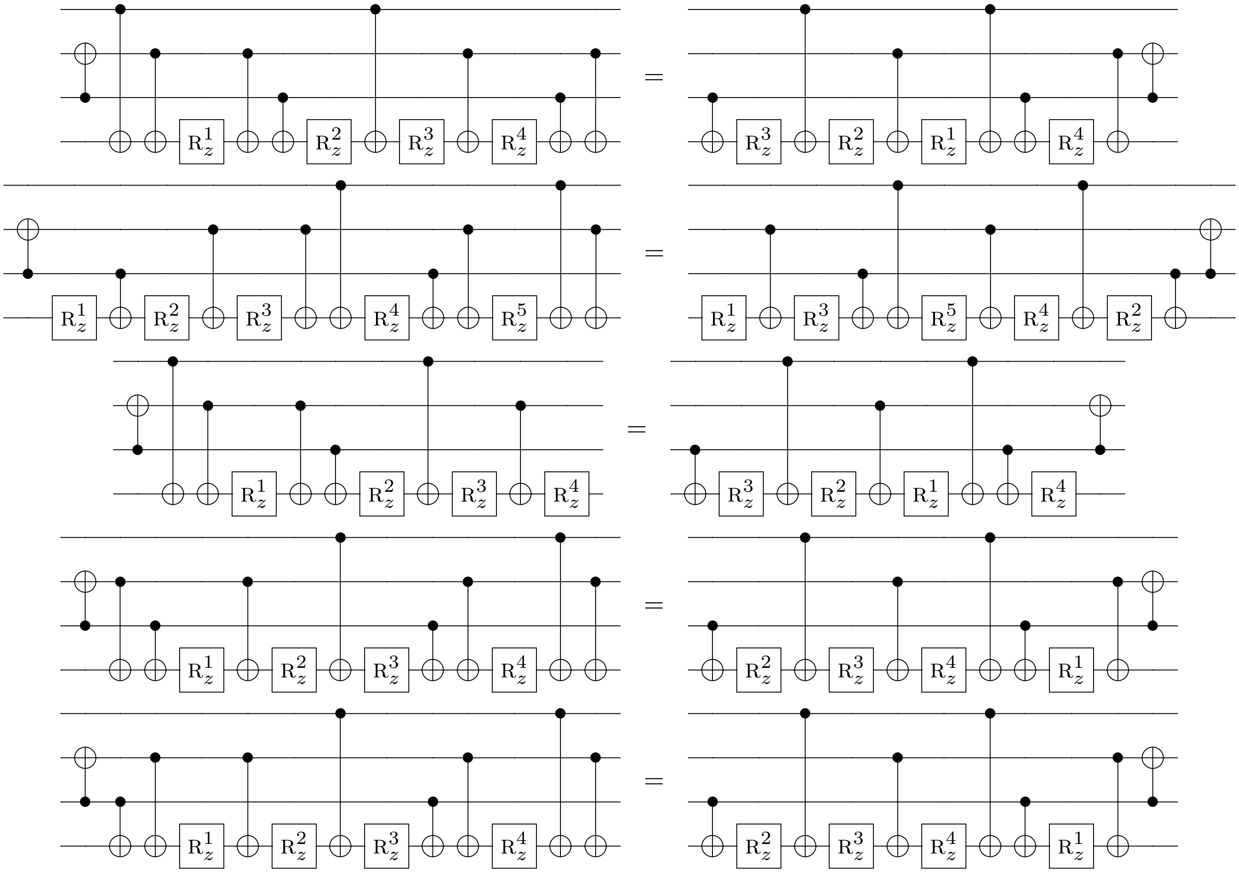


Figure 4: Gate count reducing rewriting rules employed in Routine 5.

We then repeat this sequence until no further optimization is achieved. We chose this sequence based on the principle that first exposing $\{\text{CNOT}, \text{R}_z\}$ gates while reducing Hadamard gates (1) allows for greater reduction in the cancellation routines (3, 2, 3), and in particular frees up two-qubit CNOT gates to facilitate single-qubit gate reductions and vice versa. Applying the replacement rule (1) may enable more reductions after the first four optimization subroutines. We then look for additional single-qubit gate cancellation and merging (2). This enables faster identification of the $\{\text{NOT}, \text{CNOT}, \text{R}_z\}$ subcircuit regions to look for further R_z count optimizations (4), after which we check for residual cancellations of the gates (3, 2).

The Heavy version of the optimizer applies the sequence

1, 3, 2, 3, 1, 2, 5.

Similarly, we repeat this sequence until no further optimization is achieved. The first six steps of the Heavy optimization sequence are identical to that of the Light optimizer. The difference is that in the Heavy optimizer, we take advantage of floating R_z gates. This allows us to find locations for the R_z gates that admit better CNOT gate reductions, including the use of gate count preserving rewriting rules to expose further gate cancellations and gate count reducing rewriting rules to remove any remaining inefficiency.

3.5 Special-purpose optimizations

In addition to the general-purpose optimization algorithms described above, we employ two specialized optimizations to improve circuits with particular structures.

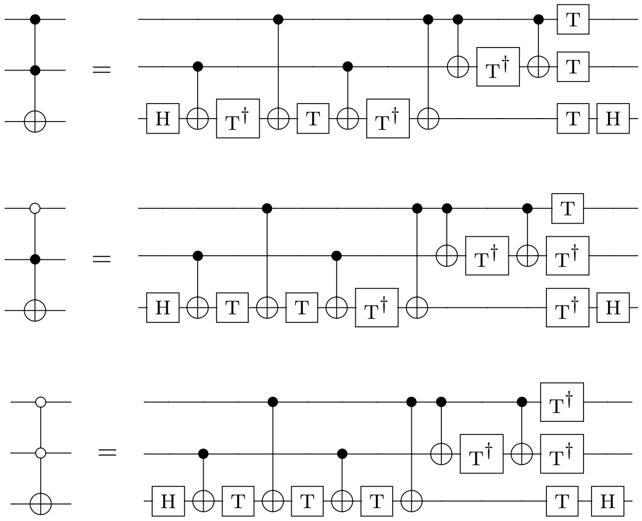


Figure 5: Toffoli gate implementations.

- *$\mathcal{LCR}$ optimizer:* Some quantum algorithms—such as product formula simulation algorithms—involve repeating a fixed block multiple times. To optimize such a circuit, we first run the optimizer on a single block to obtain its optimized version, $\mathcal{O}$. To find simplifications across multiple blocks, we optimize the circuit $\mathcal{O}^2$ and call the result $\mathcal{LR}$, where $\mathcal{L}$ is the maximal prefix of $\mathcal{O}$ in the optimization of $\mathcal{O}^2$. We then optimize $\mathcal{O}^3$. Provided optimizations only occur near the boundaries between blocks, we can remove the prefix $\mathcal{L}$ and the suffix $\mathcal{R}$ from the optimized version of $\mathcal{O}^3$, and call the remaining circuit $\mathcal{C}$. Assuming we can find such $\mathcal{L}$, $\mathcal{C}$, and $\mathcal{R}$ (which is always the case in practice), then we can simplify $\mathcal{O}^t$ to $\mathcal{LC}^{t-2}\mathcal{R}$.

- *Toffoli decomposition:* Many quantum algorithms are naturally described using Toffoli gates. Our optimizer can handle Toffoli gates with both positive and negative controls. Since we ultimately aim to express circuits over the gate set $\{\text{NOT}, \text{CNOT}, \text{H}, \text{R}_z\}$, we must decompose the Toffoli gate in terms of these elementary gates. We take advantage of different ways of doing this to improve the quality of optimization.

  Specifically, we expand the Toffoli gates in terms of one- and two-qubit gates using the identities shown in Figure 5, keeping in mind that we also obtain the desired Toffoli gate by exchanging T and T† in those circuit decompositions (because the Toffoli gate is self-inverse). Initially, the optimizer leaves the *polarity* of T/T† gates (i.e., the choice of which gates include the dagger and which do not) in each Toffoli decomposition undetermined. The optimizer symbolically processes the indeterminate T and T† gates by simply moving their locations in a given quantum circuit, keeping track of their relative polarities. The optimization is considered complete when movements of the indeterminate T and T† gates cannot further reduce the gate count. Finally, we choose the polarities of each Toffoli gate (subject to the fixed relationships between them) with the goal of minimizing the T count in the optimized circuit. We perform this minimization in a greedy way, choosing polarities for each Toffoli gate in the order of appearance of the associated T/T† gates in the nearly-optimized circuit, so as to reduce the T count as much as possible.

  Overall, this polarity selection process takes time $O(g)$. After choosing the polarities, we run Routine 3 and Routine 2, since particular choices of polarities may lead to further cancellations of the CNOT gates and single-qubit gates that were otherwise not possible due to the presence of the indeterminate gates blocking the desired commutations.

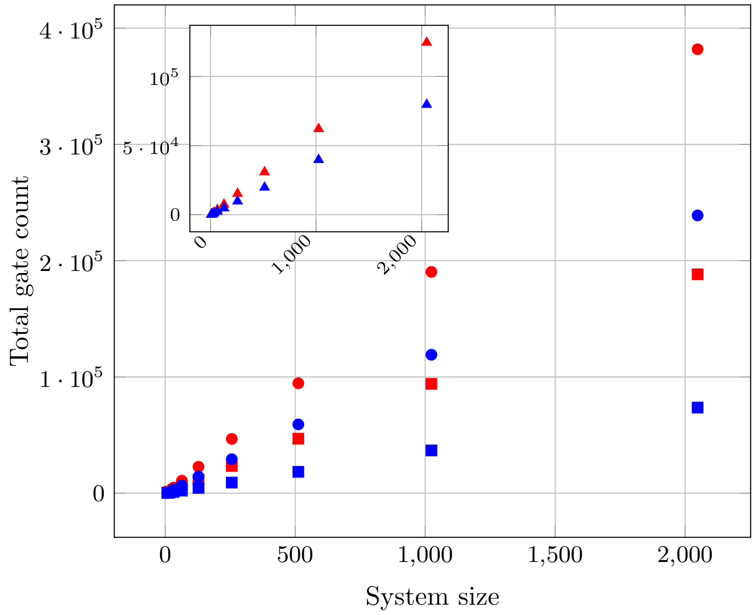


Figure 6: Total gate count for the approximate quantum Fourier transform (QFT, inset), Quipper library adder, and Fourier-based adders (QFA). The points in red/blue represent gate counts before/after optimization and the symbols square/circle/triangle represent gate counts for the Quipper library adder/QFA/QFT, respectively.

4 Results

We implemented our optimizer in the Fortran programming language and tested it using three sets of benchmark circuits. All results were obtained using a machine with a 2.9 GHz Intel Core i5 processor and 8 GB of 1867 MHz DDR3 memory, running OS X El Capitan.

We considered quantum circuits that include components of Shor's integer factoring algorithm, namely the *quantum Fourier transform* (*QFT*) and the integer adders. We also considered circuits for the *product formula* (*PF*) approach to Hamiltonian simulation [4]. In both cases, we focused on circuit sizes likely to be useful in applications that outperform classical computation, and ran experiments with different types of adders and product formulas. Finally, we considered a set of benchmark circuits from Ref. [1], consisting of various arithmetic circuits (including a family of Galois field multipliers) and implementations of multiple-control Toffoli gates. Files containing circuits before and after optimization are available at [29].

To check correctness of our optimizer, we verified the functional equivalence (i.e., equality of the corresponding unitary matrices) of various test circuits before and after optimization. Of course, such a test is only feasible for circuits with a small number of qubits. We performed this test for all 8-qubit benchmarks in Table 1 and Table 2, all 10-qubit benchmarks in Table 3, and the following benchmarks from Table 4: Mod 5_4, VBE-Adder$_3$, CSLA-MUX$_3$, RC-Adder$_6$, Mod-Red$_{21}$, Mod-Mult$_{55}$, Toff-Barenco$_{3..5}$, Toff-NC$_{3..5}$, GF(2^4)-Mult, and GF(2^5)-Mult.

4.1 QFT and adders

The QFT is a fundamental subroutine in quantum computation, appearing in many quantum algorithms with exponential speedup. The standard circuit for the exact n-qubit QFT uses R_z gates, some with angles that are exponentially small in n. It is well known that one can perform a highly accurate approximate QFT by omitting gates with very small rotation angles [7]. We choose to omit rotations by angles at most $\pi/2^{13}$, which ensures sufficient accuracy of the approximate QFT for circuits of the sizes we consider. These small rotations are removed before optimization, so their omission does not contribute to the improvements we report.

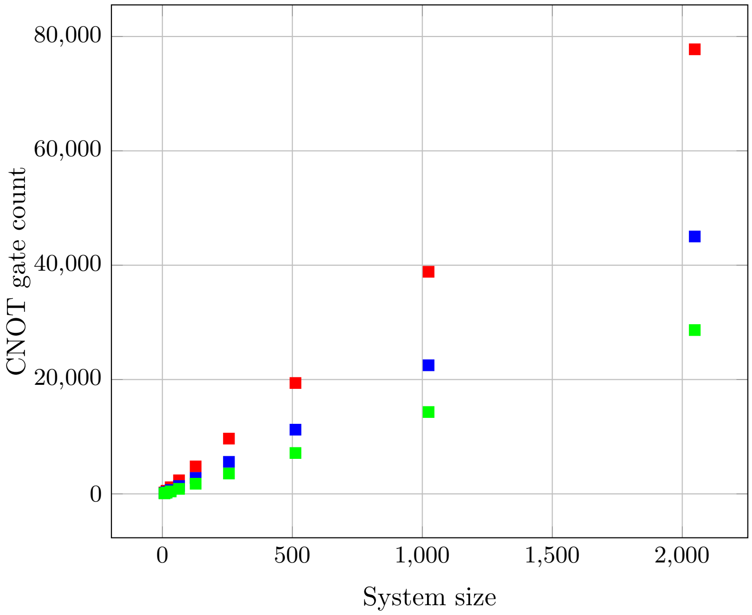


Figure 7: Number of CNOT gates for Quipper library adders. The points in red/blue/green represent the gate counts in pre-/post-Light/post-Heavy optimization, respectively.

In Figure 6 (inset) we plot total gate counts for the approximate QFT before and after optimization. We observe a savings ratio of larger than 36% for the QFT with 512 or more qubits. The optimization comes entirely from reducing the number of R_z gates, the most expensive resource in a fault-tolerant implementation.

We consider two types of integer adders: an in-place modulo 2^q adder as implemented in the Quipper library [13] and an in-place adder based on the QFT [9] (hereafter denoted *QFA*). The QFA circuits use an approximate QFT in which the rotations by angles less than $\pi/2^{13}$ are removed, as described above. Adders are a basic component of Shor's quantum algorithm for integer factoring [30]. We report gate counts before and after optimization for the Quipper adders and the QFAs for circuits acting on 2^L qubits, with L ranging from 4 to 11. Adders with $L = 10$ are used in Shor's algorithm for factoring 1,024-bit numbers. Recall that the related RSA-1024 challenge remains unsolved [37].

The results of Light optimization of the adder circuits are shown in Table 1 and Figure 6. For the Quipper library adders, we used the standard Light optimizer. For the QFA optimization, we instead used a modified Light optimizer with the sequence of routines 1, 3, 2, 3, 1, 2, omitting the final three routines 4, 3, 2 of the full Light optimizer. We did this because we saw no additional gate savings from those routines in small instances ($n \leq 256$).

Observe that the simplified Quipper library adder outperforms the QFA by a wide margin, suggesting that it may be preferred in practice. For the Quipper library adder, we see a reduction in the T gate count by a factor of up to 5.2. We emphasize that this reduction is obtained entirely by automated means, without using any prior knowledge of the circuit structure. Since Shor's integer factoring algorithm is dominated by the cost of modular exponentiation, which in turn relies primarily on integer addition, this optimization reduces the cost of executing the overall factoring algorithm by a factor of more than 5.

We also applied the Heavy optimizer to the QFT and adder circuits. For the QFT and QFA circuits, the Heavy setting does not improve the gate counts. The results of the Heavy optimization for the Quipper adder are shown in Table 2. We find a reduction in the CNOT count by a factor of 2.7, compared to a factor of only 1.7 for the Light optimization. Figure 7 illustrates the total CNOT counts of the Quipper library adder before optimization, after Light optimization, and after Heavy optimization, showing the reduction in the CNOT count by the two types of optimization.

Table 1: Light optimization of adder circuits: QFA (top) and Quipper library (bottom).

	Gate Counts for Approximate QFA						Software Runtime (seconds)
	Before Optimization			After Optimization			
n	CNOT	R_z	H	CNOT	R_z	H	
8	184	276	16	184	122	16	< 0.001
16	716	1,074	32	716	420	32	0.001
32	1,900	2,850	64	1,900	1,076	64	0.002
64	4,268	6,402	128	4,268	2,388	128	0.004
128	9,004	13,506	256	9,004	5,012	256	0.08
256	18,476	27,714	512	18,476	10,260	512	0.018
512	37,420	56,130	1024	37,420	20,756	1,024	0.045
1,024	75,308	112,962	2,048	75,308	41,748	2,048	0.115
2,048	151,084	226,626	4,096	151,084	83,732	4,096	0.215

	Gate Counts for Quipper Library Adder								Software Runtime (seconds)
	Before Optimization				After Optimization				
n	CNOT	T	H	S	CNOT	T	H	S	
8	243	266	76	0	143	56	28	12	0.001
16	547	602	172	0	319	120	60	28	0.003
32	1,155	1,274	364	0	671	248	124	60	0.014
64	2,371	2,618	748	0	1,375	504	252	124	0.057
128	4,803	5,306	1,516	0	2,783	1,016	508	252	0.244
256	9,667	10,682	3,052	0	5,599	2,040	1,020	508	1.099
512	19,395	21,434	6,124	0	11,231	4,088	2,044	1,020	5.292
1,024	38,851	42,938	12,268	0	22,495	8,184	4,092	2,044	25.987
2,048	77,763	85,946	24,556	0	45,023	16,376	8,188	4,092	145.972

4.2 Quantum simulation

The first explicit polynomial-time quantum algorithm for simulating Hamiltonian dynamics was introduced in [23]. This approach was later generalized to higher-order product formulas [4], giving improved asymptotic complexities. We report gate counts before and after optimization for the PF algorithms of orders 1, 2, 4, and 6 (for orders higher than 1, the order of the standard Suzuki construction is even). For concreteness, we implement these algorithms for a one-dimensional Heisenberg model with periodic boundary conditions in a random, site-dependent magnetic field, evolving the system for the time proportional to its size, and choose the algorithm parameters to ensure the Hamiltonian simulation error is at most 10^{-3} using known bounds on the error of the product formula approximation.

The results of Light optimization of product formula algorithms are reported in Table 3 and illustrated in Figure 8. For these algorithms, we find that Heavy optimization offers no further improvement. The 2nd-, 4th-, and 6th-order algorithms admit a ~33.3% reduction in the CNOT count and a ~28.5% reduction in the R_z count, roughly corresponding to the reductions relevant to physical-level and logical-level implementations. The 1st-order formula algorithm did not exhibit CNOT or R_z gate optimization. In all product formula algorithms, the number of Phase and Hadamard gates reduced significantly, by a factor of roughly 3 to 6.

4.3 Comparison with prior approaches

Quantum circuit optimization is already a well-developed field (see for example [1, 27, 31, 33]). However, to the best of our knowledge, no prior work on circuit optimization has considered large-scale quantum circuits of the kind that could outperform classical computers. For instance, in [1], the complexity of optimizing a g-gate circuit is $O(g^3)$ (sections 6.1 and 7), making optimization of large-scale circuits unrealistic. Table 3

Table 2: Heavy optimization of Quipper library adder.

	Gate Counts for Quipper Library Adder								Software Runtime (seconds)
	Before Optimization				After Optimization (H)				
n	CNOT	T	H	S	CNOT	T	H	S	
8	243	266	76	0	94	56	28	12	0.006
16	547	602	172	0	206	120	60	28	0.018
32	1,155	1,274	364	0	430	248	124	60	0.066
64	2,371	2,618	748	0	878	504	252	124	0.598
128	4,803	5,306	1,516	0	1,774	1,016	508	252	4.697
256	9,667	10,682	3,052	0	3,566	2,040	1,020	508	34.431
512	19,395	21,434	6,124	0	7,150	4,088	2,044	1,020	307.141
1,024	38,851	42,938	12,268	0	14,318	8,184	4,092	2,044	2,446.336
2,048	77,763	85,946	24,556	0	28,654	16,376	8,188	4,092	23,886.841

in [27] shows running times ranging from 0.07 to 1.883 seconds for numbers of qubits from $n = 10$ to 35 and gate counts from 60 to 368, whereas our optimizer ran for a comparable time when optimizing the Quipper adders up to $n = 256$ with around 23,000 gates, as shown in Table 1. Reference [31] relies on peep-hole optimization using optimal gate libraries. This is expensive, as is evidenced by the runtimes reported in Tables I and II therein, taking already more than 100 seconds for a 20-qubit, 1,000-gate circuit.

To compare our results to those reported previously, we consider T count, CNOT count, and a scalar cost metric that accounts for the relative difficulty of performing CNOT and T gates in a fault-tolerant implementation. While the T gate is considerably more expensive due to the need for state distillation [5], neglecting the cost of the CNOT gates may lead to a significant underestimate if there are many such gates [26]. Roughly speaking, a fault-tolerant T gate may be about $10-100$ times more expensive to implement than a local, fault-tolerant CNOT gate. The true overhead depends on many details, including the fault tolerance scheme, the error model, the size of the computation, architectural restrictions, the extent to which the implementation of the T gate can be optimized, and whether T state production happens offline so its cost can be (partially) discounted; it is beyond the scope of this paper to account for all these factors. For a rough comparison, we suppose that the T gate is 20 times as expensive as a typical CNOT gate, and we call the CNOT gate count plus 20 times the T gate count the *aggregate cost*.

We directly compare our results to those reported in [1], which aims to reduce the T count and T depth using techniques based on matroid partitioning. We refer to that approach as T-*par*. We use our approach to optimize a set of benchmark circuits appearing in that work and compare the results with the T-par optimization, as shown in Table 4.

The benchmark circuits fall into three categories. The first set consists of a selection of arithmetic operations. For these circuits, we obtained better or matching T counts compared to [1] while also obtaining much better CNOT counts. Note that we excluded circuit CSLA-MUX$_3$ from the comparison since we do not believe T-par optimized it correctly (for more detail, see the first footnote in Table 4). To illustrate the advantage of our approach using the aggregate cost metric, observe that we reduced the cost of the RC-Adder$_6$ circuit from 1,494 to 1,011.

The second set of benchmarks consists of multiple-control Toffoli gates. While our optimizer matched the T count obtained by the T-par and substantially reduced the CNOT count, neither our optimizer nor [1] could find the best known implementations constructed directly in [25]. This is not surprising, given the very different circuit structure employed in [25].

The third set of benchmarks contains Galois field multiplier circuits. We saw no advantage from the Heavy optimizer over the Light optimizer in the cases we tested, so we did not apply the Heavy optimizer to the four largest instances (the corresponding entries are left blank in Table 4). Our T count again matches that of the T-par optimizer, but our CNOT count is much lower, resulting in the circuits that are clearly preferred. For example, the optimized GF(2^{64}) multiplier circuit in [1] uses 180,892 CNOT gates, whereas our optimized implementation uses only 24,765 CNOT gates; the aggregate cost is thus reduced from 509,852 to 353,725 despite no change in the T count, illustrating the advantage of our approach. This comparison

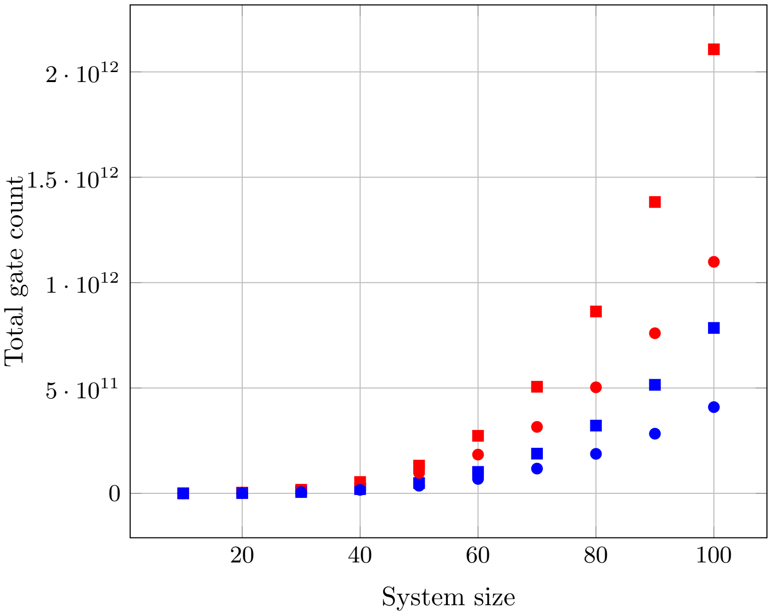


Figure 8: Total gate count for product formula algorithms. The points in red/blue represent gate counts before/after optimization and the symbols square/circle represent gate counts for the 2nd-/4th-order formula, respectively.

demonstrates that the discrepancy between T count and true cost predicted in theory [26] is manifested in practice. The efficiency of our Light optimizer allowed us to optimize of the GF(2^{131}) and GF(2^{163}) multiplier quantum circuits, corresponding to instances of the elliptic curve discrete logarithm problem that remain unsolved [6]. Given the reported T-par runtimes [1], an instance of this size appears to be intractable for the T-par optimizer.

4.4 Overall performance

Our numerical optimization results are summarized across Table 1, Table 2, Table 3, and Table 4. These tables contain benchmarks relevant to practical quantum computations that are beyond the reach of classical computers. In Table 1 and Table 2 these are the 1,024- and 2,048-qubit QFT and integer adders used in classically-intractable instances of Shor's factoring algorithm [37]. In Table 3 these include all instances with $n \gtrsim 50$, for which direct classical simulation of quantum dynamics is currently infeasible. In Table 4 these are Galois field multipliers over binary fields of sizes 131 and 163, which are relevant to quantum attacks on unsolved Certicom ECC Challenge problems [6]. This illustrates that our optimizer is capable of handling quantum circuits that are sufficiently large to be practically relevant.

Our optimizer can be applied more generally than previous work on circuit optimization. It readily accepts composite gates, such as Toffoli gates (which may have negated controls). It also handles gates with continuous parameters, a useful feature for algorithms that naturally use R_z gates, including Hamiltonian simulation and factoring. Many quantum information processing technologies natively support such gates, including both trapped ions [8] and superconducting circuits [16], so our approach may be useful for optimizing physical-level circuits.

Fault-tolerant quantum computations generally rely on a discrete gate set, such as Clifford+T, and optimal Clifford+T implementations of R_z gates are already known [21, 32]. Nevertheless, the ability to optimize circuits with continuous parameters is also valuable in the fault-tolerant setting. This is because optimizing with respect to a natural continuously-parametrized gate set before compiling into a discrete fault-tolerant set will likely result in smaller final circuits.

Finally, unlike previous approaches [1, 27, 31], our optimizer preserves the structure of the original circuit.

Table 3: Optimization of product formula algorithms, showing the CNOT gate count reduction (top) and the R_z gate count reduction (bottom). Software runtimes range from 0.004 s (1st-order, $n = 10$) to 0.137 s (6th-order, $n = 100$). The Clifford gate reduction ranges from 62.5% for Hadamard and 75% for Phase gates (for the 1st-order formula, independent of n) to 75% for Hadamard and 85% for Phase gates (for the 6th-order formula, again independent of n). The notation "(× 1000)" indicates that the gate counts for the 1st-order formula are in units of thousands (no rounding errors). The notation "(L)" denotes the standard Light optimization.

	CNOT Counts for Product Formula Algorithms							
	1st order		2nd order		4th order		6th order	
n	Before (×1000)	After (L) (×1000)	Before	After (L)	Before	After (L)	Before	After (L)
10	9,600,024	9,600,024	49,622,280	33,081,540	82,152,000	54,768,020	833,073,000	555,382,020
20	307,200,192	307,200,192	793,571,040	529,047,400	927,468,000	618,312,040	8,376,270,000	5,584,180,040
30	2,332,800,648	2,332,800,648	4,016,805,120	2,677,870,140	3,830,076,000	2,553,384,060	32,322,240,000	21,548,160,060
40	9,830,401,536	9,830,401,536	12,694,063,680	8,462,709,200	10,477,257,600	6,984,838,480	84,262,560,000	56,175,040,080
50	30,000,003,000	30,000,003,000	30,989,866,200	20,659,910,900	22,869,948,000	15,246,632,100	177,187,560,000	118,125,040,100
60	74,649,605,184	74,649,605,184	64,258,513,920	42,839,009,400	43,278,861,600	28,852,574,520	325,230,480,000	216,820,320,120
70	161,347,208,232	161,347,208,232	119,044,086,000	79,362,724,140	74,215,289,400	49,476,859,740	543,505,116,000	362,336,744,140
80	314,572,812,288	314,572,812,288	203,080,443,840	135,386,962,720	118,409,788,800	78,939,859,360	847,991,544,000	565,327,696,160
90	566,870,417,496	566,870,417,496	325,291,230,720	216,860,820,660	178,795,738,800	119,197,159,380	1,255,450,374,000	836,966,916,180
100	960,000,024,000	960,000,024,000	495,789,866,400	330,526,577,800	258,496,092,000	172,330,728,200	1,783,355,700,000	1,188,903,800,200

	R_z Counts for Product Formula Algorithms							
	1st order		2nd order		4th order		6th order	
n	Before (×1000)	After (L) (×1000)	Before	After (L)	Before	After (L)	Before	After (L)
10	6,400,016	6,400,016	28,946,330	20,675,960	47,922,000	34,230,010	485,959,250	347,113,760
20	204,800,128	204,800,128	462,916,440	330,654,620	541,023,000	386,445,020	4,886,157,500	3,490,112,520
30	1,555,200,432	1,555,200,432	2,343,136,320	1,673,668,830	2,234,211,000	1,595,865,030	18,854,640,000	13,467,600,030
40	6,553,601,024	6,553,601,024	7,404,870,480	5,289,193,240	6,111,733,600	4,365,524,040	49,153,160,000	35,109,400,040
50	20,000,002,000	20,000,002,000	18,077,421,950	12,912,444,300	13,340,803,000	9,529,145,050	103,359,410,000	73,828,150,050
60	49,766,403,456	49,766,403,456	37,484,133,120	26,774,380,860	25,246,002,600	18,032,859,060	189,717,780,000	135,512,700,060
70	107,564,805,488	107,564,805,488	69,442,383,500	49,601,702,570	43,292,252,150	30,923,037,320	317,044,651,000	226,460,465,070
80	209,715,208,192	209,715,208,192	118,463,592,240	84,616,851,680	69,072,376,800	49,337,412,080	494,661,734,000	353,329,810,080
90	377,913,611,664	377,913,611,664	189,753,217,920	135,538,012,890	104,297,514,300	74,498,224,590	732,346,051,500	523,104,322,590
100	640,000,016,000	640,000,016,000	289,210,755,400	206,579,111,100	150,789,387,000	107,706,705,100	1,040,290,825,000	743,064,875,100

In particular, the set of two-qubit interactions used by the optimized circuit is a subset of those used in the original circuit. This holds because neither the preprocessing step nor our optimizations introduce any new two-qubit gates. By keeping the number of interactions under control (in stark contrast to T-par, which dramatically increases the set of interactions used), our optimized implementations are better suited for architectures with limited connectivity. For example, given a layout of the original quantum circuit on hardware with limited connectivity, this property allows one to use the same layout for the optimized circuit.

5 Conclusions and future work

In this paper, we studied the problem of optimizing large-scale quantum circuits, namely those appearing in quantum computations that are beyond the reach of classical computers. We developed Light and Heavy optimization algorithms and implemented them in software. Our algorithms are based on a carefully chosen sequence of basic optimizations, yet they achieve substantial reductions in the gate counts, improving over more mathematically sophisticated approaches such as T-par optimization [1]. The simplicity of our approach is reflected in very fast runtimes, especially using the Light version of the optimizer.

We expect that further improvements can lead to even greater circuit optimization, as demonstrated by the Heavy version of our optimizer. To further improve the output, one could revise the routines for reducing R_z count by implementing more extensive (and thus more computationally demanding) algorithms for composing stages of CNOT and R_z gates, possibly with some Hadamard gates included. One may also consider incorporating template-based [27] and peep-hole [31] optimizations. It may be worthwhile to expand the set of subcircuit rewriting rules and explore the performance of the approach on other benchmark circuits. Finally, considering the relative cost of different resources (e.g., different types of gates, ancilla qubits) could lead to optimizers that favorably trade off these resources.

Table 4: T-par comparison. The names of the algorithms are taken verbatim from Ref. [1], except that we write Toff-Barenco and Toff-NC to denote implementations of multiple-control Toffoli gates from [3] and [30], respectively. The notation "(L)" denotes the standard Light optimization, whereas "(H)" denotes the standard Heavy optimization. The symbol —〃— indicates that there was no improvement in the Heavy optimization over the Light optimization.

Circuit	Pre-Optimization			Ref. [1] Post-Optimization				Our Post-Optimization (L)				Our Post-Optimization (H)			
	Total	CNOT	T	Total	CNOT	T	Runtime (s)	Total	CNOT	T	Runtime (s)	Total	CNOT	T	Runtime (s)
Mod 5_4	63	28	28	76	48	16	< 0.001	51	28	16	< 0.001	—〃—	—〃—	—〃—	0.001
VBE-Adder $_3$	150	70	70	161	114	24	0.001	89	50	24	< 0.001	—〃—	—〃—	—〃—	0.001
CSLA-MUX $_3$	170	80	70	508	425	62[a]	0.001	161	76	64	< 0.001	155	70	64	0.009
CSUM-MUX $_9$	420	168	196	593	411	112[b]	0.005	294	168	84	< 0.001	266	140	84	0.009
QCLA-Com $_7$	443	186	203	751	583	95	0.003	284	132	95	0.001	—〃—	—〃—	—〃—	0.016
QCLA-Mod $_7$	884	382	413	1,572	1,185	249	0.008	636	302	237	0.004	624	292	235	0.077
QCLA-Adder $_{10}$	521	233	238	972	737	162	0.018	411	195	162	0.002	399	183	162	0.044
Adder $_8$	900	409	399	1,288	920	215	0.004	646	331	215	0.004	606	291	215	0.101
RC-Adder $_6$	200	93	77	326	234	63	0.001	142	73	47	< 0.001	140	71	47	0.004
Mod-Red $_{21}$	278	105	119	425	301	73	0.001	184	81	73	< 0.001	180	77	73	0.008
Mod-Mult $_{55}$	119	48	49	223	166	37	< 0.001	91	40	35	< 0.001	—〃—	—〃—	—〃—	0.002
Toff-Barenco $_3$	58	24	28	82	54	16	< 0.001	42	20	16	< 0.001	40	18	16	0.001
Toff-NC $_3$	45	18	21	65	41	15	< 0.001	35	14	15	< 0.001	—〃—	—〃—	—〃—	< 0.001
Toff-Barenco $_4$	114	48	56	141	90	28	< 0.001	78	40	28	< 0.001	72	34	28	0.001
Toff-NC $_4$	75	30	35	102	63	23	< 0.001	55	22	23	< 0.001	—〃—	—〃—	—〃—	< 0.001
Toff-Barenco $_5$	170	72	84	206	132	40	0.001	114	60	40	< 0.001	104	50	40	0.003
Toff-NC $_5$	105	42	49	148	94	31	< 0.001	75	30	31	< 0.001	—〃—	—〃—	—〃—	0.001
Toff-Barenco $_{10}$	450	192	224	517	328	100	0.004	294	160	100	0.001	264	130	100	0.012
Toff-NC $_{10}$	255	102	119	361	232	71	0.002	175	70	71	< 0.001	—〃—	—〃—	—〃—	0.004
GF(2^4)-Mult	225	99	112	419	324	68	0.001	187	99	68	0.001	—〃—	—〃—	—〃—	0.009
GF(2^5)-Mult	347	154	175	682	535	111	0.004	296	154	115	0.001	—〃—	—〃—	—〃—	0.020
GF(2^6)-Mult	495	221	252	842	649	150	0.008	403	221	150	0.003	—〃—	—〃—	—〃—	0.047
GF(2^7)-Mult	669	300	343	1,245	992	217	0.031	555	300	217	0.004	—〃—	—〃—	—〃—	0.105
GF(2^8)-Mult	883	405	448	1,560	1,256	264	0.052	712	405	264	0.006	—〃—	—〃—	—〃—	0.192
GF(2^9)-Mult	1,095	494	567	2,096	1,701	351	0.110	891	494	351	0.010	—〃—	—〃—	—〃—	0.347
GF(2^{10})-Mult	1,347	609	700	2,655	2,176	410	0.227	1,070	609	410	0.009	—〃—	—〃—	—〃—	0.429
GF(2^{16})-Mult	3,435	1,581	1,792	7,714	6,592	1,040	5.079	2,707	1,581	1,040	0.065	—〃—	—〃—	—〃—	5.566
GF(2^{32})-Mult	13,562	6,268	7,168	37,563	33,269	4,128	602.577	10,601	6,299	4,128	1.834	—〃—	—〃—	—〃—	275.698
GF(2^{64})-Mult	61,629	24,765	28,672	197,674	180,892	16,448	95,447.466	41,563	24,765	16,448	58.341				
GF(2^{128})-Mult	246,141	98,685	114,688	N/A	N/A	N/A	N/A	165,051	98,685	65,664	1,744.746				
GF(2^{131})-Mult	258,065	103,616	120,127	N/A	N/A	N/A	N/A	173,370	103,616	69,037	1,953.353				
GF(2^{163})-Mult	399,021	159,900	185,983	N/A	N/A	N/A	N/A	267,558	159,900	106,765	4,955.927				

[a] Our simulation found an error in the circuit optimized by T-par. Specifically, the circuit maps $|1024\rangle \mapsto \frac{|1025\rangle+|1030\rangle+|1161\rangle+|1166\rangle}{2}$ whereas it is supposed to perform the mapping $|1024\rangle \mapsto |1088\rangle$.

[b] Note that our software reduced the T-count of the original pre-optimization circuit used by T-par to 0. It turned out that the circuit used by T-par is incorrect. In our optimization reported in this table, we used the correct original circuit [36, Figure 5].

Acknowledgements

This work was supported in part by the Army Research Office (grant W911NF-16-1-0349), the Canadian Institute for Advanced Research, and the National Science Foundation (grant CCF-1526380).

This material was partially based on work supported by the National Science Foundation during DM's assignment at the Foundation. Any opinion, finding, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

References

[1] M. Amy, D. Maslov, and M. 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. Preprint available from arXiv:1303.2042v2.

[2] M. Amy, D. Maslov, M. Mosca, and M. Roetteler. A meet-in-the-middle algorithm for fast synthesis of depth-optimal quantum circuits. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 32(6):818–830, 2013. Preprint available from arXiv:1206.0758.

[3] A. Barenco, C. H. Bennett, R. Cleve, D. P. DiVincenzo, N. Margolus, P. Shor, T. Sleator, J. Smolin, and H. Weinfurter. Elementary gates for quantum computation. *Physical Review A*, 52(5):3457–3467, 1995. Preprint available from arXiv:quant-ph/9503016.

[4] D. W. Berry, G. Ahokas, R. Cleve, and B. C. Sanders. Efficient quantum algorithms for simulating sparse Hamiltonians. *Communications in Mathematical Physics*, 270(2):359–371, 2007. Preprint available from arXiv:quant-ph/0508139.

[5] S. Bravyi and A. Kitaev. Universal quantum computation with ideal Clifford gates and noisy ancillas. *Physical Review A*, 71:022316, 2005. Preprint available from arXiv:quant-ph/0403025.

[6] Certicom. The Certicom ECC challenge. Last accessed: October 19, 2017. https://www.certicom.com/...ecc-challenge.html.

[7] D. Coppersmith. An approximate Fourier transform useful in quantum factoring. 1994. Preprint available from arXiv:quant-ph/0201067.

[8] S. Debnath, N. M. Linke, C. Figgatt, K. A. Landsman, K. Wright, and C. Monroe. Demonstration of a small programmable quantum computer with atomic qubits. *Nature*, 536:63–66, 2016. Preprint available from arXiv:1603.04512.

[9] T. Draper. Addition on a quantum computer. 2000. Preprint available from arXiv:quant-ph/0008033.

[10] EPSRC. UK national quantum technologies programme. Last accessed: October 19, 2017. http://uknqt.epsrc.ac.uk.

[11] R. P. Feynman. Simulating physics with computers. *International Journal of Theoretical Physics*, 21(6-7):467–488, 1982.

[12] E. Gibney. Europes billion-euro quantum project takes shape. *Nature*, 545(7652):16, May 3, 2017. https://www.nature.com/news/europe-s-billion-euro-quantum-project-takes-shape-1.21925.

[13] A. S. Green, P. L. Lumsdaine, N. J. Ross, P. Selinger, and B. Valiron. Quipper: A scalable quantum programming language. *ACM SIGPLAN Notices*, 48(6):333–342, 2013. Preprint available from arXiv:1304.3390.

[14] R. Hackett. IBM sets sight on quantum computing. *Fortune*, March 6, 2017. http://fortune.com/2017/03/06/ibm-quantum-computer/.

[15] IBM. IBM makes quantum computing available on IBM cloud to accelerate innovation. May 4, 2016. https://www-03.ibm.com/press/us/en/pressrelease/49661.wss.

[16] IBM Research. Quantum Experience. Last accessed: September 22, 2017. http://www.research.ibm.com/quantum/.

[17] Intel. Intel invests US$50 million to advance quantum computing. September 3, 2015. https://newsroom.intel.com/news-releases/intel-invests-us50-million-to-advance-quantum-computing/.

[18] D. Janzing, P. Wocjan, and T. Beth. Identity check is QMA-complete. 2003. Preprint available from arXiv:quant-ph/0305050.

[19] S. P. Jordan. Quantum Algorithm Zoo. Last accessed: October 19, 2017. http://math.nist.gov/quantum/zoo/.

[20] R. Juskalian. Practical quantum computers. *MIT Technology Review*, March/April 2017. https://www.technologyreview.com/...practical-quantum-computers/.

[21] V. Kliuchnikov, D. Maslov, and M. Mosca. Fast and efficient exact synthesis of single qubit unitaries generated by Clifford and T gates. *Quantum Information & Computation*, 13(7–8):607–630, 2013. Preprint available from arXiv:1206.5236v4.

[22] A. K. Lenstra, H. W. Lenstra, Jr., M. S. Manasse, and J. M. Pollard. The number field sieve. In *Proceedings of the Twenty-second Annual ACM Symposium on Theory of Computing*, pages 564–572, 1990.

[23] S. Lloyd. Universal quantum simulators. *Science*, 273(5278):1073–1078, 1996.

[24] J. Markoff. Microsoft spends big to build a computer out of science fiction. November 21, 2016. https://www.nytimes.com/.../microsoft-spends-big-to-build-quantum-computer.html.

[25] D. Maslov. Advantages of using relative-phase Toffoli gates with an application to multiple control Toffoli optimization. *Physical Review A*, 93:022311, 2016. Preprint available from arXiv:1508.03273.

[26] D. Maslov. Optimal and asymptotically optimal NCT reversible circuits by the gate types. *Quantum Information and Computation*, 16(13-14):1096–1112, 2016. Preprint available from arXiv:1602.02627.

[27] D. Maslov, G. W. Dueck, D. M. Miller, and C. Negrevergne. Quantum circuit simplification and level compaction. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 27(3):436–444, 2008. Preprint available from arXiv:quant-ph/0604001.

[28] National Science and Technology Council. Advancing Quantum Information Science: National Challenges and Opportunities. July, 2016. https://www.whitehouse.gov/...final.pdf.

[29] Y. Nam, N. J. Ross, Y. Su, A. M. Childs, and D. Maslov, https://github.com/njross/optimizer.

[30] M. A. Nielsen and I. L. Chuang. *Quantum Computation and Quantum Information*. Cambridge University Press, 2002.

[31] A. K. Prasad, V. V. Shende, I. L. Markov, J. P. Hayes, and K. N. Patel. Data structures and algorithms for simplifying reversible circuits. *ACM Journal of Emerging Technologies in Computing Systems*, 2(4):277–293, 2006.

[32] N. J. Ross and P. Selinger. Optimal ancilla-free Clifford+T approximation of z-rotations. *Quantum Information & Computation*, 16(11&12):901–953, 2016. Preprint available from arXiv:1403.2975.

[33] M. Saeedi and I. L. Markov. Synthesis and optimization of reversible circuits—a survey. *ACM Computing Surveys*, 45(2): Article 21, 2013. Preprint available from arXiv:1110.2574.

[34] P. W. Shor. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM Journal on Computing*, 26(5):1484–1509, 1997. Preprint available from arXiv:quant-ph/9508027.

[35] T. Simonite. Google's quantum dream machine. *MIT Technology Review*, December 18, 2015. https://www.technologyreview.com/s/544421/googles-quantum-dream-machine/.

[36] R. Van Meter and K. M. Itoh. Fast quantum modular exponentiation. *Physical Review A*, 71:052320, 2005. Preprint available from arXiv:quant-ph/0408006.

[37] Wikipedia. RSA factoring challenge. Last accessed: October 19, 2017. https://en.wikipedia.org/wiki/RSA_Factoring_Challenge.