ETH Library

Code Optimization for Quantum Computing

Doctoral Thesis

Author(s):
Häner, Thomas

Publication date:
2018

Permanent link:
https://doi.org/10.3929/ethz-b-000320035

Rights / license:
In Copyright - Non-Commercial Use Permitted

This page was generated automatically upon download from the ETH Zurich Research Collection.
For more information, please consult the Terms of use.

DISS. ETH NO. 25562

Code optimization for quantum computing

A thesis submitted to attain the degree of
DOCTOR OF SCIENCES of ETH ZURICH
(Dr. sc. ETH Zurich)

presented by

THOMAS HÄNER

Master of Science ETH in Computational Science and Engineering

born on 23.09.1990

citizen of
Zullwil (SO)

accepted on the recommendation of
Prof. Dr. Matthias Troyer, examiner
Prof. Dr. Torsten Hoefler, co-examiner
Dr. Martin Roetteler, co-examiner

2018

Thomas Häner
Code optimization for quantum computing
Diss. ETH No. 25562

Digital Object Identifier doi:10.3929/ethz-b-000320035

E-mail haenert@phys.ethz.ch

Abstract

This thesis reports on different aspects of code optimization for quantum computing in three parts. The first pertains to automatic optimization of quantum circuits by compilers, the second to optimizing code for classical simulation of quantum algorithms, and the third to developing optimized circuit designs for evaluating classical functions on quantum computers.

More specifically, we develop a software methodology for compiling quantum programs in the first part of this thesis. The methodology aims to provide the necessary layers of abstraction in order to make the implementation of high-level quantum algorithms less time-consuming and the resulting code more efficient. We introduce a new optimization methodology which inspects the Hoare triples of all invoked subroutines to exploit optimization opportunities that could not be identified as such by previous methods. As an additional approach to optimization, we address the problem of managing approximation errors that occur during compilation and illustrate the benefits and drawbacks of our proposed solution.

In the second part, we reduce the resources required for classical simulation of quantum computers via two different approaches. The first is a fully-optimized state vector simulator which was used to simulate 45 qubits on the Cori II supercomputer. Subsequently, we introduce the concept of a quantum circuit *emulator* and demonstrate that this new approach is able to outperform simulators by several orders of magnitude, especially for quantum circuits that evaluate mathematical functions.

The third and final part is devoted to manual optimization of quantum circuits. In particular, we focus on circuits for integer and fixed-point arithmetic. We develop a new addition circuit which allows for space-savings by borrowing idle qubits from other parts of the computation. These idle qubits may be in an arbitrary state and entangled with other qubits in the system. In addition, we then use our construction to implement Shor's algorithm and achieve an asymptotic scaling advantage over previous space-efficient implementations. Finally, we develop a fixed-point library for evaluating mathematical functions that are often encountered in the quantum algorithm literature. To evaluate these functions, our scheme efficiently combines a host of low-degree polynomials, each approximating the function on a small subdomain. This allows to obtain very accurate approximations at a cost that is similar to evaluating just a single low-degree polynomial.

Zusammenfassung

Diese Arbeit behandelt unterschiedliche Aspekte der Codeoptimierung für Quantencomputing in drei Teilen. Der erste Teil ist der automatischen Optimierung von Quantenschaltkreisen durch Compiler gewidmet. Der zweite Teil befasst sich mit der Optimierung klassischer Simulationen von Quantenalgorithmen und im dritten Teil werden optimierte Quantenschaltkreise für die Auswertung klassischer Funktionen auf Quantenrechnern entwickelt.

In einem ersten Schritt wird eine Softwaremethode zur Kompilierung von Quantenprogrammen erarbeitet und die nötigen Abstraktionsebenen mit dem Ziel eingeführt, das Implementieren abstrakter Quantenalgorithmen weniger zeitaufwändig und den resultierenden Code effizienter zu machen. Anschliessend wird eine neuartige Optimierungsmethode für Quantenschaltkreise eingeführt, welche die Hoare-Tripel aller aufgerufenen Subroutinen in Betracht zieht, um Optimierungsmöglichkeiten zu nutzen, die mit bisherigen Methoden nicht als solche erkannt werden konnten. Als eine weitere Möglichkeit der Schaltkreisoptimierung wird das Bestimmen der Fehlertoleranzen während des Kompilierens behandelt und die Vor- und Nachteile der vorgeschlagenen Lösungsmethode werden diskutiert.

Im zweiten Teil der Arbeit werden die für die Simulation von Quantenrechnern auf klassischer Hardware erforderlichen Rechenressourcen mithilfe zweier unterschiedlicher Herangehensweisen verringert. Bei der ersten handelt es sich um einen optimierten Zustandsvektorsimulator, der bei der Simulation von 45 Qubits auf dem Cori II Supercomputer zum Einsatz gekommen ist. Anschliessend wird das neue Konzept der *Emulation* von Quantenschaltkreisen eingeführt und gezeigt, dass dieser Ansatz es erlaubt, die Laufzeit gegenüber der Simulation um viele Grössenordnungen zu verkürzen. Dies gilt insbesondere für die Emulation von Quantenschaltkreisen, die mathematische Funktionen auswerten.

Der dritte und letzte Teil widmet sich der manuellen Optimierung von Quantenschaltkreisen. Dabei werden insbesondere Schaltkreise für Ganzzahl- und Fixpunktarithmetik berücksichtigt. Entwickelt wird ein neuer Additionsschaltkreis, der Speichererersparnisse ermöglicht, indem inaktive Qubits von anderen Teilen der Berechnung zeitweise übernommen werden. Diese Qubits können in einem beliebigen Zustand und insbesondere auch mit anderen Qubits des Systems verschränkt sein.

Zusätzlich wird diese Schaltkreiskonstruktion dazu verwendet, Shors Algorithmus zu implementieren und auf diese Weise einen asymptotischen Skalierungsvorteil gegenüber bisherigen speichereffizienten Implementationen zu erzielen. Schliesslich wird eine Bibliothek für Fixpunktarithmetik entwickelt. Diese erlaubt die Auswertung jener mathematischen Funktionen, denen man häufig in der Literatur zu Quantenalgorithmen begegnet. Dazu werden mehrere Polynome niedrigen Grades so kombiniert, dass diese Funktionen mit hoher Genauigkeit ausgewertet werden können, ohne dass dabei die Laufzeit oder die Speicheranforderungen des resultierenden Schaltkreises signifikant grösser würden als dies bei der Auswertung eines einzigen Polynoms niedrigen Grades der Fall ist.

Contents

Abstract iii

Zusammenfassung v

Contents vii

1 Introduction 1
- 1.1 Quantum computing . . . 1
- 1.2 Qubits and gates . . . 3
- 1.3 Programming quantum computers . . . 4
- 1.4 Classical simulation of quantum circuits . . . 6
- 1.5 Outline . . . 9

I Compilation and optimization of quantum programs 11

2 A software methodology for compiling quantum programs 15
- 2.1 Quantum programs . . . 15
- 2.2 A toolchain for quantum programming . . . 18
  - 2.2.1 Providing abstractions . . . 18
  - 2.2.2 Enabling performance . . . 21
  - 2.2.3 From logical operations to hardware . . . 22
  - 2.2.4 Software and hardware backends . . . 23
  - 2.2.5 Implementation: ProjectQ . . . 24

3 Using Hoare logic to optimize quantum programs 27
- 3.1 Quantum programs . . . 28
- 3.2 Hoare tiples and their use for optimization . . . 29
- 3.3 Compiler optimization via Hoare logic . . . 30
- 3.4 Practical example: Optimizing floating-point arithmetic . . . 33
- 3.5 Formalization and generalization . . . 37
  - 3.5.1 Formalization of our basic methodology . . . 38

3.5.2 Generalized optimization methodology 39
3.6 Implementation using ProjectQ and Z3 42
3.7 Results and comparison 44
3.8 Summary and future work 47

4 Reducing resource requirements by optimizing error tolerances 49
4.1 Compilation and approximation 50
4.2 Error-propagation in quantum circuits 52
4.3 Example application 56
4.4 Implementation and numerical results 58
4.5 Conclusion 62

II Improved simulation of quantum computers 63

5 0.5 petabyte simulation of a 45-qubit quantum circuit 67
5.1 Optimizations 70
5.1.1 Standard optimizations 70
5.1.2 Single-core 70
5.1.3 Single-node 72
5.1.4 Multi-node 73
5.1.5 Global gate specialization 74
5.1.6 Circuit optimizations: Gate scheduling and qubit mapping . 75
5.2 Implementation and results 79
5.2.1 Cori II 79
5.2.2 Edison 83
5.3 Conclusion 85

6 Emulation of quantum circuits 87
6.1 Quantum computer emulation 88
6.1.1 Arithmetic operations and mathematical functions 89
6.1.2 Quantum Fourier transform 90
6.1.3 Quantum phase estimation 90
6.1.4 Measurements 91
6.2 Performance results 92
6.2.1 Experimental setup 92
6.2.2 Arithmetic operations and mathematical functions 92
6.2.3 Quantum Fourier transform 93
6.2.4 Quantum phase estimation 94
6.2.5 Comparison against other simulators 96
6.3 Conclusion 98

III Quantum circuits for mathematical functions 99

7 Factoring using 2n+2 qubits with Toffoli based modular multiplication 103
- 7.1 Toffoli based in-place addition 105
  - 7.1.1 Serial implementation 107
  - 7.1.2 Runtime analysis of the serial implementation 109
  - 7.1.3 Parallel / low-depth version 109
- 7.2 Modular multiplication 110
- 7.3 Implementation and simulation results 110
- 7.4 Advantages of Toffoli circuits 113
  - 7.4.1 Single-qubit rotation gate synthesis 113
  - 7.4.2 Design for testability 113
- 7.5 Conclusion 114

8 Optimizing quantum circuits for arithmetic 115
- 8.1 Learning from classical libraries 116
- 8.2 Evaluation of piecewise polynomial approximations 117
- 8.3 Software stack module for piecewise smooth functions 120
- 8.4 Inverse square root 121
- 8.5 Arcsine 123
- 8.6 Conclusion 124

Implementation details 125
- 8.A Basic circuit building blocks for fixed-point arithmetic 125
- 8.B Resource estimates for polynomial evaluation 126
- 8.C (Inverse) Square root 128
  - 8.C.1 Reversible implementation 128
  - 8.C.2 Resource estimates 130
- 8.D Arcsine 131
  - 8.D.1 Reversible implementation 132
  - 8.D.2 Resource estimates 133
- 8.E Simulation results 134
  - 8.E.1 Piecewise polynomial approximation 134
  - 8.E.2 (Inverse) Square root 134
  - 8.E.3 Arcsine 135

9 Conclusion and outlook 139

List of publications 141

Bibliography 143

Acknowledgments **155**

Curriculum Vitae **157**

Chapter 1

Introduction

This chapter discusses the basics of quantum computing and concludes with an outline of the thesis.

1.1 Quantum computing

Quantum computing makes use of the laws of quantum mechanics to solve certain computational tasks asymptotically faster than classical computers. Examples for tasks at which quantum computers excel are factoring [13], function inversion [14], accelerating Markov chain based algorithms [15], and the simulation of quantum systems [16].

More precisely, *factoring* denotes the problem of finding natural numbers p and q for a given input number N such that $p \cdot q = N$. For an n-bit input number, Shor's algorithm solves this problem using $\mathcal{O}(n^3)$ quantum operations. This is a superpolynomial improvement over the best known classical algorithm, the number field-sieve [17]. Many of today's encryption schemes are based on factoring or similar problems because they are believed to be difficult to solve classically. As a result, the advent of quantum computers will require a switch to encryption schemes that are resilient to quantum attacks. Possible candidates are, for instance, lattice-based cryptography schemes such as *learning with errors* [18].

The problem of *function inversion* is as follows: Given an oracle which efficiently identifies a correct answer among M possibilities, this answer can be found using $\mathcal{O}(\sqrt{M})$ queries to the oracle using a quantum algorithm called Grover search. Classically, unstructured search requires $\Theta(M)$ queries to the oracle which makes this a quadratic advantage.

The third example—the acceleration of Markov chain based algorithms—can be seen as a generalization of Grover search. For certain problems, however, even exponential speedups are possible [19].

The final example is the *simulation of quantum systems* which, due to the fact that quantum computers themselves behave quantum mechanically, can be achieved using exponentially fewer resources than are required classically [16]. This may have a large impact on the development of medicine and new materials which is the reason that it is currently considered to be the most promising application of future quantum computers.

Given this list of applications, the question of how or *why* quantum computers solve these problems more efficiently immediately arises. A first step toward answering this question is to understand the concept of *quantum superposition*.

Classical computers compute by manipulating bits, which is the most basic unit of memory and which can be represented as a switch that is either off (zero) or on (one). In contrast, quantum computers operate on quantum bits or *qubits*. Similar to a classical bit, a qubit can also be in one of two states, which we denote by $|0\rangle$ (*ket zero*) and $|1\rangle$ (*ket one*). The state of the qubit $|q\rangle$ can then be described as

$$|q\rangle = \alpha_0 |0\rangle + \alpha_1 |1\rangle \ ,$$

where α_0 and α_1 are complex numbers such that $|\alpha_0|^2 + |\alpha_1|^2 = 1$. Thus, if either of these two coefficients is 0 and the other is 1, then $|q\rangle = |0\rangle$ or $|q\rangle = |1\rangle$, just like a classical bit. Consider a qubit $|q\rangle$ in one of the states

$$|+\rangle := \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle) \text{ or } |-\rangle := \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle) \ .$$

The qubit is somehow in both states 0 and 1 *simultaneously*. If we were to look at the qubit, i.e., *measure* its value, we would find it in state $|0\rangle$ with probability $|\alpha_0|^2 = 0.5$, and in $|1\rangle$ with probability $|\alpha_1|^2 = 0.5$. One may of course argue that this is a classical bit, but one that is flipped with a certain probability, making the outcome of a measurement probabilistic.

For an example describing a multi-qubit system, consider a system of two qubits and denote its state by $|q_1 q_0\rangle$. The system may be in a superposition of all possible classical states of these two qubits,

$$\begin{aligned} |q_1 q_0\rangle &= \alpha_{00} |00\rangle + \alpha_{01} |01\rangle + \alpha_{10} |10\rangle + \alpha_{11} |11\rangle \\ &= \alpha_0 |0\rangle + \alpha_1 |1\rangle + \alpha_2 |2\rangle + \alpha_3 |3\rangle \ , \end{aligned}$$

where we have interpreted the two bits as binary numbers in the second line, e.g., $10_2 = 2$. The normalization condition is again $\sum_i |\alpha_i|^2 = 1$.

To illustrate what *quantum entanglement* is, let $\alpha_0 = \alpha_3 = \frac{1}{\sqrt{2}}$ and $\alpha_1 = \alpha_2 = 0$. When measuring the first qubit, the probability of finding it in $|0\rangle$ is $p = 1/2$, as before. This time, however, a measurement of one of the qubits also reveals information about the other qubit since in this superposition, the qubits are either both zero, or both one. Once the state of one of the qubits has been measured, the measurement outcome of the other qubit is already fixed. However, before doing so, either qubit has a 50% chance of ending up in $|0\rangle$ or $|1\rangle$. The phenomenon of quantum entanglement plays a crucial role in quantum information theory.

The final difference between a quantum mechanical description and a classical probabilistic description is that classical probabilities are real numbers between 0 and 1, whereas the *quantum probability amplitudes* α_i are complex numbers which, in particular, may be negative. This, in turn, leads to a third phenomenon called *quantum interference*. When combined, superposition, entanglement and interference allow for quantum algorithms where the probability amplitudes of wrong answers cancel each other while boosting the probability amplitude of the sought answer. At an abstract level, this is what happens in quantum algorithms such as Grover search [14].

1.2 Qubits and gates

This section introduces general n-qubit systems more formally, together with operations, so-called *quantum gates*, that may be applied to these systems in order to carry out computations.

The state of a quantum system consisting of n qubits may be described by assigning a complex probability amplitude to every possible assignment of n classical bits. The set of all such assignments is also called the *computational basis*. The state $|\psi\rangle$ of an n-qubit quantum computer is then

$$|\psi\rangle = \sum_{i=0}^{2^n-1} \alpha_{i_{n-1}i_{n-2}\cdots i_0} |i_{n-1}i_{n-2}\cdots i_0\rangle = \sum_{i=0}^{2^n-1} \alpha_i |i\rangle \ ,$$

where the n-bit string $i_{n-1}i_{n-2}\cdots i_0$ can be interpreted as the binary representation of the corresponding integer i in order to arrive at a more compact notation. The normalization condition is $\sum_i |\alpha_i|^2 = 1$.

Because the computational basis is orthonormal, we can identify each basis state $|i\rangle$ with the unit vector $e_i \in \mathbb{C}^{2^n}$ which has entries

$$(e_i)_j = \delta_{ij} \ ,$$

where δ_{ij} is the Kronecker delta which is 1 if $i = j$ and 0 otherwise. As a result, $|\psi\rangle$ can be identified with a column vector $\vec{\alpha}$ with entries α_i and its conjugate transpose $|\psi\rangle^\dagger$ with $\vec{\alpha}^\dagger$. The normalization condition can then be written as $\vec{\alpha}^\dagger\alpha = 1$ or, using Dirac notation with $\langle\psi| := |\psi\rangle^\dagger$, equivalently as $\langle\psi|\psi\rangle = 1$.

Let $U \in \mathbb{C}^{2^n \times 2^n}$ be an operation (or quantum gate) acting on n qubits. The normalization condition on the new state $|\psi'\rangle := U|\psi\rangle$ is

$$\langle\psi'|\psi'\rangle = \langle\psi| U^\dagger U |\psi\rangle = 1 \ ,$$

which must hold for all input states $|\psi\rangle$. Equivalently, U should be norm preserving and, by the polarization identity, this is the case if and only if U preserves inner-products [20, p. 33]. In other words, U should be unitary, meaning that

$$U^\dagger U = UU^\dagger = \mathbb{1}_{2^n \times 2^n} \ .$$

In particular, operations being unitary implies that they must be *reversible*: Any unitary operation U can be undone by applying $U^\dagger$, since $U^{-1}U = U^\dagger U = \mathbb{1}$.

Such unitary operations may also be applied *controlled* on another qubit, meaning that they are applied if the control qubit is 1. Formally, the controlled version of U is

$$U^c := |0\rangle\langle 0| \otimes \mathbb{1} + |1\rangle\langle 1| \otimes U \ ,$$

where $|c\rangle\langle c|$ is the projector onto the subspace in which the control qubit has the value $c \in \{0, 1\}$ and $\otimes$ denotes the tensor product. Since the control qubit may be in a superposition, the state after applying U^c is in a superposition of having and not having applied U.

To see that by applying unitary operators one can, in particular, carry out any classical computation, note that any such computation can be made reversible [21] if the input is kept along with the result. Intermediate results can be *uncomputed* [21] by copying out the final result before running the entire computation in reverse. As an example, consider evaluating $r(x)$ which produces $g(x)$ as an intermediate result, where the input $|x\rangle$ may be in a superposition:

$$\begin{aligned} |x\rangle|0\rangle|0\rangle|0\rangle &\mapsto |x\rangle|g(x)\rangle|r(x)\rangle|0\rangle \\ &\mapsto |x\rangle|g(x)\rangle|r(x)\rangle|r(x)\rangle \\ &\mapsto |x\rangle|0\rangle|0\rangle|r(x)\rangle \end{aligned}$$

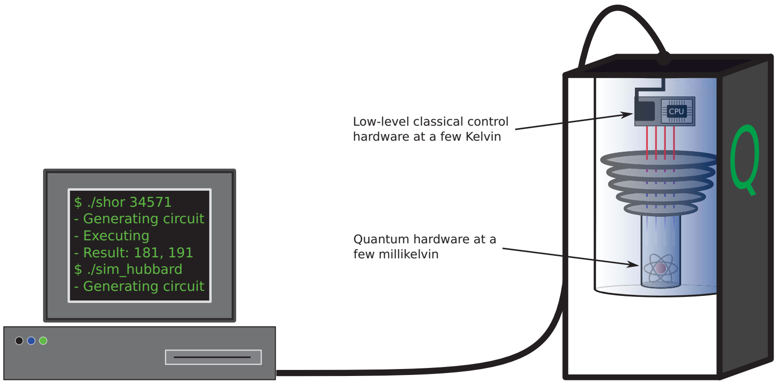
Therefore, *uncomputation* allows to achieve the reversible mapping

$$|x\rangle|0\rangle \mapsto |x\rangle|r(x)\rangle$$

efficiently for any classical function $r(x)$, albeit with an overhead in both space and time. Due to this overhead, the implementation details of classical functions may have significant ramifications on the run time of quantum programs. We thus consider automatic optimization of such circuits in the first part of this thesis and provide hand-optimized implementations in the third part.

1.3 Programming quantum computers

In this thesis, we adopt an abstract, hardware-agnostic view of a quantum computer. This is important especially because there are several competing qubit



```
$ ./shor 34571
- Generating circuit
- Executing
- Result: 181, 191
$ ./sim_hubbard
- Generating circuit
```

Low-level classical control hardware at a few Kelvin
CPU
Q
Quantum hardware at a few millikelvin

Figure 1.1: Abstract view of a quantum computer which consists of the actual quantum processing unit (QPU) and a classical computer which controls the quantum hardware by sending instructions to be executed by the QPU. Certain architectures require very low temperatures to operate, but this does not hold in general. (reprint from Ref. [1])

technologies that are being investigated in various hardware laboratories, including trapped ion [22] and superconducting qubit systems [23].

Specifically, the machine model we consider is a combination of a classical computer and a quantum processing unit (QPU). The classical computer produces a sequence of quantum instructions which are then executed on the QPU. In this setting, the QPU is very similar to today's classical accelerators such as graphics processing units (GPU), field-programmable gate arrays (FPGA), or application-specific integrated circuits (ASIC) [1, 24]. The fact that quantum computing allows to speed up only certain tasks strengthens this analogy further.

While for certain technologies, additional classical hardware at lower levels may be employed in order to deal with issues involving, e.g., latency or heat dissipation from control lines, this abstract view remains valid: A classical processor passes commands to the QPU which, ultimately, responds with measurement results to be interpreted by a classical processor. For a depiction of this model, see Fig. 1.1.

In each clock cycle, the state of the QPU evolves according to the quantum instructions dictated by the classical host. The overall action during each clock cycle can be described by a $2^n \times 2^n$-dimensional unitary matrix acting on the entire 2^n-dimensional state vector of the system. However, many-body interactions are typically not supported natively by the hardware due to the difficulties in engineering and calibrating such interactions. We thus require that it is known how to decompose these matrices in terms of the target gate set, which typically consist of single-qubit gates and at least one two-qubit gate.

It is known that any unitary operation on n qubits can be written in terms of single- and two-qubit gates [25, 26]. However, it is necessary that a decomposition of U in terms of the target gate set is known because for large n, it would be infeasible to store U, let alone to find an efficient decomposition. For small n, however, such matrices can still be stored and many procedures to decompose small matrices in terms of various gate sets are known [26, 27, 28].

The resulting sequence of one- and two-qubit gates can be depicted as a *quantum circuit*. In such a circuit, each qubit is represented by a horizontal line and operations are drawn as boxes or other symbols on these lines, with time advancing from left to right. For an example of a quantum circuit and for further details, see chapter 2. For a list of often-encountered single- and two-qubit gates, including their matrix- and circuit-representations, see Table 1.1.

The next section is concerned with the simulation of quantum computers on classical hardware. In particular, it is shown how to convert a given k-qubit gate (with $k < n$) to the full $2^n \times 2^n$-sized unitary matrix which can then be applied to the 2^n-dimensional state vector.

1.4 Classical simulation of quantum circuits

Current quantum computers feature tens of qubits with fairly high error rates, making it impossible to successfully execute any quantum program beyond the most basic examples. Despite this, it is possible to implement and test quantum algorithms featuring up to 45 qubits using classical (super)computers [4]. While there are several approaches to the simulation of quantum circuits [29, 30, 31], we discuss the approach which is best-suited for the (full) simulation of circuits featuring large depths.

To this end, the quantum state of an n-qubit system can be described by 2^n complex numbers $\alpha_0, ..., \alpha_{2^n-1}$, where each of these α_i is the probability amplitude corresponding to the computational basis state

$$|i\rangle := |i_{n-1}, ..., i_0\rangle := |i_{n-1}\rangle \otimes |i_{n-2}\rangle \otimes \cdots \otimes |i_0\rangle \ ,$$

and i_k denotes the k-th bit of the integer k starting from the least significant bit (LSB). An n-qubit quantum circuit is a sequence of M quantum operations $(g_1, ..., g_M)$, where each operation g_m can be described by a unitary matrix which acts on one or multiple of these n qubits.

Let g be a k-qubit operation. The corresponding unitary matrix U^g is

$$\begin{pmatrix} U^g_{0,0} & U^g_{0,1} & \cdots & U^g_{0,2^k-1} \\ U^g_{1,0} & U^g_{1,1} & \cdots & U^g_{1,2^k-1} \\ \vdots & \vdots & \ddots & \vdots \\ U^g_{2^k-1,0} & U^g_{2^k-1,1} & \cdots & U^g_{2^k-1,2^k-1} \end{pmatrix} \in \mathbb{C}^{2^k \times 2^k} \ ,$$

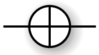
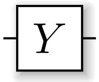
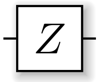
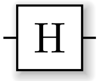
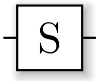
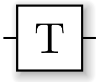
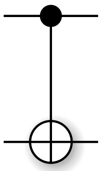
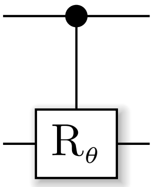
Gate	Matrix	Symbol
NOT or X gate	$\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$	
Y gate	$\begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix}$	
Z gate	$\begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}$	
Hadamard gate	$\frac{1}{\sqrt{2}}\begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}$	
S gate	$\begin{pmatrix} 1 & 0 \\ 0 & i \end{pmatrix}$	
T gate	$\begin{pmatrix} 1 & 0 \\ 0 & e^{i\pi/4} \end{pmatrix}$	
Rotation-Z gate	$\begin{pmatrix} e^{-i\theta/2} & 0 \\ 0 & e^{i\theta/2} \end{pmatrix}$	
Controlled NOT (CNOT)	$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix}$	
Conditional phase-shift (CR)	$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & e^{i\theta} \end{pmatrix}$	

Table 1.1: Standard quantum gates with their corresponding matrices and symbols from Ref. [5].

where $U^g_{j,i}$ can be interpreted as the transitioning amplitude from the computational basis state $|i\rangle$ to $|j\rangle$. This means that an incoming computational basis state $|i\rangle$ gets a contribution of $U^g_{j,i}$ in the output vector entry corresponding to $|j\rangle$. U^g can thus also be written as

$$U^g = \sum_{i,j=0}^{2^k-1} U^g_{i,j} \left|j\right\rangle \left\langle i\right| \ .$$

In quantum computing, gates are specified as k-qubit matrices, even if the entire quantum system features n qubits, where n is typically much larger than k. However, it is known that an operation acting on the n-qubit state $|\psi\rangle$ can be written as a $2^n \times 2^n$ unitary matrix, and that the output state can be obtained by multiplying the amplitude vector of $|\psi\rangle$ by this unitary matrix. The construction of this larger unitary $U \in \mathbb{C}^{2^n \times 2^n}$ from a given $U^g \in \mathbb{C}^{2^k \times 2^k}$ can be performed as follows: First, write

$$U = \sum_{\hat{i},\hat{j}=0}^{2^n-1} U_{\hat{j},\hat{i}} \, |\hat{j}\rangle \langle\hat{i}| = \sum_{l,m=0}^{2^{n-k}-1} \sum_{i,j=0}^{2^k-1} U_{m|j,l|i} \, |(m|j)\rangle \langle(l|i)| \ ,$$

where $|\hat{i}\rangle, |\hat{j}\rangle$ are n-qubit computational basis states and $l|i$ denotes merging of the two integers l and j as bit-strings in the order of the qubits in the state vector. I.e., the bit-pattern of $l|i$ is given by (1) the k qubits upon which U^g acts are in state i and (2) the remaining $n-k$ qubits are in state l.

Since U may act nontrivially only on the k qubits to which U^g is applied,

$$U_{m|j,l|i} = 0 \text{ if } l \neq m \ .$$

As a result,

$$U = \sum_{m=0}^{2^{n-k}-1} \sum_{i,j=0}^{2^k-1} U_{m|j,m|i} \, |(m|j)\rangle \langle(m|i)| = \sum_{m=0}^{2^{n-k}-1} \sum_{i,j=0}^{2^k-1} U^g_{j,i} \, |(m|j)\rangle \langle(m|i)| \, .$$

Therefore, to simulate the evolution of the state vector $\overline{\alpha} := (\alpha_0, ..., \alpha_{2^n-1})^T$ when applying the k-qubit gate g which is specified via the complex $2^k \times 2^k$ matrix U^g one proceeds as follows: For each fixed $m = 0, 1, ..., 2^{n-k}-1$, perform a 2^k-dimensional matrix-vector multiplication $\overline{\beta^m_{new}} = U_g \overline{\beta^m}$, where $\overline{\beta^m}$ consists of those values from $\overline{\alpha}$ which correspond to computational basis states that have the inactive qubits in state $|m\rangle$, meaning that the first inactive bit is $m_0 \in \{0,1\}$, the second is $m_1 \in \{0,1\}$, and so on. These 2^k values are ordered in the usual way such that $\overline{\beta^m}_j$ corresponds to the target qubits being in the computational basis state j. After the 2^k-dimensional matrix-vector multiplication has been executed, the 2^k values from $\overline{\alpha}$ are the replaced by their new values $\overline{\beta^m_{new}}$ and m is increased by one. This is then repeated until the last iteration with $m = 2^{n-k}-1$ is complete.

In summary, an n-qubit quantum circuit consisting of a sequence of M gates $(g_1, g_2, ..., g_M)$ can be simulated on a classical computer gate by gate, where each k_i-qubit gate g_i involves 2^{n-k_i} matrix-vector multiplications of dimension 2^{k_i}. These 2^{k_i}-dimensional subvectors are extracted from the overall state vector of dimension 2^n and, after successful multiplication by the gate matrix, the output is stored back into the overall state vector.

1.5 Outline

Thinking about quantum programs in terms of circuits or even matrices makes it difficult to see the similarity between quantum computers and, e.g., today's GPUs. In the following chapter, we thus introduce a methodology for programming and compiling quantum computers. Subsequently, in chapters 3 and 4, we present two additional methods for quantum program optimization. The first method uses the *Hoare triples* [32] of all subroutines in order to identify optimization opportunities. In contrast, the second method does not operate at the circuit level. Instead it aims to optimize the tolerances with which the compiler approximates subroutines that have no exact representation in terms of the target gate set. The tolerances are optimized in order to reduce the number of elementary operations while satisfying a user-specified error bound.

As a consequence, it must be known how the cost of subroutines behaves as a function of the target accuracy. While upper bounds on these costs may exist, these have been found to be loose by several orders of magnitude in certain cases [33, 34] by carrying out numerical studies. The quality of the resulting numerical estimates can be improved by carrying out larger simulations. To this end, we improve classical techniques for simulating quantum computers in the second part of this thesis. This includes a high-performance implementation of a full state simulator which is presented in chapter 5. It allows speedups of up to 14× when compared to state-of-the-art simulations of random quantum circuits, which is the worst-case scenario for our implementation. Furthermore, in a setting where a high-level description of the quantum circuit is available, a variety of classical shortcuts can be employed in a process called *emulation*. We introduce the concept of a quantum circuit emulator in chapter 6 and compare it to simulation in terms of run time.

When combined, the above methods allow to carry out a wide variety of quantum program optimizations automatically. Similar to classical high-performance computing, however, manual optimizations have been shown to greatly improve upon the code generated by design automation and optimizing compilers [8]. In the third and final part of this thesis, we design and optimize quantum circuits to evaluate mathematical functions. We develop a new addition circuit in chapter 7 which makes use of qubits that may be in an arbitrary state (and possibly entangled with other qubits in the system) in order to save qubits without resorting to costly rotation gates [35]. We then use our construction to implement Shor's algorithm for factoring with an asymptotic reduction in run time compared to previous space-efficient implementations. In chapter 8, we develop a parallel polynomial evaluation scheme which allows to approximate piecewise smooth functions at a cost that is similar to evaluating a single low-degree polynomial. We then use this scheme to provide gate estimates for several mathematical functions that occur frequently in the quantum algorithm literature.

Part I

Compilation and optimization of quantum programs

Introduction to Part I

The first part of this thesis is concerned with the compilation and optimization of quantum programs. In the first chapter of this part, which is a modified version of Ref. [1], we propose a software methodology for compiling quantum programs. We present the tools and layers of abstraction necessary for quantum software development and discuss backends which can be used before large-scale quantum hardware is available. Our methodology has since been implemented and released as ProjectQ [9].

Following this overview of our quantum programming toolchain, we focus on automatic optimization of quantum programs. In the second chapter, which was published as Ref. [2], we develop a new optimization methodology. In contrast to previous work, it is not inspired by or related to optimizations from the realm of classical compilers such as constant-folding or common subexpression elimination. Instead, our methodology exploits the structure that is present in superposition states if subsystems (sets of qubits) are entangled. To this end, it employs the Hoare triples of all invoked subroutines in order to gather information about said structure. By combining this information with conditions for sequences of operations to be trivial, our method is able to exploit optimization opportunities which could not be identified as such by previous methods.

The final chapter of this part, which was published as Ref. [3], discusses an entirely different type of optimization. Instead of transforming a given circuit to an equivalent circuit of lower cost, we consider the compilation step before such optimizations are applied. In order to generate the circuit for a specific target system, compilers must choose tolerances for approximations that must be made during the compilation process. In addition to guaranteeing a given overall accuracy, compilers may choose these tolerances to also reduce resource requirements. We detail the workings of such a compiler module and discuss the benefits and tradeoffs involved by compiling a quantum program to simulate a transverse-field Ising model.

Chapter 2

A software methodology for compiling quantum programs

As a result of the rapid progress in engineering prototypes for quantum computing devices over recent years, the benefits of a supporting software stack have increased. Not only does such a stack allow for a more efficient software development process, but it additionally enables more efficient hardware/software co-design. In particular, a quantum software stack enables rapid prototyping via repeated compilation of chosen benchmarks for different hardware constraints. In light of this, several distinct software efforts have emerged, e.g., Refs. [36, 37].

In this chapter, which is based on Ref. [1], we introduce a software methodology for compiling quantum programs which, in contrast to previous efforts, employs concepts from classical high performance computing. In particular, we introduce a quantum analog of pragma-statements, which allow the compiler to generate code that is identical to hand-optimized code, despite additional levels of abstraction. Furthermore, our methodology allows for several intermediate gate sets which enables even peephole optimizers to perform fairly powerful optimizations. In addition to the quantum programming language and the compiler, we discuss potential backends which can be used before large-scale quantum hardware is available. Ultimately, the software methodology introduced in this chapter led to the development of the ProjectQ software framework [9].

2.1 Quantum programs

A quantum program consists of classical and quantum instructions. Some of the classical instructions are to be executed on the host computer, i.e., the classical computer which runs the program, and some are to be executed by the lower-level control hardware. Naturally, all quantum instructions in the code are sent to the

QPU for processing. Upon measuring a qubit, the QPU exposes the outcome to the classical controller which, in turn, may use this information to determine the next sequence of quantum instructions that is required to advance the computation.

Examples of such hybrid quantum/classical programs are quantum teleportation [38], the variational quantum eigensolver [39], and the quantum approximate optimization algorithm [40]. In each of these algorithms, feedback from the quantum processor is used in order to determine the next steps to advance the overall computation. The simplest example of the above is teleportation, which is discussed in the following box.

Example (Quantum Teleportation)

Alice has a qubit in state $|\psi\rangle := \alpha\,|0\rangle + \beta\,|1\rangle$ which she would like to send to Bob who is currently overseas. Luckily, Alice and Bob decided to share a *Bell pair* during their last encounter, which means that they each have one qubit of

$$|\Psi_{BP}\rangle := \frac{1}{\sqrt{2}}(|0\rangle_A\,|0\rangle_B + |1\rangle_A\,|1\rangle_B)\ .$$

Alice can entangle her qubit in $|\psi\rangle$ with her share of the Bell pair using a controlled NOT gate. All that is left for Alice to do now is to measure her two qubits, i.e., her share of the Bell pair and her qubit which started out in $|\psi\rangle$ in the Z- and X-basis, respectively, and to send the measurement outcomes which are just two classical bits to Bob. Conditional on these bits, Bob can apply a correction such that his share of the Bell pair ends up in $|\psi\rangle$, as desired. In code, Alice runs

```
CNOT | (psi, bp_A)
H | psi
Measure | (psi, bp_A)
# send measurement outcomes to Bob
msg = bool(psi), bool(bp_A)
send_to(Bob, msg)
```

Bob receives the message `msg` and performs two conditional corrections

```
if msg[1]:
  X | bp_B
if msg[0]:
  Z | bp_B
```

From these two code segments, it is easy to see that there is no quantum channel required between Alice and Bob—the exchange of a quantum state is performed by using the entanglement between `bp_A` and `bp_B`, in addition

to a classical channel, which is used to transmit merely two classical bits of information.
Note that this protocol does not violate the No-Cloning Theorem [41] since Alice measures her qubit and thus collapses $|\psi\rangle$ onto $|+\rangle$ or $|-\rangle$. Furthermore, while the entanglement causes changes performed on one qubit of the Bell pair to appear instantaneously at the location of its entanglement partner, this alone does not constitute transmission of information. Without the corrections by Bob, there are only random correlations at a distance. As a consequence, there occurs no faster-than-light transmission of information.

While code was used to illustrate the details of the example above, quantum circuits are still the primary way of communicating algorithm constructions in the quantum computing research community. In a quantum circuit diagram, qubits are drawn as horizontal lines and operations are drawn on these lines with time advancing from left to right. Once measured, qubits are projected onto the measurement outcome—either $|0\rangle$ or $|1\rangle$ and, as a result, they become classical bits. In order to make this distinction clearly visible in circuit diagrams, classical bits are drawn as double lines. As an example, the quantum circuit for teleportation is depicted in Fig. 2.1.

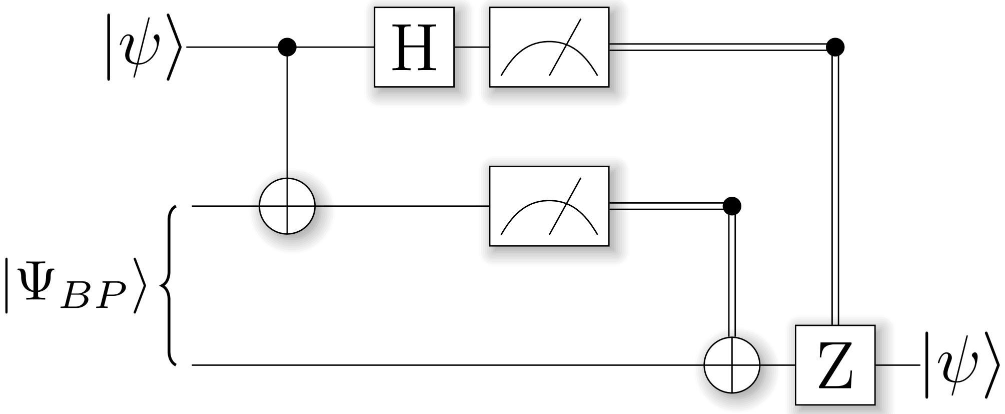


Figure 2.1: Quantum circuit for the teleportation example. Alice entangles her qubit in $|\psi\rangle$ with her share of the Bell pair $|\Psi_{BP}\rangle$ using a CNOT. She then measures her two qubits in the X- and Z-basis and sends the outcomes to Bob (the classical bits are represented by double lines) who applies the appropriate correction (X- and/or Z-gate).

Thus far, we have merely described quantum programs at the level of elementary quantum gates such as Hadamard, Pauli-{X,Y,Z}, and CNOT gates. In the literature, this level of programming language is often labeled QASM which is short for *quantum assembly*. It is worth noting, however, that QASM languages such as OpenQASM [42] offer even less abstraction than their classical counterparts. For instance, QASM languages do not feature the reversible analog of an AND gate as native instructions, let alone additions or multiplications. These have to be constructed from single- and two-qubit gates [6, 7, 43, 35, 44] that are typically available in these low-level languages, an example being the gate set

$$\{H, CNOT, T\}\ .$$

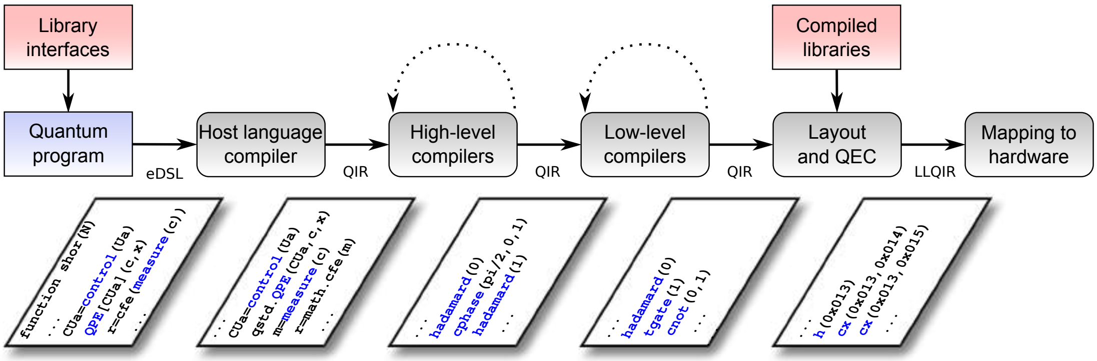
It is clear that means of abstraction from these low-level instructions in combination with library implementations of commonly used language constructs are necessary for serious software development. Furthermore, the compiler, which ultimately translates from the abstract language to such a low-level representation, should be able to optimize the code during this translation process. In the following sections, we present further features and discuss potential implementations of a software methodology for compiling quantum programs.

2.2 A toolchain for quantum programming

While programming at the hardware-level has benefits in terms of execution speed or size of the resulting quantum program, it is inefficient to program at such a low level of abstraction. It is thus crucial to develop a software stack which provides the necessary abstraction but, at the same time, does not introduce too much overhead in the resulting code. Because similar issues arise in classical high-performance computing, we may borrow ideas from the classical domain in order to address them. Additionally, new programming language and/or software stack features are required for constructs that are specific to quantum computing.

2.2.1 Providing abstractions

In a programming language, themes that occur frequently in programs are usually included in the standard library of the language. If useful for certain often-reoccurring themes, these may even be added to the language itself, be it to boost performance or to reduce compilation time. Having such standard functionality available as modules facilitates code maintenance, reduces development times, and makes the code less prone to error. Following what has proven effective in classical computing, we apply the same techniques to develop the programming language of our methodology. Furthermore, the design of the software toolchain itself should also allow for code-reuse and it should be quickly adaptable to new or upcoming quantum hardware. As such, a modular design which is open to extension is required. In order to optimally reuse existing libraries and compilation functionality



```
function shor(N)
...
CUa=control(Ua)
QPE[CUa](c,x)
r=cfe(measure(c))
...
```

```
...
CUa=control(Ua)
qstd.QPE(CUa,c,x)
m=measure(c)
r=math.cfe(m)
...
```

```
...
hadamard(0)
cphase(pi/2,0,1)
hadamard(1)
...
```

```
...
hadamard(0)
tgate(1)
cnot(0,1)
...
```

```
...
h(0x013)
cx(0x013,0x014)
cx(0x013,0x015)
...
```

Figure 2.2: High-level view of our proposed methodology. (reprint from [1])

for classical code, we envision a quantum programming language that is embedded in an existing classical host language. This allows for short development times of the framework and lets software developers use their standard tools. As an additional benefit, interfaces to hardware can be deployed very quickly, especially if the chosen host language is used in hardware laboratories, as is the case for Python or *C++*.

From a high-level perspective, our methodology for compilation operates in five distinct phases, see Fig. 2.2: In a first phase, the host language is interpreted or compiled. Then, a series of high- and low-level compilers translates and optimizes the quantum intermediate representation (QIR) of the program, taking into account some specifics of the target hardware such as the target gate set. Finally, the program is made fault-tolerant by employing quantum error correction (QEC) and the necessary operations are sent to the hardware via low-level QIR (LLQIR).

The quantum program consists of *logical* operations acting on *logical* quantum bits. This means that programmers need not be bothered by qubit errors or the specifics of the underlying architecture, e.g., the connectivity of the physical qubits in the device. The translation from the logical description of the algorithm to a fault-tolerant implementation for the target hardware is performed automatically by the series of compilers depicted in Fig. 2.2. In addition, programmers can rely on library implementations in order to speed up their development process. The importance of having libraries for quantum computing becomes apparent when implementing quantum algorithms which employ subroutines that, e.g., compute classical functions on a superposition of inputs. Exponentials, $\sin(x)$, and other trigonometric functions are widely available in classical programming languages and also find their application in various quantum algorithms [45, 46, 47]. However, efficient implementations amenable to quantum hardware are rare and sometimes even nonexistent. To remedy this inefficiency in quantum programming, we propose various libraries that should be available in any language for quantum

quantum	**system**	**user**
– qutypes – qugates – qucontrol – quarithmetic – qumath – qualg	host language standard library	user defined librares

Table 2.1: Library components in our software toolchain.

programming. In the third part of this thesis, we address this specific problem in more detail by designing efficient implementations of mathematical functions for integer and fixed-point arithmetic [6, 7].

For our toolchain, we envision the library components listed in Table 2.1. As previously mentioned, we reuse existing tools that are available in and for the host language in which we embed the quantum programming language. This includes the standard library of the host language. Furthermore, libraries should be extensible and user contributions should be made available to other quantum software developers. Such user-contributed libraries fall under the “user” category. For our methodology, the “quantum” libraries are the most relevant and thus we elaborate on the details of its components.

qutypes library. The first quantum library contains a collection of types for quantum computing and is called `qutypes`. The most basic type is a (logical) `qubit` which can be allocated, operated upon, and deallocated. As an intermediate type between the `qubit` and more abstract types, there is the quantum register type `qureg`, which is a list of `qubit` objects. Such a quantum register can be interpreted in various ways, the simplest being an integer which we label `quint`. Continuing this analogy to classical types, we also envision quantum versions of fixed- and floating-point numbers, `qufixed` and `qufloat`. For all of these types, operator overloads allow easy manipulation and conversion between representations.

qugates library. Gates are operations on qubits and the `qugates` library provides programmers with a collection of the most common ones. In particular, this library contains technology-independent gates as well as operations that are very specific to certain architectures, thereby providing the means to optimize library functions hardware-specifically.

qucontrol library. One of the most-encountered themes in quantum computing is that operations or subroutines are executed controlled on other qubits. The `qucontrol` library provides the means to annotate quantum operations accordingly. In particular, the library supports quantum control flow statements which describe temporary basis changes—so-called compute/uncompute sections—, loops, and conditional execution which is the quantum analog of an if-then-else block. While such annotations primarily serve the purpose of reducing code development times, they can also be used to improve optimization capabilities of the compiler. This will be discussed in more detail in Sec. 2.2.2.

quarithmetic and qumath libraries. Similar to classical high-performance computing, arithmetic functions must be optimized for optimal utilization of the capabilities of the underlying hardware. The `quarithmetic` and `qumath` libraries provide optimized low- and high-level mathematical functions that are tailored to be executed on a quantum computer. In particular, these implementations are reversible and optimized for low quantum resource requirements. The `qumath` library employs the lower-level functions from `quarithmetic` such as additions and multiplications, which may be optimized specifically for the target hardware. Together, these two libraries provide support for mathematical functions that are common in the quantum algorithm literature.

qualg library. The quantum algorithm library contains implementations of known algorithms which may then be used as subroutines in order to develop new ones. Typical examples of subroutines which occur frequently throughout the algorithm literature are quantum phase estimation (QPE), quantum Fourier transform (QFT), and amplitude amplification.

2.2.2 Enabling performance

The main downside of introducing abstractions in computing is the possible degradation of code performance. As a remedy, high-level languages may expose all low-level and even hardware-specific quantum instructions, similar to inline assembly or Intel® Intrinsics in classical high-level languages such as *C* or *C++*. However, this solution is not sufficient for practical purposes as it provides no benefits over QASM languages.

In order to enable highly-efficient code even in the presence of abstractions, our methodology makes use of code annotations such as the control flow statements from the `qucontrol` library in order to optimize the code at a more global scale. The compute/uncompute annotation in combination with a controlled execution, for example, allows to drastically reduce the number of controlled quantum oper-

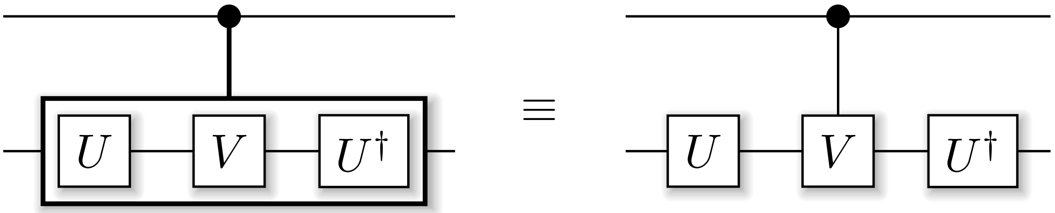


Figure 2.3: Controlled execution of a subroutine which contains a compute and uncompute section can be optimized by only controlling gates which are not part of the basis change. Thus, the basis changes U and $U^\dagger$ require no additional controls. (reprint from [1])

ations. The compute/uncompute annotation describes a temporary basis change with operations in between. The basis change occurs with a unitary U, upon which some action V is performed before the basis change is undone via $U^\dagger$ (the inverse of U). The circuit for executing such a section controlled on one or several qubits can be optimized as shown in Fig. 2.3 by employing appropriate code annotations. Namely, only the V-operation must be controlled on the control qubit since, if the control qubit is zero, $U^\dagger U = \mathbb{1}$ is trivial. If the control qubit is one, the entire sequence $U^\dagger V U$ is applied, as desired. Similar optimizations are possible for quantum if-then-else and loop statements [1]. In particular, controlled adjoint, which is a special case of an if-then-else block, can be optimized [48].

In addition to code annotations, we use several intermediate gate sets with optimization passes at each level. This allows to merge or cancel even high-level gates acting on the same qubits. A compiler which translates the program directly to low-level gates before optimizing would most-likely miss many optimization opportunities, especially if rotations have already been approximated by a rotation synthesis algorithm such as the one in Ref. [27]. The lower the level of gates, the harder it becomes to reconstruct the origin of a given quantum operation. As a result, even identifying and merging two successive additions becomes nontrivial at best.

2.2.3 From logical operations to hardware

After passing through a series of optimizers and translation modules, the high-level quantum program has been successfully converted to a sequence of low-level quantum instructions. However, this sequence still consists of *logical* operations which act on perfect, logical qubits with arbitrary connectivity.

In order to bridge this gap between logical circuit and actual hardware, the lowest-level compiler must map the quantum circuit to the connectivity graph of the target hardware. To this end, it must keep track of the physical locations of all qubits and make them adjacent on this graph via swap gates or teleportation, before they can interact via a multi-qubit gate such as the controlled NOT.

Additionally, if the hardware qubits have noise rates that are too large for a successful execution of the quantum program in question, the circuit has to be transformed to a fault-tolerant implementation employing a quantum error correction protocol such as the surface code [49]. While there exist quantum analogs of classical repetition codes such as the 9-qubit Shor code [50], the workings of such protocols are different in quantum computing. The main difference is due to the quantum No-Cloning Theorem [51], which prevents simple error correction by having multiple copies of all states available. For an introduction to the topic of quantum error correction and more details, we refer to Ref. [41].

Currently, quantum hardware is still too limited to run large-scale quantum error correction protocols. However, it is of great practical interest to have detailed overhead estimates and circuit-level implementations of various protocols available in order to determine more accurate and practically relevant error thresholds.

2.2.4 Software and hardware backends

Once a quantum program has been implemented successfully, it can be compiled and then run on a quantum computer. The caveat being that large-scale quantum hardware which would enable the execution of nontrivial algorithms does not yet exist. In particular, at the time of writing, quantum hardware is not yet able to outperform classical supercomputers at a well-defined computational task such as the ones in Refs. [52, 53, 54]. In spite of this, implementations are still of tremendous use even if they cannot be run on the appropriate hardware yet. Our methodology thus provides a wide variety of different backends, each with its own specific use cases.

Hardware backends. The primary purpose of a toolchain for quantum computing the compilation of quantum programs for actual quantum hardware. There are several competing technologies which all are at different stages of the scaling process. The support for multiple technologies enables comparisons between different qubit technologies such as Ref. [55], as well as more efficient hardware/software co-design.

Classical simulators. In the absence of large-scale quantum computers, it is important to have the ability to test algorithms and implementations thereof at small scales. In addition to debugging, simulation results from small problem sizes can be extrapolated in order to estimate the performance of future large-scale quantum computing devices. This is important especially because theoretic bounds are often off by several orders of magnitude and simulation is the only tool that allows identifying such discrepancies between theory and practice [33, 34]. In chapter 5, the implementation of a high-performance quantum circuit simulator is discussed in great detail.

Classical emulators. Starting from a high-level description of a quantum algorithm, it is typically compiled to single- and two-qubit gates before simulating the algorithm. This is most likely an artifact from the early days of quantum programming, where algorithms were primarily specified in terms of one- and two-qubit gates. While this allows for a particularly simple implementation of the classical simulator, it hides important optimization opportunities. For instance, the quantum Fourier transform can either be decomposed into single- and two-qubit gates, or it can directly be executed as a (fast) Fourier transform of the state vector, for which highly-optimized implementations exist [56, 57]. As an additional benefit, emulators allow quick testing of quantum algorithms that make heavy use of oracles that implement classical functions because they can just call the oracle without having access to an actual implementation. More details and performance results are presented in chapters 5 and 6.

Resource counters/estimators. In the search for practical applications of quantum computing devices, it is crucial to determine the constants that are hidden in the Big O notation which is typically used to compare algorithms. To this end, resource counters and estimators can be used as software backends to our methodology. Such a backend keeps track of circuit properties such as its size, width, and depth. Similar to the emulation idea, resource estimators can make use of the high-level description of the algorithm in order to take shortcuts where possible. For example, gates with similar or even identical implementation costs may be grouped together, allowing to estimate the resource requirements by performing the actual translation to hardware-instructions for only one instance of each such group.

2.2.5 Implementation: ProjectQ

In later work, we implemented this methodology with the described abstractions and optimizations in collaboration with Damian S. Steiger. This resulted in the launch of the ProjectQ framework [9]. ProjectQ features all the backends that were mentioned in the description of our methodology. Furthermore, the benefit of having code annotations and multiple gate sets with intermediate optimization passes was investigated subsequently in Ref. [10]. There are several libraries which extend the functionality of ProjectQ such as RevKit [11], which allows to automatically translate Python functions to quantum circuits, and FermiLib [12], which interfaces to classical electronic structure packages and helps to generate quantum circuits that perform time evolution under the resulting Hamiltonians. For more details on the ProjectQ framework, we refer to Refs. [9, 58].

Chapter 3

Using Hoare logic to optimize quantum programs

The software methodology which was introduced in the previous chapter already provides several means of optimization. In addition to constant-folding, which allows to combine multiple gates acting in sequence on some small set of qubits, optimizations at a more global scale are enabled via code-annotations. These can be used by the compiler in order to identify and leverage common patterns in quantum programming such as *compute/action/uncompute* sequences. Further optimization opportunities can be created by employing a set of commutation relations [59] to reorder operations. In general, however, such commutator based approaches incur a cost that is exponential in the number of qubits that the reordered operations act upon. Furthermore, several methods have been developed for exact circuit synthesis with certain optimality guarantees [60, 61, 62, 63, 64]. However, these methods are suited only for optimization of quantum circuits with a small number of qubits (or even single-qubit gates in the case of Ref. [63]).

Compilers with scalable optimization capabilities [9, 65, 36, 37, 66, 67] already allow for significant improvements [10]. Yet, because these optimization approaches do not take into account the state of the quantum computer throughout the computation, they fail to identify certain optimization opportunities. A straightforward way to implement this would be to simulate the entire quantum program during the optimization process. The difficulty, however, lies in devising a *scalable* optimization algorithm.

To this end, we propose to employ the Hoare triples of all invoked subroutines in order to gather information about the state. In order to perform circuit optimization, this information is then combined with conditions which, for every given operation, specify under which circumstances the operation is trivial. We implement this optimization methodology in ProjectQ [9] using the Z3 Theorem Prover [68] and perform comparisons to demonstrate the benefits of our approach.

When compared to the state of the art, our optimization methodology achieves reductions in circuit area of up to 5×. Specifically, it achieves a 2× reduction for floating-point mantissa renormalization and a 5× reduction for mapping an entangling circuit to a 1D linear chain. In particular, the use of Hoare triples allows our methodology to perform certain optimizations that would typically be performed only by humans.

This chapter is a slightly modified version of Ref. [2].

3.1 Quantum programs

In this chapter, we consider a slightly simplified machine model. Namely a combination of a classical von Neumann architecture and quantum circuits which get sent to the quantum co-processor for execution. We thus decide to ignore the possibility of additional classical controllers for lower-level tasks, since the focus lies solely on circuit optimizations. In this setting, the host sends circuits to the quantum co-processor which can be seen as a sequence of *quantum instructions*:

Definition 3.1.1: Quantum instruction

Let $O\left|q_1,...,q_k\right\rangle$ denote a *quantum instruction*. It consists of an operation O and a k-tuple of qubits $(q_1,...,q_k)$, where the operation may be a quantum gate or a classical instruction (allocation, deallocation, measurement).

Every circuit consists of the following 4 steps:

1. Allocate n qubits in state $\left|0\right\rangle^{\otimes n} := \left|0\cdots 0\right\rangle$ (n zeros)
2. Apply quantum gates to these qubits
3. Measure some or all of the qubits
4. Deallocate measured qubits

Upon completion of the execution of a circuit, the quantum co-processor returns a set of classical bits, the so-called *measurement results*. Conditional on these results, the classical processor may then provide further quantum circuits to evaluate in order to solve the computational problem at hand. At the end of the entire quantum program, all qubits are returned to the deallocated state.

3.2 Hoare tiples and their use for optimization

Hoare logic [32] is a system which can be used in order to verify the correctness of a given program. To this end, every subroutine in the program is equipped with additional information, the so-called Hoare triple, consisting of preconditions, the function, statement or subroutine to execute, and the corresponding postconditions. This is written as

$$\{P\}\,F\,\{Q\}\ ,$$

where P denotes the preconditions, F the function to execute, and Q the postconditions.

From an abstract point of view, the given program can then be seen as a sequence of transition rules, where each such rule is given by a Hoare triple. If each such transition happens in a way that ensures the preconditions of the next transition, the Hoare triple of the entire program, taking the initial conditions of the first transition to the postconditions of the final Hoare triple is provably correct. Whether the specifications of pre- and postconditions actually agree with the implementation is a different issue that we shall not be concerned with in this work.

A first step toward optimization via Hoare triples is to provide multiple implementations that ensure identical postconditions. The most general implementation assumes minimal preconditions for the operation to be sensible. As a result, this implementation will incur performance loss in cases where additional conditions are satisfied that would allow for a less general implementation. Such additional conditions can be added to the preconditions of another implementation which uses this additional information to achieve a performance advantage. In a scenario where these additional preconditions are satisfied, the optimized implementation may then be chosen automatically by the compiler in order to improve upon the generic implementation. Next we provide two simple examples in the quantum domain for this type of optimization.

Example

A straightforward example is that controlled operations U^c can be removed if the control qubit $|c\rangle$ is known to be $|0\rangle$, or that the control can be removed and the operation U can be applied directly if the control qubit is in definite $|1\rangle$. Since implementing U^c is more expensive than implementing just U, either case yields an advantage. In terms of Hoare triples, this can be phrased as follows:

precondition: Instruction qubit(s) in $|c\rangle\,|\psi\rangle$

operation: Apply U^c
postcondition: Instruction qubit(s) in $|c\rangle U^c |\psi\rangle$

However, also the identity operation (which has zero implementation cost) ensures the same postcondition albeit with a more restrictive precondition:

precondition: Instruction qubit(s) in $|0\rangle |\psi\rangle$
operation: None
postcondition: Instruction qubit(s) in $|c\rangle U^c |\psi\rangle$

And similarly for the control qubit in definite $|1\rangle$.

Example

A more practical example is addition by a classical constant, i.e., the n-qubit mapping

$$|x\rangle \mapsto |x+c\rangle \ ,$$

where the compiler can either use the addition circuit from Ref. [6] or, if extra n clean qubits in $|0\rangle$ are available, a full addition circuit such as the one in Ref. [69] to perform

$$|x\rangle |0\rangle \mapsto |x\rangle |c\rangle \mapsto |x+c\rangle |c\rangle \mapsto |x+c\rangle |0\rangle \ ,$$

which requires $\mathcal{O}(n)$ gates as opposed to $\mathcal{O}(n \log n)$ for the implementation without additional work qubits in Ref. [6].
The translation to Hoare triples is again straight-forward: With the additional precondition that n qubits are available as work qubits, the $\mathcal{O}(n)$ gate addition circuit from Ref. [69] ensures the instruction qubits to be in $|x+c\rangle$.

While optimizations mentioned thus far can be performed using Hoare triples, knowledge of the Hoare triple is not strictly necessary to do so. For example, simple constant-folding can be used to remove controlled gates, where the control qubit is in the computational basis state $|0\rangle$.

In what follows, we extend the optimization capabilities of compilers to handle cases which could not be optimized without the additional information that is provided by Hoare triples.

3.3 Compiler optimization via Hoare logic

In order to exploit the structure in superposition states that is present due to entanglement for the purpose of optimization, we gather additional information about the state of the system via Hoare triples. Because our aim is to optimize circuits and not to prove correctness of the entire quantum program (which includes measurements and feedback), we focus on pure states rather than employing the more general quantum Hoare logic developed in Ref. [70]. In particular, we introduce a formalism to describe the entanglement between the qubits of the system throughout the execution of the quantum circuit. This entails statements which assert entanglement descriptions (to be defined next), that is, statements of the form

$$\text{“}\mathtt{q} == f(\mathtt{q},\ \mathtt{r})\text{”},\ \text{“}\mathtt{q} \geq f(\mathtt{q},\ \mathtt{r})\text{”},\ \text{etc.},$$

where `q,r` refer to quantum registers and f is a function of two registers returning one register of bits. Since `q,r` refer to quantum registers, they may be in superposition and entangled with other qubits in the system. As a result, we need to assign a precise meaning to these *entanglement description assertions* with respect to the state vector of the entire n-qubit quantum computer,

$$|\psi\rangle = \sum_{i=0}^{2^n-1} \alpha_i |i\rangle \ .$$

Definition 3.3.1: Entanglement description assertion.

An *entanglement description assertion* on the n-qubit quantum state $|\psi\rangle$ above asserts $A(q,r)$ on $|\psi\rangle$, where

$$A(q,r) = \mathtt{q} \text{ cmp } f(\mathtt{q},\ \mathtt{r}) \ ,$$

with cmp being a comparison operator, $f : \{0,1\}^k \times \{0,1\}^m \to \{0,1\}^k$ a function on $k+m$ bits returning k bits, and `q,r` refering to quantum registers consisting of k and m qubits, respectively. Asserting $A(q,r)$ on $|\psi\rangle$ means that

$$\forall i \in \{0,...,2^n-1\} : (|\alpha_i| > 0 \implies A(\mathbf{q}(i), \mathbf{r}(i))) \ ,$$

where $\mathbf{q}(i), \mathbf{r}(i)$ extract the bits corresponding to the quantum registers `q` and `r`, respectively, from the computational basis state $|i\rangle = |i_{n-1}, ..., i_0\rangle$.

With this definition in place, let us revisit the $|0\rangle$-control qubit example from the previous section and cast it as an *entanglement description assertion.*

Example

To express that a qubit `c` is in a definite state $|0\rangle$, let $f(\cdot,\cdot) = 0$ and `cmp` be the equals comparison operator in the above definition. Then $A(c,-) = (\texttt{c} == 0)$. For the corresponding state $|\psi\rangle$, this means that $\alpha_i = 0$ whenever i corresponds to a state where the control qubit is 1. As a result, the action of the controlled gate on $|\psi\rangle$ is always trivial.

This shows that such assertions can be used to express knowledge about qubits that are in a definite state—a piece of information which, when combined with classical constant-folding, can be used for optimization. However, in order to do so in a more general setting, the optimizer also needs information which specifies the conditions for an operation to be trivial. We call this information *triviality conditions.*

Definition 3.3.2: Triviality condition.

A *triviality condition* of a quantum operation U is specified using an *entanglement description assertion* that asserts $A(q,r)$ on the quantum state $|\psi\rangle$. The following holds

$$A(q,r) \implies U|\psi\rangle = |\psi\rangle \ ,$$

meaning that U acts as the identity if $A(q,r)$ is satisfied by $|\psi\rangle$.

Example

Continuing the $|0\rangle$-control qubit example, the triviality condition of the controlled unitary U^c would read $\{c == 0\}$ and, if this is satisfied as in the previous example, U^c can be removed from the circuit.

Using these two definitions, we can thus describe and carry out classical constant-folding. In order to see that this approach is strictly more powerful in terms of potential for optimization, consider the following example.

Example

Let $|\psi\rangle$ denote the quantum state of a two-qubit quantum computer. Initially, $|\psi\rangle = |00\rangle$ and our quantum program consists of two operations: 1) Prepare a Bell-pair and 2) perform a Swap gate. The Bell-pair preparation routine has $\{q_0 == 0, q_1 == 0\}$ as preconditions and ensures that $\{q_0 == q_1\}$. In particular, given that the precondition is satisfied, our Bell-pair preparation circuit transforms the state $|00\rangle$ to

$$\frac{1}{\sqrt{2}}(|00\rangle + |11\rangle) \ ,$$

The amplitudes of this quantum state are $\alpha_{00} = 1/\sqrt{2}$, $\alpha_{01} = \alpha_{10} = 0$, and $\alpha_{11} = 1/\sqrt{2}$. It is easy to check that

$$\forall i \in \{0, 1, 2, 3\} : |\alpha_i| > 0 \implies \{i_0 == i_1\}$$

holds, where i_0 and i_1 denote the 0th and 1st bit of i, respectively. Since the Swap gate acts trivially if $q_0 == q_1$, which is satisfied by the state above, we see that it can be removed from the circuit.

Since the quantum state above is in a superposition, it is clear that regular constant-folding would fail to perform this optimization. Using our entanglement description assertions, however, this becomes feasible. While the Bell-pair Swap example may be a rather contrived one, we will now present more involved examples that are of practical interest and that allow for significant resource savings.

3.4 Practical example: Optimizing floating-point arithmetic

In this section, we discuss the optimization of floating-point arithmetic using entanglement description assertions.

Besides functions that are inherently quantum such as the quantum Fourier transform or the Hadamard transform [41], quantum computers also need to evaluate classical functions albeit on a superposition of inputs. Because the input is in a superposition, one cannot simply evaluate these functions on a classical computer as this would require reading out the state of the system, which would collapse the superposition and, thus, destroy any quantum speedup. Rather, these functions have to be implemented as quantum gates in order to run them directly on the quantum computer.

Examples where the evaluation of such classical functions on a quantum computer is necessary are 1) Shor's algorithm for factoring integers, which requires an implementation of modular exponentiation [13], and 2) certain algorithms for solving quantum chemistry problems. In Ref. [71], the authors reduce the asymptotic runtime of a chemistry simulation algorithm by computing the entries of the Hamiltonian on-the-fly. This involves evaluating the Coulomb potential and various other mathematical functions which, e.g., describe the chosen orbitals. In order to provide such functionality, one may start with implementing basic modules for floating-point arithmetic such as addition and multiplication. These modules can then be combined to enable evaluating polynomials and further higher-level mathematical functions.

As a practical example for our optimization methodology, we consider a subroutine which is omnipresent in floating-point arithmetic, namely that of *renormalization*. Renormalization is used during floating-point computations in order to bring intermediate results back into proper floating-point form. This can be achieved using two subroutines: The first subroutine determines the position p of the first nonzero bit of the mantissa. The second subroutine then shifts the mantissa to the left by the output of the first subroutine. A quantum circuit which determines the position of the first nonzero bit is shown in Fig. 3.1 and a circuit which shifts the mantissa $|x\rangle$ by $|p\rangle$ positions is depicted in Fig. 3.2. In order for the shift circuit to work properly for any input, it must allocate $2^{n_p}-1$ extra work qubits in order to catch the overflow from the shifted $|x\rangle$, where n_p is the number of qubits in the position register $|p\rangle$. However, in the case where the input to the shift circuit gets initialized by the circuit which determines the position of the first one, such an overflow never occurs. As a result, the $2^{n_p}-1$ work qubits can be eliminated from the combined circuit.

However, to identify this optimization from the circuit description (combine circuits in Fig. 3.1 and Fig. 3.2) is nontrivial and without some description of the action of gates or entire subroutines, such an optimization becomes completely infeasible for large circuits (as it would require simulation thereof for all inputs). We thus introduce a notion of how gates and subroutines interact by providing appropriate entanglement description assertions.

For this concrete example, take the postcondition of `determine_first_one`, which asserts that the first `pos` qubits of `x` are zero, i.e.,

$$\forall i \in 0..\texttt{pos-1} : \texttt{x[i] == 0} \ ,$$

where `pos` and `x` are entangled quantum variables. We can express this equivalently as an entanglement description assertion with

$$A_{FO}(x,p) = (x < 2^{n-p}) \ ,$$

where x is interpreted as an integer with x_0 as the most-significant bit (MSB) and p corresponds to the position register `pos` from above with p_0 being the least-significant bit (LSB). Using this postcondition, we now optimize the circuit in Fig. 3.2, which achieves the desired shift. Clearly, the red Fredkin gates can be removed since they act on newly allocated qubits which are zero (the postcondition

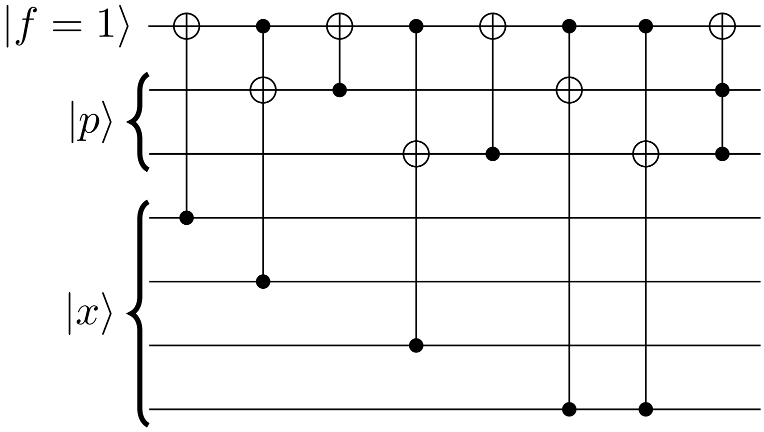


Figure 3.1: Example of a circuit which finds the first nonzero bit of $|x\rangle$ and stores its position in $|p\rangle$ where $|x\rangle$ is a 4-qubit register and the position register $|p\rangle$ consists of two qubits. The flag qubit $|f\rangle$ is one as long as the first one has not been found.

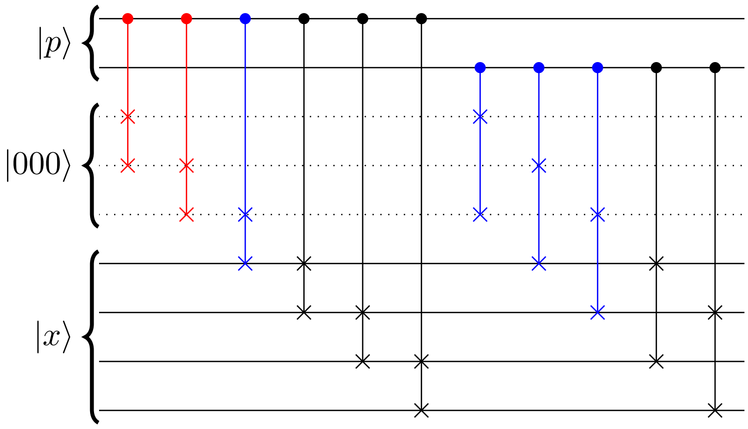


Figure 3.2: Optimization of the shift circuit which can be performed if $|p\rangle$ is the output of the circuit which determines the position of the first nonzero bit, see Fig. 3.1. All colored *Fredkin gates* [72] can be removed using the postconditions of the gates in Fig. 3.1 on $|p\rangle$. As a result, all $2^{n_p}-1$ work qubits can be eliminated (dotted lines) since no gates act on them.

of `q = allocate(n)` is that $q == 0$). The left-most blue Fredkin gate is a Swap gate controlled on the 0-th bit of $|p\rangle$ and thus acts trivially if $p_0 == 0$. Furthermore, the Swap itself is trivial if $x_0 == 0$ because all ancilla qubits are still in $|0\rangle$. Combining these two triviality conditions of the controlled Swap gate with the postcondition above yields that the blue Fredkin gate may act nontrivially only if

$$(p > 0) \wedge (x < 2^{n-p}) \wedge (x_0 \neq 0) \ ,$$

where x_0 denotes the MSB of the n-qubit register x. Clearly, these conditions cannot hold simultaneously and, as a result, the first blue Fredkin gate in Fig. 3.2 can be removed. Combining the postconditions of the Fredkin gates with $A_{FO}(x, p)$ yields a new assertion with

$$\begin{aligned} A_{new}(x,p) &= (2^{-p_0}x < 2^{n-p}) \\ &= (x < 2^{n-p+p_0}) \ , \end{aligned}$$

because if the first bit of the position register p_0 is one, we have just shifted all of x by one position. Since we successfully removed the first blue Fredkin gate, we can employ regular constant-folding to cancel the second blue Fredkin gate as well (all ancilla qubits are still in $|0\rangle$). For the final two blue Fredkin gates, note that they act nontrivially only if

$$(p_1 \neq 0) \wedge ((x_0 \neq 0) \vee (x_1 \neq 0)) \ .$$

From which we can use $p_1 \neq 0$ and combine it with the updated postcondition with a case-distinction on p_0: If p_0 is zero, then $p \geq 2$ and if p_0 is one, we have that $p \geq 3$ and that there is a shift of $+1$ in the exponent of the updated postcondition. Thus, in both cases,

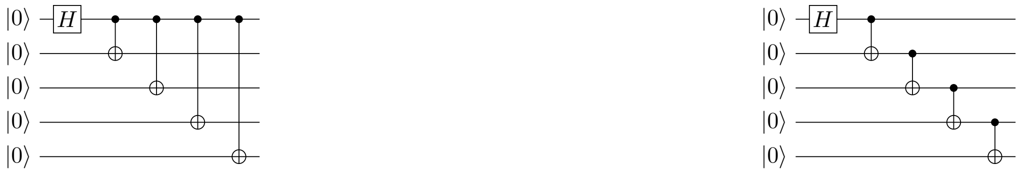
$$x < 2^{n-2} \ ,$$

and hence, the two most-significant bits x_0, x_1 of x must be zero. The action of the remaining two blue Fredkin gates is therefore always trivial and they can also be removed from the circuit. Finally, since none of the allocated overflow qubits will be used anymore throughout the computation (as their content is always trivial in this application), they will eventually get deallocated without any operations having acted on them. It is then a simple local optimization to cancel allocations with subsequent deallocations, allowing to reduce the width of the resulting circuit by $2^{n_p} - 1$ qubits, as desired.

While in this example we used explicit postconditions of `determine_first_one`, we demonstrate in the implementation section that it is enough to provide post- and triviality conditions of three operations—NOT, Swap, and Allocate—, in addition to the triviality condition of the control modifier, which turns a given gate U into its controlled version U^c.

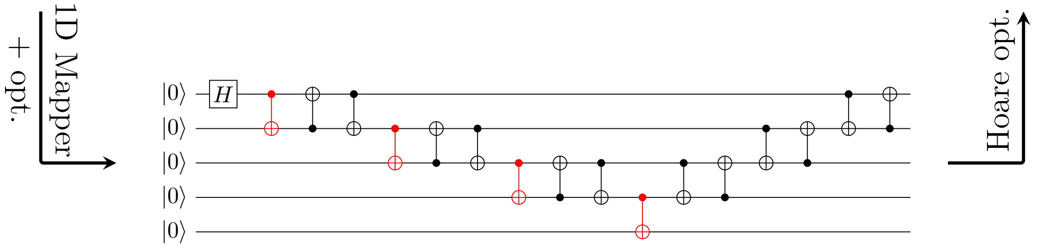
Furthermore, we note that such optimization opportunities also arise when using a fixed-point representation. For example, carrying out range reductions by 2^k in order to evaluate functions such as

$$\log_2(y) = \log_2(x2^{-k}) = \log_2(x) + \log_2(2^{-k}) = \log_2(x) - k \ ,$$



(a) Original circuit.

(c) Circuit for LNN after optimization.



(b) Circuit for LNN before Hoare optimization.

Figure 3.3: Optimizing a chain of CNOTs for a linear nearest-neighbor (LNN) architecture by employing the Hoare triple for a CNOT which can be obtained by combining the `ctrl` modifier with the Hoare triple of the Pauli X gate. The benefit of our optimization can be seen clearly when comparing the circuits in **(b)** and **(c)**: No Swaps are necessary in **(c)**, resulting in much lower gate count and circuit depth.

with $x \in [1, 2)$ allows for a very similar optimization opportunity: When determining x and k, one can shift y without using ancilla qubits, which is analogous to our optimization for floating-point renormalization.

3.5 Formalization and generalization

In this section, we formalize the deduction rules necessary to carry out all optimization examples mentioned thus far before introducing a generalization which is strictly more powerful.

3.5.1 Formalization of our basic methodology

In order to formalize the basics of our methodology, we first define the Hoare triples of the quantum subroutines that are required for our examples.

$$
\begin{gathered}
\Big\{\, q = \emptyset; n \in \mathbb{N} \,\Big\}\, q = \mathit{Alloc}(n) \,\Big\{\, q = |0\rangle^{\otimes n} \,\Big\} \\
\Big\{\, q = |0\rangle^{\otimes n} \,\Big\}\, \mathit{Dealloc}(q) \,\Big\{\, q = \emptyset \,\Big\} \\
\Big\{\, q_i = A, q_j = B \,\Big\}\, \mathit{Swap}(q_i, q_j) \,\Big\{\, q_i = B, q_j = A \,\Big\} \\
\Big\{\, q = A, A \in \{0,1\} \,\Big\}\, X(q) \,\Big\{\, q = A \oplus 1 \,\Big\}
\end{gathered}
$$

where $X(q)$ denotes application of a Pauli-X gate to qubit q. From the pre- and postconditions of the Swap operation, it is also apparent that a Swap is trivial if $q_i == q_j$; a fact which we already used in our examples.

In addition to the Hoare triples above, we require a formal description of the control modifier, which turns a given quantum subroutine U into its controlled version U^c, where c refers to the control qubit. The corresponding Hoare triple is

$$
\Big\{\, c \in \{0,1\}, q = |\psi\rangle \,\Big\}\, \mathit{control}(U)(c,q) \,\Big\{\, q = U^c\,|\psi\rangle \,\Big\}\ ,
$$

where U^c denotes U raised to the c-th power, i.e., it is U if $c = 1$ and $U^c = \mathbb{1}$ if $c = 0$.

The Hoare triple of the control modifier can be combined in arbitrary ways with the Hoare triples of our subroutines. In particular, a combination of the Swap routine with the control modifier yields the rules that were used to remove trivial Fredkin gates in the circuit in Fig. 3.2. Combining it with the NOT or Pauli X gate, on the other hand, lets us optimize the Bell-pair example where, after an initial Hadamard gate H on $|00\rangle$, the controlled NOT gate was applied as follows

$$
(H \otimes \mathbb{1})\,|0\rangle\,|0\rangle = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)\,|0\rangle \overset{CNOT}{\mapsto} \frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)\ .
$$

Since this controlled NOT gate above flips the qubit in $|0\rangle$ if the control qubit is one, we immediately get the postconditions for the two qubits q_0 and q_1

$$
\{q_1 = X^{q_0}\,|0\rangle\} \implies \{q_1 == q_0\}\ ,
$$

by combining the two Hoare triples. Together with the triviality condition of the Swap gate acting on two qubits q_i and q_j,

$$
\{q_i == q_j\}\ ,
$$

we can again remove the Swap gate from the circuit of the Bell-pair example.

Similarly, the Hoare triple for CNOT can be used to identify the optimization opportunity in the following example.

Example

In addition to circuit optimizations at the logical level, entanglement description assertions and triviality conditions can be used to optimize the circuit for a specific target architecture. Consider the compilation steps outlined in Fig. 3.3. After mapping the circuit in **(a)** to a linear nearest-neighbor connectivity with additional optimizations to cancel intermediate partial Swap chains results in the circuit **(b)**. As before, we can employ our Hoare logic optimizer to remove trivial CNOT gates using the fact that after each red CNOT gate acting on q_i and q_{i+1}, it holds that $q_i == q_{i+1}$. The optimized circuit is shown in Fig. 3.3**(c)**.

3.5.2 Generalized optimization methodology

Generalizing the basic methodology above allows to greatly increase its optimization capabilities. Thus far, our optimizer considers single gates at any given point together with all available postconditions of previously executed subroutines. For each such gate, it then determines whether it can be removed from the circuit without altering its output. The generalized strategy considers multiple gates and checks whether the supplied postconditions allow to deduce that the combined action of these gates is trivial, in which case all of these gates can be removed from the circuit.

Definition 3.5.1: Set of control qubits.

For an instruction $U\,|q_1, ..., q_k\rangle$ acting with a (unitary) gate U on k qubits, a set of qubits $\mathcal{S} \subset \{q_1, ..., q_k\}$ is called a *set of control qubits* if there exists a sequence of Swap gates $s_1, ..., s_t$ acting on pairs from $\{q_1, ..., q_k\}$ and a unitary U' such that with S denoting the unitary which performs $s_1, ..., s_t$, the following three statements hold.

$$(1)\ \ SUS^\dagger = (\mathbb{1} - |1\cdots1\rangle\langle1\cdots1|) \otimes \mathbb{1} + |1\cdots1\rangle\langle1\cdots1| \otimes U'\ ,$$

(2) the sequence of Swaps $(s_1, ..., s_t)$ permutes $(q_1, ..., q_k)$ such that the first $|\mathcal{S}|$ qubits of the resulting tuple are in $\mathcal{S}$, and (3) $\mathcal{S}$ is the largest such set. For instructions where U is not unitary, the set of control qubits is empty.

We note that there may be multiple distinct sets of control qubits for a given instruction. For instructions where multiple choices exist, we choose a set of control qubits once and keep it invariant throughout the optimization process.

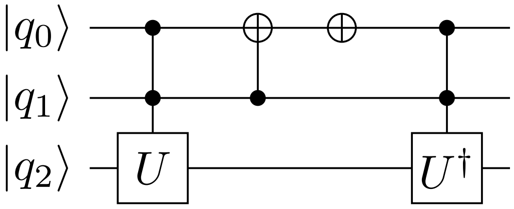


Figure 3.4: Simple example of our multi-gate optimization methodology: The first gate is applied if and only if the last gate is applied. Since the U gate is the inverse of $U^\dagger$, we can cancel the two doubly-controlled gates.

Example (Set of control qubits may be nonunique).

As an example of an instruction where multiple choices exist for the set of control qubits, consider the Z^c operation applied to $|q_1 q_0\rangle$, where Z acts with a (-1)–phase on $|1\rangle$ and leaves $|0\rangle$ invariant. It is easy to check that

$$\begin{aligned} Z^c &= |0\rangle\langle 0| \otimes \mathbb{1} + |1\rangle\langle 1| \otimes Z \\ &= \mathbb{1} \otimes |0\rangle\langle 0| + Z \otimes |1\rangle\langle 1| \ , \end{aligned}$$

since for Z^c to be nontrivial, both qubits need to be in $|1\rangle$. Either qubit can thus be chosen to be the control qubit and, thus, the set of control qubits is nonunique.

Definition 3.5.2: Target qubit.

A qubit q in an instruction $U|..., q, ...\rangle$ is called a *target qubit* if it is not in the set of control qubits of the instruction $U|..., q, ...\rangle$.

Definition 3.5.3: Target-successive instructions.

Two instructions I_1, I_2 with identical target qubits are called *target-successive* if no other instructions are scheduled to be executed between I_1 and I_2 that involve the target qubits in a way that does not commute with neither I_1 nor I_2.

Our generalized methodology considers $M \geq 1$ target-successive instructions at once, where all M instructions have the same t target qubits and arbitrary controls.

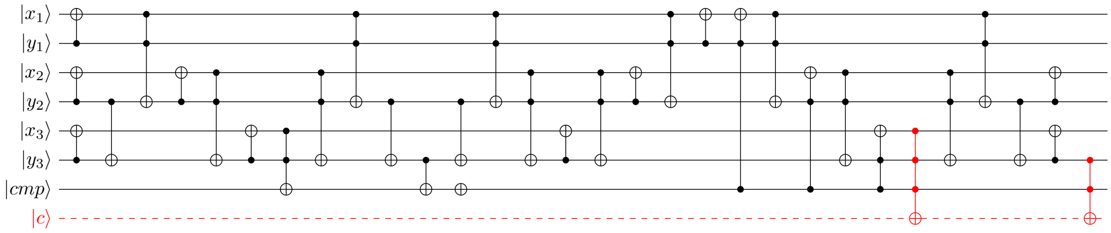


Figure 3.5: Three-qubit example of a modular adder subroutine which performs the modular reduction. It consists of a comparison, the result of which is stored in the qubit $|cmp\rangle$, and a conditional subtraction. In this setting, our generalized methodology is able to deduce that the two red multi-controlled NOT gates can be canceled, allowing to completely remove the carry qubit $|c\rangle$.

Ignoring the control qubits, let $U_1, ..., U_M$ denote the t-qubit gate matrices of these instructions. An optimization can be performed if

$$U_1 \cdots U_M = \mathbb{1}_{2^t \times 2^t}$$

and the postconditions on the control qubits are such that either all or none of the gates get executed. A simple example with $M = 2$ and $t = 1$ is depicted in Fig. 3.4, where the two doubly-controlled gates can be canceled using the reasoning above.

We now give a practical example where our multi-gate optimization strategy performs better than the single-gate methodology discussed thus far.

Example

Consider a circuit which performs addition modulo a number N that is stored in another quantum register, i.e.,

$$|a\rangle\,|b\rangle\,|N\rangle \mapsto |(a+b) \bmod N\rangle\,|b\rangle\,|N\rangle \ .$$

A possible implementation is to first perform the regular addition, followed by a modular reduction if the result is greater than $|N\rangle$. Since we only subtract N if $(a+b) \geq N$, the result will always be non-negative and, as a consequence, the final carry will always be zero and the qubit can thus be removed from the circuit. When using the addition circuit from Ref. [69], the optimizer needs to remove the two red multi-controlled NOT gates in Fig. 3.5 which act on the carry qubit in order to exploit this optimization opportunity. Neither of these gates is trivial by itself but in this setting, either both or none of the two gates are triggered. As a result, this optimization can only be performed using our generalized approach. The achieved reduction in circuit width and depth can be found in the results section.

3.6 Implementation using ProjectQ and Z3

In this section, we discuss our implementation of this optimization methodology. For each quantum operation for which we would like to add nontrivial optimization capabilities using our approach, we require

1. Postconditions
2. Triviality conditions

Additionally, preconditions may be supplied which would allow to test the program for correctness. For our generalized methodology, we only require the triviality condition of the control modifier in addition to information which lets us determine whether a sequence of operations $U_1, ..., U_M$ acts as the identity. However, the latter is already available in ProjectQ.

We extend the definitions of several gate operations in ProjectQ with their respective post- and triviality conditions by providing additional member functions which employ the Z3 Theorem Prover package [68] to express these conditions. These member functions are invoked by our custom optimizing compiler engine, which then employs the Z3 solver in order to check whether certain operations are guaranteed to be trivial, in which case they can be removed.

While we do not elaborate on the details of the ProjectQ compilation framework, we point out that optimization and compilation is carried out during circuit generation time. As a result, all parameters of the circuit are already known. In particular, the length of quantum registers is determined since all classical input to the quantum program has been supplied. The circuit can thus be optimized specifically to the problem size in question—a feature that is crucial especially for near- and intermediate-term devices which have very limited resources, making such additional optimizations very valuable. In turn, this enables more powerful optimizations when employing our methodology because we are not required to carry out parametric proofs. It is of course theoretically possible to prove such statements by induction, however, there is only limited support in automatic theorem provers such as Z3 [68]. Since all classical parameters have a definite value upon circuit generation, we can unroll many quantified statements and thereby generate statements that are much easier to (dis)prove.

As an example, we show how the definition of the ProjectQ Swap operation was altered in order to enable our optimization engine to carry out the optimizations discussed so far. The definition of `SwapGate` was extended by merely the following two member functions:

```
class SwapGate(SelfInverseGate):
    [...]
    def trivial_if(self, x1, x2):
        return (x1 == x2)

    def postconditions(self, x1, x2, y1, y2):
        return And(x1 == y2, x2 == y1)
```

Clearly, these are very minor modifications which provide exactly the information required: Postconditions and triviality conditions of the Swap gate. The `trivial_if` member function of every gate is invoked by the optimizer with one symbolic boolean variable for each target qubit of the gate (two in this case). The returned expression is negated and then added to the solver together with the expression `ctrls_one = And(v[cqb`$_1$`], v[cqb`$_2$`], ...)`, which is true if and only if all variables `v[cqb`$_i$`]` corresponding to control qubits `cqb`$_i$ are true / equal to one:

```
solver.push()
solver.add(And(ctrls_one, Not(cmd.gate.trivial_if(*target_vars))))
if solver.check() == unsat:
  skip_current_op()
solver.pop()
```

where `target_vars` are the Z3 variables corresponding to the target qubits of the current gate before it is executed. If the solver finds a solution which satisfies all previous conditions and the negated conditions of `trivial_if`, the gate cannot be removed since it may have a nontrivial effect on the state of the quantum computer $|\psi\rangle$ at that point. If there is no such solution, on the other hand, this means that the gate is trivial and it can thus be removed from the circuit. After this triviality check, the conditions of the Z3 solver are updated according to the postconditions of the operation which hold irrespective of whether the gate was removed: For each target qubit, a new boolean Z3 variable is created and the `postconditions` member function of the gate relates the old variables (before applying the gate) to the new ones. In particular, operations are handled by adding two Z3 `Implies(...)` statements:

1. The control qubit(s) being all ones implies that the new target variables are now related to the old ones via the `postconditions` function, i.e.,

   ```
   Implies(ctrls_one, cmd.gate.postconditions(*(target_vars+
       new_target_vars)))
   ```

   is added to the solver, where `new_target_vars` are the Z3 variables that correspond to the target qubits after applying the gate.

2. The control qubit(s) not being all ones implies that the new target variables are equal to the old ones, i.e., for all i we add the expression

   ```
   Implies(Not(ctrls_one), new_target_vars[i] == target_vars[i]))
   ```

to the solver.

If there are no control qubits, (1) and (2) are of the form

$$\{\texttt{true} \implies y = f(x)\} \text{ and } \{\texttt{false} \implies y = x\} ,$$

respectively and, therefore, are equivalent to stating that $y = f(x)$ holds after the gate has been applied, where f is given by the `postconditions` member function.

Also, note that the ProjectQ Swap gate derives from `SelfInverseGate`, stating that the Swap operation is its own inverse. This information is useful for our generalized optimization approach, which is employed whenever the circuit buffer size of the optimizer exceeds a user-defined threshold. When this happens, the stored circuit is traversed in order to identify target-successive operations which may be removed from the circuit. For each such sequence of gates, the Z3 solver is used to determine whether there is an assignment to the control qubits which agrees with all previous postconditions and which causes $0 < m < M$ operations to be executed. If there is no such assignment, either all or none of these M operations are executed, meaning that they always act trivially. As a result, the entire sequence of gates can be removed from the circuit.

3.7 Results and comparison

In this section, we report the results that were obtained using the implementation of our optimization methodology. We analyze the performance of our Hoare logic based optimizer with respect to different quantum circuits. The first circuit performs floating-point mantissa renormalization, see Figs. 3.1 and 3.2, the second entangles a linear chain of qubits, see Fig. 3.3, and the third performs modular reduction, see Fig. 3.5, which is a subroutine that is used in constructing a modular adder. For all circuits, we compare two compiler setups—one which features a simple local optimizer capable of merging/canceling subsequent operations that act on the same qubits, and a second configuration which additionally contains our Hoare logic based optimizer. As a gate set, we choose {CNOT, X, H, S, $T, T^\dagger$} for all configurations.

In order to compare these compiler configurations, we use circuit width, depth, and area as benchmark numbers. The circuit area is computed as

$$A_C := \text{depth} \times \text{width}$$

where the depth is the depth of the directed acyclic graph (DAG) associated with the circuit, and the width corresponds to the maximal number of alive qubits at any point throughout the execution of the circuit.

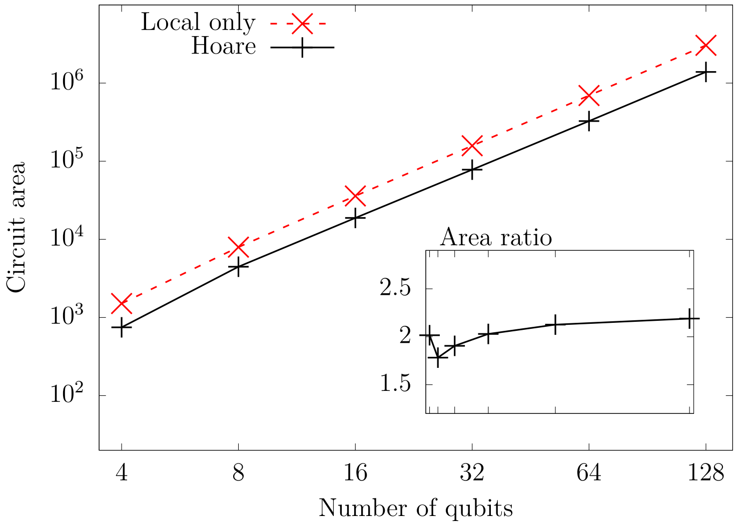


Figure 3.6: Optimizer comparison for the floating-point renormalization circuit with different number of qubits n for the size of the mantissa. The position register was chosen as $p := \lceil \log_2 n \rceil$. The Hoare optimizer achieves a roughly 2× reduction in circuit area over the local optimizer from ProjectQ [9].

Number of bits n	Max. circuit width	DAG depth
4	9 (11)	168 (315)
8	17 (19)	592 (639)
16	33 (35)	1240 (1287)
32	65 (67)	2536 (2583)
64	129 (131)	5128 (5175)
128	257 (259)	10312 (10359)

Table 3.1: Optimizer comparison for a modular reduction circuit on n qubits using the addition circuit in Ref. [69]. Our optimizer is able to remove the carry qubit of the conditional subtraction since it is always in $|0\rangle$ at the end of the circuit and not involved in the computation when non-zero. We note that this is only possible using our multi-gate or *generalized* approach.

The comparisons can be found in Fig. 3.6 and Fig. 3.7 for the first and second circuit, respectively. Both cases clearly demonstrate the benefits of our Hoare logic based optimizer, which is able to reduce the circuit area by a factor of approximately 2× and 5× for the first and second circuit, respectively.

In the first circuit, our optimizer is able to eliminate $2^{n_p} - 1$ ancilla qubits in addition to a few Fredkin gates. It is thus to be expected that the circuit area is

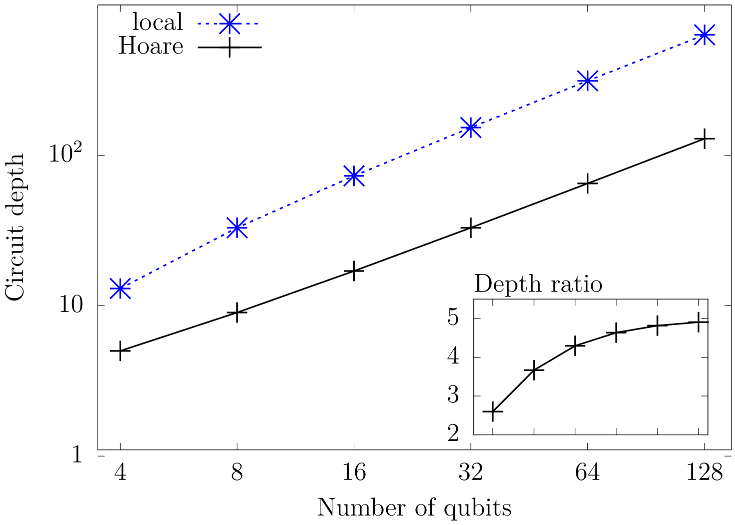


Figure 3.7: Optimizer comparison for the entangling circuit on n qubits depicted in Fig. 3.3. The Hoare optimizer achieves a 5× improvement in circuit depth for large n.

reduced by approximately a factor of two.

In the second circuit, all CNOT gates resulting from swap operations can be removed when using the Hoare based optimization strategy. We thus expect the circuit depth to grow by $4(n-2)$ gates for $n \geq 2$ when turning off Hoare logic optimization. The ratio between the resulting circuit depths for $n \geq 2$ is thus

$$\frac{4(n-2)+n}{n} = \frac{5n-8}{n} \overset{n\to\infty}{\to} 5\,,$$

which agrees with the experimental results in Fig. 3.7 and constitutes an up to 5× improvement over state-of-the-art optimizers.

The third circuit, which is a subroutine for modular addition, can be optimized by identifying a pattern similar (but more complex) to the one shown in Fig. 3.4. In this case, the target qubit is the carry qubit of the controlled subtraction and upon removing the two multi-controlled NOT operations, no operations on the carry qubit remain. As a result, this additional qubit can be removed from the circuit. Furthermore, due to the removal of multi-controlled NOT gates, no extra work qubits are required for Toffoli ladders [26].

3.8 Summary and future work

We have presented an optimization methodology that extends the scope of automatic circuit optimizations. In particular, our methodology allows to carry out certain optimizations that are typically performed by humans. This is achieved by taking into account post- and triviality conditions of all subroutines that get invoked by the quantum program that is being optimized. Our implementation in ProjectQ has managed to achieve up to 5× reduction in circuit area for our examples when compared to the state of the art.

Our generalized methodology currently performs optimizations if the overall action of a sequence of gates is trivial. Future work could address more general cases where, e.g., control qubits are in a state that only triggers subsets of these gates that, when combined, correspond to trivial operations. Additionally, symbolic computation on entanglement description assertions may be incorporated. This would allow to optimize iterative procedures such as the Newton-Raphson method which can be used to evaluate high-level arithmetic functions on a quantum computer [7]: For many such functions, the initial guesses can be chosen to be very simple (e.g., integer powers of two). The first iteration of a Newton-Raphson method may then be applied symbolically to the output of the initial guess routine. Such optimizations have been shown to yield significant resource savings when performed manually [7]. Automating such procedures would thus allow for the same benefits without the need for labor-intensive manual code optimization.

Chapter 4

Reducing resource requirements by optimizing error tolerances

Quantum program optimizations usually take place at the circuit level, such as the ones introduced in this thesis thus far. However, there are several choices that must be made by the compiler before these circuits are fully determined. For the purpose of quantum program optimization, a crucial choice in the compilation process is the selection of approximation error tolerances. While programmers may have pretty good estimates of the overall error which a given program can tolerate without significantly affecting its output, this is no longer true at the level of, e.g., single- and two-qubit gates.

To see that the interplay among subroutines with respect to such errors is nontrivial, consider the example of performing phase estimation on some unitary operation. Increasing the accuracy of quantum phase estimation, requires more applications of the controlled unitary operation. This, in turn, requires each of these controlled operations to be executed with higher accuracy. The resource requirements thus increase two-fold: Once due to the larger number of controlled unitary operators, and once because accuracy requirements increase as the number of operations increases.

Automating this process enables a better choice of accuracy parameters and, in turn, lower resource requirements. In this chapter, which is a slightly modified version of Ref. [3], we develop an optimization methodology to achieve this task. For demonstration purposes, we use our annealing-based optimizer to optimize error tolerances of a transverse-field Ising model simulation and report the achieved reduction in cost.

4.1 Compilation and approximation

In order to estimate the required resources of a given quantum algorithm, the high-level representation of the algorithm must be translated to a universal low-level set of operations that can be realized on the target hardware. One of the standard gate sets that is often considered is the so-called Clifford+T gate set, which can be generated from a few single-qubit operations and the CNOT (controlled-NOT) gate. In particular, arbitrary single-qubit rotations must be translated to this discrete gate set employing a *rotation synthesis* algorithm [73, 63], where the resulting gate sequences get longer as the desired accuracy is increased. Therefore, besides the problem of compiling abstract high-level functions to the native gate set, the resulting approximation errors need to be managed in a way that ensures that the resulting code performs the desired overall operation within a certain (user-specified) tolerance. We address this problem by introducing a method capable of handling these errors automatically.

The need for approximation. While it is not possible to perform error correction over a continuous set of quantum operations (gates), this can be achieved over a discrete gate set such as the aforementioned Clifford+T gate set. As a consequence, certain operations must be approximated using gates from this discrete set. An example is the operation which achieves a rotation around the z-axis,

$$\mathrm{Rz}_\theta = \begin{pmatrix} e^{-i\theta/2} & 0 \\ 0 & e^{i\theta/2} \end{pmatrix}.$$

To implement such a gate over Clifford+T, synthesis algorithms such as the ones in Refs. [73, 63] can be used. Given the angle θ of this gate, such a rotation synthesis algorithm will produce a sequence of $\mathcal{O}(\log \varepsilon_R^{-1})$ Clifford+T gates which approximate Rz_θ up to a given tolerance ε_R. We measure approximation error ε_R with respect to distance in operator norm. As we focus on unitary channels this is equivalent to diamond distance, i.e., approximation errors can be composed safely.

In most error correction protocols, the T-gate is the most expensive operation to realize, as it cannot be executed natively but requires a distillation protocol to distill many noisy magic states into one good state, which can then be used to apply the gate. As a consequence, it is crucial to reduce the number of these T-gates as much as possible in order to allow executing a certain quantum computation.

Compilation of quantum programs. The job of a quantum program compiler is to translate a high-level description of a given quantum program to hardware-specific machine-level instructions. As in classical computing, such compilation frameworks can be implemented in a hardware-agnostic fashion by introducing backend-independent intermediate representations of the quantum code [1].

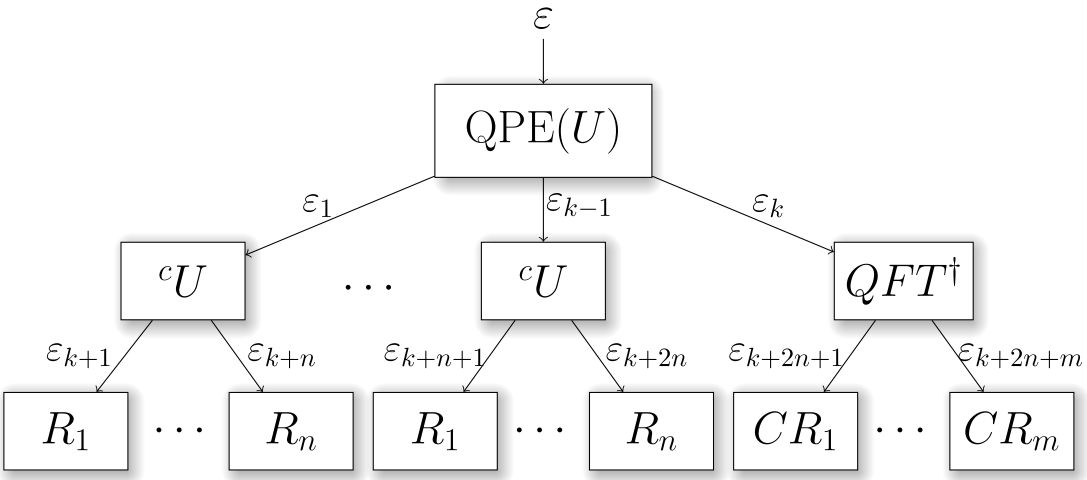


Figure 4.1: Abstract depiction of the compilation process for a quantum phase estimation (QPE) applied to a given unitary U. The parameters ε_i which get introduced during the compilation must be chosen such that the overall target accuracy ε is achieved while reducing the resulting cost as much as possible.

During the compilation process, it is crucial to optimize as much as possible in order to reduce the overall depth of the resulting circuit to keep the overhead of the required quantum error correction schemes manageable. Optimizations include quantum versions of constant-folding (such as merging consecutive rotation gates, or even additions by constants) and recognition of compute/action/uncompute sections to reduce the number of controlled gates [1]. To allow such optimizations, it is important to introduce multiple layers of abstractions instead of compiling directly down to low-level machine instructions [1, 9], which would make it impossible to recognize, e.g., two consecutive additions by constants. Even canceling a gate followed by its inverse becomes computationally hard, or even impossible once continuous gates have been approximated.

To translate an intermediate representation to the next lower level of abstraction, a set of decomposition rules is used, some of which introduce additional errors which can be made arbitrarily small at the cost of an increasing circuit size or depth, which in turn implies a larger overhead when applying quantum error correction. Therefore, it is important to choose these error tolerances such that the computation succeeds with high probability given the available resources (number and quality of qubits). See Fig. 4.1 for an abstract depiction of the compilation process of a quantum phase estimation on a given unitary U. At each level of abstraction, the compiler introduces additional accuracy parameters (in the figure denoted by ε_i) which must be chosen such that

1. the overall error lies within the specifications of the algorithm and

2. the implementation cost is as low as possible while 1) is satisfied.

As mentioned above, it is important to measure approximation errors in a way that is composable to avoid potential issues in underreporting actual approximation errors [74, 75] which could be devastating when composing complex quantum algorithms. This leads to diamond distance as the preferred way to measure closeness to the target operation as it composes. As unraveling a complex quantum algorithm eventually leads to primitive gates that are unitary—such as the mentioned Rz_θ rotations which are implemented on subsystem of a constant number of qubits—bounding the approximation error in operator norm implies error in diamond norm, i.e., estimates of approximation error can be composed.

4.2 Error-propagation in quantum circuits

The time-evolution of a closed quantum system can be described by a unitary operator. As a consequence, each time-step of our quantum computer can be described by a unitary matrix of dimension $2^n \times 2^n$ (excluding measurement), where n denotes the number of quantum bits (qubits). When decomposing such a quantum operation U into a sequence of lower-level operations $U_M \cdots U_1$, the resulting total error can be estimated from the individual errors ε of the lower-level gates as follows:

Lemma 4.2.1: Unitary error

Given a unitary decomposition of U such that $U = U_M \cdot U_{M-1} \cdots U_1$ and unitaries V_i which approximate the unitary operators U_i such that $\|V_i - U_i\| < \varepsilon_i \; \forall i$, the total error can be bounded as follows:

$$\|U - V_M \cdots V_1\| \leq \sum_{i=1}^{M} \varepsilon_i.$$

This lemma can be shown straightforwardly from the 'hybrid argument' [76] based on the triangle inequality and submultiplicativity of $\|\cdot\|$ with $\|U\| \leq 1$.

Note that using only this Lemma in the compilation process to automatically optimize the individual ε_i would make the resulting optimization problem infeasibly large. What is even worse is that the number of parameters to optimize would vary throughout the optimization process since the number of lower-level gates changes when implementing a higher-level operation at a different accuracy, which in turn changes the number of distinct ε_i. To address these two issues, we introduce Theorem 4.2.1 which generalizes Lemma 4.2.1. First, however, we require the following definitions.

Definition 4.2.1: Partitioning of subroutines

Let $V_{M(\varepsilon)} \cdots V_1$ be an approximate decomposition of the target unitary U such that $\|U - V_{M(\varepsilon)} \cdots V_1\| \leq \varepsilon$. A set of subroutine sets $\mathcal{S}(U, \varepsilon) = \{S_1, ..., S_K\}$ is a *partitioning of subroutines of* U if $\forall i \exists! k : V_i \in S_k$ and we denote by $S(V)$ the function which returns the subroutine set S such that $V \in S$.

Such a partitioning will be used to assign to each V_i the accuracy $\varepsilon_{S(V_i)} = \varepsilon_{S_k}$ with which all $V_i \in S_k$ are implemented. In order to decompose the cost of U, however, we also need the notion of a *cost-respecting partitioning of subroutines of* U and the costs of its subsets:

Definition 4.2.2: Cost-respecting partitioning of subroutines

et $\mathcal{S}(U, \varepsilon) = \{S_1, ..., S_K\}$ be a set of subroutine sets. $\mathcal{S}(U, \varepsilon)$ is a *cost-respecting partitioning of subroutines of* U w.r.t. a given cost measure $C(U, \varepsilon)$ if $\forall \varepsilon, i, j, k : (V_i \in S_k \wedge V_j \in S_k \Rightarrow C(V_i, \varepsilon) = C(V_j, \varepsilon))$. The cost of a subroutine set S is then well-defined and given by $C(S, \varepsilon) := C(V, \varepsilon)$ for any $V \in S$.

With these definitions in place, we are equipped to generalize Lemma 4.2.1.

Theorem 4.2.1: Decomposition of cost

Let $\mathcal{S}(U, \varepsilon) = \{S_1, ..., S_K\}$ be a cost-respecting partitioning of subroutines for a given decomposition of U w.r.t. the cost measure $C(U, \varepsilon)$ denoting the number of elementary gates required to implement U. Then the cost of U can be expressed in terms of the costs of all subroutine sets $S \in \mathcal{S}(U, \varepsilon_U)$ as follows

$$C(U, \varepsilon) = \sum_{S \in \mathcal{S}(U, \varepsilon_U)} C(S, \varepsilon_S) f_S(\varepsilon_U)$$

$$\text{with} \sum_{S \in \mathcal{S}(U, \varepsilon_U)} \varepsilon_S f_S(\varepsilon_U) \leq \varepsilon - \varepsilon_U,$$

where $f_S(\varepsilon_U)$ gives the number of subroutines in the decomposition of U that are in S, given that the decomposition of U would introduce error ε_U if all subroutines were to be implemented exactly and ε_S denotes the error in implementing subroutines that are in S.

Proof. It is easy to see that the cost $C(U,\varepsilon)$ can be decomposed into a sum of the costs of all subroutines V_i. Furthermore, since $\varepsilon_V = \varepsilon_S \ \forall V \in S$,

$$\begin{aligned} C(U,\varepsilon) &= \sum_i C(V_i, \varepsilon_{V_i}) \\ &= \sum_i C(V_i, \varepsilon_{S(V_i)}) \\ &= \sum_{S\in\mathcal{S}} |\{i : V_i \in S\}| C(S, \varepsilon_S) \end{aligned}$$

and $f_S(\varepsilon_U) := |\{i : V_i \in S\}| \ \forall S \in \mathcal{S}(U, \varepsilon_U)$.
To prove that the overall error remains bounded by ε, let $\tilde{U}$ denote the unitary which is obtained by applying the decomposition rule for U with accuracy ε_U, i.e., $\|U - \tilde{U}\| \leq \varepsilon_U$ (where all subroutines are implemented exactly). Furthermore, let V denote the unitary which will ultimately be executed by the quantum computer, i.e., the unitary which is obtained after all decomposition rules and approximations have been applied. By the triangle inequality and Lemma 4.2.1,

$$\begin{aligned} \|U - V\| &\leq \|U - \tilde{U}\| + \|\tilde{U} - V\| \\ &\leq \varepsilon_U + \sum_{S\in\mathcal{S}(U,\varepsilon_U)} \varepsilon_S f_S(\varepsilon_U) \\ &\leq \varepsilon \end{aligned}$$

□

In Fig. 4.1, for example, the left-most ${}^cU$ box gets ε_1 as its error budget. Depending on the implementation details of ${}^cU$, some of this budget may already be used to decompose ${}^cU$ into its subroutines, even assuming that all subroutines of ${}^cU$ are implemented exactly. The remaining error budget is then distributed among its subroutines, which is exactly the statement of the above theorem.

The decomposition of the cost can be performed at different levels of granularity. This translates into, e.g., having a larger set $\mathcal{S}(U,\varepsilon)$ and more functions $f_S(\varepsilon_U)$ that are equal to 1. The two extreme cases are

1. $f_S(\varepsilon) = 1 \ \forall S \in \mathcal{S}(U,\varepsilon)$, $|\mathcal{S}(U,\varepsilon)| =$ #gates needed to implement U:

   A different ε_U for each gate,

2. $f_S(\varepsilon) =$ #gates needed to implement $U \ \forall S \in \mathcal{S}(U,\varepsilon)$, $|\mathcal{S}(U,\varepsilon)| = 1$:

   The same $\varepsilon_\varnothing$ for all gates.

Therefore, this solves the first issue of Lemma 4.2.1: In a practical implementation, the size of the set $\mathcal{S}(U,\varepsilon)$ can be adaptively chosen such that the resulting

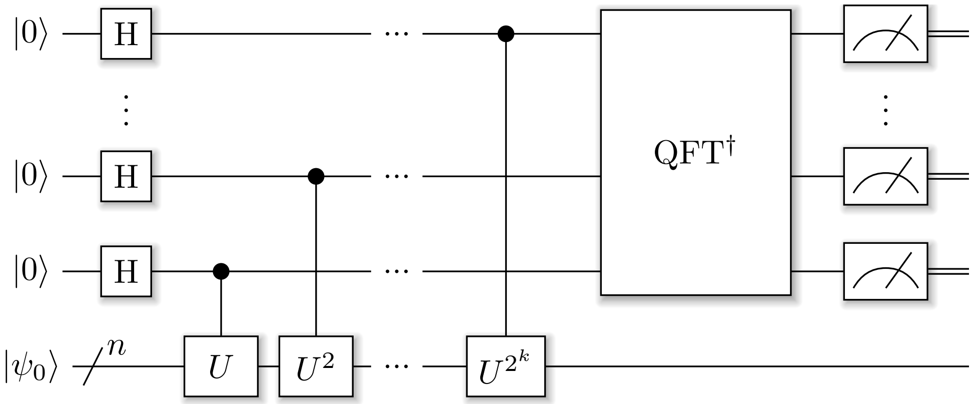


Figure 4.2: Quantum circuit of a quantum phase estimation applied to the time evolution operator $U = e^{-itH}$, where H is the Hamiltonian of the quantum system being simulated, e.g., a transverse-field Ising model as in the text. After the inverse quantum Fourier transform (QFT†), a measurement yields the phase which was picked up by the input state. For the ground state $|\psi_0\rangle$, this is $U|\psi_0\rangle = e^{-iHt}|\psi_0\rangle = e^{-iE_0t}|\psi_0\rangle$, allowing to extract (a $(k+1)$-bit approximation of) the energy E_0 of $|\psi_0\rangle$.

optimization problem which is of the form

$$\begin{aligned}&(\varepsilon^\star_{S_1}, \cdots, \varepsilon^\star_{S_N}) \in \arg\min C_{\text{Program}}(\varepsilon_{S_1}, \cdots, \varepsilon_{S_N})\\ &\text{such that}\ \ \varepsilon_{\text{Program}}(\varepsilon^\star_{S_1}, \cdots, \varepsilon^\star_{S_N}) \le \varepsilon\end{aligned}$$

for a user- or application-defined over-all tolerance ε, can be solved using a reasonable amount of resources. Moreover, the costs of optimization can be reduced by initializing the initial trial parameters ε_{S_i} to the corresponding solution accuracies of a lower-dimensional optimization problem where $\mathcal{S}(U, \varepsilon)$ had fewer distinct subroutines. This approach is similar to multi-grid schemes which are used to solve partial differential equations.

The second issue with a direct application of Lemma 4.2.1 is the varying number of optimization parameters, which is also resolved by Theorem 4.2.1. Of course one can simply make $\mathcal{S}(U, \varepsilon)$ tremendously large such that most of the corresponding $f_S(\varepsilon)$ are zero. This, however, is a rather inefficient solution which would also be possible when using Lemma 4.2.1 directly. A better approach is to inspect $\mathcal{S}(U, \varepsilon)$ for different ε and to then choose A auxiliary subroutine sets $S^a_1, ..., S^a_A$ such that each additional subroutine V^a_k which appears when changing ε (but is not a member of any S of the original $\mathcal{S}(U, \varepsilon)$) falls into exactly one of these sets. The original set $\mathcal{S}(U, \varepsilon)$ can then be extended by these auxiliary sets before running the optimization procedure. Again, the level of granularity of these auxiliary sets and thus the number of such sets A can be tuned according to the resources that are available to solve the resulting optimization problem.

4.3 Example application

As an example application, we consider the simulation of a quantum mechanical system called the *transverse-field Ising model* [77] (TFIM), which is governed by the Hamiltonian

$$\hat{H} = -\sum_{\langle i,j\rangle} J_{ij}\sigma_z^i\sigma_z^j - \sum_i \Gamma_i\sigma_x^i,$$

where J_{ij} are coupling constants and Γ_i denotes the strength of the transverse field at location i. σ_x^i and σ_z^i are the Pauli matrices, i.e.,

$$\sigma_x = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \quad \text{and} \quad \sigma_z = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}$$

acting on the i-th spin. The sum over $\langle i, j\rangle$ loops over all pairs of sites (i, j) which are connected. In our example, this corresponds to nearest-neighbor sites on a one-dimensional spin chain (with periodic boundary conditions) of length N. Given an approximation $|\tilde{\psi}_0\rangle$ to the ground state $|\psi_0\rangle$ of $\hat{H}$, we would like to determine the ground state energy E_0 such that

$$\hat{H}\,|\psi_0\rangle = E_0\,|\psi_0\rangle\,.$$

It is well-known that quantum phase estimation (QPE) can be used to achieve this task which leads to a general circuit structure as in Fig. 4.2.

Individual compilation stages. We now analyze the QPE algorithm for TFIM ground state estimation and the resulting optimization problem for approximation errors. First note that if the overlap between $|\psi_0\rangle$ and $|\tilde{\psi}_0\rangle$ is large, a successful application of QPE followed by a measurement of the energy register will collapse the state vector onto $|\psi_0\rangle$ and output E_0 with high probability (namely $p = |\langle\tilde{\psi}_0|\psi_0\rangle|^2$).

There are various ways to implement QPE [41], but the simplest to analyze is the coherent QPE followed by a measurement of all control qubits, see Fig. 4.2 for an illustration of the circuit. This procedure requires $16\pi/\varepsilon_{\text{QPE}}$ applications of (the controlled version of) the time-evolution operator $U_\delta = \exp(-i\delta\hat{H})$ for a success probability of $1/2$, where ε_{QPE} denotes the desired accuracy (bit-resolution of the resulting eigenvalues) [33]. Using a Trotter decomposition of U_δ, i.e., for large M

$$\begin{aligned}U_\delta &\approx \left(U^J_{\frac{\delta}{M}} U^\Gamma_{\frac{\delta}{M}}\right)^M \\ &= \left(e^{-i\frac{\delta}{M}\sum_i J_{i,i+1}\sigma_z^i\sigma_z^{i+1}} e^{-i\frac{\delta}{M}\sum_i \Gamma_i\sigma_x^i}\right)^M \\ &= \left(\prod_i e^{-i\frac{\delta}{M}J_{i,i+1}\sigma_z^i\sigma_z^{i+1}} \prod_i e^{-i\frac{\delta}{M}\Gamma_i\sigma_x^i}\right)^M ,\end{aligned}$$

allows to implement the global propagator U_δ using a sequence of local operations. These consist of z- and x-rotations in addition to nearest-neighbor CNOT gates to compute the parity (before the z-rotation and again after the z-rotation to uncompute the parity). The rotation angles are $\theta_z = 2\frac{\delta}{M}J_{i,i+1}$ and $\theta_x = -2\frac{\delta}{M}\Gamma_i$ for z- and x-rotations, respectively. The extra factor of two arises from the the definitions of the Rz and Rx gates, see Sec. 4.1.

In order to apply error correction to run the resulting circuit on actual hardware, these rotations can be decomposed into a sequence of Clifford+T gates using rotation synthesis. Such a discrete approximation up to an accuracy of ε_R features $\mathcal{O}(\log \varepsilon_R^{-1})$ T-gates if the algorithms in [73, 63] are used, where even the constants hidden in the $\mathcal{O}$ notation were explicitly determined.

Casting the example into our framework. The first compilation step is to resolve the QPE library call. In this case, it is known that the cost of QPE applied to a general propagator U is

$$C(\mathrm{QPE}_U, \varepsilon) = \frac{16\pi}{\varepsilon_{\mathrm{QPE}}} C({}^cU, \varepsilon_U),$$

where ${}^cU$ denotes the controlled version of the unitary U, i.e.,

$${}^cU := |0\rangle\langle 0| \otimes \mathbb{1} + |1\rangle\langle 1| \otimes U.$$

Furthermore, the chosen tolerances must satisfy

$$\frac{16\pi}{\varepsilon_{\mathrm{QPE}}}\varepsilon_U \leq \varepsilon - \varepsilon_{\mathrm{QPE}}.$$

The next step is to approximate the propagator using a Trotter decomposition. Depending on the order of the Trotter formula being used, this yields

$$\begin{aligned}C({}^cU, \varepsilon_U) &= M(\varepsilon_{\mathrm{Trotter}})(C({}^cU_1, \varepsilon_{U_1}) + C({}^cU_2, \varepsilon_{U_2})) \\ &\quad \text{with} \;\; M(\varepsilon_{\mathrm{Trotter}})(\varepsilon_{U_1} + \varepsilon_{U_2}) \leq \varepsilon_U - \varepsilon_{\mathrm{Trotter}}.\end{aligned}$$

In the experiments section, we will choose $M(\varepsilon_{\text{Trotter}}) \propto \frac{1}{\sqrt{\varepsilon_{\text{Trotter}}}}$ as an example. Finally, approximating the (controlled) rotations in ${}^cU_1$ and ${}^cU_2$ by employing rotation synthesis,

$$\begin{gathered} C({}^cU_i, \varepsilon_{U_i}) = 2N \cdot 4 \log \varepsilon_R^{-1} \\ \text{with } \; 2N\varepsilon_R \leq \varepsilon_{U_i} \; \text{ for } i \in \{1, 2\}. \end{gathered}$$

Collecting all of these terms and using that $C({}^cU_1, \cdot) = C({}^cU_2, \cdot)$ yields

$$\begin{aligned} C(\text{QPE}_U, \varepsilon) &= \frac{16\pi}{\varepsilon_{\text{QPE}}} M(\varepsilon_{\text{Trotter}}) \cdot 2 \cdot 2N \cdot 4 \log \varepsilon_R^{-1}, \\ \text{with } \varepsilon_{\text{QPE}} &+ \frac{16\pi}{\varepsilon_{\text{QPE}}}(2M(\varepsilon_{\text{Trotter}}) \cdot 2N\varepsilon_R + \varepsilon_{\text{Trotter}}) \leq \varepsilon. \end{aligned}$$

Note that this example is a typical application for a quantum computer which at the same time can serve as a proxy for other, more complex simulation algorithms.

While the individual compilation stages may be different for other applications, the basic principle of iterative decomposition and approximation during compilation is ubiquitous. In particular, a similar compilation procedure would be employed when performing quantum chemistry simulations, be it using a Trotter-based approach [78] or an approach that is based on a truncated Taylor series [79].

4.4 Implementation and numerical results

In this section, we present implementation details and numerical results of our error management module. While the optimization procedure becomes harder for fine-grained cost and error analysis, the benefits in terms of the cost of the resulting circuit are substantial.

Optimization methodology. We use a two-mode annealing procedure for optimization, in which two objective functions are reduced as follows: The first mode is active whenever the current overall error is larger than the target accuracy ε. In this case, it performs annealing until the target accuracy has been reached. At this point, the second mode becomes active. It performs annealing-based optimization to reduce the circuit cost function. After each such step, it switches back to the error-reduction subroutine if the overall error increased above ε.

Both annealing-based optimization modes follow the same scheme, which consists of increasing/decreasing a randomly chosen ε_i by multiplying/dividing it by a random factor $f \in (1, 1+\delta]$, where δ can be tuned to achieve an acceptance rate of roughly 50%. Then, the new objective function value is determined, followed by either a rejection of the proposed change in ε_i or an acceptance with probability

$$p_{\text{accept}} = \min(1, e^{-\beta \Delta E}),$$

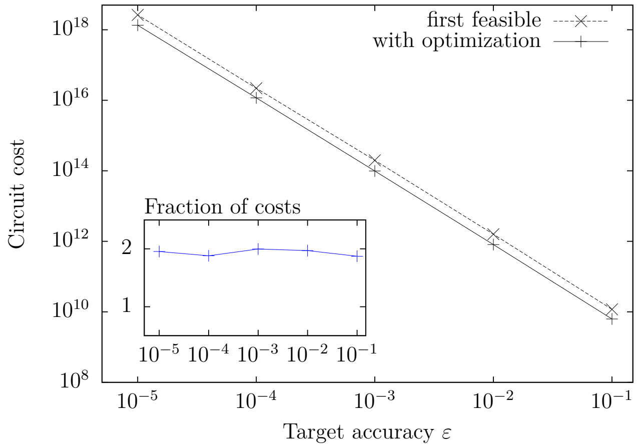
```
$\beta$ = 0
$\bar{\varepsilon}$ = $(0.1, 0.1, \cdots, 0.1)$
cost = get_cost($\bar{\varepsilon}$)
error = get_total_error($\bar{\varepsilon}$)
for step in range(num_steps):
  i = floor(rnd() * len(eps))
  old_$\bar{\varepsilon}$ = $\bar{\varepsilon}$
  if rnd() < 0.5:
    $\bar{\varepsilon}_i$ *= 1 + (1 - rnd()) * $\delta$
  else:
    $\bar{\varepsilon}_i$ /= 1 + (1 - rnd()) * $\delta$

  if error <= goal_error:
    # reduce cost
    $\Delta E$ = get_cost($\bar{\varepsilon}$) - cost
  else:
    # reduce error
    $\Delta E$ = get_total_error($\bar{\varepsilon}$) - error
  $p_{\text{accept}}$ = min(1, $e^{-\beta\Delta E}$)
  if rnd() > $p_{\text{accept}}$:
    $\bar{\varepsilon}$ = old_$\bar{\varepsilon}$
  $\beta$ += $\Delta\beta$
```

Listing 4.1: High-level description of the annealing-based algorithm to solve the resulting optimization problem. The actual implementation features different scaling constants for ΔE depending on the mode (error reduction vs. cost reduction).

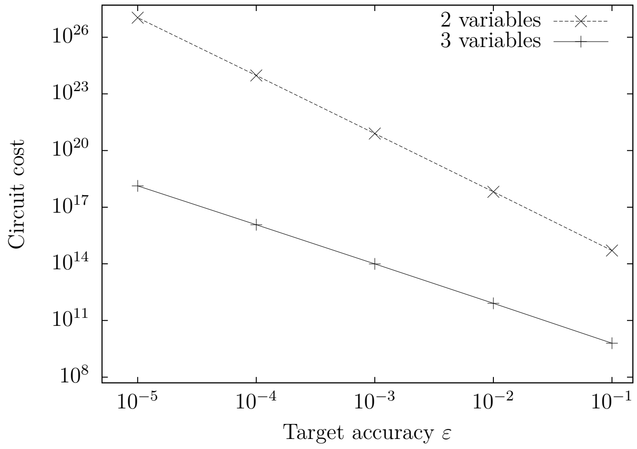
where $\beta = T^{-1}$ and T denotes the annealing temperature. This means, in particular, that moves which do not increase the energy, i.e., $\Delta E \leq 0$ are always accepted. The pseudo-code of this algorithm can be found in Listing 4.1.

Results. Using the example of a transverse-field Ising model which was discussed in Sec. 4.3, we determine the benefits of our error management module by running two experiments. The first experiment aims at assessing the difference between a feasible solution, i.e., values ε_i which produce an overall error that is less than the user-defined tolerance, and an optimized feasible solution. In the first case, we only run the first mode until a feasible solution is obtained and in the latter, we employ both modes as outlined above. Fig. 4.3a depicts the costs of the resulting circuit as a function of the desired overall accuracy ε.

The second experiment aims to show the benefit of an increased number of ε_i parameters in the same example. The difference between the circuit costs when using just two such parameters (i.e., setting $\varepsilon_R = \varepsilon_{\text{Trotter}}$) and using all three is depicted in Fig. 4.3b.



(a) Additional optimization allows to reduce the circuit cost by almost a factor of two over the first encountered feasible solution (see inset).



(b) Performing a three-variable optimization enables a reduction of the resulting circuit cost by several orders of magnitude when compared to two-variable optimization.

Figure 4.3: Numerical results for the optimization problem resulting from the transverse-field Ising model example discussed in Sec. 4.3. Improving the first encountered feasible solution by further optimization allows to reduce the cost by almost a factor of two and the number of different parameters can influence the resulting cost by several orders of magnitude.

Finally, we measure the robustness of the optimization procedure by introducing redundant parameters, i.e., additional rotation gate synthesis tolerances ε_{R_i},

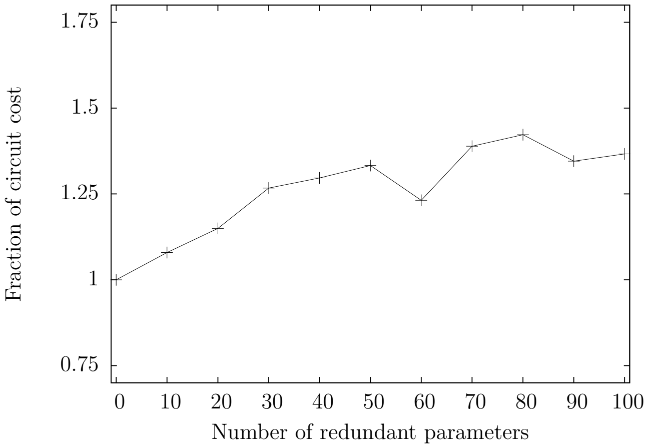


Figure 4.4: The circuit cost with $i \in \{0, 10, ..., 100\}$ redundant parameters divided by the cost achieved with no redundancies. As expected, the problem becomes harder to optimize as more parameters are added. The best result of 1000 runs (with different random number seeds) is reported. 5000 annealing steps to $\beta_{max} = 10$ were performed for each run.

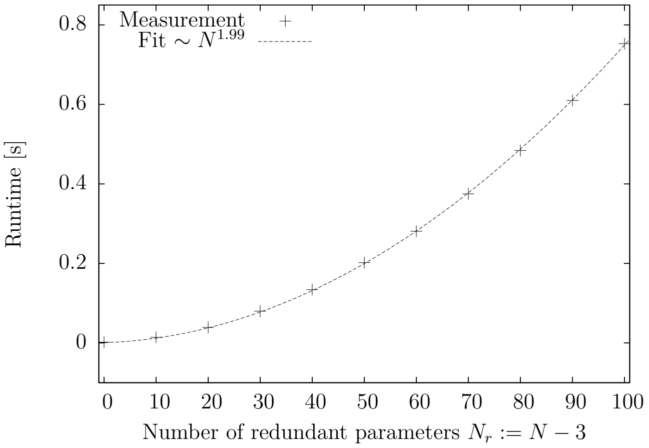


Figure 4.5: Runtime for finding an initial feasible solution which was then optimized further in order to reduce the circuit cost. As expected, the time increases quadratically with the number of parameters.

where the optimal choice would be $\varepsilon_R = \varepsilon_{R_i} = \varepsilon_{R_j}$ for all i, j. However, because the resulting optimization problem features more parameters, it is harder to solve and the final circuit cost is expected to be higher. In addition, the time it takes to find an initial feasible solution will grow. See Figs. 4.4 and 4.5 for the results which indicate that this approach is scalable to hundreds of variables if the goal is to find a feasible solution. However, as the number of parameters grows, it becomes increasingly harder to simultaneously optimize for the cost of the circuit. This could be observed, e.g., with 100 additional (redundant) parameters, where further optimization of the feasible solution reduced the cost from $1.65908 \cdot 10^{12}$ to $1.10752 \cdot 10^{12}$, which is far from the almost $2x$ improvement which was observed for smaller systems in Fig. 4.3a. Also, the scaling of the runtime in Fig. 4.5 can be explained since new updates are proposed by selecting $i \in [0, ..., N-1]$ uniformly at random (followed by either increasing or decreasing ε_i). Due to this random walk over $i \in [0, ..., N-1]$, the overall runtime is also expected to behave like the expected runtime of a random walk and, therefore, to be in $\mathcal{O}(N^2)$.

4.5 Conclusion

We have presented a methodology for managing approximation errors in compiling quantum algorithms. Given that the way in which the overall target error is distributed among subroutines greatly influences the resource requirements, it is crucial to optimize this process, in particular for large-scale quantum algorithms that are composed of many subroutines. Our scheme leverages an annealing-based procedure to find an initial feasible solution which then is optimized further.

Our scheme for error management only addresses errors that occur in approximations during the compilation process into a fault-tolerant gate set. Future work might include hardware errors, e.g., systematic over- or under-rotations of gates performed by the target device. Furthermore, additional numerical studies for various quantum algorithms can be performed in order to arrive at heuristics for choosing the number of optimization parameters. Moreover, building on our error management methodology, one can automate the entire process of resource estimation for certain subclasses of quantum algorithms. This would yield a useful tool for assessing the practicality of known quantum algorithms, similar to the analysis carried out manually in Ref. [33].

Part II

Improved simulation of quantum computers

Introduction to Part II

In chapter 4, the introduced methodology for managing approximation errors mainly employed error bounds which have been derived analytically. However, these bounds are often very loose when considering specific instances. To optimize bounds for these cases, numerical simulations can be employed [33, 34]. In particular, smaller instances can be simulated using classical (super)computers and the results can then be extrapolated to larger system sizes. By pushing the boundary of what is simulatable toward larger problem sizes, the quality of these predictive estimates can be improved.

To this end, we discuss two ways to extend the simulation capabilities of classical computers in part II of this thesis. In chapter 5, we employ one of the world's largest supercomputers *Cori II* at LBNL together with code optimization at every level of parallelism—instruction-level, thread-level, node-level, and cluster-level—allowing a reduction in time-to-solution over the state of the art by more than one order of magnitude.

In chapter 6, we introduce the concept of quantum circuit *emulation*. This new approach makes use of the mathematical description of the algorithm being simulated in order to take classical shortcuts. While this primarily leads to a reduction in run time and memory footprint, emulation can also be used to perform analyses of approximation errors as it offers a way to test against ideal, error-free implementations.

Chapter 5

0.5 petabyte simulation of a 45-qubit quantum circuit

This chapter is a slightly modified version of Ref. [4]. The gate scheduling optimizer (Sec. 5.1.6) was implemented by Damian S. Steiger.

Experimental devices featuring close to 50 qubits will soon be available and may be able to perform well-defined computational tasks which would classically require the world's most powerful supercomputers. Going even beyond these capabilities means entering the realm of *Quantum Supremacy* [52, 80]. While one of the computational tasks proposed to demonstrate this supremacy—the execution of low-depth random quantum circuits, see Fig. 5.1—is not scientifically useful on its own, running such circuits is still of great use to calibrate, validate, and benchmark near-term quantum devices [52].

Several implementations of quantum circuit simulators exist [81]. The massively parallel simulator from [82, 83] was used to simulate 42 qubits on the Jülich supercomputer in 2010, which set the new world record in number of simulated quantum bits. Recently, qHiPSTER [84] was specialized for the simulation of quantum supremacy circuits and then used to simulate these circuits up to 42 qubits [52].

In this chapter, we improve the strong scaling behavior of the compute kernels underlying quantum circuit simulation in order to reduce time-to-solution when employing multi- and many-core processors. In the multi-node domain, we employ a communication scheme similar to [83] and introduce an additional layer of optimization to reduce the amount of communication: We apply a clustering algorithm to the quantum circuit in order to improve the scheduling of quantum gate operations. While this pre-computation terminates in 1-3 seconds on a laptop, it greatly reduces the number of communication steps. We then simulate quantum supremacy circuits of various sizes and report speedups of over one order of magnitude at every scale. Finally, we simulate a 45-qubit quantum supremacy circuit

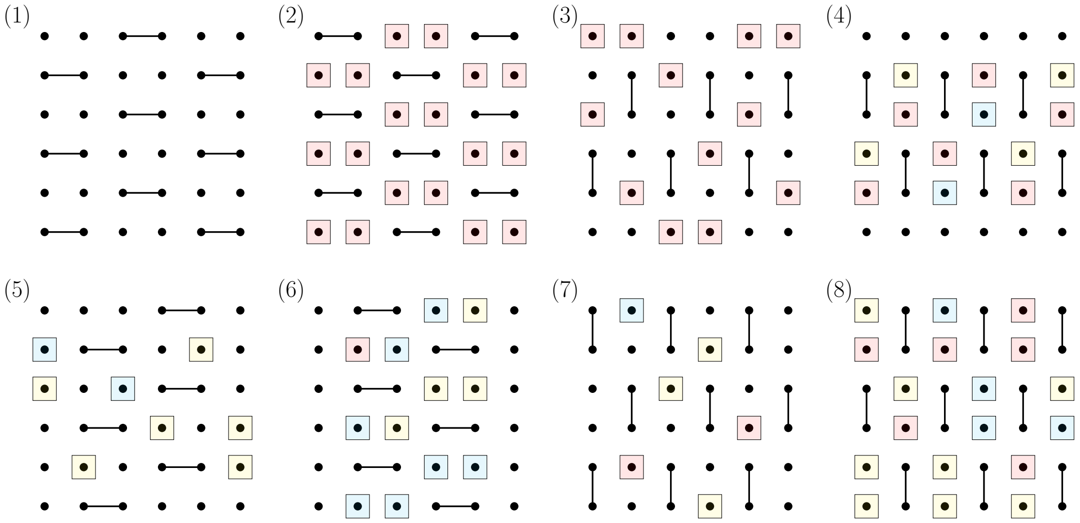
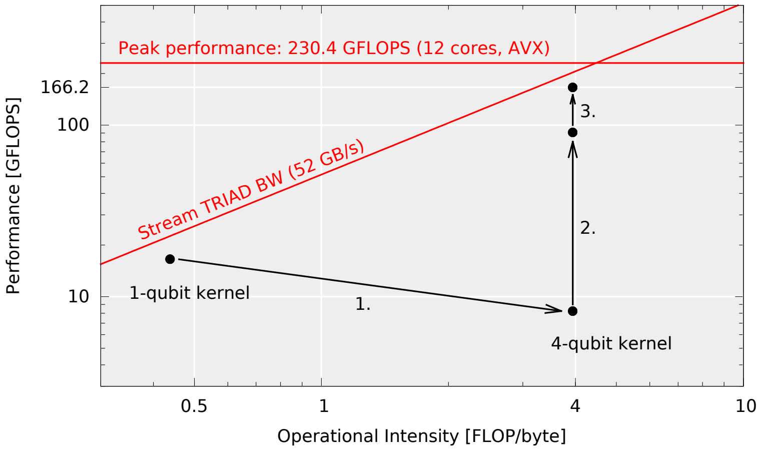
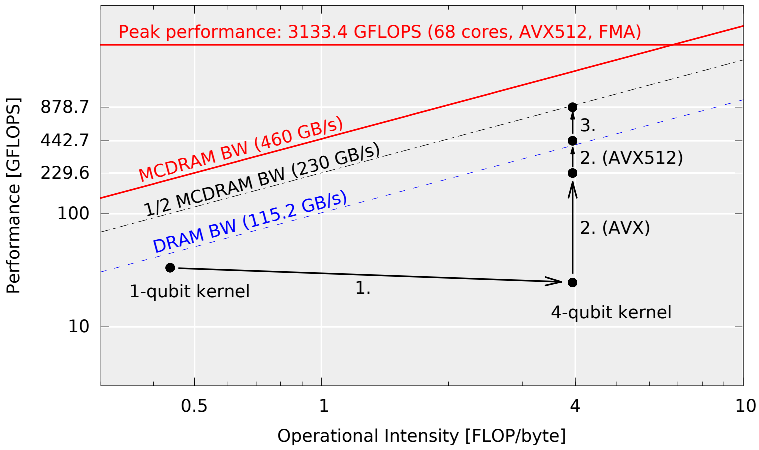


Figure 5.1: Low-depth random quantum circuit proposed by Google to show quantum supremacy [52]. We generated identical circuits using the following rules: At clock cycle 0, a Hadamard gate is applied to each qubit. Afterwards, eight different patterns of controlled Z (CZ) gates are applied repeatedly until the desired circuit depth is achieved. See the 8 different CZ patterns above in clock cycles 1-8 for a 6 × 6 qubit circuit, where the CZ gates are represented by a line between two qubits. This pattern ensures that all possible two qubit interactions on this 2D nearest neighbor architecture are executed every 8 cycles. In addition to the CZ gates, single qubit gates are applied to all qubits which in the previous cycle (but not in the current cycle) performed a CZ gate. The single qubit gates are randomly chosen to be either a T (red), $X^{1/2}$ (blue), or $Y^{1/2}$ (yellow) gate, except that the second single-qubit gate on each qubit (the first is the Hadamard gate in cycle 0) is always a T gate and when randomly choosing a single-qubit gate, it must be different from the previous single-qubit gate on that qubit.

on the Cori II supercomputing system using 0.5 petabytes of memory and 8,192 nodes. To our knowledge, this constitutes a new record in the maximal number of simulated qubits as of May 2017. The classical simulation of such circuits is believed to be impossible already for 49 qubits which, according to Ref. [85], is the threshold for quantum computers outperforming the largest supercomputers available today at the task of sampling from the output distribution of random low-depth quantum circuits. While we do not carry out a classical simulation of 49 qubits, we provide numerical evidence that this may be possible. Our optimizations allow reducing the number of communication steps required to simulate the entire circuit to just two all-to-alls, making it possible to use, e.g., solid-state



(a) Roofline plot for one Edison socket.



(b) Roofline plot for one KNL node of Cori II.

Figure 5.2: Roofline plots illustrating the performance improvements from Sec. 5.1.2 and 5.1.3. Step 1 introduces lazy evaluation, making the application more compute-bound. Step 2 adds explicit vectorization and instruction re-ordering, followed by step 3 which applies blocking for registers in addition to a pre-computation on the gate matrix, re-ordering and permuting the complex-valued matrix entries to improve the FLOP/instruction ratio. An additional optimization specific to KNL is the blocking for MCDRAM, which is introduced in step 1.

drives if the available memory is less than the 8 petabytes required.

5.1 Optimizations

Our simulator was implemented and optimized using a layered approach. The first layer aims to improve the single-core performance of our quantum gate kernels by employing explicit vectorization using compiler intrinsics, instruction reordering, and blocking to reduce register-spilling. The second layer uses OpenMP to enable a good strong scaling behavior on an entire node. The third and final layer implements the inter-node communication scheme using MPI. This allows to simulate up to 45 qubits on current supercomputers, in addition to reducing the time-to-solution when executing quantum circuits featuring fewer qubits.

5.1.1 Standard optimizations

In order to be able to simulate large systems, it is important not to actually store the $2^n \times 2^n$ matrix acting on the state vector. Instead, one can exploit its regular structure and implement methods which, given the state vector, mimic a multiplication by this matrix. A standard implementation features two state vectors (one input, one output). To determine one entry of the output vector, two complex multiplications and one complex addition have to be carried out on two entries of the input vector when applying a general single-qubit gate. In total, there are thus

$$2 \cdot (4[\text{mul}] + 2[\text{add}]) + 2[\text{add}] = 14 \text{ FLOP}$$

per complex entry of the output state vector. One complex double-precision entry requires 16 bytes of memory and the input vector has to be loaded from memory and the output vector has to be written back to memory. The operational intensity is therefore less than 1/2, which shows that this application is memory-bandwidth bound on most systems.

5.1.2 Single-core

In order to reduce the memory requirements by a factor of 2x, this complex sparse matrix-vector multiplication can be performed in-place, at the cost of a cache-unfriendly access pattern. Moreover, k-qubit gates require more operations for larger k, allowing to better utilize hardware with strong compute capabilities. In fact, the number of operations grows exponentially with k, since applying a k-qubit gate amounts to performing one scalar product of dimension 2^k per (output) entry.

To apply a k-qubit gate (of dimension $2^k \times 2^k$) to a state vector of size 2^n, where n denotes the number of qubits, the entries corresponding to all 2^k indices of the gate matrix have to be loaded into a 2^k-sized temporary vector, which then gets multiplied by the matrix before it is written back to the state vector. The indices of these state vector entries, when represented in binary, are bit-strings of the form $c_{n-k-1}x_{i_{k-1}}...c_j...x_{i_1}...c_0$, where $i_0, i_1, ..., i_{k-1}$ denote the k qubits indices to which the gate is being applied. Extracting and combining the bits x_{i_j} from the index of an entry, i.e.,

$$x = x_{i_{k-1}}...x_{i_1}x_{i_0} \ ,$$

yields the index of this entry with respect to the temporary vector. All 2^k entries which have an identical $c = c_{n-k-1}c_{n-k-2}...c_0$ index substring are part of this matrix-vector multiplication. Once all entries have been gathered, multiplied by the matrix, and stored back into the state vector, the next $c' = c + 1$ index substring can be dealt with. In total, this amounts to performing 2^{n-k} complex matrix-vector multiplications of dimension 2^k.

A first observation is that the same matrix is used 2^{n-k} times. One can thus permute the matrix entries before-hand in order to always have sorted qubit indices, which results in memory accesses to occur in a more local fashion.

When applying the matrix-vector product, doing so in the usual manner, i.e.,

$$\tilde{v}_l = \sum_{i=0}^{2^k-1} m_{l,i}v_i \ ,$$

would require all entries of the temporary vector v to be in register (and already loaded from memory). In order to address this issue, we employ blocking of the computation and determine the block size using an automatic code-generation / benchmarking feedback loop. For each block index $b = 0, 1, ..., \frac{2^k}{B} - 1$, all indices l of the temporary output vector $\tilde{v}$ are updated according to

$$\tilde{v}_l \mathrel{+}= \sum_{j<B} m_{l,i(b,j)}v_{i(b,j)} \ ,$$

where $i(b, j) = b \cdot B + j$, before moving on to the next block.

We employ explicit vectorization to parallelize updates for consecutive values of l. Since we are dealing with complex double-precision values, this theoretically allows to speed up the execution by a factor of $2x$ or even $4x$ when using AVX or AVX512, respectively. Denoting by a_R and a_I the real and imaginary parts of a, respectively, we now inspect the update above more closely. Multiplying one complex entry $v_l = (v_R, v_I)$ of the temporary vector v with one complex entry of the gate matrix $m = (m_R, m_I)$ and summing the result into the temporary output vector $\tilde{v}$ can be written as follows:

$$(\tilde{v}_R, \tilde{v}_I) \mathrel{+}= (v_R \cdot m_R - v_I \cdot m_I, v_I \cdot m_R + v_R \cdot m_I) \tag{5.1}$$

Yet, implementing this update results in wasted compute resources due to artificial dependencies and additional permutes. However, these instructions can be reordered as follows

$$(\tilde{v}_R, \tilde{v}_I) \mathrel{+}= (v_R \cdot m_R, v_I \cdot m_R) \tag{5.2}$$

$$(\tilde{v}_R, \tilde{v}_I) \mathrel{+}= (v_I \cdot -1 \cdot m_I, v_R \cdot m_I) \tag{5.3}$$

in order to increase the maximal achievable performance. Namely, having both (m_R, m_R) and $(-1 \cdot m_I, m_I)$ available, this update requires only two fused multiply-accumulate instructions instead of several individual multiplications, additions, and permutations. This is an improvement in both FLOP/instruction and FLOP/-FMA ratios.

Note that v_l can be permuted once upon loading (and then kept in register), as it is re-used for 2^k such complex multiplications. Also, since the matrix m is used in 2^{n-k} matrix-vector multiplications, the pre-computation to build up these two matrices consisting of (m_R, m_R) and $(-1 \cdot m_I, m_I)$ is essentially free.

5.1.3 Single-node

The optimizations discussed above do not change the fact that the operational intensity for applying a 1-qubit gate is very low, making it harder to fully utilize the power of multi- and manycore processors (see, e.g., Fig. 5.10). Yet, as mentioned previously, applying a k-qubit gate requires more operations for larger values of k and as long as the application remains memory bound, larger gates can be applied in (almost) the same amount of time. The benefit—besides increased operational intensity—is that larger gates can be used to execute an entire sequence of single- and two-qubit gates at once. In particular, multiple gates acting on k different qubits can be combined into one large k-qubit gate.

Which value of k to choose depends on the peak performance, the memory-bandwidth, the cache-size & associativity of the system, and the circuit to simulate. The cache specifications are important especially when gates are applied to qubits with larger indices, which cause memory access strides of large powers of two. For low-associativity caches, this causes conflicts to arise already for small kernel sizes. Since 2^k values need to be loaded from the state vector (which are at least 2^m apart, where m is the lowest qubit index) for each of the 2^{n-k} matrix-vector multiplications, a 2^k-way cache should map the corresponding cache-lines to different locations, no matter how large m is. This allows to directly access these values from cache for the next matrix-vector multiplication. See Fig. 5.6 and Fig. 5.9 for experimental results.

Finally, these k-qubit gate kernels are parallelized using OpenMP with NUMA-aware initialization of the state vector to ensure scaling beyond 1 NUMA node. Depending on the qubits to which the gate is applied, the outer-most loop may perform very few iterations, prohibiting a good strong scaling behavior. The OpenMP `collapse` directive remedies this problem.

Please see Fig. 5.2a and Fig. 5.2b, which show the improvements in performance when applying all mentioned optimizations and running the kernels on one socket of Edison or Cori II, which feature one 12-core Intel® Xeon® Processor E5-2695 v2 and one 68-core Intel® Xeon Phi™ Processor 7250 (KNL), respectively.

5.1.4 Multi-node

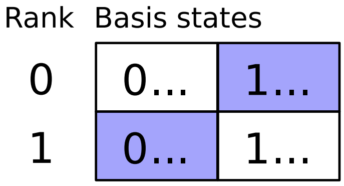
The simulation of quantum computers featuring many more than 30 qubits requires multiple nodes in order for the state vector to fit into memory. We use MPI to communicate between 2^g nodes, each node having its own state vector of size 2^l, where g and l denote the number of global and local qubits, respectively. Gate operations on local qubits, i.e., qubits with index $i < l$, require no communication. Qubits with index $i \geq l$, on the other hand, do require communication.

There are two basic schemes which can be used to perform multi-node quantum circuit simulations. The first [82] keeps global qubits global and applies global gates by employing 2 pair-wise exchanges of half the state vector. The second scheme [83] swaps global qubits with local ones, applies gates to local qubits in the usual fashion and, if need be, swaps them again with global qubits. Note that swapping in a global qubit and then immediately swapping it back out requires the same amount of communication as the first scheme. We thus expect the global-to-local scheme to perform better and focus on this scheme.

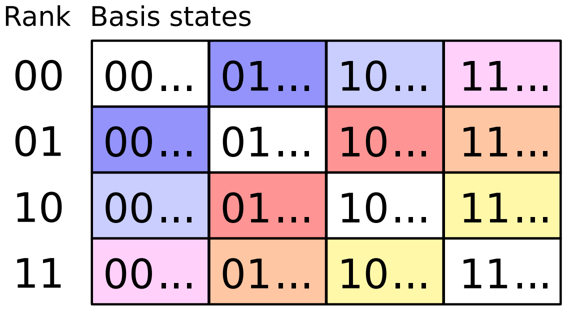
1-Qubit Example (see Fig. 5.3a). For the case of two ranks, swapping the highest-order qubit (highest bit in the local index) with the global qubit (first bit of the rank number) can be achieved as follows: The first block of rank 0 remains unchanged, since swapping 0 with 0 has no effect. Swapping 0 (global) and 1 (local) for the second block requires sending the entire block to rank 1, where these coefficients are associated with the local qubit being 0. Proceeding in this manner results in an exchange of the colored blocks, which is equivalent to an all-to-all.

2-Qubit Example (see Fig. 5.3b). To swap two global qubits with the two highest-order local qubits for the case of four ranks, each rank sends its i-th quarter of the state vector to rank number i. Therefore, all identically-colored state vector parts are exchanged, which results again in one all-to-all.

Additionally, as done in [83], we generalize this scheme to swap multiple or even all global qubits with local ones. Yet, in contrast to [83], we do not iteratively copy out parts of the state vector and carry out the pair-wise exchanges manually. Instead, we employ higher-level abstractions to achieve the same task, with the



(a) Single-qubit swap.



(b) Two-qubit swap.

Figure 5.3: Illustration of a single- and multi-qubit global-to-local swap using one (group-) all-to-all. The blocks labeled, e.g., 01... represent the coefficients corresponding to the global basis state which starts with the bit-string $r01$, where r is the bit-representation of the rank (see text).

benefit that optimized implementations for, e.g., specific network topologies are likely to be already available. A q-qubit global-to-local swap, which exchanges q global with q local qubits, can be achieved using 1 group-local all-to-all for each of the 2^{g-q} groups of processes. Therefore, turning all global qubits into local ones amounts to executing one all-to-all on the `MPI_COMM_WORLD` communicator. This allows swapping the k qubits with highest local index with k global ones. In order to allow for arbitrary local qubits to be exchanged, we first use our optimized kernels to achieve local swaps between highest-index qubits and those to be swapped. We then perform the group-local all-to-all and, if need be, another local swap (with lower-index qubits) in order to improve data locality in our k-qubit gate kernels.

5.1.5 Global gate specialization

While a general global gate always requires communication, there are a few common ones which do not. Examples include the controlled-NOT gate (or controlled-X) which, when applied to global qubits, causes merely a re-numbering of ranks.

The (diagonal) controlled-Z gate either turns into a conditional global phase or a local Z-gate which, depending on the rank, is executed or not. Finally, the T-gate is also diagonal and results in a global phase, which can be absorbed into the next gate matrix to be applied. Making use of such insights allows to further reduce the number of global-to-local swaps without increasing the amount of computation performed locally.

For 36-qubit quantum supremacy circuits, this optimization enables a reduction of the required communication by another factor of $2x$: Only one global-to-local swap is required to run the entire depth-25 circuit. For 42- and 45-qubit circuits, 2 global-to-local swaps are necessary, whereas 3 are required without gate specialization.

5.1.6 Circuit optimizations: Gate scheduling and qubit mapping

In addition to performing implementation optimizations, also the circuit requires optimization in order to reduce the number of communication steps and to use our highly-tuned kernels in a more efficient manner. We will demonstrate the different optimizations for gate scheduling and qubit mapping using the quantum supremacy circuits from Ref. [52], for which we also present performance results in the next section. Our optimizations are general and can be applied to any quantum circuit. In fact, these quantum supremacy circuits happen to be designed in a way that is least suitable for these kinds of performance optimizations. We thus expect even larger improvements when employing these techniques for the simulation of other circuits.

The construction of these random, low-depth quantum circuits is shown in Fig. 5.1. These circuits are designed to be run on a quantum computer architecture featuring a 2D nearest-neighbor connectivity graph. By design, all possible two qubit gates are applied within 8 cycles, which makes the system highly entangled. Note that a simulator can skip the initial Hadamard gates in cycle 0 and initialize the wave function directly to $(2^{-\frac{n}{2}}, ..., 2^{-\frac{n}{2}})^T$, instead of starting in state $|0...0\rangle = (1, 0, ..., 0)^T$. Furthermore, we do not simulate the final CZ gates as they only alter the phases of the probability amplitudes α_i, but not the probabilities $p_i = |\alpha_i|^2$ which we are interested in.

Gate scheduling

The most important optimization on the quantum circuit is gate scheduling, as it drastically reduces the amount of communication in the multi-node setting and also the number of k-qubit gate kernels on the single-node level. The optimizations can be broken into three steps:

1. Minimize number of communication steps In a first optimization step, gate scheduling minimizes the number of global-to-local swaps which is the most important parameter in the multi-node setting. Executing every clock-cycle of the circuit on its own requires at least one communication step for every cycle which features a non-diagonal global gate.

However, as explained in the multi-node strategy, it is beneficial not to execute those global gates but rather swap global qubits with local qubits and then execute these gates locally. In order for this scheme to be most beneficial, the gate scheduling algorithm reorders (if possible) the gates into stages, where each stage consists of a sequence of quantum gates acting only on local qubits, see Fig. 5.4. Gates acting on the same qubit never commute for quantum supremacy circuits by design, making classical simulation harder. Nevertheless, we can reorder gates which act on different qubits as they commute trivially. After completing a stage, some local qubits are swapped with global qubits, and a new stage is started. This scheme reduces the number of communication steps significantly. A depth-25 42-qubit supremacy circuit requires only two global-to-local swaps, see Fig. 5.5b. An important feature of our gate scheduling algorithm is that the number of global-to-local swaps is mostly independent of the number of local qubits (29, 30, 31, or 32). This allows for a good strong scaling behavior. Fig. 5.5a shows how the number of global-to-local swaps behaves as a function of circuit depth.

We decided to always swap global qubits with the lowest-order local qubits to arrive at an upper bound for the number of communication steps required. In addition, we apply a cheap search algorithm to find better local qubits to swap with. In case of a 36-qubit supremacy circuit, this results in a $2x$ reduction in the number of global-to-local swaps, from two swaps to just one. Note that our stage-finding algorithm assumes the worst-case scenario, in which all randomly picked global single-qubit gates are dense, meaning that we cannot apply our gate specialization for T gates to reduce the amount of communication.

2. Minimize number of k-qubit gates In a second step, we schedule all the gates within a stage such that we can merge sequences of consecutive 1- or 2-qubit gates into a k-qubit gate and execute this k-qubit gate instead of many single- and two-qubit gates. See Fig. 5.4, which shows how such a cluster with $k = 3$ can be built. We greedily try to increase the number of qubits k within a cluster while still maintaining the condition that $k \leq k_{max}$, where k_{max} is the largest k for which the k-qubit gate kernel still shows good performance on the target system. To reduce the over-all number of clusters, we perform a small local search in order to build the largest cluster with gates not yet assigned, before assigning the remaining gates to new clusters. We summarize the required number of clusters to execute a quantum supremacy circuit in Table 5.1. Clearly, even for these circuits, more

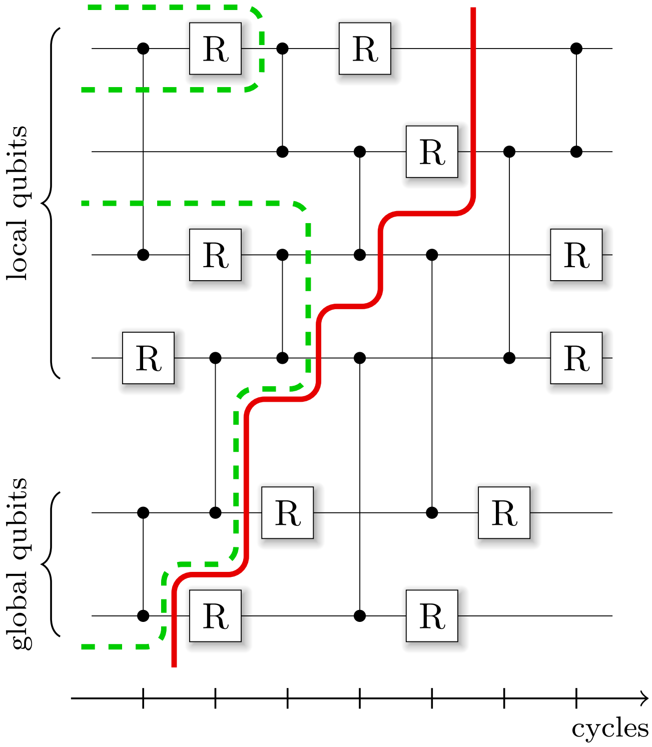
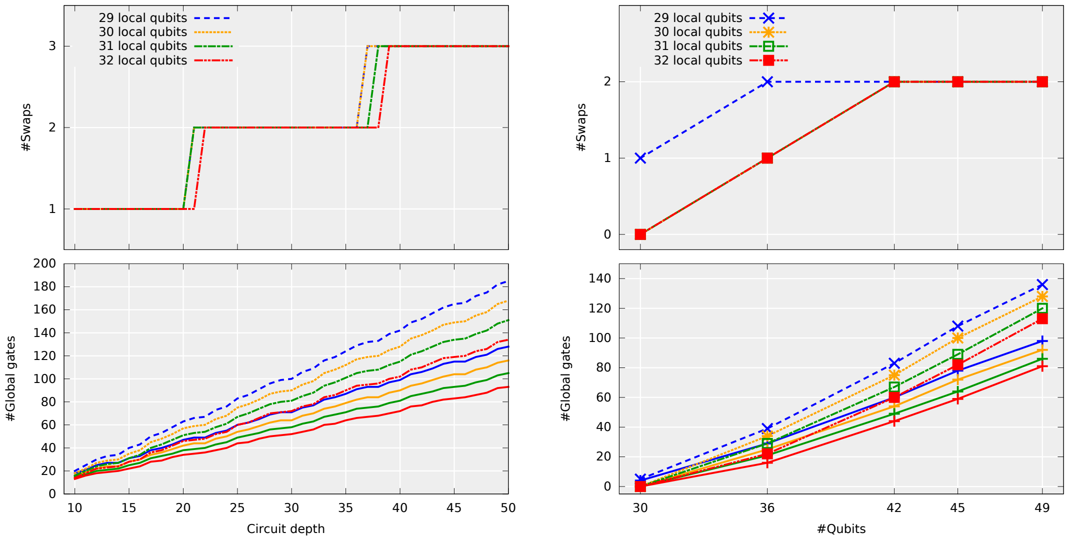


Figure 5.4: Example of gate scheduling for a circuit with CZ gates and dense single-qubit rotations gates (R). Note that we use gate specialization for CZ gates, which means we can apply them without communication on global qubits. First, instead of applying the gates cycle by cycle, we identify the largest first stage of gates which can be applied without communication. These are all the gates on the left of the solid red line. Second, we schedule the gates within a stage into clusters. For example, we can combine all the gates on the left of the dashed green line into one 3-qubit gate instead of applying 7 individual gates.

than k gates can be merged into one k-qubit cluster on average.

3. Local adjustments of global-to-local swaps The last cluster within each stage tends to contain a lower number of single- and two-qubit gates. In order to increase the average number of gates in each cluster and thereby decrease the total number of clusters in the circuit, we try to remove the last clusters of each stage by performing the global-to-local swap earlier if this is possible without increasing the total number of global-to-local swaps.



(a) Scaling of the required communication for circuit depths 10 to 50 for 42-qubit quantum supremacy circuits.

(b) Scaling of the required communication for quantum supremacy circuits with a fixed circuit depth of 25.

Figure 5.5: Scaling of the required number of communication steps for quantum supremacy circuits as a function of circuit depth (a) or number of qubits (b). The lower two panels show the number of global gates which require communication if executed individually as in Ref. [52]. In contrast, the top two panels show the number of global-to-local swaps required to execute the full circuit when using our strategy of reordering gates and swapping global with local qubits. Note that one global-to-local swap (of all global qubits) requires the same amount of communication as one global gate. Averaged over the different global qubits, executing a dense global gate takes approximately 1/2 of the time required to swap all global qubits with local qubits, because applying a dense gate to low-order global qubits is faster due to the increased locality of the communication, see Ref. [52]. Note that the dashed lines are for worst case instances (only dense random gates on global qubits) and solid lines are for median hard instances, which we only consider in the two lower panels.

Qubit mapping

Last, the bit-location of each qubit is optimized in order to reduce the number of clusters experiencing the performance decrease resulting from the set-associativity of the last-level cache. Since this performance decrease only occurs if the gate is applied to high-order bit-locations, this can be achieved by remapping. The following heuristic allowed for a $2x$ decrease in time-to-solution:

Assign the qubit to bit-location 0 such that the number of clusters accessing

Number of Qubits	Number of Gates	Number of clusters		
		$k_{max} = 3$	$k_{max} = 4$	$k_{max} = 5$
30	369	82	46	36
36	447	98	53	41
42	528	111	58	46
45	569	111	73	51

Table 5.1: Re-scheduling of gates for depth-25 quantum supremacy circuits into clusters (using 30 local qubits). Clusters are built to contain $k \leq k_{max}$ qubits using a heuristic which tries to maximize the number of gates merged into one cluster. Clearly more than k_{max} individual gates can be combined into one single cluster on average. These optimizations take less than 3 seconds using Python and can be reused for all instance of the same size.

bit-location 0 is maximal. From now on, ignore all clusters which act on this qubit and assign bit-locations 1, 2, and 3 in the same manner. Bit locations 4, 5, 6, and 7 are assigned the same way, except that after each step, only clusters acting on two of these four bit-locations are ignored when assigning the next higher bit-location. For non-random circuits, it would pay off to perform a few local swaps between some bit-locations over the course of the algorithm, in order to maximize the number of clusters acting on low-order qubits.

5.2 Implementation and results

All optimizations mentioned in the previous sections were implemented in C++, except for the code generator for the k-qubit kernels and the circuit scheduler/qubit mapper, which were both implemented in Python.

5.2.1 Cori II

We performed simulations of quantum supremacy circuits featuring 30, 36, 42, and 45 qubits on the Cori II system at the Lawrence Berkeley National Laboratory (LBNL). Cori II consists of 9,304 single-socket compute nodes, each containing one 68-core Intel® Xeon Phi™ Processor 7250 (KNL) at 1.40GHz. The nodes are interconnected by a Cray Aries high speed "dragonfly" [86] topology interconnect

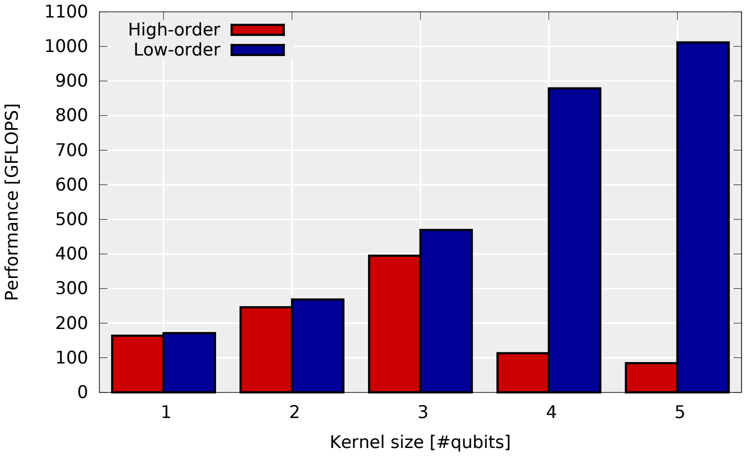


Figure 5.6: Decrease in performance when applying k-qubit gate kernels to qubits with large indices (high-order qubits) as opposed to low indices (low-order qubits). These experiments were run on all 68 cores of a Cori II KNL node. As mentioned in Sec. 5.1.3, this performance drop occurs when 2^k is larger than the set-associativity of the last-level cache. While the L2-cache is 16-way set-associative, it is shared between 2 cores.

and offer a combined theoretical peak performance of 29.1 PFLOPS and 1 PB of aggregate memory.

Node-level performance

These experiments were run on a single 68-core Intel® Xeon Phi™ Processor 7250 (KNL) node of the Cori II supercomputing system in the quad/cache setting. For $k \in \{1, 2, 3\}$-qubit gate kernels, four threads per core were used, as this resulted in the best performance. For $k = 4$ and $k = 5$, the best performance was achieved when using two and one thread per core, respectively. As mentioned in Sec. 5.1.3, the set-associativity of caches plays a crucial role in the performance of these k-qubit gate kernels. In particular, we find the theoretical predictions from Sec. 5.1.3 to agree perfectly with observations, see Fig. 5.6. The strong scaling behavior of executing one k-qubit gate kernel on a state vector of 28 qubits can be seen in Fig. 5.7.

Multi-node performance

The strong scaling of our simulator for a 36- and 42-qubit quantum circuit running on $\{16, 32, 64\}$ and $\{1024, 2048, 4096\}$ KNL nodes of Cori II, respectively, is depicted in Fig. 5.8. Following these scaling experiments, we ran a 45-qubit quan-

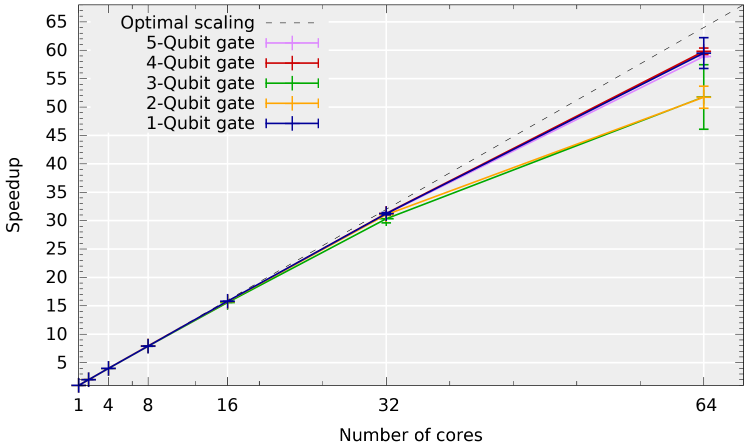


Figure 5.7: Strong scaling for applying k-qubit kernels to a 28-qubit system using 2^p, $p \in \{0, ..., 6\}$ cores of the 68-core Intel® Xeon Phi™ Processor 7250 and 4, 2, and 1 OpenMP thread(s) per core for $k \leq 3$, $k = 4$, and $k = 5$, respectively.

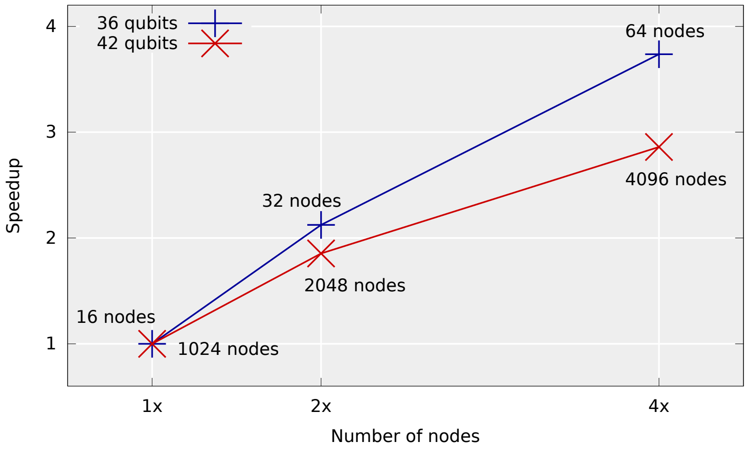


Figure 5.8: Strong scaling of our simulator running a 36- and 42-qubit quantum supremacy circuit on $\{16, 32, 64\}$ and $\{1024, 2048, 4096\}$ nodes of Cori II, respectively.

tum supremacy circuit using 8, 192 KNL nodes and a total of 0.5PB of memory. To our knowledge, this is the largest quantum circuit simulation ever carried out. Averaged over the entire simulation time (i.e., including communication time), this simulation achieved 0.428 PFLOPS. There are two reasons for this drop in performance. First, the time spent in communication and synchronization is 78%, and overlaying computation and communication would not improve this behavior due to the low k-qubit gate times (less than 1 second).

#Qubits	#Gates	#Nodes	Time [s]	Comm.	Speedup
6×5	369	1	9.58	0%	$14.8x$
6×6	447	64	28.92	42.9%	$12.8x$
7×6	528	4096	79.53	71.8%	$12.4x$
9×5	569	8192	552.61	78.0%	N/A

Table 5.2: Results for all simulations carried out on Cori II. Circuit simulation time and speedup are given with respect to the depth-25 quantum supremacy circuit simulations performed in [52]. The column Comm. gives the percentage of circuit simulation time spent in communication and synchronization.

Second, the performance of our kernels suffers in the regime where only few k-qubit gates are applied before a global-to-local swap needs to be performed. This is due to the fact that blocking for MCDRAM requires a sequence of several gates acting on qubits below bit-location 29. While our mapping procedure aims to maximize this number, the total number of gates being applied is not large enough. Yet, this is mainly due to the artificial construction of random circuits and does not occur in actual quantum algorithms, where interactions remain local over longer periods of time. As our 4-qubit gate kernel achieves 1/2 of the MCDRAM bandwidth which corresponds to roughly $2x$ the bandwidth of DRAM (see Fig. 5.2b), we expect a $2x$ drop in performance if memory requirements exceed the MCDRAM size of 16GB. Averaging the performance of our k-qubit kernels in Fig. 5.6 and including this $2x$ reduction yields approximately 250 GFLOPS per node. In total, we thus expect a performance of $22\% \times 8,192 \times 250$ GFLOPS ≈ 0.45 PFLOPS, which agrees with the measurement results given that we also apply a few 3- and 2-qubit gate kernels for left-over gates.

For a summary of all runs carried out on Cori II, see Table 5.2. Our implementation for, e.g., 42 qubits behaves as expected from Fig. 5.5a: For a depth-25 circuit, the communication scheme used in [52] requires about 50 global gates, while our simulator performs 2 global-to-local swaps (of all global qubits). Including the fact that one such global-to-local swap requires the same amount of communication and that, averaged over all global qubits, a global gate is $2x$ faster than if it is applied to the highest-order global qubit due to the network bisection bandwidth (see [52]), yields a reduction in communication of

$$\frac{50x}{2 \cdot 2} = 12.5x ,$$

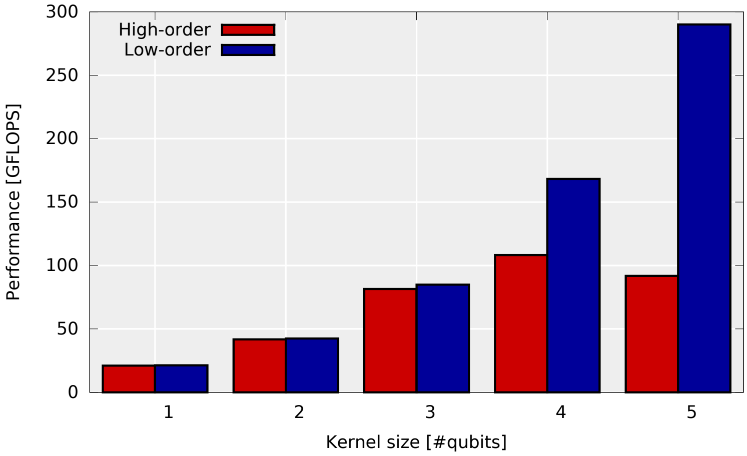


Figure 5.9: Performance decrease when k-qubit gate kernels are applied to high-order qubits instead of low-order ones on a two-socket Edison node. The findings again correspond to the set-associativity of the caches, which is $2^3 = 8$ in this case. For $k \leq 3$, there is only a negligible drop in performance, since all 2^k entries are mapped to different locations in the cache and the next 2^k-sizes matrix-vector multiplication can access the next 2^k values directly from cache, see Sec. 5.1.3

and since we achieve a similar reduction in time-to-solution for the circuit simulation on each node, this is also the expected overall speedup.

5.2.2 Edison

In order to be able to compare our results directly to [52], we also ran 30- and 36-qubit quantum supremacy circuits on the Edison system, also at LBNL. We used up to 64 sockets, each featuring a 12-core Intel® Xeon® Processor E5-2695 v2 at 2.4GHz. The $5,586$ 2-socket Edison nodes are interconnected by a Cray Aries “dragonfly” [86] topology interconnect and the theoretical peak performance of the entire system is 2.57 PFLOPS.

Node-level performance

The performance reduction from applying gates to high-order qubits due to the 8-way set-associativity of the L1- and L2-caches in Intel® Ivy Bridge™ processors can be seen in Fig. 5.9. These experiments were run on an entire two-socket node on all 24 cores with one OpenMP thread per core and using AVX vectorization.

The strong scaling of these k-qubit kernels with respect to the number of cores is depicted in Fig. 5.10. While the 5-qubit gate kernel scales best to the full node, the performance drop when applying it to high-order qubits is much greater than

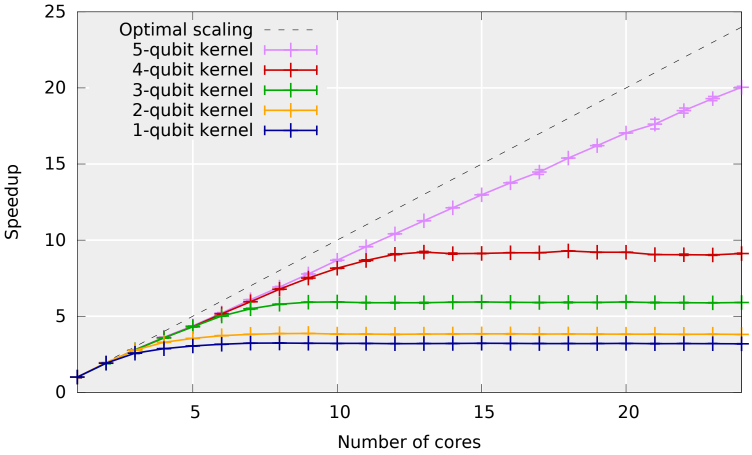


Figure 5.10: Strong scaling of the k-qubit kernels using up to 24 cores of a two-socket Edison node, which features one 12-core Intel® Ivy Bridge™ processor per socket. Up to and including $k = 4$, the kernels are memory bandwidth limited. This in combination with Fig. 5.9 suggests that $k = 4$ is the best kernel size to use on this system (with 1 MPI process per socket).

it is for 4-qubit gates. In addition, the 4-qubit gate kernel scales nearly perfectly to all 12 cores of a single socket, which suggests to use 2 MPI processes per node in the multi-node setting.

Running a single-socket simulation of a 30-qubit quantum supremacy circuit yields an improvement in time-to-solution by $3x$.

Multi-node performance

In order to compare the present work directly to the state-of-the-art simulator in [52], we performed a simulation of a 36-qubit quantum supremacy circuit using identical hardware: 64 sockets of the Edison supercomputer. We calculated the entropy of a depth-25 quantum supremacy circuit in 99 seconds, of which 90.9 seconds were spent in actual simulation and the remaining 8.1 seconds were used to calculate the entropy, which requires a final reduction. This constitutes an improvement in time-to-solution of over $4x$ and indicates that the obtained speedups were not merely a consequence of a new generation of hardware.

The kernels perform at an average of 47% theoretical peak, or 218 GFLOPS on every node during the execution of a 36-qubit quantum supremacy circuit. When including communication time, the entire simulation achieved 30% of the theoretical peak performance of 64 Edison sockets, which is 4.4 TFLOPS.

5.3 Conclusion

We demonstrated simulations of up to 45 qubits using up to $8,192$ nodes. With the same amount of compute resources, the simulation of 46 qubits is feasible when using single-precision floating point numbers to represent the complex amplitudes. The presented optimizations are general and our code generator improves performance portability across a wide range of processors. Extending the range of the code generator to the domain of GPUs is an ongoing project. Additional optimizations on the quantum circuit description reduced the required communication by an order of magnitude. As a result, the simulation of a 49-qubit quantum supremacy circuit would require only two global-to-local swap operations. While the memory requirements to simulate such a large circuit are beyond what is possible today, the low amount of communication would allow to use, e.g., solid-state drives.

Our simulation approach depends linearly on the number of gates being simulated. For quantum supremacy circuits of low depth, however, different simulation methods are capable of handling more qubits [29, 31], at least if the two-qubit gate is chosen to be sparse.

Subsequent updates to the specifications of quantum supremacy circuits [52] have increased both circuit depth and the density of two-qubit gates. As a result, these alternative approaches loose their advantage. Our approach is only affected mildly by these changes as it can no longer perform the *global CZ-gate* optimization. Our simulator thus requires up to one additional global-to-local swap for circuits of equal depth. The number of such swaps scales linearly with the circuit depth and, as a result, we expect the run time to increase by a factor $< 4x$ for a depth-40 circuit with dense two-qubit gates.

Chapter 6

Emulation of quantum circuits

In contrast to random circuits, the additional structure that is present in quantum algorithms can be exploited in order to speed up the simulation. In this chapter, which is a slightly modified version of Ref. [5], we thus introduce the concept of a quantum computer *emulator*. Emulation is an extension of simulation to the setting in which a comprehensive compilation framework for quantum programs is available. More specifically, an emulator may employ direct classical emulation for quantum subroutines at the level of their mathematical description rather than compiling them into elementary gates before carrying out the simulation. As a consequence, the run time of quantum algorithms run on classical hardware is drastically reduced.

We present various examples of such optimizations, accompanied by run time measurements which show the merits of quantum computer emulation. Furthermore, in order to arrive at heuristics for cases where multiple classical shortcuts exist, an analysis of crossover points is carried out. Finally, to demonstrate that our simulator, against which we achieve a speedup using our new quantum emulator, is state-of-the-art, we benchmark it against other existing simulators on a subset of quantum circuits.

While emulation is a widely recognized tool commonly used in many areas of computer science, we are not aware of previous work on emulation of quantum programs. A fitting example in the classical domain is the Structural Simulation Toolkit (SST) [87], which enables the user to run a program on various hardware models. An efficient emulation process is achieved by running the computations at different levels of accuracy and detail, similar to our quantum circuit emulator. Another example is the Intel® Software Development Emulator tool [88], which allows emulation of upcoming or experimental hardware features, such as new SIMD or transactional memory extensions, before they become available.

6.1 Quantum computer emulation

Simulation and emulation of quantum computers are inherently different concepts. As a *simulation* of a quantum computer we understand the exact calculation of the effects of every single gate. This directly mimics the operations that a quantum computer performs and the simulation can also include effects of classical and quantum noise as well as calibration or control errors.

Quantum computer *emulation*, on the other hand, is only required to return the same result as a perfect and noiseless quantum computation would. Instead of compiling an algorithm down to elementary gates for specific quantum hardware, certain high-level subroutines can be replaced by calls to faster classical shortcuts to be executed by the emulator. Depending on the level of abstraction at which the emulation is carried out, there is a large potential for optimizations and substantial speedup [1].

To illustrate this point, consider performing classical functions on a quantum computer which is needed in order to apply classical functionality to a superposition of inputs. The most famous application of this is Shor's algorithm [13]. In order to satisfy the reversibility constraint of quantum mechanics, these functions need to be implemented reversibly, which leads to a large overhead in the number of quantum gates compared to a non-reversible classical computation. This is due to the fact that temporary variables need to be reset by employing a so called uncomputation step [21].

A straight-forward approach to translating a classical function to a reversible quantum circuit is to replace all NAND gates by the reversible Toffoli gate (also called CCNOT), which requires an additional bit for each NAND to store the result. After completion of the circuit, the result can be copied using CNOT gates prior to clearing all (temporary) work bits by running the entire circuit in reverse [21]. This can also be run on a quantum computer using a quantum version of the Toffoli gate (which can be composed from single-qubit gates and CNOT gates). This transformation causes a doubling of gates and an overhead of one additional qubit for each original NAND gate. There are more sophisticated approaches [89] which reduce the number of work qubits by uncomputing intermediate results early. Yet, those intermediate results have to be recomputed[1] during the uncomputation step which follows after completion of the circuit, resulting in an increase of gate operations. This is bad news for a simulator, since both approaches cause a significant increase in runtime. Hence, the simulation of a classical function on a quantum computer is a very costly endeavor.

An emulator, on the other hand, does not need to compile the classical function down to reversible gates, nor does it have to simulate the additional work qubits

---

[1] As the uncomputation of an uncomputate step is recomputing the original result.

that may be needed during function execution. Instead, the emulator can just evaluate the classical function directly for each of its arguments, thereby saving huge amounts of computational power. In the next section, we will discuss four examples where emulation may gain a substantial performance advantage over simulation.

6.1.1 Arithmetic operations and mathematical functions

The most straight-forward example is the execution of arithmetic operations and mathematical functions on a quantum computer. Instead of simulating the vast number of Toffoli gates required to implement, *e.g.*, a multiplication or a trigonometric function reversibly, one can perform the classical multiplication or trigonometric function directly for each computational basis state using the hardware implementation available on classical computers.

We consider the multiplication and division of two numbers a and b into a new register c as examples. Specifically, we implement the mapping for multiplication

$$\begin{aligned}(a, b, c = 0) &= (a_1, ..., a_N, b_1, ..., b_N, 0, ..., 0)\\ &\mapsto (a, b, ab) \ ,\end{aligned}$$

and for division (with remainder r),

$$\begin{aligned}(a, b, c = 0) &= (a_1, ..., a_N, b_1, ..., b_N, 0, ..., 0)\\ &\mapsto (r, b, a/b) \ ,\end{aligned}$$

where the N-qubit input registers a and b may be in an arbitrary superposition, allowing this computation to be carried out on all (exponentially many) possible input states in parallel on a quantum computer.

On a simulator, the $3N$-qubit wavefunction is stored as a vector of 2^{3N} complex numbers with indices $i \in \{0, 1\}^{3N}$, which can be written as $i = a_1, ..., a_N, b_1, ..., b_N$,-$0, ..., 0$ in binary notation, where x_k denotes the k-th bit of x. The action of a multiplication corresponds to a permutation of the state vector, mapping the complex value at location i to the index $j = a_1, ..., a_N, b_1, ..., b_N, (ab)_1, ..., (ab)_N$. In order to achieve this transformation, a simulator would apply the corresponding Toffoli network. An emulator, on the other hand, can simply perform the described mapping directly. The simulation and emulation of a division can be carried out analogously.

To benchmark the simulation, we implement these operations using the adder of Ref. [44] combined with a repeated-addition-and-shift and a repeated-subtraction-and-shift approach for multiplication and division, respectively. The runtimes of emulation and simulation can be found in section 6.2.

6.1.2 Quantum Fourier transform

The quantum Fourier transform (QFT) [41] is a common quantum subroutine that is used in many quantum algorithms due to its ability to detect periods and patterns. At a formal mathematical level, the QFT performs a Fourier transform on the state vector α of n qubits, where each entry α_l ($0 \leq l < 2^n$) gets transformed as

$$\alpha_l \mapsto \frac{1}{2^{n/2}} \sum_{k=0}^{2^n-1} \alpha_k \exp\left(2\pi i \frac{kl}{2^n}\right) \ . \tag{6.1}$$

On a quantum computer, the QFT can be implemented by a sequence of $\mathcal{O}(n^2)$ Hadamard and conditional phase shift gates. Simulating this circuit is expensive as each gates acts on the state vector of size 2^n. An emulator, on the other hand, can just directly perform a Fast Fourier Transform (FFT) on the state vector using optimized classical libraries.

6.1.3 Quantum phase estimation

Quantum phase estimation (QPE) [41] is another subroutine that is used in many quantum algorithms, such as Shor's algorithm for factoring [13]. Given a circuit of a unitary operator U acting on n qubits and an eigenvector $\vec{u}$ stored in an n-qubit register, the QPE algorithm calculates the corresponding eigenvalue $e^{i\theta}$.[2] In a wave function simulator picture, the operator U is described by a unitary $2^n \times 2^n$ matrix. While there are many versions of the QPE algorithm, they all are based on repeatedly applying the controlled operator U.[3] Specifically, the operators

$$U^1, U^2, U^4, U^8, \ldots, U^{2^{b-1}} \tag{6.2}$$

need to be applied in order to arrive at a b-bit estimate of the eigenvalue angle θ. In addition, at least one work qubit and an inverse QFT are required when implementing the QPE as done in Ref. [90]. In the following, we assume that U is implemented on a quantum computer through a sequence of G gates, i.e.,

$$U = \prod_{i=1}^{G} U_i \ ,$$

where U_i is a single or two-qubit gate.

A simulator implements U_i through multiplications of sparse $2^n \times 2^n$ matrices with the wave function. Applying powers of U corresponds to repeatedly applying

---

[2] All eigenvalues of a unitary matrix can be written in the form of $e^{i\theta}$. QPE calculates the angle θ.

[3] Since the cost of simulating the application of the controlled version of U is essentially the same as simulating U itself, we ignore this detail in the further analysis.

the sequence of G gates which each has complexity $\mathcal{O}(2^n)$. From equation 6.2, it follows that we need to apply U exactly $2^b - 1$ times. Hence, the runtime complexity of QPE without accounting for the inverse QFT is $\mathcal{O}(G2^{n+b})$ for a quantum computer simulator using an algorithm with the minimal number of one ancilla (temporary work) qubit as done in [90]. Coherent phase estimation algorithms [41] that use b ancillas to optimize runtime will incur an additional factor $\mathcal{O}(2^b)$ in simulation effort.

An emulator can take a shortcut by first building a (dense) matrix representation of the unitary operator U and then using repeated squaring to calculate U^{2^i} iteratively for $i = 0, 1, ..., b-1$. Building the matrix representation of U requires $\mathcal{O}(G2^{2n})$ effort. Using standard matrix-matrix multiplication, repeated squaring can be performed in time $\mathcal{O}(2^{3n}b)$. Using Strassen's algorithm [91], the complexity can be reduced to $\mathcal{O}(2^{2.8n}b)$. Since G is typically polynomial in n, $G2^n$ is subdominant. There is an advantage in the asymptotic scaling when switching from quantum simulation to emulation if $b \geq 2n$, or $b > (\log_2 7 - 1)n \approx 1.8n$ when using Strassen.

Alternatively, a dense matrix eigensolver can be employed to directly classically compute the eigenvalues of U with effort $\mathcal{O}(G2^{2n} + 2^{3n})$ for approaches based on Hessenberg reduction [92], which again will have a scaling advantage compared to simulation for $b > 2n$. Given the cost of an eigendecomposition, this is advantageous especially when performed for a coherent QPE, which requires not just one, but b ancilla qubits, making the effort of simulation $\mathcal{O}(G2^{n+2b})$. In this case we have a scaling advantage for $b > n$.

Which of these approaches is more efficient depends on the required precision and the size of the matrix. An analysis of this trade-off and the respective timing results are presented in section 6.2.

6.1.4 Measurements

Finally, emulators have an advantage over actual quantum computers when it comes to estimating the expectation values of measurements. On a quantum computer, a measurement of n qubits only yields n bits of information, returning one of the states i as the result with probability given by $|\alpha_i|^2$. Our classical simulations are $\mathcal{O}(2^n)$ times more expensive than running the algorithm on a quantum computer, as we have to operate on the exponentially large vector representation. However, in return, we get the complete distribution of measurements and not just a single measurement sample.

While a quantum computer will often have to repeat an algorithm many times to get a (statistical) measurement with high enough accuracy, the classical emulation of such repeatedly executed measurements can easily be done in one step and the expectation value can immediately be evaluated. This removes the need for sampling and hence greatly reduces the overall simulation time.

As the time savings of emulation compared to simulation are just the number of repetitions of the circuit, no benchmarks are performed.

6.2 Performance results

6.2.1 Experimental setup

We compare the performance of quantum simulation and emulation on several systems.

For the distributed QFT and phase estimation we use the Stampede [93] system at the Texas Advanced Computing Center (TACC)/Univ. of Texas, USA (#10 in the current TOP500 list). It consists of 6400 compute nodes, each of which is equipped with two sockets of Xeon E5-2680 connected via QPI and 32GB of DDR4 memory per node (16GB per socket), as well as one Intel® Xeon Phi™ SE10P co-processor. Each socket has 8 cores, with hyperthreading disabled. The nodes are connected via a Mellanox FDR 56 Gb/s InfiniBand interconnect. We use OpenMP 4.0 [94] to parallelize the computation among threads. We have used the Intel® Compiler v15.0.2 with Intel® Math Kernel Library (MKL) v11.2.2, and Intel® MPI Library v5.0.

Additional single-node and single-core benchmarks were performed on an Intel® Core™i7-5600U processor, unless specified otherwise.

6.2.2 Arithmetic operations and mathematical functions

All experiments for arithmetic operations were performed on a single core of an Intel® Xeon E5-2697v2 processor due to the tremendous overhead in time when performing calculations with numbers consisting of more qubits than one node can handle. Such cases can only be dealt with by emulating the classical function, which effectively performs one global permutation of the (distributed) state vector. Also, using multiple cores is not very profitable, due to the fact that these operations are heavily memory-bandwidth bound.

Figure 6.1 shows performance results comparing the runtimes of simulating and emulating a multiplication of two m-bit integers a and b into a third register c. The advantage of the emulator, performing more than one hundred times faster, can clearly be seen.

A much larger advantage can be seen for division, which requires additional work qubits to perform the calculation. This incurs an exponential cost on a simulator. In Figure 6.2, the runtime advantage for a division can be seen. It can

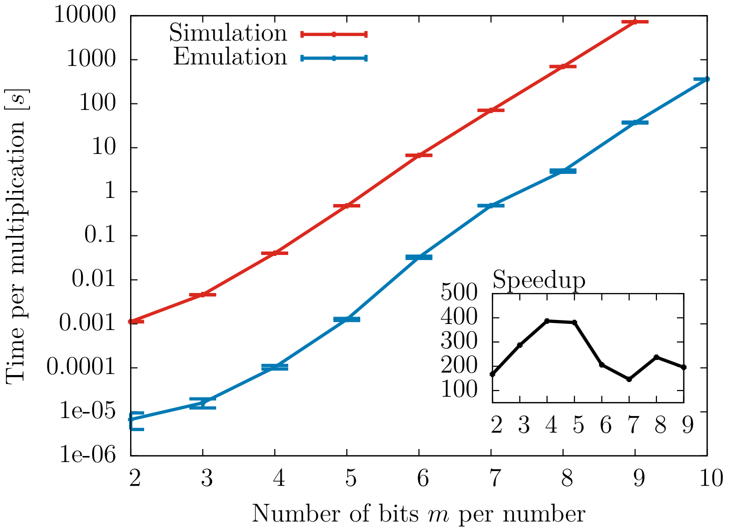


Figure 6.1: Timings for emulation and simulation of a multiplication of two m-qubit numbers into a third register consisting of m qubits (requiring a total of $n = 3m$ qubits). There is a clear speedup when emulating this operation instead of simulating it at gate-level. The drop in speedup when going from 5- to 6-bit numbers is due to the data exceeding the limits of the L3-Cache (4 MBytes), since $2^{3 \cdot 6}$ [entries] $\cdot$ 16 [bytes/entry] = 4.2 MBytes.

be observed that the overhead grows with the number of qubits used to represent the integers, as the number of required work-qubits grows as well.

Even more dramatic effects can be expected when dealing with complex mathematical operations such as trigonometric functions, where some kind of series expansion or iterative procedure with many intermediate results is used. For each of these temporary values, additional m qubits are required, causing an exponential overhead of the simulation in both space and time. Emulating such classical reversible functions not only pays off but makes it feasible on today's classical supercomputers, which otherwise would not be able to handle the enormous memory requirements.

6.2.3 Quantum Fourier transform

To benchmark the quantum Fourier transform, we use parallel implementations of both the simulator and the emulator, storing the wave function for 28 qubits locally and using 2^{N-28} nodes for $N(\geq 28)$ qubits, which corresponds to weak scaling (keeping the problem size per node constant). In Figure 6.3 one can clearly see that simulating the QFT circuit is worse than directly performing a one-dimensional

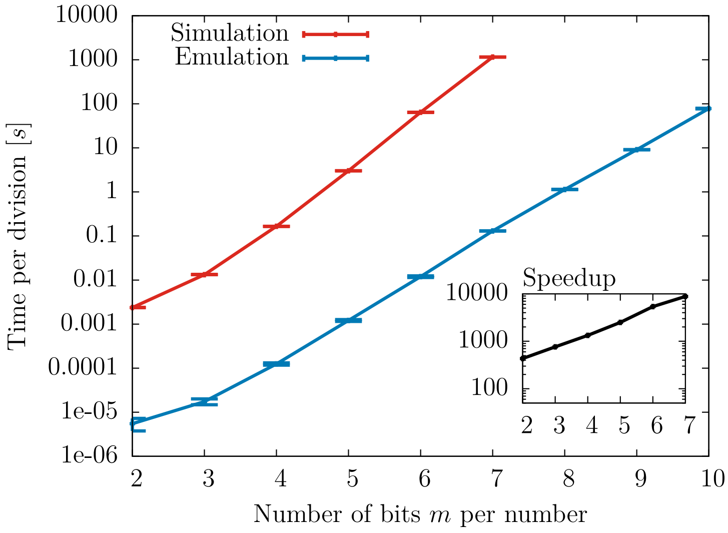


Figure 6.2: Timings for emulation and simulation of an integer division of two m-qubit numbers into a third register consisting of m qubits (requiring a total of $n = 3m$ qubits). The speedup is far greater than for multiplication, which is due to the extra work qubits required to do the test for less/equal by checking for overflow. In addition, the numbers used for the division are limited to 7 bits due to the larger memory requirements caused by the extra work qubits.

distributed classical fast Fourier transform. For the latter we used the Intel® Cluster FFT from the Intel® MKL library, which we found to be faster than FFTW [57].

We observe that quantum emulation is 15× faster than quantum simulation on a single node. As we increase both the system size and the number of nodes, we observe a degraded weak scaling behavior. This is expected due to the increasing amount of communication. Despite this performance degradation, emulation still achieves a substantial $6 - 15\times$ speedup over simulation.

6.2.4 Quantum phase estimation

We have used the Intel® MKL implementations of complex matrix-matrix multiplication (`zgemm`) and a general eigensolver (`zgeev`) to perform repeated squaring and to determine the eigensystem, respectively. In typical applications, one can use the high-order (distributed) qubits as control qubits, enabling to execute the unitary operators on low-order (local) qubits and to avoid using the ScalaPACK implementation of zgeev, which scales very poorly with the number of nodes. Therefore, we focus on single-node performance for the QPE timings. Table 6.1

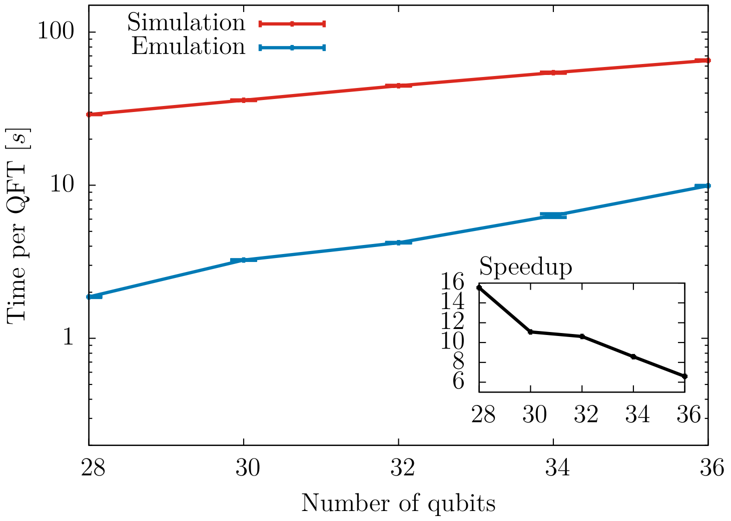


Figure 6.3: Execution times for emulation and simulation of a quantum Fourier transform of N-qubits. Both emulation and simulation are run on 2^{N-28} nodes in order to keep the problem size per node constant. The emulator shows a clear advantage even when executing the QFT on a large number of qubits.

# qubits n acted on by U	**8**	**9**	**10**	**11**	**12**	**13**	**14**
Number of gates G	**29**	**33**	**37**	**41**	**45**	**49**	**53**
$T_{\text{apply } U \text{ with simulator}}$ [s]	$1.44 \cdot 10^{-4}$	$1.60 \cdot 10^{-4}$	$1.80 \cdot 10^{-4}$	$2.11 \cdot 10^{-4}$	$2.44 \cdot 10^{-4}$	$3.46 \cdot 10^{-4}$	$4.92 \cdot 10^{-4}$
$T_{\text{construction of dense } U}$ [s]	$7.60 \cdot 10^{-4}$	$3.46 \cdot 10^{-3}$	$1.55 \cdot 10^{-2}$	$6.88 \cdot 10^{-2}$	$3.02 \cdot 10^{-1}$	1.32	5.69
$T_{\text{zgemm of dense} U}$ [s]	$8.39 \cdot 10^{-4}$	$6.71 \cdot 10^{-3}$	$5.37 \cdot 10^{-2}$	$4.29 \cdot 10^{-1}$	3.44	$2.75 \cdot 10^{1}$	$2.20 \cdot 10^{2}$
$T_{\text{zgeev of dense } U}$ [s]	$9.60 \cdot 10^{-2}$	$5.27 \cdot 10^{-1}$	1.70	6.72	$3.22 \cdot 10^{1}$	$1.80 \cdot 10^{2}$	$9.01 \cdot 10^{2}$
Crossover [# bits of precision]							
Repeated Squaring	6	9	12	15	18	21	24
Eigendecomposition	10	12	14	15	18	19	21

Table 6.1: Timings of the various steps involved when simulating or emulating a quantum phase estimation. Our example is for the time evolution of a one-dimensional transverse field Ising model. The lower panel shows the crossover precision in bits at which emulation using repeated squaring or eigendecomposition becomes advantageous over direct simulation.

depicts the results for applying QPE to a unitary operator U acting on different numbers of qubits $n \in \{8, ..., 14\}$. For each n, we determine the number of bits of precision corresponding to the crossover point at which emulation becomes faster

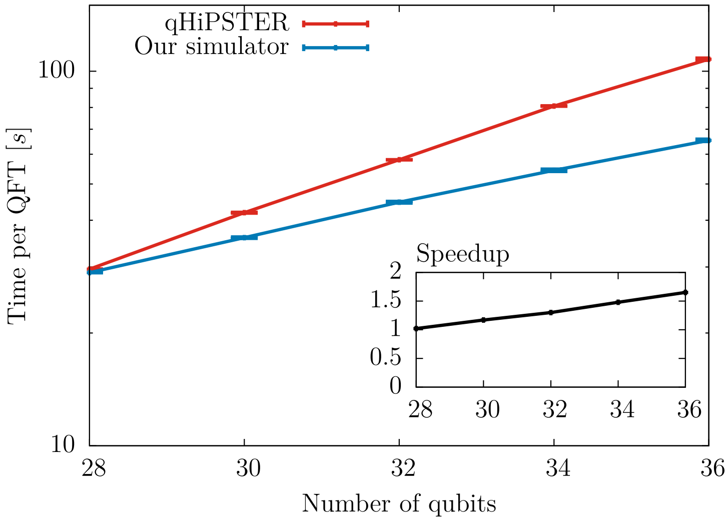


Figure 6.4: Comparison between qHiPSTER and our simulator for applying a quantum Fourier transform. The performance advantage of our simulator grows with the required communication, allowing simulations of larger systems.

than quantum simulation.

6.2.5 Comparison against other simulators

In order to show that the obtained speedups result from emulation and do not originate from a suboptimal implementation of our simulator, we provide benchmarks comparing the performance of our simulator to other state-of-the-art simulators, namely qHiPSTER [84] and LIQ$Ui\,|\rangle$ [95].

The simulator benchmarks consist of two operations: Applying a QFT and an entangling operation, where the latter applies a Hadamard gate to the first qubit, followed by a series of CNOTs acting on all other qubits, all conditioned on the first qubit.

Since only qHiPSTER provides a distributed multi-node implementation, the parallel QFT comparison is exclusively carried out between qHiPSTER and our simulator. We show the weak-scaling behavior in Figure 6.4, where N varies between 28 and 36 and the number of sockets is chosen to keep the memory per node constant (i.e. using from 1 to 256 nodes). Note that our parallel simulator shows a growing advantage as the requirement for communication increases. This stems from the fact that our simulator takes advantage of the structure of gate matrices, allowing, e.g., to reduce the communication for diagonal gates such as the conditional phase shift.

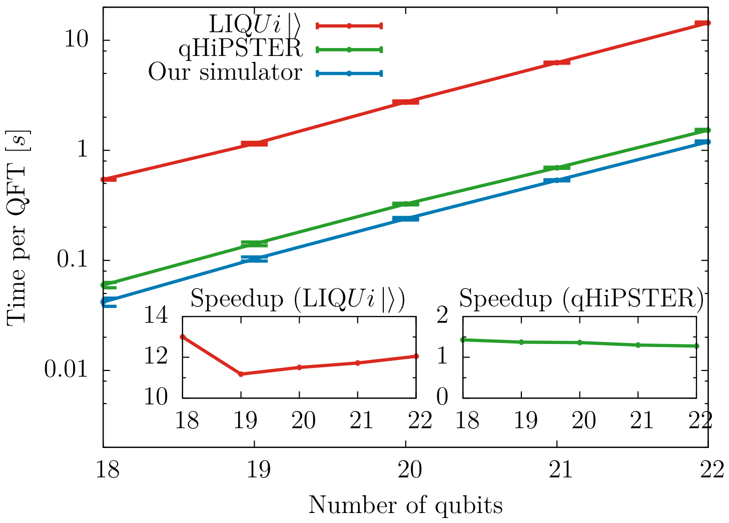


Figure 6.5: Comparison of our simulator to qHiPSTER and LIQ$Ui\,|\rangle$ for applying a quantum Fourier transform on a single node. Our simulator clearly shows the best performance.

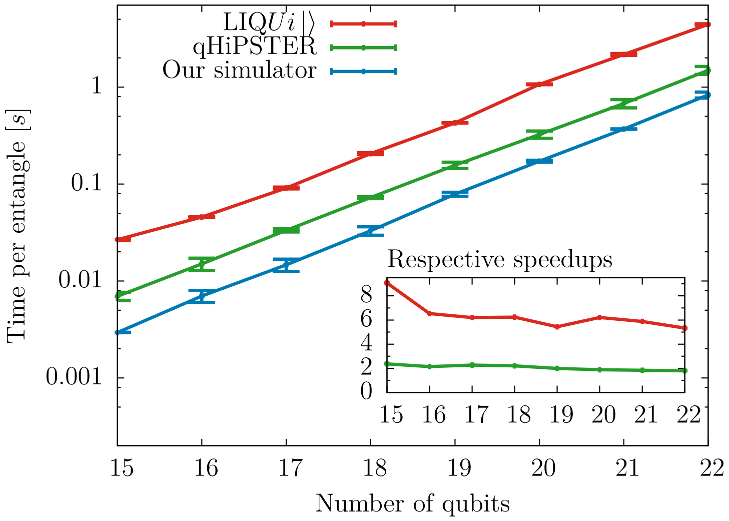


Figure 6.6: Comparison of our simulator to qHiPSTER and LIQ$Ui\,|\rangle$ for applying an entangling operation on a single node. Our simulator achieves significant speedups of 2× and 6×, respectively.

The single node performance is depicted in Figure 6.5 for a QFT, and in Figure 6.6 for the entangling operation, which provides further proof of our simulator outperforming the other two simulators. As a consequence, there will be an even larger advantage of our *emulator* against those simulators.

6.3 Conclusion

The development of quantum algorithms that promise to solve important open computational problems has caused quantum computing to be viewed as a viable long-term candidate for post-exascale computing. Due to the current lack of universal quantum computers, the testing, debugging, and development of algorithms is done on classical systems, employing high-performance simulators. For the case of noiseless, perfect simulations, we propose to emulate the algorithms instead, making use of the optimizations presented in this chapter. Yet, this emulation is only possible if the quantum program is available in a high-level language, where the higher levels of abstractions are easy to identify. This is the case in the compilation framework described in [1], where emulators have been suggested at various levels, and can also be integrated into LIQ$Ui\,|\rangle$ [95], Quipper [37], or any other quantum programming language.

Our results show that quantum program emulation allows to test and debug large quantum circuits at a cost that is substantially reduced when compared to previous approaches. The advantage is already substantial for operations such as the quantum Fourier transforms, and grows to many orders of magnitude for arithmetic operations, since emulation avoids simulating ancilla qubits (needed for reversible arithmetic) which would incur an additional cost that is exponential in the number of such ancilla qubits. Emulation will thus be a crucial tool for testing, debugging and evaluating the performance of quantum algorithms involving arithmetic operations, which includes quantum-accelerated Monte Carlo sampling [96] and machine learning applications [97, 98, 99].

Subsequent work has extended the emulation capabilities of our emulator further by adding support for modular arithmetic. As a result, Shor's algorithm for factoring $N = 4,028,033 = 2,003 \cdot 2,011$ can be run on a regular laptop in less than 3 minutes [9]. In contrast, full state vector simulation would need to keep track of at least $2n + 1 = 45$ qubits [100], where $n = \lceil \log_2(N) \rceil$. Full simulation of the same instance would thus require at least 0.5 petabytes of memory. As such, this is an excellent demonstration of how beneficial emulators are in practice.

Part III

Quantum circuits for mathematical functions

Introduction to Part III

During the compilation process, simulation results and theoretical bounds may be employed in order to make decisions regarding the parameters of available decompositions. Developing new efficient decompositions into lower-level gates for specific operations, however, is a nontrivial task. While design automation [8, 11], may be sufficient to perform a first feasibility analysis of algorithms involving classical oracles, this usually results in circuits with much greater resource requirements than designs by humans [8]. Therefore, to reduce the resource requirements of quantum algorithms and, thereby, move crossover points toward smaller system sizes, manual optimization of circuits is still necessary. This is analogous to classical high-performance computing, where system-specific optimizations are essential to utilize most of the available computational resources.

Many quantum algorithms evaluate classical functions on a superposition of inputs. Examples include Shor's algorithm [13] and various algorithms for simulating physical systems [45, 71, 96] and for solving linear systems of equations [46, 101]. It is well-known that the overhead resulting from transforming a classical computation to a reversible computation is polynomial in the number of basic operations in both space and time [21]. While this conversion is thus *efficiently* possible, the change in resource requirements greatly influences the problem sizes at which a quantum algorithm outperforms its classical competition. It is thus crucial to analyze the costs of such subroutines in order to determine practical applications of the first large-scale quantum computers. In addition, such analyses allow to identify performance bottlenecks.

This part contains two chapters. In chapter 7, which is a slightly modified version of Ref. [6], we develop a new Toffoli based addition circuit which requires no (clean) work qubits to perform the n-qubit mapping

$$|x\rangle \mapsto |x+c\rangle$$

for a classical constant c. Previous Toffoli based circuits achieving this mapping required n clean work qubits, where n denotes the number of qubits in the quantum register $|x\rangle$. The only implementation working without extra qubits was Draper's addition in Fourier space [35], which has the downside of requiring a quantum

Fourier transform and its inverse; both of which feature rotations that need to be rewritten in terms of a discrete gate set to allow for a fault-tolerant implementation.

To save these n work qubits without resorting to such rotation gates, our new addition circuit makes use of *dirty qubits*, that is, it borrows qubits which are already in use by the ongoing computation. Using our recursive construction it is possible to achieve the above mapping in depth $\mathcal{O}(n)$ by borrowing a single dirty qubit.

We then use the resulting addition circuit to implement Shor's algorithm for factoring an n-bit number. Since we require no clean work qubits and no rotations, we are able to reduce the circuit depth by $\Theta(\log \varepsilon^{-1})$ compared to the state of the art, where $\varepsilon \in \mathcal{O}(n^{-3})$ denotes the accuracy of rotation synthesis, while keeping the circuit width constant at $2n+2$ qubits [102]. As an additional benefit, we are able to fully test our Toffoli based modular multiplication circuit.

In chapter 8, which is a slightly modified version of Ref. [7], we analyze the costs of mathematical functions that occur frequently in the quantum algorithm literature. We use insights from classical high-performance computing in order to optimize the resulting library for fixed-point arithmetic. Our main technical innovation lies in a new resource-efficient approach to evaluate piece-wise smooth functions to high accuracy on a quantum computer. In essence, we propose an optimized circuit to combine a host of different low-degree polynomial approximations, each of which approximates the target function on a small subdomain. We present and implement an algorithm to determine these subdomains and the corresponding polynomials, and report resource estimates for various functions that occur often in the quantum algorithm literature.

Chapter 7

Factoring using $2n + 2$ qubits with Toffoli based modular multiplication

In this chapter, we introduce a new type of addition circuit. In contrast to previous implementations achieving the mapping $|x\rangle \mapsto |x + c\rangle$ for a classical constant c, our circuit uses no quantum Fourier transforms and only one work qubit which may be in an arbitrary state.

We then use our addition circuit to construct the modular exponentiation required in Shor's algorithm [13] to factor the n-bit number N. The quantum part of Shor's algorithm consists of the following steps: (1) Initialize $|x\rangle$ to the uniform superposition by applying a Hadamard gate to each of the $2n$ qubits in $|x\rangle$. (2) Perform modular exponentiation of the number a which was chosen uniformly at random from $[2, ..., N - 1]$ with input $|x\rangle$, i.e., perform

$$|x\rangle\,|0\rangle \mapsto |x\rangle\,|a^x \bmod N\rangle\,.$$

(3) Apply the inverse quantum Fourier transform to $|x\rangle$. (4) Measure $|x\rangle$ and try to infer the period of $f(x) := a^x \bmod N$ and then the factors of N. (5) If unsuccessful (for details, see Refs. [13, 41]), the entire procedure is repeated, choosing a again uniformly at random from $[2, ..., N - 1]$.

Implementing the modular exponentiation above using $2n$ conditional modular multiplications [90] results in the circuit depicted in Fig. 7.1.

There are many possible implementations of Shor's algorithm, all of which offer deeper insight into space/time trade-offs by, e.g., using different ways of implementing the circuit for adding a known classical constant c to a quantum register $|a\rangle$, see Table 7.1.

The implementation given in Ref. [102] features the lowest known number of

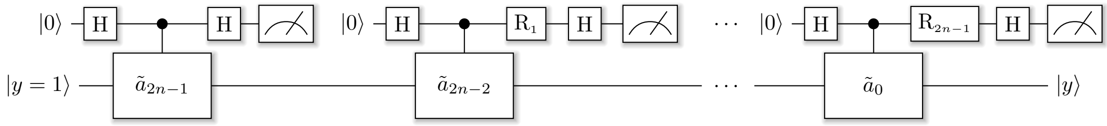


Figure 7.1: Circuit for Shor's algorithm as in [90], using the single-qubit semi-classical quantum Fourier transform from [103]. In total, $2n$ modular multiplications by $\tilde{a}_i = a^{2^i} \bmod N$ are required (denoted by $\tilde{a}_i$-gates in the circuit). The phase-shift gates R_k are given by $\left(\begin{smallmatrix} 1 & 0 \\ 0 & e^{i\theta_k} \end{smallmatrix}\right)$ with $\theta_k = -\pi \sum_{j=0}^{k-1} 2^{k-j} m_i$, where the sum runs over all previous measurements j and $m_j \in \{0,1\}$ denotes the respective measurement result (m_0 denotes the least significant bit of the final answer and is obtained in the first measurement).

	Cuccaro [44]	Takahashi [69]	Draper [35]	Our adder
Size	$\Theta(n)$	$\Theta(n)$	$\Theta(n^2)$	$\Theta(n \log n)$
Depth	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$
Ancillas	$n+1$ (clean)	n (clean)	0	$[1, n]$ dirty

Table 7.1: Costs associated with various implementations of addition $|a\rangle \mapsto |a + c\rangle$ of a value a by a classical constant c. Our adder uses 1 dirty ancilla to achieve a depth in $\Theta(n)$. More dirty qubits (up to n) allow to reduce the depth by constant factors.

qubits and uses Draper's addition in Fourier space [35], allowing factoring to be achieved using only $2n+2$ qubits at the cost of a circuit size in $\Theta(n^3 \log n)$. Furthermore, the QFT circuit features many (controlled) rotations, which in turn imply a large T-gate count when quantum error-correction (QEC) is required. Implementations using classically-inspired adders as in Ref. [44], on the other hand, yield circuits with as few as $3n + \mathcal{O}(1)$ qubits and $\mathcal{O}(n^3)$ size. Such classical reversible circuits have several advantages over Fourier-based arithmetic. In particular,

1. they can be efficiently simulated on a classical computer, i.e., the logical circuits can be tested on a classical computer,

2. they do not suffer from the overhead of single-qubit rotation synthesis [27, 73, 104] when employing QEC.

We construct our $\mathcal{O}(n^3 \log n)$-sized implementation of Shor's algorithm from our Toffoli based in-place constant-adder, which adds a classically known n-bit constant c to the n-qubit quantum register $|x\rangle$, i.e., which implements $|x\rangle \mapsto |x + c\rangle$

where x is an arbitrary n-bit input and $x + c$ is an n-bit output (the final carry is ignored).

Our main technical innovation is to obtain space savings by making use of *dirty* ancilla qubits which the circuit is allowed to borrow during its execution. By a dirty ancilla we mean—in contrast to a clean ancilla which is a qubit that is initialized in a known quantum state—a qubit which can be in an arbitrary state and, in particular, may be entangled with other qubits. In our circuits, whenever such dirty ancilla qubits are borrowed and used as scratch space, they are then returned in exactly the same state as they were in when they were borrowed.

Our addition circuit requires $\mathcal{O}(n \log n)$ Toffoli gates and has an overall depth of $O(n)$. Following Beauregard [90], we construct a modular multiplication circuit using this adder and report the gate counts of Shor's algorithm in order to compare our implementation to the one of Takahashi et al. [102], who used Fourier addition [35] as a basic building block.

7.1 Toffoli based in-place addition

One possible way to construct an (inefficient) adder is to note that one can calculate the final bit r_{n-1} of the result $r = a + c$ using $n - 1$ borrowed dirty qubits g. Takahashi et al. hardwired a classical ripple-carry adder to arrive at a similar circuit, which they used to optimize the modular addition in Shor's algorithm [102]. We construct our CARRY circuit from scratch, which allows to save $\mathcal{O}(n)$ NOT gates as follows.

Since there is no way of determining the state of the g-register without measuring, one can only use toggling of qubits to propagate information, as done by Barenco et al. for the multiply-controlled-NOT using just one borrowed dirty ancilla qubit [26]. We choose to encode the carry using such qubits, i.e., the toggling of qubit g_i, which we denote as $g_i = 1$, indicates the presence of a carry from bit i to bit $i + 1$ when adding the constant c to the bits of a. Thus, g_i must toggle if (at least) one of the following statements is true:

$$a_i = c_i = 1, \quad g_{i-1} = a_i = 1, \quad \text{or} \;\; g_{i-1} = c_i = 1.$$

If $c_i = 1$, one must toggle g_{i+1} if $a_i = 1$, which can be achieved by placing a CNOT gate with target g_{i+1} and control $a_i = 1$. Furthermore, there may be a carry when $a_i = 0$ but $g_{i-1} = 1$. This is easily solved by inverting a_i and placing a Toffoli gate with target g_i, conditioned on a_i and g_{i-1}. If, on the other hand, $c_i = 0$, the only way of generating a carry is for $a_i = g_{i-1} = 1$, which can be solved with the Toffoli gate from before.

Thus, in summary, one always places the Toffoli gate conditioned on g_{i-1} and a_i, with target g_i and, if $c_i = 1$, one first adds a CNOT and a NOT gate. This

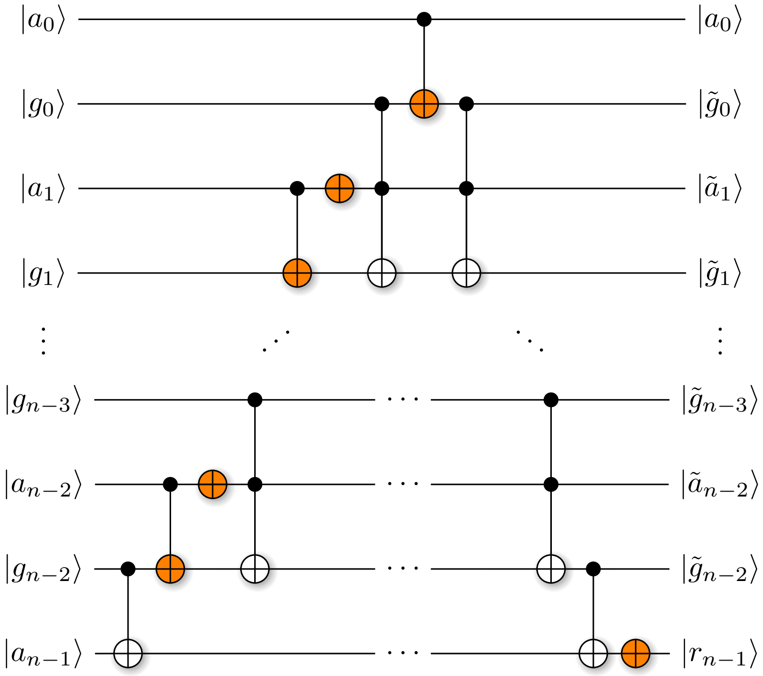


Figure 7.2: Circuit computing the last bit of $r = a + c$ using dirty qubits g. An orange (dark) gate acting on a qubit with index i must be dropped if the i-th bit of the constant c is 0. The entire sequence must be run again in reverse (without the gates acting on a_{n-1}) in order to reset all qubits to their initial value except for r_{n-1}.

classical conditioning during circuit-generation time is indicated by colored gates in Fig. 7.2. In order to apply the Toffoli gate conditioned on the toggling of g_{i-1}, one places it before the potential toggling, and then again afterwards such that if both are executed, the two gates cancel. Finally, the borrowed dirty qubits and the qubits of a need to be restored to their initial state (except for the highest bit of a, which now holds the result). This is done by running the entire circuit backwards, ignoring all gates acting on a_{n-1}.

One can easily save the qubit g_0 in Fig. 7.2 by conditioning the Toffoli gate acting on g_1 directly on the value of a_0 (instead of testing for toggling of g_0). If $c_0 = 0$, the two Toffoli gates can be removed altogether since the CNOT acting on g_0 would not be present and the two Toffolis would cancel. If, on the other hand, $c_0 = 1$, the two Toffoli gates can be replaced by just one, conditioned on a_0. See Fig. 7.3 for the complete circuit computing the last bit of a when adding the constant $c = 11$.

If one were to iteratively calculate the bits $n-2, ..., 1, 0$, one would arrive at an $O(n^2)$-sized addition circuit using $n-1$ borrowed dirty ancilla qubits. This is

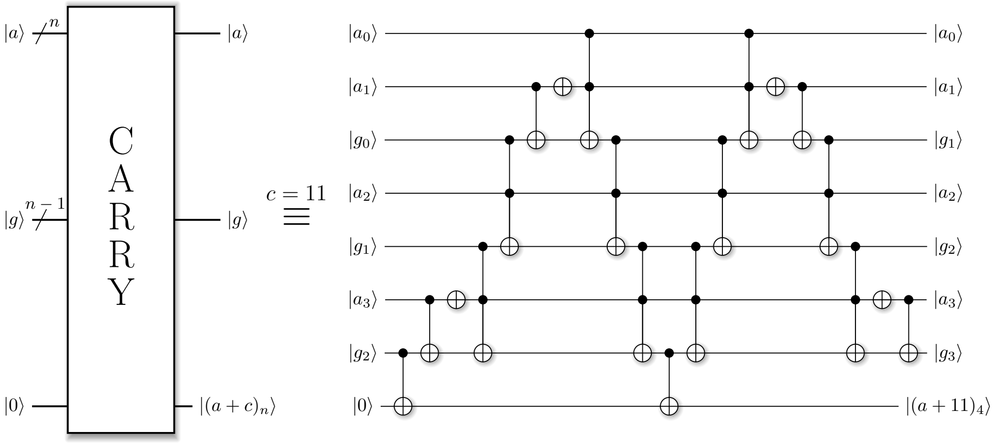


Figure 7.3: Example circuit computing the final carry of $r = a + 11$ derived from the construction depicted in Fig. 7.2. The binary representation of the constant c is $c = 11 = 1011_2$, i.e., the orange gates in Fig. 7.2 acting on qubit index 2 have been removed since $c_2 = 0$. Furthermore, the optimization mentioned in the text has been applied, allowing to remove g_0 in Fig. 7.2.

the same size as the Fourier addition circuit [35], unless one uses an approximate version of the quantum Fourier transform bringing the size down to $\mathcal{O}(n \log \frac{n}{\varepsilon})$ [105]. We improve our construction to arrive at a size in $\mathcal{O}(n \log n)$ in the next subsection.

7.1.1 Serial implementation

An $\mathcal{O}(n \log n)$-sized addition circuit can be achieved by applying a divide-and-conquer scheme to the straight-forward addition idea mentioned above (see Fig. 7.4), together with the incrementer proposed in [106], which runs in $\mathcal{O}(n)$. Since we have many dirty ancillae available in our recursive construction, the n-borrowed qubits incrementer in [106] is sufficient: Using the ancilla-free adder by Takahashi [69], which requires no incoming carry, and its reverse to perform subtraction, one can perform the following sequence of operations to achieve an incrementer using n borrowed ancilla qubits in an unknown initial state $|g\rangle$:

$$\begin{aligned} |x\rangle\,|g\rangle &\mapsto |x-g\rangle\,|g\rangle \\ &\mapsto |x-g\rangle\,|g'-1\rangle \\ &\mapsto |x-g-g'+1\rangle\,|g'-1\rangle \\ &\mapsto |x+1\rangle\,|g\rangle\,, \end{aligned}$$

where g' denotes the two's-complement of g and $g'-1 = \overline{g}$, the bit-wise complement of g. A conditional incrementer can be constructed by either using two controlled

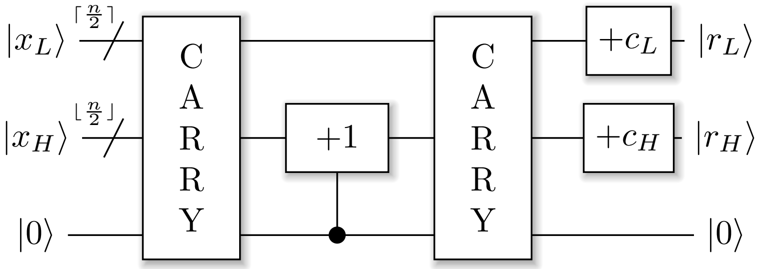


Figure 7.4: Circuit for adding the constant a to the register x. x_H and x_L denote the high- and low-bit part of x. The CARRY gate computes the carry of the computation $x_L + a_L$ into the qubit with initial state $|0\rangle$, borrowing the x_H qubits as dirty ancillae. This carry is then taken care of by an incrementer gate acting on the high-bits of x. Applying this construction recursively yields an $\mathcal{O}(n \log n)$ addition circuit with just one ancilla qubit (the $|0\rangle$ qubit in this figure).

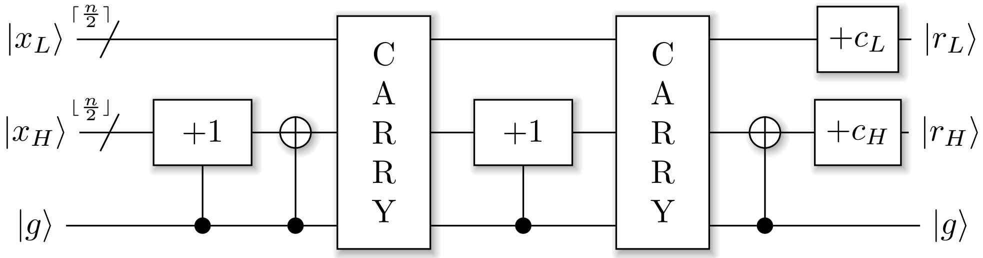


Figure 7.5: The circuit of Fig. 7.4 for the case when the ancilla qubit is dirty (unknown initial state $|g\rangle$, left unchanged by the computation).

adders as explained, or by applying an incrementer to a register containing both the target and control qubits of the conditional incrementer, where the control qubit is now the least significant bit [106]. Then, the incrementer can be run on the larger register, followed by a final NOT gate acting on the control qubit (since it will always be toggled by the incrementer). In the latter version, one can either use one more dirty ancilla qubit for cases where $n \bmod 2 = 0$ or, alternatively, split the incrementer into two smaller ones as done in Ref. [106]. We will use the construction with an extra dirty qubit, since there are plenty of idle qubits available in Shor's algorithm.

In order to make the circuit depicted in Fig. 7.4 work with a borrowed dirty qubit, the incrementer has to be run twice with a conditional inversion in between. The resulting circuit can be seen in Fig. 7.5. At the lowest recursion level, only 1-bit additions are performed, which can be implemented as a NOT gate on x_i if $c_i = 1$; all carries are accounted for earlier.

7.1.2 Runtime analysis of the serial implementation

In the serial version, we always reuse the one borrowed dirty ancilla qubit to hold the output of the CARRY gate, which is implemented as shown in Fig. 7.3. The CARRY gate has a Toffoli count of $T_{carry}(n) = 4n + \mathcal{O}(1)$ (including the uncomputation of the ancilla qubits) and the controlled incrementer using n borrowed dirty qubits features a Toffoli count of $T_{incr}(n) = 4n + \mathcal{O}(1)$ (2 additions). Both of these circuits have to be run twice on roughly $\frac{n}{2}$ qubits. Therefore, the first part of the recursion has a Toffoli count of $T_{rec}(n) = 8n + \mathcal{O}(1)$. The recursion for the Toffoli count $T_{add}(n)$ of the entire addition circuit thus yields

$$\begin{aligned} T_{add}(n) &= T_{add}\left(\left\lceil \frac{n}{2} \right\rceil\right) + T_{add}\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + T_{rec}(n) \\ &= 8n \log_2 n + \mathcal{O}(n). \end{aligned}$$

For a controlled addition, only the two CNOT gates acting on the last bit in Fig. 7.3 need to be turned into their controlled versions, which is another nice property of this construction.

7.1.3 Parallel / low-depth version

If the underlying hardware supports parallelization, one can compute the carries for the additions $+c_L$ and $+c_H$ in Fig. 7.4 in parallel, at the cost of one extra qubit in state $|0\rangle$ which will then hold the output of the CARRY computation of $+c_H$. Doing this recursively and noting that there must be at least two qubits of x per CARRY gate, one sees that this circuit can be parallelized at a cost of $\frac{n}{2}$ ancilla qubits in state $|0\rangle$. Using the construction depicted in Fig. 7.5 allows us to use $\frac{n}{2}$ borrowed dirty qubits instead. To see that this construction can be used in our implementation of Shor's algorithm, consider that during the modular multiplication

$$|x\rangle\, |0\rangle \mapsto |x\rangle\, |(ax) \bmod N\rangle \ ,$$

we perform additions into the second register, conditioned on the output of the comparator in Fig. 7.6. Therefore, n qubits of the x-register are readily available to be used as borrowed dirty qubits, thus reducing the depth of our addition circuit to $\mathcal{O}(n)$.

Note that this is also possible if there is only one dirty ancilla available: Applying one round of the recursion in Fig. 7.5 allows to run the low-depth version of $+c_L$ using x_H as dirty qubits, before executing $+c_H$ using r_L as dirty qubits.

7.2 Modular multiplication

The modular multiplier can be constructed from a modular addition circuit using a repeated-addition-and-shift approach, as done in Refs. [90, 102]:

$$\begin{aligned}(ax) \bmod N &= (a(x_{n-1}2^{n-1} + \cdots + x_0 2^0)) \bmod N \\ &= (((a2^{n-1}) \bmod N)x_{n-1} + \cdots + ax_0)\ ,\end{aligned}$$

where $x_{n-1}, ..., x_0$ is the binary expansion of x, and addition is carried out modulo N. Since $x_i \in \{0, 1\}$, this can be viewed as modular additions of $(a2^i) \bmod N$ conditioned on $x_i = 1$. The transformation by Takahashi et al. [102] allows to construct an efficient modulo-N addition circuit from a non-modular adder. For an illustration of the procedure see Fig. 7.6, where the comparator can be implemented by applying our carry circuit on the inverted bits of b. Also, note that it is sufficient to turn the final CNOT gates (see Fig. 7.3) of the comparator in Fig. 7.6 into Toffoli gates in order to arrive at controlled modular addition, since the subsequent add/subtract operation is executed conditionally on the output of the comparator.

The repeated-addition-and-shift algorithm transforms the input registers

$$|x\rangle\,|0\rangle \mapsto |x\rangle\,|(a \cdot x) \bmod N\rangle\ .$$

In Shor's algorithm, $2n$ such modular multiplications are required and in order to keep the total number of $2n + 2$ qubits constant, the uncompute method from Ref. [90] can be used: After swapping the two n-qubit registers, one runs another modular multiplication circuit, but this time using subtraction instead of addition and with a new constant of multiplication, namely the inverse a^{-1} of a modulo N. This achieves the transformation

$$\begin{aligned}|x\rangle\,|(ax) \bmod N\rangle &\mapsto |(ax) \bmod N\rangle\,|x\rangle \\ &\mapsto |(ax) \bmod N\rangle\,\left|(x - a^{-1}ax) \bmod N\right\rangle \\ &= |(ax) \bmod N\rangle\,|0\rangle\ ,\end{aligned}$$

as desired. In total, this procedure requires $2n + 1$ qubits: $2n$ for the two registers and 1 to achieve the modular addition depicted in Fig. 7.6.

7.3 Implementation and simulation results

In Shor's algorithm, a controlled modular multiplier is needed for the modular exponentiation which takes the form of a quantum phase estimation, since

$$\begin{aligned}a^x \bmod N &= a^{2^{n-1}x_{n-1} + 2^{n-2}x_{n-2} \cdots + x_0} \bmod N \\ &= a^{2^{n-1}x_{n-1}} \cdot a^{2^{n-2}x_{n-2}} \cdots a^{x_0},\end{aligned}$$

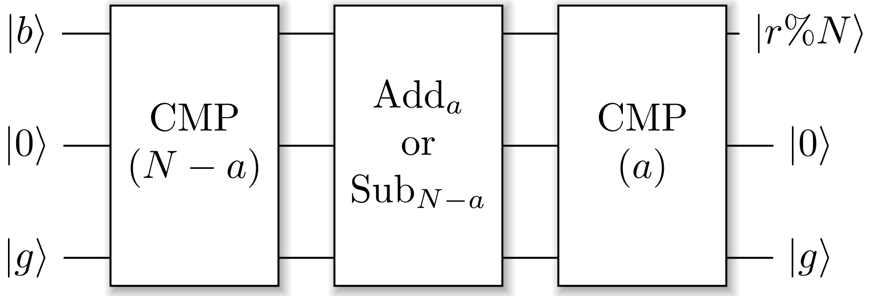


Figure 7.6: Taken from [102]: Construction of a modular adder $|b\rangle \mapsto |r \bmod N\rangle$ with $r = a + b$, using a non-modular adder. The CMP gate compares the value in register b to the classical value $N - a$, which we implement using our carry gate. The result indicates whether $b < N - a$, i.e., it indicates whether we must add a or $a - N$. Finally, the indicator qubit is reset to $|0\rangle$ using another comparison gate. In our implementation, the add/subtract operation uses between 1 (serial) and $\frac{n}{2}$ (parallel) qubits of g.

where again $x_i \in \{0, 1\}$ and multiplication is carried out modulo N. Thus, modular exponentiation can be achieved using modular multiplications by constants $\tilde{a}_i$ conditioned on $x_i = 1$, where

$$\tilde{a}_i = a^{2^i} \bmod N.$$

We do not have to condition our inner-most adders; we can get away with adding two controls to the comparator gates in Fig. 7.6, which turns the CNOT gates acting on the last bit in Fig. 7.3 into 3-qubit-controlled-NOT gates, which can be implemented using 4 Toffoli gates and one of the idle garbage qubits of g [26]. Note that there are n idle qubits available when performing the controlled addition/-subtraction in Fig. 7.6 ($n - 1$ qubits in g plus the x_i qubit the comparator was conditioned upon). The controlled addition/subtraction circuit can thus borrow $\frac{n}{2}$ dirty qubits from the g register to achieve the parallelism mentioned in subsection 7.1.2, and the remaining $\frac{n}{2}$ dirty qubits can be used to decrease the depth of the incrementers in the recursive execution of the circuit in Fig. 7.5.

We implemented the controlled modular-multiplier performing the operation

$$\begin{aligned} |x\rangle\,|0\rangle &\mapsto |x\rangle\,|(ax) \bmod N\rangle \\ &\mapsto |(ax) \bmod N\rangle\,|0\rangle \end{aligned}$$

in the LIQ$Ui\,|\rangle$ quantum software architecture [95]. We extended LIQ$Ui\,|\rangle$ by a reversible circuit simulator to enable large scale simulations of Toffoli based circuits.

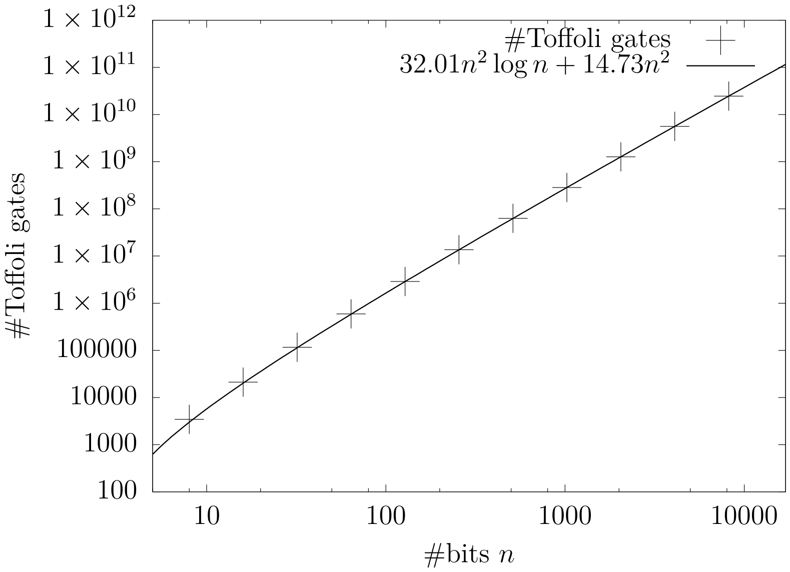


Figure 7.7: Scaling of the Toffoli count $T_M(n)$ with bit size n for the controlled modular multiplier. Each data point represents a modular multiplication run (including uncompute of the x-register) using $n = 2^m$ bits for each of the two registers, with $m \in \{3, ..., 13\}$.

To test our circuit designs and gate estimates, we simulated our circuits on input sizes of up to 8, 192-bit numbers. The scaling results of the Toffoli count $T_{mult}(n)$ of our controlled modular-multiplier are as expected. Each of the two (controlled) multiplication circuits (namely compute/uncompute) use n (doubly-controlled) modular additions. The modular addition is constructed using two (controlled) addition circuits, which have a Toffoli count of $T_{add}(n) = 8n \log_2 n + \mathcal{O}(n)$. Thus we have

$$T_{mult}(n) = 4nT_{add}(n) = 32n^2 \log_2 n + \mathcal{O}(n^2) \ .$$

The experimental data and the fit confirm this expected scaling, as shown in Fig. 7.7. Since $2n$ modular multiplications have to be carried out for an entire run of Shor's algorithm, the overall Toffoli count is

$$T_{\text{Shor}}(n) = 64n^3 \log_2 n + \mathcal{O}(n^3) \ .$$

7.4 Advantages of Toffoli circuits

7.4.1 Single-qubit rotation gate synthesis

In order to apply a QEC scheme, arbitrary rotation gates have to be decomposed into sequences of gates from a (universal) discrete gate set—a process called *gate synthesis*—for which an algorithm such as the ones in Refs. [27, 73, 104] may be used. One example of a universal discrete gate set consists of the Pauli gates $(\sigma_x, \sigma_y, \sigma_z)$, CNOT, Hadamard, and the T gate $\left(\begin{smallmatrix} 1 & 0 \\ 0 & e^{i\pi/4} \end{smallmatrix}\right)$. This synthesis implies a growth on the order of $\Theta(\log \frac{1}{\varepsilon})$ in the total number of gates, where ε denotes the target precision of the synthesis.

In space-efficient implementations of Shor's algorithm by Beauregard [90] and Takahashi et al. [102], the angles of the approximate QFT (AQFT) require synthesis. From the total number of gates, the target precision of the gate synthesis can be estimated to be in $\omega(\frac{1}{n^3})$. Therefore, the overall gate count and depth of the previous circuits by Takahashi et al. and Beauregard are in $\Theta(n^3 \log^2 n)$ and $\Theta(n^3 \log n)$, respectively.

Toffoli based networks, on the other hand, do not suffer from synthesis overhead. A Toffoli gate can be decomposed exactly into Clifford and T gates using 7 T-gates, or less if global phases can be ignored [26, 107]. While the rotation gates from the semi-classical inverse QFT (see Fig. 7.1) require synthesis, this does not affect the asymptotic scaling. Therefore, the overall gate count and depth of our circuit remain in $\mathcal{O}(n^3 \log n)$ and $\mathcal{O}(n^3)$, respectively.

7.4.2 Design for testability

In classical computing, thoroughly tested hardware and software components are preferred over the ones that are not, especially for applications where system-failure could have catastrophic effects. The same may be true for quantum computing: Both software and hardware will need to be tested in order to guarantee the correctness of each and every component involved in a computation for building large circuits such as the ones used for factoring using Shor's algorithm. While a full functional simulation may be possible for arbitrary circuits up to almost 50 qubits with high-performance simulators run on supercomputers [5, 84], simulating a moderately-sized future quantum computer with just 100 qubits is not feasible on a (future) classical computer due to the exponential scaling of the required resources. For Toffoli networks, on the other hand, classical reversible simulators can be used, which run the circuit on a computational basis state and only update one single state for each gate. This enables thorough testing of logical level circuits such as the modular multiplication circuit presented in this chapter.

7.5 Conclusion

We presented a Toffoli based in-place addition circuit which can be used to implement Shor's algorithm using $2n + 2$ qubits. Our implementation features a size in $\mathcal{O}(n^3 \log n)$, and a depth in $\mathcal{O}(n^3)$. In contrast to previous space-efficient implementations [90, 102], our modular multiplication circuit only consists of Toffoli and Clifford gates. In addition to facilitating the process of debugging future implementations, having a Toffoli based circuit also eliminates the need for single-qubit-rotation synthesis when employing quantum error-correction. This results in a better scaling of both size and depth by a factor in $\Theta(\log n)$.

Our main technical innovation is the implementation of an addition by a constant that can be performed in $\mathcal{O}(n \log n)$ operations and that uses between 1 and n ancillas, all of which can be dirty, i.e., can be taken from other parts of the computation that are currently idle.

As mentioned in [102], it would be interesting to see whether one can find a linear-time constant-adder which does not require $\Theta(n)$ clean ancilla qubits. This would allow to decrease the size of our circuit to its current depth of $\mathcal{O}(n^3)$ without having to increase the total number of qubits to $3n + 2$. Also, similar to [108, 109], it would be interesting to find implementations of Shor's algorithm that are geometrically constrained but yet make use of dirty ancillas to reduce the overall number of qubits required.

In subsequent work [100], the number of qubits was reduced from $2n + 2$ to $2n + 1$, albeit at the expense of a large increase in circuit depth [100]. However, this perfectly illustrates how the methods presented in this chapter and in later work [100] can be employed to arrive at new implementations which trade space (number of qubits) for time (circuit depth).

Chapter 8

Optimizing quantum circuits for arithmetic

Besides integer arithmetic which is required, for instance, in Shor's algorithm for factoring, many promising applications of quantum computing involve functions of fractional numbers.

In this chapter, which is a slightly modified version of Ref. [7], we thus design and present new quantum circuits for fixed-point arithmetic which can be added to any quantum software stack, e.g., LiQ$Ui\,|\rangle$ [95], Quipper [37], ScaffCC [36], Q# [66], and ProjectQ [9]. In particular, we discuss the implementation of general smooth functions via a piecewise polynomial approximation, followed by functions that are used in specific applications. We analyze the costs of implementing an inverse square root ($1/\sqrt{x}$) using a reversible fixed-point version of the method used in the computer game Quake III Arena [110] and we then combine this with our evaluation scheme for smooth functions in order to arrive at an implementation of $\arcsin(x)$.

Having reversible implementations of these functions available enables more detailed cost analyses of various quantum algorithms such as HHL [101], where the inverse square root can be used to arrive at $x \mapsto 1/x$ and $\arcsin(x)$ can be used to get $1/x$ from the computational basis state into the amplitude. Similar use cases arise in Quantum Metropolis sampling [45], Gibbs state preparation [111] and in the widely applicable framework of Quantum Rejection Sampling [47] to transform one or more samples of a given quantum state into a quantum state with potentially different amplitudes, while maintaining relative phases. In all these examples the computation of $\arcsin(x)$ is useful for the rejection sampling step. Further applications of numerical functions can be anticipated in quantum machine learning, where sigmoid functions may need to be evaluated on a superposition of values employing $\tanh(x)$, and $1/\sqrt{x}$ can be used for (re-)normalization of intermediate results [112]. In quantum algorithms for chemistry, further examples for numerical functions arise for on-the-fly computation of the one- and two-body integrals [71]. There, $1/\sqrt{x}$ as well as the evaluation of smooth functions such as Gaussians is needed. Similarly, on-the-fly computation of finite element matrix elements often involves the evaluation of functions such as $\sin(x)$ and $\cos(x)$ [46].

As a result of the large impact that the implementation details of such functions may have on the practicality of a given quantum algorithm, there is a vast number of circuits in the literature for various low-level arithmetic functions such as addition [69, 35, 44, 43]. Furthermore, Refs. [113, 114, 115] discuss implementations of higher-level arithmetic functions such as $\sin(x)$, $\arcsin(x)$ and $\sqrt{x}$ which we also consider in the present work, although using different approaches. In particular, our piecewise polynomial evaluation circuit enables evaluating piecewise smooth functions to high accuracy using polynomials of very low degree. As a result, we require only a small number of additions and multiplications, and few quantum registers to hold intermediate results in order to achieve reversibility. While Ref. [113] employs several evaluations of the $\sin(x)$ function in order to hone in on the actual value of its inverse, our implementation of $\arcsin(x)$ features costs that are similar to just one invocation of $\sin(x)$ for $x \in [-0.5, 0.5]$. Otherwise, if $x \in [-1, 1]$, our implementation also requires an evaluation of the square root. For evaluating inverse square roots, we optimize the initial guess which was also used in Ref. [114] in order to reduce the number of required Newton iterations by 1 (which corresponds to a reduction by 20-25%). In contrast to the mentioned works, we implement all our high-level arithmetic functions at the level of Toffoli gates in the quantum programming language LIQ$Ui\,|\rangle$. As a result, we were able to test our circuits on various test vectors using a Toffoli circuit simulator, ranging up to several hundreds of qubits.

We adapt ideas from classical high-performance computing in order to reduce the required resources in the quantum setting. While the methods we introduce allow reducing Toffoli and qubit counts significantly, the resulting circuits are still expensive, especially in terms of the number of gates that are required. We expect that this highlights the fact that more research in the implementation of quantum algorithms is necessary in order to further reduce the cost originating from arithmetic in the computational basis.

8.1 Learning from classical libraries

While there is no need for computations to be reversible when using classical computers, a significant overlap of techniques from reversible computing can be found in vectorized high-performance libraries. In quantum computing, having an if-statement collapses the state vector, resulting in a loss of all potential speedup. Similarly, if-statements in vectorized code require a read-out of the vector, followed by a case distinction and a read-in of the handled values, which incurs a tremendous overhead and results in a deterioration of the expected speedup or even an overall slowdown. Analogous considerations have to be taken into account when dealing with, e.g., loops. Therefore, classical high-performance libraries may offer ideas and insights applicable to quantum computing, especially for mathematical functions such as (inverse) trigonometric functions, exponentials, logarithms, etc., of which highly-optimized implementations are available in, e.g., the Cephes math library [116] or games such as Quake III Arena (their fast inverse square root [110] is reviewed in [117]).

Although some of these implementations rely on a floating-point representation, many ideas carry over to the fixed-point domain, and remain efficient enough even when requiring reversibility. Specifically, we adapt implementations of the arcsine function from [116] and the fast inverse square root from [117] to the quantum domain by providing reversible low-level implementations. Furthermore, we describe a parallel version of the classical Horner scheme [118], which enables the conditional evaluation of many polynomials in parallel and, therefore, efficient evaluation of piecewise polynomial approximations.

8.2 Evaluation of piecewise polynomial approximations

A basic scheme to evaluate a single polynomial on a quantum computer in the computational basis is the classical Horner scheme, which evaluates

$$P(x) = \sum_{i=0}^{d} a_i x^i$$

by iteratively performing a multiplication by x, followed by an addition of a_i for $i \in \{d, d-1, ..., 0\}$. This amounts to performing the following operations:

$$\begin{aligned} a_d x + a_{d-1} &\mapsto a_d x^2 + a_{d-1} x + a_{d-2} \\ &\cdots \\ &\mapsto a_d x^d + \cdots + a_0 \ . \end{aligned}$$

A reversible implementation of this scheme simply stores all intermediate results. At iteration i, the last iterate y_{i-1} is multiplied by x into a new register y_i, followed by an addition by the (classically-known) constant a_i, which may make use, e.g., the addition circuit by Takahashi [69] (if there is an extra register left), or the in-place constant adder by Häner et al. [6], which does not require an ancilla register but is more costly in terms of gates. Due to the linear dependence of

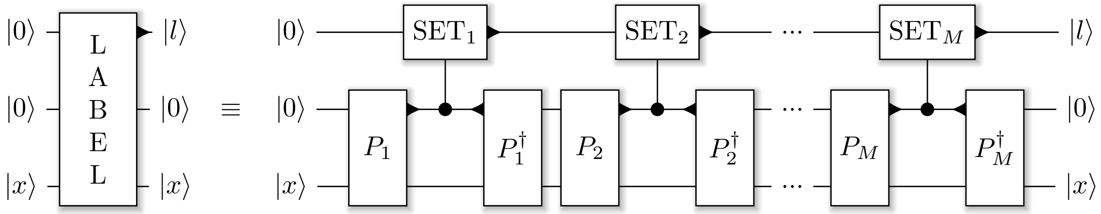


Figure 8.1: The LABEL gate initializes the label register $|l\rangle$, which consists of $\lceil \log_2(M) \rceil$ qubits, to indicate the subdomain Ω_l to which x belongs. P_i computes the predicate indicating whether $x \in \Omega_i$ into the ancilla qubit. Conditioned on this result, the label is then initialized to the value chosen to represent the i-th interval.

successive iterates, a pebbling strategy can be employed in order to optimize the space/time trade-offs according to some chosen metric [89].

Oftentimes, the degree d of the minimax approximation over a domain Ω must be chosen to be very high in order to achieve a certain $L_\infty(\Omega)$-error. In such cases, it makes sense to partition Ω, i.e., find Ω_i such that

$$\Omega = \bigcup_{i=0}^{M} \Omega_i \ , \ \Omega_i \cap \Omega_j = \emptyset \ \forall i \neq j \ ,$$

and to then perform a case distinction for each input, evaluating a different polynomial for $x \in \Omega_i$ than for $y \in \Omega_j$ if $i \neq j$. A straight-forward generalization of this approach to the realm of quantum computing would loop over all subdomains Ω_i and, conditioned on a case-distinction or label register $|l\rangle$, evaluate the corresponding polynomial. Thus, the cost of this inefficient approach grows linearly with the number of subdomains.

In order to improve upon this approach, one can parallelize the polynomial evaluation if the degree d is constant over the entire domain Ω. Note that merely adding the label register $|l\rangle$ mentioned above and performing

$$\begin{aligned} |y_{l,i-1}x\rangle\,|0\rangle\,|l\rangle &\mapsto |y_{l,i-1}x\rangle\,|a_{l,i}\rangle\,|l\rangle && (8.1)\\ &\mapsto |y_{l,i-1}x + a_{l,i}\rangle\,|a_{l,i}\rangle\,|l\rangle && (8.2)\\ &\mapsto |y_{l,i}\rangle\,|0\rangle\,|l\rangle \ , && (8.3) \end{aligned}$$

enables the evaluation of multiple polynomials in parallel. Despite this, the resource requirements are nearly identical to the circuit for evaluating a single polynomial, as will be shown in more detail in Appendix 8.B. We note that the depth of the circuit is the same, since the initialization step (8.1) can be performed while multiplying the previous iterate y_{i-1} by x, see Fig. 8.2. An illustration of the circuit computing the label register $|l\rangle$ can be found in Fig. 8.1. A slight drawback

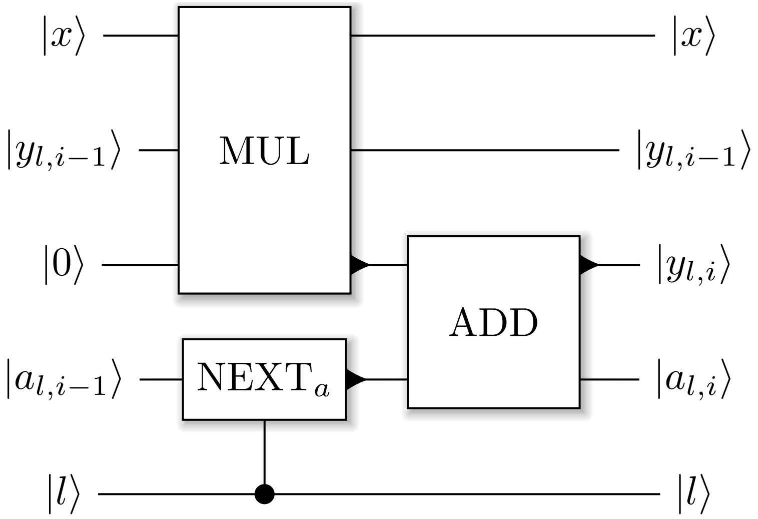


Figure 8.2: Our parallel polynomial evaluation circuit. NEXT$_a$ changes the register to hold the next set of coefficients (in superposition) $\sum_l |l\rangle |a_{l,i-1}\rangle \mapsto \sum_l |l\rangle |a_{l,i}\rangle$. MUL and ADD perform a multiplication and an addition, respectively. The small triangle indicates the output of the ADD and MUL gates.

of this parallel evaluation is that it requires one extra ancilla register for the last iteration, since the in-place addition circuit [6] can no longer be used. Resource estimates of a few functions which were implemented using this approach can be found in Table 8.E.1. The small overhead of using many intervals allows to achieve good approximations already for low-degree polynomials (and thus using few qubit registers).

Using reversible pebble games [119], it is possible to trade the number of registers needed to store the iterates with the depth of the resulting circuit. The parameters are: the number n of bits per register, the total number m of these n-qubit registers, the number r of Horner iterations, and the depth d of the resulting circuit. The trade-space we consider involves m, r, and d. In particular, we consider the question of what the optimal circuit depth is for a fixed number m of registers and a fixed number r of iterations. As in [120, 89] we use dynamic programming to construct the optimal strategies as the dependency graph is just a line which is due to the sequential nature of Horner's method (the general pebbling problem is much harder to solve, in fact finding the optimal strategy for general graphs is known to be PSPACE complete [121]). The optimal number of pebbling steps as a function of m and r can be found in Table 8.1.

$m\backslash r$	1	2	3	4	5	6	7	8	16	32	64
1	1	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞
2	1	3	∞	∞	∞	∞	∞	∞	∞	∞	∞
3	1	3	5	9	∞	∞	∞	∞	∞	∞	∞
4	1	3	5	7	11	15	19	25	∞	∞	∞
5	1	3	5	7	9	13	17	21	71	∞	∞
6	1	3	5	7	9	11	15	19	51	193	∞
7	1	3	5	7	9	11	13	17	49	145	531
8	1	3	5	7	9	11	13	15	47	117	369

Table 8.1: Optimal pebbling strategies for fixed number m of registers and fixed number r of sequential iterations. This table can be used for both the Horner scheme for polynomial evaluation, where r corresponds to the polynomial degree, and for Newton's method, where r denotes the number of iterations. The number for entry (m, r) denotes how many pebbling steps it takes to compute the entire sequence. The circuit width and depth can be obtained from these numbers.

8.3 Software stack module for piecewise smooth functions

In order to enable automatic compilation of an oracle which implements a piecewise smooth function, the Remez algorithm [122] can be used in a subroutine to determine a piecewise polynomial approximation, which can then be implemented using the circuit described in the previous section.

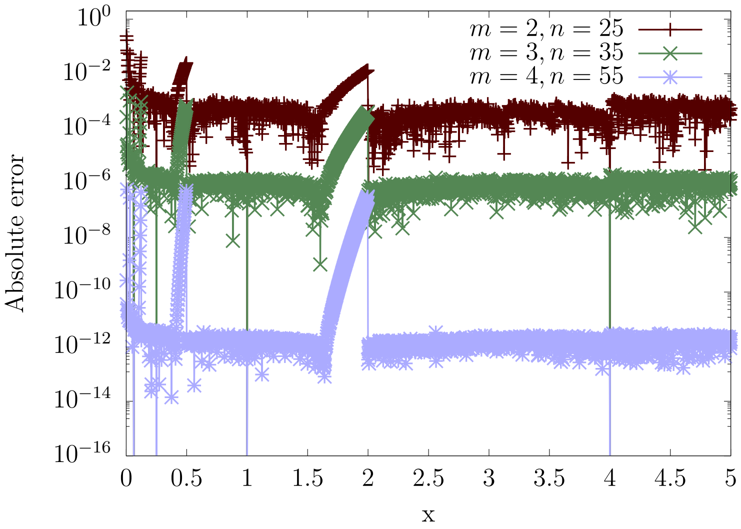
In particular, we aim to implement the oracle with a given precision, accuracy, and number of available quantum registers (or, equivalently, the polynomial degree d if no pebbling is employed) over a user-specified interval $\Omega = [a, a + L)$. Our algorithm proceeds as follows: In a first step, run the Remez algorithm which, given a function $f(x)$ over a domain $\Omega \subset \mathbb{R}$ and a polynomial degree d, finds the polynomial $P(x)$ which approximates $f(x)$ with minimal $L_\infty(\Omega)$-error, and check whether the achieved error is low enough. If it is too large, reduce the size of the domain $\Omega_1 := [a, a + \frac{L}{2})$ and check again. Repeating this procedure and carrying out binary search on the right interval border will eventually lead to the first subdomain $\Omega_1 = [a, b_1)$ which is the largest interval such that the corresponding degree d polynomial achieves the desired accuracy. Next, one determines the next subdomain $\Omega_2 = [b_1, b_2)$ using the same procedure. This is iterated until $b_i \geq b$, meaning that all required subdomains and their corresponding polynomials have been determined and $f(x)$ can be implemented using a parallel polynomial evaluation circuit. This algorithm was implemented and then run for various functions, target accuracies, and polynomial degrees in order to determine approximate resource estimates for these parameters, see Table 8.E.1 in the appendix.

8.4 Inverse square root

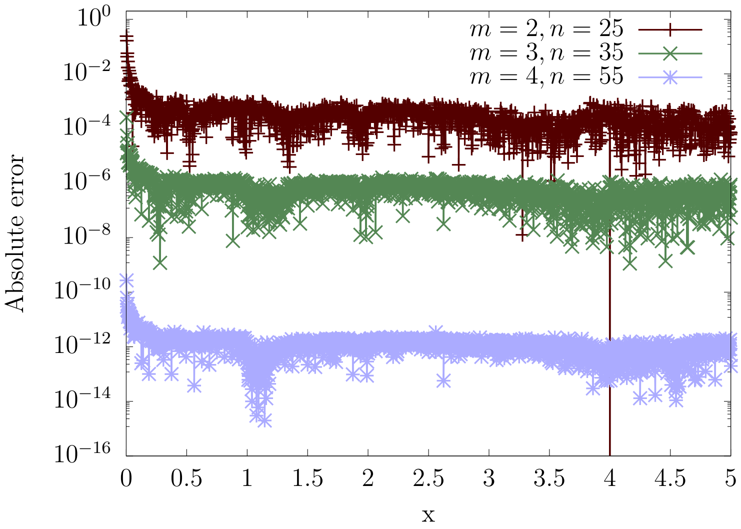
For quantum chemistry or machine learning applications, also non-smooth functions are required. Most notably, the inverse square root can be used in both examples, namely for the calculation of the Coulomb potential and to determine the reciprocal when employing HHL [101] for quantum machine learning.

In classical computing, inverse square roots appear in computer graphics and the term “fast inverse square root” is often used: It labels the procedure to approximate the inverse square root using bit-operations on the floating-point representation of the input, as it was done in Quake III Arena [110] (see [117] for a review). The code ultimately performs a Newton-Raphson iteration in order to improve upon a pretty accurate initial guess, which it finds using afore-mentioned bit-operations. Loosely speaking, the bit-operations consist of a bit-shift to divide the exponent by two in order to approximate the square root, followed by a subtraction of this result from a *magic number*, effectively negating the exponent and correcting the mantissa, which was also shifted together with the exponent. The *magic number* can be chosen using an auto-tuning procedure and varies depending on the objective function being used [117]. This provides an extremely good initial guess for the Newton iteration at very low cost.

In our reversible implementation, we use a similar procedure to compute the inverse square root using fixed-point arithmetic. While we cannot make use of the floating-point representation, we can still find a low-cost initial guess which allows for a small number of Newton iterations to be sufficient (i.e., 2-4 iterations). This includes determining the position of the first one in the bit-representation of the input, followed by an initialization which involves a case distinction on the *magic number* to use. Our three magic constants (see Appendix 8.C) were tuned such that the error peaks near powers of two in Fig. 8.3a vanish. The peaks appear due to the fact that the initial guess takes into account the location of the first one but completely ignores the actual magnitude of the input. For example, all inputs in $[1, 2)$ yield the same initial guess. The error plot with tuned constants is depicted in Fig. 8.3b. One can clearly observe that an entire Newton iteration can be saved when aiming for a given L_∞-error.



(a) Before constant-tuning.



(b) After constant-tuning.

Figure 8.3: Absolute errors of the inverse square root before and after tuning the constants (see Eqn. 8.5). The errors were evaluated for $N = 2000$ (equidistant) points in the interval $[\frac{1}{N}, 5]$ using $m \in \{2, 3, 4\}$ Newton iterations and corresponding bit sizes $n \in \{25, 35, 55\}$. The fixed-point position is $p = 12$, in order to ensure that no overflow occurs for small inputs.

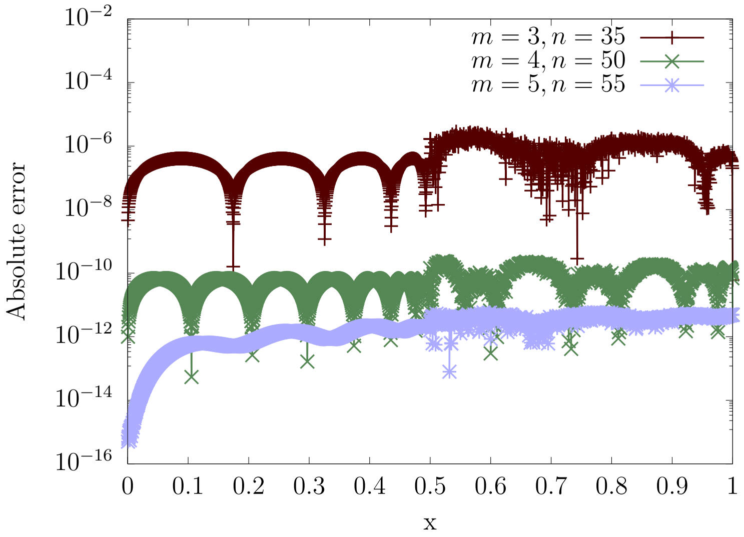


Figure 8.4: Absolute error on $[0, 1]$ for $N = 2000$ points of our reversible implementation of the arcsine using $m \in \{3, 4, 5\}$ Newton iterations for calculating the inverse square root. The fixed-point position is chosen to be $p = 2$ and total bit size n was chosen to be in $\{35, 50, 55\}$.

8.5 Arcsine

Following the implementation used in the classical math library Cephes [116], an arcsine can be implemented as a combination of polynomial evaluation and the square root. Approximating the arcsine using only a polynomial allows for a good approximation in $[-0.5, 0.5]$, but not near ± 1 (where it diverges). The Cephes math library remedies this problem by adding a case distinction, employing a "double-angle identity" for $|x| \geq 0.5$. This requires computing the square root, which can be achieved by first calculating the inverse square root, followed by $x \cdot \frac{1}{\sqrt{x}} = \sqrt{x}$. Alternatively, the new square root circuit from Ref. [115] can be used.

We have implemented our circuit for arcsine and we show the resulting error plot in Fig. 8.4. The oscillations stem from the minimax polynomial which is used to approximate the arcsine on $[-0.5, 0.5]$. More implementation details and resource estimates can be found in Appendix 8.D.

Note that certain applications may allow to trade off error in the arcsine with, e.g., probability of success by rescaling the input such that the arcsine needs to be computed only for values in $[-0.5, 0.5]$. This would allow one to remove the case-distinction and the subsequent calculation of the square root: One could evaluate the arcsine at a cost that is similar to the implementation costs of sin/cos.

Estimates for the Toffoli and qubit counts for this case can also be found in the appendix, see Table 8.E.1.

8.6 Conclusion

We have presented efficient quantum circuits for the evaluation of many mathematical functions, including (inverse) square root, Gaussians, hyperbolic tangent, exponential, sine/cosine, and arcsine. Our circuits can be used to obtain accurate resource estimates for various quantum algorithms and the results may help to identify the first large-scale applications as well as bottlenecks in these algorithms where more research is necessary in order to make the resource requirements practical. When embedded in a quantum compilation framework, our general parallel polynomial evaluation circuit can be used for automatic code generation when compiling oracles that compute piecewise smooth mathematical functions in the computational basis. This tremendously facilitates the implementation of quantum algorithms which employ oracles that compute such functions on a superposition of inputs.

Appendix: Details

This appendix contains implementation details, resource estimates, and simulation results for all mathematical functions that were presented in chapter 8. This appendix was previously published in Ref. [7].

8.A Basic circuit building blocks for fixed-point arithmetic

In fixed-point arithmetic, one represents numbers x using n bits as

$$x = \underbrace{x_{n-1} \cdots x_{n-p}}_{p} . \underbrace{x_{n-p-1} \cdots x_0}_{n-p} ,$$

where $x_i \in \{0, 1\}$ is the i-th bit of the binary representation of x, and the point position p denotes the number of binary digits to the left of the binary point. We choose both the total number of bits n and the point position p to be constant over the course of a computation. As a consequence, over- and underflow errors are introduced, while keeping the required bit-size from growing with each operation.

Fixed-point addition. We use a fixed-point addition implementation, which keeps the bit-size constant. This amounts to allowing over- and underflow, while keeping the registers from growing with each operation.

Fixed-point multiplication. Multiplication can be performed by repeated-addition-and-shift, which can be seen from

$$x \cdot y = x_{n-1} 2^{n-1} y + \cdots + x_0 2^0 y ,$$

where $x = \sum_i x_i 2^i$ with $x_i \in \{0, 1\}$ denotes the binary expansion of the n-bit number x. Thus, for $i \in \{0, ..., n-1\}$, $2^{i-(n-p)} y$ is added to the result register (which is initially zero) if $x_i = 1$. This can be implemented using n controlled additions on

$1, 2, ..., n$ bits if one allows for pre-truncation: Instead of computing the $2n$-bit result and copying out the first n bits before uncomputing the multiplication again, the additions can be executed on a subset of the qubits, ignoring all bits beyond the scope of the n-bit result. Thus, each addition introduces an error of at most $\varepsilon_A = \frac{1}{2^{n-p}}$. Since there are (at most) n such additions, the total error is

$$\varepsilon = \frac{n}{2^{n-p}} \ ,$$

a factor n larger than using the costly approach mentioned above.

Negative multipliers are dealt with by substituting the controlled addition by a controlled subtraction when conditioning on the most significant bit [123] because it has negative weight $w_{MSB} = -2^{n-1}$ in two's-complement notation. The multiplicand is assumed to be positive throughout, which removes the need for conditional inversions of input and output (for every multiplication), thus tremendously reducing the size of circuits that require many multiplications such as, e.g., polynomial evaluation.

Fixed-point squaring. The square of a number can be calculated using the same approach as for multiplication. Yet, one can save (almost) an entire register by only copying out the bit being conditioned on prior to performing the controlled addition. Then, the bit can be reset using another CNOT gate, followed by copying out the next bit and performing the next controlled addition. The gate counts are identical to performing

$$|x\rangle |0\rangle |0\rangle \mapsto |x\rangle |x\rangle |0\rangle \mapsto |x\rangle |x\rangle \left|x^2\right\rangle \mapsto |x\rangle \left|x^2\right\rangle |0\rangle \ ,$$

while allowing to save $n-1$ qubits.

8.B Resource estimates for polynomial evaluation

The evaluation of a degree d polynomial requires an initial multiplication $a_d \cdot x$, an addition of a_{d-1}, followed by $d-1$ multiply-accumulate instructions. The total number of Toffoli gates is thus equal to the cost of d multiply-accumulate instructions. Furthermore, $d+1$ registers are required for holding intermediate and final result(s) if no in-place adder is used for the last iteration (and no non-trivial pebbling strategy is applied). Other strategies may be employed in order to reduce the number of ancilla registers, at the cost of a larger gate count, see Table 8.1 for examples.

Note that all multiplications can be carried out assuming $x > 0$, i.e. x can be conditionally inverted prior to the polynomial evaluation (and the pseudo-sign bit is copied out). The sign is then absorbed into the coefficients: Before adding a_i into the $|y_{i-1}x\rangle$-register, it is inverted conditioned on the sign-bit of x being set if the coefficient corresponds to an odd power. This is done because it is cheaper to implement a fixed-point multiplier which can only deal with y_{i-1} being negative (see Sec. 8.A).

The Toffoli gate count of multiplying two n-bit numbers (using truncated additions as described in Sec. 8.A) is

$$\begin{aligned} T_{\mathrm{mul}}(n,p) &= \sum_{i=0}^{p-1} T_{\mathrm{cadd}}(n-i) + \sum_{i=1}^{n-p} T_{\mathrm{cadd}}(n-i) \\ &= \sum_{i=0}^{p-1} 3(n-i) + \sum_{i=1}^{n-p} 3(n-i) + 3n \\ &= \frac{3}{2}n^2 + 3np + \frac{3}{2}n - 3p^2 + 3p \end{aligned}$$

if one uses the controlled addition circuit by Takahashi et al. [69], which requires $3n+3$ Toffoli gates to (conditionally) add two n-bit numbers. The subsequent addition can be implemented using the addition circuit by Takahashi et al. [69], featuring $2n-1$ Toffoli gates. Thus, the total cost of a fused multiply-accumulate instruction is

$$T_{\mathrm{fma}}(n,p) = \frac{3}{2}n^2 + 3np + \frac{7}{2}n - 3p^2 + 3p - 1 \ .$$

Therefore, the total Toffoli count for evaluating a degree d polynomial is

$$T_{\mathrm{poly}}(n,d,p) = \frac{3}{2}n^2 d + 3npd + \frac{7}{2}nd - 3p^2 d + 3pd - d \ .$$

Evaluating M polynomials in parallel for piecewise polynomial approximation requires only $n + \lceil \log_2 M \rceil$ additional qubits (since one n-qubit register is required to perform the addition in the last iteration, which is no longer just a constant) and $2M$ $\lceil \log_2 M \rceil$-controlled NOT gates, which can be performed in parallel with the multiplication. This increases the circuit size by

$$T_{\mathrm{extra}}(M) = 2M(4\lceil \log_2 M \rceil - 8)$$

Toffoli gates per multiply-accumulate instruction, since a k-controlled NOT can be achieved using $4(k-2)$ Toffoli gates and $k-2$ dirty ancilla qubits [26], which are readily available in this construction.

The label register $|l\rangle$ can be computed using 1 comparator per subinterval

$$I_i = [a_i, a_{i_{i+1}}),\ a_0 < a_1 < \ldots < a_{M-1} \ .$$

The comparator stores its output into one extra qubit, flipping it to $|1\rangle$ if $x \leq a_{i+1}$. The label register is then incremented from $i-1$ to i, conditioned on this output qubit still being $|0\rangle$ (indicating that $x > a_i$). Incrementing $|l\rangle$ can be achieved using CNOT gates applied to the qubits that correspond to ones in the bit-representation of $(i-1) \oplus i$. Finally, the comparator output qubit is uncomputed again. This procedure is carried out M times for $i = 0, ..., M-1$ and requires 1 additional qubit. The number of extra Toffoli gates for this label initialization is

$$\begin{aligned} T_{\text{label}}(M,n) &= M \cdot 2T_{\text{cmp}}(n) \\ &= 4Mn \ , \end{aligned}$$

where, as a comparator, we use the CARRY-circuit from [6], which needs $2n$ Toffoli gates to compare a classical value to a quantum register, and another $2n$ to uncompute the output and intermediate changes to the n required dirty ancilla qubits.

In total, the parallel polynomial evaluation circuit thus requires

$$\begin{aligned} T_{\text{pp}}(n,d,p,M) &= T_{\text{poly}}(n,d,p) + d \cdot T_{\text{extra}}(M) \\ &\quad + T_{\text{label}}(M,n) \\ &= \frac{3}{2}n^2 d + 3npd + \frac{7}{2}nd - 3p^2 d + 3pd - d \\ &\quad + 2Md(4\lceil \log_2 M \rceil - 8) + 4Mn \end{aligned}$$

Toffoli gates and $(d+1)n + \lceil \log_2 M \rceil + 1$ qubits.

8.C (Inverse) Square root

The inverse square root, i.e.,

$$f(x) = \frac{1}{\sqrt{x}}$$

can be computed efficiently using Newton's method. The iteration looks as follows:

$$x_{n+1} = x_n \left(1.5 - \frac{a x_n^2}{2}\right) \ ,$$

where a is the input and $x_n \overset{n\to\infty}{\longrightarrow} \frac{1}{\sqrt{a}}$ if the initial guess is sufficiently close to the true solution.

8.C.1 Reversible implementation

Initial guess and first round. Finding a good initial guess $x_0 \approx \frac{1}{\sqrt{a}}$ for Newton's zero-finding routine is crucial for (fast) convergence. A crude approximation which turns out to be sufficient is the following:

$$\frac{1}{\sqrt{a}} = \left(2^{\log_2 a}\right)^{-\frac{1}{2}} = 2^{-\frac{\log_2 a}{2}} \approx 2^{\lfloor -\frac{\lfloor \log_2 a \rfloor}{2} \rceil} = \tilde{x}_0 \ ,$$

where $\lfloor \log_2 a \rfloor$ can be determined by finding the first "1" when traversing the bit-representation of a from left to right (MSB to LSB). While the space requirement for $\tilde{x}_0$ is in $\mathcal{O}(\log_2 n)$, such a representation would be impractical for the first Newton round. Furthermore, noting that the first iteration on $\tilde{x}_0 = 2^k$ leads to

$$\tilde{x}_1 = 2^k \left(1.5 - \frac{a2^{2k}}{2}\right) =: x_0 \ , \tag{8.4}$$

one can directly choose this x_0 as the initial guess. The preparation of x_0 can be achieved using $(n-1)+n+1$ ancilla qubits, which must be available due to the space requirements of the subsequent Newton steps. The one ancilla qubit is used as a flag indicating whether the first "1" from the left has already been encountered. For each iteration $i \in \{n-1, ..., 1, 0\}$, one determines whether the bit a_i is 1 and stores this result r_i in one of the n work qubits, conditioned on the flag being unset. Then, conditioned on $r_i = 1$, the flag is flipped, indicating that the first "1" has been found. If $r_i = 1$, the x_0-register is initialized to the value in (8.4) as follows: Using CNOTs, the x_0-register can be initialized to the value 1.5 shifted by $k = \frac{p-2i}{2}$, where p denotes the binary point position of the input, followed by subtracting the $(3k-1)$-shifted input a from x_0, which may require up to $n-1$ ancilla qubits.

In order to improve the quality of the first guess for numbers close to 2^k for some $k \in \mathbb{Z}$, one can tune the constant 1.5 in (8.4), i.e., turn it into a function $C(k)$ of the exponent k. This increases the overall cost of calculating x_0 merely by a few CNOT gates but allows to save an entire Newton iteration even when only distinguishing three cases, namely

$$C(k) := \begin{cases} 1.613, & k < 0 \\ 1.5, & k = 0 \\ 1.62, & k > 0 \end{cases} . \tag{8.5}$$

The Newton iteration. Computing x_{n+1} from x_n by

$$x_{n+1} = x_n \left(1.5 - \frac{ax_n^2}{2}\right) ,$$

can be achieved as follows:

1. Compute the square of x_n into a new register.
2. Multiply x_n^2 by the shifted input to obtain $ax_n^2/2$.
3. Initialize another register to 1.5 and subtract $ax_n^2/2$.

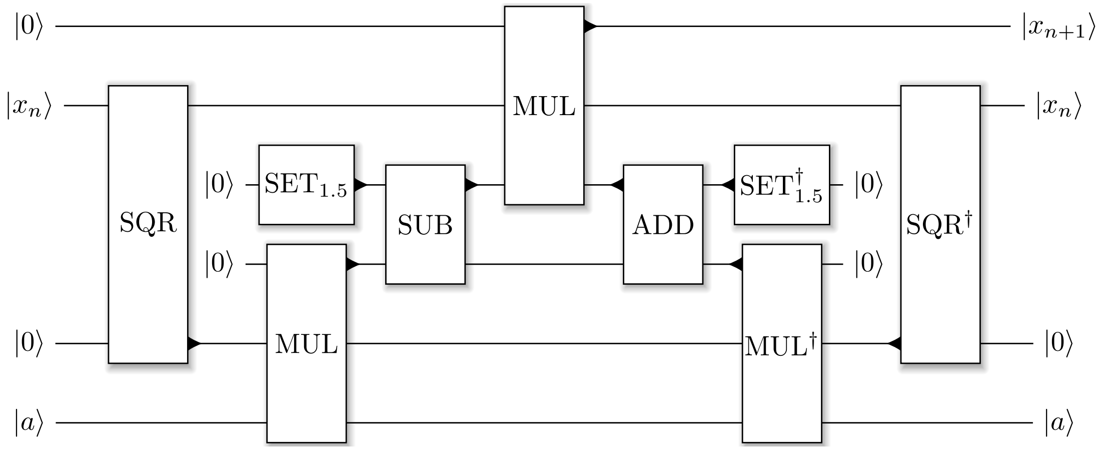


Figure 8.C.1: Circuit for the n-th Newton iteration of computing the inverse square root of a, given in a quantum superposition in $|a\rangle$. SQR computes the square of the previous iterate x_n into an empty result-register, which is then multiplied by the input a (MUL), followed by subtracting (SUB) this intermediate result from the value 1.5 (initialized using the $\mathrm{SET}_{1.5}$-gate). Finally, the next iterate, i.e., $x_{n+1} = x_n(1.5 - \frac{1}{2}ax_n^2)$ can be computed by multiplying this intermediate result by x_n. All temporary results are then cleared by running the appropriate operations in reverse order.

4. Multiply the result by x_n to arrive at x_{n+1}.
5. Uncompute the three intermediate results.

The circuit of one such Newton iteration is depicted in Fig. 8.C.1.

Therefore, for m Newton iterations, this requires $m + 3$ n-qubit registers if no pebbling is done on the Newton iterates, i.e., if all x_i are kept in memory until the last Newton iteration has been completed.

8.C.2 Resource estimates

Computing the initial guess for the fast inverse square root requires n controlled additions of two n-bit numbers plus $2n$ Toffoli gates for checking/setting the flag (and uncomputing it again). Thus, the Toffoli count for the initial guess is

$$T_{\text{init}}(n) = nT_{\text{cadd}}(n) + 2n = 3n^2 + 5n \ .$$

Each Newton iteration features squaring, a multiplication, a subtraction, a final multiplication (yielding the next iterate), and then an uncomputation of the three intermediate results. In total, one thus employs 5 multiplications and 2 additions (of which 2 multiplications and 1 addition are run in reverse), which yields the Toffoli count

$$\begin{aligned} T_{\text{iter}}(n,p) &= 5T_{\text{mul}}(n,p) + 2T_{\text{add}}(n) \\ &= \frac{15}{2}n^2 + 15np + \frac{23}{2}n - 15p^2 + 15p - 2 \ . \end{aligned}$$

The number of Toffoli gates for the entire Newton procedure (without uncomputing the iterates) for m iterations thus reads

$$\begin{aligned} T_{\text{invsqrt}}(n,m,p) &= T_{\text{init}}(n) + mT_{\text{iter}}(n,p) \\ &= n^2(\frac{15}{2}m + 3) + 15npm + n(\frac{23}{2}m + 5) \\ &\quad - 15p^2m + 15pm - 2m \ . \end{aligned}$$

Since each Newton iteration requires 3 ancilla registers (which are cleaned up after each round) to produce the next iterate, the total number of qubits is $n(m+4)$, where one register holds the initial guess x_0.

Note that this is an upperbound on the required number of both qubits and Toffoli gates. Since Newton converges quadratically, there is no need to perform full additions and multiplications at each iteration. Rather, the number of bits n used for the fixed point representation should be an (increasing) function of the Newton iteration.

The square root can be calculated using

$$\sqrt{x} = x \cdot \frac{1}{\sqrt{x}} \ ,$$

i.e., at a cost of an additional multiplication into a new register. Note that this new register would be required anyway when copying out the result and running the entire computation in reverse, in order to clear registers holding intermediate results. Thus, the total number of logical qubits remains unchanged.

8.D Arcsine

While $\sin(x)$ and $\cos(x)$ are very easy to approximate using, e.g., polynomials, their inverses are not. The main difficulty arises near ±1, where

$$\frac{d \arcsin(x)}{dx} = \frac{1}{\sqrt{1-x^2}}$$

diverges. Therefore, it makes sense to use an alternative representation of $\arcsin(x)$ for larger values of x, e.g.,

$$\begin{aligned}\arcsin(x) &= \frac{\pi}{2} - \arccos(x) \\ &= \frac{\pi}{2} - \arcsin\left(\sqrt{1-x^2}\right) \ .\end{aligned}$$

Applying the double-argument identity to the last expression yields

$$\arcsin(x) = \frac{\pi}{2} - 2\arcsin\left(\sqrt{\frac{1-x}{2}}\right) \ , \tag{8.6}$$

a very useful identity which was already used in a classical math library called Cephes [116]. We use the same partitioning of the interval, using a minimax polynomial to approximate $\arcsin(x)$ for $x \in [0, 0.5)$, and the transformation in (8.6) for $x \in [0.5, 1]$. We use our inverse square root implementation to compute $\sqrt{z}$ for

$$z = \frac{1-x}{2} \ ,$$

which satisfies $z \in [0, 0.25]$, for $x \in [0.5, 1]$. Therefore, the fixed point position has to be chosen large, as the inverse square root diverges for small x. Luckily, the multiplication by x after this computation takes care of the singularity and, since most bits of low-significance of $\frac{1}{\sqrt{x}}$ will cause underflow for small x, we can get away with computing a shifted version of the inverse square root. This optimization reduces the number of extra bits required during the evaluation of the inverse square root.

It is worth noting that in many applications, evaluating $\arcsin(x)$ only on the interval $[0, 0.5]$ may be sufficient. In such cases, the cost is much lower since this can be achieved using our parallel polynomial evaluation circuit. The Toffoli counts for this case can be found in Table 8.E.1.

8.D.1 Reversible implementation

The Arcsine is implemented as a combination of polynomial evaluation and the inverse square root to extend the polynomial approximation on $[0, 0.5]$ to the entire domain $[0, 1]$ employing the double-argument identity above. First, the (pseudo) sign-bit of x is copied out and x is conditionally inverted (modulo two's-complement) to ensure $x \geq 0$. Since there are plenty of registers available, this can be achieved by conditionally initializing an extra register to $|1\rangle$ and then using a normal adder to increment $\overline{x}$ by one, where $\overline{x}$ denotes the bit- or one's-complement of x. Since $x \in [0, 1]$, one can determine whether $x < 0.5$ using just one Toffoli gate (and 4 NOT gates). The result of this comparison is stored in an ancilla qubit denoted by $|a\rangle$. $z = (1-x)/2$ can be computed using an adder (run in reverse) acting on x shifted by one and a new register, after having initialized it to 0.5 using a NOT gate. Then, conditioned on $|\overline{a}\rangle$ (i.e., on a being 0), this result is copied into the polynomial input register $|p_{\text{in}}\rangle$ and, conditioned on $|a\rangle$, x is squared into $|p_{\text{in}}\rangle$. After having applied our polynomial evaluation circuit (which uncomputes intermediate results) to this input, $|p_{\text{in}}\rangle$ can be uncomputed again, followed by computing the square root of z. Then, the result of the polynomial evaluation must be multiplied by either $\sqrt{z}$ or x, which can be achieved using $2n$ controlled swaps and one multiplier. The final transformation of the result consists of an initialization to $\pi/2$ followed by a subtraction, both conditioned on $|\overline{a}\rangle$, and a copy conditioned on $|a\rangle$. Finally, the initial conditional inversion of x can be undone after having (conditionally) inverted the output.

8.D.2 Resource estimates

Following this procedure, the Toffoli count for this arcsine implementation on n-bit numbers using m Newton iterations for calculating $\sqrt{z}$ and a degree-d polynomial to approximate $\arcsin(x)$ on $[0, 0.5]$ can be written as

$$\begin{aligned}
T_{\text{arcsin}} &= 3T_{\text{inv}} + (2T_{\text{poly}} - T_{\text{fma}}) \\
&\quad + 2T_{\text{csquare}} + T_{\text{mul}} + T_{\text{cadd}} \\
&\quad + (2T_{\text{invsqrt}} + T_{\text{mul}}) + 5n + 2 \\
&\quad + T_{\text{add}} \\
&= 3T_{\text{add}} + 2T_{\text{poly}} + 3T_{\text{mul}} \\
&\quad + T_{\text{cadd}} + 2T_{\text{invsqrt}} + 9n + 2 \\
&= d(3n^2 + n(6p+7) - 6(p-1)p - 2) \\
&\quad + m(n(15n + 30p + 23) - 30p(p-1) - 4) \\
&\quad + 9(n+1)p + \frac{9}{2}n(n+1) \\
&\quad + 6n^2 + 28n - 9p^2 + 2
\end{aligned}$$

where $T_{\text{inv}}(n)$ denotes the Toffoli count for computing the two's-complement of an n-bit number and $T_{\text{csquare}}(n,p) = T_{\text{mul}}(n,p) + 2n$ is the number of Toffoli gates required to perform a conditional squaring operation. Furthermore, $2n$ Toffoli gates are needed to achieve the conditional n-bit swap operation (twice), and another $3n$ are used for (conditional) copies.

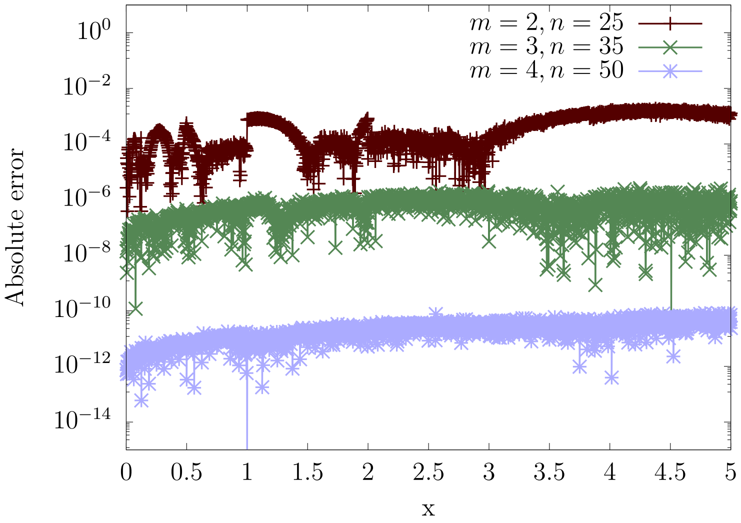


Figure 8.D.1: Absolute error on $[0, 5]$ for $N = 2000$ equidistant points of our reversible implementation of the square root for $m \in \{2, 3, 4\}$ Newton iterations and corresponding bit sizes $n \in \{25, 35, 50\}$. The fixed-point position is chosen to be $p = 5$.

8.E Simulation results

All circuits were implemented at the gate level and tested using a reversible simulator extension to LIQ$Ui\,|\rangle$. The results are presented in this section.

8.E.1 Piecewise polynomial approximation

A summary of the required resource for implementing $\tanh(x)$, $\exp(-x^2)$, and $\sin(x)$ can be found in Tbl. 8.E.1. For each function, one set of parameters was implemented reversibly at the level of Toffoli gates in order to verify the proposed circuits.

8.E.2 (Inverse) Square root

The convergence of our reversible fast inverse square root implementation with the number of Newton iterations can be found in Fig. 8.3b, where the bit sizes and point positions have been chosen such that the roundoff errors do not interfere significantly with the convergence. For all practical purposes, choosing between 3 and 5 Newton iterations should be sufficient. The effect of tuning the constants in the initial guess (see Eqn. 8.5) can be seen when comparing Fig. 8.3a to Fig. 8.3b:

The initial guess is obtained from the location of the first non-zero in the bit-representation of the input, which results in large rounding-effects for inputs close to an integer power of two. Tuning the initial guess results in almost uniform convergence, which allows to save an entire Newton iteration for a given L_∞-error.

The square root converges better than the inverse square root for small values, which can be expected, since

$$\sqrt{x} = x \cdot \frac{1}{\sqrt{x}}$$

has a regularizing effect for small x. The error after m Newton iterations when using n bits for the fixed point representation is depicted in Fig. 8.D.1. Additionally, the initial guess could be improved by tuning the constants in Eqn. 8.4 such that the error is minimal after multiplying $x \cdot \frac{1}{\sqrt{x}}$, instead of just optimizing for the inverse square root itself.

8.E.3 Arcsine

Our implementation of Arcsine uses both the polynomial evaluation and square root subroutines. The oscillatory behavior which can be seen in Fig. 8.4 is typical for minimax approximations. For $x > 0.5$, the resolution is lower due to the wider range of $\frac{1}{\sqrt{x}}$, which was accounted for by calculating a shifted version of the inverse square root. While this allows to save a few qubits (to the left of the binary point), the reduced number of qubits to the right of the binary point fail to resolve the numbers as well, which manifests itself by bit-noise for $x > 0.5$ in Fig. 8.4. The degrees of the minimax approximation were chosen to be 7, 13, and 17 for $m = 3, 4, 5$, respectively. Since $\arcsin(x)$ is an odd function, this amounts to evaluating a degree 3, 6, and 8 polynomial in x^2, followed by a multiplication by x.

Function	L_∞ error	Polynomial degree	Number of subintervals	Number of qubits	Number of Toffoli gates
$\tanh(x)$					
	10^{-5}				
		3	15	136	12428
		4	9	169	13768
		5	7	201	15492
		6	5	234	17544
	10^{-7}				
		3	50	166	27724

		4	23	205	23095
		5	14	244	23570
		6	10	284	26037
	10^{-9}				
		3	162	192	77992
		4	59	236	41646
		5	30	281	35460
		6	19	327	36578
$\exp(-x^2)$					
	10^{-5}				
		3	11	132	10884
		4	7	163	12141
		5	5	195	14038
		6	4	226	15863
	10^{-7}				
		3	32	161	20504
		4	15	199	19090
		5	10	238	21180
		6	7	276	23254
	10^{-9}				
		3	97	187	49032
		4	36	231	32305
		5	19	275	30234
		6	12	319	31595
$\sin(x)$					
	10^{-5}				
		3	2	113	6188
		4	2	141	7679
		5	2	169	9170
		6	2	197	10661
	10^{-7}				
		3	3	142	9444
		4	2	176	11480
		5	2	211	13720

		6	2	246	15960
	10^{-9}				
		3	7	167	13432
		4	3	207	15567
		5	2	247	18322
		6	2	288	21321
$\exp(-x)$					
	10^{-5}				
		3	11	116	8106
		4	6	143	8625
		5	5	171	10055
		6	4	198	11245
	10^{-7}				
		3	31	149	17304
		4	15	184	15690
		5	9	220	16956
		6	7	255	18662
	10^{-9}				
		3	97	175	45012
		4	36	216	28302
		5	19	257	25721
		6	12	298	26452
$\arcsin(x)$					
	10^{-5}				
		3	2	105	4872
		4	2	131	6038
		5	2	157	7204
		6	2	183	8370
	10^{-7}				
		3	3	134	7784
		4	2	166	9419
		5	2	199	11250
		6	2	232	13081
	10^{-9}				

		3	6	159	11264
		4	3	197	13138
		5	3	236	15672
		6	2	274	17938

Table 8.E.1: Costs associated with the evaluation of Gaussian, hyperbolic tangent, $\sin(x)$, $\exp(-x)$ for $x \geq 0$, and $\arcsin(x)$ on $[-0.5, 0.5]$ using piecewise polynomial approximation in combination with our parallel evaluation scheme. All Toffoli counts are for compute only (i.e., there is an additional factor of 2 for uncompute). For even/odd functions, the given degree corresponds to the evaluation cost, i.e., the actual polynomial being implemented has degree $2d$ or $2d + 1$, respectively.

Chapter 9

Conclusion and outlook

In this thesis, different aspects of code optimization for quantum computing were discussed in three parts. The first part focused on optimizations which can be performed automatically by compilers. A new type of optimizer was introduced which, by employing the Hoare triples of all invoked subroutines, is able to exploit optimization opportunities that could not be identified as such by previous approaches. In the last chapter of the first part, a methodology was presented which is capable of optimizing approximation errors during compilation in order to reduce the resource requirements of the compiled quantum program. Integration of our implementation into a software framework is ongoing work and will greatly facilitate acquiring resource estimates for complex quantum algorithms.

In the second part, two vastly different approaches to the simulation of quantum circuits were discussed. By employing code optimizations at the core, node, and cluster level, our implementation of a full state simulator successfully outperformed previous state-of-the-art simulations by more than an order of magnitude at every scale. After introducing the new concept of quantum circuit *emulation*, we showed that it is able to outperform direct simulation by several orders of magnitude, especially for circuits evaluating high-level mathematical functions. Despite the conceptual differences between the two approaches, both have proven to be valuable assets to quantum software development. As a consequence, both have been made publicly available via the ProjectQ framework [9].

In the third and final part of this thesis, new designs of quantum circuits for evaluating mathematical functions were introduced, since implementation details have significant ramifications on the run time of quantum programs. To this end, a new addition circuit was developed in chapter 7. By borrowing idle qubits from other parts of the computation, our circuit was shown to reduce the resource requirements of Shor's algorithm [13]. In addition, the resulting circuit is almost fully classical, making it amenable to testing using reversible logic simulators. Finally, optimized reversible implementations of various higher-level mathematical functions were developed and presented in chapter 8. The resulting resource estimates may serve as guidance for future work aiming to reduce the cost of arithmetic in quantum programs. Additionally, our implementations can be added as a quantum math library to any software framework for quantum computing.

List of publications

[1] Thomas Häner, Damian S Steiger, Krysta Svore, and Matthias Troyer. A software methodology for compiling quantum programs. *Quantum Science and Technology*, 3(2):020501, 2018. URL: `http://stacks.iop.org/2058-9565/3/i=2/a=020501`.

[2] Thomas Häner, Torsten Hoefler, and Matthias Troyer. Using Hoare logic for quantum circuit optimization. *ArXiv e-prints*, October 2018. `arXiv:1810.00375`.

[3] Thomas Häner, Martin Roetteler, and Krysta M. Svore. Managing approximation errors in quantum programs. *ArXiv e-prints*, July 2018. `arXiv:1807.02336`.

[4] Thomas Häner and Damian S. Steiger. 0.5 petabyte simulation of a 45-qubit quantum circuit. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '17, pages 33:1–33:10, New York, NY, USA, 2017. ACM. URL: `http://doi.acm.org/10.1145/3126908.3126947`, `doi:10.1145/3126908.3126947`.

[5] Thomas Häner, Damian S. Steiger, Mikhail Smelyanskiy, and Matthias Troyer. High performance emulation of quantum circuits. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '16, pages 74:1–74:9, Piscataway, NJ, USA, 2016. IEEE Press. URL: `http://dl.acm.org/citation.cfm?id=3014904.3015003`.

[6] Thomas Häner, Martin Roetteler, and Krysta M. Svore. Factoring using $2n + 2$ qubits with Toffoli based modular multiplication. *Quantum Info. Comput.*, 17(7-8):673–684, June 2017. URL: `http://dl.acm.org/citation.cfm?id=3179553.3179560`.

[7] Thomas Häner, Martin Roetteler, and Krysta M. Svore. Optimizing Quantum Circuits for Arithmetic. *ArXiv e-prints*, May 2018. `arXiv:1805.12445`.

[8] Thomas Haener, Mathias Soeken, Martin Roetteler, and Krysta M Svore. Quantum circuits for floating-point arithmetic. In *International Conference on Reversible Computation*, pages 162–174. Springer, 2018. URL: `https://doi.org/10.1007/978-3-319-99498-7_11`, `doi:10.1007/978-3-319-99498-7_11`.

[9] Damian S. Steiger, Thomas Häner, and Matthias Troyer. ProjectQ: an open source software framework for quantum computing. *Quantum*, 2:49, January 2018. URL: `https://doi.org/10.22331/q-2018-01-31-49`, `doi:10.22331/q-2018-01-31-49`.

[10] Damian S. Steiger, Thomas Häner, and Matthias Troyer. Advantages of a modular high-level quantum programming framework. *ArXiv e-prints*, June 2018. `arXiv:1806.01861`.

[11] Mathias Soeken, Thomas Haener, and Martin Roetteler. Programming quantum computers using design automation. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2018*, pages 137–146. IEEE, 2018. `doi:10.23919/DATE.2018.8341993`.

[12] J. R. McClean, I. D. Kivlichan, K. J. Sung, D. S. Steiger, Y. Cao, C. Dai, E. Schuyler Fried, C. Gidney, B. Gimby, P. Gokhale, T. Häner, T. Hardikar, V. Havlícek, C. Huang, J. Izaac, Z. Jiang, X. Liu, M. Neeley, T. O'Brien, I. Ozfidan, M. D. Radin, J. Romero, N. Rubin, N. P. D. Sawaya, K. Setia, S. Sim, M. Steudtner, Q. Sun, W. Sun, F. Zhang, and R. Babbush. OpenFermion: The Electronic Structure Package for Quantum Computers. *ArXiv e-prints*, October 2017. `arXiv:1710.07629`.

Bibliography

[13] Peter W. Shor. Algorithms for quantum computation: discrete logarithms and factoring. In *Proceedings 35th Annual Symposium on Foundations of Computer Science*, pages 124–134, Nov 1994. `doi:10.1109/SFCS.1994.365700`.

[14] Lov K. Grover. A fast quantum mechanical algorithm for database search. In *Proceedings of the Twenty-eighth Annual ACM Symposium on Theory of Computing*, STOC '96, pages 212–219, New York, NY, USA, 1996. ACM. URL: `http://doi.acm.org/10.1145/237814.237866`, `doi:10.1145/237814.237866`.

[15] Mario Szegedy. Quantum speed-up of Markov chain based algorithms. In *45th Annual IEEE Symposium on Foundations of Computer Science*, pages 32–41, Oct 2004. `doi:10.1109/FOCS.2004.53`.

[16] Richard P. Feynman. Simulating physics with computers. *International Journal of Theoretical Physics*, 21(6):467–488, Jun 1982. URL: `https://doi.org/10.1007/BF02650179`, `doi:10.1007/BF02650179`.

[17] John M. Pollard. Factoring with cubic integers. In Arjen K. Lenstra and Hendrik W. Lenstra, editors, *The development of the number field sieve*, pages 4–10, Berlin, Heidelberg, 1993. Springer Berlin Heidelberg. `doi:10.1007/BFb0091536`.

[18] Oded Regev. On lattices, learning with errors, random linear codes, and cryptography. In *Proceedings of the Thirty-seventh Annual ACM Symposium on Theory of Computing*, STOC '05, pages 84–93, New York, NY, USA, 2005. ACM. URL: `http://doi.acm.org/10.1145/1060590.1060603`, `doi:10.1145/1060590.1060603`.

[19] Andrew M. Childs, Richard Cleve, Enrico Deotto, Edward Farhi, Sam Gutmann, and Daniel A. Spielman. Exponential algorithmic speedup by a quantum walk. In *Proceedings of the Thirty-fifth Annual ACM Symposium on Theory of Computing*, STOC '03, pages 59–68, New York, NY,

USA, 2003. ACM. URL: http://doi.acm.org/10.1145/780542.780552, doi:10.1145/780542.780552.

[20] Gerald Teschl. Mathematical methods in quantum mechanics. *Graduate Studies in Mathematics*, 99, 2009. doi:10.1090/gsm/157.

[21] Charles H. Bennett. Logical reversibility of computation. *IBM J. Res. Dev.*, 17(6):525–532, November 1973. URL: http://dx.doi.org/10.1147/rd.176.0525, doi:10.1147/rd.176.0525.

[22] Juan I. Cirac and Peter Zoller. Quantum computations with cold trapped ions. *Phys. Rev. Lett.*, 74:4091–4094, May 1995. URL: https://link.aps.org/doi/10.1103/PhysRevLett.74.4091, doi:10.1103/PhysRevLett.74.4091.

[23] Andreas Wallraff, David I. Schuster, Alexandre Blais, L. Frunzio, R-S Huang, J. Majer, S. Kumar, Steven M. Girvin, and Robert J. Schoelkopf. Strong coupling of a single photon to a superconducting qubit using circuit quantum electrodynamics. *Nature*, 431(7005):162, 2004. doi:10.1038/nature02851.

[24] Frederic T. Chong, Diana Franklin, and Margaret Martonosi. Programming languages and compiler design for realistic quantum hardware. *Nature*, 549(7671):180, 2017. doi:10.1038/nature23459.

[25] David Deutsch, Adriano Barenco, and Artur Ekert. Universality in quantum computation. *Proceedings of the Royal Society of London A: Mathematical, Physical and Engineering Sciences*, 449(1937):669–677, 1995. URL: http://rspa.royalsocietypublishing.org/content/449/1937/669, arXiv:http://rspa.royalsocietypublishing.org/content/449/1937/669.full.pdf, doi:10.1098/rspa.1995.0065.

[26] Adriano Barenco, Charles H. Bennett, Richard Cleve, David P. DiVincenzo, Norman Margolus, Peter Shor, Tycho Sleator, John A. Smolin, and Harald Weinfurter. Elementary gates for quantum computation. *Phys. Rev. A*, 52:3457–3467, Nov 1995. URL: https://link.aps.org/doi/10.1103/PhysRevA.52.3457, doi:10.1103/PhysRevA.52.3457.

[27] Vadym Kliuchnikov, Dmitri Maslov, and Michele Mosca. Practical approximation of single-qubit unitaries by single-qubit quantum Clifford and T circuits. *IEEE Transactions on Computers*, 65(1):161–172, Jan. 2016. URL: doi.ieeecomputersociety.org/10.1109/TC.2015.2409842, doi:10.1109/TC.2015.2409842.

[28] Raban Iten, Roger Colbeck, Ivan Kukuljan, Jonathan Home, and Matthias Christandl. Quantum circuits for isometries. *Phys. Rev. A*, 93:032318, Mar 2016. URL: `https://link.aps.org/doi/10.1103/PhysRevA.93.032318`, `doi:10.1103/PhysRevA.93.032318`.

[29] Edwin Pednault, John A. Gunnels, Giacomo Nannicini, Lior Horesh, Thomas Magerlein, Edgar Solomonik, and Robert Wisnieff. Breaking the 49-Qubit Barrier in the Simulation of Quantum Circuits. *ArXiv e-prints*, October 2017. `arXiv:1710.05867`.

[30] David M. Miller, Mitchell A. Thornton, and David Goodman. A decision diagram package for reversible and quantum circuit simulation. In *2006 IEEE International Conference on Evolutionary Computation*, pages 2428–2435, July 2006. `doi:10.1109/CEC.2006.1688610`.

[31] Sergio Boixo, Sergei V. Isakov, Vadim N. Smelyanskiy, and Hartmut Neven. Simulation of low-depth quantum circuits as complex undirected graphical models. *ArXiv e-prints*, December 2017. `arXiv:1712.05384`.

[32] Charles A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, October 1969. URL: `http://doi.acm.org/10.1145/363235.363259`, `doi:10.1145/363235.363259`.

[33] Markus Reiher, Nathan Wiebe, Krysta M. Svore, Dave Wecker, and Matthias Troyer. Elucidating reaction mechanisms on quantum computers. *Proceedings of the National Academy of Sciences*, 2017. URL: `http://www.pnas.org/content/early/2017/06/30/1619152114`, `arXiv:http://www.pnas.org/content/early/2017/06/30/1619152114.full.pdf`, `doi:10.1073/pnas.1619152114`.

[34] David Poulin, Matthew B. Hastings, Dave Wecker, Nathan Wiebe, Andrew C. Doberty, and Matthias Troyer. The trotter step size required for accurate quantum simulation of quantum chemistry. *Quantum Info. Comput.*, 15(5-6):361–384, April 2015. URL: `http://dl.acm.org/citation.cfm?id=2871401.2871402`.

[35] Thomas G. Draper. Addition on a Quantum Computer. *eprint arXiv:quant-ph/0008033*, August 2000. `arXiv:quant-ph/0008033`.

[36] Ali JavadiAbhari, Shruti Patil, Daniel Kudrow, Jeff Heckey, Alexey Lvov, Frederic T. Chong, and Margaret Martonosi. ScaffCC: A Framework for Compilation and Analysis of Quantum Computing Programs. In *Proceedings of the 11th ACM Conference on Computing Frontiers*, CF '14, pages 1:1–1:10,

New York, NY, USA, 2014. ACM. URL: http://doi.acm.org/10.1145/2597917.2597939, doi:10.1145/2597917.2597939.

[37] Alexander S. Green, Peter LeFanu Lumsdaine, Neil J. Ross, Peter Selinger, and Benoît Valiron. Quipper: A scalable quantum programming language. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, pages 333–342, New York, NY, USA, 2013. ACM. URL: http://doi.acm.org/10.1145/2491956.2462177, doi:10.1145/2491956.2462177.

[38] Charles H. Bennett, Gilles Brassard, Claude Crépeau, Richard Jozsa, Asher Peres, and William K. Wootters. Teleporting an unknown quantum state via dual classical and einstein-podolsky-rosen channels. *Phys. Rev. Lett.*, 70:1895–1899, Mar 1993. URL: https://link.aps.org/doi/10.1103/PhysRevLett.70.1895, doi:10.1103/PhysRevLett.70.1895.

[39] Alberto Peruzzo, Jarrod McClean, Peter Shadbolt, Man-Hong Yung, Xiao-Qi Zhou, Peter J. Love, Alán Aspuru-Guzik, and Jeremy L O'brien. A variational eigenvalue solver on a photonic quantum processor. *Nature communications*, 5:4213, 2014. doi:10.1038/ncomms5213.

[40] Edward Farhi, Jeffrey Goldstone, and Sam Gutmann. A Quantum Approximate Optimization Algorithm. *ArXiv e-prints*, 2014. arXiv:1411.4028.

[41] Michael A Nielsen and Isaac L Chuang. *Quantum computation and quantum information*. Cambridge university press, 2010. URL: https://doi.org/10.1017/CBO9780511976667, doi:10.1017/CBO9780511976667.

[42] Andrew W. Cross, Lev S. Bishop, John A. Smolin, and Jay M. Gambetta. Open Quantum Assembly Language. *ArXiv e-prints*, July 2017. arXiv:1707.03429.

[43] Thomas G. Draper, Samuel A. Kutin, Eric M. Rains, and Krysta M. Svore. A logarithmic-depth quantum carry-lookahead adder. *Quantum Info. Comput.*, 6(4):351–369, July 2006. URL: http://dl.acm.org/citation.cfm?id=2012086.2012090.

[44] Steven A. Cuccaro, Thomas G. Draper, Samuel A. Kutin, and David Petrie Moulton. A new quantum ripple-carry addition circuit. *eprint arXiv:quant-ph/0410184*, October 2004. arXiv:quant-ph/0410184.

[45] Kristan Temme, Tobias J. Osborne, Karl Gerd Vollbrecht, David Poulin, and Frank Verstraete. Quantum Metropolis Sampling. *Nature*, 471(87), 2011. doi:10.1038/nature09770.

[46] Artur Scherer, Benoît Valiron, Siun-Chuon Mau, Scott Alexander, Eric van den Berg, and Thomas E. Chapuran. Concrete resource analysis of the quantum linear-system algorithm used to compute the electromagnetic scattering cross section of a 2D target. *Quantum Information Processing*, 16(3):60, Jan 2017. URL: `https://doi.org/10.1007/s11128-016-1495-5`, `doi:10.1007/s11128-016-1495-5`.

[47] Maris Ozols, Martin Roetteler, and Jérémie Roland. Quantum rejection sampling. *ACM Trans. Comput. Theory*, 5(3):11:1–11:33, August 2013. URL: `http://doi.acm.org/10.1145/2493252.2493256`, `doi:10.1145/2493252.2493256`.

[48] Dave Wecker, Matthew B. Hastings, Nathan Wiebe, Bryan K. Clark, Chetan Nayak, and Matthias Troyer. Solving strongly correlated electron models on a quantum computer. *Phys. Rev. A*, 92:062318, Dec 2015. URL: `https://link.aps.org/doi/10.1103/PhysRevA.92.062318`, `doi:10.1103/PhysRevA.92.062318`.

[49] Sergey B. Bravyi and Alexei Y. Kitaev. Quantum codes on a lattice with boundary. *eprint arXiv:quant-ph/9811052*, November 1998. `arXiv:quant-ph/9811052`.

[50] Peter W. Shor. Scheme for reducing decoherence in quantum computer memory. *Phys. Rev. A*, 52:R2493–R2496, Oct 1995. URL: `https://link.aps.org/doi/10.1103/PhysRevA.52.R2493`, `doi:10.1103/PhysRevA.52.R2493`.

[51] William K. Wootters and Wojciech H. Zurek. A single quantum cannot be cloned. *Nature*, 299(5886):802–803, 1982. `doi:10.1038/299802a0`.

[52] Sergio Boixo, Sergei V. Isakov, Vadim N. Smelyanskiy, Ryan Babbush, Nan Ding, Zhang Jiang, Michael J. Bremner, John M. Martinis, and Hartmut Neven. Characterizing quantum supremacy in near-term devices. *Nature Physics*, 14(6):595, 2018. URL: `https://doi.org/10.1038/s41567-018-0124-x`, `doi:10.1038/s41567-018-0124-x`.

[53] Scott Aaronson and Alex Arkhipov. The computational complexity of linear optics. In *Proceedings of the Forty-third Annual ACM Symposium on Theory of Computing*, STOC '11, pages 333–342, New York, NY, USA, 2011. ACM. URL: `http://doi.acm.org/10.1145/1993636.1993682`, `doi:10.1145/1993636.1993682`.

[54] Aram W. Harrow and Ashley Montanaro. Quantum computational supremacy. *Nature*, 549(7671):203, 2017. `doi:10.1038/nature23458`.

[55] Norbert M. Linke, Dmitri Maslov, Martin Roetteler, Shantanu Debnath, Caroline Figgatt, Kevin A. Landsman, Kenneth Wright, and Christopher Monroe. Experimental comparison of two quantum computing architectures. *Proceedings of the National Academy of Sciences*, 114(13):3305–3310, 2017. URL: `http://www.pnas.org/content/114/13/3305`, `arXiv:http://www.pnas.org/content/114/13/3305.full.pdf`, `doi:10.1073/pnas.1618020114`.

[56] Peter A. Milder, Franz Franchetti, James C. Hoe, and Markus Püschel. Computer generation of hardware for linear digital signal processing transforms. *ACM Transactions on Design Automation of Electronic Systems*, 17(2), 2012.

[57] Matteo Frigo and Steven G. Johnson. The design and implementation of FFTW3. *Proceedings of the IEEE*, 93(2):216–231, Feb 2005. `doi:10.1109/JPROC.2004.840301`.

[58] ProjectQ website, 2018. URL: `projectq.ch`.

[59] Yunseong Nam, Neil J. Ross, Yuan Su, Andrew M. Childs, and Dmitri Maslov. Automated optimization of large quantum circuits with continuous parameters. *npj Quantum Information*, 4(1):23, 2018. URL: `https://doi.org/10.1038/s41534-018-0072-4`, `doi:10.1038/s41534-018-0072-4`.

[60] Daniel Große, Xiaobo Chen, Gerhard W. Dueck, and Rolf Drechsler. Exact sat-based toffoli network synthesis. In *Proceedings of the 17th ACM Great Lakes Symposium on VLSI*, GLSVLSI '07, pages 96–101, New York, NY, USA, 2007. ACM. URL: `http://doi.acm.org/10.1145/1228784.1228812`, `doi:10.1145/1228784.1228812`.

[61] Daniel Große, Robert Wille, Gerhard W. Dueck, and Rolf Drechsler. Exact multiple-control toffoli network synthesis with sat techniques. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 28(5):703–715, May 2009. `doi:10.1109/TCAD.2009.2017215`.

[62] Matthew Amy, Dmitri Maslov, Michele Mosca, and Martin Roetteler. A meet-in-the-middle algorithm for fast synthesis of depth-optimal quantum circuits. *Trans. Comp.-Aided Des. Integ. Cir. Sys.*, 32(6):818–830, June 2013. URL: `http://dx.doi.org/10.1109/TCAD.2013.2244643`, `doi:10.1109/TCAD.2013.2244643`.

[63] Vadym Kliuchnikov, Dmitri Maslov, and Michele Mosca. Fast and efficient exact synthesis of single-qubit unitaries generated by Clifford and T gates. *Quantum Info. Comput.*, 13(7-8):607–630, July 2013. URL: `http://dl.acm.org/citation.cfm?id=2535649.2535653`.

[64] Giulia Meuli, Mathias Soeken, and Giovanni De Micheli. Sat-based {CNOT, T} quantum circuit synthesis. In *International Conference on Reversible Computation*, pages 175–188. Springer, 2018. doi:10.1007/978-3-319-99498-7_12.

[65] Robert S. Smith, Michael J. Curtis, and William J. Zeng. A Practical Quantum Instruction Set Architecture. *ArXiv e-prints*, August 2016. arXiv:1608.03355.

[66] Krysta Svore, Alan Geller, Matthias Troyer, John Azariah, Christopher Granade, Bettina Heim, Vadym Kliuchnikov, Mariia Mykhailova, Andres Paz, and Martin Roetteler. Q#: Enabling scalable quantum computing and development with a high-level dsl. In *Proceedings of the Real World Domain Specific Languages Workshop 2018*, RWDSL2018, pages 7:1–7:10, New York, NY, USA, 2018. ACM. URL: http://doi.acm.org/10.1145/3183895.3183901, doi:10.1145/3183895.3183901.

[67] IBM QISKit, 2018. URL: https://qiskit.org.

[68] Leonardo de Moura and Nikolaj Bjørner. Z3: An Efficient SMT Solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg. doi:10.1007/978-3-540-78800-3_24.

[69] Yasuhiro Takahashi, Seiichiro Tani, and Noboru Kunihiro. Quantum addition circuits and unbounded fan-out. *Quantum Info. Comput.*, 10(9):872–890, September 2010. URL: http://dl.acm.org/citation.cfm?id=2011464.2011476.

[70] Mingsheng Ying. Floyd–hoare logic for quantum programs. *ACM Trans. Program. Lang. Syst.*, 33(6):19:1–19:49, January 2012. URL: http://doi.acm.org/10.1145/2049706.2049708, doi:10.1145/2049706.2049708.

[71] Ryan Babbush, Dominic W. Berry, Ian D. Kivlichan, Annie Y. Wei, Peter J. Love, and Alán Aspuru-Guzik. Exponentially more precise quantum simulation of fermions in second quantization. *New Journal of Physics*, 18(3):033032, 2016. URL: http://stacks.iop.org/1367-2630/18/i=3/a=033032.

[72] Edward Fredkin and Tommaso Toffoli. Conservative logic. *International Journal of theoretical physics*, 21(3-4):219–253, 1982.

[73] Neil J. Ross and Peter Selinger. Optimal ancilla-free Clifford+T approximation of Z-rotations. *Quantum Info. Comput.*, 16(11-12):901–953, September 2016. URL: `http://dl.acm.org/citation.cfm?id=3179330.3179331`.

[74] Easwar Magesan, Daniel Puzzuoli, Christopher E. Granade, and David G. Cory. Modeling quantum noise for efficient testing of fault-tolerant circuits. *Phys. Rev. A*, 87:012324, Jan 2013. URL: `https://link.aps.org/doi/10.1103/PhysRevA.87.012324`, `doi:10.1103/PhysRevA.87.012324`.

[75] Mauricio Gutiérrez, Lukas Svec, Alexander Vargo, and Kenneth R. Brown. Approximation of realistic errors by Clifford channels and Pauli measurements. *Phys. Rev. A*, 87:030302, Mar 2013. URL: `https://link.aps.org/doi/10.1103/PhysRevA.87.030302`, `doi:10.1103/PhysRevA.87.030302`.

[76] Ethan Bernstein and Umesh Vazirani. Quantum complexity theory. *SIAM J. Comput.*, 26(5):1411–1473, 1997. `doi:10.1137/S0097539796300921`.

[77] Pierre Pfeuty. The one-dimensional ising model with a transverse field. *Annals of Physics*, 57(1):79 – 90, 1970. URL: `http://www.sciencedirect.com/science/article/pii/0003491670902708`, `doi:https://doi.org/10.1016/0003-4916(70)90270-8`.

[78] Dave Wecker, Bela Bauer, Bryan K. Clark, Matthew B. Hastings, and Matthias Troyer. Gate-count estimates for performing quantum chemistry on small quantum computers. *Phys. Rev. A*, 90:022305, Aug 2014. `doi:10.1103/PhysRevA.90.022305`.

[79] Dominic W. Berry, Andrew M. Childs, Richard Cleve, Robin Kothari, and Rolando D. Somma. Simulating Hamiltonian dynamics with a truncated Taylor series. *Phys. Rev. Lett.*, 114:090502, Mar 2015. URL: `https://link.aps.org/doi/10.1103/PhysRevLett.114.090502`, `doi:10.1103/PhysRevLett.114.090502`.

[80] Scott Aaronson and Lijie Chen. Complexity-theoretic foundations of quantum supremacy experiments. In *Proceedings of the 32Nd Computational Complexity Conference*, CCC '17, pages 22:1–22:67, Germany, 2017. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. URL: `https://doi.org/10.4230/LIPIcs.CCC.2017.22`, `doi:10.4230/LIPIcs.CCC.2017.22`.

[81] List of QC simulators, 2017. `http://www.quantiki.org/wiki/List_of_QC_simulators` (Last accessed 03/31/2017).

[82] Doan Binh Trieu. *Large-scale simulations of error prone quantum computation devices*, volume 2. Forschungszentrum Jülich, 2009.

[83] Koen De Raedt, Kristel Michielsen, Hans De Raedt, Binh Trieu, Guido Arnold, Marcus Richter, Th. Lippert, H. Watanabe, and N. Ito. Massively parallel quantum computer simulator. *Computer Physics Communications*, 176(2):121–136, 2007. `doi:10.1016/j.cpc.2006.08.007`.

[84] Mikhail Smelyanskiy, Nicolas P. D. Sawaya, and Alán Aspuru-Guzik. qHiPSTER: The Quantum High Performance Software Testing Environment. *ArXiv e-prints*, January 2016. `arXiv:1601.07195`.

[85] Masoud Mohseni, Peter Read, Hartmut Neven, Sergio Boixo, Vasil Denchev, Ryan Babbush, Austin Fowler, Vadim Smelyanskiy, and John Martinis. Commercialize quantum technologies in five years. *Nature*, 543(7644):171–174, 2017. `doi:10.1038/543171a`.

[86] John Kim, Wiliam J. Dally, Steve Scott, and Dennis Abts. Technology-driven, highly-scalable dragonfly topology. *SIGARCH Comput. Archit. News*, 36(3):77–88, June 2008. URL: `http://doi.acm.org/10.1145/1394608.1382129`, `doi:10.1145/1394608.1382129`.

[87] Arun F. Rodrigues, Karl S. Hemmert, Brian W. Barrett, Chad Kersey, Ron Oldfield, Marlo Weston, R. Risen, Jeanine Cook, Paul Rosenfeld, Elliot Cooper-Balis, and Bruce Jacob. The structural simulation toolkit. *SIGMETRICS Perform. Eval. Rev.*, 38(4):37–42, March 2011. URL: `http://doi.acm.org/10.1145/1964218.1964225`, `doi:10.1145/1964218.1964225`.

[88] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, pages 190–200, New York, NY, USA, 2005. ACM. URL: `http://doi.acm.org/10.1145/1065010.1065034`, `doi:10.1145/1065010.1065034`.

[89] Alex Parent, Martin Roetteler, and Krysta M. Svore. Reversible circuit compilation with space constraints. *ArXiv e-prints*, October 2015. `arXiv:1510.00377`.

[90] Stephane Beauregard. Circuit for Shor's algorithm using 2n+3 qubits. *Quantum Info. Comput.*, 3(2):175–185, March 2003. URL: `http://dl.acm.org/citation.cfm?id=2011517.2011525`.

[91] Volker Strassen. Gaussian elimination is not optimal. *Numerische mathematik*, 13(4):354–356, 1969.

[92] Gene Golub, Stephen Nash, and Charles Van Loan. A Hessenberg-Schur method for the problem AX + XB = C. *IEEE Transactions on Automatic Control*, 24(6):909–913, December 1979. `doi:10.1109/TAC.1979.1102170`.

[93] Texas Advanced Computing Center (TACC). Retrieved: 2014-12-01. URL: `https://www.tacc.utexas.edu/stampede/`.

[94] ARB OpenMP. OpenMP 4.0 specification, 2013.

[95] Dave Wecker and Krysta M. Svore. LIQUi$|\rangle$: A Software Design Architecture and Domain-Specific Language for Quantum Computing. *ArXiv e-prints*, February 2014. `arXiv:1402.4467`.

[96] Rolando D. Somma, Sergio Boixo, Howard Barnum, and Emanuel Knill. Quantum simulations of classical annealing processes. *Phys. Rev. Lett.*, 101:130504, Sep 2008. URL: `https://link.aps.org/doi/10.1103/PhysRevLett.101.130504`, `doi:10.1103/PhysRevLett.101.130504`.

[97] Nathan Wiebe, Ashish Kapoor, and Krysta M. Svore. Quantum Deep Learning. *ArXiv e-prints*, December 2014. `arXiv:1412.3489`.

[98] Nathan Wiebe, Ashish Kapoor, and Krysta M. Svore. Quantum algorithms for nearest-neighbor methods for supervised and unsupervised learning. *Quantum Info. Comput.*, 15(3-4):316–356, March 2015. URL: `http://dl.acm.org/citation.cfm?id=2871393.2871400`.

[99] Patrick Rebentrost, Masoud Mohseni, and Seth Lloyd. Quantum support vector machine for big data classification. *Phys. Rev. Lett.*, 113:130503, Sep 2014. URL: `https://link.aps.org/doi/10.1103/PhysRevLett.113.130503`, `doi:10.1103/PhysRevLett.113.130503`.

[100] Craig Gidney. Factoring with n+2 clean qubits and n-1 dirty qubits. *ArXiv e-prints*, June 2017. `arXiv:1706.07884`.

[101] Aram W. Harrow, Avinatan Hassidim, and Seth Lloyd. Quantum algorithm for linear systems of equations. *Phys. Rev. Lett.*, 103:150502, Oct 2009. URL: `https://link.aps.org/doi/10.1103/PhysRevLett.103.150502`, `doi:10.1103/PhysRevLett.103.150502`.

[102] Yasuhiro Takahashi and Noboru Kunihiro. A quantum circuit for Shor's factoring algorithm using 2n + 2 qubits. *Quantum Info. Comput.*, 6(2):184–192, March 2006. URL: `http://dl.acm.org/citation.cfm?id=2011665.2011669`.

[103] Robert B. Griffiths and Chi-Sheng Niu. Semiclassical Fourier transform for quantum computation. *Phys. Rev. Lett.*, 76:3228–3231, Apr 1996. URL: `https://link.aps.org/doi/10.1103/PhysRevLett.76.3228`, `doi:10.1103/PhysRevLett.76.3228`.

[104] Alex Bocharov, Martin Roetteler, and Krysta M. Svore. Efficient synthesis of probabilistic quantum circuits with fallback. *Phys. Rev. A*, 91:052317, May 2015. URL: `https://link.aps.org/doi/10.1103/PhysRevA.91.052317`, `doi:10.1103/PhysRevA.91.052317`.

[105] Richard Cleve and John Watrous. Fast parallel circuits for the quantum Fourier transform. In *Proceedings of the 41st Annual Symposium on Foundations of Computer Science*, FOCS '00, pages 526–, Washington, DC, USA, 2000. IEEE Computer Society. URL: `http://dl.acm.org/citation.cfm?id=795666.796591`.

[106] Craig Gidney. StackExchange: Creating bigger controlled nots from single qubit, Toffoli, and CNOT gates, without workspace, 2015. URL: `http://cs.stackexchange.com/questions/40933/creating-bigger-controlled-nots-from-single-qubit-toffoli-and-cnot-gates-with`.

[107] Cody Jones. Low-overhead constructions for the fault-tolerant toffoli gate. *Phys. Rev. A*, 87:022328, Feb 2013. URL: `https://link.aps.org/doi/10.1103/PhysRevA.87.022328`, `doi:10.1103/PhysRevA.87.022328`.

[108] Rodney Van Meter and Kohei M. Itoh. Fast quantum modular exponentiation. *Phys. Rev. A*, 71:052320, May 2005. URL: `https://link.aps.org/doi/10.1103/PhysRevA.71.052320`, `doi:10.1103/PhysRevA.71.052320`.

[109] Samuel A. Kutin. Shor's algorithm on a nearest-neighbor machine. *eprint arXiv:quant-ph/0609001*, August 2006. `arXiv:quant-ph/0609001`.

[110] Quake III source code, 2018. URL: `https://github.com/id-Software/Quake-III-Arena/blob/master/code/game/q_math.c#L552`.

[111] David Poulin and Pawel Wocjan. Sampling from the thermal quantum gibbs state and evaluating partition functions with a quantum computer. *Phys. Rev. Lett.*, 103:220502, Nov 2009. URL: `https://link.aps.org/doi/10.1103/PhysRevLett.103.220502`, `doi:10.1103/PhysRevLett.103.220502`.

[112] Chen-Fu Chiang, Daniel Nagaj, and Pawel Wocjan. Efficient circuits for quantum walks. *Quantum Info. Comput.*, 10(5):420–434, May 2010. URL: http://dl.acm.org/citation.cfm?id=2011362.2011366.

[113] Yudong Cao, Anargyros Papageorgiou, Iasonas Petras, Joseph Traub, and Sabre Kais. Quantum algorithm and circuit design solving the Poisson equation. *New Journal of Physics*, 15(1):013021, 2013. URL: http://stacks.iop.org/1367-2630/15/i=1/a=013021.

[114] Mihir K. Bhaskar, Stuart Hadfield, Anargyros Papageorgiou, and Iasonas Petras. Quantum algorithms and circuits for scientific computing. *Quantum Info. Comput.*, 16(3-4):197–236, March 2016. URL: http://dl.acm.org/citation.cfm?id=3179448.3179450.

[115] Edgard Muñoz-Coreas and Himanshu Thapliyal. T-count and qubit optimized quantum circuit design of the non-restoring square root algorithm. *CoRR*, abs/1712.08254, 2017. URL: http://arxiv.org/abs/1712.08254, arXiv:1712.08254.

[116] Stephen L. Moshier. Cephes math library, 2000. URL: http://www.moshier.net.

[117] Chris Lomont. Fast inverse square root. *Tech-315 nical Report*, 32, 2003.

[118] Donald E. Knuth. Evaluation of polynomials by computer. *Commun. ACM*, 5(12):595–599, December 1962. URL: http://doi.acm.org/10.1145/355580.369074, doi:10.1145/355580.369074.

[119] Charles H. Bennett. Time/space trade-offs for reversible computation. *SIAM J. Comput.*, 18(4):766–776, August 1989. URL: http://dx.doi.org/10.1137/0218053, doi:10.1137/0218053.

[120] Emanuel Knill. An analysis of Bennett's pebble game. *CoRR*, abs/math/9508218, 1995. URL: http://arxiv.org/abs/math/9508218, arXiv:math/9508218.

[121] Siu Man Chan. *Pebble games and complexity.* PhD thesis, Electrical Engineering and Computer Science, UC Berkeley, 2013. URL: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2013/EECS-2013-145.html.

[122] Eugene Y. Remez. Sur la détermination des polynômes d'approximation de degré donnée. *Comm. Soc. Math. Kharkov*, 10:41–63, 1934.

[123] John F. Wakerly. *Digital design*, volume 3. Prentice Hall, 2000.

Acknowledgments

First, I would like to thank my advisor, Matthias Troyer, for introducing me to the fascinating topic of quantum computing and for giving me the opportunity to work in this exciting field of research. I am extremely grateful to him for the freedom I enjoyed during my PhD work and for his invaluable advice and guidance.

Second, I am indebted to my first co-examiner Torsten Hoefler for his highly-appreciated advice and feedback, and for all the interesting and enlightening discussions we have had.

Third, I would like to express my gratitude to my second co-examiner Martin Roetteler and to Krysta Svore for collaborating with me on various projects during my PhD, for the freedom I enjoyed during my time working for Microsoft Research, and for their valuable advice.

I would like to thank my friends and colleagues at ETH Zurich and all over the world. A special thanks goes to Damian Steiger for the many late nights we worked to successfully meet deadlines, be it for paper submissions or software releases. It has been a great pleasure, even in stressful times. I would also like to thank Mathias Soeken for many fruitful discussions and for collaborating with me on several projects. Furthermore, I am grateful to Dominik Gresch, Mario Könz, and Donjan Rodic for many valuable and/or interesting discussions about various aspects of programming, physics, and beyond.

Last and most importantly, I would like to thank Jessica; for her unconditional love and support despite my many absences, and for our time together that has provided me with new energy and motivation time and again.

Curriculum Vitae

Personal data

Name:	Thomas Häner
Date of birth:	23.09.1990
Citizen of:	Zullwil (SO)
Nationality:	Swiss

Education

2016 – 2018	Graduate student in the group of Prof. M. Troyer ETH Zurich, Switzerland
2014 – 2016	MSc in CSE ETH Zurich, Switzerland
2011 – 2014	BSc in CSE ETH Zurich, Switzerland
2006 – 2010	High-School Diploma Gymnasium Laufental-Thierstein, Switzerland

Employment

2016 – 2018	Research and teaching assistant Institut for Theoretical Physics, ETH Zurich, Switzerland
2014 – 2015	Research assistant ETH Zurich, Switzerland Implementation of a PDE-Framework for the lecture “Numerical Methods for Partial Differential Equations” (D-MATH)
2013 – 2015	Teaching assistant ETH Zurich, Switzerland “Analysis I” and “Discrete Mathematics” (D-INFK)