Programming Quantum Computers Using Design Automation

(Executive Session Paper)

Mathias Soeken[1] Thomas Haener[2] Martin Roetteler[3]
[1]Integrated Systems Laboratory, EPFL, Lausanne, Switzerland
[2]Institute for Theoretical Physics, ETH Zurich, Switzerland
[3]Station Q, QuArC, Microsoft Research, Redmond, WA, USA

***Abstract*—Recent developments in quantum hardware indicate that systems featuring more than 50 physical qubits are within reach. At this scale, classical simulation will no longer be feasible and there is a possibility that such quantum devices may outperform even classical supercomputers at certain tasks. With the rapid growth of qubit numbers and coherence times comes the increasingly difficult challenge of quantum program compilation. This entails the translation of a high-level description of a quantum algorithm to hardware-specific low-level operations which can be carried out by the quantum device. Some parts of the calculation may still be performed manually due to the lack of efficient methods. This, in turn, may lead to a design gap, which will prevent the programming of a quantum computer. In this paper, we discuss the challenges in fully-automatic quantum compilation. We motivate directions for future research to tackle these challenges. Yet, with the algorithms and approaches that exist today, we demonstrate how to automatically perform the quantum programming flow from algorithm to a physical quantum computer for a simple algorithmic benchmark, namely the hidden shift problem. We present and use two tool flows which invoke RevKit. One which is based on ProjectQ and which targets the IBM Quantum Experience or a local simulator, and one which is based on Microsoft's quantum programming language Q#.**

I. Introduction

With the rapid development of quantum hardware, quantum computers will soon reach sizes—measured in the numbers of qubits on which they operate—which allow them to solve problems that are out of reach for any of the best classical supercomputers. Quantum computers get this computational advantage over classical computers from the principles of *superposition* of states and *interference* of computational paths. Arguably the most simple case in which superposition manifests itself is a single qubit which can be in any (normalized) linear combination of two basis states. In contrast, a bit in conventional computers is always in a single state. By linearly increasing the number of qubits, superposition allows quantum computers to exponentially increase their computational space, while still being able to execute operations on this exponentially large space at low cost (in concrete terms this means that, everything else being equal, e.g., a 17-qubit quantum computer is twice as powerful as a 16-qubit quantum computer). While classical probabilistic computation also allows to access an exponentially large space with only linear resources, a quantum computer can leverage the principle of interference which allows to amplify or reduce the probability of computational paths. When designed properly, a *quantum algorithm* can combine the power to explore exponentially many computational paths at low cost with the ability to cancel out useless paths in such a way that a *measurement* of the remaining paths reveals the answer to an interesting computational problem.

Several quantum algorithms that are computationally superior to their classical counterparts have already been found. The most prominent one arguably is Shor's algorithm [1] that allows to factorize integers in polynomial time, whereas for classical computing nothing better than a sub-exponential upper bound is known [2]. Consequently, Shor's algorithm can break public-key cryptography which is based on the assumption that integer factorization is a hard task. Recently, precise cost estimates to implement Shor's algorithm for factoring [3] and elliptic curve dlog [4] were obtained, based on implementing and testing large-scale Toffoli networks.

In addition to Shor's algorithm, there are other quantum algorithms which play a role in scientific applications of interest. Examples include:

- Grover's search algorithm [5], which enables quadratically faster search in unstructured databases if the correct element can be recognized efficiently by a predicate (e.g., NP-complete problems). This has implications for the choice of security parameters in a post-quantum cryptographic world. Perhaps surprisingly, it turns out that the overhead due to implementing the defining predicate in a reversible way can be quite substantial [6].
- The HHL algorithm [7], which offers an exponential speedup for solving linear equations. Finding practical use cases of the HHL algorithm remains a challenge and the few real-world applications that have been identified so far [8], [9] require large overheads due to implementation of the classical subroutines that define the problem.
- Quantum simulation [10] (see, e.g., [11], [12] for overviews and pointers to the literature) to model atomic-scale interactions efficiently [13], [14] allowing to approximate behavior in drugs, organics, and materials science [15], [16], and has applications for simulating quantum field theories [17].

In order to execute a quantum algorithm on a physical quantum computer, the algorithm must be expressed in terms of elementary *quantum operations* that can be understood by a quantum computer—very much like classical computer programs need to be expressed in terms of low-level machine instructions to run on a classical computer. *Quantum compilers* are software programs that take a high-level description of a quantum algorithm and map them into so-called *quantum circuits*. Quantum circuits are *not* a physical entity, but an abstraction of the physical operations that can be performed to qubits of the physical system [18]. They are represented in terms of sequences of low-level quantum operations. Quantum circuits can be considered the "assembly code" of a quantum computer, in which qubits play the role of registers. The goal of quantum compilers is to find a quantum circuit that meets the number of available qubits and minimizes the number of quantum operations. A challenge for quatum compilers is to map combinational non-quantum operations into quantum circuits, while not exceeding the resource constraints due to the limited number of qubits. This is a difficult problem, and no satisfiable and sufficient solution is provided by today's state-of-the-art quantum compilers.

Quantum computing has made a big leap this year, as research on physical devices is moving from the academic environment into several companies [19]. Microsoft, Google, IBM, Intel, Alibaba, as well as the rapidly growing startup companies IonQ and Rigetti, are investing into building the first scalable quantum computer. As of today, the largest publicly available fully-programmable quantum computers[1] are by IBM which features 17 qubits [20] and by Rigetti which features 19 qubits [21]. Recently, Intel announced a quantum computer with 17 qubits [22] and IBM quantum computers with up to 50 qubits [23]. These sizes are not yet practical, since it has been shown that supercomputers are able to simulate low-depth quantum circuits with up to 56 qubits classically [24], and full state vector simulation is possible for up to 45 qubits [25]. The rapid progress in quantum computing and quantum simulation underlines the importance of having reliable and robust quantum programming toolchains.

II. Quantum Programming Languages

Several quantum programming languages were proposed in recent years, ranging from imperative to functional and low-level to high-level [26]. Languages such as Quipper [27], ScaffCC/Scaffold [28], [29], LIQ$Ui|\rangle$ [30], QWire [31], Quil [32], Q# [33] and ProjectQ [34] enable programming of quantum computers. Quipper is a strongly-typed, functional quantum programming language embedded in Haskell; Scaffold is a stand-alone C-like programming language and its compiler ScaffCC leverages the LLVM framework; QWire is embedded in the proof system Coq; LIQ$Ui|\rangle$ is embedded in F#; Q# is a stand-alone F#-like language, and ProjectQ and Quil are embedded in Python.

All mentioned languages offer extensible frameworks for quantum circuit description and manipulation, and some of them offer gate decomposition and circuit optimization methods, some classical control flow, and exporting of quantum circuits for rendering or resource costing purposes.

Theoretically, it would be sufficient if a programming language for quantum computing supported the gate set of the target hardware. The similarity between such an approach and classical assembly language brought into existence quantum assembly languages such as QASM [35] and OPENQASM [36]. While sufficient for today's quantum hardware which is able to perform a few gate operations on less than 20 qubits, programming in such a language is neither scalable nor particularly user-friendly. Rather, a quantum programming language should provide high-level abstractions in order to shorten development times and to enable portability across a wide range of quantum hardware backends, similar to today's compilers for classical high-level languages such as C++.

[1]in contrast to special-purpose quantum computers such as the D-Wave quantum annealer

In addition to purely classical and purely quantum subroutines, typical quantum algorithms also require classical functions to be evaluated on a superposition of inputs, e.g., the modular exponentiation in Shor's algorithm for factoring [1]. Therefore, such "mixed" constructs must also be supported by the language and the compiler must be able to translate these constructs to instructions which can be executed by the quantum hardware.

III. Quantum computing basics

This section introduces the necessary background on quantum algorithms and quantum circuits. This introduction is kept brief on purpose and focuses on the most important notations and definitions that are necessary in the course of this paper. For a more detailed overview on the matter, we refer the reader to the standard literature [37].

A *quantum algorithm* is implemented in terms of a quantum program, which is a sequence of high-level quantum operations that are performed on a set of qubits. A *qubit state* is modeled as a column vector $|\varphi\rangle = \binom{\alpha_0}{\alpha_1}$ with two complex-valued elements α_0 and α_1, called *amplitudes*, such that $|\alpha_0|^2 + |\alpha_1|^2 = 1$. The values $|\alpha_0|^2$ and $|\alpha_1|^2$ are the probabilities of whether the qubit state will be 0 or 1 after measuring it, respectively. The classical states for a logic 0 and logic 1 are $|0\rangle = \binom{1}{0}$ and $|1\rangle = \binom{0}{1}$, respectively. Hence, we may also write the state of a qubit as $|\varphi\rangle = \alpha_0|0\rangle + \alpha_1|1\rangle$. The notation $|\cdot\rangle$ is called Dirac or *bra-ket* notation and typical for denoting quantum states. A state in which the measurement outcome has an equal probability of being 0 or 1 is for example $\frac{1}{\sqrt{2}}\binom{1}{1}$, which is abbreviated as $|+\rangle$, since it occurs very frequently in the design of quantum algorithms. A different state with the same measurement probabilities is $|-\rangle = \frac{1}{\sqrt{2}}\binom{1}{-1}$. Although the measurement probability is the same, the quantum state is not, which is one reason that makes quantum computing significantly different from probabilistic computing.

Qubit registers refer to quantum states involving multiple qubits. As an example, a 2-qubit register is represented by the state $\begin{pmatrix}\alpha_{00}\\\alpha_{01}\\\alpha_{10}\\\alpha_{11}\end{pmatrix} = \alpha_{00}|00\rangle + \alpha_{01}|01\rangle + \alpha_{10}|10\rangle + \alpha_{11}|11\rangle$, which has four amplitudes, one for each classical state $|00\rangle$, $|01\rangle$, $|10\rangle$, and $|11\rangle$. In general, an n-qubit register is a column vector $|\varphi\rangle = \sum_{b\in\mathbb{B}^n}\alpha_b|b\rangle$ with 2^n amplitudes. This reflects the exponential power of qubits.

Quantum operations are modeled in terms of unitary matrices, called *quantum gates*. A matrix U is unitary if $UU^\dagger = U^\dagger U = I$, where $U^\dagger$ refers to the conjugate transpose of U (also referred to as the Hermitian or adjoint of U), and I

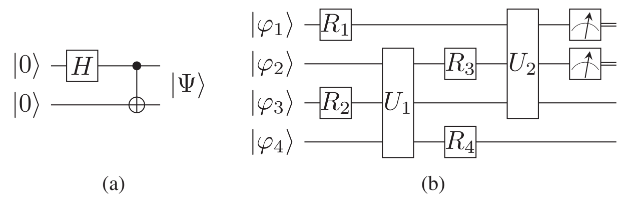

Fig. 1: Some basic examples for quantum circuits. Circuits are read from left to right. Shown in (a) is a simple quantum circuit that entangles two qubits. The circuit consists of a Hadamard gate H and a controlled NOT gate and which creates upon input $|0\rangle|0\rangle$ the resulting output state $|\Psi\rangle = \frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)$. Shown in (b) is an example for a larger quantum circuit consisting of local rotations $R_1, \ldots, R_4$ acting on single qubits, larger unitary gates U_1, U_2 acting on several qubits, and two measurement operations applied to the top two qubits.

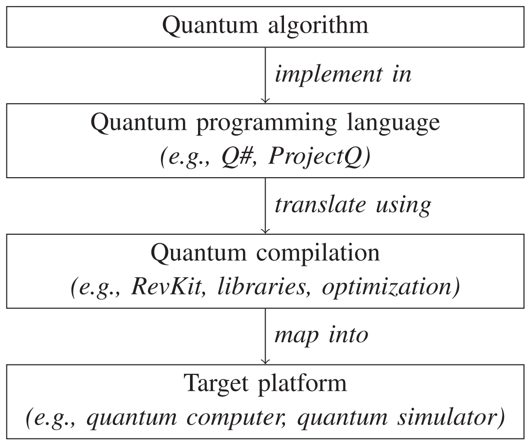
Quantum algorithm
implement in
Quantum programming language
(e.g., Q#, ProjectQ)
translate using
Quantum compilation
(e.g., RevKit, libraries, optimization)
map into
Target platform
(e.g., quantum computer, quantum simulator)

Fig. 2: The high-level design flow for mapping quantum algorithms to quantum computers.

is the identity matrix. A unitary matrix is length-preserving and therefore maps one qubit state into another qubit state. A quantum operation that acts on a single qubit is a 2×2 unitary matrix, and a quantum operation that acts on an n-qubit register is a $2^n \times 2^n$ unitary matrix. An example for a single qubit operation is the so-called *Hadamard* operation $H = \frac{1}{\sqrt{2}}\left(\begin{smallmatrix} 1 & 1 \\ 1 & -1 \end{smallmatrix}\right)$. This operation can be used to create a superposition of the two basis states $|0\rangle$ and $|1\rangle$, since $H|0\rangle = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$. The *CNOT* operation is a 2-qubit quantum operation that maps $|\varphi_1\rangle|\varphi_2\rangle \mapsto |\varphi_1\rangle|\varphi_1 \oplus \varphi_2\rangle$, where '$\oplus$' is the exclusive-OR operation. The CNOT operation inverts the *target qubit* $|\varphi_2\rangle$ if the *control qubit* $|\varphi_1\rangle$ is 1. It can be represented as the unitary matrix $\left(\begin{smallmatrix} 1&0&0&0 \\ 0&1&0&0 \\ 0&0&0&1 \\ 0&0&1&0 \end{smallmatrix}\right)$. The unitary matrix of the CNOT operation is a permutation matrix. A quantum operation whose unitary matrix is a permutation matrix is called a *classical* operation. This also means that all classical operations must be reversible, since otherwise they cannot be represented in terms of a permutation matrix.

A quantum algorithm describes the interaction of quantum operations with qubits. Researchers use *quantum circuits* as an abstraction to illustrate these interactions. Fig. 1(a) shows an abstract representation of such a quantum circuit. The horizontal lines represent qubits, the boxes represent quantum operations that interact with the qubits, and time moves from left to right. Consequently, the vertical direction corresponds to space (i.e., number of qubits) and the horizontal direction to time (i.e., number of quantum operations). There are three types of operations: (i) quantum operations ($R_1, \ldots, R_4$ in the figure), (ii) classical operations (U_1 and U_2 in the figure), and (iii) measurements which are illustrated by a meter. Classical operations perform classical computations, such as arithmetic operations—but acting on qubits rather than bits. A quantum circuit can be seen as a way to represent a large unitary matrix composed of smaller ones. The absence of a gate in a circuit corresponds to the identity matrix. Fig. 1(b) shows a simple quantum algorithm consisting of a Hadamard gate followed by a CNOT operation. The CNOT operation has a special notation with a solid circle for the control qubit and an '$\oplus$' symbol for the target qubit. This quantum algorithm takes as input two qubits initialized in state $|0\rangle$ and creates the 2-qubit state $\frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)$. This state is entangled, i.e., by measuring one qubit the outcome of the second is immediately determined. This also means that the explicit state of one of the qubits cannot be described individually. Sequential composition of two gates in a quantum circuit corresponds to matrix multiplication and parallel composition of gates corresponds to taking the Kronecker product, denoted '$\otimes$'. The unitary matrix represented by the quantum circuit in Fig. 1(b) is $\mathrm{CNOT}(H \otimes I)$.

Note that today's quantum algorithms rely on a variety of different combinational calculations. Factoring needs constant modular arithmetic [1], computing elliptic curve discrete logarithms using a quantum algorithm requires generic modular arithmetic [4], the HHL algorithm needs reciprocals and Newton type methods [7], amplitude amplification algorithms need implementations for search and collision [5], and quantum simulation algorithms need addressing and indexing functions for sparse matrices as well as computing Hamiltonian terms on the fly [11].

IV. Quantum design automation: general flow

Fig. 2 abstractly illustrates the overall programming flow for quantum computers. The capabilities of the targeted quantum computer are taken into account when developing the quantum algorithm. A quantum algorithm consists of quantum parts and classical combinational operations. The quantum algorithm must be translated into a quantum circuit. While automatic and satisfactory solutions exist for translating the quantum parts, no sufficient solution exists for automatically translating the combinational operations. In fact, the current quantum programming flow depends on predefined library components for which manually derived quantum circuits exist. Such a manual flow is time-consuming, not flexible, and not scalable.

Rather, one would like to express the quantum program at a high level of abstraction and have a compiler which is able to automatically translate the entire circuit, even if no manually optimized libraries are available. It is thus crucial that the quantum programming language used to express the quantum program supports such a design flow.

V. Compiling Boolean functions

The translation of classical combinational operations into quantum circuits involves *reversible logic synthesis* [39]. Due to the physical properties of quantum states, all operations need to be performed in a reversible manner. State-of-the-art approaches first create a reversible logic circuit with reversible gates, which are Boolean abstractions of classical reversible operations. Other methods translate reversible gates into quantum circuits [40], [41], [42]. Many approaches for reversible logic synthesis have been proposed in the last 15 years (e.g., [43], [44], [45], [46]).

It is customary to distinguish reversible synthesis algorithms based on whether the Boolean function that is input to the algorithm is already a reversible function or not. For a reversible Boolean function $f : \mathbb{B}^n \to \mathbb{B}^n$, reversible synthesis algorithms find an n qubit quantum circuit that realizes the unitary

$$U : |x\rangle \mapsto |f(x)\rangle. \tag{1}$$

Several algorithms have been proposed for this task. They differ depending on f's representation. Most of the early algorithms expect f to be represented as a truth table (see, e.g., [43], [47], [48], [49], [50]). The explicit truth table representation limits the application to large functions, i.e., $n > 20$. Alternative implementations have been proposed that work on symbolic representations of f, e.g., as binary decision diagrams (BDDs) [46], [51] or Boolean satisfiability problems [52]. These approaches are able to find quatum circuits also for some Boolean functions that are much larger. However, the symbolic function representation does not always guarantee a compact function representation. Nonetheless, the main drawback of such reversible functions algorithms is that they require a reversible input function, which is rarely the case in most algorithms of interest.

The second class of reversible synthesis algorithms considers irreversible functions $f : \mathbb{B}^n \to \mathbb{B}^m$ as input. Since a quantum circuit can not represent irreversible functions, f must be embedded into a reversible function. This may be done either *explicitly* or *implicitly*. In the explicit case one finds a reversible function $g : \mathbb{B}^r \to \mathbb{B}^r$ such that

$$g(x_1, \dots, x_n, 0, \dots, 0) = (y_1, \dots, y_m, y_{m+1}, \dots, y_r), \tag{2}$$

if $f(x_1, \dots, x_n) = (y_1, \dots, y_m)$. Finding g such that r is minimum is coNP-hard [53] and does therefore not scale to larger functions, although symbolic methods can help to slightly increase the range of applicable functions [53], [54]. An embedding as in (2) is referred to as *in-place embedding*, since the input values are not restored after the application of g. As an example, explicit embedding with symbolic reversible logic synthesis was applied to find in-place reversible circuits for the reciprocal function $1/x$ up to $n = m = 16$ digits in x and $r = 31$ (see [55]).

One can easily show that there exists a reversible function g with $r = m + n$, by chosing

$$g(x, y) = (x, y \oplus f(x)) \tag{3}$$

where $x = x_1, \dots, x_n$, $y = y_1, \dots, y_m$, and '$\oplus$' refers to the bitwise application of the XOR operation in this case. Such an embedding is also referred to as *Bennett embedding*. So-called ESOP (exclusive sum-of-products) based reversible synthesis approaches [56], [57], [58] find reversible circuits that realize (3). In order to apply ESOP-based synthesis one must find 2-level ESOP expressions for each of the m outputs in f (see, e.g., [59], [60]). This approach can be time consuming and limits the application to large functions. In [55] ESOP-based synthesis was successfully applied up to $n = 25$ for the reciprocal function.

Scalable reversible synthesis algorithms require additional helper qubits, called ancillae. Given an irreversible Boolean function $f : \mathbb{B}^n \to \mathbb{B}^m$, they find an $(n + m + k)$ qubit quantum circuit that realizes the unitary

$$U : |x\rangle|y\rangle|0^k\rangle \mapsto |x\rangle|y \oplus f(x)\rangle|0^k\rangle. \tag{4}$$

If $k = 0$, the synthesis problem corresponds to ESOP-based synthesis, but for $k > 0$, the synthesis algorithm can use the k additional qubits to store intermediate computations. The most effective methods use multi-level logic network representations such as BDDs [45], [61], [62], And-inverter graphs [63], [64], XOR-majority graphs [55], or LUT networks [65]. These methods are referred to as *hierarchical* reversible logic synthesis. Intermediate results represented by internal nodes in the corresponding logic networks are mapped on the additional qubits. If the network has many internal nodes, many ancillae are required, however, pebbling strategies [66] may be employed to trade off the number of qubits for quantum operations [67]. One of the biggest problems in hierarchical reversible logic synthesis is the fact that k is a result of the synthesis algorithm, i.e., it is determined by the algorithm's requirements for temporary storage. One of the largest challenges in reversible logic synthesis is to find reversible synthesis algorithms that take k as an input parameter and guarantee to return a quantum circuit that satisfies the space requirements.

VI. Illustrative Example

In this and the following sections, we use the example of the hidden shift problem for Boolean functions to illustrate the complete flow of programming a quantum computer. For this purpose, we leverage the quantum programming languages ProjectQ [34] and Q# [33] interfaced with the quantum compilation framework RevKit [68]. ProjectQ and Q# allow for a high-level description of the algorithm using several meta-constructs, and enables interfacing a physical quantum computer. RevKit is used to automatically translate the combinational parts in the quantum algorithm for the hidden shift problem into quantum gates.

ProjectQ is an open source software framework for quantum computing with a modular compiler design which allows domain experts to easily extend its functionality. Furthermore, this modularity enables portability of quantum algorithm implementations. Specifically, once an algorithm has been implemented, it can be run using various types of backends, be it software (simulator, emulator, resource counter, etc.) or hardware (classical and/or quantum).

Q# [33] is a scalable, multi-paradigm, domain-specific programming language for quantum computing by Microsoft. The Q# framework allows describe how instructions are executed on quantum machines. The machines that can be targeted include many different levels of abstraction, ranging from various simulators to actual quantum hardware. Q# is multi-paradigm in that it supports functional and imperative programming styles. Q# is scalable in that it allows to write programs to target machines of various sizes, ranging from small machines with only a few hundred qubits to large machines with millions of qubits. Finally, being a bona-fide stand-alone language, Q# allows a programmer to code complex quantum algorithms, offers rich and informative error reporting, and allows to perform various tasks such as debugging, profiling, resource estimation, and certain special-purpose simulations.

RevKit is an open source C++ framework and library that implements a large set of reversible synthesis, optimization, and mapping algorithms. By default, RevKit is executed as a command-based shell application, which allows to perform synthesis scripts by combining a variety of different commands. As an example, the command sequence

```
revgen --hwb 4; tbs; revsimp; rptm; tpar; ps -c  (5)
```

generates a reversible function describing the 4-input reversible hidden-weighted bit function, synthesizes it into a reversible circuit using transformation-based synthesis [43], performs simplification of the resulting circuit, maps it into Clifford+T gates using the mapping described in [42], optimizes the T count using the T-par algorithm presented in [69], and finally prints statistics about the final quantum circuit. All RevKit commands provided by the shell can also be accessed via a Python interface, e.g., '`revkit.revgen(hwb = 4)`' for the first command in (5). Using the Python interface, RevKit can be executed from within ProjectQ using the `projectq.libs.revkit` module.

A. Quantum algorithm: the Boolean hidden shift problem

For the illustrative example, we review the hidden shift problem for Boolean functions [70]. In general, the hidden shift problem is a quite natural source of problems for which a quantum computer might have an advantage over a classical computer as it exploits the property that fast convolutions can be performed by computing Fourier transforms and pointwise multiplication. See [71] for general background on hidden shifts and related problems and [70] for the case of hidden shifts over Boolean functions. Recently, the hidden shift problem for bent functions was also studied in [72] from the point of view of classical simulation of the resulting quantum circuits. The problem of computing hidden shifts for Boolean functions is the following:

Definition 1 (Hidden shift problem): Let $n \geq 1$ and let $f, g : \mathbb{B}^n \to \mathbb{B}$ be two Boolean functions such that the following conditions hold: (i) f, and g are bent functions, and (ii) there exist $s \in \mathbb{B}^n$ such that $g(x) = f(x + s)$ for all $x \in \mathbb{B}^n$.

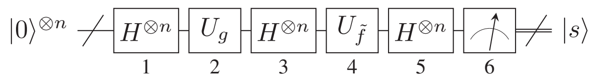

Fig. 3: Quantum algorithm for the hidden shift problem for a bent functions f. The quantum circuit assumes access to the shifted function $g(x) = f(x + s)$ which is implemented by the diagonal unitary operator $U_g = \sum_x (-1)^{g(x)} |x\rangle\langle x|$. Also, the algorithm needs access to the dual bent function $\widetilde{f}$, which again is computed into the phase via a diagonal unitary.

Moreover, let oracle access for g and the dual bent function $\widetilde{f}$ be given. The task is then to find the hidden shift s.

Bent functions are Boolean functions which have a perfectly flat Fourier (i.e., Hadamard) spectrum, which in a sense makes them resemble random noise. It is easy to see that bent functions can only exist if the number of variables n is even. What makes the hidden shift problem for bent functions attractive is that it can be shown that classical algorithms cannot find the shift efficiently, whereas quantum algorithms can find the shift with only 1 query to g and 1 query to $\widetilde{f}$. Moreover, the quantum algorithm to find hidden shifts is very simple as shown in Fig. 3: the gates needed are Hadamard gates, diagonal unitaries to implement the shifted function and the dual bent function, and measurements in the computational basis. An attractive feature of the algorithm is that—assuming perfect gates—the answer is deterministic, i.e., the measured bit pattern directly corresponds to the hidden shift. We assign each operation in the quantum algorithm an index from 1 to 6, written below each gate.

B. Maiorana-McFarland bent functions

Arguably, the most simple example for a bent function is the inner product $f(x, y) = xy^t = \sum_{i=1}^n x_i y_i$ of two bit-vectors $x = x_1, \dots, x_n$ and $y = y_1, \dots, y_n$. Note that this is a Boolean function $f : \mathbb{B}^{2n} \to \mathbb{B}$ on an even number $2n$ of variables. The function can be generalized to

$$f(x, y) = x\pi(y)^t + h(y) \tag{6}$$

for an arbitrary permutation $\pi \in S_{2^n}$ on all 2^n boolean bitvectors of length n and an arbitrary Boolean function $h : \mathbb{B}^n \to \mathbb{B}$. This leads to the class of so-called Maiorana-McFarland bent functions[2]. The dual bent function is $\widetilde{f}(x, y) = \pi^{-1}(x)y^t + h(\pi^{-1}(x))$ [70]. Asymptotically, the size of this class scales as $O(2^{cn2^n})$ which is doubly exponential in n, however, which is just an exponentially small fraction of the set of all Boolean functions on $2n$ variables. A simple counting argument shows that most permutation π do not have an efficient circuit, however, there exist natural families of Maiorana-McFarland bent function for which the permutation π as well as the Boolean function h can be implemented efficiently.

The same basic circuit as shown in Fig. 3 can be used to solve the hidden shift problem for Maiorana-McFarland

[2]Named after mathematicians James A. Maiorana (1946–2014) and Robert L. McFarland who were the first to study these functions about 50 years ago.

```
1  from projectq.cengines import MainEngine
2  from projectq.ops import All, H, X, Measure
3  from projectq.meta import Compute, Uncompute
4  from projectq.libs.revkit import PhaseOracle
5
6  # phase function
7  def f(a, b, c, d):
8      return (a and b) ^ (c and d)
9
10 eng = MainEngine()
11 x1, x2, x3, x4 = qubits =
        eng.allocate_qureg(4)
12
13 # circuit
14 with Compute(eng):
15     All(H) | qubits
16     X | x1
17 PhaseOracle(f) | qubits
18 Uncompute(eng)
19
20 PhaseOracle(f) | qubits
21 All(H) | qubits
22 Measure | qubits
23
24 eng.flush()
25
26 # measurement result
27 print("Shift is {}".format(8 * int(x4) + 4 *
        int(x3) + 2 * int(x2) + int(x1)))
```

Fig. 4: ProjectQ python code for an instance of the hidden shift problem where $f(x) = x_1x_2 \oplus x_3x_4$ and $g(x) = f(x+1)$.

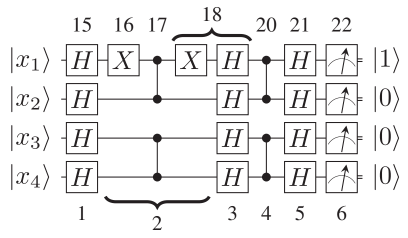

Fig. 5: Quantum circuit that is implemented by the Python code in Fig. 4; indexes below the gates correspond to the steps in Fig. 3, indexes above the gates correspond to the lines in Fig. 4.

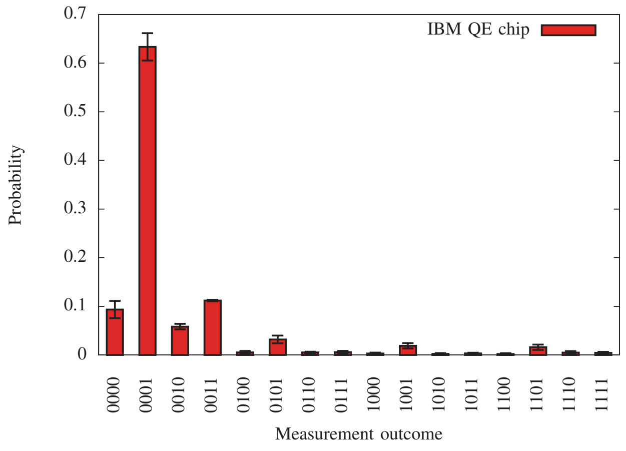

Fig. 6: Histogram depicting the average and standard deviation of the outcome probabilities of three runs of the code in Fig. 4. Each run consists of 1024 executions of the circuit on the IBM Quantum Experience chip. The correct shift $s = 1$ was found with average probability $p \approx 0.63$.

bent functions. Note however, that in contrast to the case of the inner product function for which $\tilde{f} = f$ holds, for the more general case where π is not the identity permutation, the diagonal unitary $U_f = \sum_x (-1)^{f(x)} |x\rangle\langle x|$ implementing the bent function f and the shift g is different from the diagonal unitary $U_{\tilde{f}} = \sum_x (-1)^{\tilde{f}(x)} |x\rangle\langle x|$ implementing the dual bent function. For the Maiorana-McFarland family specifically, the difference in implementing f and $\tilde{f}$ is to use the inverse permutation π^{-1} instead of π and to apply it to the x-variables instead of the y variables and similarly the role of x and y has to be changed in the evaluation of h.

VII. Interop with ProjectQ and Simulator / IBM Backend

In this section, we show how to program a concrete instance of the hidden shift problem using ProjectQ and RevKit. We choose $f(x) = x_1x_2 \oplus x_3x_4$ as a Boolean function on 4 variables, and $g(x) = f(x+1)$, i.e., $s = 1$. It can be shown that $f = \tilde{f}$.

Fig. 4 shows the ProjectQ Python code for this example. The corresponding quantum circuit that is generated by the code is shown in Fig. 5. Lines 10–11 initialize a ProjectQ engine with 4 qubits, named `x1`, `x2`, `x3`, and `x4`, and stored in a list `qubits`. Line 15 performs step 1 of the quantum algorithm described in Fig. 3. Line 16 describes the shift by $s = 1$, implemented using an X operation on the least-significant qubit `x1`. Together with the phase circuit computed for f in line 17, it computes step 2 in the quantum algorithm. As input to the `PhaseOracle` statement we can provide a predicate `f` implemented as Python function. The `PhaseOracle` statement converts the Python code in `f` into a Boolean expression. This expression is then passed to RevKit, which automatically compiles the expression into a circuit computing the function described by `f` into the global phase of the circuit. The `Uncompute` statement in line 18 uncomputes all operations that were specified in the `Compute` block in lines 14–16, by applying all operations in inverse order. This will also add step 3 of the algorithm to the quantum circuit. Since $\tilde{f} = f$, we again compute the phase circuit for `f` in line 20, apply Hadamard gates to each qubit for step 5 of the algorithm, and finally measure all qubits in line 22. The resulting state of the qubits, computed using simulation, corresponds to the shift $s = 1$. The program outputs '`Shift is 1`.' By changing two lines of code in 4, the backend can be changed to the IBM Quantum Experience chip. Doing so and running three times 1024 shots of the circuit yielded the results depicted in Fig. 6.

Fig. 7 shows a Python code that implements an instance of the hidden shift problem for a Maiorana-McFarland bent function where $n = 3$, $\pi = [0, 2, 3, 5, 7, 1, 4, 6]$, and $h = 0$. Fig. 8 shows the corresponding circuit. The program is similar

```python
from projectq.cengines import MainEngine
from projectq.ops import All, H, X, Measure
from projectq.meta import Compute, Uncompute,
     Dagger
from projectq.libs.revkit import PhaseOracle,
     PermutationOracle
import revkit

# phase function
def f(a, b, c, d, e, f):
  return (a and b) ^ (c and d) ^ (e and f)

# permutation
pi = [0, 2, 3, 5, 7, 1, 4, 6]

eng = MainEngine()
qubits = eng.allocate_qureg(6)
x = qubits[::2] # qubits on odd lines
y = qubits[1::2] # qubits on even lines

# circuit
with Compute(eng):
  All(H) | qubits
  All(X) | [x[0], x[1]]
  PermutationOracle(pi) | y
PhaseOracle(f) | qubits
Uncompute(eng)

with Compute(eng):
  with Dagger(eng):
    PermutationOracle(pi, synth=revkit.dbs) | x
PhaseOracle(f) | qubits
Uncompute(eng)

All(H) | qubits
Measure | qubits

eng.flush()

# measurement result
print("Shift is {}".format(sum(int(q) << i for
     i, q in enumerate(qubits))))
```

Fig. 7: ProjectQ python code for an instance of the hidden shift problem where $f(x,y) = x\pi(y)^t$, $\pi = [0,2,3,5,7,1,4,6]$, and $s = 5$.

to the program in Fig. 4. We create 6 qubits and partition them into three qubits `x` for x_1, x_2, x_3 and three qubits `y` for y_1, y_2, y_3. The inner product is realized by the bent function specified by the Python function `f`. We use the function `PermutationOracle` to create a quantum circuit from a permutation, which is then applied to the qubits in `x`. The function `PermutationOracle` calls RevKit using transformation-based synthesis [43] followed by a mapping of Toffoli gates into Clifford+T gates using the algorithm presented in [42]. For the second part of the circuit, we need a quantum circuit for the inverse permutation π^{-1}. Instead of inverting π, we compute another quantum circuit for π and invert the circuit using the `Dagger` statement. Note that for this compilation we chose decomposition-based synthesis [47] for finding a Toffoli network for the permutation. Since each permutation is uncomputed after the phase circuit for the inner product, the final circuit consists of four subcircuits realizing either π or its inverse. These are emphasized using dashed boxes in Fig. 8.

VIII. Interop with Q# and simulator backend

In the following we describe a programming flow that implements the same high-level algorithm, i.e., an instance of the hidden shift problem for Maiorana-McFarland functions, but implements it in Q#. While at a high level, the interop between RevKit and Q# happens as described in Fig. 2, the actual invocation of RevKit in the design flow is slightly different from the RevKit/ProjectQ interop in that RevKit is used as a pre-processor to produce the code for the permutation oracle as Q# native code. Subsequently, the Q# compiler is then invoked to compile the algorithm and to target a simulator backend that is part of the Microsoft Quantum Development Kit (QDK).

The code for the hidden shift problem shown in Fig. 9 is structurally quite similar to familiar languages such as C# and Java in its use of semicolons to end statements, curly brackets to group statements, and double-slash to introduce comments. Q# also uses namespaces to group definitions together, and allows references to elements from other namespaces.

The Q# code begins with a namespace statement (line 1) which declares the symbols and makes then available for other projects. The mechanism to include other namespaces is via the `open` keyword. This is used here in line 3 to include the basic gates such as the Hadamard gate H and in line 5 to include the "canon" which is a large library of useful operations, functions, and combinators. For the current example we use operations `ApplyToEach` and `MResetZ` from the canon. The implementation of the permutation oracle itself is provided in another namespace which is included in line 7. The basic unit in Q# to model side effects on quantum data is an operation such as the operation `HiddenShift` declared in line 9. Besides operations, Q# also supports functions which allow to modify state that is purely classical. Note that the definition of an operation or a function must begin with a declaration of the type signature of the function, including its input and output types. This is done in lines 10–14 of the present example.

Operations and functions are first-class citizens in Q#, i.e., they can be passed as arguments. In the present case, `Ustar` is an operation that implements the diagonal operator $U_{\tilde{f}}$ as defined earlier. If an operation changes the state of a quantum register (modeled here as `Qubit[]` array), then its type is `Qubit[] => ()`, where `()` denotes the unit type. Operations are the only way the state of an abstract quantum machine model can be manipulated. Q# can be used to target many abstract quantum machine models, including future physical implementations of scalable quantum computers. Currently, the main target of Q# is a state-of-the-art simulator that can easily handle up to 30 qubits on a standard computer and over 40 qubits on a distributed computer using an MPI-based implementation.

The `body` element on line 15 specifies the implementation of the operation. Q# operations may also specify implementations for variants, or derived operations, that are common in quantum computing. These variants indicated by `adjoint` (inverse), a `controlled` and `controlled adjoint`. If the key-word `auto` is provided, then the compiler auto-

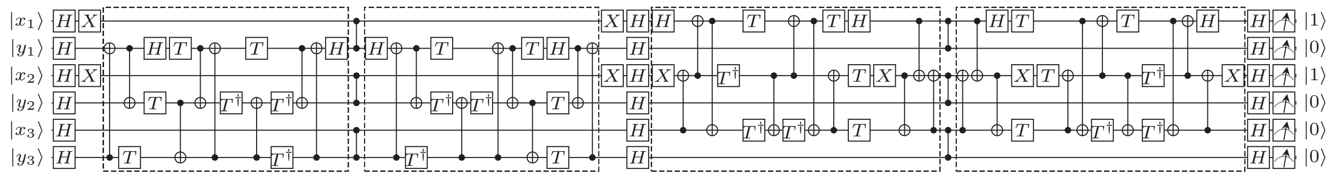

Fig. 8: Quantum circuit that is implemented by the Python code in Fig. 7. The dashed boxes emphasize the subcircuits which correspond to realizations of π and its inverse.

```
1  namespace Microsoft.Quantum.HiddenShift{
2  // basic operations: Hadamard, CNOT, etc
3  open Microsoft.Quantum.Primitive;
4  // useful lib functions and combinators
5  open Microsoft.Quantum.Canon;
6  // permutation defining the instance
7  open Microsoft.Quantum.PermOracle;
8
9  operation HiddenShift
10 // signature of input types
11 (Ufstar : (Qubit[] => ()),
12 Ug : (Qubit[] => ()), n : Int) :
13 // signature of output type
14 Result[] {
15    body {
16       mutable resultArray = new Result[n];
17       // allocate n clean qubits
18       using(qubits=Qubit[n]) {
19          ApplyToEach(H, qubits);
20          Ug(qubits);
21          ApplyToEach(H, qubits);
22          Ufstar(qubits);
23          ApplyToEach(H, qubits);
24          // measure and reset qubits
25          for (idx in 0..(n-1)) {
26             set resultArray[idx] =
                   MResetZ(qubits[idx]);
27          }
28       }
29       Message($"result: {resultArray}");
30       return resultArray;
31    }
32 }}
```

Fig. 9: Implementation of the correlation algorithm for the Boolean hidden shift problem in Q#. This code is shipped as an algorithm sample with the Microsoft QDK [33].

```
1  namespace Microsoft.Quantum.PermOracle{
2  open Microsoft.Quantum.Primitive;
3
4  operation PermutationOracle
5  // signature of input types
6  (qubits : Qubit[]) :
7  // signature of output type
8  () {
9     body {
10       CNOT(qubits[2], qubits[1]);
11       H(qubits[0]);
12       T(qubits[2]);
13       T(qubits[1]);
14       T(qubits[0]);
15       CNOT(qubits[1], qubits[2]);
16       CNOT(qubits[0], qubits[1]);
17       CNOT(qubits[2], qubits[0]);
18       (Adjoint T)(qubits[1]);
19       CNOT(qubits[2], qubits[1]);
20       (Adjoint T)(qubits[2]);
21       (Adjoint T)(qubits[1]);
22       T(qubits[0]);
23       CNOT(qubits[0], qubits[1]);
24       CNOT(qubits[2], qubits[0]);
25       CNOT(qubits[1], qubits[2]);
26       H(qubits[0]);
27       CNOT(qubits[0], qubits[1]);
28       CNOT(qubits[1], qubits[2]);
29    }
30    adjoint auto
31    controlled auto
32    controlled adjoint auto
33 }
34
35 operation BentFunctionImpl
36 (n : Int, qs : Qubit[]) : () {
37    body {
38       let xs = qs[0..(n-1)];
39      let ys = qs[u..(2*n-1)];
40      (Adjoint PermutationOracle)(ys);
41      for (idx in 0..(n-1)) {
42         (Controlled Z)([xs[idx]], ys[idx]);
43      }
44      PermutationOracle(ys);
45    }
46 }
47
48 function BentFunction
49 (n : Int) : (Qubit[] => ()) {
50   return BentFunctionImpl(n, _);
51 }}
```

Fig. 10: Q# code for an instance of the hidden shift problem where $f(x, y) = x\pi(y)^t$, $\pi = [0, 2, 3, 5, 7, 1, 4, 6]$.

matically calculates the inverse or controlled version of the operation based on the body, but in general it can make sense to provide these implementations separately as more efficient circuits might be known. While variants do not occur in the implementation of the `HiddenShift` operation, they do occur in the implementation of the operation `PermutationOracle` further below.

Q# allows the introduction of mutable variable as in line 16 which is returned to a driver program (which can be written in a .NET language such as C# or F#) in line 30. Further notable elements used in this code snippet are the allocation of clean qubits (which by definition are initialized in the $|0\rangle$ state) in line 18 by using the `using` keyword. Q# offers classical flow and control constructs like in line 25 where the code iterates through a range of integers using `for`. Finally, we remark that Q# supports mutable and immutable types. The syntax for declaring a new `mutable` variable is shown in line 16 of an array that will hold the final result of the computation. Assignment of mutable variables is done using `set` statements as in line 26.

The definition of the instance U_g and $U_{\tilde{f}}$ of the hidden shift problem itself is done by calling RevKit first during a pre-processing state. The input for this is a description of the permutation π to be implemented. The output of this stage is another Q# program which is shown as the `PermutationOracle` operation in Fig. 10. Note that this operation makes use of primitive gates that are built-into the Q# language and that are native to the underlying abstract quantum machine model, such as H, T, and CNOT. Also note that the `Adjoint` functor is used in lines 18, 20, and 21 which computes the inverse of the invoked operation.

The instance of the bent function is defined in the block starting at line 48 and returns a function with signature `Qubit[] => ()`. The implementation of this function, which depends on the number of variables (here denoted by integer `n` using the `Int` primitive type) invokes another operation from which the function is constructed using partial application, which is the basic mechanism in which e.g. currying can be implemented in Q#.

For space reasons, not all subroutines used in the implementation of the shifted bent functions and the test harness are shown as snippets, however, these can be inspected as sample Q# code that was shipped with the QDK [33]. The test subroutine consists of a C# part that invokes the above Q# program and targets the built-in simulator.

IX. Challenges and Conclusions

In this paper, we illustrated and discussed the high-level design flow for mapping a quantum algorithm to quantum computers using quantum programming languages. Expressive syntactical constructs and a rich API in combination with effective automatic compilation algorithms allow us to express quantum algorithms at a high level without being burdened with specifying each single quantum operation. This ultimately leads to implement (i) more scalable algorithms, since tedious manual compilation of combinational components is performed automatically, and (ii) more complex algorithms by combining abstract high-level syntactic constructs offered by the programming language. Therefore, programming quantum computers is catching up with its classical counterpart in which a variety of high-level programming languages and significant effort in the development of compilers render manual assembly descriptions unnecessary.

Several challenges remain and are awaiting satisfactory solutions. In this paper, we only considered simple reversible synthesis methods which do not require additional ancilla qubits for the realization of the quantum circuit. This limits their application to small functions with up to about 25 variables. In order to automatically compile larger functions, reversible logic synthesis methods require additional qubits. These are typically determined during the execution of the algorithm, and cannot be bounded ahead of time. Synthesis methods that find a solution without exceeding a given number of ancillae are rare and the state of available solutions is still in its infancy [65], [67].

Another issue is the verification of the synthesized circuits. Simulating the quantum circuit may require to represent the complete quantum state, which is exponentially large in the number of qubits. Verified compilers that are "correct-by-construction" address this issue [73]. However, when applying post-optimization, one needs to verify that the optimized circuit did not change the functionality, requiring to simulate complete quantum states in the worst-case.

References

[1] P. W. Shor, "Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer," *SIAM Journal on Computing*, vol. 26, no. 5, pp. 1484–1509, 1997.

[2] C. Pomerance, "A tale of two sieves," *Notices of the AMS*, vol. 43, no. 12, pp. 1473–1485, 1996.

[3] T. Häner, M. Roetteler, and K. M. Svore, "Factoring using $2n+2$ qubits with Toffoli based modular multiplication," *Quantum Information and Computation*, vol. 18, no. 7&8, pp. 673–684, 2017.

[4] M. Roetteler, M. Naehrig, K. Svore, and K. Lauter, "Quantum resource estimates for computing elliptic curve discrete logarithms," in *Proceedings of the 23rd Annual International Conference on the Theory and Applications of Cryptology and Information Security (ASIACRYPT'17), Hong King, China*, ser. Lecture Notes in Computer Science, vol. 10625. Springer, 2017, pp. 241–270.

[5] L. K. Grover, "A fast quantum mechanical algorithm for database search," in *Symposium on Theory and Computing*, 1996, pp. 212–219.

[6] M. Grassl, B. Langenberg, M. Roetteler, and R. Steinwandt, "Applying Grover's algorithm to AES: quantum resource estimates," in *Proceedings of the 7th International Conference on Post-Quantum Cryptography (PQCrypto'16), Fukuoka, Japan*, ser. Lecture Notes in Computer Science, vol. 9606. Springer, 2016, pp. 29–43.

[7] A. W. Harrow, A. Hassidim, and S. Lloyd, "Quantum algorithm for linear systems of equations," *Physical Review Letters*, vol. 103, no. 15, p. 150502, 2009.

[8] B. D. Clader, B. C. Jacobs, and C. R. Sprouse, "Preconditioned quantum linear system algorithm," *Physical Review Letters*, vol. 110, no. 25, p. 250504, 2013.

[9] A. Scherer, B. Valiron, S. Mau, D. S. Alexander, E. van den Berg, and T. 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*, vol. 16, no. 3, p. 60, 2017.

[10] R. P. Feynman, "Simulating physics with computers," *International Journal of Theoretical Physics*, vol. 21, pp. 467–488, 1982.

[11] T. H. Johnson, S. R. Clark, and D. Jaksch, "What is a quantum simulator?" *EPJ Quantum Technology*, vol. 1, no. 10, pp. 1–12, 2014.

[12] D. W. Berry, A. M. Childs, and R. Kothari, "Hamiltonian simulation with nearly optimal dependence on all parameters," in *IEEE 56th Annual Symposium on Foundations of Computer Science, FOCS 2015*, 2015, pp. 792–809.

[13] A. Aspuru-Guzik, A. D. Dutoi, and M. Love, Peter J.and Head-Gordon, "Simulated quantum computation of molecular energies," *Science*, vol. 309, pp. 1704–1707, 2005.

[14] D. Wecker, M. B. Hastings, N. Wiebe, B. K. Clark, C. Nayak, and M. Troyer, "Solving strongly correlated electron models on a quantum computer," *Physical Review A*, vol. 92, p. 062318, 2015.

[15] R. Somma, G. Ortiz, J. E. Gubernatis, E. Knill, and R. Laflamme, "Simulating physical phenomena by quantum networks," *Physical Review A*, vol. 65, p. 04323, 2002.

[16] B. Bauer, D. Wecker, A. J. Millis, M. B. Hastings, and M. Troyer, "Hybrid quantum-classical approach to correlated materials," *Physical Review X*, vol. 6, p. 031045, 2016.

[17] S. P. Jordan, K. S. M. Lee, and J. Preskill, "Quantum algorithms for quantum field theories," *Science*, vol. 336, pp. 1130–1133, 2012.

[18] F. T. Chong, D. Franklin, and M. Martonosi, "Programming languages and compiler design for realistic quantum hardware," *Nature*, vol. 549, no. 7671, pp. 180–187, 2017.

[19] D. Castelvecchi, "Quantum computers ready to leap out of the lab in 2017," *Nature*, vol. 541, no. 7635, pp. 9–10, 2017.

[20] IBM, "IBM builds its most powerful universal quantum computing processors," 2017, press release by IBM, posted online May 17, 2017.

[21] Rigetti, "Unsupervised machine learning on Rigetti 19Q with Forest1.2," 2017, press release by Rigetti, Inc., posted online December 18, 2017.

[22] Intel, "Intel delivers 17-qubit superconducting chip with advanced packaging to QuTech," 2017, press release by Intel, posted online October 10, 2017.

[23] IBM, "IBM announces advances to IBM quantum systems & ecosystem," 2017, press release by IBM, posted online Nov 10, 2017.

[24] E. Pednault, J. A. Gunnels, G. Nannicini, L. Horesh, T. Magerlein, E. Solomonik, and R. Wisnieff, "Breaking the 49-qubit barrier in the simulation of quantum circuits," *arXiv preprint arXiv:1710.05867*, 2017.

[25] T. Häner and D. S. Steiger, "0.5 petabyte simulation of a 45-qubit quantum circuit," in *Int'l Conf. on High Performance Computing, Networking, Storage and Analysis*, 2017, pp. 33:1–33:10.

[26] J. Miszczak, "Models of quantum computation and quantum programming languages," *Bull. Pol. Acad. Sci.-Tech. Sci.*, vol. 59, no. 3, pp. 305–324, 2011.

[27] A. Green, P. L. Lumsdaine, N. Ross, P. Selinger, and B. Valiron, "Quipper: A scalable quantum programming language," in *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*, 2013, pp. 333–342.

[28] A. JavadiAbhari, S. Patil, D. Kudrow, J. Heckey, A. Lvov, F. T. Chong, and M. Martonosi, "Scaffcc: Scalable compilation and analysis of quantum programs," *Parallel Computing*, vol. 45, pp. 2–17, 2015.

[29] J. Heckey, S. Patil, A. JavadiAbhari, A. Holmes, D. Kudrow, K. R. Brown, D. Franklin, F. T. Chong, and M. Martonosi, "Compiler management of communication and parallelism for quantum computation," in *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '15, Istanbul, Turkey, March 14-18, 2015*. ACM, 2015, pp. 445–456.

[30] D. Wecker and K. M. Svore, "LIQUi|>: A software design architecture and domain-specific language for quantum computing," 2014.

[31] J. Paykin, R. Rand, and S. Zdancewic, "QWIRE: a core language for quantum circuits," in *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, 2017, pp. 846–858.

[32] R. S. Smith, M. J. Curtis, and W. J. Zeng, "A practical quantum instruction set architecture," 2016, arXiv: 1608.03355.

[33] "Microsoft Quantum Development Kit," 2017, `https://github.com/microsoft/quantum`.

[34] D. S. Steiger, T. Haener, and M. Troyer, "ProjectQ: An open source software framework for quantum computing," *arXiv preprint arXiv:1612.08091*, 2016.

[35] K. M. Svore, A. V. Aho, A. W. Cross, I. Chuang, and I. L. Markov, "A layered software architecture for quantum computing design tools," *IEEE Computer*, vol. 39, no. 1, pp. 74–83, 2006.

[36] A. W. Cross, L. S. Bishop, J. A. Smolin, and J. M. Gambetta, "Open quantum assembly language," *arXiv preprint arXiv:1707.03429*, 2017.

[37] M. A. Nielsen and I. L. Chuang, *Quantum Computation and Quantum Information*. Cambridge University Press, 2000.

[38] M. Roetteler, M. Naehrig, K. M. Svore, and K. Lauter, "Quantum resource estimates for computing elliptic curve discrete logarithms," *Int'l Conf. on the Theory and Applications of Cryptology and Information Security*, 2017.

[39] M. Saeedi and I. L. Markov, "Synthesis and optimization of reversible circuits - a survey," *ACM Computing Surveys*, vol. 45, no. 2, pp. 21:1–21:34, 2013.

[40] A. Barenco, C. H. Bennett, R. Cleve, D. P. DiVincenzo, N. Margolus, P. Shor, T. Sleator, J. A. Smolin, and H. Weinfurter, "Elementary gates for quantum computation," *Physical Review A*, vol. 52, no. 5, p. 3457, 1995.

[41] N. Abdessaied, M. Amy, M. Soeken, and R. Drechsler, "Technology mapping of reversible circuits to Clifford+T quantum circuits," in *Int'l Symp. on Multiple-Valued Logic*, 2016, pp. 150–155.

[42] D. Maslov, "Advantages of using relative-phase Toffoli gates with an application to multiple control Toffoli optimization," *Physical Review A*, vol. 93, p. 022311, 2016.

[43] D. M. Miller, D. Maslov, and G. W. Dueck, "A transformation based algorithm for reversible logic synthesis," in *Design Automation Conference*, 2003, pp. 318–323.

[44] V. V. Shende, A. K. Prasad, I. L. Markov, and J. P. Hayes, "Synthesis of reversible logic circuits," *IEEE Trans. on CAD of Integrated Circuits and Systems*, vol. 22, no. 6, pp. 710–722, 2003.

[45] R. Wille and R. Drechsler, "BDD-based synthesis of reversible logic for large functions," in *Design Automation Conference*, 2009, pp. 270–275.

[46] M. Soeken, R. Wille, C. Hilken, N. Przigoda, and R. Drechsler, "Synthesis of reversible circuits with minimal lines for large functions," in *Asia and South Pacific Design Automation Conference*, 2012, pp. 85–92.

[47] A. De Vos and Y. Van Rentergem, "Young subgroups for reversible computers," *Advances in Mathematics of Communications*, vol. 2, no. 2, pp. 183–200, 2008.

[48] M. Saeedi, M. S. Zamani, M. Sedighi, and Z. Sasanian, "Reversible circuit synthesis using a cycle-based approach," *ACM Journal on Emerging Technologies in Computing Systems*, vol. 6, no. 4, p. 13, 2010.

[49] D. Große, R. Wille, G. W. Dueck, and R. Drechsler, "Exact synthesis of elementary quantum gate circuits," *Multiple-Valued Logic and Soft Computing*, vol. 15, no. 4, pp. 283–300, 2009.

[50] D. Maslov, G. W. Dueck, and D. M. Miller, "Techniques for the synthesis of reversible Toffoli networks," *ACM Trans. Design Autom. Electr. Syst.*, vol. 12, no. 4, p. 42, 2007.

[51] M. Soeken, L. Tague, G. W. Dueck, and R. Drechsler, "Ancilla-free synthesis of large reversible functions using binary decision diagrams," *Journal of Symbolic Computation*, vol. 73, pp. 1–26, 2016.

[52] M. Soeken, G. W. Dueck, and D. M. Miller, "A fast symbolic transformation based algorithm for reversible logic synthesis," in *Int'l Conf. on Reversible Computation*, 2016, pp. 307–321.

[53] M. Soeken, R. Wille, O. Keszocze, D. M. Miller, and R. Drechsler, "Embedding of large Boolean functions for reversible logic," *ACM Journal on Emerging Technologies in Computing Systems*, vol. 12, no. 4, pp. 41:1–41:26, 2016.

[54] A. Zulehner and R. Wille, "Make it reversible: Efficient embedding of non-reversible functions," in *Design, Automation and Test in Europe*, 2017, pp. 458–463.

[55] M. Soeken, M. Roetteler, N. Wiebe, and G. De Micheli, "Design automation and design space exploration for quantum computers," in *Design, Automation and Test in Europe*, 2017, pp. 470–475.

[56] K. Fazel, M. A. Thornton, and J. E. Rice, "ESOP-based Toffoli gate cascade generation," in *Pacific Rim Conference on Communications, Computers and Signal Processing*, 2007.

[57] A. Mishchenko and M. A. Perkowski, "Logic syntheis of reversible wave cascades," in *Int'l Workshop on Logic and Synthesis*, 2002.

[58] C. Bandyopadhyay, H. Rahaman, and R. Drechsler, "Improved cube list based cube pairing approach for synthesis of ESOP based reversible logic," *Transactions on Computational Science*, vol. 24, pp. 129–146, 2014.

[59] R. Drechsler, "Preudo-Kronecker expressions for symmetric functions," *IEEE Trans. on Computers*, vol. 48, no. 9, pp. 987–990, 1999.

[60] A. Mishchenko and M. A. Perkowski, "Fast heuristic minimization of exclusive-sum-of-products," in *Reed-Muller Workshop*, 2001.

[61] M. Soeken, R. Wille, and R. Drechsler, "Hierarchical synthesis of reversible circuits using positive and negative Davio decomposition," in *Int'l Design and Test Symp.*, 2010, pp. 143–148.

[62] A. Chattopadhyay, A. Littarru, L. G. Amarù, P.-E. Gaillardon, and G. De Micheli, "Reversible logic synthesis via biconditional binary decision diagrams," in *Int'l Symp. on Multiple-Valued Logic*, 2015, pp. 2–7.

[63] M. Soeken and A. Chattopadhyay, "Unlocking efficiency and scalability of reversible logic synthesis using conventional logic synthesis," in *Design Automation Conference*, 2016, pp. 149:1–149:6.

[64] B. Valiron, "Generating reversible circuits from higher-order functional programs," in *Int'l Conf. on Reversible Computation*, 2016, pp. 289–306.

[65] M. Soeken, M. Roetteler, N. Wiebe, and G. De Micheli, "Hierarchical reversible logic synthesis using LUTs," in *Design Automation Conference*, 2017, pp. 78:1–78:6.

[66] R. Královic, "Time and space complexity of reversible pebbling," in *Conf. on Current Trends in Theory and Practice of Informatics*, 2001, pp. 292–303.

[67] A. Parent, M. Roetteler, and K. M. Svore, "REVS: A tool for space-optimized reversible circuit synthesis," in *Int'l Conf. on Reversible Computation*, 2017, pp. 90–101.

[68] M. Soeken, S. Frehse, R. Wille, and R. Drechsler, "RevKit: A toolkit for reversible circuit design," *Multiple-Valued Logic and Soft Computing*, vol. 18, no. 1, pp. 55–65, 2012.

[69] M. Amy, D. Maslov, and M. Mosca, "Polynomial-time T-depth optimization of Clifford+T circuits via matroid partitioning," *IEEE Trans. on CAD of Integrated Circuits and Systems*, vol. 33, no. 10, pp. 1476–1489, 2014.

[70] M. Roetteler, "Quantum algorithms for highly non-linear Boolean functions," in *ACM-SIAM Symp. on Discrete Algorithms*, 2010, pp. 448–457.

[71] A. M. Childs and W. van Dam, "Quantum algorithms for algebraic problems," *Reviews of Modern Physics*, vol. 82, no. 1, pp. 1–52, 2010.

[72] S. Bravyi and D. Gosset, "Improved classical simulation of quantum circuits dominated by Clifford gates," *Physical Review Letters*, vol. 116, no. 25, p. 250501, 2016.

[73] M. Amy, M. Roetteler, and K. Svore, "Verified compilation of space-efficient reversible circuits," in *Computer Aided Verification*, 2017, pp. 3–21.