From Boolean functions to quantum circuits: A scalable quantum compilation flow in C++

Bruno Schmitt, Fereshte Mozafari, Giulia Meuli, Heinz Riener, Giovanni De Micheli
Integrated Systems Laboratory (LSI), EPFL, Switzerland

***Abstract*—We propose a flow for automated quantum compilation. Our flow takes a Boolean function implemented in Python as input and translates it into a format appropriate for reversible logic synthesis. We focus on two quantum compilation tasks: uniform state preparation and oracle synthesis. To illustrate the use of our flow, we solve IBM's virtual hackathon challenge of 2019, called *the Zed city problem*, an instance of vertex coloring, by using quantum search algorithms. The expressiveness of Python in combination with automated compilation algorithms allows us to express quantum algorithms at a high level of abstraction, which reduces the effort to implement them, and leads to better and more flexible implementations. We show that our proposed flow generates a lower-cost circuit implementation of the oracle needed to solve IBM's challenge when compared to the winning submission.**

***Index Terms*—quantum, design automation, compilation, reversible logic synthesis**

I. Introduction

A *quantum oracle* is a "black-box" operator that is used as an input to another algorithm. Such an oracle can often be understood as a classical computation specified by a Boolean function. The oracle gives access to the Boolean function, meaning that an algorithm can query the function on some input and observe its output. Oracles are widely used for studying the complexity of quantum algorithms[1] [1]. Counting the number of oracle queries needed to evaluate a function is easier than counting the number of computational steps. Thus, to try inferring nontrivial lower bounds more readily, quantum algorithm researchers characterize the computational complexity of an algorithm by the asymptotic growth rate of the number of queries with growing input size—an analysis that only assumes the existence of an oracle, and does not require its implementation.

However, to execute a quantum algorithm on a quantum computer, a concrete implementation has to be provided for each oracle. Such implementations must consist of the elementary quantum operators that are supported by the specific quantum computer. Also, due to the physical properties of quantum computation [2], all quantum operators need to be reversible. Therefore, a Boolean function that defines the behavior of an oracle must be reversible. Real-world problems, however, are often specified using oracles defined by complex and irreversible functions.

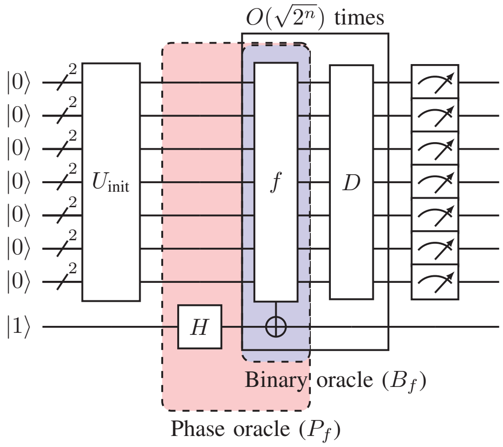


Fig. 1. Quantum circuit implementing a Grover's algorithm: U_{init} initializes the search space; H plus the bit oracle B_f define the phase oracle P_f; D is the diffusion operator. The circuit performs $O(\sqrt{2^n})$ iterations to find the unique satisfying input assignment of the Boolean function f

In this paper, we propose an automated compilation flow implemented in C++ to translate Boolean functions defined at a high level of abstraction into elementary quantum operators. As an illustrative example, we solve IBM's virtual hackathon challenge of 2019, called *the Zed city problem*. We cast the problem to an instance of Boolean satisfiability and use Grover's search algorithm [3], [4] to solve it. Our compilation enables the programmer to describe quantum circuits by defining each component at a high-level of abstraction, as, e.g., shown in Fig. 1. In particular, the oracles are specified by Boolean functions.

A Boolean satisfiability problem [5] consists of a set of Boolean variables and a set of logical relations (or "constraints"). A solution is a variable assignment that maps every variable to a Boolean value such that all constraints are satisfied, which we represent by a bit string. When using Grover's algorithm to solve such a problem, we need a Boolean function f capable of recognizing solutions, an oracle—which takes as input a variable assignment and evaluates to true only if it satisfies all constraints. We map each variable assignment to a quantum state. The algorithm starts the search by using U_{init} to create an uniform superposition of all such states. Then it iterates over a quantum implementation of f that marks all solution states, and a quantum operator D, known as diffusion, that amplifies the amplitudes of these marked states. Finally, since we amplified the amplitudes of the solution states, the final measurements will return a solution with high probability.

This research was supported by the EPFL Open Science Fund, by the Swiss National Science Foundation (200021-169084 MAJesty) and by the ERC project H2020-ERC-2014-ADG 669354 CyberCare.

[1] quantumalgorithmzoo.org/#oracular

978-3-9819263-5-4/DATE21/©2021 EDAA

In Boolean satisfiability, there is often a degree of commonality between various non-solutions. For example, one typically knows beforehand that some assignments (or combinations of assignments) of the variables are inconsistent, i.e., violate one or more of the constraints, and cannot participate in any solution. In our example, we characterize these commonalities using a Boolean function g and exploit them to direct the search to a solution. In practice, we modify U_{init} to create a uniform superposition that rules out trivial non-solutions states. This increases the cost of implementing U_{init}, but can substantially simplify the implementation of f and decrease the number of iterations over f and D.

The framework allows both f and g to be defined in Python. The Python functions are transformed into classical logic networks which, in turn, are compiled into elementary quantum operators. We delve into the details of the state-of-the-art techniques used (1) to compile g and, thus, create U_{init} (i.e., a process called uniform quantum state preparation) and, (2) to compile f into the oracle implementation.

II. Background

A. Boolean Functions

A *Boolean variable* x is a variable that takes one of the two values from the domain $\mathbb{B} = \{0, 1\}$. A *positive literal* is the Boolean variable x and a *negative literal* is its complement $\bar{x}$. The Boolean AND of k literals is a *cube*, or product, i.e., $c = l_1 \wedge \cdots \wedge l_k$ (we may omit the symbol $\wedge$ in forming cubes, e.g., $l_1 \wedge \cdots \wedge l_k = l_1 \cdots l_k$). If a variable is not represented by a positive or negative literal in a cube, then its value is said to be a *don't care literal*. A *minterm* is a cube, in which every variable is represented by either a negative or positive literal. A cube with k don't care literal covers 2^k minterms.

For the sake of clarity, we limit our discussion to single-output Boolean functions and formally define a *Boolean function* as a mapping $f : \mathbb{B}^n \to \mathbb{B}$. A *cofactor* is the function derived by substituting constant values for some of its variables. For example, *Boole's expansion* of a function f, often also called *Shannon's expansion*, is defined as $f = x_i f_{x_i} + \bar{x}_i f_{\bar{x}_i}$ where $f_{\bar{x}_i} = f(x_i = 0)$ and $f_{x_i} = f(x_i = 1)$ are the negative and positive cofactors of the function f with respect to variable x_i, respectively.

B. Quantum Computing

Quantum computers are physical machines consisting of an array of quantum bits, the so-called *qubits*, whose state can be modified by quantum operators, typically referred to as quantum gates. A quantum algorithm describes how to transform the state of a quantum computer to solve a computational task. Such an algorithm contains both classical operations and quantum operators, and its execution takes place on a classical host computer that decides on the sequence of quantum operators to be send to a quantum co-processor. This sequence is a *quantum circuit*. Fig. 1 shows a quantum circuit diagram. This diagram is read from left to right with each horizontal line representing a qubit and each box/symbol on a line representing a quantum gate.

Formally, a qubit is in a quantum state that is a column vector $|\varphi\rangle = \binom{\alpha}{\beta}$ of two complex numbers α and β, called amplitudes, such that $|\alpha|^2 + |\beta|^2 = 1$. The squared amplitudes $|\alpha|^2$ and $|\beta|^2$ indicate, respectively, the probability that the quantum state will collapse to one of the classical states 0 or 1 after the qubit is measured. Hence, $|0\rangle = \binom{1}{0}$ and $|1\rangle = \binom{0}{1}$ are two orthonormal basis states, $\{|0\rangle, |1\rangle\}$, called the *computational basis states*, that span the two-dimensional linear vector space of a qubit. We model quantum operators that modify the state of one qubit using 2×2 unitary matrices. For example, the Hadamard gate $H = \frac{1}{\sqrt{2}} \left(\begin{smallmatrix} 1 & 1 \\ 1 & -1 \end{smallmatrix}\right)$ transforms the computational basis state $|0\rangle$ into the state $\frac{1}{\sqrt{2}}\binom{1}{1}$, which is in the uniform superposition between $|0\rangle$ and $|1\rangle$.

Quantum states over n qubits are represented by a column vector of 2^n complex amplitudes α_x with $x \in \mathbb{B}^n$ such that $\sum_{x \in \mathbb{B}^n} |\alpha_x|^2 = 1$. Each squared amplitude $|\alpha_x|^2$ indicate the probability that after measurement the n qubits are in the classical state x. Quantum states can be combined by applying the Kronecker product to form larger ones, e.g., $\binom{1}{0} \otimes \frac{1}{\sqrt{2}}\binom{1}{1} = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 \\ 1 \\ 0 \\ 0 \end{pmatrix}$, which represents a 2-qubit state that is in the uniform superposition between the classical states 00 and 01. Such operators act on n qubits and are represented in terms of $2^n \times 2^n$ unitary matrices.

C. Oracle access to Boolean functions

We say that the oracle gives access to a Boolean function, meaning that an algorithm that uses the oracle only has access to the function's input and output, not its internal structure. In the following, we describe two natural ways of implementing an oracle characterized by a Boolean function f on a quantum computer that are necessary to understand our flow and the Grover search example.

a) Bit oracle: A bit oracle is a quantum operator B_f specified by a Boolean function f for which the effect on all computational basis states is given by

$$B_f : |x\rangle|y\rangle|0\rangle^a \mapsto |x\rangle|y \oplus f(x)\rangle|0\rangle^a, \tag{1}$$

where '$\oplus$' is the logical exclusive-or operator and $a \geq 0$ corresponds to the number of extra qubits used to store intermediate results for the computation of $f(x)$, the so-called *ancillæ qubits*.

b) Phase oracle: A phase oracle is also a quantum operator specified by a Boolean function f. However, its effect on all computational basis states is given by

$$P_f |x\rangle = (-1)^{f(x)} |x\rangle. \tag{2}$$

Eq. 2 means that if x is not a satisfying input, the oracle does nothing to its corresponding state $|x\rangle$. Otherwise, it rotates the states' phase by π (or 180 degrees).

It turns out that these two oracle models are almost equivalent: the phase oracle can be obtained from one use of the bit oracle and the use of the Hadamard operator on the output qubit as shown in Fig. 1.

D. Grover's Algorithm

Grover has shown that, by using quantum mechanics, a database of unsorted data can be searched quadratically faster than any classical search [3]. Given a Boolean function $f(x)$, where x is a bit string of length n, which evaluates to true for exactly one assignment $\hat{x}$, i.e., $f(\hat{x}) = 1$, the algorithm determines $\hat{x}$ with a high probability by querying f only $O(\sqrt{2^n})$. Classical algorithms for the same task require $O(2^n)$ queries.

The basic idea of Grover's algorithm is to invert the phase of the desired basis state, and then invert all the basis states about the average amplitude of all the states. The algorithm uses $n+1$ qubits, where the first n of them are initialized with $|0\rangle$ and the last one is initialized with $|1\rangle$. The initialization operator U_{init} creates a uniform superposition of all classical states that are inputs to the oracle function f and then repeatedly applies two operators to the state: (1) The first operator is a bit oracle of the Boolean function f that is cast into a phase oracle. (2) The second operator

$$D = U_{\text{init}} \cdot (2|0^n\rangle\langle 0^n| - I_{2^n}) \cdot U_{\text{init}}^{\dagger}$$

is a $2^n \times 2^n$ diffusion operator. Here, $|0^n\rangle$ is the classical state represented by the bit string with n zeros, I_{2^n} is the identity operator of size $2^n \times 2^n$, and $U_{\text{init}}^{\dagger}$ is the adjoint operator of $U_{\text{init}}^{\dagger}$. In Fig. 1, the last gates on the first n qubits denote a measurement operation.

Note that Grover's algorithm also works for Boolean functions f that evaluate to true under multiple assignments [3], [6]. In general, any search problem can be recast as the problem of finding the value(s) of x at which an "oracle" Boolean function $f(x)$ evaluates to true.

III. Example: The Zed City problem

We will use a simple example to illustrate our compilation flow, called *the Zed city problem*, which is an instance of vertex coloring. In this section, we will first introduce the problem and then show how to apply Grover's search to solve it.

Vertex coloring is the problem of assigning colors to vertices of a graph such that adjacent vertices are not of the same color. Our example is a variation of this problem presented in the final challenge of IBM's virtual hackathon[2]: Zed city is a newly established (fictitious) municipality in Tokyo composed of 11 districts. Four convenience store chains A, B, C, and D have each built their first store in this new city. The goal is to use vertex coloring to distribute stores in the districts that still do not have one yet, while ensuring that there is only one store per district and that adjacent districts do not have stores from the same chain. Fig. 2 shows Zed city as an undirected graph.

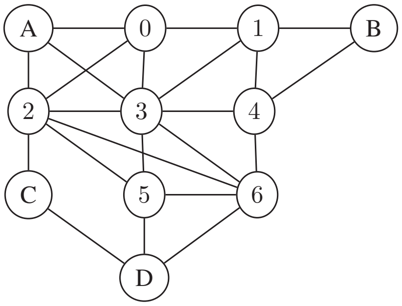


Fig. 2. Zed city as an undirected graph: each district represented by a node, an edge between two nodes indicates that two district are adjacent.

For Boolean modelling, we assign to each vertex in the graph a binary string that represents a color and formulate the following constraints: (1) every vertex must have one color assigned to it and (2) two adjacent vertices cannot have the same color. We define the four colors, A ('00'), B ('01'), C ('10'), and D ('11'). For each vertex $i = 0, \dots, 6$, we create a variable v_i, which is a bit string of length 2. The problem can then be modelled as an oracle, a Boolean function $f(v_0, \dots, v_6)$ that evaluates to 1 only for those variable assignment that represent a graph coloring satisfying all constraints. Classically, we can solve a vertex coloring problem that is model as such by querying f with all input combinations until we find one for which f evaluates to one. Hence, in the worst case, we will query the oracle $O(2^n)$ times. As discussed in Section II-D, however, a quantum computer can solve such a problem with high probability by querying f only $O(\sqrt{2^n})$.

We need to concretely implement the circuit in Fig. 1 to execute it on a quantum computer. Naïvely, the initialization creates the superposition of all classical states by applying the Hadamard gate H to all qubits. Next, we synthesize B_f from a high-level classical definition of f:

Listing 1. Python implementation of f

```
def f(v0, ..., v6 : BitVec(2)) -> BitVec(1):
    c0 = (v0 != ’00’)
    c1 = (v1 != ’01’) and (v1 != v0)
    c2 = (v2 != ’00’) and (v2 != ’10’) and
         (v2 != v0)
    c3 = (v3 != ’00’) and (v3 != v0) and
         (v3 != v1)   and (v3 != v2)
    c4 = (v4 != ’01’) and (v4 != v1) and (v4 != v3)
    c5 = (v5 != ’11’) and (v5 != v2) and (v5 != v3)
    c6 = (v6 != ’11’) and (v6 != v2) and
         (v6 != v3)   and (v6 != v4) and (v6 != v5)
    return c0 and c1 and c2 and
           c3 and c4 and c5 and c6
```

Clearly, this function returns 1 only when all vertices are colored and no adjacent vertices have the same color. We can simplify f by ensuring that we will never call it with input combinations beforehand known to be inconsistent:

Listing 2. Hand-optimized Python implementation of f

```
def f(v0, ..., v6 : BitVec(2)) -> BitVec(1):
    c1 = (v1[0] == v1[1]) and (v3 != v1)
    c023 = ((v0 ^ v2 ^ v3) == ’00’)
    c4 = (v4 != n1) and (v4 != v3)
    c5 = (v5 != n2) and (v5 != v3)
    c6 = ((v2 ^ v3 ^ v5 ^ v6) == ’00’) and
         (v6 != v4)
    return c1 and c023 and c4 and
           c5 and c6
```

IV. Compilation flow

The compilation of the circuit in Fig. 1 can be broken into two tasks: (1) The compilation U_{init} using an uniform state preparation technique and (2) the compilation B_f using oracle synthesis. We leverage the EPFL quantum compilation

[2] github.com/quantum-challenge/2019/

libraries [7] Tweedledum[3], Angel[4], and Caterpillar[5], to create our flow. Tweedledum is a C++ library for synthesizing, manipulating, and optimizing quantum circuits. It provides the other libraries with the means to represent quantum circuits in various levels of abstraction that can be part of the same circuit. Angel is used to prepare a uniform quantum state given as input a Boolean function. Caterpillar is used to automatically translate the combinational parts of a quantum algorithm into quantum gates. We interface the flow with IBM's Qiskit framework [8] to execute the compiled circuit on a quantum computer or in a high-performance simulator.

A. Compiling U_{init}

To use the optimized version of f, we need an initial uniform quantum state $|\varphi_{\text{init}}\rangle$ in which the amplitude of the invalid classical input states are zero. For example, in the Zed city problem, the states in which $v_0 = 00$ are non-solutions. We characterize the uniform quantum state $|\varphi_{\text{init}}\rangle$ by a Boolean function $g(v_0, \dots, v_6)$ such that

$$|\varphi_{\text{init}}\rangle = \frac{1}{\sqrt{|\mathsf{Min}(g)|}} \sum_{x \in \mathsf{Min}(g)} |x\rangle, \tag{3}$$

where x is the concatenation of the variables v_0 to v_6 into a single bit string, and $\mathsf{Min}(g)$ is the set of all minterms of g. We can define g on the same level of abstraction as f:

Listing 3. Python implementation of g

```
def g(v0, ..., v6 : BitVec(2)) -> BitVec(1):
    return (v0 != ’00’) and (v1 != ’01’) and
           (v2 != ’00’) and (v2 != ’10’) and
           (v3 != ’00’) and (v4 != ’01’) and
           (v5 != ’11’) and (v6 != ’11’)
```

Given an n-qubit uniform quantum state $|\varphi_{\text{init}}\rangle$, the *uniform quantum state preparation problem* asks for a quantum circuit in terms of rotation gates and CNOTs, specified by quantum operation

$$U_{\text{init}} : |0\rangle^{\otimes n} \to |\varphi_{\text{init}}\rangle, \tag{4}$$

that, when applied to the canonical n-qubit quantum state $|0\rangle^{\otimes n}$ transforms $|0\rangle^{\otimes n}$ into $|\varphi_{\text{init}}\rangle$.

Using the Boolean function g, our flow uses a technique that employs the Shannon decomposition ($g = x_i g_{x_i} + \bar{x}_i g_{\bar{x}_i}$) to solve the state preparation problem, recursively. The technique described in [9] takes as input a *Binary Decision Diagrams* (BDDs) [10] representation of g and constructs the desired quantum circuit by iterating over the variables of g in an given order. These variables x_i correspond to the qubits q_i. We prepare the qubits one by one by computing the probability of their corresponding variable being zero given the probabilities of previously prepared qubits $p(\bar{x}_i)$. This computational step requires to count the number of ones for each recursive cofactor of g. The probability is then the number of ones of the current decomposed function divided by the number of ones of its negative cofactor. In other words, we can formulate the general idea of our state preparation algorithm as

$$U_{\text{init}}|0\rangle^{\otimes n} = (U_{\text{init}_{\bar{x}_i}} \oplus U_{\text{init}_{x_i}})(G(p(\bar{x}_i)) \otimes I_{2^n-1})|0\rangle^{\otimes n}$$

where $G(p(\bar{x}_i)$ is a unitary transformation gate that satisfies

$$G(p(\bar{x}_i)|0\rangle = \sqrt{p(\bar{x}_i}|0\rangle + \sqrt{1 - p(\bar{x}_i)}|1\rangle. \tag{5}$$

The resulting quantum circuit consists of a sequence of multiple-controlled $p(\bar{x}_i)$. From the definition of $R_y(\theta)$ one can readily derive that

$$G(p(\bar{x}_i)) = R_y\left(2\cos^{-1}(\sqrt{p(\bar{x}_i)})\right). \tag{6}$$

Consequently, by replacing all gates on the target line by R_y gates, we obtain a circuit consisting only of multiple-controlled R_y gates. Different decomposition methods exist used to transform these gates into a sequence of elementary quantum gates {CNOTs, $R_y(\theta)$} [11]–[13].

B. Compiling B_f

In this section we describe how our framework tackles the problem of compiling quantum circuits implementing Boolean functions. As the native operations of quantum systems are reversible, a reversible circuit must be derived from the specification of the Boolean function. The literature presents several algorithms for the the synthesis of reversible circuits, nevertheless some of them require a reversible input Boolean function [14], [15]. The methods described here are capable of compiling the oracle even if it is specified by an irreversible Boolean function.

Given an irreversible function f, it is known that there must exist a reversible Boolean function $f' : \{0,1\}^{n+1} \mapsto \{0,1\}^{n+1}$ such that

$$f'(x, y) = (x, y \oplus f(x)),$$

where $x = x_0, \dots, x_n$ and '$\oplus$' refers to the XOR operation. Such an embedding is also referred to as Bennett embedding [16], and implies the existence of the following quantum operation:

$$B_f : |x\rangle|y\rangle \mapsto |x\rangle|y \oplus f(x)\rangle \tag{7}$$

The operation B_f is also known as a single-target gate. Single-target gates describe complex operations that cannot generally be implemented natively on a quantum computer, hence the need to automatically compile them into quantum gates. Tweedledum provides several techniques to directly compile single-target gates. Among them, for our illustrative experiments we will make use on an ESOP-based technique. Starting from a functional representation of f, i.e., a truth table, the technique synthesizes a special case of a 2-level ESOP expression for f, a Pseudo-Kronecker expression [17]. The expression is used to decompose the single-target gate into multiple-control Toffoli gates, that are then decomposed using state-of-the-art techniques. This approach can perform the decomposition using at most one ancilla. However, it is only applicable to small Boolean functions as it can be both very time consuming and generate quantum circuit with a prohibitive number of gates [18], [19].

[3] github.com/boschmitt/tweedledum
[4] github.com/fmozafari/angel
[5] github.com/gmeuli/caterpillar

For a more scalable solution, we combine similar direct methods with the hierarchical synthesis approaches provided by Caterpillar. The latter allows us to achieve scalability by decomposing the initial function and storing intermediate results on ancilla qubits. Given an irreversible Boolean function f they find an $(n+1+a)$-qubit quantum circuit that realizes the unitary

$$B_f : |x\rangle|y\rangle|0\rangle^a \mapsto |x\rangle|y \oplus f(x)\rangle|0\rangle^a \tag{8}$$

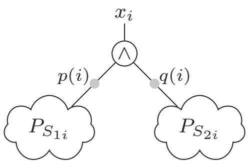
where $a > 0$, which means that the synthesis algorithm can use the a additional qubits to store intermediate computations. We describe here two of such hierarchical approaches.

a) XAG-based [20]: This technique is based on representing the Boolean function using a logic network over the gate basis $\{\neg, \oplus, \wedge\}$, i.e. an Xor-And-inverter Graph (XAG). Formally, an XAG is a logic networks in which each local function is either a 2-input AND or a 2-input XOR, i.e., each step has one of the two following forms:

$$x_i = x_{j_{1i}} \oplus x_{j_{2i}} \quad \text{or} \quad x_i = x_{j_{1i}}^{p_{1i}} \wedge x_{j_{2i}}^{p_{2i}}, \tag{9}$$

where p_{1i} and p_{2i} are Boolean constants used to possibly complement the gate's fan-in.

The key idea in reference [20, Algorithm 1] is to look at the two inputs of an AND gate as two parity functions over variables that are either primary inputs or preceding AND steps:



,

where $P_{S_{1i}}$ and $P_{S_{2i}}$ are the parity functions over the set of variables indexed by S_{1i}, $S_{2i} \subseteq [1, \ldots, i-1]$. Since the parity functions are reversible, they can be easily computed in-place, i.e., without the need for an extra qubit. The AND steps, on the other hand, uses out-of-place computation in order to exploit state-of-the-art quantum implementation of the logic AND function with a reduced number of gates [21]–[23].

The algorithm starts by computing each AND step in topological order in three simple steps: first, it computes the input parity functions in-place, then it implements the logic AND using a new ancilla, and finally it cleans up the parity functions. Once all AND steps are computed, the states of qubits holding the output steps are copied to the output qubits. Finally, all ancillæ are restored to $|0\rangle$ by cleaning up all AND steps in anti-topological order.

One of the major advantages of XAG-based synthesis is that the resources required by the resulting quantum circuits can be predicted by inspecting some structural properties of the graph. This for example includes the number of ancillæ, which depends on the number of AND steps in the XAG that is commonly called *multiplicative complexity*. It follows that we can rely on classical logic synthesis techniques to optimize the multiplicative complexity of the initial XAG. Nevertheless, as the problem of determining the multiplicative complexity is intractable, a hard limit on the number of ancillæ may be prohibitive for such approach. Then, one option provided is to use SAT to find the best strategy to compute and uncompute the required intermediate results using a target number of qubits a. Such techniques are called pebbling strategies and enables to explore the trade-off between qubits and gates.

b) LUT-based [24]: Another hierarchical method available in Caterpillar is based on k-LUT networks. The parameter k provides a certain control over the number of ancillæ a. The technique is based on the usage of k-feasible Boolean logic networks (k-LUT networks), which consist of lookup-tables (LUTs) with at most k inputs. Synthesis proceeds in two steps: first, each k-LUT is translated to a single-target gate with k control lines in a reversible logic network. Second, each single-target gate is compiled. Note that by using bigger LUTs, we can minimize the number of ancilla qubits. The size of the LUTs, however, is limited by the scalability of the single-target gate synthesis approach. As detailed in [24] the k-LUT network must isolate LUTs with functions conveniently matched to generate fewer gates upon decomposition of the corresponding single-target gate into quantum gates. For example, when using Gray synthesis [25] to decompose single-target gates, the LUT functions should contain few non-zero coefficients into the Rademacher-Walsh spectrum.

As for the XAG-based technique, we can incorporate pebbling strategies to further trade-off the number of qubits for quantum operators.

This section described how the techniques available in our tools can control the number of generated ancillæ a. Nevertheless, the existence of a compilation technique that takes a as an input parameter and guarantees to return a quantum circuit that satisfies the space requirement is still to be determined.

V. Experimental Results

We evaluate our flow by solving the Zed city problem in IBM's challenge, which further imposes a constraint on the number of qubits: a solution must use at most 32 qubits. We use IBM's challenge as an example and evaluation because highly-optimized handcraft solutions are available. We use these solutions as a baseline. First, we compare the code readability: The oracle in Listing 1 is objectively simpler to understand and implement than any of the submitted solutions. The hand-optimized version, Listing 2, is more involved, but arguably less complicated than the top submissions[6], which define U_{init} and B_f in terms of primitive quantum operators.

In Table I, we report the results of compiling B_f when using both the hand-optimized and non-optimized versions of f. As baseline, we use IBM's sample solution and the winner submission from team `Whit3z` with the same cost function as in the challenge, i.e., $\text{cost} = n_{1\text{q}} + 10 \cdot n_{2\text{q}}$, where $n_{1\text{q}}$ is the number one-qubit operators and $n_{2\text{q}}$ the number of two-qubit operators.

First, obverse that the non-optimized implementation only meet the 32-qubit constraint when synthesized with pebbling. The results for our hand-optimized oracle have a slightly lower cost than the handcrafted solutions. As expected, we obverse a trade-off between the number of operations and the number of qubits.

[6] github.com/quantum-challenge/2019/tree/master/top ten submissions

TABLE I
QUALITY OF RESULTS FOR B_f (HAND-OPTIMIZED AND NON-OPTIMIZED)

	Hand-optimized		Non-optimized	
	Qubits	cost	Qubits	cost
IBM's solution	32	5004		
`Whit3z` solution	32	2474		
XAG-based flow	31	2202	56	4347
XAG-based flow with pebbling	21	4497	30	7737

The use of our hand-optimized oracle has a significant impact on the implementation of U_{init} since it requires ensuring that the amplitude of computational basis states corresponding to known non-solutions is zero. Due to its generality and lack of high-level information, Angel's uniform state preparation technique generates a high-cost implementation of U_{init}.

VI. Challenges and conclusions

We presented a design flow for automated quantum compilation of Boolean functions. Our flow takes as input a quantum program implemented in Python and translates it using logic synthesis techniques. Automated compilation enables rapid design-space exploration and gives designers the flexibility to optimize for a variety of different cost metrics such as the number of T gates or the number of qubits.

Particularly, we focused on two quantum compilation tasks: uniform state preparation and oracle synthesis. To illustrate our flow, we use them to solve IBM's virtual hackathon challenge of 2019. The expressiveness of Python in combination with scalable compilation algorithms allows us to express quantum algorithms at a high level without being burdened with specifying each single quantum operation in detail. Ultimately, this (1) simplifies the implementation task, since the tedious manual compilation of combinational components is automatized, and (2) enables rapid development of more complex algorithms by using abstract high-level constructs. These capabilities are absent in existing approaches for quantum circuit compilation.

Several challenges remain and are awaiting satisfactory solutions. Automated compilation of large functions requires reversible logic synthesis methods that need additional qubits. Typically, the execution of the compilation algorithm determines the number of ancilla qubits, i.e., it cannot be bounded apriori. Techniques that find a solution without exceeding a given number of ancillæ are still rare. Advances in automated state preparation are required to raise the level of abstraction and to identify world-level relation among input bits in order to minimize the cost of initializing an input quantum state—an ubiquitous operation in quantum algorithms.

References

[1] A. Ambainis, "Understanding quantum algorithms via query complexity," *arXiv preprint arXiv:1712.06349*, vol. 244, 2017.

[2] M. Nielsen and I. Chuang, *Quantum Computation and Quantum Information*, 2nd ed. Cambridge University Press, 2010.

[3] L. K. Grover, "A fast quantum mechanical algorithm for database search," in *Proceedings of the Twenty-Eighth Annual ACM Symposium on the Theory of Computing, Philadelphia, Pennsylvania, USA, May 22-24, 1996*, 1996, pp. 212–219. [Online]. Available: https://doi.org/10.1145/237814.237866

[4] ——, "Quantum mechanics helps in searching for a needle in a haystack," *Physical Review Letters*, vol. 79, no. 2, pp. 325–328, 1997. [Online]. Available: https://link.aps.org/doi/10.1103/PhysRevLett.79.325

[5] A. Biere, M. Heule, and H. van Maaren, *Handbook of satisfiability*. IOS press, 2009, vol. 185.

[6] M. Boyer, G. Brassard, P. Høyer, and A. Tapp, "Tight bounds on quantum searching," *Fortschritte der Physik*, vol. 46, no. 4-5, pp. 493–505, 1998.

[7] M. Soeken, H. Riener, W. Haaswijk, E. Testa, B. Schmitt, G. Meuli, F. Mozafari, and G. De Micheli, "The EPFL logic synthesis libraries," *arXiv preprint arXiv:1805.05121v2*, 2019.

[8] H. Abraham *et al.*, "Qiskit: An open-source framework for quantum computing," 2019.

[9] F. Mozafari, M. Soeken, H. Riener, and G. De Micheli, "Automatic uniform quantum state preparation using decision diagrams," in *IEEE International Symposium on Multiple-Valued Logic, ISMVL 2020, Miyazaki, Japan, November 9-11, 2020*, 2020, p. to appear.

[10] R. E. Bryant, "Graph-based algorithms for Boolean function manipulation," *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.*, vol. 35, no. 8, pp. 677–691, 1986. [Online]. Available: https://doi.org/10.1109/TC.1986.1676819

[11] M. Möttönen, J. J. Vartiainen, V. Bergholm, and M. M. Salomaa, "Quantum circuits for general multiqubit gates," *Physical Review Letters*, vol. 93, no. 13, p. 130502, 2004.

[12] D. Maslov, "Advantages of using relative-phase Toffoli gates with an application to multiple control Toffoli optimization," *Physical Review A*, vol. 93, no. 2, p. 022311, 2016.

[13] M. Soeken, F. Mozafari, B. Schmitt, and G. De Micheli, "Compiling permutations for superconducting QPUs," in *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2019, pp. 1349–1354.

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

[15] 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. [Online]. Available: http://dx.doi.org/10.3934/amc.2008.2.183

[16] C. H. Bennett, "Logical reversibility of computation," *IBM Journal of Research and Development*, vol. 17, no. 6, pp. 525–532, 1973. [Online]. Available: http://ieeexplore.ieee.org/document/5391327/

[17] M. Davio, A. Thayse, and J. P. Deschamps, *Discrete and switching functions*. McGraw-Hill, 1978.

[18] B. Schmitt, M. Soeken, G. De Micheli, and A. Mishchenko, "Scaling-up ESOP synthesis for quantum compilation," in *2019 IEEE 49th International Symposium on Multiple-Valued Logic (ISMVL)*. IEEE, 2019, pp. 13–18. [Online]. Available: https://ieeexplore.ieee.org/document/8758744/

[19] H. Riener, R. Ehlers, B. d. O. Schmitt, and G. De Micheli, "Exact synthesis of ESOP forms," in *Advanced boolean techniques*. Springer, 2020, pp. 177–194.

[20] G. Meuli, M. Soeken, E. Campbell, M. Roetteler, and G. De Micheli, "The role of multiplicative complexity in compiling low T-count oracle circuits," in *Proceedings of the International Conference on Computer-Aided Design, ICCAD 2019, Westminster, CO, USA, November 4-7, 2019*, 2019, pp. 1–8. [Online]. Available: https://doi.org/10.1109/ICCAD45719.2019.8942093

[21] C. Jones, "Low-overhead constructions for the fault-tolerant Toffoli gate," *Phys. Rev. A*, vol. 87, no. 2, p. 022328, 2013.

[22] P. Selinger, "Quantum circuits of T-depth one," *Phys. Rev. A*, vol. 87, p. 042302, 2013.

[23] C. Gidney, "Halving the cost of quantum addition," *Quantum*, vol. 2, no. 74, pp. 10–22 331, 2018.

[24] G. Meuli, M. Soeken, M. Roetteler, and G. De Micheli, "ROS: Resource constrained oracle synthesis for quantum circuits," in *Quantum Physics and Logic*, 2019.

[25] M. Amy, P. Azimzadeh, and M. Mosca, "On the controlled-NOT complexity of controlled-NOT–phase circuits," *Quantum Science and Technology*, vol. 4, no. 1, p. 015002, 2019.