IEEE Transactions on Quantum Engineering

Received 29 January 2023; revised 11 April 2023; accepted 26 April 2023; date of publication 18 May 2023; date of current version 4 July 2023.

Digital Object Identifier 10.1109/TQE.2023.3275868

isQ: An Integrated Software Stack for Quantum Programming

JINGZHE GUO[1], HUAZHE LOU[2], JINTAO YU[2], RILING LI[3], WANG FANG[3], JUNYI LIU[3], PEIXUN LONG[3], SHENGGANG YING[3], AND MINGSHENG YING[1,3]

[1]Department of Computer Science and Technology, Tsinghua University, Beijing 100190, China
[2]Arclight Quantum Computing Inc., Beijing 100086, China
[3]State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences, Beijing 100045, China

Corresponding authors: Riling Li; Mingsheng Ying (e-mail: lirl@ios.ac.cn; yingms@ios.ac.cn).

This work was supported in part by the National Key Research and Development Program of China under Grant 2018YFA0306700, in part by the National Natural Science Foundation of China under Grant 61832015, and in part by the Young Scientists Fund of the National Natural Science Foundation of China under Grant 62002349. *(Jingzhe Guo and Huazhe Lou contributed equally to this work.)*

ABSTRACT We introduce isQ, a new software stack for quantum programming in an imperative programming language, also named isQ. The aim of isQ is to make programmers write quantum programs as conveniently as possible. In particular, 1) the isQ language and its compiler contain many useful features, including but not limited to classical control flow, such as recursion, decomposition of self-defined unitary gates, and oracle programming and its circuit realization. 2) To make it flexible, an isQ program can be compiled into several different kinds of intermediate representation and assemblies, including QIR, eQASM, OpenQASM 3.0, and QCIS (specially tailored for the superconducting quantum hardware at the University of Science and Technology of China). 3) Besides interfacing isQ with real superconducting hardware, a QIR simulator is also developed for the demonstration and testing of isQ programs. The isQ software stack encompasses abundant compiler optimizations of high-level quantum programs. To realize it, a distinct multilevel intermediate representation (MLIR) dialect name isQ-IR is proposed.

INDEX TERMS Compilers, programming languages, quantum computing stacks.

I. INTRODUCTION

Recent progress in quantum hardware has convinced people of the possibility that quantum computers outperform their classical predecessors in solving some important problems [1], [2], [3], [4]. As quantum hardware keeps advancing, we have entered a noisy-intermediate-scale-quantum (NISQ) era. However, just like classical computers, quantum hardware cannot unleash its full power unless equipped with a series of quantum computing software, including but not limited to the following:

1) quantum programming language(s) for writing quantum programs;
2) quantum compilers for transformation and optimization of quantum programs, as well as compiling quantum programs to different hardware, e.g., superconducting and trapped-ion devices;
3) quantum simulators for debugging small-scale quantum programs.

This article presents isQ, an integrated software stack for quantum programming.

The structure of isQ is visualized in Fig. 1. isQ was first proposed as an experimental and educational language. The first version of its compiler was implemented in Python using the PLY [5] module, converting the abstract syntax tree from the input isQ programs directly into a modified Quil intermediate representation (IR) [6]. The original isQ language has limited features, and the compiler is not extensible, e.g., it is hard to implement sophisticated compiler optimizations [7]. To design a more powerful software stack that permits more language features and various compiler optimizations, we extended the isQ language and reimplemented the compiler based on multilevel intermediate representation (MLIR) [8] infrastructure. Specifically, we designed isQ-IR, an MLIR dialect, as our compiler frontend target and implemented various transformation passes that are useful for compiling quantum programs, e.g., gate decomposition and qubit mapping. We also implemented code generators that convert isQ-IR to different kinds of low-level and assemblies, including QIR [9], eQASM [10], OpenQASM 3.0 [11], and QCIS [12].

Main Contributions: More explicitly, the contributions of this article are as follows.

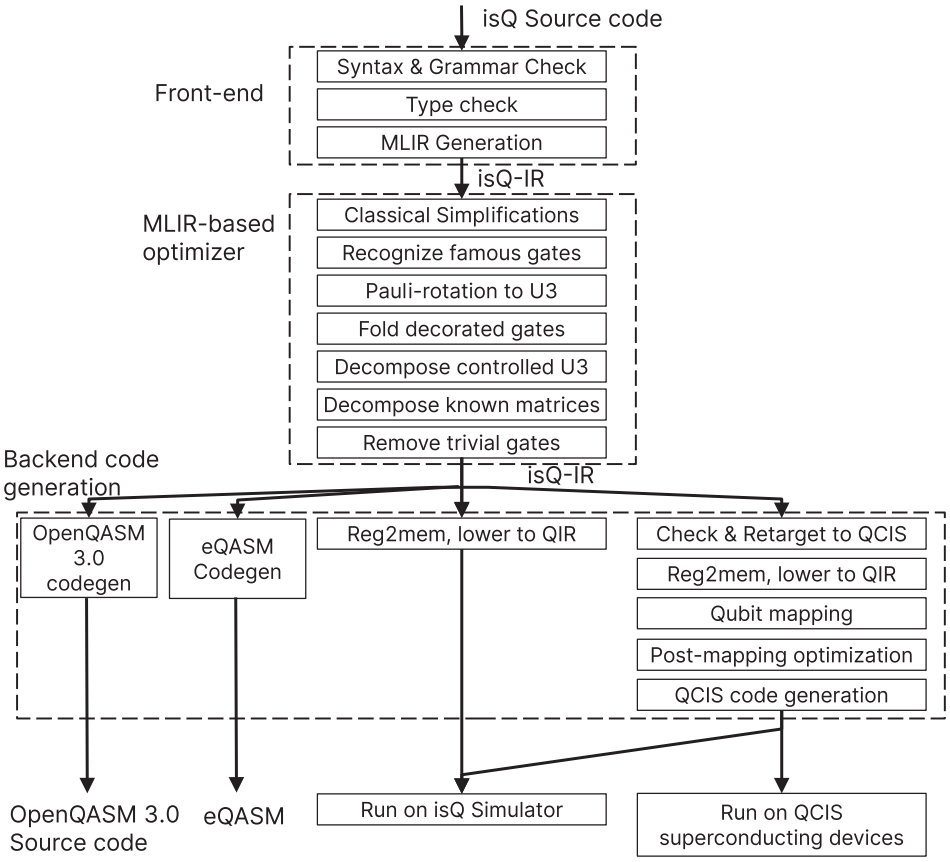

FIGURE 1. Structure of isQ compilation stack. The compiler first compiles the input programs into isQ-IR and then generates the target codes, such as QCIS, OpenQASM 3.0, and QIR. Optimizations are permitted throughout the whole compilation process. Finally, the output QCIS and QIR codes could be executed on corresponding hardware and simulators.

1) We propose an imperative quantum programming language named isQ.
2) We propose isQ-IR, an IR for quantum programs, leveraging MLIR [8] dialect infrastructure and representing qubit dataflow in static-single-assignment (SSA) form, thus allowing flexible transformations of quantum programs.
3) Around isQ-IR, we built a versatile software stack: we implement a compiler that can compile isQ programs into isQ-IR; then, we leverage the MLIR framework to transform and optimize quantum programs in isQ-IR, and lower isQ-IR into multiple backends including QIR, OpenQASM 3, and real-device QCIS.
    a) isQ programs can be easily lowered to QIR. Moreover, for debugging purposes, we implemented our own QIR simulator that can execute QIR programs compiled by the isQ compiler. The simulator provides an interface for supporting different backends, including built-in CPU and compute unified device architecture (CUDA) backends.
    b) isQ programs without feedback control can be compiled to QCIS assembly, which can be executed on the superconducting quantum hardware at the University of Science and Technology of China (USTC).
    c) isQ-IR can also be compiled into OpenQASM 3.0, while most high-level structures in isQ can be preserved and converted to OpenQASM 3.0 control flow.

The source code of the isQ project is in [13]. A simple help document with a download link to an executable of isQ could also be found in [14].

II. RELATED WORKS

A. QUANTUM PROGRAMMING LANGUAGES AND COMPILERS

In [15] and [16], the theoretical foundations of quantum programming are introduced. Meanwhile, a series of works in quantum programming languages and compilers [17], [18], [19], [20], [21], [22], [23], [24], [25] targeting different levels of abstraction are proposed. For example, Qiskit [18], proposed by IBM, is a well-established and popular Pythonic circuit construction framework for OpenQASM 2 [26] and OpenQASM 3 [11], which allows classical control flow and can run on IBM's hardware backends. There are also languages such as Q# [19] and Silq [20] that support powerful language features. Q# has its independent compiler, and compiled quantum programs can be executed on real hardware or the simulator while Silq programs can only run on a simulator.

A closely related work proposed previously is QCOR [23], which also aims at building a practical and versatile quantum programming software stack. QCOR modified Clang so that quantum kernels (written in various quantum programming languages) could be interleaved with C++ source code, allowing existing C++ libraries to be imported out of convenience for quantum programmers, e.g., the gradient descent methods in variational quantum algorithms. By proposing its own compiler framework, XACC [27], QCOR can compile one quantum program into multiple backends.

While many of these works provide abundant optimizations at the quantum circuit level, one of the most distinctive features of the isQ software stack is its compiler optimizations on high-level program structures. These optimizations could be accomplished without unrolling the programs into static quantum circuits. In particular, a quantum loop structure is considered, which is elaborated in Section V-C.

B. QUANTUM IRs

Many quantum compilers employed their own IR to transform and optimize quantum programs. For example, XACC defines a general IR for representing quantum programs, allowing compiling from various quantum programming languages to various backends. ScaffCC [28] extended LLVM IR to represent quantum operations to leverage the LLVM framework for analyzing quantum programs, as well as relying on LLVM to enable fast code generation (called instrumentation-driven approach). QIR [9], one compilation target of our toolchain, is another LLVM-based IR that introduces quantum functionality by using LLVMs opaque struct mechanism. Quantum MLIR dialect [29] is proposed to use MLIR to fill the gap between quantum languages and QIR.

These IRs usually choose to model quantum operations as "opaque operations." While this allows an easy IR definition and implementation, this IR form obscures the quantum dataflow between gates in the program, adds to difficulties in tracking and recording qubits over quantum gates, and, therefore, adds to difficulties in performing optimizations on the program.

There have also been several works using SSA for representing quantum programs. QIRO [30] is proposed as an MLIR dialect that allows exposure of quantum dataflow as use-def chains, thus allowing quantum dataflow analysis and optimizations. QSSA [31] further proposed a single-use analysis to statically check if the no-cloning theorem is obeyed in the program. Both IRs share some commonalities: For example, they defined a handful of native gates as primitive operations; instead of using MLIRs memref dialect designed for representing memory, they both invented their own extract-merge styled array SSA [32] for representing multiqubit registers.

Compared with these MLIR-based SSA IRs, isQ-IR is designed with extensibility and MLIR interoperability in mind. To achieve extensibility, for example, no built-in gates are defined by isQ-IR; instead, isQ-IR introduces a unified way of defining and using gates, applying modifiers to defined gates, and marking gates with special properties, thus allowing generic transformation passes based on gate properties to be easily implemented. For interoperability, isQ-IR complies with MLIRs philosophy of modular dialect definitions by reusing MLIR built-in dialects and passes when possible instead of reinventing the wheel; this allows our compiler to benefit from MLIRs powerful built-in analysis to optimize quantum programs.

C. REAL-TIME CLASSICAL CONTROL VERSUS HYBRID CLASSICAL-QUANTUM COMPUTING

Many quantum programming languages and software support combining the power of classical and quantum computing. There are roughly two types of cooperation between the classical and quantum parts: 1) hybrid classical-quantum computing and 2) real-time classical control.[1]

Hybrid classical-quantum computing executes by performing quantum computation and classical computation alternatively. This is usually done on a classical computer connected to a quantum device. The classical computer first uploads a simple quantum kernel to the quantum device, waits for the quantum device to finish executing the kernel, and collects its execution results. The classical computer then performs some classical computation and uploads the next quantum kernel. Note that between two executions of the quantum kernel, the quantum device is reset, and the state of qubits cannot be preserved between the two executions. Examples of such hybrid quantum algorithms in the NISQ era include VQE [33] and QAOA [34].

Real-time classical control allows classical computation to be interleaved with quantum computation within one decoherence time. This often comes with support for intermediate measurements (i.e., measuring some qubits in the middle of the program) and classical control flow constructs like branching statements, loops, and recursions. Since the decoherence time of near-term quantum devices is very short, the real-time classical computation is usually carried out on the near-qubit microcontroller, usually implemented on a field programmable gate array (FPGA) chip [10], [35], rather than on an external CPU. Allowing real-time classical control flows could help simplify the design of some quantum algorithms, e.g., reducing the number of ancilla quantum qubits [36], [37] used, and reducing the circuit depth such as preparing a cat state [38]. Moreover, this kind of real-time feedback control is indispensable for implementing quantum error-correcting codes [39], [40].

Many quantum programming languages and toolchains support hybrid classical-quantum programs [18], [23], [25], [41], [42], [43], usually by adding support for constructing and launching quantum kernels [e.g., OpenQASM or embedded domain-specific language (DSL) in Python] in a commonly-used programming language (e.g., C++ or

[1] In [11], these two types of classical computations are also referred to as near-time classical computing and real-time classical computing.

Python). Recently, with the progress of quantum control hardware, there have been quantum programming languages also providing support for real-time classical control [11], [18], [22]. Our isQ software stack supports real-time classical control in isQ language and supports near-time hybrid computing by providing Pythonic wrappers for interaction between quantum kernels and classical computers. See the work in [14] for an example of computing the ground state energy of a hydrogen molecule.

III. ISQ PROGRAMMING LANGUAGE

We designed isQ as a simple yet powerful imperative language for writing quantum programs. isQ assumes that the program is run on a classical processor with real-time access to qubits and real-time measurement feedback control; if the program does not use any feedback control, isQ can also be compiled into flat quantum circuits.

Example III.1: We first give a simple program written in isQ simulating quantum teleportation of one qubit, showing the most basic features and the style of isQ language. More examples are given in Section V.

```
procedure create_bell(qbit p[2]){
    H(p[0]);
    CNOT(p[0], p[1]);
}
procedure teleport(qbit x, qbit p[2]){
    CNOT(x, p[0]);
    H(p[0]);
    // Measurements.
    if(M(p[0])){
        X(p[1]);
    }

    if(M(x)){
        Z(p[1]);
    }
}
procedure main(){
    qbit x;
    qbit p[2];
    H(x);
    create_bell(p);
    teleport(x, p);
    print M(p[1]);
}
```

A. CORE CONSTRUCTS

We introduce some basic features of isQ for quantum-classical hybrid programming.

1) QUANTUM COMPUTING CAPABILITY

isQ provides basic support for computing using qubits: qubit allocation, quantum gates, and measurements. isQ allows allocating qubits in local scopes, which will be automatically traced out when these qubits go out of the scope; a handful of standard gates, both fixed and parametric, as well as computational basis measurement, are given as built-in operations.

2) CLASSICAL COMPUTATION AND CLASSICAL CONTROL

isQ provides support for real-time classical computation during the execution of quantum programs, including integer arithmetic and float-point arithmetic ope. isQ also provides structured classical control flow constructs support including *if*-statements, *for*- and *while*-loops, and *procedure* calls. Moreover, isQ allows parametric gates to accept floating-point values, a powerful type of classical control on quantum devices.

B. NOTABLE LANGUAGE FEATURES

Next, we describe several notable high-level features of isQ in detail, which are as follows.

1) USER-DEFINED GATES

isQ introduces two ways for defining new quantum gates by users: 1) by specifying a unitary matrix and 2) by specifying a procedure.

Gate definition by unitary matrix: isQ allows developers to define new gates by directly specifying the unitary matrix of the gate. The unitarity of the matrix is checked on the compilation of the program.

Gate definition by procedure: isQ allows user-defined gates by adding "*deriving gate*" notation to a procedure definition. However, this leads to the problem that an arbitrary procedure may not describe a unitary quantum gate, e.g., the procedure contains measurements. Moreover, we want to support defining *parametric* quantum gates by specifying a procedure with classical parameters.

We address the following constraints for a procedure to be used as a gate definition.

1) The parameters of the procedure must start with zero or more classical (i.e., Boolean, floating, and integer) parameters and then zero or more *qbit* parameters. No arrays are allowed. Thus, all user-defined gates are fixed-sized.
2) The procedure should have no return value.
3) If an adjoint version of the gate is used, no classical control flow statements should appear in the procedure body.
4) No measurement or side effects are allowed in the procedure body.
5) While ancilla qubit allocation is allowed, the programmer should guarantee that all ancilla qubits are reset to $|0\rangle$ when released.

Example III.2: We hereby define a U_3 gate using a procedure that accepts three angles and one qubit

$$U_3(\theta, \phi, \lambda) = e^{i(\phi+\lambda)/2} R_Z(\phi) R_Y(\theta) R_Z(\lambda) \tag{1}$$

$$= \begin{pmatrix} \cos\frac{\theta}{2} & -e^{i\lambda}\sin\frac{\theta}{2} \\ e^{i\phi}\sin\frac{\theta}{2} & e^{i(\phi+\lambda)}\cos\frac{\theta}{2} \end{pmatrix}. \tag{2}$$

```
procedure my_U3(double theta, double phi, double
     lam, qbit q){
    GPhase((phi+lambda)/2);
    Rz(lam, q);
    Ry(theta, q);
    Rz(phi, q);
} deriving gate
```

2) DECORATED GATES AND GATE DERIVING NOTATION

Controlled and adjoint quantum gates have been proven useful in constructing quantum circuits. isQ supports using quantum gates in their *decorated* form, i.e., in (multi)controlled or adjoint (inverse) version as follows.

1) "**ctrl**⟨N⟩" can be added before gate calling to add N controller bits to the gate. ⟨N⟩ can be omitted for $N = 1$.
2) "**nctrl**⟨N⟩" can be added to add N negated-controller bits to the gate, i.e., appending X gates before and after controller bits. ⟨N⟩ can be omitted for $N = 1$.
3) "**inv**" can be added to use the adjoint (inverse) version of the gate.

When decorated forms of user-defined gates are used, isQ will automatically generate controlled and adjoint versions of these gates. For controlled gates, new qubit parameters are inserted after the classical parameters and before the original *qbit* parameters.

Example III.3: For the example above, if we want to perform a controlled U_3 gate on two qubits a and b, we need to write

```
ctrl my_U3(lam, theta, phi, a, b);
```

In this example, a is the control qubit, and b is the target qubit.

3) ORACLE SUPPORT

Quantum oracles are important constructs in quantum algorithms like Grover search [44] or recursive Fourier sampling (RFS) [45]. Currently, isQ language permits oracle definition by directly writing out the truth table of oracle functions with type $f : \{0, 1\}^m \to \{0, 1\}^n$, or using a simple expression containing some input variable(s).

Example III.4: For Boolean function $f(x, y) = x \wedge \neg y$, we may define a quantum oracle computing the function as

$$F\,|a_0 a_1 b\rangle = |a_0 a_1\rangle\,|b \oplus (a_0 \wedge \neg a_1)\rangle\,. \tag{3}$$

isQ allows defining the oracle gate in different ways. After defining the oracle, it can be used directly as a quantum gate

```
// F and G are equivalent oracles.
oracle F(2,1) = [0,0,1,0];
oracle bool[1] G(bool a[2]) {
    bool res[] = {a[1] && !a[0]};
    return res;
}

// Usage:
qbit a[2], b, c[1];
F(a[0], a[1], b); // apply to qbits
G(a, c);          // apply to qbit arrays
```

4) BUNDLE OPERATION

isQ provides a compact way named *bundle operation* for expressing parallel quantum operations. If a quantum gate or measurement is applied to all or a slice of the qubits in an array, the array or the slice can be used as the arguments of the functions. To represent a slice of an array, we adopt a syntax similar to the one used in Python. A slice contains three parts that represent the start index, end index (excluded), and incremental step, respectively, separated by two colons (:). These parts can be omitted, and default values would be applied.

Example III.5: The following code example shows the usage of bundle operations:

```
qbit p[3], q[6];
H(p);      // = H(p[0]); H(p[1]); H(p[2]);
M(q[2:5]); // = M(q[2]); M(q[3]); M(q[4]);
CNOT(p, q[::3]); // = CNOT(p[0], q[0]);
                 //   CNOT(p[1], q[3]);
```

Note that the bundle operation can also be used in multiqubit gates, e.g., CNOT. In that case, the number of actual operations is determined by the *shortest* array or slice, i.e., q[:: 3], in the previous case.

IV. isQ COMPILER ARCHITECTURE

In this section, we describe the design of our compiler that provides fundamental support for features of the isQ language. The compiler is based on MLIR [8] infrastructure, a highly extensible compiler framework that supports representation, transformation, and code generation for domain-specific computing, e.g., neural network and circuit logic. Specifically, for our purpose, the following statements hold.

1) We defined isQ-IR, an MLIR dialect supporting certain quantum operations, as our IR for representing and transforming quantum programs. Our isQ frontend generates isQ-IR directly.
2) Based on MLIR's powerful general IR transformation framework, we utilized both existing general transformation passes provided by MLIR, as well as our own quantum-specific transformation passes, to transform and optimize quantum programs. In particular, many optimizations on high-level program structures are implemented.
3) By utilizing MLIRs code generation and lowering infrastructure, we are able to generate different types of output code, including both high-level representations like QIR and OpenQASM 3.0 [11] and low-level real-device instruction sets like QCIS.

A. DIALECT DEFINITION OF ISQ-IR

The principles for defining our isQ-IR dialect include the following.

1) Allowing easy reuse of compilation infrastructures provided by MLIR, including compilation infrastructures originally designed for transforming classical programs. We designed our dialect in integration with existing MLIR dialects (**scf**, **affine**, **memref**, etc.) and only added a minimum number of required operations enough for quantum programs while preserving the necessary properties for quantum-agnostic transformation passes to transform the IR correctly.

2) Facilitating gate-level optimizations. We adopted an SSA form of representing qubit dataflow, which allows local quantum-circuit fragments to be exposed as def-use chains by a memory dependency analysis. MLIRs powerful rewrite framework also simplifies the definition of gate identities and transformations.
3) Preserving high-level program structures. isQ-IR allows high-level program information, e.g., decorated gates, built-in gate optimization hints, and gate definition by matrices and oracles, to be represented. We can perform high-level optimizations (e.g., canceling out user-defined $UU^\dagger$), lower the high-level operations (e.g., inlining and gate-decomposition), perform low-level optimizations, and finally generate low-level code.
4) Allowing extension for future features. Our dialect is designed to be extensible so that we can add more features easily, e.g., new compilation passes and new ways of defining gates.

1) NEW MLIR TYPES

isQ-IR represents quantum dataflow in SSA form. isQ-IR defined two new types for describing quantum programs: **!isq.qstate** and **!isq.gate**⟨N, *hints*⟩.

!isq.qstate (referred to as *qstate* for short) represents an intermediate state (which may be entangled with other qubits) of a single qubit, which can be seen as an "open wire" in a quantum circuit fragment. We do not define new types for representing qubits or qubit arrays; instead, we model them using **memref**⟨$n \times$ **!isq.qstate**⟩. Qubit allocation/deallocation is represented by **memref.alloc** and **memref.free**; for quantum operations, we need to use **memref.load** operation (or **affine.load**) to extract the qstate out, perform operations to obtain a new value, and use **memref.store** (or **affine.store**) to store it back.

Using quantum SSA representation poses additional requirements for legality, which are as follows.

1) Every qstate must be stored in the exact memory location where it was loaded from. This is guaranteed by our input IR.
2) Two qstates that are both "alive" at a certain point of a program must belong to two different qubits. They should be seen as distinct "open wires" in a quantum circuit, on which we can safely perform multiqubit gates or freely switch gates on disjoint qubits.
3) Passes should not introduce qstates that are not finally stored back to its **memref** or store an invalid qstate back. This condition ensures that there are no extra unused **!isq.apply** statements. To meet this requirement, our gate rewrite passes carefully remove redundant SSA values and the store operations that are associated with them.

!isq.gate⟨N, **hints**⟩ represents a pure quantum N-qubit gate as an SSA value that can be decorated, applied to qubits, and passed around. **hints** in the gate type describes special properties about the gate. Currently, supported hints include the following.

1) *hermitian*, indicating that the gate is Hermitian, e.g., CNOT, H.
2) *diagonal*, indicating that the gate has a diagonal matrix form, e.g., CZ, R_Z.
3) *antidiagonal*, indicating that the single-qubit gate is antidiagonal, e.g., X.
4) *symmetric*, indicating that the order of gate operands does not matter, e.g., *SWAP* and *CZ* gates.
5) *phase*, indicating the gate is in the form $U_n = \sum_{0 \leqslant i < 2^n - 1} |i\rangle \langle i| + e^{i\theta} |111\cdots1\rangle \langle 111\cdots1|$. All *phase* gates are naturally diagonal and symmetric.

2) NEW OPERATIONS

Table 1 lists isQ-IR-defined MLIR operations. Instead of defining a built-in basic gate set in the IR, all gates are defined through **isq.defgate**. Both nonparametric gate and parametric gate families can be defined. The operation is attached with an attribute "*definition*," allowing specifying multiple ways of defining the operation, as follows:

1) "qir," definition by a QIR [9] function;
2) "unitary," specifying a unitary matrix definition;
3) "oracle," specifying an oracle truth-table definition;
4) "decomposition," specifying gate decomposition by a **builtin.func**.

A **builtin.func** can be used to describe a decomposition of a gate only if the following holds.

1) The function accepts gate parameters and n qstates as arguments exactly.
2) The function returns an n-qstate tuple exactly.
3) Returned qstates correspond to argument qstates in exact order.
4) There are no external quantum operation calls (e.g., measurements) in the function body.
5) For autogeneration of the adjoint version, the function should only consist of one basic block and contains no operations with subregions.

Note that these requirements correspond with our requirements for gate-defined procedures in the isQ language. These requirements allow us to automatically generate a controlled (and adjoint) version of a gate.

A defined gate can be referenced using **isq.use** operation. If the gate has classical parameters, they must be specified as operands of **isq.use**. A gate can then be applied to qstates using an **isq.apply** operation, returning new SSA qstate values that can be applied on by the next gates or stored back.

TABLE 1. New MLIR Operations Defined by isQ-IR

Operation	Usage	Example
isq.defgate	Global definition of a gate that can be used.	isq.defgate @X {definition = #X_DEF} : !isq.gate<1, hermitian>
isq.use	Use a globally defined gate as a value.	%X = isq.use @X : !isq.gate<1, hermitian>
isq.apply	Apply a gate value onto qubit states.	%b = isq.apply %X(%a) : !isq.gate<1, hermitian>
isq.decorate	Extend control qubits onto a gate or invert a gate.	%CNOT = isq.decorate (%X: !isq.gate<1, hermitian>) {ctrl = [true], adjoint = true} : !isq.gate<2>
isq.downgrade	Auxiliary operation for removing hints.	%H2 = isq.downgrade (%H: !isq.gate<1, hermitian>) : !isq.gate<1>
isq.declare_qop	Define an external quantum operation.	isq.declare_qop @measure : [1]()->i1
isq.call_qop	Calls an external quantum operation.	%b, %outcome = isq.call_qop @measure(%a): [1]()->i1
isq.apply_gphase	Applying global phase.	isq.apply_gphase %g : !isq.gate<0>
isq.accumulate_gphase	Auxiliary operation for "moving" global phase out of local system.	isq.accumulate_gphase %q : memref<1x!isq.qstate>

isq.decorate is added to represent decorated (controlled and adjoint) operations. An **isq.decorate** operation is attached with two attributes, *ctrl* and *adjoint*. The *ctrl* attribute is a list of Boolean values, where "false" indicates the corresponding controller bit is negated. The operand of **isq.decorate** is an **!isq.gate**, and the result is also an **!isq.gate** with correct size and hints as follows.

1) Adding controller bits increases the gate size.
2) Adjoint version of the Hermitian gate is equivalent to an nonadjoint version.
3) Adjoint version of *diagonal* is still diagonal. The same is true for *antidiagonal* and *symmetric*.
4) Controlled diagonal gates are still diagonal gates. The same is true for positively-controlled phase gates.

isq.declare_qop represents external quantum operations other than pure gates (e.g., measurements) that can be called by **isq.call_qop** that operates on zero or more qubits and zero or more classical inputs, interacts with the external environment, and return zero or more classical output.

We use signature $[n](input) \to output$ to represent a quantum operation that works on n qstates and classical values $input$, returning n qstates and classical values $output$. For example, (computational basis) measurement is an operation that operates on one qubit and yields an **i1** value and, thus, has signature $[1]() \to i1$. This is equivalent to traditional function signature $(!isq.qstate) \to (!isq.qstate, i1)$. The signature for qubit reset is $[1]() \to ()$, and for printing integer $[0](index) \to ()$.

Several auxiliary operations are defined as well. **isq.downgrade** removes hints from a gate's type signature so that the gate fits into other operations that require a different yet compatible type of gate as arguments. **isq.apply_gphase** and **isq.accumulate_gphase** are defined as side-effective for applying global phase. While global phases have no effect, generating controlled versions of user-defined gates requires the global phase to be preserved.

Example IV.1: Consider the following function definition. The gates on ancilla qubit $\%Q$ introduce global phase (-1) to the system and can be optimized out. However, if we try to generate a controlled version of this function, eliminating them early will result in an error in a relative phase.

```
func @error_1(%q0: !isq.qstate)->!isq.qstate {
    %Q = memref.alloc() : memref<1x!isq.qstate>
    %q = affine.load %Q[%0]: memref<1x!isq.qstate
        >
    %X = isq.use @X : !isq.gate<1>

    %Z = isq.use @Z : !isq.gate<1>
    %q2 = isq.apply %X(%q) : !isq.gate<1>
    %q3 = isq.apply %Z(%q2) : !isq.gate<1>
    %q4 = isq.apply %X(%q3) : !isq.gate<1>
    affine.store %q4, %Q[0]: memref<1x!isq.qstate
        >
    memref.dealloc(%Q) : memref<1x!isq.qstate>
    return %q0: !isq.qstate
}
```

isq.apply_gphase accepts an **!isq.gate**$\langle 0 \rangle$, indicating applying the specified global phase to the system. When being "controlled," they will be converted to **isq.apply** operations. **isq.accumulate_gphase** operates on **memref**$\langle ? \times$ **!isq.qstate**$\rangle$ so that the corresponding local qubits are considered "alive" and will not be optimized out.

Global phase operations can be simply removed after generating all controlled versions of gates. In the example above, in the function for the original gate, we can safely remove $\%Q$ and corresponding gates while in the function for the controlled-gate $\%Q$ will be preserved.

B. TRANSFORMATIONS ON ISQ-IR

1) REUSED CLASSICAL TRANSFORMATION PASSES

Our definition of isQ-IR allows for multiple transformation passes provided by MLIR to be reused without modification.

1) *Canonicalization, CSE, Symbol-DCE:* these are useful SSA transformations that could be used to simplify code and remove unnecessary computation.
2) *Memref subview folding:* recognized as "fold-memref-subview-ops" in MLIR. This pass removes probable redundant **memref.subview** ops generated by our front end, thus removing unnecessary potential memory aliasing and simplifying store-load forwarding.

3) *Store-load forwarding:* recognized as "affine-scalrep" in MLIR. This pass can be used to forward **affine.store**-d SSA values to following **affine.load**, thus exposing the use-def chain that we need for gate cancellation.
4) *Loop unrolling:* recognized as "affine-loop-unroll" in MLIR. This pass unrolls loops in the program and is useful for canceling more gates when targeting real devices.
5) *MLIR built-in lowering passes:* These are useful passes for lowering other MLIR dialects to QIR.

2) GATE-LEVEL TRANSFORMATION PASSES

Leveraging MLIRs rewrite framework, we implemented a handful of transformations that decompose and transform the quantum part of the program.

Canonicalization: Several useful local peephole optimization patterns are added to MLIRs canonicalizer, executed after every transformation pass.

1) *Decorate-op folding:* A rewrite pattern that folds two consecutive **isq.decorate** operations into one.
2) *Gate cancellation:* Rewrite pattern that cancels out pairs of Hermitian gates and $UU^\dagger$ gate pairs.
3) *Symmetric operand rearranging:* When all inputs of a *symmetric* gate are from outputs of one gate, reorder the operands to match the output order of the previous gate. This is useful for canceling two CZ gates in "reverse" directions.
4) *CZ cancellation by commutation:* Two CZ gates on the same pair of qubits, if there are no gates or only diagonal gates between them, can be canceled out.

Recognize famous gates: This pass inserts the definition of isQ builtin gates, including Paulis, Pauli rotations, U_3, CNOT, and $Toffoli$.

Pauli-rotation to U_3: This pass converts all parametric Pauli rotations into parametric U_3 gates.

Fold decorate gates: This pass folds all **isq.decorate** operations, which are as follows.

1) Negated controller bits are eliminated by inserting X gates.
2) For matrix-defined gates, a new matrix definition of the decorated gate is generated.
3) For decomposition-defined gates, the **builtin.func** is cloned and modified by adding controller bits and/or inverting the gate sequence.

After this pass, global phase auxiliary operations can be removed.

Decompose controlled U_3: This pass decomposes controlled, parametric U_3 gate using $AXBXC$ rule [46], resulting in controlled-X, controlled-$GPhase$ (controlled phase shift), and R_Z and R_Y operations. Controlled-X and controlled-$GPhase$ are decomposed recursively according to the work in [46]. This pass eliminates all controlled parametric gates.

Decompose known matrices: This pass decomposes unitary-matrix-defined gates into a bunch of basic gates using quantum Shannon decomposition [47]. Till now, all gates are basic gates.

Remove trivial gates: This pass removes constant gates that are very close to the identity gate.

Oracle synthesis: This pass parses the defined oracles and converts them into reversible circuits. To enhance the performance, we adopt algorithms such as the Quine-McCluskey algorithm and implement some optimizations like the reuse of dirty ancilla qubits.

3) HIGH-LEVEL PROGRAM STRUCTURE TRANSFORMATION PASSES

Moreover, some transformation passes on high-level program structures are implemented within isQ-IR. These transformation passes directly optimize high-level program structures without unrolling them into quantum circuits.

Example IV.2: Consider the following two lines of isQ codes:

```
H (q[a]);
H (q[b]);
```

Here, q is a qubit array, and a, b are two classical variables. We cannot directly cancel out these two Hadamard gates since the values of a and b are uncertain until runtime. Therefore, the compiler does not know beforehand whether they can be executed in parallel. However, considering that quantum operations are usually more expensive than classical operations, the codes could be converted into the following form with the semantics unchanged:

```
if (a == b){
    // do nothing;
} else {
    H (q[a,b]); //They can definitely be
    executed in parallel
}
```

This simple example above shows that optimizations on high-level program structures may sometimes bring benefit. Within isQ-IR, we implement some characteristic optimizations targeted at structures like for-loop and recursion, which are elaborated in Sections V-C and V-D.

C. ISQ-IR TO DIFFERENT LOW-LEVEL IRs

1) LOWER TO QIR

After these passes, we defined lowering passes to convert isQ-IR to QIR.

Expand decomposition: Replace all gates defined by decomposition with **builtin.call** to decomposition function or QIR stub.

Reg2mem: Finally, we convert qstate to auxiliary type **!isq.qir.qubit** and gate applications to QIR calls. Measurements, resets, and print operations, represented using **isq.call_qop**, are converted to corresponding QIR functions.

Lower to LLVM: lower auxiliary type **!isq.qir.qubit** to **llvm.struct**⟨"**Qubit**", **opaque**⟩, the corresponding type in QIR. Applying lowering rules of other dialects (*affine*, *scf*, etc.) results in legal QIR.

2) CODEGEN TO QCIS

Our isQ compiler supports generating QCIS assembly that can execute on superconducting hardware. Challenges for targeting QCIS superconducting hardware include the following.

1) QCIS hardware uses a different instruction set: for single-qubit parametric gates, QCIS supports R_Z; for two-qubit gates, QCIS provides *CZ* instead of CNOT. Therefore, retargeting gates in the program are required.
2) QCIS hardware does not support classical control flow or feedback control. The "quantum" part of our program must be extracted and flattened.
3) Superconducting hardware has qubit connectivity constraints. Qubit mapping is required to map logical qubits in the isQ program onto real hardware qubits.

To solve these problems, we proposed new passes to convert an isQ program into a valid QCIS assembly that can execute on real devices.

1) We first check input isQ-IR to make sure there are no reset statements or uses of measurement results. This prevents feedback control in the program.
2) We retarget gates in the program into QCIS instruction set by converting CNOT into *H-CZ-H* triples and U_3 into *RZ*, *X2P*, *X2M* [12], [18].
3) We extract all quantum gates by executing the program on our simulator and collect every gate call. This effectively flattens high-level control flow structures in the program (loops and conditional branches) into a gate list.
4) We perform qubit mapping on the collected gate list by our qubit mapper based on the work in [48], generating QCIS assembly executable on superconducting hardware.

3) CODEGEN TO OPENQASM 3.0

We implemented a direct code generator from isQ-IR to logical-level OpenQASM 3.0, a high-level quantum assembly language with control flow support. Since the control flow primitives in OpenQASM3 are high-level ones (e.g., if-statement, while-statements) instead of low-level ones (e.g., goto statements, basic blocks), we map structured MLIR control-flow operations directly to OpenQASM3, e.g., **scf.if** onto if-statements, **affine.for** onto for-statements.

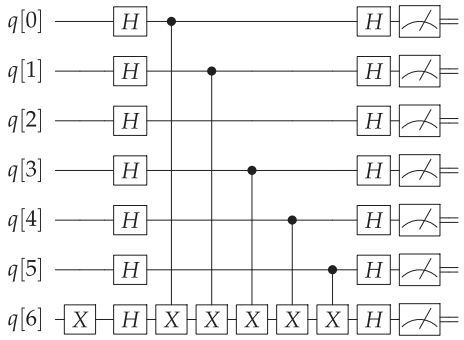

FIGURE 2. 7-qubit Bernstein–Vazirani circuit with the secret string as 110111.

V. EXAMPLES

In this section, we present several illustrative implementations of some interesting quantum algorithms that can clearly manifest the main features of isQ.

A. PURE QUANTUM PROGRAM

We use a simple example, Fourier sampling, also called the Bernstein–Vazirani algorithm [49], to show the main workflow of isQ software stack, from isQ language to hardware-supported QCIS assembly.

Example V.1 (Bernstein–Vazirani algorithm): Assume a function generates a Boolean value by

$$f(x) = s \cdot x \pmod 2$$

where s is a "secret string" and the dot $(\cdot)$ represents bitwise product sum. Bernstei–Vazirani algorithm computes s by applying f once. We first give the implementation of a small-scale Bernstein–Vazirani algorithm on seven qubits, with $s = 110111$.

```
procedure main(){
    qbit q[7];
    X(q[6]);
    H(q);

    // Implements f(x) = 110111 * x
    CNOT(q[0], q[6]);
    CNOT(q[1], q[6]);
    CNOT(q[3], q[6]);
    CNOT(q[4], q[6]);
    CNOT(q[5], q[6]);

    H(q);
    M(q); // The first six bits should be 110111
}
```

The above isQ program represents a 7-qubit circuit shown in Fig. 2. Note that *H* gates and measurement are bundle operations that are applied to all the qubits. After canonicalization and constant folding, the program above can be compiled to the following isQ-IR:

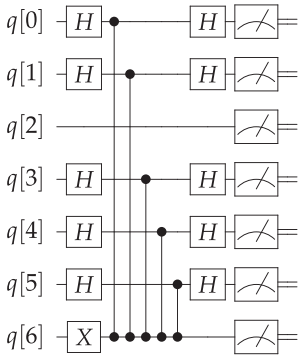

FIGURE 3. Optimized circuit whose original form is shown in Fig. 2.

```
func @__isq__main() {
    %0 = memref.alloc() : memref<7x!isq.qstate>
    %X = isq.use @X : !isq.gate<1>
    %2 = affine.load %0[6] : memref<7x!isq.qstate
        >
    %3 = isq.apply %X(%2) : !isq.gate<1>
    affine.store %3, %0[6] : memref<7x!isq.qstate
        >
    affine.for %arg0 = 0 to 7 {
      %H = isq.use @H : !isq.gate<1>
      %15 = affine.load %0[%arg0] : memref<7x!isq
          .qstate>
      %16 = isq.apply %H(%15) : !isq.gate<1>
      affine.store %16, %0[%arg0] : memref<7x!isq
          .qstate>
    }
    /* omitted */
    memref.dealloc %0 : memref<7x!isq.qstate>
    return
}
```

First, we perform loop unrolling to obtain unrolled loop kernel

```
%2 = affine.load %0[6] : memref<7x!isq.qstate>
%3 = isq.apply %X(%2) : !isq.gate<1> // X(q[6])
affine.store %3, %0[6] : memref<7x!isq.qstate>
%5 = affine.load %0[0] : memref<7x!isq.qstate>
%6 = isq.apply %H(%5) : !isq.gate<1> // H(q[0])
affine.store %6, %0[0] : memref<7x!isq.qstate>
/* omitted */
%17 = affine.load %0[6] : memref<7x!isq.qstate>
%18 = isq.apply %H(%17) : !isq.gate<1> // H(q[6])
affine.store %18, %0[6] : memref<7x!isq.qstate>
```

Applying MLIR-builtin load-store forwarding pass is able to recover the dataflow on $q[6]$ and other qubits

```
%2 = affine.load %0[6] : memref<7x!isq.qstate>
%3 = isq.apply %X(%2) : !isq.gate<1> // X(q[6])
%17 = isq.apply %H(%3) : !isq.gate<1> // H(q[6])
```

Now qubit data dependency is exposed by the use-def chain upon which our quantum passes can take effects: A conversion pass converts CNOT into CZ and H supported by QCIS while the canonicalizer eliminates consecutive H gate pairs on $q[2]$, effectively eliminating the qubit. The circuit form of the program at this point is given in Fig. 3.

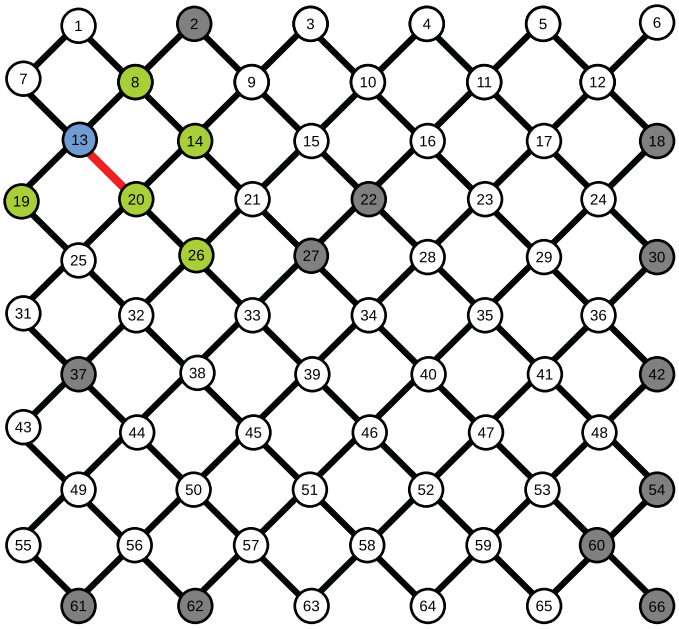

FIGURE 4. Qubit topology of a 56-qubit processor in our experiments. Gray nodes represent unavailable qubits on the device. Green and blue qubits are used qubits. Red edge indicates a SWAP gate is inserted here during qubit mapping.

To extract the flattened circuit, we first lower isQ-IR to auxiliary QIR form

```
%0 = memref.alloc() : memref<7x!isq.qir.qubit>
scf.for %arg0 = 0 to 7 step 1 {
  %5 = call @__quantum__rt__qubit_allocate() : ()
       -> !isq.qir.qubit
  memref.store %5, %0[%arg0] : memref<7x!isq.qir.
      qubit>
}
%1 = memref.load %0[6] : memref<7x!isq.qir.qubit>
call @__quantum__qis__x__body(%1) : (!isq.qir.
    qubit) -> ()
%2 = memref.load %0[0] : memref<7x!isq.qir.qubit>
call @__quantum__qis__h__body(%2) : (!isq.qir.
    qubit) -> ()
```

The snippet will be further lowered to QIR, compiled by LLVM, and linked with our simulator. We implemented a special backend in the simulator that converts to code-generation calls, e.g., *__quantum__qis__x__body*(i) generating a "X Q_i" QCIS instructions. These instructions will be collected and mapped to real hardware using our qubit mapping pass. In the pass, we adopt an algorithm in [48] that uses simulated annealing to find a good initial mapping scheme and then uses heuristic search to schedule operations, i.e., insert swap gates. For demonstration, we choose a 56-qubit superconducting quantum processor. The qubit topology of this processor and the chosen qubits by the mapper are shown in Fig. 4. The circuit immediate after the mapping pass is given in Fig. 5.

After the qubit mapping pass, we re-import the circuit as isQ-IR to perform retargeting (to eliminate the *SWAP* gate),

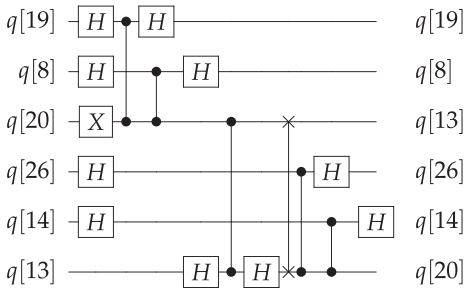

FIGURE 5. Circuit representation after qubit mapping.

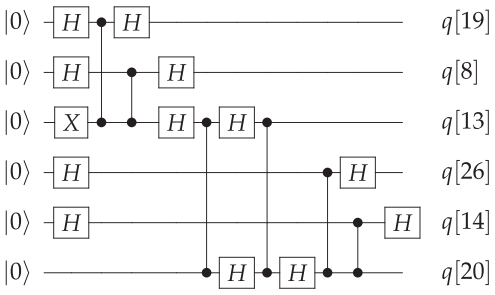

FIGURE 6. Final program after postmapping optimization.

canonicalization (to cancel Hadamard and *CZ* gates from the *SWAP* gate), lowering to QIR and QCIS code generation, to get the final runnable QCIS program.

Finally, to illustrate the effect of these optimizations, we conducted real-hardware experiments on the processor depicted in Fig. 4. We ran the circuit in Fig. 6, comprising 19 gates. For comparison, we ran another circuit generated by running the circuit in Fig. 2 on the processor, using the same qubit mapping scheme but without any optimizations. This generated circuit comprises 39 gates. The results are shown in Fig. 7. Both circuits were executed for 12 000 shots. For the optimized circuit, 5687 shots return the correct results, i.e., the success rate is 47.39%; in contrast, the other circuit only has 1103 correct shots, i.e., the success rate is 9.19%. These experiments show that our optimizations bring noticeable improvement.

B. PROGRAM WITH CLASSICAL CONTROL

We then give the second example, iterative phase estimation, a quantum algorithm that requires real-time feedback control.

Example V.2 (Iterative phase estimation): Iterative phase estimation (IPE) [36] is a kind of phase estimation algorithm requiring only one ancilla qubit [11], [37]. IPE requires real-time intermediate measurement and feedback control: For every iteration, the ancilla qubit is operated on, measured, and reset; the parameter of R_Z gate in each iteration is determined by previous measurement outcomes; all gates and measurements must be finished within decoherence time. The circuit of IPE is depicted in Fig. 8 The following is our isQ program for a two-qubit, 20-bit precision IPE instance:

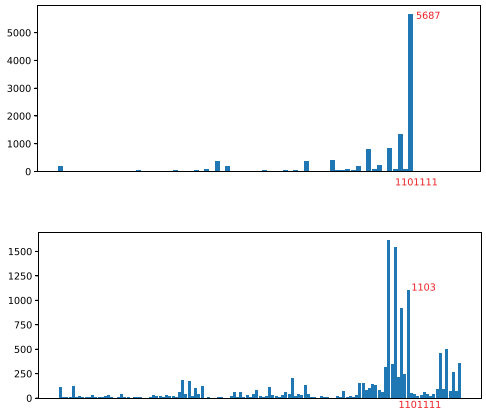

FIGURE 7. Top: The probabilistic distribution of results of the optimized circuit. Bottom: The probabilistic distribution of results of the circuit directly from mapping the raw circuit.

```
/* the phase angle is
   2*pi*867893/(2**20) = 2*pi*0.8276872 */
int x = 867893;
int m = 20;  // estimation precision
double theta(){
    double y = 2 ** m;
    return 2 * pi * x / y;
}
procedure U(double theta, qbit q){
    nctrl GPhase(theta, q);
} deriving gate
procedure pow2_ctrlU(int n, qbit anc, qbit ev){
    double t = theta() * (2 ** n);
    ctrl U(t, anc, ev);
}
double ipe_U(int n, qbit ev){
    qbit anc;
    double phi = 0;
    for i in 0:n {
        anc = |0>;
        H(anc);
        pow2_ctrlU(n - i - 1, anc, ev);
        Rz(-phi * 2 * pi, anc);
        H(anc);
        int res = M(anc);
        phi = (phi + (res / 2.0)) / 2.0;
    }
    return phi * 2.0;
}
procedure main(){
    qbit ev = |0>; // eigen vector of U
    double phi = ipe_U(m, ev);
    print phi * (2 ** m); // should be 867893
}
```

We created two examples, *IPE* the example above, and *IPE-9*, an IPE instance but U is a 9-qubit gate. We write two examples in both isQ and Q#, and compile and simulate them to compare the compilation and simulation performance. All experiments are carried out on the same PC with an Intel i9-11900 K CPU.

Table 2 shows that our toolchain has much smaller overhead for both compilation and execution for *IPE*. While Q# toolchain compiles the controlled-U in *IPE-9* into mutiqubit controlled gates and runs such controlled gates directly on the simulator, we are able to compile the program into finer-grain CNOT and U_3 gates and simulate them with a low overhead while preserving overall performance.

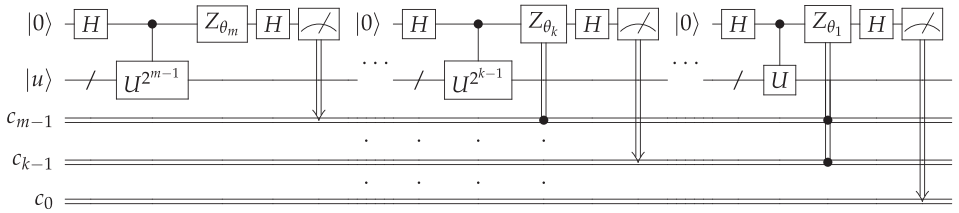

FIGURE 8. Circuit of iterative phase estimation.

TABLE 2. Evaluation Results Between isQ and Q# on Comping and Simulating Two Programs

Time (s)	IPE	IPE-9
Q# compile	4.31 ± 0.14	5.24 ± 0.09
Q# simulate	1.25 ± 0.01	1.41 ± 0.05
isQ compile	0.40 ± 0.02	1.70 ± 0.06
isQ simulate	0.003 ± 0.000	2.11 ± 0.02

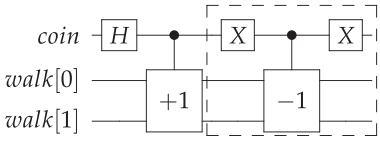

FIGURE 9. Loop body in circuit form. The +1 and −1 boxes represent the "increment" gate and its inverse. The dashed box represents the negated controller to the inversed gate.

C. COMPILING LOOP PROGRAMS

To show the optimizing capability of our compiler on high-level program structures, we give an example of using our compiler to compile and schedule a quantum loop program.

Example V.3: (Quantum random walk) Quantum random walk is the quantum analog of classical random walk. Quantum random walk is done by alternatively performing a coin-tossing operator on the coin space and a walking operator on both the coin space and walk space. We give an example of performing a quantum random walk using a Hadamard coin on a 2-qubit walking space, or equivalently, on the graph of a square. The loop body is given in Fig. 9, where the "increment" procedure and its inverse are represented by the boxes marked as +1 and −1, respectively. Note that two X gates are needed to change the controlled gate to one with the negated controller.

```
procedure increment(qbit walk_0, qbit walk_1){
    CNOT(walk_0, walk_1);
    X(walk_0);
} deriving gate
procedure main(){
    qbit coin;
    qbit walk[2];
    int n = 10; // parameter
    for i in 0:n{
        H(coin);
        ctrl increment(coin, walk[0], walk[1]);
        nctrl inv increment(coin, walk[0], walk[
    1]);
    }
    M(walk[0]);
    M(walk[1]);
}
```

Traditionally, optimizing quantum program compilers perform circuit optimization on unrolled quantum circuits only. While optimizing the entire quantum circuit allows maximizing circuit optimization, it requires the compilation time to grow as the number of gates in the entire program grows.

Specifically, for simple quantum loop programs consisting of *for*-loops, isQ supports performing gate-level optimization at loop-level without having to unroll the entire loop [50]: if a gate at the beginning of an iteration can be canceled out or merged into one U_3 gate with another gate from the next iteration. By leveraging MLIR support for affine expressions, our compiler is able to analyze which gates across iterations can be merged, therefore reducing the number of gates and total depth.

Example V.4: For the loop body in Fig. 9, if we unroll the loop body and allow U_3 gate, starting from the second iteration every H gate can be merged with the last X gate from the previous iteration. By detecting this pattern, we can compile the original loop into an equivalent loop program that performs this gate merging:

```
...
// Prologue
H(coin)
// Loop kernel
for i in 0:(n-1){
    ctrl increment(coin, walk[0], walk[1]);
    X(coin);
    ctrl inv increment(coin, walk[0], walk[1]);
    HX(coin); // HX is the merged U3 gate.
}
// Epilogue
ctrl increment(coin, walk[0], walk[1]);
X(coin);
ctrl inv increment(coin, walk[0], walk[1]);
X(coin);
...
```

We compared our compiler with the Qiskit compiler in compiling the program in Example V.3. Both compilers compile the program into $\{CZ, U_3\}$ basis. Since Qiskit does not support interiteration optimizations, we conducted a third group of experiments where we instruct Qiskit to first unroll the loop and then optimize the unrolled circuit. The following two metrics are used in the comparison.

1) Average loop kernel depth. This metric characterizes the average circuit depth of a single iteration of the loop. For isQ and Qiskit without explicit loop unrolling, the average loop kernel depth is obtained by counting the circuit depth of the optimized loop body.

TABLE 3. Evaluation Results of isQ and Qiskit Compiling a Loop Program

Compiler	Opt. level	Iteration number	Depth	Compilation time (s)
isQ	N/A	10	22	0.125 ± 0.005
	N/A	50	22	0.128 ± 0.005
	N/A	200	22	0.130 ± 0.006
Qiskit w/t unrolling	O1	50	27	0.006 ± 0.001
	O2	50	27	0.011 ± 0.001
	O3	50	24	0.026 ± 0.002
Qiskit w/ unrolling	O3	10	23.1	0.182 ± 0.004
	O3	50	23.02	0.950 ± 0.043
	O3	200	23.005	3.769 ± 0.077
	O1	50	26.02	0.217 ± 0.025
	O2	50	26.02	0.269 ± 0.026

For Qiskit with explicit loop unrolling, the average loop kernel is obtained by performing optimization on the unrolled circuit and dividing the total depth by the number of iterations.

2) Compilation time. We tested multiple optimization levels of Qiskit and measured the time of the whore compiling process.

The evaluation result is presented in Table 3. When Qiskit compiles the loop without unrolling, the resulted depth is larger than isQ, meaning the resulted circuit would run a longer time on a quantum computer. When the circuit is first unrolled, the average loop kernel depth becomes smaller. However, it is still slightly larger than isQ. The reason is that the default O3 transpiler of Qiskit does not remove a redundant identity single-qubit gate generated in the compilation.

In addition, our compiler also saves compilation time by optimizing directly on the loop program without having to unroll the loop. It can be seen from the table that the compilation time is nearly the same for different iteration numbers. However, the compilation time for Qiskit scales with the iteration number. Although current NISQ hardware may not support the large number of gates listed in the table, we anticipate that as quantum hardware develops to support deeper circuits and higher-level control flow, direct optimization of quantum loop programs will become increasingly important, just like its role in the compilation of classical programs.

D. COMPILING RECURSIVE QUANTUM PROGRAMS

In Section V-B, quantum programs with classical control are discussed. Nowadays, many quantum software tools support the classical control flow. We still take Qiskit for example. After the OpenQASM 3.0 [11] was proposed, Qiskit has expanded its support for various classical control flow features, including mid-circuit measurements, if-branches, for-loops, and while-loops. However, it currently does not support recursion. In this section, we present two examples of quantum programs that incorporate recursion. We demonstrate how to write these programs using the isQ language and compile them into low-levels IRs.

Example V.5: The first example from the work in [15] demonstrates a recursive procedure characterized by a flowchart depicted in Fig. 10. In this flowchart, the term "discard r" refers to a reset operation, i.e., "$r := |0\rangle$." The example serves to showcase isQ's capabilities in compiling recursive quantum programs and implementing optimizations on high-level program structures.

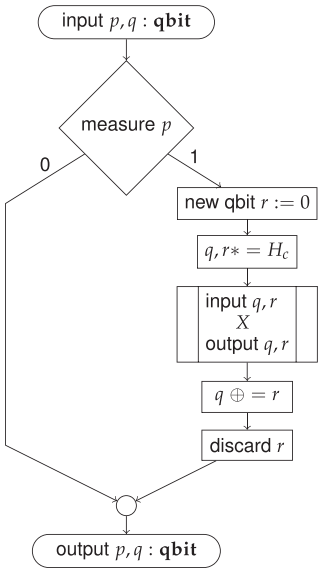

FIGURE 10. Flowchart of a recursive procedure.

We implement the example in isQ, as shown in the following:

```
procedure proc_X(qbit p, qbit q){
    int a = M(p);
    if (a == 1){
        qbit r;
        ctrl H(q, r);
        proc_X(q, r);
        CNOT(r, q);
    }
}

procedure main(){
    qbit a, b;
    H(a);
    H(b);
    proc_X(a, b);
}
```

Compiling the program directly without any optimizations, as shown in the flowchart, leads to the creation of a new ancilla qubit at every level of recursion, which restricts the recursion depth when only a small number of qubits are available. Whereas, the isQ compiler identifies an optimization opportunity by reusing measured qubit: Once qubit p has been measured with the outcome stored in a classical variable, it can be recycled and reused at the next recursion level; when the ancilla is freed, the state of original p can be easily restored according to previous measurement outcome. Consequently, only three qubits are required overall, regardless of the recursion depth.

We demonstrate how the optimization is done at IR level. The procedure **proc_X** is compiled to IR snippet as follows:

```
func @proc_X(%p: memref<1x!isq.qstate>, %q:
    memref<1x!isq.qstate>) {
    %1 = affine.load %p[0] :memref<1x!isq.qstate>
    %2, %m = isq.call_qop
        @__isq__builtin__measure(%1) : [1] () ->
         i1
    affine.store %2, %p[0] :memref<1x!isq.qstate>
    scf.if %m {

      %3 = memref.alloc() :memref<1x!isq.qstate>
      /* omitted */
      affine.store %x, %3[0] :memref<1x!isq.
          qstate>
      memref.dealloc %3 :memref<1x!isq.qstate>
    }
    return
}
```

Our compiler is able to find the following.

1) The lifetime of the qubit allocated in the loop is outlived by the lifetime of the qubit in the procedure parameter.
2) As of allocation, the state of the outer qubit just went through a measurement.

Therefore, the measured qubit can be reused as the ancilla and restored when the ancilla usage is finished, eliminating the allocation in the loop:

```
func @proc_X(%p: memref<1x!isq.qstate>, %q:
    memref<1x!isq.qstate>) {
    %1 = affine.load %p[0] :memref<1x!isq.qstate>
    %2, %m = isq.call_qop
        @__isq__builtin__measure(%1) : [1] () ->
         i1
    affine.store %2, %p[0] :memref<1x!isq.qstate>
    scf.if %m {
      %4 = affine.load %p[0] :memref<1x!isq.
          qstate>
      /* omitted */
      affine.store %x, %p[0] :memref<1x!isq.
          qstate>
      // Reset qubit "p" to state by %m.
      %x1 = affine.load %p[0] :memref<1x!isq.
          qstate>
      %x2 = isq.call_qop @__isq__builtin__reset(%
          x1) : [1] () -> ()
      affine.store %x2, %p[0] :memref<1x!isq.
          qstate>
      scf.if %m {
        %X = isq.use @X : !isq.gate<1>
        %x3 = affine.load %p[0] :memref<1x!isq.
            qstate>
        %x4 = isq.apply %X(%x3) : !isq.gate<1>
        affine.store %x4, %p[0] :memref<1x!isq.
            qstate>
      }
    }
    return
}
```

The second example is the RFS including both recursion and oracle definitions.

Example V.6 (Recursive Fourier Sampling): RFS [49], [51] is the recursive version of the Bernstein–Vazirani algorithm in Section V-A. The height-h RFS problem is defined to be a h-depth tree with each subtree of the root corresponding to a height-$(h-1)$ RFS. The RFS problem could be solved recursively. The following is an instance of RFS of height-2.

```
oracle A(4,1)=[0,1,1,0,0,0,0,0,0,0,1,1,0,1,0,1];
oracle g(2,1)=[1,0,1,0];


int a;
qbit q[4], p[3];

procedure recursive_fourier_sampling(int k){
    if (k == 2){
        A(q[0], q[1], q[2], q[3], p[2]);
    }else{
        H(q[2*k]);
        H(q[2*k+1]);
        X(p[k+1]);
        H(p[k+1]);

        recursive_fourier_sampling(k+1);

        H(q[2*k]);
        H(q[2*k+1]);
        g(q[2*k],q[2*k+1], p[k]);
        H(q[2*k]);
        H(q[2*k+1]);

        recursive_fourier_sampling(k+1);

        H(q[2*k]);
        H(q[2*k+1]);
        H(p[k+1]);
        X(p[k+1]);
    }
}

procedure main(){
    a = 0;
    recursive_fourier_sampling(a);
    M(p[0]);
}
```

The isQ software stack can compile this RFS instance into various IRs, such as eQASM and QCIS. The difference is that eQASM supports dynamic classical control while compiling it to QCIS requires unrolling the entire program, resulting in an extensive static circuit. We have evaluated the compiled output using both the eQASM simulator and the QCIS simulator we implemented. In both cases, the correct result of 1 was obtained, indicating that $g(s) = 1$ for this instance.

VI. CONCLUSION

This article describes a software stack for quantum programming, including a high-level quantum programming language and tools to compile and simulate the programs. The isQ programming language provides a series of high-level language features to facilitate writing quantum programs. By leveraging MLIR infrastructure, we defined isQ-IR, an MLIR dialect for representing quantum-classical hybrid programs, and built an isQ compiler, allowing some powerful transformations and optimizations of isQ programs. Finally, backend code-generators and simulators allow isQ programs to be finally compiled to and tested out on different backends, both simulated environment and real superconducting hardware, and also allow cooperation with lower-level compilation toolchains.

A. FUTURE WORK

Potential future works include further extending the isQ programming stack and adding more optimization passes, resource analysis and adding assertion functions, etc. Targeting isQ to more different kinds of quantum hardware, e.g., trapped-ion quantum devices, is also one of the future tasks.

ACKNOWLEDGMENT

The authors thank Zhicheng Zhang, Kezhen Zhang, Qisheng Wang, Xiangzhen Zhou, Keli Zheng, and Ji Guan for their kind help and enlightening discussion. The authors are also grateful to the fellows in USTC for providing access to their superconducting hardware.

REFERENCES

[1] F. Arute et al., "Quantum supremacy using a programmable superconducting processor," *Nature*, vol. 574, no. 7779, pp. 505–510, 2019, doi: 10.1038/s41586-019-1666-5.

[2] H.-S. Zhong et al., "Quantum computational advantage using photons," *Science*, vol. 370, no. 6523, pp. 1460–1463, 2020, doi: 10.1126/science.abe8770.

[3] Q. Zhu et al., "Quantum computational advantage via 60-qubit 24-cycle random circuit sampling," *Sci. Bull.*, vol. 67, no. 3, pp. 240–245, 2022, doi: 10.1016/j.scib.2021.10.017.

[4] "February 23, 2022 IIonQ Aria Furthers Lead as World's Most Powerful Quantum Com." [Online]. Available: https://ionq.com/news/february-23-2022-ionq-aria-furthers-lead

[5] D. M. Beazley, "PLY (Python Lex-Yacc)," 2020. [Online]. Available: https://ply.readthedocs.io/

[6] R. S. Smith, M. J. Curtis, and W. J. Zeng, "A practical quantum instruction set architecture," 2016, *arXiv:1608.03355*, doi: 10.48550/arXiv.1608.03355.

[7] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools*. Noida, India: Pearson Education India, 2007.

[8] C. Lattner et al., "MLIR: A compiler infrastructure for the end of Moore's law," 2020, *arXiv:2002.11054*, doi: 10.48550/arXiv.2002.11054.

[9] Q. Alliance, "QIR specification," 2022. [Online]. Available: https://github.com/qir-alliance/qir-spec

[10] X. Fu et al., "eQASM: An executable quantum instruction set architecture," in *Proc. IEEE Int. Symp. High Perform. Comput. Architecture*, 2019, pp. 224–237, doi: 10.1109/HPCA.2019.00040.

[11] A. W. Cross et al., "OpenQASM 3: A broader and deeper quantum assembly language," *ACM Trans. Quantum Comput.*, vol. 3, no. 3, 2022, Art. no. 12, doi: 10.1145/3505636.

[12] "QCIS instruction set introduction," 2023. [Online]. Available: https://quantumcomputer.ac.cn/UserBook.html

[13] "The iSQ project," 2023. [Online]. Available: https://github.com/arclight-quantum/isQ-Compiler

[14] "The website of iSQ docs," 2023. [Online]. Available: http://www.arclightquantum.com/isq/index.html

[15] P. Selinger, "Towards a quantum programming language," *Math. Struct. Comput. Sci.*, vol. 14, no. 4, pp. 527–586, 2004, doi: 10.1017/S0960129504004256.

[16] M. Ying, *Foundations of Quantum Programming*. San Mateo, CA, USA: Morgan Kaufmann, 2016. [Online]. Available: https://shop.elsevier.com/books/foundations-of-quantum-programming/ying/978-0-12-802306-8

[17] A. S. Green, P. L. Lumsdaine, N. J. Ross, P. Selinger, and B. Valiron, "Quipper: A scalable quantum programming language," in *Proc. 34th ACM SIGPLAN Conf. Program. Lang. Des. Implementation*, 2013, pp. 333–342, doi: 10.1145/2491956.2462177.

[18] Qiskit contributors, "Qiskit: An open-source framework for quantum computing," 2023.

[19] K. Svore et al., "Q#: Enabling scalable quantum computing and development with a high-level DSL," in *Proc. Real World Domain Specific Lang. Workshop*, 2018, pp. 1–10, doi: 10.1145/3183895.3183901.

[20] B. Bichsel, M. Baader, T. Gehr, and M. Vechev, "Silq: A high-level quantum language with safe uncomputation and intuitive semantics," in *Proc. 41st ACM SIGPLAN Conf. Program. Lang. Des. Implementation*, 2020, pp. 286–300, doi: 10.1145/3385412.3386007.

[21] S. Sivarajah, S. Dilkes, A. Cowtan, W. Simmons, A. Edgington, and R. Duncan, "t | ket⟩: A retargetable compiler for NISQ devices," *Quantum Sci. Technol.*, vol. 6, no. 1, Nov. 2020, Art. no. 014003, doi: 10.1088/2058-9565/ab8e92.

[22] X. Fu et al., "Quingo: A programming framework for heterogeneous quantum-classical computing with NISQ features," *ACM Trans. Quantum Comput.*, vol. 2, no. 4, pp. 1–37, 2021, doi: 10.1145/3483528.

[23] A. Mccaskey, T. Nguyen, A. Santana, D. Claudino, T. Kharazi, and H. Finkel, "Extending C for heterogeneous quantum-classical computing," *ACM Trans. Quantum Comput.*, vol. 2, no. 2, pp. 1–36, 2021, doi: 10.1145/3462670.

[24] N. Khammassi et al., "OpenQL: A portable quantum programming framework for quantum accelerators," *ACM J. Emerg. Technol. Comput. Syst.*, vol. 18, no. 1, pp. 1–24, 2021, doi: 10.1145/3474222.

[25] A. Matsuura et al., "An Intel quantum software development kit for efficient execution of variational algorithms," in *Proc. APS Mar. Meeting Abstr.*, 2022, vol. 2022, pp. N36–006.

[26] A. W. Cross, L. S. Bishop, J. A. Smolin, and J. M. Gambetta, "Open quantum assembly language," 2017, *arXiv:1707.03429*, doi: 10.48550/arXiv.1707.03429.

[27] A. J. McCaskey, D. I. Lyakh, E. F. Dumitrescu, S. S. Powers, and T. S. Humble, "XACC: A system-level software infrastructure for heterogeneous quantum-classical computing," *Quantum Sci. Technol.*, vol. 5, no. 2, 2020, Art. no. 024002, doi: 10.1088/2058-9565/ab6bf6.

[28] A. JavadiAbhari et al., "ScaffCC: Scalable compilation and analysis of quantum programs," *Parallel Comput.*, vol. 45, pp. 2–17, 2015, doi: 10.1016/j.parco.2014.12.001.

[29] A. McCaskey and T. Nguyen, "A MLIR dialect for quantum assembly languages," in *Proc. IEEE Int. Conf. Quantum Comput. Eng.*, 2021, pp. 255–264, doi: 10.1109/QCE52317.2021.00043.

[30] D. Ittah, T. Häner, V. Kliuchnikov, and T. Hoefler, "Qiro: A static single assignment-based quantum program representation for optimization," *ACM Trans. Quantum Comput.*, vol. 3, no. 3, Jun. 2022, doi: 10.1145/3491247.

[31] A. Peduri, S. Bhat, and T. Grosser, "QSSA: An SSA-based IR for quantum computing," in *Proc. 31st ACM SIGPLAN Int. Conf. Compiler Construction*, 2022, pp. 2–14, doi: 10.1145/3497776.3517772.

[32] K. Knobe and V. Sarkar, "Array SSA form and its use in parallelization," in *Proc. 25th ACM SIGPLAN-SIGACT Symp. Princ. Program. Lang.*, 1998, pp. 107–120, doi: 10.1145/268946.268956.

[33] E. Farhi, J. Goldstone, and S. Gutmann, "A quantum approximate optimization algorithm," 2014, *arXiv:1411.4028*, doi: 10.48550/arXiv.1411.4028.

[34] A. Peruzzo et al., "A variational eigenvalue solver on a photonic quantum processor," *Nature Commun.*, vol. 5, no. 1, Jul. 2014, Art. no. 4213, doi: 10.1038/ncomms5213.

[35] M. Zhang et al., "Exploiting different levels of parallelism in the quantum control microarchitecture for superconducting qubits," in *Proc. IEEE/ACM 54th Annu. Int. Symp. Microarchitecture*, 2021, pp. 898–911, doi: 10.1145/3466752.3480116.

[36] M. Dobší ček, G. Johansson, V. Shumeiko, and G. Wendin, "Arbitrary accuracy iterative quantum phase estimation algorithm using a single ancillary qubit: A two-qubit benchmark," *Phys. Rev. A*, vol. 76, no. 3, pp. 399–406, 2007, doi: 10.1103/PhysRevA.76.030306.

[37] A. D. Córcoles et al., "Exploiting dynamic quantum circuits in a quantum algorithm with superconducting qubits," *Phys. Rev. Lett.*, vol. 127, Aug. 2021, Art. no. 100501, doi: 10.1103/PhysRevLett.127.100501.

[38] A. B. Watts, R. Kothari, L. Schaeffer, and A. Tal, "Exponential separation between shallow quantum circuits and unbounded fan-in shallow classical circuits," in *Proc. 51st Annu. ACM SIGACT Symp. Theory Comput.*, 2019, pp. 515–526, doi: 10.1145/3313276.3316404.

[39] P. W. Shor, "Scheme for reducing decoherence in quantum computer memory," *Phys. Rev. A*, vol. 52, no. 4, 1995, Art. no. R2493, doi: 10.1103/PhysRevA.52.R2493.

[40] E. Dennis, A. Kitaev, A. Landahl, and J. Preskill, "Topological quantum memory," *J. Math. Phys.*, vol. 43, no. 9, pp. 4452–4505, 2002, doi: 10.1063/1.1499754.

[41] M. Broughton et al., "TensorFlow quantum: A software framework for quantum machine learning," Aug. 2021, *arXiv:2003.02989*, doi: 10.48550/arXiv.2003.02989.

[42] D. S. Steiger, T. Häner, and M. Troyer, "ProjectQ: An open source software framework for quantum computing," *Quantum*, vol. 2, p. 49, Jan. 2018, doi: 10.22331/q-2018-01-31-49.

[43] Rigetti Computing, "pyQuil Documentation," 2023. [Online]. Available: https://pyquil-docs.rigetti.com

[44] L. K. Grover, "Fixed-point quantum search," *Phys. Rev. Lett.*, vol. 95, no. 15, 2005, Art. no. 150501, doi: 10.1103/PhysRevLett.95.150501.

[45] E. Bernstein and U. Vazirani, "Quantum complexity theory," *SIAM J. Comput.*, vol. 26, no. 5, 1997, doi: 10.1137/S0097539796300921.

[46] M. A. Nielsen and I. L. Chuang, *Quantum Computation and Quantum Information*. Cambridge, U.K.: Cambridge Univ. Press, 2000.

[47] V. V. Shende, S. S. Bullock, and I. L. Markov, "Synthesis of quantum logic circuits," *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, vol. 25, no. 6, pp. 1000–1010, Jun. 2006, doi: 10.1109/TCAD.2005.855930.

[48] X. Zhou, S. Li, and Y. Feng, "Quantum circuit transformation based on simulated annealing and heuristic search," *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, vol. 39, no. 12, pp. 4683–4694, Dec. 2020, doi: 10.1109/TCAD.2020.2969647.

[49] E. Bernstein and U. V. Vazirani, "Quantum complexity theory," *SIAM J. Comput.*, vol. 26, no. 5, pp. 1411–1473, 1997, doi: 10.1137/S0097539796300921.

[50] J. Guo and M. Ying, "Software pipelining for quantum loop programs," *IEEE Trans. Softw. Eng.*, vol. 49, no. 4, pp. 2815–2828, Apr. 2023, doi: 10.1109/TSE.2022.3232623.

[51] S. Aaronson, "Quantum lower bound for recursive Fourier sampling," *Quantum Inf. Comput.*, vol. 3, no. 2, pp. 165–174, 2002.