`staq`– A full-stack quantum processing toolkit

Matthew Amy*[1,2] and Vlad Gheorghiu†[1,3,4]

[1]softwareQ Inc., Kitchener ON, Canada
[2]Department of Mathematics & Statistics, Dalhousie University, Halifax NS, Canada
[3]Institute for Quantum Computing, University of Waterloo, Waterloo ON, Canada
[4]Department of Combinatorics and Optimization, University of Waterloo, Waterloo ON, Canada

Version of June 10, 2020

Abstract

We describe `staq`, a full-stack quantum processing toolkit written in standard C++. `staq` is a quantum compiler toolkit, comprising of tools that range from quantum optimizers and translators to physical mappers for quantum devices with restricted connectives. The design of `staq` is inspired from the UNIX philosophy of "less is more", i.e. `staq` achieves complex functionality via combining (piping) small tools, each of which performs a single task using the most advanced current state-of-the-art methods. We also provide a set of illustrative benchmarks.

Contents

1 Introduction **2**

2 `staq` **3**
2.1 Overview . . . 3
2.2 Circuit synthesis . . . 5
2.3 Optimization . . . 7
2.4 Hardware mapping . . . 9
2.4.1 Initial layout generation . . . 10
2.4.2 CNOT mapping . . . 10
2.5 Compilation . . . 14
2.6 Performance . . . 14

3 Conclusions and future directions **15**

A Transpiler example **20**

*matt.amy@dal.ca
†vlad@softwareq.ca

1 Introduction

Quantum computing is a new paradigm of physics that promises significant computational advantages for a plethora of applications, ranging from optimizing, material design, drug discovery to sensing and measurement and secure communication. The idea of harnessing the power of quantum mechanics to perform computations that are believed to be much harder or even intractable by classical computers dates back to Feynman [1]. However, due to experimental challenges, the first public-access small programmable quantum computing platforms appeared within the last five years, whereas the first demonstration of a computational task that can be performed significantly faster by a quantum computer was publicly released on October 2019 [2].

The currently available quantum computing platforms are not (yet) fault tolerant, i.e. they can not perform arbitrary long quantum computations with arbitrarily low error, but consist mainly of noisy qubits with restricted connectivity, for which the computation length is restricted by the depth of the logical circuit to be run. Such platforms are informally termed "Noisy Intermediate-Scale Quantum" computers (NISQ) [3], and represent the first step towards realization of large-scale fault-tolerant quantum platforms.

Quantum algorithms are usually described in a high-level language (e.g. plain English or quantum "pseudo-code") then "translated" into a quantum circuit consisting of a series of quantum gates applied in a sequential manner[1], followed by a measurement from which the end result of the algorithm is being inferred via classical post-processing techniques, see e.g. [6] for more details.

The translation from a high-level quantum algorithm to a quantum circuit is informally called "quantum compiling", and consists of a series of steps, which depend on the particular quantum architecture that is being used. Those steps are usually thought of as a quantum compiling top-down "stack" (with the most abstract layers higher up in the stack), and may involve e.g. translation of parts of a high level quantum algorithm to logical Boolean circuits (such as the mapping of quantum oracles in quantum searching algorithms to Boolean functions), converting Boolean functions to quantum reversible circuits, optimizing the latter in terms of a particular cost and taking into account connectivity constraints (if any), and finally mapping the resulting quantum circuit to a specific physical architecture, or translating it to some particular kind of quantum machine assembly language.

Any optimizations or improvements along the stack affect (beneficially) the quantum computation speed (QPU cycles/wall time), and may even allow longer computations on NISQ devices, which otherwise would be infeasible due to their prohibitively-large circuit depth. Therefore constructing "full-stack" quantum processing toolkits is of paramount importance for both the NISQ regime and also far-future large-scale fault-tolerant quantum platforms.

`staq` represents a joint effort at softwareQ Inc. to construct such a full-stack quantum computing toolkit. Our effort is not the first and likely not the last in the fast dynamic field of quantum software – the myriad of existing quantum compilers and toolkits includes Quipper [7], ScaffCC [8], Quil/PyQuil [9], ProjectQ [10], $Q|SI\rangle$ [11], Q# [12] and Strawberry Fields [13] to name just a few. In contrast to those compilers, `staq` – inspired by the UNIX [14] philosophy of building up complex functionality by combining (piping) small command-line tools – is designed as a collection of minimalist tools, ranging from circuit optimizers and translators to physical mappers for NISQ architectures with restricted connectives, operating on a single textual language, openQASM [15]. While modularity in compilers is hardly a novel concept, `staq` is differentiated by a Clang [16] style approach of representing programs by, and operating directly on, syntax trees, rather than various internal representations. This approach allows us to export compiler modules directly as single-purpose source-to-source tools that make minimal, independent changes to a given program, and which can then be combined at will with other tools or compilers.

As a compiler and software toolkit, `staq` integrates *in a production setting* the latest state-of-the-art methods in quantum compiling targeting the whole spectrum of the quantum software stack, starting from the abstract higher algorithmic layers and ending at the physical mapping layer. In particular, `staq` includes state-of-the-art techniques which have either never been implemented, or otherwise only implemented in

[1]The quantum gate model is not the only quantum computing model, and others exist, such as Measurement-Based Quantum Computing [4], Adiabatic Quantum Computing [5], etc. However, in this paper, we will focus on the gate model, as currently it seems to be the favourite among the community that believes in full-scale fault-tolerant quantum computation.

restricted academic settings. Moreover, `staq` is highly portable, being written in standard C++ (using the C++17 standard), and fast, as shown by our benchmarks in Section 2.6. `staq` also offers some unique features, such as the ability to use and synthesize classical logic, specified in an industrial-strength language such as Verilog [17], directly within openQASM code. Finally, `staq` can generate quantum code in a variety of formats that encompass many of the currently available quantum platforms [15, 18, 19, 9, 10, 12].

The remainder of this paper describes the `staq` toolkit and its functionality and uses cases while providing a set of benchmarks (Section 2), followed by conclusions and future directions (Section 3).

2 staq

`staq` is a new compiler and software toolkit for the openQASM language [15] written in C++. The primary goal is to provide a suite of transformation, optimization and compilation tools that can operate on a single, common language, and output to a number of different simulation and hardware execution platforms. On the technical side, a focus of `staq` is to support state-of-the-art circuit transformation algorithms, which are typically implemented on small subsets of circuits or in restricted research contexts, and apply them natively to *any valid quantum program*. This algorithmic focus is distinct from other quantum computing toolchains, which are typically slow to adopt bleeding-edge techniques. We first give a brief review of the openQASM language, with the remainder of this section giving an overview of the architecture, usage, and algorithmic methods of `staq`.

The openQASM language The official specification of openQASM can be found in [15], but we provide a brief overview here. Programs in openQASM are structured as sequences of declarations and commands. As an intermediate- to low-level language, openQASM provides a small number of basic programming features: declaration of static-size classical or quantum registers, definition of (unitary) circuits or *gates*, gate application, measurement and initialization of qubits, and finally classically controlled gates. The listing below gives an example of an openQASM program performing quantum teleportation:

```
OPENQASM 2.0;
include "qelib1.inc";

gate bellPrep x,y {
  h x;
  cx x,y;
}

qreg q[3];
creg c0[1];
creg c1[1];

bellPrep q[1],q[2];
cx q[0],q[1];
h q[0];
measure q[0] -> c0[0];
measure q[1] -> c1[0];
if(c0==1) z q[2];
if(c1==1) x q[2];
```

2.1 Overview

To support the minimalist philosophy of small, single-function tools, `staq` was designed from the bottom-up to allow the manipulation, transformation, compilation and translation of QASM files according to the

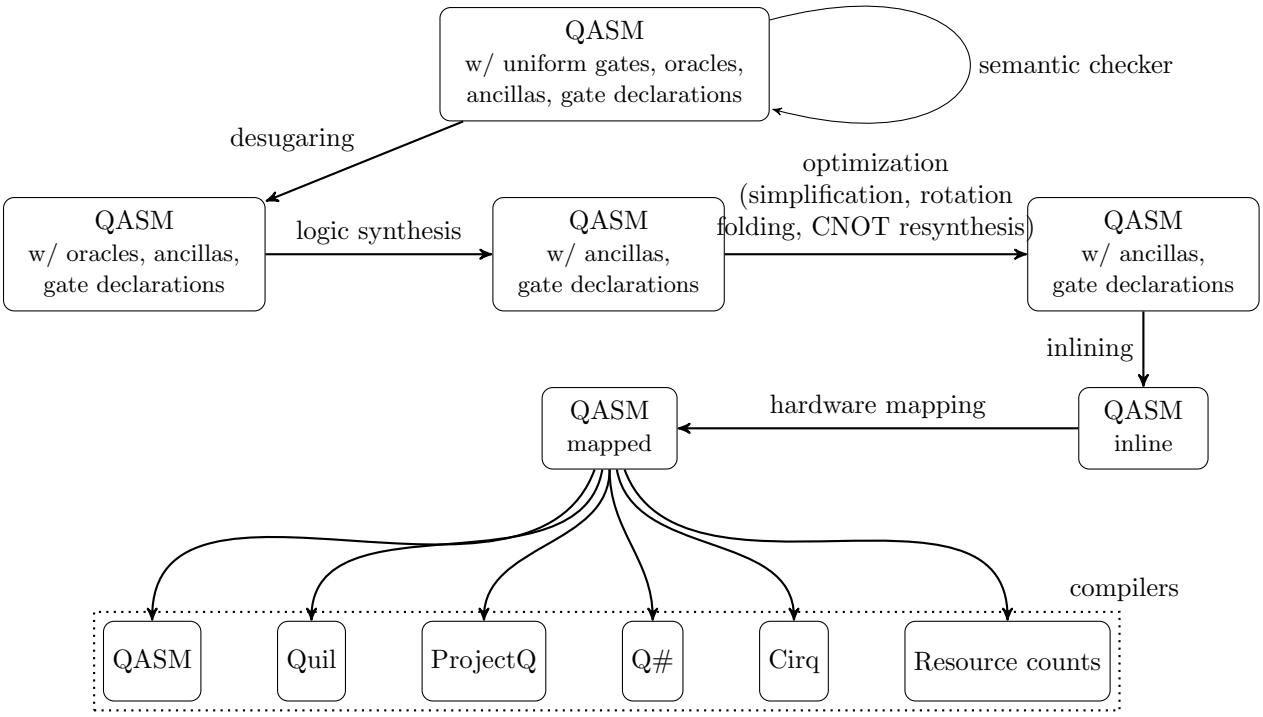


Figure 1: Overview of the `staq` toolchain

following goal:

no process should affect the original structure of the program more than absolutely necessary.

In particular, an un-transformed program should output to something that looks identical to the input source code, modulo changes in whitespace. Similarly, one should be able to optimize programs without disturbing the structure of the original program, or to the extent that the developer wishes to enable further optimizations, for instance by first inlining and then performing *whole-program* optimization.

To achieve this, `staq` stores and operates *directly on QASM syntax trees*, rather than an intermediate representation. This approach was inspired by Clang [16], which acts as an effective middle-end for the analysis and transformation of C code. In keeping with the Clang style of program analysis, `staq` provides a powerful set of *Visitor* classes for performing different types of traversal as well as AST splicing, on which all of our transformations are built.

Figure 1 gives an overview of the `staq` toolchain and typical usage. The main command line compiler – `staq` – offers a flexible *pass registration* system, whereby passes are given via command-line arguments and are executed in the order given. In particular, it is often useful to perform basic gate simplifications both before and after other optimizations, or to inline certain gates (e.g., `ccx`) first, optimize, then inline fully to primitive gates – these types of usage patterns are supported by the pass registration system, for instance with[2] `staq -s -r -s circuit.qasm` in the former case.

As all of the transformations are defined directly on QASM ASTs, the order of operations is generally interchangeable, with two major exceptions:

1. desugaring must occur before any other transformation, and
2. the program must be *fully* inlined before mapping.

[2]This sequence of passes is also given the explicit name `-O1`, in analogy to the basic optimization level of GCC.

For this reason, the compiler automatically applies a desugaring pass after parsing and semantic checks, and an inlining pass preceding a hardware mapping pass. Desugaring mainly involves replace *uniform gates* – gates applied to registers – with a sequence of gates applied to individual qubits. For instance, if `x` and `y` are qubit registers of length 2, the desugarer will replace `cx x,y;` with

```
cx x[0],y[0];
cx x[1],y[1];
```

The semantic checker ensures that all such uniform gates are well-formed according to the specification in [15], as well as other semantic properties such as correct argument types.

The general inline pass supports *overrides*, whereby the user can specify which gates should *not* be inlined. By default, the gates defined in `qelib1.inc` [15] are not inlined, except before hardware mapping. The remaining synthesis, optimization, and mapping passes are described in more detail in the follow sections.

Tool suite In addition to a single compiler, the `staq` software package also includes a suite of light-weight command line tools which can be chained together using Unix-style *pipelines* to perform a range of compilation tasks. Each tool reads a QASM file from `stdin`, performs a specific function, and outputs the transformed QASM source on `stdout` – as an exception, the *compiler* tools output the QASM source in various other languages. This offers a more flexible and customizable compilation pipeline at the expense of extra parsing stages, as well as the option to only build the relevant tools for a particular use case. For a full description of the available tools, the reader is directed to the `staq` wiki[3].

openQASM extensions `staq` supports a number of extensions to the openQASM language, both implemented and planned. In particular, `staq` supports the declaration and use of ancillas local to gate declarations, as well as the declaration of quantum oracles from classical Verilog logic files. These extensions are described in more detail in Section 2.2. Future planned versions will support iteration and register arguments to gates à la metaQASM [20].

2.2 Circuit synthesis

A unique feature of `staq` is the ability to splice classical logic *directly into quantum programs*, and moreover the ability to *synthesize* a circuit implementing the classical logic during compilation. This is done through a QASM language extension adding *oracle* gate declarations, with synthesis handled by the EPFL Logic Synthesis Libraries [21]. At this time, `staq` supports classical logic written in (the combinational subset of) the Verilog hardware description language.

To declare an oracle gate in a `staq`-QASM file, the keyword `oracle` is used in place of `gate`, and the classical logic defining the gate is given in the body as the name of a verilog file:

```
oracle MUX sel,x,y,out { "mux.v" }
```

The listing below shows the corresponding Verilog file `mux.v`.

```
// mux.v
module top (a, b, c, d);
  input a,b,c;
  output d;
  wire tmp1,tmp2,tmp3;
  assign tmp1 = a & b;
  assign tmp2 = ~a & c;
  assign tmp3 = ~tmp1 & ~tmp2;
  assign d = ~tmp3;
endmodule
```

[3] https://github.com/softwareQinc/staq/wiki/The-staq-tool-suite.

Combinational logic can be written in Verilog as a sequence of assignments of logical expressions to either *outputs* or temporary *wires*. For a full overview of the Verilog programming language, the reader is directed to [22]. Due to the reversibility of quantum oracles, there must be exactly one oracle input for every input and output of the given Verilog file. Oracle inputs are mapped to Verilog inputs and outputs sequentially, regardless of naming.

The logic synthesis pass of the compiler visits the AST and replaces each oracle declaration with a corresponding gate declaration. When an oracle declaration is encountered, it parses the specified file to generate an Majority-inverter graph (MIG), which is then synthesized over the Clifford group and T gates using the EPFL implementation of the LUT-based Hierarchical Reversible Logic Synthesis (LHRS) framework [23].

In general, a classical function may require *ancillas* to be implemented reversibly, and so synthesis of classical logic may require additional ancillas that have not been given directly as inputs to the gate. `staq` handles the introduction of ancillas with another QASM language extension adding support the declaration of local ancillas within gate bodies. Inside gate declarations, both *clean* and *dirty* ancilla registers – registers initialized in the state $|00\cdots0\rangle$ or in some unknown state, respectively – can be declared similarly to regular QASM registers by using the keyphrases `ancilla` or `dirty ancilla`, respectively. All ancillas are assumed to be returned to their initial state at the end of the gate body, and it remains the programmer's responsibility to ensure that this requirement is satisfied. Lightweight verification methods such as *path-sum verification* [24] can be adopted in the future to ensure that all ancillas are properly cleaned at the end of a gate.

The result of synthesizing the `MUX` gate above is shown below. Despite the use of temporary wires in the input Verilog file, the LHRS synthesis algorithm is able to find an ancilla-free implementation, hence the resulting local ancilla register has size 0.

```
gate MUX sel,x,y,out {
    h out;
    cx x,out;
    tdg out;
    cx sel,out;
    t out;
    cx x,out;
    tdg out;
    cx sel,out;
    t out;
    cx sel,x;
    tdg x;
    cx sel,x;
    t sel;
    t x;
    h out;
---------\\----------
    h out;
    cx y,out;
    t out;
    cx sel,out;
    t out;
    cx y,out;
    tdg out;
    cx sel,out;
    tdg out;
    cx sel,y;
    tdg y;
    cx sel,y;
    t sel;
    tdg y;
    h out;
}
```

Remark 2.1. The synthesized circuit above has T-count 14, since the 3-qubit multiplexor can be implemented with 2 Toffoli gates. This can further be reduced to 8 with the light optimization pass, `-O1`.

Ancilla management As local ancilla allocation is non-standard QASM and moreover not supported by most QPUs, `staq` performs automatic ancilla management during the inlining phase of compilation. In particular, local ancilla declarations are hoisted, as regular qubit registers, to the top of the global scope when a gate is inlined.

Since ancillas are assumed to be returned *clean* – i.e. returned in their initial state – it is not always necessary to allocate a new ancilla for every gate application. `staq` handles the re-use of ancillas during compilation by maintaining a pool of previously allocated ancillas and available dirty qubits, re-using qubits from these pools to fulfill ancilla requirements whenever possible. Figure 2 shows an example of both dirty

```
OPENQASM 2.0;
include "qelib1.inc";

gate foo a {
  ancilla b[1];
  cx a,b[0];
}

gate bar a {
  ancilla b[1];
  dirty ancilla c[1];
  cx a,b[0];
  cx a,c[0];
}

qreg x[2];
foo x[0];
bar x[0];
```

(a) Before inlining

```
OPENQASM 2.0;
include "qelib1.inc";

gate foo a {
  ancilla b[1];
  cx a,b[0];
}

gate bar a {
  ancilla b[1];
  dirty ancilla c[1];
  cx a,c[0];
  cx c[0],b[0];
}

qreg anc[1]
qreg x[2];
cx x[0],anc[0];
cx x[0],x[1];
cx x[1],anc[0];
```

(b) After inlining

Figure 2: Automatic ancilla sharing

and clean ancilla allocation and sharing between gate applications.

2.3 Optimization

Circuit optimization is necessary to produce efficient circuits which both utilize existing technology to the best of its ability, and to provide accurate resource estimates to guide the development of quantum algorithms and hardware. In contrast to other quantum computing software packages, `staq` was designed with circuit optimization as an integral part of the compiler. In this section we provide an overview of the optimization algorithms implemented: `simplify`, `rotation folding`, and `CNOT resynthesis`.

Gate simplifications The `simplify` optimization pass performs basic gate cancellations. In particular, it scans the program dependence graph and removes pairs of adjacent inverse gates whenever found, repeating until a fixpoint is reached. By (implicitly) using the program dependence graph rather than looking for gates adjacent in the AST, trivial commutations of gates acting on different qubits are modded out. As an example,

```
s x; h x; t y; h x; sdg x;
```

reduces to `t y;` with the `simplify` optimization pass.

In general, because each pair of eliminated gates may open up other simplifications, the fixpoint computation may run $O(l)$ times, where l is the number of gates in the program. In some cases this extra cost – making the optimization quadratic in the length of the program – may be prohibitive. In these cases, the user can opt to run single-pass simplifications instead of repeating until a fixpoint is reached.

Rotation folding The main optimization of the `staq` compiler is an extended implementation of Fang and Chen's T-count optimization[4] algorithm [27]. Their algorithm reframes the problem of merging T gates in Clifford+T circuits in the *Pauli sum* view of Gosset *et al.* [28], in contrast to the *phase polynomial* approach [29, 26]. We give a brief overview of their algorithm here.

Recall that the T gate can be written as the following sum of Pauli gates:

$$T := \frac{1 + e^{i\pi/4}}{2} I + \frac{1 - e^{i\pi/4}}{2} Z.$$

Following [27] we write $R(P)$, where P is an n-qubit Pauli operator possibly with a phase, for the Pauli sum

$$R(P) := \frac{1 + e^{i\pi/4}}{2} I_n + \frac{1 - e^{i\pi/4}}{2} P$$

As Clifford gates permute Pauli operators by conjugation, it can be observed that the following commutation rules hold for any Clifford gate C and Paulis P, P':

$$CR(P) = R(CPC^\dagger)C \tag{1}$$

$$PP' = P'P \implies R(P)R(P') = R(P')R(P) \tag{2}$$

Moreover, since the following equations hold:

$$R(P)R(P) = R(P)^2 \tag{3}$$

$$R(P)R(-P) = I \tag{4}$$

T gates can be merged by repeatedly applying commutations 1 and 2 and merging with any rotations satisfying 3 or 4.

The `staq` implementation provides a number of extensions to Fang and Chen's algorithm – notably, to handle X-, Y-, and Z-axis rotations *of any angle* natively, and to allow arbitrary gates outside of the Clifford+$\{R_X, R_Y, R_Z\}$ gate set. This allows the optimization to be performed directly over arbitrary QASM programs, and further adds additional applicability to NISQ-style circuits which use rotations of general angles in all Pauli axes. To extend Fang and Chen's algorithm in these ways we write $R(\theta, P)$ for a rotation of angle θ around the Pauli P, that is

$$R(\theta, P) := \frac{1 + e^{i\theta}}{2} I_n + \frac{1 - e^{i\theta}}{2} P,$$

and then extend the commutation rules to

$$CR(\theta, P) = R(\theta, CPC^\dagger)C \tag{5}$$

$$PP' = P'P \implies R(\theta, P)R(\theta', P') = R(\theta', P')R(\theta, P) \tag{6}$$

$$U \text{ and } P \text{ act non-trivially on disjoint qubits} \implies UR(\theta, P) = R(\theta, P)U \tag{7}$$

where U refers to an arbitrary unitary gate. The merging equations are likewise extended to

$$R(\theta, P)R(\theta', P) = R(\theta + \theta', P) \tag{8}$$

$$R(\theta, P)R(\theta', -P) = e^{i\theta'} R(\theta - \theta', P) \tag{9}$$

which are both easy to verify.

Internally, `staq` provides a library for working with circuits in the Pauli sum representation, with classes for generic Clifford operators, Pauli rotations and uninterpreted gates, and methods for testing commutations and rotation merging. The `rotation folding` optimization is implemented as a Visitor on the AST which builds a representation of each basic block (i.e. gate bodies and the main program body) as a circuit in the Pauli sum representation and applies the above equations to determine which gates can be merged or cancelled. Figure 3 shows an example of `staq`'s rotation folding optimization. Our implementation can also optionally be configured to ignore irrelevant global phases.

[4]While T-count optimization itself is not particularly important in the context of NISQ programs, it has been shown [25, 26] that reducing the number of T and other phase gates can frequently lead to significant CNOT reductions, which are much more costly in a NISQ setting.

```
OPENQASM 2.0;
include "qelib1.inc";

qreg q[3];

t q[0];
t q[0];

rx(0.3) q[1];
h q[1];
rz(-0.2) q[1];
h q[1];
```

(a) Before rotation folding

```
OPENQASM 2.0;
include "qelib1.inc";

qreg q[3];

s q[0];

h q[1];
rz(0.1) q[1];
h q[1];
```

(b) After rotation folding

Figure 3: Folding rotation gates

CNOT resynthesis In contrast to fault tolerant quantum computing, where small-angle R_Z gates dominate the cost of quantum programs, in NISQ settings it is preferable to reduce the number of two-qubit gates, typically the CNOT gate. For these purposes `staq` includes an optimization pass – `CNOT resynthesis` – to reduce the number of `cx` (i.e. CNOT) gates in a program by performing the CNOT-dihedral algorithm `Gray-synth` of Amy, Azimzadeh, and Mosca [25]. Specifically, the algorithm computes CNOT-dihedral blocks – circuit blocks containing only CNOT, X, and arbitrary phase rotations – and then resynthesizes these blocks using using a Gray code inspired algorithm to construct an efficient tour of the necessary rotations. For more details on the algorithm, the reader is directed to [25].

The `CNOT resynthesis` algorithm is highly dependent on the number of R_Z gates, and hence is typically most effective when used *after* a `rotation folding` pass. In contrast to the implementation in [25], CNOT-dihedral blocks are computed greedily, and so not all foldable R_Z are found with the `CNOT resynthesis` algorithm alone. While the `CNOT resynthesis` optimization can be used to optimize `cx` gates in a device-independent way, `staq` also includes an extension of `Gray-synth`, described in Section 2.4, which performs hardware mapping simultaneously with CNOT resynthesis to achieve better CNOT counts when the connectivity of the device is known.

2.4 Hardware mapping

The NISQ era of quantum computing [3] carries with it specific hardware challenges – notably, that of efficiently *mapping* or *routing* a quantum program onto a hardware device with constrained two-qubit interactions and noisy gates. In particular, this involves (1) mapping qubits of the program to physical qubits of the device, and (2) rewriting the circuit so that all two-qubit gates act on coupled qubits, and further satisfy the direction of the coupling in the case of directed topologies. While a complete overview of the range of existing hardware mapping techniques is beyond the scope of this work, [30] provides a comprehensive review of existing permutation-based methods and performance comparisons.

The `staq` compiler performs hardware mapping in two stages – by first selecting an initial mapping from qubits of the program to physical qubits as in [31], and then adjusting each two-qubit gate to conform to the given device topology. In this section we describe the algorithms for each stage implemented in `staq`. Currently, only physical CNOT gates are supported by `staq`.

Devices Devices in the `staq` toolchain are instances of the `Device` class, which at minimum specifies a number n of qubits addressable on the device with addresses $0, \ldots, n-1$, and a digraph where each directed edge represents an admissible CNOT gate with target at the edge's endpoint. A device may additionally

specify average one- and two-qubit gate fidelities for each qubit and digraph edge, respectively, as floating-point numbers between 0 and 1. The `Device` class further contains a number of useful utilities for mapping circuits to devices with or without known fidelities; notably, the ability to retrieve the available couplings in order of decreasing fidelity, as well as fidelity-weighted shortest-path computation and approximation of minimal weighted Steiner trees – trees spanning a subset of nodes.

While at present devices are hard-coded, built-in devices include the 8- and 16-qubit Rigetti Agave and Aspen4 chips, respectively, the 20-qubit IBM Tokyo device, a generic 9-qubit square lattice and fully-connected devices for any number of qubits.

2.4.1 Initial layout generation

It is known that the efficiency of hardware mapping is highly dependent on the chosen initial placement of qubits [31]. While better gate counts can often be achieved if the mapping algorithm is allowed to modify the initial placement [32, 33], a good initial layout can reduce CNOT counts by over 50% [34].

`staq` currently implements three layout generation algorithms: `linear`, `eager`, and `best-fit`. The `linear` layout generator functions as a basic layout, whereby physical qubits are assigned in-order as virtual qubits are allocated. By contrast, the `eager` and `best-fit` algorithms attempt to generate an initial layout which has a high degree of overlap between the CNOT gates present in the program, and the couplings present in the device.

The `eager` layout generator assigns highest-fidelity couplings on a first-come, first-serve basis. In particular, when a CNOT gate is encountered in the circuit, the highest-fidelity coupling which is *compatible* with the control and target – that is, doesn't invalidated previous assignments of the control or target to physical qubits – is chosen. This strategy typically results in lower CNOT counts compared to the `linear` strategy when combined with basic swapping for CNOT mapping.

To generate a better initial layout for CNOT mapping algorithms which are *not* based on local qubit swapping, an additional `best-fit` layout generation algorithm is implemented in `staq`. The `best-fit` algorithm, in contrast to the `linear` and `eager` strategies first scans the entire program before assigning physical qubits to virtual ones. In particular, it builds a histogram of couplings between virtual qubits, assigning the highest-fidelity couplings to virtual qubits with the most CNOT gates between them. Experimentally, we found that such an initial layout works best when qubits are not permuted intermittently by the CNOT mapping algorithm.

To illustrate the different initial placement approaches, Figure 4 gives an example of each layout generation algorithm applied to a circuit for a simple square lattice shown in Figure 4b.

2.4.2 CNOT mapping

The problem of mapping two-qubit gates to a topologically constrained architecture has received a great deal of attention recently [34, 33, 35, 31, 32, 30]. Most common techniques (e.g., the IBM-QX contest-winning [32]) rely on inserting swap gates, or more generally permutations, so that the a given two-qubit gate or set of gates satisfies the device topology. Figure 5 shows an example of this technique.

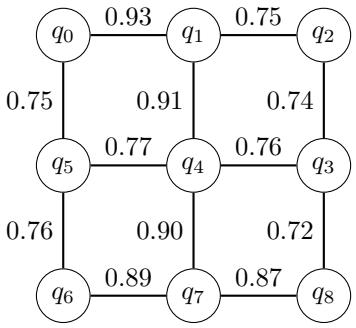
`staq` implements a version of permutation-based mapping (`swap`) where for a CNOT gate between uncoupled qubits, the endpoints are swapped along the shortest (weighted) path in the coupling graph until they are adjacent. In the case of directed edges, Hadamard gates are inserted to flip the control and target of a CNOT. Rather than swap the qubits back to their original position as in Figure 5, the resulting permutation is propagated through the rest of the circuit.

Steiner tree mapping An alternative to permutation-based mappings which has recently been gaining popularity is *topologically-constrained synthesis* [34]. With this technique, a circuit or subcircuit is re-synthesized using circuit synthesis techniques that *directly* account for the topology of the intended architecture. For circuits consisting solely of CNOT gates, [34] and [35] simultaneously developed methods of synthesizing efficient circuits satisfying a given topology by performing *constrained Gaussian elimination*, whereby the rows which can be added to one another, corresponding to qubits, are restricted by the

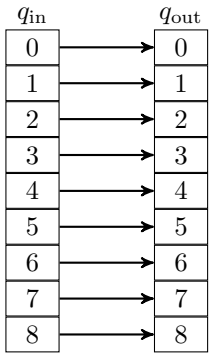
```
OPENQASM 2.0;

qreg q[9];
CX q[2],q[1];
CX q[2],q[1];
CX q[6],q[8];
CX q[7],q[3];
CX q[7],q[3];
CX q[7],q[3];
CX q[5],q[7];
CX q[5],q[7];
CX q[5],q[7];
CX q[5],q[7];
CX q[4],q[0];
```

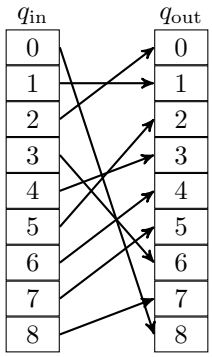
(a) Initial circuit



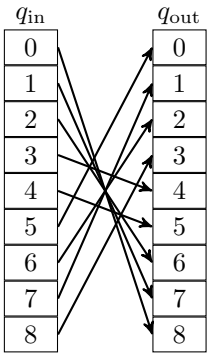
(b) Device topology & fidelities



(c) Linear layout



(d) Eager layout



(e) Best-fit layout

Figure 4: Laying out a circuit on a 9-qubit square lattice

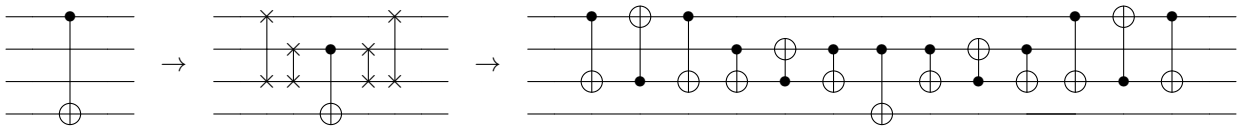
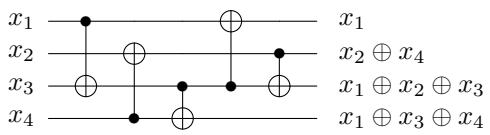


Figure 5: Mapping a CNOT gate via local swaps to a device with couplings $(0,2),(1,2),(1,3)$



(a) A circuit over CNOT gates

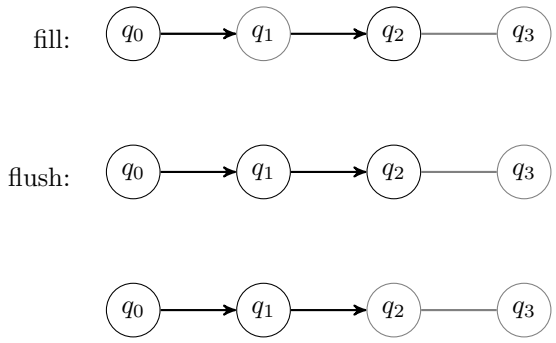
$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \begin{bmatrix} x_1 \\ x_2 \oplus x_4 \\ x_1 \oplus x_2 \oplus x_3 \\ x_1 \oplus x_3 \oplus x_4 \end{bmatrix}$$

(b) The corresponding binary matrix

Figure 6: Linear reversible circuits

underlying architecture. These results show that in the case of CNOT – or linear reversible circuits – constrained Gaussian elimination typically results in lower CNOT counts than permutation-based techniques [34]. Both results further sketch extensions to the topologically-constrained synthesis of CNOT-dihedral circuits. Along with the `swap` mapping algorithm, `staq` includes a mapping algorithm (`steiner`) based on constrained CNOT and CNOT-dihedral synthesis in the style of [34] and [35]. We give a brief overview of the `steiner` mapper here.

The standard method[5] of synthesizing an n-qubit CNOT circuit, is to perform Gaussian elimination on the $n \times n$ binary matrix giving the classical function (see Figure 6) and reverse the row operations, corresponding to CNOT gates. When the hardware topology is constrained however, it may not be possible to "zero-out" all the off-diagonal entries of a column by adding the pivot row to them directly. Instead, a path in the coupling graph from the pivot to each row with a leading 1 may be used by first *filling* in 1's along the path by applying CNOT gates, then *flushing* by applying CNOT gates along the path in reverse. For example, with the "straight-line" topology, the 1 in entry $(2, 0)$ of the matrix in Figure 6b can be eliminated by first filling in 1's along the shortest path from 0 to 2, then removing them in reverse:



$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 \end{bmatrix} \xrightarrow{R_1 := R_0 \oplus R_1} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 \end{bmatrix}$$

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 \end{bmatrix} \xrightarrow{R_2 := R_1 \oplus R_2} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 \\ 1 & 0 & 1 & 1 \end{bmatrix}$$

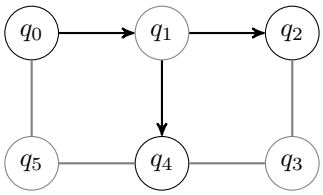
$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 \\ 1 & 0 & 1 & 1 \end{bmatrix} \xrightarrow{R_1 := R_0 \oplus R_1} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 \\ 1 & 0 & 1 & 1 \end{bmatrix}$$

To zero-out all the leading 1's, excluding the pivot, a *tree* with root at the pivot qubit and endpoints at all rows with a leading 1 can be used instead. As noted in [34, 35], computing a minimal such tree is the well-known *Steiner tree problem*, which is NP-hard in general but admits effective polynomial-time approximations, notably via all-pairs-shortest-paths and minimal spanning tree algorithms. Zeroing all non-pivot rows for a given column proceeds similarly, by first filling 1's into every node of the tree by adding rows along every edge leading to a 0 – called *Steiner points* – then zero-ing all non-root nodes by traversing the tree and adding rows along edges in reverse. An example is given below with a minimal tree spanning

[5] With a small modification, this method is in fact asymptotically optimal [36].

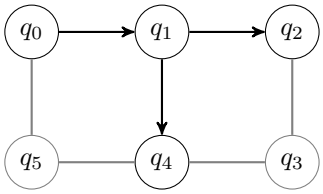
$\{q_0, q_2, q_4\}$ with edges shown in bold:

fill:

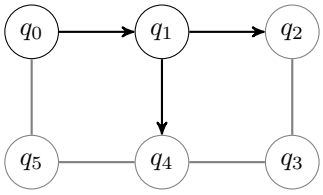


$$\begin{bmatrix} 1 & 0 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 0 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \xrightarrow{R_1 := R_0 \oplus R_1} \begin{bmatrix} 1 & 0 & 0 & 1 & 0 & 1 \\ 1 & 1 & 0 & 1 & 1 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

flush:

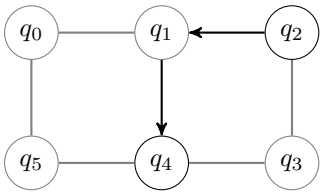


$$\begin{bmatrix} 1 & 0 & 0 & 1 & 0 & 1 \\ 1 & 1 & 0 & 1 & 1 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \xrightarrow{\substack{R_2 := R_1 \oplus R_2 \\ R_4 := R_1 \oplus R_4}} \begin{bmatrix} 1 & 0 & 0 & 1 & 0 & 1 \\ 1 & 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$



$$\begin{bmatrix} 1 & 0 & 0 & 1 & 0 & 1 \\ 1 & 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \xrightarrow{R_1 := R_0 \oplus R_1} \begin{bmatrix} 1 & 0 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

Crossing the diagonal The "fill-then-flush" method may fail when the computed tree contains nodes *above* the diagonal, as this may propagate 1's to the left of the current column, as in the following example:



$$\begin{bmatrix} 1 & 0 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \xrightarrow{\substack{R_1 := R_1 \oplus R_2 \\ R_4 := R_1 \oplus R_1 \\ R_1 := R_1 \oplus R_2}} \begin{bmatrix} 1 & 0 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

In [34], the above-diagonal dependencies are handled by ordering the rows according to a *Hamiltonian path* in the graph (if it exists), so that the matrix can be reduced to echelon form without crossing the diagonal. As not all possible topologies admit a Hamiltonian path – and in general computing one is an NP-hard problem – they also give a recursive method which works for arbitrary graphs. In contrast, [35] doesn't assume the existence of a Hamiltonian path and instead uses an *uncompute* stage to effectively "undo" all changes to other columns.

The implementation in `staq` follows the method of [35], with the exception that *only changes to rows with (transitive) dependencies on above-diagonal rows are uncomputed.* In practice this reduces the number of CNOT gates required, as not every iteration will cross the diagonal.

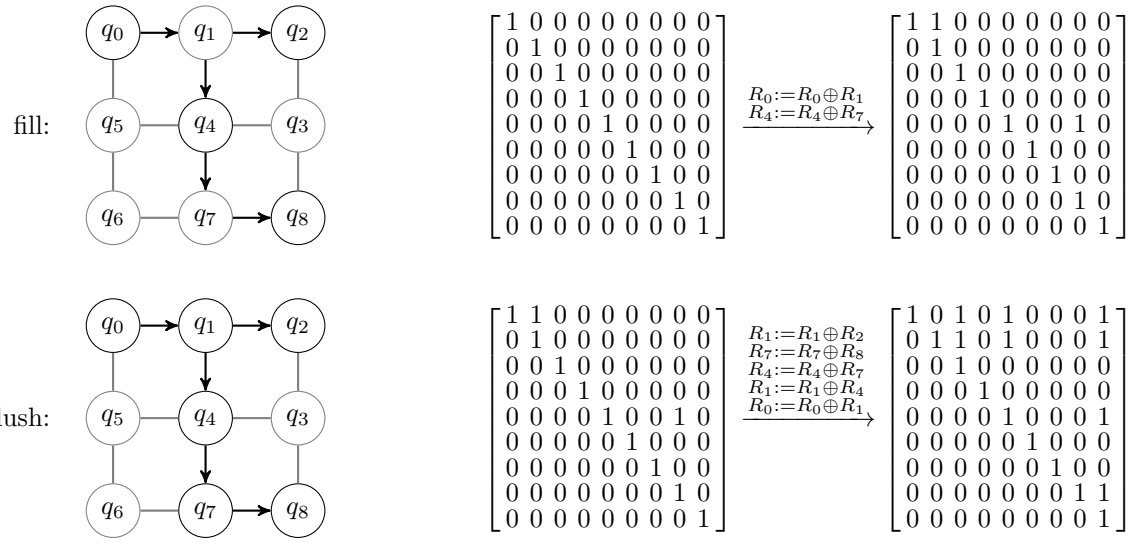
Constrained Gray-synth The `Steiner` mapping algorithm in `staq` actually implements a more general form of re-synthesis, targeting CNOT-dihedral[6] circuits by using the `Gray-synth` CNOT-optimization algorithm [25] extended with constrained Gaussian synthesis. Similar extensions were considered in [34] and [35] – by comparison `staq` contains a full-scale implementation which operates on arbitrary circuits by generating *synthesis events* whenever a non-CNOT-dihedral gate is encountered. In the case when no $R_Z(\theta)$ gates are present, the algorithm coincides with the basic constrained Gaussian synthesis.

Again, the implementation of constrained `Gray-synth` differs from those sketched in [34, 35], so we give a brief overview of our method here. The `Gray-synth` algorithm functions by ordering a given set of *parities* – corresponding to the states "being rotated on" by phase gates – of the inputs so that an efficient tour can be

[6] Circuits over $\{\text{CNOT}, X, R_Z(\theta)\}$

constructed. Moreover, this ordering is elaborated as the circuit is synthesized, by recursively partitioning and synthesizing the remaining parities and updating the remaining parities as the state is modified by the synthesized CNOT gates.

`staq` performs constrained Gray-synthesis by delaying CNOT gates *until a partition of size* 1 *is reached*. Such a partition corresponds to the computation of a parity $x_i \mapsto x_i \oplus f(x_1, \ldots, x_n)$ where f is a linear (parity) function over $\{x_{j\neq i}\}$. Again, a (Steiner) tree rooted at x_i and spanning $S = \{x_j \mid x_j \text{ appears in } f\}$ can be used to synthesize the above parity – however, in contrast to the Gaussian elimination situation, where the root is added to each leaf, *each leaf needs to be added to the root*. This is done by first adding each Steiner point (nodes in the tree but not in S) to its predecessors in breadth-first order, then adding each node to its predecessors in reverse breadth-first order. An example of this process is given below for the parity $x_0 \oplus x_1 \oplus x_2 \oplus x_4 \oplus x_8$ rooted at x_0 on a square lattice. Note that the matrix in this case gives the function computed by the series of row additions (i.e. CNOT gates).



The first row of the final linear transformation above corresponds to the function $x_0 \mapsto x_0 \oplus x_2 \oplus x_4 \oplus x_8$ as required.

Rather than uncompute the changes to the other rows, the `Gray-synth` algorithm takes the linear transformation into account when recursively partitioning the remaining parities. Once all parity computations have been completed, the algorithm computes the final linear transformation (see [25]) using regular constrained Gaussian synthesis.

2.5 Compilation

Along with the default QASM output, `staq` includes a suite of source-to-source compilers or *"transpilers"*, currently supporting output to Quil [9], ProjectQ [10], Q# [12], and Cirq [18]. Effort has been made to translate QASM code to idiomatic code in the target language as much as possible – in particular, translating `qelib1.inc` gates and gate declarations to standard library gates and idiomatic gate declarations in the target language whenever possible. Figure 7 gives an example of the Q# output for a QASM program.

`staq` also includes an option to output just the resource counts of a program. By default the resource counter un-boxes resource counts for all declared gates except for those from the standard library, but the resource counter can be configured with a list of gates to leave boxed.

2.6 Performance

To assess the performance of `staq`, we compare it against the well-known software toolkit and compiler Qiskit [19]. We chose Qiskit to compare our work over the many other existing tools as it is arguably the

largest compilation toolchain publicly available which supports the openQASM language. In particular, we compare each tool's highest standard optimization setting and default hardware mapping settings for total gate counts and CNOT counts, respectively. In particular, we compare the Qiskit transpiler's level 3 optimization against `staq`'s `-O2` command line option. The default mapping setting in `staq` applies the `steiner` mapping algorithm with the `best-fit` initial layout. For the optimization experiments, both tools unbox the program to the following subset of `qelib1.inc`:

$$\{\texttt{u3}, \texttt{cx}, \texttt{h}, \texttt{rx}, \texttt{ry}, \texttt{rz}\}.$$

We use a common benchmark suite [29, 26] to benchmark our compiler, consisting of largely reversible arithmetic and a few quantum algorithms. All experiments were run on 2.3GHz Intel Core i7 processor with 8GB of RAM running Arch Linux.

The results of optimization passes and mapping passes are reported in Tables 1 and 2, respectively. The best results in either case are identified in bold. For circuit optimization, `staq` beats Qiskit (over this gate set) for all but one of the benchmark circuits, achieving 31.7% reduction in gate counts on average compared to Qiskit's 25.9% average reduction. Similarly, in all but one benchmark circuit with the highest number of qubits, `staq` was also significantly faster. It remains a focus of future work to improve the scalability of `staq`'s optimization algorithms as the number of qubits increases.

For hardware mapping benchmarks, IBM's 20 qubit Tokyo chip was selected as the target architecture, and as such only the benchmark circuits which fit onto the chip are reported in Table 2. In contrast to optimization, the experimental results for hardware mapping were mixed. While the `Steiner` mapping algorithm with `best-fit` initial layout was consistently orders of magnitude faster than Qiskit's default mapping algorithm, Qiskit outperformed `staq` in terms of CNOT counts in the majority of cases. As hardware mapping is very sensitive to initial qubit placement [31], similar to [34] a simple *hill-climb* algorithm was implemented to optimize the initial layout and combined with the `Steiner` mapping algorithm (last two columns of Table 2). With this qubit layout optimization, `staq`'s default mapping algorithm outperforms Qiskit in the majority of cases, with an average CNOT-count increase of 103.4% compared to Qiskit's 125.7%, while still running faster than Qiskit in almost all cases. Moreover, many of the cases where Qiskit outperformed `staq` appear to be pathological cases for the underlying `Gray-synth` algorithm [25]. It remains to be seen whether more sophisticated layout optimization algorithms which avoid local minima – for instance, simulated annealing – can improve the results further.

3 Conclusions and future directions

In this article we described `staq` along with its main use cases, and provided a set of benchmarks. `staq` is a modular quantum compiling toolkit, which is easy to extend, its design being inspired by Clang [16].

The dynamic field of quantum software is evolving rapidly, being driven by a variety of factors, ranging from progress in quantum hardware to improved compilation techniques. While we cannot foresee what the future will reserve, we still have a set of QASM-based features we will most likely add to `staq`, such as: i) more QASM syntax extensions that will not break backwards compatibility, ii) ability to perform iterations and loops, iii) having registers as arguments to gates instead of qubits. Such extensions will allow the user to design quantum software libraries in a relatively straight-forward manner while focusing on efficiently achieving the desired functionality.

References

[1] Richard Feynman. Simulating physics with computers. *International Journal of Theoretical Physics*, 21(6):467–488, June 1982.

[2] Frank Arute, Kunal Arya, Ryan Babbush, Dave Bacon, Joseph C. Bardin, Rami Barends, Rupak Biswas, Sergio Boixo, Fernando G. S. L. Brandao, David A. Buell, Brian Burkett, Yu Chen, Zijun Chen,

Benchmark	n	Original		Qiskit		`staq`		
		# gates	depth	# gates	time (s)	# gates	depth	time (s)
Grover_5	9	1023	320	769	9.64	**673**	203	0.203
Mod 5_4	5	79	59	60	2.605	**53**	37	0.029
VBE-Adder_3	10	190	63	134	3.421	**105**	38	0.031
CSLA-MUX_3	15	210	37	160	3.638	**158**	28	0.039
CSUM-MUX_9	30	532	38	420	6.132	**336**	26	0.091
QCLA-Com_7	24	659	81	416	6.287	**341**	51	0.095
QCLA-Mod_7	26	1120	137	838	10.888	**740**	97	0.246
QCLA-Adder_10	36	657	61	494	7.069	**443**	40	0.128
Adder_8	24	1128	157	869	11.485	**751**	113	0.254
RC-Adder_6	14	244	45	185	3.977	**173**	34	0.055
Mod-Red_21	11	346	87	261	4.618	**241**	60	0.055
Mod-Mult_55	9	147	37	117	3.146	**109**	29	0.036
Mod-Adder_1024	28	5425	1188	3871	52.066	**3335**	793	1.039
GF(2^4)-Mult	12	289	54	213	4.256	**203**	37	0.079
GF(2^5)-Mult	15	447	66	327	5.322	**317**	44	0.097
GF(2^6)-Mult	18	639	78	**357**	7.039	444	52	0.163
GF(2^7)-Mult	21	865	90	627	8.906	**606**	59	0.29
GF(2^8)-Mult	24	1139	106	819	11.018	**791**	71	0.513
GF(2^9)-Mult	27	1419	114	1023	13.551	**987**	74	0.782
GF(2^{10})-Mult	30	1747	126	1257	16.386	**1202**	82	1.15
GF(2^{16})-Mult	48	4459	202	3179	42.435	**3059**	131	10.324
GF(2^{32})-Mult	96	17658	396	12536	4m	**12042**	253	5m
Ham_15 (low)	17	535	165	425	6.689	**391**	108	0.111
Ham_15 (med)	17	1599	528	1154	15.027	**994**	344	0.241
Ham_15 (high)	20	6712	2244	4768	1m	**3982**	1448	1.395
HWB_6	7	319	93	248	16.151	**232**	68	0.056
HWB_8	12	18220	3102	14059	5m	**12827**	2257	2.939
QFT_4	5	187	117	161	3.927	**178**	117	0.040
$\Lambda_3(X)$	5	57	34	44	2.531	**40**	26	0.027
$\Lambda_3(X)$ (Barenco)	5	76	40	56	2.955	**50**	32	0.024
$\Lambda_4(X)$	7	95	40	73	2.92	**72**	28	0.028
$\Lambda_4(X)$ (Barenco)	7	146	64	109	3.178	**98**	44	0.034
$\Lambda_5(X)$	9	133	40	101	3.148	**90**	28	0.035
$\Lambda_5(X)$ (Barenco)	9	218	76	162	3.792	**146**	54	0.035
$\Lambda_{10}(X)$	19	323	40	247	4.741	**215**	28	0.05
$\Lambda_{10}(X)$ (Barenco)	19	578	76	427	6.545	**386**	54	0.077
Average reduction (%)				25.9		**31.7**		

Table 1: Benchmark optimization results

Benchmark	n	Original	Qiskit		`staq`		`staq` (w/ layout opt.)	
		# CNOTs	# CNOTs	time (s)	# CNOTs	time (s)	# CNOTs	time (s)
Grover_5	9	288	703	31.226	592	0.204	**422**	5.763
Mod 5_4	5	28	58	4.040	43	0.037	**40**	0.084
VBE-Adder_3	10	70	134	7.770	78	0.039	**68**	0.968
CSLA-MUX_3	15	80	208	8.419	240	0.055	**177**	5.202
RC-Adder_6	14	93	239	9.307	**150**	0.071	151	3.514
Mod-Red_21	11	105	243	10.766	280	0.081	**187**	2.777
Mod-Mult_55	9	48	123	6.013	142	0.043	**109**	1.01
GF(2^4)-Mult	12	99	**284**	10.483	422	0.077	337	2.495
GF(2^5)-Mult	15	154	**472**	17.922	589	0.126	478	6.599
GF(2^6)-Mult	18	221	**749**	26.492	851	0.196	765	15.847
Ham_15 (low)	17	236	**794**	26.806	887	0.146	887	3.605
Ham_15 (med)	17	534	1630	1m	**1406**	0.330	**1406**	9.934
Ham_15 (high)	20	2149	5852	4m	**5512**	1.466	**5512**	55.176
HWB_6	7	116	**237**	11.427	259	0.074	259	0.430
HWB_8	12	7129	19650	9m	22687	4.303	**17428**	5m
QFT_4	5	46	**72**	6.271	100	0.058	90	0.284
$\Lambda_3(X)$	5	18	**18**	3.157	32	0.027	30	0.091
$\Lambda_3(X)$ (Barenco)	5	24	**24**	3.446	35	0.037	35	0.076
$\Lambda_4(X)$	7	30	**43**	4.218	61	0.035	47	0.359
$\Lambda_4(X)$ (Barenco)	7	48	**48**	3.351	73	0.036	58	0.417
$\Lambda_5(X)$	9	42	86	5.448	101	0.036	**70**	1.041
$\Lambda_5(X)$ (Barenco)	9	72	126	6.469	150	0.046	**102**	1.654
$\Lambda_{10}(X)$	19	102	224	12.916	238	0.078	**134**	24.583
$\Lambda_{10}(X)$ (Barenco)	19	192	470	13.832	404	0.117	**231**	26.959
Average increase (%)			125.7		146.4		**103.4**	

Table 2: Hardware mapping results

Ben Chiaro, Roberto Collins, William Courtney, Andrew Dunsworth, Edward Farhi, Brooks Foxen, Austin Fowler, Craig Gidney, Marissa Giustina, Rob Graff, Keith Guerin, Steve Habegger, Matthew P. Harrigan, Michael J. Hartmann, Alan Ho, Markus Hoffmann, Trent Huang, Travis S. Humble, Sergei V. Isakov, Evan Jeffrey, Zhang Jiang, Dvir Kafri, Kostyantyn Kechedzhi, Julian Kelly, Paul V. Klimov, Sergey Knysh, Alexander Korotkov, Fedor Kostritsa, David Landhuis, Mike Lindmark, Erik Lucero, Dmitry Lyakh, Salvatore Mandrà, Jarrod R. McClean, Matthew McEwen, Anthony Megrant, Xiao Mi, Kristel Michielsen, Masoud Mohseni, Josh Mutus, Ofer Naaman, Matthew Neeley, Charles Neill, Murphy Yuezhen Niu, Eric Ostby, Andre Petukhov, John C. Platt, Chris Quintana, Eleanor G. Rieffel, Pedram Roushan, Nicholas C. Rubin, Daniel Sank, Kevin J. Satzinger, Vadim Smelyanskiy, Kevin J. Sung, Matthew D. Trevithick, Amit Vainsencher, Benjamin Villalonga, Theodore White, Z. Jamie Yao, Ping Yeh, Adam Zalcman, Hartmut Neven, and John M. Martinis. Quantum supremacy using a programmable superconducting processor. *Nature*, 574(7779):505–510, 2019.

[3] John Preskill. Quantum Computing in the NISQ Era and Beyond. *Quantum*, 2:79, August 2018.

[4] Robert Raussendorf and Hans J. Briegel. A one-way quantum computer. *Phys. Rev. Lett.*, 86(22):5188–5191, 2001.

[5] Tameem Albash and Daniel A. Lidar. Adiabatic quantum computation. *Rev. Mod. Phys.*, 90:015002, Jan 2018.

[6] Michael A. Nielsen and Isaac L. Chuang. *Quantum Computation and Quantum Information*. Cambridge University Press, Cambridge, 5th edition, 2000.

[7] Alexander S. Green, Peter LeFanu Lumsdaine, Neil J. Ross, Peter Selinger, and Benoît Valiron. Quipper: A Scalable Quantum Programming Language. In *Proceedings of the 34th ACM SIGPLAN conference on Programming Language Design and Implementation*, PLDI '13, pages 333–342, 2013.

[8] Ali JavadiAbhari, Shruti Patil, Daniel Kudrow, Jeff Heckey, Alexey Lvov, Frederic T. Chong, and Margaret Martonosi. ScaffCC: Scalable Compilation and Analysis of Quantum Programs. *Parallel Computing*, 45(C):2–17, 2015.

[9] Robert S. Smith, Michael J. Curtis, and William J. Zeng. A Practical Quantum Instruction Set Architecture. *arXiv preprint*, 2016.

[10] Damian S. Steiger, Thomas Häner, and Matthias Troyer. ProjectQ: An Open Source Software Framework for Quantum Computing. *Quantum*, 2:49, 2018.

[11] Shusen Liu, Xin Wang, Li Zhou, Ji Guan, Yinan Li, Yang He, Runyao Duan, and Mingsheng Ying. $Q|SI\rangle$: A Quantum Programming Environment. *arXiv preprint*, 2017.

[12] Krysta Svore, Alan Geller, Matthias Troyer, John Azariah, Christopher Granade, Bettina Heim, Vadym Kliuchnikov, Mariia Mykhailova, Andres Paz, and Martin Roetteler. *Q#*: Enabling Scalable Quantum Computing and Development with a High-level DSL. In *Proceedings of the 3rd ACM International Workshop on Real World Domain Specific Languages*, RWDSL '18, pages 7:1–7:10, 2018.

[13] Nathan Killoran, Josh Izaac, Nicolás Quesada, Ville Bergholm, Matthew Amy, and Christian Weedbrook. Strawberry Fields: A Software Platform for Photonic Quantum Computing. *Quantum*, 3:129, March 2019.

[14] Dennis M. Ritchie and Ken Thompson. The unix time-sharing system. *Commun. ACM*, 17(7):365–375, July 1974.

[15] Andrew W. Cross, Lev S. Bishop, John A. Smolin, and Jay M. Gambetta. Open Quantum Assembly Language. arXiv preprint, 2017.

[16] Clang: a C language family frontend for LLVM.

[17] Ieee standard for verilog hardware description language. *IEEE Std 1364-2005 (Revision of IEEE Std 1364-2001)*, pages 1–590, April 2006.

[18] Cirq: A python framework for creating, editing, and invoking Noisy Intermediate Scale Quantum (NISQ) circuits.

[19] Héctor Abraham, Ismail Yunus Akhalwaya, Gadi Aleksandrowicz, Thomas Alexander, Gadi Alexandrowics, Eli Arbel, Abraham Asfaw, Carlos Azaustre, Panagiotis Barkoutsos, George Barron, Luciano Bello, Yael Ben-Haim, Daniel Bevenius, Lev S. Bishop, Samuel Bosch, David Bucher, CZ, Fran Cabrera, Padraic Calpin, Lauren Capelluto, Jorge Carballo, Ginés Carrascal, Adrian Chen, Chun-Fu Chen, Richard Chen, Jerry M. Chow, Christian Claus, Christian Clauss, Abigail J. Cross, Andrew W. Cross, Juan Cruz-Benito, Cryoris, Chris Culver, Antonio D. Córcoles-Gonzales, Sean Dague, Matthieu Dartiailh, Abdón Rodríguez Davila, Delton Ding, Eugene Dumitrescu, Karel Dumon, Ivan Duran, Pieter Eendebak, Daniel Egger, Mark Everitt, Paco Martín Fernández, Albert Frisch, Andreas Fuhrer, IAN GOULD, Julien Gacon, Gadi, Borja Godoy Gago, Jay M. Gambetta, Luis Garcia, Shelly Garion, Gawel-Kus, Juan Gomez-Mosquera, Salvador de la Puente González, Donny Greenberg, John A. Gunnels, Isabel Haide, Ikko Hamamura, Vojtech Havlicek, Joe Hellmers, Łukasz Herok, Hiroshi Horii, Connor Howington, Shaohan Hu, Wei Hu, Haruki Imai, Takashi Imamichi, Raban Iten, Toshinari Itoko, Ali Javadi-Abhari, Jessica, Kiran Johns, Naoki Kanazawa, Anton Karazeev, Paul Kassebaum, Arseny Kovyrshin, Vivek Krishnan, Kevin Krsulich, Gawel Kus, Ryan LaRose, Raphaël Lambert, Joe Latone, Scott Lawrence, Dennis Liu, Peng Liu, Panagiotis Barkoutsos ZRL Mac, Yunho Maeng, Aleksei Malyshev, Jakub Marecek, Manoel Marques, Dolph Mathews, Atsushi Matsuo, Douglas T. McClure, Cameron McGarry, David McKay, Srujan Meesala, Antonio Mezzacapo, Rohit Midha, Zlatko Minev, Michael Duane Mooring, Renier Morales, Niall Moran, Prakash Murali, Jan Müggenburg, David Nadlinger, Giacomo Nannicini, Paul Nation, Yehuda Naveh, Nick-Singstock, Pradeep Niroula, Hassi Norlen, Lee James O'Riordan, Pauline Ollitrault, Steven Oud, Dan Padilha, Hanhee Paik, Simone Perriello, Anna Phan, Marco Pistoia, Alejandro Pozas-iKerstjens, Viktor Prutyanov, Jesús Pérez, Quintiii, Rudy Raymond, Rafael Martín-Cuevas Redondo, Max Reuter, Diego M. Rodríguez, Mingi Ryu, Martin Sandberg, Ninad Sathaye, Bruno Schmitt, Chris Schnabel, Travis L. Scholten, Eddie Schoute, Ismael Faro Sertage, Nathan Shammah, Yunong Shi, Adenilton Silva, Yukio Siraichi, Seyon Sivarajah, John A. Smolin, Mathias Soeken, Dominik Steenken, Matt Stypulkoski, Hitomi Takahashi, Charles Taylor, Pete Taylour, Soolu Thomas, Mathieu Tillet, Maddy Tod, Enrique de la Torre, Kenso Trabing, Matthew Treinish, TrishaPe, Wes Turner, Yotam Vaknin, Carmen Recio Valcarce, Francois Varchon, Desiree Vogt-Lee, Christophe Vuillot, James Weaver, Rafal Wieczorek, Jonathan A. Wildstrom, Robert Wille, Erick Winston, Jack J. Woehr, Stefan Woerner, Ryan Woo, Christopher J. Wood, Ryan Wood, Stephen Wood, James Wootton, Daniyar Yeralin, Jessie Yu, Laura Zdanski, Zoufalc, anedumla, azulehner, bcamorrison, brandhsn, dennis-liu 1, drholmie, elfrocampeador, fanizzamarco, gruu, kanejess, klinvill, lerongil, ma5x, merav aharoni, mrossinek, ordmoj, strickroman, tigerjack, yang.luh, and yotamvakninibm. Qiskit: An open-source framework for quantum computing, 2019.

[20] Matthew Amy. Sized Types for Low-Level Quantum Metaprogramming. In Michael Kirkedal Thomsen and Mathias Soeken, editors, *Reversible Computation*, pages 87–107, Cham, 2019. Springer International Publishing.

[21] Mathias Soeken, Heinz Riener, Winston Haaswijk, Eleonora Testa, Bruno Schmitt, Giulia Meuli, Fereshte Mozafari, and Giovanni De Micheli. The EPFL logic synthesis libraries, November 2019. arXiv:1805.05121v2.

[22] D.E. Thomas and P.R. Moorby. *The Verilog® Hardware Description Language*. Springer US, 2013.

[23] Mathias Soeken, Martin Roetteler, Nathan Wiebe, and Giovanni De Micheli. LUT-Based Hierarchical Reversible Logic Synthesis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 38(9):1675–1688, 2019.

[24] Matthew Amy. Towards Large-Scale Functional Verification of Universal Quantum Circuits. In *Proceedings of the 15th International Conference on Quantum Physics and Logic*, QPL '18, 2018.

[25] Matthew Amy, Parsiad Azimzadeh, and Michele Mosca. On the CNOT-complexity of CNOT-PHASE Circuits. *Quantum Science and Technology*, 4(1):015002, 2018.

[26] Yunseong Nam, Neil J. Ross, Yuan Su, Andrew M. Childs, and Dmitri Maslov. Automated Optimization of Large Quantum Circuits with Continuous Parameters. *npj Quantum Information*, 4(1):23, 2018.

[27] Fang Zhang and Jianxin Chen. Optimizing T gates in Clifford+T circuit as $\pi/4$ rotations around Paulis. arXiv preprint, 2019.

[28] David Gosset, Vadym Kliuchnikov, Michele Mosca, and Vincent Russo. An Algorithm for the T-count. *Quantum Information and Computation*, 14(15-16):1261–1276, November 2014.

[29] Matthew Amy, Dmitri Maslov, and Michele Mosca. Polynomial-Time T-depth optimization of Clifford+T circuits via matroid partitioning. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 33(10):1476–1489, 2014.

[30] Alexander Cowtan, Silas Dilkes, Ross Duncan, Alexandre Krajenbrink, Will Simmons, and Seyon Sivarajah. On the Qubit Routing Problem. In Wim van Dam and Laura Mancinska, editors, *14th Conference on the Theory of Quantum Computation, Communication and Cryptography (TQC 2019)*, volume 135 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 5:1–5:32, Dagstuhl, Germany, 2019. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[31] Alexandru Paler. On the influence of initial qubit placement during nisq circuit compilation. In Sebastian Feld and Claudia Linnhoff-Popien, editors, *Quantum Technology and Optimization Problems*, pages 207–217, Cham, 2019. Springer International Publishing.

[32] A. Zulehner, A. Paler, and R. Wille. An Efficient Methodology for Mapping Quantum Circuits to the IBM QX Architectures. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 38(7):1226–1236, 2019.

[33] Gushu Li, Yufei Ding, and Yuan Xie. Tackling the qubit mapping problem for nisq-era quantum devices. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, pages 1001–1014, New York, NY, USA, 2019. ACM.

[34] Aleks Kissinger and Arianne Meijer-van de Griend. CNOT circuit extraction for topologically-constrained quantum memories. arXiv preprint, 2019.

[35] Beatrice Nash, Vlad Gheorghiu, and Michele Mosca. Quantum circuit optimizations for NISQ architectures. arXiv preprint, 2019.

[36] Ketan N. Patel, Igor L. Markov, and John P. Hayes. Optimal Synthesis of Linear Reversible Circuits. *Quantum Information & Computation*, 8(3):282–294, 2008.

A Transpiler example

```
OPENQASM 2.0;
include "qelib1.inc";

gate bell x,y {
  h x;
  cx x,y;
}

qreg q[1];
qreg anc[2];
creg c0[1];
creg c1[1];

bell anc[0],anc[1];
cx q,anc[0];
h q;
measure q -> c0[0];
measure anc[0] -> c1[0];
if(c0==1) z anc[1];
if(c1==1) x anc[1];
```

(a) QASM source

```
namespace Quantum.staq {
    open Microsoft.Quantum.Intrinsic;
    open Microsoft.Quantum.Convert;
    open Microsoft.Quantum.Canon;
    open Microsoft.Quantum.Math;

    operation U(theta : Double, phi : Double, lambda : Double, q : Qubit) : Unit {
        Rz(lambda, q);
        Ry(theta, q);
        Rz(phi, q);
    }
    operation u3(theta : Double, phi : Double, lambda : Double, q : Qubit) : Unit {
        U(theta, phi, lambda, q);
    }
    operation u2(phi : Double, lambda : Double, q : Qubit) : Unit {
        U(PI()/2.0, phi, lambda, q);
    }
    operation u0(gamma : Double, q : Qubit) : Unit {
        U(0.0, 0.0, 0.0, q);
    }
    operation cy(a : Qubit, b : Qubit) : Unit {
        (Adjoint S)(b);
        CNOT(a, b);
        S(b);
    }
    operation swap(a : Qubit, b : Qubit) : Unit {
        CNOT(a, b);
        CNOT(b, a);
        CNOT(a, b);
    }
    operation cu3(theta : Double, phi : Double, lambda : Double, c : Qubit, t : Qubit) : Unit {
        Rz((lambda-phi)/2.0, t);
        CNOT(c, t);
        u3(-(theta/2.0), 0.0, -((phi+lambda)/2.0), t);
        CNOT(c, t);
        u3(theta/2.0, phi, 0.0, t);
    }

    operation bell(x : Qubit, y : Qubit) : Unit {
        H(x);
        CNOT(x, y);
    }

    operation Circuit() : Unit {
        using (q = Qubit[1]) {
            using (anc = Qubit[2]) {
                mutable c0 = new Result[1];
                mutable c1 = new Result[1];
                bell(anc[0], anc[1]);
                CNOT(q[0], anc[0]);
                H(q[0]);
                set c0 w/= 0 <- M(q[0]);
                set c1 w/= 0 <- M(anc[0]);
                if (ResultArrayAsInt(c0) == 1) {
                    Z(anc[1]);
                }
                if (ResultArrayAsInt(c1) == 1) {
                    X(anc[1]);
                }

                ResetAll(anc);
            }
            ResetAll(q);
        }
    }
}
```

(b) Translated Q# code

Figure 7: Translation between QASM and Q#