8 Quantum Languages, Compilers, and Tools

At this point, we understand the principles of quantum computing, the important foundational algorithms, and the basics of quantum error correction. We have developed a compact and reasonably fast infrastructure for exploration and experimentation with the presented material.

The infrastructure is working but still far away from enabling high programmer productivity. Composing algorithms is labor-intensive and error prone. Circuits with maybe 10^6 gates are supported, but some algorithms may require trillions of gates with orders of magnitude more qubits.

In classical computing, programs are being constructed at much higher levels of abstraction, which allows the targeting of several general-purpose architectures in a portable way. On a high-performance CPU, programs execute billions of instructions per second on a single core. Building quantum programs at that scale with a flat programming model like QASM stitching together individual gates does not scale. It is the equivalent of programming today's machines in assembly language, without looping constructs.

There are parallels to the 1950s, where assembly language was the trade of the day to program early computers. Just as FORTRAN emerged as one of the first compiled programming language and enabled major productivity gains, there are similar attempts today, trying to raise the abstraction level in the area of quantum computing.

In this chapter we discuss a representative cross-section of quantum programming languages, and briefly touch on tooling, such as simulators or entanglement analysis. There is also a discussion on quantum compiler optimizations, a fascinating topic with unique challenges. We write this chapter with the understanding that comparisons between toolchains are necessarily incomplete but nonetheless educational.

Section 8.5 on transpilation finishes the chapter; this is a powerful technique with many uses. It allows seamless porting of our circuits to other frameworks. This enables direct comparisons and the use of specific features of these platforms, such as advanced error models or distributed simulation. Transpilation can be used to produce circuit diagrams or LaTeX source code. The underlying compiler technology further enables implementation of several of the features found in various programming languages, such as uncomputation, entanglement analysis, and conditional blocks.

https://doi.org/10.1017/9781009099974.010 Published online by Cambridge University Press

8.1 Challenges for Quantum Compilation

Quantum computing poses unique challenges for compiler design. In this section, we provide a brief overview of some of the main challenges. The next sections discuss additional details and proposed solutions.

Quantum computing needs a programming model – what will run, how, when, and where? Unlike classical coprocessors, such as graphics processing units (GPUs), quantum computers will not offer general-purpose functionality similar to a CPU. Instead, a classical machine will entirely control the quantum computer. A model called *QRAM* was proposed for this early in the history of quantum computing. We will discuss this model in Section 8.2.

A key question is how realistic this idealized model is, or can be. Quantum circuits operate at micro-Kelvin temperatures. It will be a challenge for standard CPU manufacturing processes to operate at this temperature, even though progress has been made (Patra et al., 2020). The CPU could alternatively operate away from the quantum circuit, but the bandwidth between classical and quantum circuits may be severely limited. Current work is presented in Xue et al. (2021).

Constructing quantum circuits gate by gate is tedious and prone to error. There are additional challenges, such as the no-cloning theorem and the need for automatic error correction. Programming languages offer a higher level of abstraction and will be essential for programmer productivity. But what is the *right* level of abstraction? We sample several existing approaches to quantum programming languages in Section 8.3. Compiler construction and intermediate representation (IR) design are challenges by themselves; it is apparent that a flat, QASM-like, linked-list IR will not scale to programs with trillions of gates.

The required precision of gates is an important design parameter. We will have to approximate certain unitaries by sequences of existing, physical gates, but this introduces inaccuracies and noise. Some algorithms are robust against noise, others not at all. The toolchain plays an essential role in this area as well.

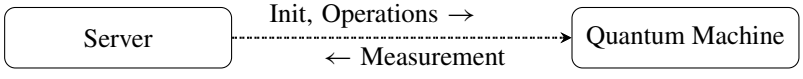
Aspects of dynamic code generation may become necessary, for example, to approximate specific rotations dynamically or to reduce noise (Wilson et al., 2020). There are challenges in fast gate approximation, compile time, accuracy, and optimality of the approximating gate sequences. To give a taste of these problems, we already detailed the Solovay–Kitaev algorithm in Section 6.15.

Compiler optimization has a novel set of transformations to consider in an exponentially growing search space. We are currently in the era of physical machines with 50–100 physical qubits, the *Noisy Intermediate-Scale Quantum Computers* (Preskill, 2018). Future systems will have more qubits and qubits with likely different characteristics than today's qubits. Compiler optimizations and code generation techniques will have to evolve accordingly as well. We discuss several optimization techniques in Section 8.4.

https://doi.org/10.1017/9781009099974.010 Published online by Cambridge University Press

8.2 Quantum Programming Model

As our standard model of computation, we assume the quantum random access model (QRAM) as proposed by Knill (1996). The model proposes connecting a general-purpose machine with a quantum computer in order to use it as an accelerator. Registers are explicitly quantum or classical. There are functions to initialize quantum registers, to program gate sequences into the quantum device, and to get results back via measurements.



On the surface, this model is not much different from today's programming models for PCIe connected accelerators,[1] such as GPUs or storage devices, which are ubiquitous today. The elegant CUDA programming model for GPUs provides clear abstractions for code that is supposed to run on either device or server (Buck et al., 2004; Nickolls et al., 2008). Source code for the accelerator and host can be mixed in the same source file to enhance programmer productivity.

QRAM is an idealization. Communication between the classical and quantum parts of a program may be severely limited. There may either be a significant lack of compute power close to the quantum circuit, which operates at micro-Kelvin temperature, or bandwidth-limited communication to a CPU further away.

It is important to keep the separation between classical and quantum in mind. In QRAM, as in our simulation infrastructure, the separation of classical and quantum is muddled, running classical loops over applications of quantum gates interspersed with print statements. This might be a good approach for learning, but it is not realistic for a real machine. To a degree, the approach is more akin to an infrastructure such as the machine learning platform Tensorflow. It first builds up computation in the form of a graph before executing the graph in a distributed fashion on CPU, GPU, or TPU.

Another aspect of the QRAM model is the expectation of available *universal* gates on the target quantum machine. Several universal sets of gates have been described in the literature (see Nielsen and Chuang, 2011, section 4.5.3). We showed how any unitary gate can be approximated by universal gates in Section 6.15. With this, we assume that any gate may be used freely in our idealized infrastructure. On real machines, however, the number of gates is limited, and there are accuracy and noise concerns.

8.3 Quantum Programming Languages

In this section, we discuss a representative cross-section of quantum programming languages with corresponding compilers and tooling. The descriptions are brief and

[1] https://en.wikipedia.org/wiki/PCI_Express

https://doi.org/10.1017/9781009099974.010 Published online by Cambridge University Press

necessarily incomplete. Most importantly, the selection does not judge the quality of the nonselected languages. Each attempt makes novel contributions over prior art, variations of which can be found in other related works.

In a hierarchy of abstractions, this is how to place quantum programming languages:

- The high abstraction level of programming languages. This level may provide automatic ancilla management, support correct program construction with advanced typing rules, offer libraries for standard operations (such as QFT), and perhaps offer meta-programming techniques.
- Programming at the level of gates. This is the level of this text. It is the construction and manipulation of individual qubits and gates.
- Direct machine control with pulses and wave forms to operate a physical device. We will not discuss related infrastructure, such as OpenPulse (Gokhale et al., 2020).

For each of the platforms, there is lots of material available online to experiment with. This section is meant to be educational – it should also inspire. For example, we could easily add several of the proposed features to our infrastructure. Also, despite all progress, the development of quantum programming languages and their compilers appears to still be in its infancy.

8.3.1 QASM

The quantum assembly language (QASM) was an early attempt to textually specify quantum circuits (Svore et al., 2006). We've already seen QASM code in Section 6.3 on quantum arithmetic, and we will see more of it in Section 8.5 on transpilation.

The structure of a QASM program is very simple. Qubits and registers are declared, and gate applications follow one by one. There are no looping constructs, function calls, or other constructs that would help to structure and densify the code. For example, a simple entangler circuit would read like this:

```
qubit x,y;
gate h;
gate cx;
h x;
cx x,y;
```

More capable variants emerged to augment QASM in a variety of ways. OpenQASM adds the ability to define new gates, control-flow constructs, and barriers (Cross et al., 2017). It also offers looping constructs. cQASM is one attempt to unify QASM dialects into a single form[2] (Khammassi et al., 2018). It offers additional language features, such as explicit parallelization, register mapping/renaming, and a

[2] See also `xkcd` cartoon #927.

https://doi.org/10.1017/9781009099974.010 Published online by Cambridge University Press

variety of measurement types. An example implementation of a three-qubit Grover algorithm takes about 50 lines of code.

8.3.2 QCL

The *quantum computing language* (QCL) was an early attempt to use classical programming constructs to express quantum computation (Ömer, 2000, 2005; QCL Online). Algorithms are run on a classical machine controlling a quantum computer and might have to run multiple times until a solution is found. Quantum and classical code are intermixed. Qubits are defined as registers of a given length, and gates are applied directly to registers:

```
qureg q[1];
qureg f[1];
H(q);
Not(f);
const n=#q;   // length of q register
for i =1 to n {  // classical loop
   Phase(pi/2^(i));
}
```

QCL defines several quantum register types. There is the unrestricted `qureg`, `quconst` defines an invariant qubit, and `quvoid` specifies a register to be *empty*. It is guaranteed to be in state $|0\rangle$. The register type `quscratch` denotes ancillae.

Code is organized into *quantum functions*. Functions and operators are reversible. Prefixing a function with an exclamation point produces the inverse, as in this example from the Grover algorithm:[3]

```
operator diffuse(qureg q) {
  H(q);                  // Hadamard transform
  Not(q);                // Invert q
  CPhase(pi,q);          // Rotate if q=1111...
  !Not(q);               // Undo inversion
  !H(q);                 // Undo Hadamard transform
}
```

QCL defines several types of functions, such as the nonreversible `procedure`, which may contain classical code and allows side effects. Functions marked as `operator` and `qufunct` are side-effect free and reversible.

[3] Note that both the Hadamard and the NOT gates are their own inverses. This might not be the most convincing example.

https://doi.org/10.1017/9781009099974.010 Published online by Cambridge University Press

To facilitate uncomputation, QCL supports a `fanout` operation. It restores scratch registers and auxiliary registers, while preserving the results, as outlined in Section 2.13 on uncomputation:

$$\begin{aligned} |x,0,0,0\rangle &\to |x,0,f(x),g(x)\rangle \\ &\to |x,g(x),f(x),g(x)\rangle \\ &\to |x,g(x),0,0\rangle \end{aligned}$$

The implementation of `fanout` is quite elegant: Assume a function $F(x,y,s)$ with x being the input, y being the output, and s being junk qubits. Allocate the ancilla t and transform F into the following, adding t to its signature:

$$F(x,y,s,t) = F^{\dagger}(x,t,s)\ \texttt{fanout}(t,y)\ F(x,t,s).$$

What makes this elegant is the fact that `fanout` is written in QCL itself:

```
cond qufunct Fanout(quconst ancilla, quvoid b) {
    int i;
    for i=0 to #ancilla-1 {
        CNot(b[i], ancilla[i]);
    }
}
```

QCL supports conditionals in interesting ways. Standard controlled gates are supported as described in Section 2.7. If a function signature is marked with the keyword `cond` and has as parameter a `quconst` condition qubit, QCL automatically transforms the operators in the functions to controlled operators:

```
  cond qufunct cinc(qureg x, quconst e) { . . . }
```

Additionally, QCL supports an `if` statement, where `if e { inc(x); }` is equivalent to a new function `cinc(x, e)` as shown above, with the if-then-else statement translating into:

```
  if e {
      inc(x);
  } else {
      !inc(x);
  }
=>
  cinc(x, e);
  Not(e);
  !cinc(x, e);
  Not(e)
```

https://doi.org/10.1017/9781009099974.010 Published online by Cambridge University Press

As an example, here is the implementation of QFT in QCL, as found in the thesis (Ömer, 2000):

```
cond qufunct flip(qureg q) {
  int i;                // declare loop counter
  for i=0 to #q/2-1 {   // swap 2 symmetric bits
    Swap(q[i],q[#q-i-1]);
  }
}

operator qft(qureg q) { // main operator
  const n=#q;           // set n to length of input
  int i; int j;         // declare loop counters
  for i=1 to n {
    for j=1 to i-1 {    // apply conditional phase gates
      V(pi/2^(i-j),q[n-i] & q[n-j]);
    }
    H(q[n-i]);          // qubit rotation
  }
  flip(q);              // swap bit order of the output
}
```

8.3.3 Scaffold

Scaffold takes a different approach (Javadi-Abhari et al., 2014). It *extends* the open-source LLVM compiler and its Clang-based front end for C/C++. Scaffold introduces data types *qbit* and *cbit* to distinguish quantum from classical data. Quantum gates, such as the X-gate or the Hadamard gate, are implemented as *built-ins*; the compiler recognizes them as such and can reason about them in transformation passes.

Scaffold supports hierarchical code structure via *modules*, which are specially marked functions. Quantum circuits do not support calls and returns, so modules representing subcircuits need to be *instantiated*, similar to, say, how Verilog modules are instantiated in a hardware design. Modules must be reversible, either by design or via automatic compiler transformations, such as full unrolling of classical loops.

Scaffold offers convenient functionality to convert classical circuits to quantum gates, via the Classical-To-Quantum-Circuit (CTQC) tool. This tool is of great utility for quantum algorithms that perform classical computation in the quantum domain. CQTC emits QASM assembly. To enable whole program optimization, Scaffold has a QASM to LLVM IR transpiler, which can be used to import QASM modules, enabling further cross-module optimization.

Modules are parameterized. This means the compiler has to manage module instantiation, for example, with IR duplication. This can lead to sizeable code bloat and correspondingly long compile times. The example given is the following code snippet, where the module `Oracle` would have to be instantiated $N = 3000$ times. Clearly, a parameterized IR would alleviate this problem considerably.

https://doi.org/10.1017/9781009099974.010 Published online by Cambridge University Press

```
#define N 3000 // iteration count
module Oracle (qbit a[1], qbit b[1], int j) {
  double theta = (-1)*pow(2.0, j)/100;
  X(a[0]);
  Rz(b[0], theta);
}

module main () {
  qbit a[1], b[1];
  int i, j;
  for (i=1; i<=N; i++) {
    for (j=0; j<=3; j++) {
       Oracle(a, b, j);
    }
  }
}
```

As a result, Javadi-Abhari et al. (2014) reports compile times ranging from 24 hours to several days for a larger triangle finding problem with size $n = 15$ (see also Magniez et al., 2005).

Hierarchical QASM

Scaffold intends to scale to very large circuits. The existing QASM model, as shown above, is *flat*, which is not suitable for large circuits. One of the main contributions of Scaffold is the introduction of *hierarchical* QASM. Additionally, the compiler employs heuristics for what code sequences to flatten or keep in a hierarchical structure. For example, the compiler distinguishes between *forall* loops to apply a gate to all qubits in a register and *repeat* loops, such as those required for the Grover iterations.

Entanglement Analysis

Scaffold includes tooling for entanglement analysis. In the development of Shor's algorithm, we observed a certain ancilla qubit that was still entangled after modular addition. How does one reason about this?

Scaffold tracks entanglement-generating gates, such as Controlled-Not gates, on a stack. As inverse gates are executed in reverse order, items are popped off the stack. If, for a given qubit, no more entangling gates are found on the stack, the qubit is marked as *unentangled*. As a result of the analysis, the generated output can be decorated to show the estimated entanglement:

```
module EQxMark_1_1 ( qbit* b , qbit* t ) {
...
Toffoli ( x[0] , b[1] , b[0] );
// x0, b1, b0
Toffoli ( x[1] , x[0] , b[2] );
```

https://doi.org/10.1017/9781009099974.010 Published online by Cambridge University Press

```
// x1, x0, b2, b1, b0
...
}
// Final entanglements:
// (t0, b4, b3, b2, b1, b0);
```

8.3.4 Q language

We can contrast this work with a pure C++-embedded approach, as presented in Bettelli et al. (2003). This approach consists of a library of C++ classes modeling quantum registers, operators, operator application, and other functions, such as reordering of quantum registers. The class library builds up an internal data structure to represent the computation, similar in nature to the infrastructure we developed here. It is interesting to ponder the question of which approach makes more sense:

- Extension of the C/C++ compiler with specific quantum types and operators, as in Scaffold.
- A C++ class library as in the *Q language*.

Both approaches appear equally powerful in principle. The compiler-based approach has the advantage of benefitting from a large set of established compiler passes, such as inlining, loop transformations, redundancy elimination, and many other scalar, loop, and inter-procedural optimizations. The C++ class library has the advantage that the management of the IR, all optimizations, and final code generation schemes are being maintained *outside* of the compiler. Since compilers can prove impenetrable for noncompiler experts, this approach might have a maintenance advantage, but at the cost of potentially having to re-implement many optimization passes.

8.3.5 Quipper

Haskell is a popular choice for programming language theorists and enthusiasts because of its powerful type system. An example of a Haskell-embedded implementation of a quantum programming system can be found with the Quantum IO Monad (Altenkirch and Green, 2013). Another even more rigid example is van Tonder's proposal for a λ-calculus to express quantum computation (van Tonder, 2004).

What these approaches have in common is the attempt to guarantee correctness by construction with support of the type system. This is also one of Quipper's core design ideas (Green et al., 2013; Quipper Online, 2021). Quipper is an embedded DSL in Haskell. At the time of Quipper's publication, Haskell lacked linear types, which could have guaranteed that objects were only referenced once, as well as dependent types, which are types combined with a value. Dependent types, for example, allow you to distinguish a QFT operator over n qubits from one over m qubits.

https://doi.org/10.1017/9781009099974.010 Published online by Cambridge University Press

Quipper is designed to scale and handle large programs with up to 10^{12} operators. Quipper has a notion of ancilla scope, with an ability to reason about ancilla live ranges. Allocating ancilla qubits turns into a register allocation problem. Ancilla live ranges have to be marked explicitly by the programmer.

At the language level, qubits are held in variables and gates are applied to these variables. For example, to generate a Bell state:

```
bell :: Qubit -> Qubit -> Circ (Qubit, Qubit)
bell a b = do
   a <- hadamard a
   (a, b) <- controlled_not a b
   return (a, b)
```

To control an entire block of gates, Quipper offers a `with_controls` construct, similar in nature to QCL's `if` blocks. Another block-level construct allows managing ancillae explicitly via `with_ancilla`. Circuits defined this way are reversed with a `reverse_simple` construct. Quipper's type system distinguishes different types of quantum data, such as simple qubits, or fixed point interpretations of multiple qubits.

Automatic Oracles

Quipper offers tooling for the automatic construction of oracles. Typically, oracles are constructed with the following four manual steps. There are open-source implementations available for these techniques (Soeken et al., 2019).

1. Build a classical oracle, for example, a permutation matrix.
2. Translate the classic oracle into classical circuits.
3. Compile classical circuits to quantum circuits, potentially using additional ancillae. We saw examples of this in Section 3.3.1.
4. Finally, make the oracle reversible, typically with an XOR construction to another ancilla.

Quipper utilizes Template Haskell to automate steps two and three. The approach has high utility and has been used to synthesize millions of gates in a set of benchmarks. In direct comparison to QCL on the Binary Welded Tree algorithm, it appears that QCL generates significantly more gates and qubits than Quipper. On the other hand, Quipper appears to generate more ancillae.

Despite the tooling, type checks, automation of oracles, and utilization of the Haskell environment, it still took 55 man months to implement the 11 algorithms in a given benchmark set (IARPA, 2010). This is certainly a productivity improvement over manually constructing all the benchmarks at the gate level, but it still compares unfavorably against programmer productivity on classical infrastructure.

https://doi.org/10.1017/9781009099974.010 Published online by Cambridge University Press

Quipper led to interesting follow-up work, such a Proto-Quipper-M (Rios and Selinger, 2018), Proto-Quipper-S (Ross, 2017), up to Proto-Quipper-D (Fu et al., 2020). These attempts are steeped in type theory and improve on program correctness by a variety of techniques, for example, using linear types to enforce the no-cloning theorem and linear *dependent* types to support the construction of type-safe families of circuits.

8.3.6 Silq

Based on a fork from the PSI probabilistic programming language (PSI Online, 2021), *Silq* is another step in the evolution of quantum programming languages, supporting safe and *automatic* uncomputation (Bichsel et al., 2020).

It explicitly distinguishes between the classical and quantum domains with syntactical constructs. Giving the responsibility for safe uncomputation to the compiler leads to two major benefits. First, the code becomes more compact. Direct comparisons against Quipper and Q# appear to show significant code size savings for Silq in the range of 30% to over 40%. Second, the compiler may choose an optimal strategy for uncomputation, minimizing the required number of ancillae. As an added benefit, the compiler may choose to skip uncomputation for simulation altogether and just renormalize states and unentangle ancillae.

Many of the Haskell embedded DSLs bemoan either the absence of linear types or difficulties handling constants. Silq resolves this by using linear types for nonconstant values and a standard type system for constants. This leads to safe semantics, even across function calls, and the no-cloning theorem falls out naturally. Function type annotations are used to aid the type checker:

- The annotation `qfree` indicates that a function can be classically computed. For example, the quantum X-gate is considered `qfree`, while the superposition-inducing Hadamard gate is not.
- Function parameters marked as `const` are preserved and not *consumed* by a function. They continue to be accessible after a function call. Parameters not marked as `const` are no longer available after the function call. Functions with only `const` parameters are called `lifted`.
- Functions marked as `mfree` promise not to perform measurements and are reversible.

Silq supports other quantum language features, such as function calls, measurement, explicit reversing of an operator via `reverse`, and an `if-then-else` construct that can be classical or quantum, similar to other quantum languages. Looping constructs must be classical. As an improvement over prior approaches, Silq supports Oracle construction with quantum gates.

With the annotations and the corresponding operational semantics, Silq can safely deduce which operations are safe to reverse and uncompute, even across function calls. The paper provides many examples of potentially hazardous corner cases that are being handled correctly (Bichsel et al., 2020).

https://doi.org/10.1017/9781009099974.010 Published online by Cambridge University Press

As a program example, the code snippet below solves one of the challenges in Microsoft's Q# Summer 2018 coding contest:[4] *Given classical binary string* $b \in \{0,1\}^n$ *with* $b[0] = 1$*, return state* $1/\sqrt{2}(|b\rangle + |0\rangle)$*, where* $|0\rangle$ *is represented using n qubits*.

The code iself demonstrates several of Silq's features, for example, the use of `!` to denote classical values and types.

```
def solve[n:|N|](bits:|!B|^n){
  // prepare superposition between 0 and 1
  x:=H(0:|!B|);
  // prepare superposition between bits and 0
  qs := if x then bits else (0:int[n]) as |!B|^n;
  // uncompute x
  forget(x=qs[0]); // valid because bits[0]==1
  return qs;
}

def main(){
  // example usage for bits=1, n=2
  x := 1:|!|int[2];
  y := x as |!B|^2;
  return solve(y);
}
```

8.3.7 Commercial Systems

Commercial systems are open-source infrastructures that are maintained by commercial entities. The most important systems appear to be Microsoft's Q# (Microsoft Q#, 2021), IBM's Qiskit (Gambetta et al., 2019), Google's Cirq (Google, 2021c), and ProjectQ (Steiger et al., 2018). Microsoft's Q# is a functional stand-alone language and a part of the Quantum Developer Kit (QDK). Qiskit, Cirq, and ProjectQ all provide Python embeddings. By the time you read this, others might have become more popular.

These ecosystems are vast, fast evolving, and provide excellent learning materials that we do not have to cover here. For further reading, we recommend (Garhwal et al., 2021), which details Q#, Cirq, ProjectQ, and Qiskit, or Chong et al. (2017), which describes some of the major challenges for quantum tool flows in general.

8.4 Compiler Optimization

Compiler optimization is a fascinating topic in classical compiler construction. For quantum compilers, it gets even more exciting, given the exponential complexity

[4] http://codeforces.com/blog/entry/60209.

https://doi.org/10.1017/9781009099974.010 Published online by Cambridge University Press

and novelty of transformations. Compiler optimizations play an important role in several areas:

- **Ancilla management**. As we use higher-level abstractions and programming languages, ancilla qubits should be managed automatically, in a manner similar to register allocation for classical compilers. The compiler can trade off circuit depth against the number of ancilla bits, supporting the goal of squeezing a circuit into limited resources. Minimizing ancillae in the general case appears to be an unsolved problem.
- **Noise reduction**. The application of quantum gates is subject to noise. Some gates introduce more noise than others. Hence, the role of the optimizer is to minimize gates as a whole and emit gate sequences to actively contain noise.
- **Gate mapping to physical machines**. Real quantum computers only support a small number of gate types. The compiler must decompose logical gates and map them to available physical gates. Furthermore, at least in the short term, the number of available qubits is extremely limited. It is one of the compiler's main roles to map circuits onto those limited resources.
- **Logical to physical register mapping**. Quantum computers have topological constraints on how qubits can interact with each other. For example, only next neighbor interactions may be possible in some cases. Multi-qubit gates spanning nonneighboring qubits thus must be decomposed into two-qubit gates between neighboring qubits.
- **Accuracy tuning**. Individual gates may not be accurate enough for a given algorithm, and multiple gates may be necessary to achieve the desired result. The compiler plays a central role in determining the required accuracy and the corresponding generation of approximating circuits.
- **Error correction**. Automatic insertion of minimal error-correcting circuitry is an important task for the compiler.
- **Tooling**. The compiler sees the whole circuit and can apply whole-program analyses, such as the entanglement analysis that we saw in Section 8.3.3.
- **Performance**. Optimization should also target circuit depth and complexity. Given the short coherence times of real machines, the shorter a circuit has to run, the fewer gates it needs to execute, the higher the chances of reliable outcomes.

The space is large and complex, and we won't be able to cover it exhaustively. Instead, we again provide representative examples of key principles and techniques, hoping to give a taste of the challenges.

8.4.1 Classic(al) Compiler Optimizations

In our infrastructure and many other platforms we described in Section 8.3, classical code is freely intermixed with quantum code. This means that classical optimizations,

https://doi.org/10.1017/9781009099974.010 Published online by Cambridge University Press

such as loop unrolling, function inlining, redundancy elimination, constant propagation, and many other scalar, loop- and inter-procedural optimizations still apply. This is necessary because all classical constructs must be eliminated before sending a circuit to the quantum accelerator. Additionally, classical techniques like the elimination of dead code and constant folding equally apply to quantum circuitry.

Scaffold is a great example of the mix of the classical and quantum worlds and the impact of classical optimizations on the performance of a quantum circuit (Javadi-Abhari et al., 2014). Scaffold represents quantum operations in a classical compiler's intermediate representation (IR) and directly benefit from the rich library of available optimization passes in LLVM (Lattner and Adve, 2004).

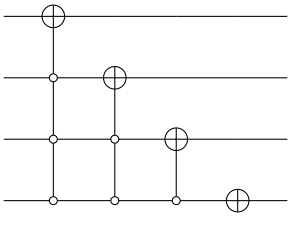
Other known classical techniques also apply. Analysis of communication overhead and routing strategies developed for distributed systems work with modifications for quantum computing (Ding et al., 2018). Register allocation can lead to optimal allocation and reuse of quantum registers (Ding et al., 2020).

8.4.2 Simple Gate Transformations

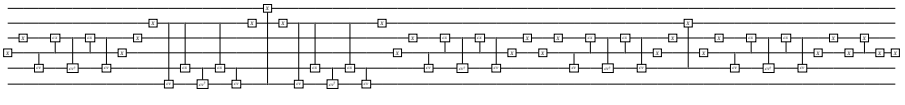
The most basic optimization is to eliminate gates that will have no effect. For example, two X-gates in a row, or two other involutory matrices in sequence acting on the same qubit, or two rotations adding up to 0; all of these can be eliminated:

$$Z_i \underbrace{X_i X_i}_{redundant} Y_i = Z_i Y_i.$$

Sequences like this can be found as a result of higher-level transformations which chain together independent circuit fragments. For example, take the four-qubit decrement circuit which we detailed in Section 6.10 on quantum random walks:

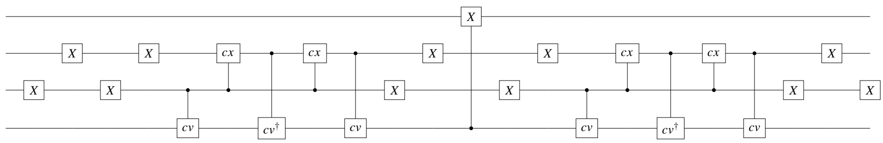


The circuit expands the multi-control gates into this much longer sequence of gates (don't worry, you are not expected to be able to decipher this):



https://doi.org/10.1017/9781009099974.010 Published online by Cambridge University Press

Zooming in at the right, we can see the opportunity to eliminate redundant X-gates:

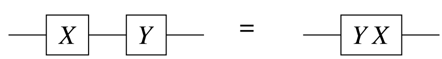


In general, for a single-qubit operator U, if the compiler can prove that the input state is an eigenstate of U with an eigenvalue of 1 (which means $U|\psi\rangle = |\psi\rangle$), it can simply remove the gate. For example, if the qubit is in the $|+\rangle$ state, the X-gate has no effect, as $X|+\rangle = |+\rangle$.

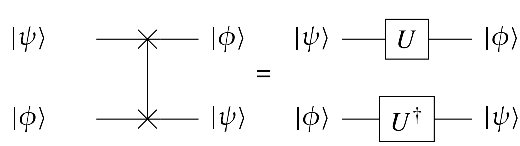
Depending on the numerical conditioning of an algorithm, the compiler may also decide to remove gates that have only small effects. As an example, we have seen the effectiveness of this technique in the approximate QFT (Coppersmith, 2002).

8.4.3 Gate Fusion

For simulation, and perhaps for physical machines with a suitable gate set, we can *fuse* consecutive gates via simple matrix multiplies. Some of the high-performance simulators apply this technique. Fusion can happen at several levels and across a varying number of qubits. The resulting gates may not be available on a physical machine, in which case the compiler will have to approximate the fused gates. This can nullify the benefits of the fusion, but in cases where two gates X and Y both have to be approximated, it may be beneficial to approximate the combined gate YX:



The compiler can also exploit the fact that qubits may be unentangled. For example, assume qubits $|\psi\rangle$ and $|\phi\rangle$ are known to be unentangled and must be swapped, potentially by a Swap gate spanning multiple qubits. Since the gates are unentangled and in a pure state, we may be able to classically find a unitary operator U such that $U|\psi\rangle = |\phi\rangle$ and $U^\dagger|\phi\rangle = |\psi\rangle$. In circuit notation:



8.4.4 Gate Scheduling

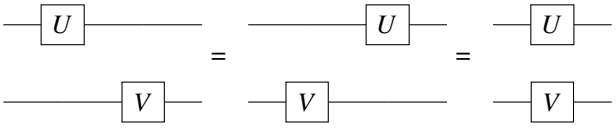
We have described many gate equivalences, and there are even more available in the literature. Which specific gate sequence to use will depend on what a specific quantum

https://doi.org/10.1017/9781009099974.010 Published online by Cambridge University Press

computer can support, topological constraints, and also on the relative cost of specific gates. For example, T-gates might be an order of magnitude slower than other gates and may have to be avoided.

In order to find the best equivalences, pattern matching can be used. To maximize the number of possible matches, you may have to reorder and reschedule gates. Valid and efficient recipes for reordering are hence a rich area of research. As a simple example, single-qubit gates applied to different qubits can be reordered and parallelized, as:

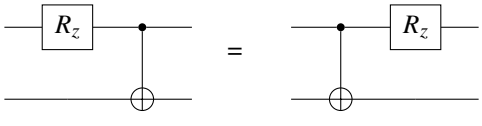
$$(U \times I)(I \times V) = (I \times V)(U \times I) = (U \times V)$$



There are many other opportunities to reorder. For example, if a gate is followed by a controlled gate of the same type, the two gates can be re-ordered:

$$Y_i CY_{ji} = CY_{ji} Y_i.$$

Rotations are a popular target for reordering. For example, the S-gate, T-gate, and Phase-gate all represent rotations, which can be applied in any order. Nam et al. (2018) provide many recipes, rewrite rules, and examples, such as the following:



In simulation, it may not help to parallelize gates, at least in our implementation. On a physical quantum computer, however, it is safe to assume that multiple gates will be able to operate in parallel. Mapping gates to parallel running qubits will improve device utilization and has the potential to reduce circuit depth. Shorter depth means shorter runtime and a higher probability to conclude an execution before decoherence.

Measurements typically happen at the end of a circuit execution. Qubits have a limited lifetime, so it is a good strategy to initialize qubits as late as possible. This is achieved with a policy to schedule gates *as late as possible* (ALAP), working backward from the measurement. This is also the default policy in IBM's Qiskit compiler. Ding and Chong (2020) detail other scheduling policies and additional techniques to minimize communication costs.

8.4.5 Peephole Optimization

Peephole optimization gets its name from the fact that it looks at only a small sliding window over code or circuitry, hoping to find exploitable patterns in this small

https://doi.org/10.1017/9781009099974.010 Published online by Cambridge University Press

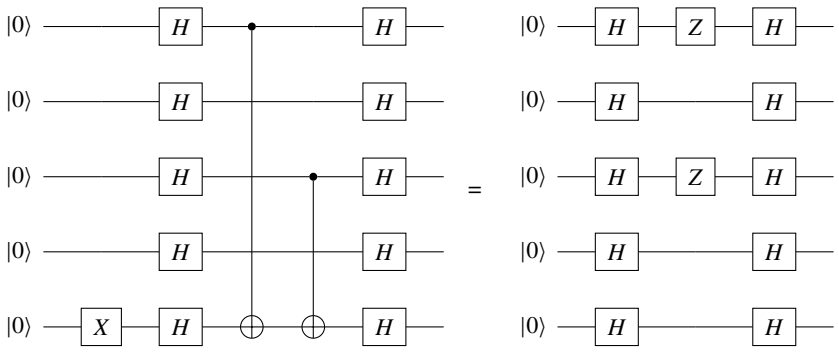


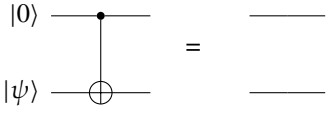
Figure 8.1 Optimized Bernstein-Vazirani circuit.

window. This is a standard technique in classical computing, but it applies to quantum computing as well (McKeeman, 1965).

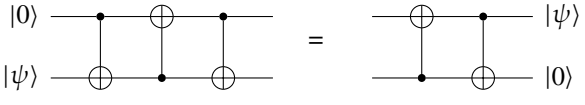
Limited window pattern matching approaches have in common that, for a given gate replacement, the underlying unitary operator must not change. This guarantees correctness of a transformation.

With *relaxed peephole optimization*, this constraint can be, well, *relaxed* (Liu et al., 2021). For example, if a controlling qubit is known to be in state $|0\rangle$ as shown above, we can eliminate the controlled gate. The circuit is still logically equivalent, but the underlying operator has changed.

We can exploit this insight. A Controlled-U operation with a controlling $|0\rangle$ qubit has no effect and can be eliminated (the compiler has to *ascertain* that the controller will be $|0\rangle$):



We can also squeeze the Swap gate if one of the inputs is known to be $|0\rangle$:



The Controlled-Not gates in the Bernstein–Vazirani oracle circuit can be replaced by simple Z-gates because the leading Hadamard gates put the qubits into the $|+\rangle$ basis. This is shown in Figure 8.1. The techniques can be generalized to multi-controlled gates as well. More examples of this technique, along with a full evaluation, can be found in (Liu et al., 2021).

https://doi.org/10.1017/9781009099974.010 Published online by Cambridge University Press

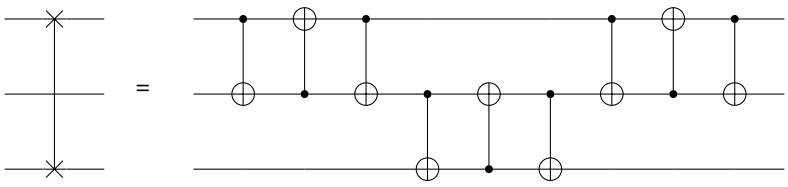


Figure 8.2 Decomposition of a Swap gate spanning three qubits into next-neighbor controlled gates.

8.4.6 High-Performance Pattern Libraries

The efficient matching of patterns to gate sequences is a challenge. A possible approach is to precompute a library of high-performance subcircuits and then transpile nonoptimal and permuted subcircuits into known high-performance circuits. This approach is similar to the end-game library in a chess-playing computer. McKeeman (1965) gives an example of building up a library of thousands of highly optimized four-qubit subcircuits, which were found with an elaborate automated search.

8.4.7 Logical to Physical Mapping

We have already seen many gate equivalences in Section 3.2. Choosing which ones to apply will depend on the physical constraints of an underlying architecture. In this context, logical to physical qubit mapping presents an optimization challenge.

For example, Swap gates may only be applied to neighboring physical qubits. If there is a swap between qubits 0 and (very large) n, it might be better to place physical qubit n right next to qubit 0. Otherwise, the *communication overhead* will be very high. For example, a construction like the following is needed to swap qubits 0 and 2 in a three-qubit circuit. The circuit as presented is not very efficient; it simply stitches together a series of two-qubit Swap gates. To bridge longer distance swaps, this ladder must be extended to more qubits, all the way down and back up as shown in the example in Figure 8.2 for a swap gate spanning over 3 qubits.

If the physical qubit assignment has been decided, gates may have to be further deconstructed to fit the topological constraints. In the example shown in Figure 8.3, a Controlled-Not from a qubit 0 to qubit 2 is being decomposed into next-neighbor controlled gates. Several other types of Controlled-Not deconstructions deconstructions are presented in Garcia-Escartin and Chamorro-Posada (2011).

A related proposed technique is *wire optimization* (Paler et al., 2016). It uses qubit lifetime analysis to *recycle* wires and qubits, with the insight being that not all qubits are needed during the execution of a full circuit. Under the assumption that we can measure and reuse qubits, this work shows drastic reductions in the number of required qubits for an algorithm, up to 90%. This mirrors the results we find with our sparse implementation. However, at the time of this writing, it does not appear that intermittent measurement and reinitialization of qubits can be performed efficiently.

https://doi.org/10.1017/9781009099974.010 Published online by Cambridge University Press

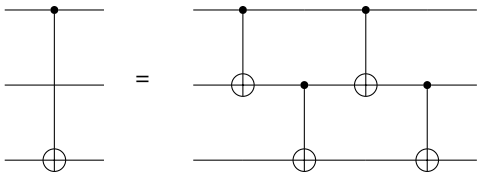


Figure 8.3 A Controlled-Not from qubit 0 to qubit 2 is decomposed into next-neighbor Controlled-Not gates.

8.4.8 Physical Gate Decomposition

Finally, an important step for compiler and optimizer is to decompose higher level gates to actually available, physical gates, respecting connectivity constraints. For example, the IBMQX5 has five qubits and the gates u_1, u_2, u_3, as well as a CNOT gate (IBM, 2021a), which can only be applied to neighboring gates.

$$
\begin{aligned}
u_1(\lambda) &= \begin{bmatrix} 1 & 0 \\ 0 & e^{i\lambda} \end{bmatrix}, \\
u_2(\phi, \lambda) &= \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & -e^{i\lambda} \\ e^{i\phi} & e^{i(\phi+\lambda)} \end{bmatrix} \\
&= R_z(\phi + \pi/2) R_x(\pi/2) R_z(\lambda - \pi/2), \\
u_3(\theta, \phi, \lambda) &= \begin{bmatrix} \cos(\theta/2) & -i \sin(\theta/2) \\ e^{i\phi} \sin(\theta/2) & e^{i(\lambda+\phi)} \cos(\theta/2) \end{bmatrix}.
\end{aligned}
$$

Other architectures offer different available gates on differently-shaped topologies. Mapping idealized gates to physical gates is challenging, especially if the physical gates have unusual structure. A broader analysis and taxonomy can be found in Murali et al. (2019).

We also discussed earlier, in Section 8.2, that in an idealized programming model we may use any gate, knowing that gates can be approximated. The key questions are:

1. What is the *best* set of gates to realize in hardware?
2. What is the impact of this choice on gate approximation or other design parameters and circuit depth?
3. What is the accuracy impact of approximations on an algorithm?
4. If approximation would require an exponentially growing set of gates, would this not nullify the complexity advantage of quantum algorithms?

Some abstract gates will be easier to approximate than others on a given physical instruction set, such as the IBM machine above. Each target and algorithm will hence require targeted methodology and compilation techniques.

https://doi.org/10.1017/9781009099974.010 Published online by Cambridge University Press

8.5 Transpilation

Transpilation is an interesting modulation of the term *compilation*. Typically, transpilation describes the process of translating a program written in one programming language to another, for example, translating Java to Javascript, where still another compilation step is needed to run the code on an actual machine. In quantum computing, the term is typically used to describe the process of mapping a circuit onto a physical quantum computer, in which case it should be called compilation. Naming is difficult.

In our context, we use the term transpilation as the compiler gods intended: to describe the process of taking a circuit written in our infrastructure and transpiling it to another platform, such as IBM's Qiskit, Google's Cirq, or our sparse representation `libq`. It is not compilation because the translation from the input onto the target machine still has to be done by the respective platforms. Transpilation appears to be the accurate term in our context, but we may also refer to the various levels of compilation as *staged compilation*. Did we mention yet that naming is hard?

In this section, we introduce a simple data structure that allows transpilation (!) of our algorithms to other formats, including QASM (Cross et al., 2017), which is supported by many platforms, including IBM's Qiskit. Other transpilers are in development; you can find them in the open-source repository. The code generators are quite simple and mostly of prototype quality. We show them in great detail to encourage experimentation and the development of additional transpilers.

8.5.1 Intermediate Representation (IR)

To compile a program into another form, we need a data structure to represent the input. In compilers, this data structure is typically called an *intermediate representation* (IR). Our IR will be very simple – just a list of nodes corresponding to the gates as they were added to the circuit. As described earlier, quantum computing has no classical control flow, which enables the use of this simple data structure for a number of purposes.

There are only three meaningful operation types in our infrastructure: the undefined operation `UNK`, a single-qubit gate `SINGLE`, and a controlled gate `CTL`. We define them all with the enumeration type `class Op(enum.Enum)`. For debugging and better formatting of generated outputs, we also introduce the notion of a *section*, but we ignore this for now.

8.5.2 IR Nodes

A node will hold all information available for the operations, such as target qubit or intended rotation angle. The nodes themselves are represented by a simple Python class with all the relevant parameters passed to its constructor. One class is enough to represent all possible node types. Again, we keep it very simple in this prototype implementation.

https://doi.org/10.1017/9781009099974.010 Published online by Cambridge University Press

```
class Node:
 """Single node in the IR."""

 def __init__(self, opcode, name, idx0=0, idx1=None, val=None):
   self._opcode = opcode
   self._name = name
   self._idx0 = idx0
   self._idx1 = idx1
   self._val = val
```

We add functions to check for node properties:

```
def is_single(self):
   return self._opcode == Op.SINGLE

 def is_ctl(self):
   return self._opcode == Op.CTL

 def is_gate(self):
   return self.is_single() or self.is_ctl()
```

Then, based on the specific node type, the transpilers will query the properties to get the node attributes:

```
@property
 def opcode(self):
   return self._opcode

 @property
 def name(self):
   if not self._name:
     return '*unk*'
   return self._name

 @property
 def desc(self):
   return self._name
[...]
```

8.5.3 IR Base Class

The IR itself is now fairly straightforward. It is just a list of nodes, functions to add nodes for single gates and controlled gates, and a function to manage quantum registers.

https://doi.org/10.1017/9781009099974.010 Published online by Cambridge University Press

```
class Ir:
 """Compiler IR."""

  def __init__(self):
    self._ngates = 0  # gates in this IR
    self.gates = []   # [] of gates
    self.regs = []    # [] of tuples (global reg index, name, reg index)
    self.nregs = 0    # number of registers
    self.regset = []  # [] of tuples (name, size, reg) for registers


  def reg(self, size, name, register):
    self.regset.append((name, size, register))
    for i in range(size):
      self.regs.append((self.nregs + i, name, i))
    self.nregs += size

  def single(self, name, idx0, val=None):
    self.gates.append(Node(Op.SINGLE, name, idx0, None, val))
    self._ngates += 1

  def controlled(self, name, idx0, idx1, val=None):
    self.gates.append(Node(Op.CTL, name, idx0, idx1, val))
    self._ngates += 1

  @property
  def ngates(self):
    return self._ngates
```

8.5.4 Quantum Circuit Extensions

To construct the IR, we moderately extend the quantum circuit class. An *eager* mode executes the circuit as it is being constructed. This is the default behavior. Setting eager to `False` will only construct the IR and not execute the circuit. We also add the IR to the quantum circuit by extending the constructor:

```
def __init__(self, name=None, eager=True):
   self.name = name
   self.psi = 1.0
   self.ir = ir.Ir()
   self.eager = eager
   state.reset()
```

https://doi.org/10.1017/9781009099974.010 Published online by Cambridge University Press

There are only two functions in the `qc` class that *apply* gates. We have to add IR-construction calls to those two functions. This is one of the benefits of having this abstraction, as alluded to in Section 4.3:

```
def apply1(self, gate, idx, name=None, *, val=None):
   if isinstance(idx, state.Reg):
     for reg in range(idx.nbits):
       self.ir.single(name, idx[reg], val)
       if self.eager:
         xgates.apply1(self.psi, gate.reshape(4), self.psi.nbits,
                       idx[reg], tensor.tensor_width)
     return
   self.ir.single(name, idx, val)
   if self.eager:
     xgates.apply1(self.psi, gate.reshape(4), self.psi.nbits, idx,
                   tensor.tensor_width)

 [...] similar for apply_controlled
```

Registers are also supported. As a matter of fact, at the time of writing, *only* quantum registers are supported for code generation. In other words, to generate valid output, the qubits have to be generated and initialized as registers.

8.5.5 Circuits of Circuits

The IR enables a few other powerful capabilities for circuits. It allows the storing away of subcircuits and then combining them later in flexible ways. This makes it easy to invert a circuit, which can be helpful for uncomputation. Furthermore, with just minor changes, we could introduce control for whole subcircuits, similar to the constructs in QCL from Section 8.3.2.

We can create multiple circuits and add them to build even larger circuits. For example, we can create a main circuit, a subcircuit, and replay this subcircuit three times with code like this:

```
main = circuit.qc('main circuit, eager execution')
[... add gates to main]

sub1 = circuit.qc('sub circuit', eager=False)
[... gates to sub1, non-eager]

# Now add three copies of sub1 to main (eager),
# all at a different offset:
main.qc(sub1, 0)
main.qc(sub1, 1)
main.qc(sub1, 2)
```

https://doi.org/10.1017/9781009099974.010 Published online by Cambridge University Press

It is important to note that as the subcircuit is being constructed, it is not yet executed. The gates and their order are recorded for replay later. This is achieved with the `eager` parameter set to `False`.

The second useful capability of this IR is that gate sequences can be inverted. To achieve this, the stored list of gates is reverted. In the process, each gate is replaced by its adjoint, and the gate names get decorated with `"^-1"`. To reverse the application of the three subcircuits in the code example above, the following code could be used. Note how using inverses can be used for uncomputation in an elegant way.

```
# Create an inverse copy of sub1 (which is still non-eager)
sub1_inv = sub1.inverse()

# Now add three copies of sub1 to main (eagar),
# at the reverted list of offsets:
main.qc(sub1_inv, 2)
main.qc(sub1_inv, 1)
main.qc(sub1_inv, 0)
```

8.5.6 Code Generation

We now briefly discuss several transpilers. There is not a lot of magic to them; the compilation from one infrastructure to the next is mostly linear, gate application by gate application, just generating different syntax. To invoke any of the code generators, we define a flag for each of them:

```
flags.DEFINE_string('libq', '', 'Generate libq output file')
flags.DEFINE_string('qasm', '', 'Generate qasm output file')
flags.DEFINE_string('cirq', '', 'Generate cirq output file')
```

For example, to produce a QASM output file, the following flag will generate this format and write it to `/tmp/test.qasm`:

```
> ... --qasm=/tmp/test.qasm
```

To enable this, we add the following functions to the quantum circuit class. The function `dump_to_file` checks for any of the flags. If one is present, it passes the flag and a corresponding code generator function to `dump_with_dumper`, which will call this function on the IR and produce the output:

```
def dump_with_dumper(self, flag: bool,
                     dumper_func: Callable[ir.Ir]) -> None:
  if flag:
    result = dumper_func(self.ir)
```

https://doi.org/10.1017/9781009099974.010 Published online by Cambridge University Press

```
      with open(flag, 'w') as f:
        print(result, file=f)

  def dump_to_file(self):
    self.dump_with_dumper(flags.FLAGS.libq, dumpers.libq)
    self.dump_with_dumper(flags.FLAGS.qasm, dumpers.qasm)
    self.dump_with_dumper(flags.FLAGS.cirq, dumpers.cirq)
```

There are, of course, better ways to structure this, especially when many more code generators and options become available. For this text, the simple implementation will do. The various code generators below also use a small number of helper functions, which can be found in the open-source repository as well.

Fractions of Pi

As we produce textual output, it greatly improves readability to print fractions of π as a fraction such as $3\pi/2$ instead of 4.71238898038. For example, the complex algorithms use the quantum Fourier transform with lots of rotations. Showing them as fractions of π makes debug prints and generated code easier to read.

```
def pi_fractions(val, pi='pi') -> str:
  """Convert a value in fractions of pi."""

  if val is None:
    return ''
  if val == 0:
    return '0'
  for pi_multiplier in range(1, 4):
    for denom in range(-128, 128):
      if denom and math.isclose(val, pi_multiplier * math.pi / denom):
        pi_str = ''
        if pi_multiplier != 1:
          pi_str = f'{abs(pi_multiplier)}*'
        if denom == -1:
          return f'-{pi_str}{pi}'
        if denom < 0:
          return f'-{pi_str}{pi}/{-denom}'
        if denom == 1:
          return f'{pi_str}{pi}'
        return f'{pi_str}{pi}/{denom}'
  # Couldn't find fractional, just return original value.
  return f'{val}'
```

8.5.7 QASM

The first dumper we present is the simplest one: QASM. It just traverses the list of nodes and outputs the nodes with their names as found. Conveniently, the names chosen for the operators already match the QASM specification. Not a coincidence.

https://doi.org/10.1017/9781009099974.010 Published online by Cambridge University Press

```
def qasm(ir) -> str:
 """Dump IR in QASM format."""

 res = 'OPENQASM 2.0;\n'
 for regs in ir.regset:
   res += f'qreg {regs[0]}[{regs[1]}];\n'
 res += '\n'

 for op in ir.gates:
   if op.is_gate():
     res += op.name
     if op.val is not None:
       res += '({})'.format(helper.pi_fractions(op.val))
     if op.is_single():
       res += f' {reg2str(ir, op.idx0)};\n'
     if op.is_ctl():
       res += f' {reg2str(ir, op.ctl)},{reg2str(ir, op.idx1)};\n'
 return res
```

That's it! It is really that simple. Here is an output example:

```
OPENQASM 2.0;
qreg q2[4];
qreg q1[8];
qreg q0[6];
creg c0[8];
h q1[0];
h q1[1];
h q1[2];
[. . .]
cu1(-pi/64) q1[7],q1[1];
cu1(-pi/128) q1[7],q1[0];
h q1[7];
measure q1[0] -> c0[0];
measure q1[1] -> c0[1];
[...]
```

QASM is fairly simple and supported by other infrastructures. Hence, it is very useful for debugging the complex algorithms and comparing the results to those other infrastructures produce.

8.5.8 LIBQ

The generation of sparse `libq` C++ code is similarly trivial. It produces C++, which requires a bit more scaffolding in the beginning and end, but the core function is similar: iterate over all nodes and convert IR nodes to C++. Compiling C++ requires

https://doi.org/10.1017/9781009099974.010 Published online by Cambridge University Press

proper include paths and initialization. Those are stubbed out below and must be set to the specifics of a given build system:

```
def libq(ir) -> str:
  """Dump IR to a compilable C++ program with libq."""

  # Configure: This code needs to change for specific build/run envs.
  res = ('// This file was generated by qc.dump_to_file()\n\n' +
         '#include <math.h>\n' +
         '#include <stdio.h>\n\n' +
         <setup specific headers>
         <setup specific dir>'quantum/libq/libq.h"\n\n' +

         'int main(int argc, char* argv[]) \n' +
         '  <specific init code>\n\n')

  total_regs = 0
  for regs in ir.regset:
    total_regs += regs[1]
  res += f'  libq::qureg* q = libq::new_qureg(0, {total_regs});\n\n'

  total_regs = 0
  for regs in ir.regset:
    for r in regs[2].val:
      if r == 1:
        res += f'  libq::x({total_regs}, q);\n'
      total_regs += 1
  res += '\n'

  for op in ir.gates:
    if op.is_gate():
      res += f'  libq::{op.name}('
      if op.is_single():
        res += f'{op.idx0}'
        if op.val is not None:
          res += ', {}'.format(helper.pi_fractions(op.val, 'M_PI'))
        res += ', q);\n'
      if op.is_ctl():
        res += f'{op.ctl}, {op.idx1}'
        if op.val is not None:
          res += ', {}'.format(helper.pi_fractions(op.val, 'M_PI'))
        res += ', q);\n'
  [...]
```

Here is an example of generated output:

```
int main(int argc, char* argv[]) {
 [...]

 libq::qureg* q = libq::new_qureg(0, 26);
```

https://doi.org/10.1017/9781009099974.010 Published online by Cambridge University Press

```
 libq::x(1, q);
 libq::x(13, q);
 libq::cu1(11, 12, M_PI/2, q);
[...]
 libq::cu1(11, 12, -M_PI/2, q);
 libq::h(12, q);

 libq::flush(q);
 libq::print_qureg(q);
 libq::delete_qureg(q);
 return EXIT_SUCCESS;
}
```

8.5.9 Cirq

The last example is the converter to Google's Cirq. This one is interesting: Because Cirq doesn't support certain gates, we have to construct workarounds as we traverse the IR. Also, the operators need to be renamed (see `op_map` below):

```
def cirq(ir) -> str:
 """Dump IR to a Cirq Python file."""

 res = ('# This file was generated by qc.dump_to_file()\n\n' +
        'import cirq\n' +
        'import cmath\n' +
        'from cmath import pi\n' +
        'import numpy as np\n\n')

 res += 'qc = cirq.Circuit()\n\n'
 res += f'r = cirq.LineQubit.range({ir.nregs})\n'
 res += '\n'

 # Map to translate gate names:
 op_map = {'h': 'H', 'x': 'X', 'y': 'Y', 'z': 'Z',
           'cx': 'CX', 'cz': 'CZ'}

 for op in ir.gates:
   if op.is_gate():
     if op.name == 'u1':
       res += 'm = np.array([(1.0, 0.0), (0.0, '
       res += f'cmath.exp(1j * {helper.pi_fractions(op.val)}))])\n'
       res += f'qc.append(cirq.MatrixGate(m).on(r[{op.idx0}]))\n'
       continue

     # [... similar for cu1, cv, cv_adj]

     op_name = op_map[op.name]
     res += f'qc.append(cirq.{op_name}('
```

https://doi.org/10.1017/9781009099974.010 Published online by Cambridge University Press

```
    if op.is_single():
      res += f'r[{op.idx0}]'
      if op.val is not None:
        res += ', {}'.format(helper.pi_fractions(op.val))
      res += '))\n'
    if op.is_ctl():
      res += f'r[{op.ctl}], r[{op.idx1}]'
      if op.val is not None:
        res += ', {}'.format(helper.pi_fractions(op.val))
      res += '))\n'

res += 'sim = cirq.Simulator()\n'
res += 'print(\'Simulate...\')\n'
res += 'result = sim.simulate(qc)\n'
res += 'res_str = str(result)\n'
res += 'print(res_str.encode(\'utf-8\'))\n'
return res
```

Writing a transpiler does not appear overly complicated.[5] At the time of writing, other transpilers are gestating in open-source, such as one for LaTeX. They were used for a few of the circuits in this book, both to evaluate performance and to generate circuit diagrams.

8.5.10 Open-Source Simulators

We discussed the basic principles of how to construct an efficient, but still bare-bones simulator. With the help of our transcoding facilities, we can now target other available simulators, for example, to utilize simulators that support distributed computation or advanced noise models. In this section, we provide a cross-section of the most cited and well-developed simulators. A more exhaustive list of simulators can be found in Quantiki (2021).

The full-state simulator qHipster implements threading, vectorization, and distributed computation via MPI and OpenMP (Smelyanskiy et al., 2016). It also uses highly optimized libraries on Intel platforms. At the time of writing, the simulator was rebranded as Intel Quantum Simulator (Guerreschi et al., 2020), available on Github (Intel, 2021). This simulator also allows the modeling of quantum noise processes, which enables the simulation of quantum hardware subject to these noise processes. This also mimics the sampling process that real quantum hardware requires.

The only sparse implementation we are aware of is libquantum (Butscher and Weimer, 2013). We used it as the foundation for our `libq`. The library is no longer actively maintained (the last release was in 2013). It offers excellent single-thread performance for circuits where the maximum number of states with non-zero amplitudes is only a small fraction of all possible states. It also makes provisions for quantum error correction and allows modeling of decoherence effects.

[5] A sentence likely falling in the category of "famous last words."

https://doi.org/10.1017/9781009099974.010 Published online by Cambridge University Press

QX is an open-source, high-performance simulator implementation (Khammassi et al., 2017). It accepts as input *quantum code*, which is a variation on QASM that supports explicit parallelism between gates, debug print statements, and looping constructs. It performs aggressive optimizations but still appears to store the full state vector. QX also supports noisy execution using a variety of error models. It is part of a larger quantum development environment from the University of Delft.

ProjectQ is a Python-embedded, compiler-supported framework for quantum computing (Steiger et al., 2018). It allows targeting of both real hardware and the simulator included in the distribution. The simulator allows "shortcuts" by setting the expected result of an expensive computation without simulating it. ProjectQ's distribution contains transpilers to several other available frameworks. It can call into RevKit (Soeken et al., 2012) to automatically construct reversible oracles from classical gates, a function of great utility.

QuEST, the Quantum Exact Simulation Toolkit, is a full state, multithreaded, distributed, and GPU-accelerated simulator (Jones et al., 2019). It hybridizes MPI and OpenMP and has demonstrated impressive scaling on large supercomputers. It supports state-vector and density matrix simulation, general decoherence channels of any size, general unitaries with any number of control and target qubits, and other advanced facilities like Pauli gadgets and higher-order Trotterisation. The related QuESTlink (Jones and Benjamin, 2020) system allows use of the QuEST features from within the Mathematica package.

Recently, Cirq published two high-performance simulators, qsim and qsimh (Google, 2021d). The former, qsim, targets single machines, while qsimh allows distributed computation via OpenMP. The implementations are vectorized and perform several optimizations, such as gate fusion. The qsim simulator is a full state Schrödinger simulator. The qsimh simulator is a Schrödinger-Feynman simulator (Markov et al., 2018), which trades performance for reduced memory requirements.

Microsoft's quantum development kit offers several simulators, including a full state simulator, several resource estimators, and an accelerated simulator for Clifford gates, which can handle millions of gates (Microsoft QDK Simulators, 2021).

The Qiskit ecosystem offers a range of simulators spanning full state simulators, resource estimation tools, noisy simulations, as well as QASM simulators (Qiskit, 2021).

https://doi.org/10.1017/9781009099974.010 Published online by Cambridge University Press