A systems view of quantum computer engineering

Wednesday, October 27, 2021
Rutgers University
Yipeng Huang
Position statement for this graduate seminar

• Quantum computer engineering has become important.

• Requires computer systems expertise beyond quantum algorithms and quantum device physics.
The role of abstractions in classical computing

What is an abstraction?

What are examples of abstractions?

Why are abstractions good and important?
The role of abstractions in classical computing

What is an abstraction?

What are examples of abstractions?
• APIs, Python, C, assembly, machine code (ISA), CPU-memory (von Neumann), pipeline, gates, binary, discrete time evolution

Why are abstractions good and important?
Hides details so that users & programmers can be creative
The role of abstractions in classical computing

Are abstractions always good?

What are examples of deliberately breaking abstractions?
The role of abstractions in classical computing

Are abstractions always good?

What are examples of deliberately breaking abstractions?

• Python calling C binary (Breaking interpreted high level PL abstraction)
• Assembly code routines (Breaking structured programming abstraction)
• FPGAs (Breaking ISA abstraction)
• ASICs (Breaking von Neumann abstraction)
These two lectures: Broad view of open challenges in quantum computer engineering

- A complete view of full-stack quantum computing.

- In short, challenges are in finding and building abstractions.

- In each layer, why we don’t or can’t have good abstractions right now.

- Recent and rapidly developing field of research.

Figure 1. Overview of the quantum computer system stack.
A Microarchitecture for a Superconducting Quantum Processor. Fu et al.
All the quantum computer abstractions we don’t yet have right now

1. Fault-tolerant, error-corrected quantum computers for algorithms
2. Programming: High level languages to aid algorithm discovery
3. Programming: Facilities for program correctness (debuggers, assertions)
4. Programming: Intermediate representations that aid analysis
5. Simulation: Support to validate and design next-gen quantum computers
6. Architecture: Standard instruction set architectures
7. Architecture: Memory hierarchy to store quantum data
8. Microarchitecture: Abundant and reliable quantum devices
All the quantum computer abstractions we don’t yet have right now

1. Fault-tolerant, error-corrected quantum computers for algorithms
2. Programming: High level languages to aid algorithm discovery
3. Programming: Facilities for program correctness (debuggers, assertions)
4. Programming: Intermediate representations that aid analysis
5. Simulation: Support to validate and design next-gen quantum computers
6. Architecture: Standard instruction set architectures
7. Architecture: Memory hierarchy to store quantum data
8. Microarchitecture: Abundant and reliable quantum devices
<table>
<thead>
<tr>
<th>Primitives</th>
<th>Quantum algorithms</th>
</tr>
</thead>
<tbody>
<tr>
<td>Entanglement protocols</td>
<td>superdense coding / quantum teleportation</td>
</tr>
<tr>
<td>Quantum (random) walks</td>
<td>tree traversal</td>
</tr>
<tr>
<td></td>
<td>graph traversal</td>
</tr>
<tr>
<td></td>
<td>satisfiability</td>
</tr>
<tr>
<td>Adiabatic</td>
<td>Ising spin model</td>
</tr>
<tr>
<td></td>
<td>quantum approximate optimization algorithm</td>
</tr>
<tr>
<td>Variational Quantum Eigensolver</td>
<td>Hamiltonian simulation</td>
</tr>
<tr>
<td>Quantum Fourier Transform (QFT)</td>
<td>phase estimation</td>
</tr>
<tr>
<td></td>
<td>period finding</td>
</tr>
<tr>
<td></td>
<td>order finding</td>
</tr>
<tr>
<td></td>
<td>hidden subgroup problem</td>
</tr>
<tr>
<td></td>
<td>linear algebra</td>
</tr>
<tr>
<td>Amplitude amplification</td>
<td>database search</td>
</tr>
<tr>
<td>Primitives</td>
<td>Quantum algorithms</td>
</tr>
<tr>
<td>------------</td>
<td>--------------------</td>
</tr>
<tr>
<td>Entanglement protocols</td>
<td>superdense coding / quantum teleportation</td>
</tr>
<tr>
<td>Quantum (random) walks</td>
<td>tree traversal</td>
</tr>
<tr>
<td></td>
<td>graph traversal</td>
</tr>
<tr>
<td></td>
<td>satisfiability</td>
</tr>
<tr>
<td>Adiabatic</td>
<td>Ising spin model</td>
</tr>
<tr>
<td></td>
<td>quantum approximate optimization algorithm</td>
</tr>
<tr>
<td>Variational Quantum Eigensolver</td>
<td>Hamiltonian simulation</td>
</tr>
<tr>
<td>Quantum Fourier Transform (QFT)</td>
<td>phase estimation</td>
</tr>
<tr>
<td></td>
<td>period finding</td>
</tr>
<tr>
<td></td>
<td>order finding</td>
</tr>
<tr>
<td></td>
<td>hidden subgroup problem</td>
</tr>
<tr>
<td></td>
<td>linear algebra</td>
</tr>
<tr>
<td>Amplitude amplification</td>
<td>database search</td>
</tr>
</tbody>
</table>

Noisy, intermediate-scale quantum algorithms

Fault tolerant, error corrected quantum algorithms
Shor’s integer factoring algorithm

Factoring underpins cryptosystems.
For number represented as N bits—

Classical algorithm: needs $O(2^{3\sqrt{N}})$ operations
Factoring 512-bit integer: 8400 years. 1024-bit integer: $13\times10^{12}$ years.

Quantum algorithm: needs $O(N^2 \log(N))$ operations
Factoring 512-bit integer: 3.5 hours. 1024-bit integer: 31 hours.

A Practical Architecture for Reliable Quantum Computers. Oskin et al.
The horizontal axis is the length of the number to be factored. The steep curve is NFS, with the marked point at \( L = 768 \) requiring 3.300 CPU-years. The vertical line at \( L = 2048 \) is NIST's 2007 recommendation for RSA key length for data intended to remain secure until 2030. The other lines are various combinations of quantum computer logical clock speed for a three-qubit operation known as a Toffoli gate (1Hz and 1MHz), method of implementing the arithmetic portion of Shor's algorithm (BCDP, D, and F), and quantum computer architecture (NTC and AC, with the primary difference being whether or not long-distance operations are supported). The assumed capacity of a machine in this graph is \( 2^L \) logical qubits. This figure illustrates the difficulty of making pronouncements about the speed of quantum computers.
Fig. 2. Performance space of quantum computers, measured by the error probability of each entangling gate in the horizontal axis (roughly inversely proportional to the total number of gates that can be executed on a NISQ machine), and the number of qubits in the system in the vertical axis. Blue dotted line approximately demarcates quantum systems that can be simulated using best classical computers, while the green colored region shows where the existing quantum computing systems with verified performance numbers lie (as of September 2018). Purple shaded region indicates computational tasks that accomplish the so-called “quantum supremacy,” where the computation carried out by the quantum computer defies classical simulation regardless of its usefulness. The different shapes illustrate resource counts for solving various problems, with solid symbols corresponding to the exact entangling gate counts and number of qubits in NISQ machines, and shaded regions showing approximate gate error requirements and number of qubits for an FT implementation (not pictured are the regions where the error gets too close to the known fault-tolerance thresholds): cyan diamond and shaded region correspond to factoring a 1024-bit number using Shor’s algorithm [14], magenta circle and shaded region represent simulation of a 72-spin Heisenberg model [20], and orange shaded region illustrates NF simulation [21].

**Fig. 2.** Performance space of quantum computers, measured by the error probability of each entangling gate in the horizontal axis (roughly inversely proportional to the total number of gates that can be executed on a NISQ machine), and the number of qubits in the system in the vertical axis. Blue dotted line approximately demarcates quantum systems that can be simulated using best classical computers, while the green colored region shows where the existing quantum computing systems with verified performance numbers lie (as of September 2018). Purple shaded region indicates computational tasks that accomplish the so-called “quantum supremacy,” where the computation carried out by the quantum computer defies classical simulation regardless of its usefulness. The different shapes illustrate resource counts for solving various problems, with solid symbols corresponding to the exact entangling gate counts and number of qubits in NISQ machines, and shaded regions showing approximate gate error requirements and number of qubits for an FT implementation (not pictured are the regions where the error gets too close to the known fault-tolerance thresholds): cyan diamond and shaded region correspond to factoring a 1024-bit number using Shor’s algorithm [14], magenta circle and shaded region represent simulation of a 72-spin Heisenberg model [20], and orange shaded region illustrates NF simulation [21].
Need applications to motivate further research.

Fig. 2. Performance space of quantum computers, measured by the error probability of each entangling gate in the horizontal axis (roughly inversely proportional to the total number of gates that can be executed on a NISQ machine), and the number of qubits in the system in the vertical axis. Blue dotted line approximately demarcates quantum systems that can be simulated using best classical computers, while the green colored region shows where the existing quantum computing systems with verified performance numbers lie (as of September 2018). Purple shaded region indicates computational tasks that accomplish the so-called “quantum supremacy,” where the computation carried out by the quantum computer defies classical simulation regardless of its usefulness. The different shapes illustrate resource counts for solving various problems, with solid symbols corresponding to the exact entangling gate counts and number of qubits in NISQ machines, and shaded regions showing approximate gate error requirements and number of qubits for an FT implementation (not pictured are the regions where the error gets too close to the known fault-tolerance thresholds): cyan diamond and shaded region correspond to factoring a 1024-bit number using Shor's algorithm [14], magenta circle and shaded region represent simulation of a 72-spin Heisenberg model [20], and orange shaded region illustrates NF simulation [21].
<table>
<thead>
<tr>
<th>Primitives</th>
<th>Quantum algorithms</th>
</tr>
</thead>
<tbody>
<tr>
<td>Entanglement protocols</td>
<td>superdense coding / quantum teleportation</td>
</tr>
<tr>
<td>Quantum (random) walks</td>
<td>tree traversal</td>
</tr>
<tr>
<td>Adiabatic</td>
<td>graph traversal</td>
</tr>
<tr>
<td></td>
<td>satisfiability</td>
</tr>
<tr>
<td>Variational Quantum</td>
<td>Ising spin model</td>
</tr>
<tr>
<td>Eigensolver</td>
<td>quantum approximate optimization algorithm</td>
</tr>
<tr>
<td>Quantum Fourier Transform (QFT)</td>
<td>phase estimation</td>
</tr>
<tr>
<td></td>
<td>period finding</td>
</tr>
<tr>
<td></td>
<td>order finding</td>
</tr>
<tr>
<td></td>
<td>hidden subgroup problem</td>
</tr>
<tr>
<td></td>
<td>linear algebra</td>
</tr>
<tr>
<td>Amplitude amplification</td>
<td>database search</td>
</tr>
</tbody>
</table>

**Noisy, intermediate-scale quantum algorithms**

**Fault tolerant, error corrected quantum algorithms**

All the quantum computer abstractions we don’t yet have right now

1. Fault-tolerant, error-corrected quantum computers for algorithms
2. Programming: High level languages to aid algorithm discovery
3. Programming: Facilities for program correctness (debuggers, assertions)
4. Programming: Intermediate representations that aid analysis
5. Simulation: Support to validate and design next-gen quantum computers
6. Architecture: Standard instruction set architectures
7. Architecture: Memory hierarchy to store quantum data
8. Microarchitecture: Abundant and reliable quantum devices
Underappreciated fact: Programming languages aid discovery of algorithms
Underappreciated fact:
Programming languages aid discovery of algorithms

https://www.geeksforgeeks.org/random-walk-implementation-python/

Take classical random walks as an example. Notice:
1. Ease of going from 1D example to 2D example (reusable code).
2. Ease of generating a visualization.
3. Code and simulation reveals properties useful for new algorithms.
Underappreciated fact:
Programming languages aid discovery of algorithms
Specification of algorithm as a procedure

https://www.geeksforgeeks.org/random-walk-implementation-python/

Take classical random walks as an example. Notice:

1. Import library for random coin toss
2. Data structures for time series
3. Standard operators for increment and decrement
What do we mean by high-level language?

// Create a list of strings
ArrayList<String> al = new ArrayList<String>();
al.add("Quantum");
al.add("Computing");
al.add("Programs");
al.add("Systems");

/* Collections.sort method is sorting the elements of ArrayList in ascending order. */
Collections.sort(al);
High-level language hides all these concerns:

• Choice of sorting algorithm implementation and comparator function on Strings
• Encoding of Strings as binary numbers
• Memory allocation and deallocation for String storage
• Execution of Java code on different ISAs
• Correctness of library implementation
• …
• We want to get to that point with quantum programming

• Want to do things such as: initialize quantum data, perform search, without worrying about the detailed implementation
Programmers’ time is scarce. Classical computing resources are abundant. Nonetheless, abstractions are expensive.

Table 1. Speedups from performance engineering a program that multiplies two 4096-by-4096 matrices. Each version represents a successive refinement of the original Python code. “Running time” is the running time of the version. “GFLOPS” is the billions of 64-bit floating-point operations per second that the version executes. “Absolute speedup” is time relative to Python, and “relative speedup,” which we show with an additional digit of precision, is time relative to the preceding line. “Fraction of peak” is GFLOPS relative to the computer’s peak 835 GFLOPS. See Methods for more details.

<table>
<thead>
<tr>
<th>Version</th>
<th>Implementation</th>
<th>Running time (s)</th>
<th>GFLOPS</th>
<th>Absolute speedup</th>
<th>Relative speedup</th>
<th>Fraction of peak (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Python</td>
<td>25,552.48</td>
<td>0.005</td>
<td>1</td>
<td>—</td>
<td>0.00</td>
</tr>
<tr>
<td>2</td>
<td>Java</td>
<td>2,372.68</td>
<td>0.058</td>
<td>1</td>
<td>10.8</td>
<td>0.01</td>
</tr>
<tr>
<td>3</td>
<td>C</td>
<td>542.67</td>
<td>0.253</td>
<td>1</td>
<td>47</td>
<td>4.4</td>
</tr>
<tr>
<td>4</td>
<td>Parallel loops</td>
<td>69.80</td>
<td>1.969</td>
<td>1</td>
<td>47</td>
<td>4.4</td>
</tr>
<tr>
<td>5</td>
<td>Parallel divide and conquer</td>
<td>3.80</td>
<td>36.180</td>
<td>100</td>
<td>18.4</td>
<td>4.33</td>
</tr>
<tr>
<td>6</td>
<td>plus vectorization</td>
<td>1.10</td>
<td>124.914</td>
<td>124.914</td>
<td>3.5</td>
<td>14.96</td>
</tr>
<tr>
<td>7</td>
<td>plus AVX intrinsics</td>
<td>0.41</td>
<td>337.812</td>
<td>337.812</td>
<td>3.5</td>
<td>14.96</td>
</tr>
</tbody>
</table>

Programming languages aid discovery of algorithms: Is it currently true for quantum computer science?

Only ~100 known quantum algorithms
Specification of quantum algorithm as a quantum procedure

As another example, QAOA on Cirq exercise: https://rutgers.instructure.com/courses/73314/assignments/1017995

Need quantum equivalent of:
1. Partition representation
2. Edge constraints
3. Way to perturb partitioning

Gates to code that we can run.
Specification of quantum algorithm as a quantum procedure

**Functional quantum programming languages**
*(emphasis on specifying the mathematics)*
- Microsoft Liquid
- Quipper
- QWIRE
- Microsoft Q#
- Etc.

**Imperative quantum programming languages**
*(emphasis on specifying the resources)*
- Scaffold
- Microsoft ProjectQ
- Rigetti PyQuil
- IBM Qiskit
- Google Cirq
- Etc.
Programming languages aid discovery of algorithms: Is it currently true for quantum computer science?

Only ~100 known quantum algorithms

Some progress in the past two years. Qiskit Aqua OpenFermion-Cirq TensorFlow Quantum

Given the implementation for Deutsch’s algorithm, is it obvious how to implement Deutsch-Jozsa’s algorithm?
Table 7 Grover’s amplitude amplification subroutine in two languages, showcasing QC-specific language syntax for reversible computation (rows 2 & 6) and controlled operations (rows 3 & 5).

<table>
<thead>
<tr>
<th>Scaffold (C syntax) [13]</th>
<th>ProjectQ (Python syntax) [36]</th>
</tr>
</thead>
<tbody>
<tr>
<td>`int j;</td>
<td># reflection across</td>
</tr>
<tr>
<td>qbit ancilla[n-1]; // scratch register</td>
<td># uniform superposition</td>
</tr>
<tr>
<td>for(j=0; j&lt;n-1; j++) PrepZ(ancilla[j],0);</td>
<td></td>
</tr>
<tr>
<td>// Hadamard on q</td>
<td>with Compute(eng):</td>
</tr>
<tr>
<td>for(j=0; j&lt;n; j++) H(q[j]);</td>
<td>All(H)</td>
</tr>
<tr>
<td>// Phase flip on q = 0...0 so invert q</td>
<td></td>
</tr>
<tr>
<td>for(j=0; j&lt;n; j++) X(q[j]);</td>
<td>with Control(eng, q[0:-1]):</td>
</tr>
<tr>
<td>// Compute x[n-2] = q[0] and ... and q[n-1]</td>
<td></td>
</tr>
<tr>
<td>CCNOT(q[1], q[0], ancilla[0]);</td>
<td></td>
</tr>
<tr>
<td>for(j=1; j&lt;n-1; j++)</td>
<td></td>
</tr>
<tr>
<td>CCNOT(ancilla[j-1], q[j+1], ancilla[j]);</td>
<td></td>
</tr>
<tr>
<td>// Phase flip Z if q=00...0</td>
<td>Z</td>
</tr>
<tr>
<td>cZ(ancilla[n-2], q[n-1]);</td>
<td></td>
</tr>
<tr>
<td>// Undo the local registers</td>
<td></td>
</tr>
<tr>
<td>for(j=n-2; j&gt;0; j--)</td>
<td># ProjectQ automatically</td>
</tr>
<tr>
<td>CCNOT(ancilla[j-1], q[j+1], ancilla[j]);</td>
<td># uncomputes control</td>
</tr>
<tr>
<td>CCNOT(q[1], q[0], ancilla[0]);</td>
<td></td>
</tr>
<tr>
<td>// Restore q</td>
<td></td>
</tr>
<tr>
<td>for(j=0; j&lt;n; j++) X(q[j]);</td>
<td>Uncompute(eng)</td>
</tr>
<tr>
<td>for(j=0; j&lt;n; j++) H(q[j]);</td>
<td></td>
</tr>
</tbody>
</table>
All the quantum computer abstractions we don’t yet have right now

1. Fault-tolerant, error-corrected quantum computers for algorithms
2. Programming: High level languages to aid algorithm discovery
3. Programming: Facilities for program correctness (debuggers, assertions)
4. Programming: Intermediate representations that aid analysis
5. Simulation: Support to validate and design next-gen quantum computers
6. Architecture: Standard instruction set architectures
7. Architecture: Memory hierarchy to store quantum data
8. Microarchitecture: Abundant and reliable quantum devices
Approaches to Software Reliability

- Social
  - Code reviews
  - Extreme/Pair programming

- Methodological
  - Design patterns
  - Test-driven development
  - Version control
  - Bug tracking

- Technological
  - “lint” tools, static analysis
  - Fuzzers, random testing

- Mathematical
  - Sound type systems
  - Formal verification

Less "formal": Lightweight, inexpensive techniques (that may miss problems)

This isn’t an either/or tradeoff… a spectrum of methods is needed!

Even the most “formal” argument can still have holes:
- Did you prove the right thing?
- Do your assumptions match reality?

  Knuth: “Beware of bugs in the above code; I have only proved it correct, not tried it.”

More “formal”: eliminate with certainty as many problems as possible.

From: https://www.seas.upenn.edu/~cis500/current/lectures/lec01.pdf
Execution in target machine w/ facilities for validation

• Exception handling.
• Printf debugging.
• GDB breakpoints.
• Assertions.
• Etc.
• “Exception handling” (i.e., quantum error correction) is costly.
• No printf. No intermediate measurements.
• Can’t set arbitrary breakpoints.
• Few obvious assertions.
Even simple quantum programming bugs lead to non-obvious symptoms.

\[
\begin{align*}
 q_0 & \quad = \quad q_0 \\
 q_1 & \quad = \quad q_1 \\
 U & \quad = \quad C \oplus B \oplus A
\end{align*}
\]
Even simple quantum programming bugs lead to non-obvious symptoms

Rz(q1, +angle/2); // C
CNOT(q0, q1);
Rz(q1, -angle/2); // B
CNOT(q0, q1);
Rz(q0, +angle/2); // D
Even simple quantum programming bugs lead to non-obvious symptoms

Elementary single-qubit operations

Rz(q1, +angle/2); // C
CNOT(q0, q1);
Rz(q1, -angle/2); // B
CNOT(q0, q1);
Rz(q0, +angle/2); // D
Even simple quantum programming bugs lead to non-obvious symptoms.

Elementary two-qubit operations

Rz(q1, +angle/2); // C
CNOT(q0, q1);
Rz(q1, -angle/2); // B
CNOT(q0, q1);
Rz(q0, +angle/2); // D

Correct, operation A unneeded
Even simple quantum programming bugs lead to non-obvious symptoms

\[ R_z(q_1, +\text{angle}/2); // C \]
\[ \text{CNOT}(q_0, q_1); \]
\[ R_z(q_1, -\text{angle}/2); // B \]
\[ \text{CNOT}(q_0, q_1); \]
\[ R_z(q_0, +\text{angle}/2); // D \]

Correct, operation A unneeded

Unneeded
Even simple quantum programming bugs lead to non-obvious symptoms

Rz(q1, +angle/2); // C
CNOT(q0, q1);
Rz(q1, -angle/2); // B
CNOT(q0, q1);
Rz(q0, +angle/2); // D

Correct, operation A unneeded

<table>
<thead>
<tr>
<th>Operation 1</th>
<th>Operation 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rz(q0, q1);</td>
<td>CNOT(q0, q1);</td>
</tr>
<tr>
<td>Rz(q1, -angle/2); // B</td>
<td>CNOT(q0, q1);</td>
</tr>
<tr>
<td>Rz(q0, q1);</td>
<td>Rz(q1, +angle/2); // A</td>
</tr>
<tr>
<td>Rz(q0, +angle/2); // D</td>
<td>Rz(q0, +angle/2); // D</td>
</tr>
</tbody>
</table>

Correct, operation C unneeded
Even simple quantum programming bugs lead to non-obvious symptoms

Rz(q1, +angle/2); // C
CNOT(q0, q1);
Rz(q1, -angle/2); // B
CNOT(q0, q1);
Rz(q0, +angle/2); // D

Correct, operation A unneeded

Rz(q0, +angle/2); // C
CNOT(q0, q1);
Rz(q1, -angle/2); // B
CNOT(q0, q1);
Rz(q0, +angle/2); // D

Correct, operation C unneeded

Rz(q1, -angle/2);
CNOT(q0, q1);
Rz(q1, +angle/2);
CNOT(q0, q1);
Rz(q0, +angle/2); // D

Incorrect, angles flipped

Many ways to translate basic quantum operations to program code—many details to get right!
Even simple quantum programming bugs lead to non-obvious symptoms

\[ |11\rangle \rightarrow e^{i \angle} |11\rangle \]

Correct, operation A unneeded

Correct, operation C unneeded

Incorrect, angles flipped

\[ |11\rangle \rightarrow e^{-i \angle} |11\rangle \]
Even simple quantum programming bugs lead to non-obvious symptoms

\[
|11\rangle \rightarrow e^{i\angle} |11\rangle
\]

Correct, operation A unneeded

Correct, operation C unneeded

Incorrect, angles flipped
Even simple quantum programming bugs lead to non-obvious symptoms

\[
|11 \rangle \rightarrow e^{i \times \text{angle}} |11 \rangle
\]
Detailed debugging of Shor’s factorization algorithm

QDB: From Quantum Algorithms Towards Correct Quantum Programs
Yipeng Huang, Margaret Martonosi | Princeton University

Image credit: Metodi, Faruque, and Chong, Quantum Computing for Computer Architects, 2nd Ed., p26
Detailed debugging of Shor’s factorization algorithm

Image credit: Metodi, Faruque, and Chong, Quantum Computing for Computer Architects, 2nd Ed., p26

QDB: From Quantum Algorithms Towards Correct Quantum Programs
Yipeng Huang, Margaret Martonosi | Princeton University

control register (N qubits)
|0> ⊗ N

Quantum part of algorithm

QFT

inverse qft

controlled function

Target register (N qubits)
|1>

Classical post processing

measurement (classical result)

classical continued fractions algorithm

result: order of f(x) with high prob.
Bug type 3-B: mistake in composing gates using mirroring

Mirror image submodules

Control Register ($N$ qubits) \{ $|0\rangle \otimes N$ \}

QFT

Inverse QFT

Measurement (Classical result)

Classical Continued Fractions Algorithm

Result: Order of $f(x)$ with high prob.

Target Register ($N$ qubits) \{ $|1\rangle$ \}

Controlled Function
#include "QFT.scaffold"
#define width 4 // number of qubits
int main () {
    // initialize quantum variable to 5
    qbit reg[width];
    for ( int i=0; i<width; i++ ) {
        PrepZ ( reg[i], (i+1)%2 ); // 0b0101
    }

    // precondition for QFT:
    assert_classical ( reg, width, 5 );
    QFT ( width, reg );

    // postcondition for QFT &
    // precondition for iQFT:
    assert_superposition ( reg, width );
    iQFT ( width, reg );

    // postcondition for iQFT:
    assert_classical ( reg, width, 5 );
}

Listing 1: Test harness for quantum Fourier transform.

Assertions on classical & superposition states help us decide whether programs are correct

Testbench for quantum Fourier transform, consisting of controlled-rotations
Assertions on classical & superposition states help us decide whether programs are correct

Testbench for quantum Fourier transform, consisting of controlled-rotations

QFT and iQFT should be inverses, but bug in controlled-rotations would lead to flawed inversion

Listing 1: Test harness for quantum Fourier transform.
Assertions on classical & superposition states help us decide whether programs are correct

Testbench for quantum Fourier transform, consisting of controlled-rotations

QFT and iQFT should be inverses, but bug in controlled-rotations would lead to flawed inversion

Listing 1: Test harness for quantum Fourier transform.
Assertions on classical & superposition states help us decide whether programs are correct.

Testbench for quantum Fourier transform, consisting of controlled-rotations.

QFT and iQFT should be inverses, but bug in controlled-rotations would lead to flawed inversion.

Flawed inversion caught in failure of classical assertion based on Chi-squared tests.

Listing 1: Test harness for quantum Fourier transform.
Assertions on classical & superposition states help us decide whether programs are correct

Testbench for quantum Fourier transform, consisting of controlled-rotations

QFT and iQFT should be inverses, but bug in controlled-rotations would lead to flawed inversion

Flawed inversion caught in failure of classical assertion based on Chi-squared tests
Toolchain for debugging programs with tests on measurements

**Breakpoint annotations**

Quantum assembly code

Simulate / execute on prototype quantum computer

Ensemble of measurements

**Assertion annotations**

**Statistical test:** Is there a bug?
Quantum program bug types

1. Quantum initial values
2. Basic operations
3. Composing operations
   A. Iteration
   B. Mirroring
4. Classical input parameters
5. Garbage collection of qubits

Defenses, debugging, and assertions

1. Preconditions
2. Subroutines / unit tests
3. Quantum specific language support
   A. Numeric data types
   B. Reversible computation
4. Algorithm progress assertions
5. Postconditions

A first taxonomy of quantum program bugs and defenses.
All the quantum computer abstractions we don’t yet have right now

1. Fault-tolerant, error-corrected quantum computers for algorithms
2. Programming: High level languages to aid algorithm discovery
3. Programming: Facilities for program correctness (debuggers, assertions)
4. Programming: Intermediate representations that aid analysis
5. Simulation: Support to validate and design next-gen quantum computers
6. Architecture: Standard instruction set architectures
7. Architecture: Memory hierarchy to store quantum data
8. Microarchitecture: Abundant and reliable quantum devices
Benchmarking ZX Calculus Circuit Optimization Against Qiskit Transpilation. Yeh et al.

| Qubits take on binary values; supply initial qubit values | q_{0m0} = |0⟩ \oplus q_{0m0} = |1⟩ | q_{1m0} = |0⟩ \oplus q_{1m0} = |1⟩ |
|----------------------------------------------------------|-------------------------------------|-------------------------------------|
| q_{0m1} = |0⟩ \oplus q_{0m1} = |1⟩ | q_{1m1} = |0⟩ \oplus q_{1m1} = |1⟩ |

| Hadamard gate | q_{0m0} = |0⟩ \land q_{0m1} = |0⟩ \implies +\frac{1}{\sqrt{2}} | q_{0m0} = |0⟩ \land q_{0m1} = |1⟩ \implies +\frac{\sqrt{2}}{2} |
|---------------|-------------------------------------|-------------------------------------|
| q_{0m0} = |1⟩ \land q_{0m1} = |0⟩ \implies +\frac{\sqrt{2}}{2} | q_{0m0} = |1⟩ \land q_{0m1} = |1⟩ \implies -\frac{1}{\sqrt{2}} |

| Phase damping noise channel | q_{0m2rv} = 0 \land q_{0m2rv} = 1 | q_{0m1} = |1⟩ \land q_{0m2rv} = 0 \implies +0.8 |
|----------------------------|-------------------------------------|-------------------------------------|
| q_{0m1} = |0⟩ \implies q_{0m2rv} = 0 | q_{0m1} = |1⟩ \land q_{0m2rv} = 1 \implies -0.6 |

| CNOT gate | q_{0m1} = |0⟩ \land q_{1m0} = |0⟩ \implies q_{1m3} = |0⟩ | q_{0m1} = |1⟩ \land q_{1m0} = |0⟩ \implies q_{1m3} = |1⟩ |
|-----------|-------------------------------------|-------------------------------------|
| q_{0m1} = |0⟩ \land q_{1m0} = |1⟩ \implies q_{1m3} = |1⟩ | q_{0m1} = |1⟩ \land q_{1m0} = |1⟩ \implies q_{1m3} = |0⟩ |
All the quantum computer abstractions we don’t yet have right now

1. Fault-tolerant, error-corrected quantum computers for algorithms
2. Programming: High level languages to aid algorithm discovery
3. Programming: Facilities for program correctness (debuggers, assertions)
4. Programming: Intermediate representations that aid analysis
5. Simulation: Support to validate and design next-gen quantum computers
6. Architecture: Standard instruction set architectures
7. Architecture: Memory hierarchy to store quantum data
8. Microarchitecture: Abundant and reliable quantum devices
Role of simulation in classical computer engineering

- VirtualBox
- Gem5
- Synopsys / Cadence
- Spice
Why is simulation important?

• "Developing good classical simulations (or even attempting to and failing) would also help clarify the quantum/classical boundary.”
  —Aram Harrow

• Development and debugging of quantum algorithm implementations
Classical simulations of quantum computing

**FIG. S40. Scaling of the computational cost of XEB using SA and SFA.** a, For a Schrödinger algorithm, the limitation is RAM size, shown as vertical dashed line for the Summit supercomputer. Circles indicate full circuits with $n = 12$ to 43 qubits that are benchmarked in Fig. 4a of the main paper. 53 qubits would exceed the RAM of any current supercomputer, and shown as a star. b, For the hybrid Schrödinger-Feynman algorithm, which is more memory efficient, the computation time scales exponentially in depth. XEB on full verifiable circuits was done at depth $m = 14$ (circle). c, XEB on full supremacy circuits is out of reach within reasonable time resources for $m = 12, 14, 16$ (stars), and beyond. XEB on patch and elided supremacy circuits was done at $m = 14, 16, 18,$ and 20.
• Until we have quantum computer systems, building and testing quantum computers will rely on classical computer systems.
All the quantum computer abstractions we don’t yet have right now

1. Fault-tolerant, error-corrected quantum computers for algorithms
2. Programming: High level languages to aid algorithm discovery
3. Programming: Facilities for program correctness (debuggers, assertions)
4. Programming: Intermediate representations that aid analysis
5. Simulation: Support to validate and design next-gen quantum computers
6. Architecture: Standard instruction set architectures
7. Architecture: Memory hierarchy to store quantum data
8. Microarchitecture: Abundant and reliable quantum devices
A small set of quantum gates are universal...

\[ |0\rangle \rightarrow \cos \left( \frac{\theta}{2} \right) |0\rangle + e^{i\phi} \sin \left( \frac{\theta}{2} \right) |1\rangle \]
\[ |1\rangle \rightarrow \cos \left( \frac{\theta}{2} \right) |1\rangle - e^{-i\phi} \sin \left( \frac{\theta}{2} \right) |0\rangle \]

FIG. 1. The rotation and controlled-NOT (CNOT) gates are an example of a universal quantum gate family when available on all qubits, with explicit evolution (above) and quantum circuit block schematics (below). (a) The single-qubit rotation gate $R(\theta, \phi)$, with two continuous parameters $\theta$ and $\phi$, evolves input qubit state $|x\rangle$ to output state $|\bar{x}\rangle$. (b) The CNOT (or reversible XOR) gate on two qubits evolves two (control and target) input qubit states $|x_C\rangle$ and $|x_T\rangle$ to output states $|\bar{x}_C = x_C\rangle$ and $|\bar{x}_T = x_C \oplus x_T\rangle$, where $\oplus$ is addition modulo 2, or equivalently the XOR operation.

Quantum Computer Systems for Scientific Discovery. Alexeev et al.
...but those universal gates decompose various ways...

**BOX 2. CHOOSING A CNOT GATE DECOMPOSITION**

A logical controlled-NOT, or CNOT, gate is an entangling operation that flips the target qubit between 1 and 0 if the control qubit is in the 1 state. It must be decomposed into a sequence of native quantum gates for the qubit technology to perform the gate operation on the specific qubit system. Two possible decompositions are shown, where RX, RY, and RZ denote rotations around the x-, y- and z-axes, respectively. A system performance simulation could provide metrics to help choose which CNOT to incorporate into the specific design.

Depending on the fidelity of the single- and two-qubit gates in the circuits and on their speed, researchers may want to choose only one to implement the logical CNOT in the system. Depending on the performance of the qubits available at a particular point in the execution of the algorithm, it is also possible to choose a different logical CNOT sequence.

*A systems perspective of quantum computing. Matsuura et al.*
...and different quantum hardware support different gates.

<table>
<thead>
<tr>
<th>Software Visible Gates</th>
<th>Native Gates</th>
<th>Qubits</th>
</tr>
</thead>
<tbody>
<tr>
<td>1Q</td>
<td>2Q</td>
<td></td>
</tr>
<tr>
<td>$R_{xy}(\theta, \varphi)$</td>
<td>$XX(x)$</td>
<td></td>
</tr>
<tr>
<td>$R_z(\lambda)$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1Q</td>
<td>2Q</td>
<td></td>
</tr>
<tr>
<td>$R_{xy}(\theta, \varphi)$</td>
<td>$XX(x)$</td>
<td></td>
</tr>
<tr>
<td>$R_z(\lambda)$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>University of Maryland</td>
<td>IBM</td>
<td>Rigetti</td>
</tr>
<tr>
<td>Yb'ion trapped in EM field</td>
<td>Superconducting Josephson Junction</td>
<td>Superconducting Josephson Junction</td>
</tr>
</tbody>
</table>

*Figure 1.* Hardware qubit technology, native gate set, and software-visible gate set in the systems used in our study. Each qubit technology lends itself to a set of native gates. For programming, vendors expose these gates in a software-visible interface or construct composite gates with multiple native gates.

All the quantum computer abstractions we don’t yet have right now

1. Fault-tolerant, error-corrected quantum computers for algorithms
2. Programming: High level languages to aid algorithm discovery
3. Programming: Facilities for program correctness (debuggers, assertions)
4. Programming: Intermediate representations that aid analysis
5. Simulation: Support to validate and design next-gen quantum computers
6. Architecture: Standard instruction set architectures
7. Architecture: Memory hierarchy to store quantum data
8. Microarchitecture: Abundant and reliable quantum devices
Fig. 1. Examples of several IBM cloud accessible devices. The top left 5-qubit device was the first one made available via the IBM Quantum Experience [40]. The one to the right of it was made available after including additional entangling gates between two pairs of qubits. A 16-qubit device was made available approximately a year after the first device. The devices in the bottom row show three variations of 20-qubit devices available to members of the IBM Q Network [41].
Compiling quantum program abstractions to optimal quantum execution

Compilation for maximum correctness, while respecting constraints:
• variable qubit, operation, measurement reliability
• connectivity constraints
• parallelism

Will be topic of chapter on “extracting success.”
All the quantum computer abstractions we don’t yet have right now

1. Fault-tolerant, error-corrected quantum computers for algorithms
2. Programming: High level languages to aid algorithm discovery
3. Programming: Facilities for program correctness (debuggers, assertions)
4. Programming: Intermediate representations that aid analysis
5. Simulation: Support to validate and design next-gen quantum computers
6. Architecture: Standard instruction set architectures
7. Architecture: Memory hierarchy to store quantum data
8. Microarchitecture: Abundant and reliable quantum devices
Fig. 2. Controlled-NOT (CNOT) gate error distributions for a variety of IBM devices. Beginning with the earlier devices (top rows), the average error rates remained quite large but have improved with continuing research. The bottom row represents the device shown in Fig. 4. The error reductions are the result of improved gate fidelities and increasing coherence times [44], [45], as well as a better understanding of spectator qubit errors.

Challenges and Opportunities of Near-Term Quantum Computing Systems. Corcoles et al.
FIGURE 7.2 The number of qubits in superconductor (SC) and trapped ion (TI) quantum computers versus year; note the logarithmic scaling of the vertical axis. Data for trapped ions are shown as squares and for superconducting machines are shown as circles. Approximate average reported two-qubit gate error rates are indicated by color; points with the same color have similar error rates. The dashed gray lines show how the number of qubits would grow if they double every two years starting with one qubit in 2000 and 2009, respectively; the dashed black line indicates a doubling every year beginning with one qubit in 2014. Recent superconductor growth has been close to doubling every year. If this rate continued, 50 qubit machines with less than 5 percent error rates would be reported in 2019. SOURCE: Plotted data obtained from multiple sources [9].

FIGURE 7.2 The number of qubits in superconductor (SC) and trapped ion (TI) quantum computers versus year; note the logarithmic scaling of the vertical axis. Data for trapped ions are shown as squares and for superconducting machines are shown as circles. Approximate average reported two-qubit gate error rates are indicated by color; points with the same color have similar error rates. The dashed gray lines show how the number of qubits would grow if they double every two years starting with one qubit in 2000 and 2009, respectively; the dashed black line indicates a doubling every year beginning with one qubit in 2014. Recent superconductor growth has been close to doubling every year. If this rate continued, 50 qubit machines with less than 5 percent error rates would be reported in 2019. SOURCE: Plotted data obtained from multiple sources [9].
Fig. 2: Number of components per integrated function for minimum cost per component extrapolated vs. time.
Moore’s Law – The number of transistors on integrated circuit chips (1971-2018)

Moore’s law describes the empirical regularity that the number of transistors on integrated circuits doubles approximately every two years. This advancement is important as other aspects of technological progress – such as processing speed or the price of electronic products – are linked to Moore’s law.
Position statement for this graduate seminar

• Quantum computer engineering has become important.

• Requires computer systems expertise beyond quantum algorithms and quantum device physics.