Emerging languages and representations for quantum computing

Monday, October 5, 2020
Rutgers University
Yipeng Huang
Progress and questions about QAOA on Cirq?

https://rutgers.instructure.com/courses/73314/assignments/1017995
Position statement for this graduate seminar

• Quantum computer engineering has become important.

• Requires computer systems expertise beyond quantum algorithms and quantum device physics.
All the quantum computer abstractions we don’t yet have right now

0. Quantum computer support for quantum computer engineering
1. Fault-tolerant, error-corrected quantum algorithms
2. **Mature, high level quantum programming languages**
3. Universally accepted quantum ISAs
4. Uniform, fully connected quantum device architectures
5. Reliable quantum gates and qubits
Quantum programming and correctness

I. Correct quantum programs
   A. Verification (proofs)
   B. Validation (debugging & assertions)

II. Quantum programming: from abstractions to execution
   A. Creation / discovery of algorithms
   B. Specification of algorithm as a procedure
   C. Compilation to native instructions while maximizing performance
   D. Execution in target machine with facilities for validation
Quantum programming and correctness

I. **Correct quantum programs**
   A. Verification (proofs)
   B. Validation (debugging & assertions)

II. Quantum programming: from abstractions to execution
   A. Creation / discovery of algorithms
   B. Specification of algorithm as a procedure
   C. Compilation to native instructions while maximizing performance
   D. Execution in target machine with facilities for validation
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
- “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
Even simple quantum programming bugs lead to non-obvious symptoms.
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

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

\[
\begin{align*}
\text{Rz}(q_1, +\text{angle}/2); & \quad \text{// C} \\
\text{CNOT}(q_0, q_1); & \\
\text{Rz}(q_1, -\text{angle}/2); & \quad \text{// B} \\
\text{CNOT}(q_0, q_1); & \\
\text{Rz}(q_0, +\text{angle}/2); & \quad \text{// D} \\
\text{Rz}(q_1, +\text{angle}/2); & \quad \text{// A} \\
\text{Rz}(q_0, +\text{angle}/2); & \quad \text{// D}
\end{align*}
\]

Correct, operation A unneeded

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

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

<table>
<thead>
<tr>
<th>Operation</th>
<th>Diagram</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rz(q1, +angle/2); // C</td>
<td>![CNOT Circuit]</td>
</tr>
<tr>
<td>CNOT(q0, q1);</td>
<td>![CNOT Circuit]</td>
</tr>
<tr>
<td>Rz(q1, -angle/2); // B</td>
<td>![CNOT Circuit]</td>
</tr>
<tr>
<td>CNOT(q0, q1);</td>
<td>![CNOT Circuit]</td>
</tr>
<tr>
<td>Rz(q0, +angle/2); // D</td>
<td>![CNOT Circuit]</td>
</tr>
<tr>
<td>CNOT(q0, q1);</td>
<td>![CNOT Circuit]</td>
</tr>
</tbody>
</table>

**Correct, operation A unneeded**

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

**Correct, operation C unneeded**

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

**Incorrect, angles flipped**

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

\[
\begin{align*}
\text{Rz}(q_1, +\text{angle}/2); & \quad \text{// C} \\
\text{CNOT}(q_0, q_1); \\
\text{Rz}(q_1, -\text{angle}/2); & \quad \text{// B} \\
\text{CNOT}(q_0, q_1); \\
\text{Rz}(q_0, +\text{angle}/2); & \quad \text{// D} \\
\end{align*}
\]

\[
\begin{align*}
\text{Correct, operation A unneeded} \\
|11\rangle \rightarrow e^{i\cdot\text{angle}} |11\rangle
\end{align*}
\]

\[
\begin{align*}
\text{Correct, operation C unneeded} \\
|11\rangle \rightarrow e^{i\cdot\text{angle}} |11\rangle
\end{align*}
\]

\[
\begin{align*}
\text{Incorrect, angles flipped} \\
|11\rangle \rightarrow e^{-i\cdot\text{angle}} |11\rangle
\end{align*}
\]
Even simple quantum programming bugs lead to non-obvious symptoms

\[
\begin{align*}
|11\rangle & \rightarrow e^{i\cdot\text{angle}}|11\rangle \\
|11\rangle & \rightarrow e^{i\cdot\text{angle}}|11\rangle \\
|11\rangle & \rightarrow e^{-i\cdot\text{angle}}|11\rangle
\end{align*}
\]
• “Exception handling” (i.e., quantum error correction) is costly.
• No printf. No intermediate measurements.
• Can’t set arbitrary breakpoints.
• Few obvious assertions.
I. Correct quantum programs
   A. Verification *(proofs)*
   B. Validation (debugging & assertions)
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
II. Correct quantum programs
   A. Verification (proofs)
   B. Validation (debugging & assertions)
We talk about three papers on these approaches.

**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
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.
Toolchain for debugging programs with tests on measurements

1. **Breakpoint annotations**
   - Quantum assembly code
   - Simulate / execute on prototype quantum computer
2. Ensemble of measurements
3. **Statistical test:** Is there a bug?

**Assertion annotations**
Detailed debugging of Shor’s factorization algorithm

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

Quantum part of algorithm:
- Control Register ($N$ qubits)
- Target Register ($N$ qubits)
- $|0\rangle \otimes N$
- $|1\rangle$
- $QFT$
- Inverse $QFT$
- Controlled Function

Classical post processing:
- Measurement (Classical result)
- Classical Continued Fractions Algorithm
- Result: Order of $f(x)$ with high prob.

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
Bug type 3-B: mistake in composing gates using mirroring

Mirror image submodules

Control Register (N qubits)
|0⟩⊗N

QFT

Target Register (N qubits)
|1⟩

Inverse QFT

Controlled Function

Measurement (Classical result)

Classical Continued Fractions Algorithm

Result: Order of f(x) with high prob.
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
Bug type 3-B: mistake in composing gates using mirroring
Toolchain for debugging programs with tests on measurements

- **Breakpoint annotations**
- Quantum assembly code
- **Assertion annotations**

Simulate / execute on prototype quantum computer → Ensemble of measurements → **Statistical test: Is there a bug?**
Quantum programming and correctness

I. Correct quantum programs
   A. Verification (proofs)
   B. Validation (debugging & assertions)

II. Quantum programming: from abstractions to execution
   A. Creation / discovery of algorithms
   B. Specification of algorithm as a procedure
   C. Compilation to native instructions while maximizing performance
   D. Execution in target machine with facilities for validation
II. Quantum programming: from abstractions to execution
   A. Creation / discovery of algorithms
   B. Specification of algorithm as a procedure
   C. Compilation to native instructions while maximizing performance
   D. Execution in target machine with facilities for validation

Provide abstractions for easy programming

Maximize performance and ensure correctness in hardware
II. Quantum programming: from abstractions to execution

A. *Creation / discovery of algorithms*
B. Specification of algorithm as a procedure
C. Compilation to native instructions while maximizing performance
D. Execution in target machine with facilities for validation
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.
Programming languages aid discovery of algorithms: Is it currently true for quantum computer science?

Only ~100 known quantum algorithms
I. Quantum programming: from abstractions to execution
   A. Creation / discovery of algorithms
   B. **Specification of algorithm as a procedure**
   C. Compilation to native instructions while maximizing performance
   D. Execution in target machine with facilities for validation
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
Specification of quantum algorithm as a quantum procedure

Quantum random walk from last week. Idea to proof to equation to gates. 

Need quantum equivalent of:
1. Coin toss
2. Position time series
3. Position increment decrement

Gates to code that we can run. 
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

Given the implementation for 1D quantum random walk, is it easier now to create 2D quantum random walk?
Programming languages aid discovery of algorithms: Is it currently true for quantum computer science?

- Quantum algorithms
- Quantum coding
- Quantum computer science
- Quantum programs

Only ~100 known quantum algorithms

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

Given the implementation for 1D quantum random walk, is it easier now to create 2D quantum random walk?
II. Quantum programming: from abstractions to execution
   A. Creation / discovery of algorithms
   B. Specification of algorithm as a procedure
   C. *Compilation to native instructions while maximizing performance*
   D. Execution in target machine with facilities for validation
Programmers’ time is scarce. Classical computing resources are abundant. Nonetheless, abstractions are expensive.

<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>11</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>47</td>
<td>4.4</td>
<td>0.03</td>
</tr>
<tr>
<td>4</td>
<td>Parallel loops</td>
<td>69.80</td>
<td>1,969</td>
<td>366</td>
<td>7.8</td>
<td>0.24</td>
</tr>
<tr>
<td>5</td>
<td>Parallel divide and conquer</td>
<td>3.80</td>
<td>36.180</td>
<td>6,727</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>23,224</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>62,806</td>
<td>2.7</td>
<td>40.45</td>
</tr>
</tbody>
</table>

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 two-week chapter on “extracting success.”
II. Quantum programming: from abstractions to execution
   A. Creation / discovery of algorithms
   B. Specification of algorithm as a procedure
   C. Compilation to native instructions while maximizing performance
   D. Execution in target machine with facilities for validation
Execution in target machine w/ facilities for validation

- Exception handling.
- Printf debugging.
- GDB breakpoints.
- Assertions.
- Etc.