SQASM: Simple Quantum Assembly

Ryan Watkins

May 24, 2016

Table of contents

Introduction
- Why SQASM?
- Quantum Mechanics

Quantum Simulator
- Overview
- Analysis

Quantum Programming Language
- Syntax

Fields

- Brings together many fields
- Physics
- Mathematics
- Computer Science

Principles

- Uncertainty Principle
- Bell's Inequality
- No Cloning Theorem

Uncertainty Principle

- 'a phenomenon which is impossible to explain in any classical way, and which has in it the heart of quantum mechanics'
- Phenomena demonstrated by double-slit experiment, see Figure
- Performed as early as 1801 by Thomas Young before knowledge of quantum mechanics
- Formalised by Werner Heisenberg in 1927

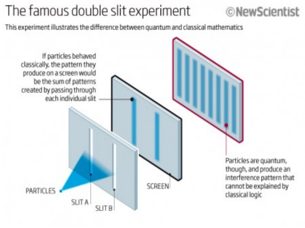

Figure: Famous double-slit experiment

Einstein, Podolsky, Rosen (1935)

- Let S_1 be a 2 qubit state: $\frac{1}{\sqrt{2}}\{\uparrow\uparrow + \downarrow\downarrow\}$
- up/down electron spin state notation, $\frac{1}{\sqrt{2}}$ is $\frac{1}{2}$ after normalized
- Implies correlation, which upset EPR, which leaves the options:
- That it was always up and up or down and down
- Or, there exists a deep non-locality in the universe
- One could say QM is insufficient and there exists some hidden variable in CM
- The second version is the QM version, that it is just how it works which has been empirically verified

Bell's Inequality

- Show's that quantum mechanics does not have a hidden classical property
- $S = \{A, B, C\}$
- $N(A, \overline{B} + N(B, \overline{C}) \geq N(A, \overline{C})$
- $N(A, \overline{B}, C) + N(\overline{A}, B, \overline{C}) \geq 0$
- Because, any set of elements is always greater than zero

Quantum Simulator

- Obeys laws of Quantum Mechanics
- Applies a QRAM Quantum Architectural
- Quantum Arithmetic
- Carry-Save Adder
- Low quantum cost multiplier
- Deutsch, Jozsa (1992) algorithm implementation
- Interface to Quantum Programming Language
- Written from scratch in Python, found at github.com/watkinsr/SQASM
- Highly extensible, can run Shor's algorithm

Overview

- Ensure QRAM architecture
- Pairwise communication between classical and quantum machine
- Classical machine specifies computation
- Quantum machine does it

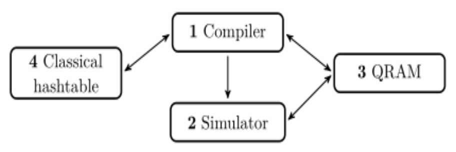

Figure: Overview of solution

Further analysis

- The Quantum Simulator (Theoretical Quantum Machine) is a blackbox that is reliant upon the classical machine for input
- Classical compiled code is input to the quantum simulator. Quantum computation is done and data is passed back
- The data is Python objects representing registers or gates
- This data gets stored in a hashtable for later use by the compiler

Quantum System initialization

```
class QReg:
        def __init__(self, n_qubits, setVal=-1):
            self.n_qubits = n_qubits
            self.qubits = [0] * n_qubits
            self.amps = [0] * (1 << n_qubits) # 2^n_qubits complex numbers
            self.amps[len(self.amps) - 1] = 1
            if (setVal != -1):
                self.amps[setVal] = 1
                if (setVal != len(self.amps) - 1):
                    self.amps[len(self.amps) - 1] = 0
            self.amps = np.matrix(self.amps).T
```

- Amplitudes are the probabilities of our quantum states, represented in Binary format
- setVal initialises a quantum register to a given state
- Amplitudes $= 2^n$, where $n =$ quantum bit size

Quantum Bits

- Can only be measured or observed
- The act of measuring causes a collapse, we return to discrete values of $\{0, 1\}$
- If $\{110\}$ or amps[6] $= 1$, then: $\{q_1, q_2\} = 1, \{q_3\} = 0$
- We can also say that these states are definite
- Superposition
- Given 2^3 amplitudes in superposition, each state $= \frac{1}{\sqrt{8}}$
- Next slide shows this in practice

Quantum Bits in practice

```
q = QReg(3, 5)  # 3, num qubits, 5 specifies index to set 1
print('QReg Amplitudes are: %s' % q.amps.T)
```

```
QReg Amplitudes are: [[0 0 0 0 0 1 0 0]]
```

Applying Quantum Gates

```
r = INITIALIZE(4)  # Get quantum system with 3 qubits
qs.applyGate(t(HAD, ID, ID, ID), r)  # Had bit1
qs.applyGate(t(ID, HAD, ID, ID), r)  # Had bit2
qs.applyGate(t(ID, ID, HAD, ID), r) # Had bit3
qs.applyGate(t(ID, ID, ID, HAD), r) # Had bit4
print(r.amps.T)
```

```
[[ 0.25+0.j  -0.25+0.j  -0.25+0.j   0.25+0.j  -0.25+0.j   0.25+0.j   0.25+0.j
 -0.25+0.j  -0.25+0.j   0.25+0.j   0.25+0.j  -0.25+0.j   0.25+0.j  -0.25+0.j
 -0.25+0.j   0.25+0.j]]
```

Specifying Quantum Gates

- Hadamard Gate in Python and mathematical representation

```
HAD = np.matrix([[1 / sqrt(2), 1 / sqrt(2)],
[1 / sqrt(2), -1 / sqrt(2)]])
```

$$\begin{bmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \end{bmatrix}$$

Class Breakdown

Class	Description
QReg	Initialize quantum registers with amplitudes and a set definite state. One can also obtain the current state of the qubit with the *getState ()* function.
QSimulator	Measurement of quantum registers and selection of specific qubits from a quantum register. Application of quantum gates to registers. NAND gate implementation. Also included are quantum gate matrices such as the MTSG and Peres gate
QAdder	Quantum Majority Gate (QMG) and Quantum Full-Adder (QFA) split into two different functions for code reusability. The adder class also permits subtraction by using *Two's Complement*
QMultiplier	Contains a complete implementation of a quantum cost efficient multiplier circuit taken from a research paper

Figure: Class breakdown for Quantum Simulator

First Quantum Algorithm

- Deutsch-Jozsa(1992) algorithm takes one evaluation time step as opposed to $2^n/2+1$ evaluations necessary in a classical machine
- Somewhat arbitrary algorithm contrived to show power of quantum computation
- $\{0,1\} \rightarrow \{0,1\}$
- $f(0) = f(1)$?
- Classically requires two operations, calculate $f(0)$ and $f(1)$ and compare

Step one

- In: $\Psi = |0\rangle\,|1\rangle$
- Hadamard both Qubits
- $\Psi = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)\frac{1}{\sqrt{2}}(|0\rangle - |1\rangle)$
- $= \frac{1}{2}(|0\rangle\,(|0\rangle - |1\rangle) + |1\rangle\,(|0\rangle - |1\rangle))$
- Note: we simplify to apply U_f

Step two

- After applying U_f:
- $\frac{1}{2}[|0\rangle(|f(0)\oplus 0\rangle - |f(0)\oplus 1\rangle) + |1\rangle(|f(1)\oplus 0\rangle - |f(1)\oplus 1\rangle)]$
- Note: $f(0) = 0 \implies (0-1)$
- $f(0) = 1 \implies (1-0) = -(0-1)$
- $= (-1)^{f(0)}(|0\rangle - |1\rangle)$
- $= \frac{1}{2}[(-1)^{f(0)}|0\rangle(|0\rangle - |1\rangle) + (-1)^{f(1)}|1\rangle(|0\rangle - |1\rangle)]$
- $= \frac{1}{2}(-1)^{f(0)}[|0\rangle + (-1)^{f(0)\oplus f(1)}|1\rangle](|0\rangle - |1\rangle)$

Step three

- Forget about second qubit
- Same $\implies |0\rangle + |1\rangle$
- Different $\implies |0\rangle - |1\rangle$
- These are familiar states: they are obtained via Hadamard
- Therefore, do the inverse of Hadamard, which is Hadamard itself

Step four

- Hadamard first qubit
- $\Psi_{out} = \frac{1}{2}(1 + (-1)^{f(0) \oplus f(1)}) |0\rangle + \frac{1}{2}(1 - (-1)^{f(0) \oplus f(1)}) |1\rangle$
- Measure first qubit,
- If 0, then same
- Else, not

Deutsch, Josza Implementation

```
r = QReg (4, 0)  # Initialise system w/ 4 qubits
qs.applyGate (t (HAD, ID, ID, ID), r)  # Had 1st qubit
qs.applyGate (t (ID, HAD, ID, ID), r)  # Had 2nd qubit
qs.applyGate (t (ID, ID, HAD, ID), r)  # Had 3rd qubit
qs.applyGate (t (ID, ID, ID, HAD), r)  # Had 4th qubit
qs.quantumOracle (function,r)
qs.applyGate (t (HAD, ID, ID, ID), r)  # Had 1st qubit
qs.applyGate (t (ID, HAD, ID, ID), r)  # Had 2nd qubit
qs.applyGate (t (ID, ID, HAD, ID), r)  # Had 3rd qubit
qs.applyGate (t (ID, ID, ID, HAD), r)  # Had 4th qubit
for qubit in range (4):
     functionChanges |= (qs.measure (r,qubit)==1)

     if functionChanges:
         print ('Function is balanced')
     else:
         print ('Function is constant')
```

SQASM Overview

Operation	Description
INITIALIZE $[r, n, pos]$	Initializes a quantum register of n qubit size with definite configuration
$[v_1]$ TENSOR $[g_1, g_2]$	Applies tensor product to unitary matrices
APPLY $[g, r]$	Applies matrix multiplication between quantum state column vector and unitary quantum gate
SELECT $[v, r, n_1, n_2]$	Selects quantum bits from a range inside a quantum register
MEASURE $[r, v]$	Measures the state of a given qubit or register
ADD $[v_1, v_2, r]$	Performs addition or subtraction between constants or variables
PEEK $[r]$	Allows one to peek into a given registers amplitudes for testing purposes
HAD, ID, CNOT, ...	Shorthand references to constant quantum gates Hadamard, Identity and Controlled-NOT respectively

Where r = register, n = number, g = gate and v = variable

Figure: SQASM Syntax Table