QuantC: A CUDA-Inspired Language for Quantum Computing
**Crash Course in Quantum Computing: Qubits**

- *Qubits* are like bits, but are based on some quantum phenomenon with a binary set of states when measured.

- Can store 0 and 1 like a regular bit, but can also be in a *superposition* of 0 and 1.

- Measuring a qubit causes it to *collapse* into either a 0 or 1 if it’s in a superposition.
Crash Course in Quantum Computing: Gates and Measuring

• A common representation of computations on qubits is as a circuit, with operations represented as quantum logic gates.

• Quantum circuits are just as computationally powerful as classical circuits.

• However, quantum circuits do have some constraints classical versions don’t have.
  • Gates/circuits must be reversible; at minimum, the # outputs = # inputs.
  • Gates can’t be used to copy or delete arbitrary quantum states.
Crash Course in Quantum Mechanics

• Value of a qubit is represented by a normalized vector in $\mathbb{C}^2$
  • First coefficient is always taken to be real
• Gates are unitary matrices in $\mathbb{C}^{2^n \times 2^n}$
  • Unitary: $UU^\dagger = U^\dagger U = I$
• Measurement corresponds to randomly picking a base state, where the probability of picking it is based on the coefficient in the qubit vector

$$|\psi\rangle = \sin\left(\frac{\theta}{2}\right)|0\rangle + \cos\left(\frac{\theta}{2}\right)e^{i\varphi}|1\rangle$$

$$|0\rangle = \begin{bmatrix} 1 \\ 0 \end{bmatrix}, |1\rangle = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$$
Examples of Quantum Gates

Classically Equivalent Gates

- **NOT** (a.k.a X-gate)

- **XOR** (a.k.a Controlled NOT, CNOT)

- **AND** (a.k.a Toffoli gate, CCNOT)

Purely Quantum Gates

- **Hadamard gate**

- **$\sqrt{NOT}$ gate**
Simple Circuit Examples

Entanglement Circuit

\[ |00\rangle \rightarrow \frac{|00\rangle + |11\rangle}{\sqrt{2}} \]

Full Adder
Design Goals

- Create a language with syntax to support creating programs with both classical and quantum components
- Minimize the overhead for learning the language
  - Extend an existing language, rather than start from scratch
  - Make distinguishing the two parts of code clear
  - Only introduce new syntax for new concepts
  - Avoid overloading the meaning of existing symbols for quantum code unless the semantics are closely related
- Abstract communication between the classical and quantum computers
**Standard C Code**

```c
void saxpy(int n, float a,
          float *x, float *y)
{
    for (int i = 0; i < n; ++i)
        y[i] = a*x[i] + y[i];
}
int N = 1<<20;

// Perform SAXPY on 1M elements
saxpy(N, 2.0, x, y);
```

**C with CUDA extensions**

```c
__global__
void saxpy(int n, float a,
          float *x, float *y)
{
    int i = blockIdx.x*blockDim.x + threadIdx.x;
    if (i < n) y[i] = a*x[i] + y[i];
}
int N = 1<<20;
cudaMemcpy(x, d_x, N, cudaMemcpyHostToDevice);
cudaMemcpy(y, d_y, N, cudaMemcpyHostToDevice);

// Perform SAXPY on 1M elements
saxpy<<<4096,256>>>(N, 2.0, x, y);
cudaMemcpy(d_y, y, N, cudaMemcpyDeviceToHost);
```
Storage Classifier: quantum

- Keyword on functions to distinguish between classical code and quantum code
- Inside quantum functions, variables can also be marked as quantum to differentiate between qubits and bits

```cpp
quantum int hello_quantum() {
    int a, b;
    quantum int q0, q1;
}
```
Calling Quantum Functions

- When calling a quantum function in classical code, the compiler substitutes it with a call to a function to transmit the compiled quantum code to the quantum computer
  - The quantum *intermediate representation* (IR) is stored as a string created by the compiler
  - It’s assumed that any additional translation from the quantum IR to quantum machine instructions is handled by the quantum computer itself
- Constraints on quantum functions
  - Can have quantum arguments, but they must be pass by reference (pointer)
  - Only functions with exclusively classical arguments passed by value can be called from classical code
  - They can return values, but only classical values
  - Recursive calls are not allowed
Semantics of Quantum Variables

• With qubits, what assignment means becomes trickier to define
  • Between two quantum variables, assignment has *move semantics* rather than *copy semantics*
  • The variables being assigned to must be uninitialized, and the variables being used in the assignment expression can’t be used again
  • When working with expressions with multiple variables, need to establish a 1-to-1 mapping of names across assignment
  • This is all checked by the compiler at compile time

```java
quantum int hello_quantum() {
    int a, b;
    quantum int q0, q1, q2, q3;

    ...

    a = b;  // Allowed; classical assignment acts like normal
    q1 = a;  // Allowed; classical to quantum assignment simply sets qubits to corresponding base states.
    a = q1;  // Denied; assignment from quantum to classical memory
             // needs to have the qubit measured first.
    a = %M q1;  // Allowed; Measurement operator
               // collapses `q1` state and stores result in `a`
    q2 = q1;  // Allowed, but `q1` is no longer usable
    auto [q2, q3] = q0 ⊕ q1;  // Allowed, but only if q2, q3 are uninitialized
    ...
```
Quantum Operations (Classical)

```latex
// Classical gates
\sim q1; \quad \text{// NOT (also called X gate)};
q1 \^ q2; \quad \text{// Implicitly CNOT; q2 becomes q1 XOR q2}
q1 \& q2 : \text{result}; \quad \text{// Implicitly Toffoli (CCNOT)}
\quad \quad \text{// result becomes result XOR q1 AND q2;}
q1 \mid q2 : \text{result}; \quad \text{// Implicitly Toffoli + NOT gates}
\quad \quad \text{// result becomes result XOR q1 OR q2;}
```
Quantum Operations (Unary)

```csharp
// Identity gates (does not change qubit)
%I q1; // Identity gates (does not change qubit)

// Pauli Y gate
%Y q1; // Pauli Y gate

// Pauli Z gate
%Z q1; // Pauli Z gate

// Hadamard
%H q1; // Hadamard

// Phase gates
// Parameters of gates must be a float
%PHASE(angle) q1; // General phase gates, parametrized by an angle
%S q1; // phase gate with angle = \pi/2
%T q1; // phase gate with angle = \pi/4

// Rotation gates
// Parameters of gates must be a float
%RX(angle) q1;
%RY(angle) q1;
%RZ(angle) q1;
```
Quantum Operations (Binary)

// Swap gates
// Parameters of gates must be a float
q1 <P>(angle) q2; // General swap; parametrized by an angle
q1 ◯ q2; // Classical swap; angle = 0
q1 <I> q2; // Imaginary swap; angle = pi/2
Quantum Operations (Modifiers)

```c
// Controlled gates
// You can add control qubits to any gate by postfixing with +c(qubit)
%H+c(q2) q1;

// Reverse gate
// Prefixing a gate with ~ will apply the inverse of that gate
~>%H (%H q2); // same as %I q2;
```
Implementation

Preprocessing: GCC

Lexer & Parser: Flex & Bison

AST & Semantics
Checking: Hand-coded

Code generation:
LLVM (Classical)
Quil (Quantum)

Linking: GCC

GCC

C++

Quil [01]
Progress

- Established compilation of minimal amount of classical/quantum code
  - Basically, can compile main() and make a call to a quantum function
  - Can compile down to Linux object files/executables
  - Can link with standard C library code
- Created a basic version of code that can communicate quantum code with a local simulator (QVM) over HTTP

**Next Phase**
- Complete quantum operation code generation
- Handle assignment of quantum variables
- Handle generate code for multiple quantum functions in the same translation unit
- Add all/most classical constructs (if/else, for loop, etc.) to both kinds of code