Quantum-Classical Compilation with the MLIR

Alex McCaskey

with Thien Nguyen, Eugene Dumitrescu, Dmitry Liakh, Anthony Santana

Oak Ridge National Laboratory

Computer Science and Mathematics

MLIR Open Design Meeting

27 May 2021

Quantum Computer Science at ORNL

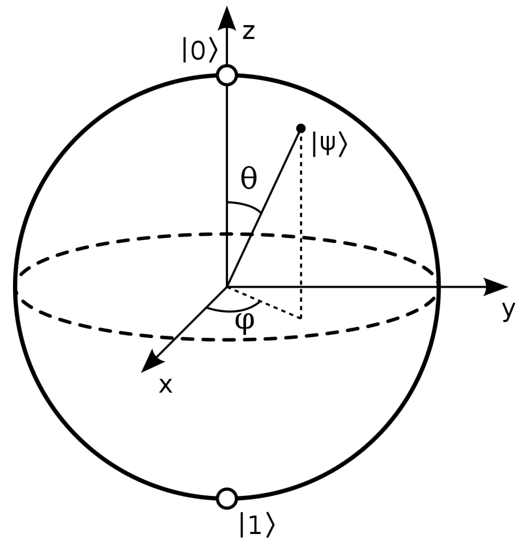
- DOE Open Science Laboratory
- Nuclear Physics, Material Science, and High Performance Computing
- Perennially houses fastest(open) supercomputers in the world
- Gearing up for Frontier
- What does a post-exascale computing architecture look like?

Outline of Today's Talk

- **Background** - quantum programming in the DOE scientific computing context
  - What's a qubit? How do we program QPUs?
  - Heterogeneous quantum-classical computing
- **Quantum programming**
  - `qcor` – Start leveraging classical compilation frameworks for quantum-classical computing
  - Move away from high-level Pythonic toolkits
  - Clang, LLVM, and MLIR: what can we borrow for quantum?
- **MLIR for Quantum** – rapid compiler prototyping via progressive lowering to the QIR
  - Built into `qcor`, Dialect for quantum languages
  - Lowering to LLVM Dialect
  - Opportunities for classical and quantum optimizations

Brief Quantum Computing 101

- Key Concepts
  - Unit of data is the quantum bit (2-level quantum mechanical system)
  - Qubit state can be |0>, |1> or a superposition
  - Qubits can be entangled
  - Quantum Instructions are unitary matrices applied to the qubit state (which can be represented as a vector in a 2^N dimensional Hilbert space)
  - Information encoded in basis state amplitudes
  - Measurement collapses to a classical state
- How do we visualize quantum programs
  - Circuit diagrams, quantum assembly language
  - Ultimately gates translate to analog pulses in hardware

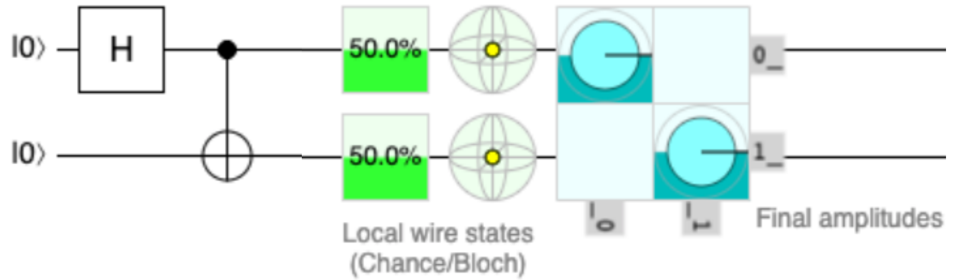


General Qubit State

$$|\psi\rangle = \cos(\theta/2)|0\rangle + e^{i\phi}\sin(\theta/2)|1\rangle$$

Probability to measure qubit in basis state:

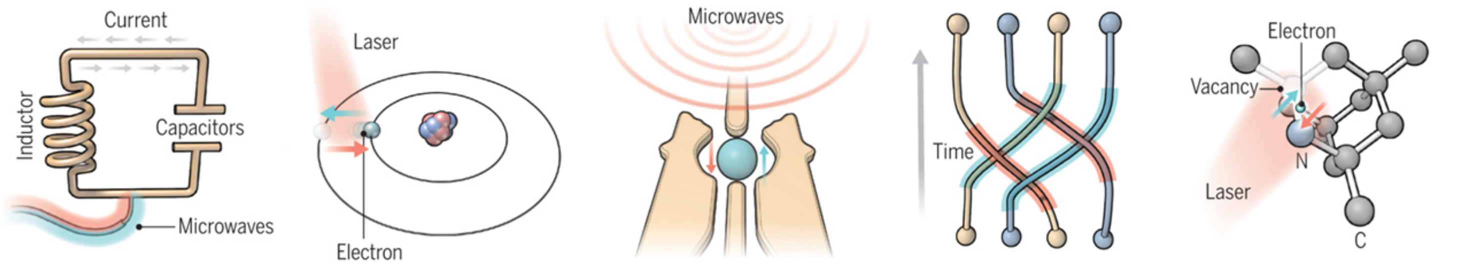
$P(0) = \cos^2(\boldsymbol{\theta}/2)$
$P(1) = \sin^2(\boldsymbol{\theta}/2)$



A Race for Quantum Technology

A bit of the action

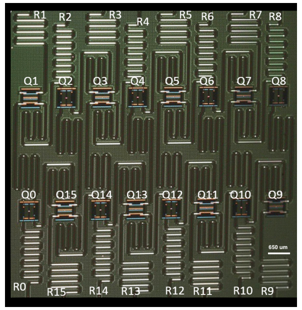
In the race to build a quantum computer, companies are pursuing many types of quantum bits, or qubits, each with its own strengths and weaknesses.



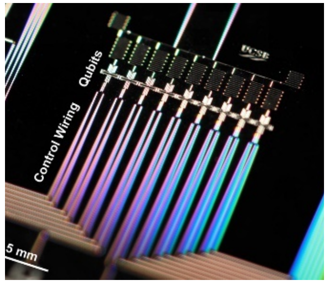
	Superconducting loops	Trapped ions	Silicon quantum dots	Topological qubits	Diamond vacancies
	A resistance-free current oscillates back and forth around a circuit loop. An injected microwave signal excites the current into super-position states.	Electrically charged atoms, or ions, have quantum energies that depend on the location of electrons. Tuned lasers cool and trap the ions, and put them in superposition states.	These "artificial atoms" are made by adding an electron to a small piece of pure silicon. Microwaves control the electron's quantum state.	Quasiparticles can be seen in the behavior of electrons channeled through semi-conductor structures. Their braided paths can encode quantum information.	A nitrogen atom and a vacancy add an electron to a diamond lattice. Its quantum spin state, along with those of nearby carbon nuclei, can be controlled with light.
Longevity (seconds)	0.00005	>1000	0.03	N/A	10
Logic success rate	99.4%	99.9%	~99%	N/A	99.2%
Number entangled	~70	22	2	N/A	6
Company support	Google, IBM, Quantum Circuits	ionQ	Intel	Microsoft, Bell Labs	Quantum Diamond Technologies
Pros	Fast working. Build on existing semiconductor industry.	Very stable. Highest achieved gate fidelities.	Stable. Build on existing semiconductor industry.	Greatly reduce errors.	Can operate at room temperature.
Cons	Collapse easily and must be kept cold.	Slow operation. Many lasers are needed.	Only a few entangled. Must be kept cold.	Existence not yet confirmed.	Difficult to entangle.

Note: Longevity is the record coherence time for a single qubit superposition state, logic success rate is the highest reported gate fidelity for logic operations on two qubits, and number entangled is the maximum number of qubits entangled and capable of performing two-qubit operations.

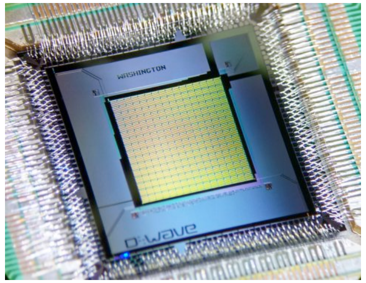
updated

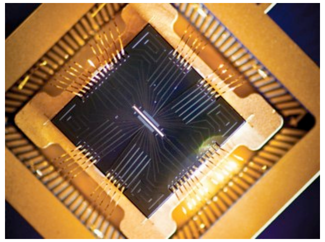

Superconducting chip, IBM

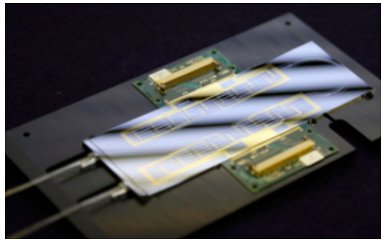

Superconducting chip, Rigetti



Superconducting chip, Google


Superconducting chip, D-Wave Systems


Ion trap chip, Sandia

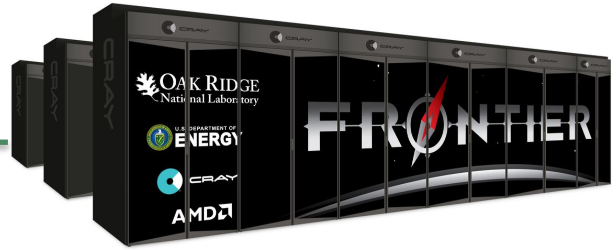

Linear optical chip, Univ. Bristol/QET Labs

DOE Scientific (Quantum) Computing

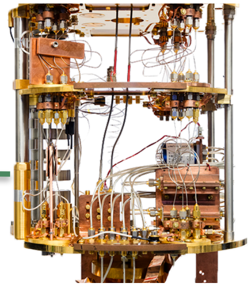
⟨quantum|gov⟩

- DOE interested in QC for scientific computing
  - Near-term -> Exascale
  - Future -> Leverage quantum computer as we do other accelerators
- Focused on
  - Algorithm development
  - Core computer science questions
  - Hardware development
- Requirements
  - System-level (HPC)
  - Hardware agnostic

{ **Quantum computers**, while *not a substitute* for classical computers, are believed to be extraordinarily powerful at solving certain problems…
DOE Office of Science }



Heterogeneous computing
CPU-QPU models
System-level SW infrastructure



Quantum Languages and Programming

Issues *– fragmentation, future performance, integration with classical compiler frameworks*

Languages *– assembly, pythonic eDSLs, hybrid languages, language extensions*

Programming Quantum Computers Today

- Pythonic programming model
  - Rapid prototyping, experimentation
  - Low learning curve
  - REST client support
- Intermediate languages
  - OpenQASM, QUIL, XASM, etc.
- Fragmentation
  - A lot of code re-writing
  - Differing feature sets, library implementations
- Future performance concerns
- Need to move toward quantum-classical languages
  - Q#, qcor, native languages
  - Tighter CPU-QPU integration model

Qiskit
Elements for building a quantum future

XACC
QUANTUM FRAMEWORK

Cirq

```
from qiskit import QuantumCircuit
from qiskit.aqua.operators import EvolvedOp, PauliTrotterEvolution
def trotter_evolve(q, exp_args, n_steps):
  qc = QuantumCircuit(q)
  for i in range(n_steps):
    for sub_op in exp_args:
      qc += PauliTrotterEvolution().convert(EvolvedOp(sub_op)).to_circuit()
  return qc
```

N	Gate Count	QCOR [secs]	Qiskit [secs]
5	2700	0.080705	4.776733875274658
10	5700	0.174581	19.232121229171753
20	11700	0.372422	93.73427820205688
50	29700	1.09762	916.3527870178223
100	59700	2.42994	- - -

(Python Generated) Quantum Assembly Languages

qiskit (IBM)

```
import qiskit
from qiskit import IBMQ
from qiskit.providers.aer import AerSimulator

# Generate 3-qubit GHZ state
circ = qiskit.QuantumCircuit(3)
circ.h(0)
circ.cx(0, 1)
circ.measure_all()

# Construct an ideal simulator
aersim = AerSimulator()

# Perform an ideal simulation
result_ideal = qiskit.execute(circ, aersim).result()
counts_ideal = result_ideal.get_counts(0)
print('Counts(ideal):', counts_ideal)
```

⇩ output qasm 2.0 for REST JSON submission API

```
OPENQASM 2.0;
include "qelib1.inc";
qreg q[2];
creg c[2];
h q[0];
cx q[0],q[1];
measure q[0] -> c[0];
measure q[1] -> c[1];
```

pyquil (Rigetti)

```
from pyquil import get_qc, Program
from pyquil.gates import CNOT, H, MEASURE

qvm = get_qc('2q-qvm')

p = Program()
p += H(0)
p += CNOT(0, 1)
ro = p.declare('ro', 'BIT', 2)
p += MEASURE(0, ro[0])
p += MEASURE(1, ro[1])
p.wrap_in_numshots_loop(10)

qvm.run(p).tolist()
```

⇩ output quil for REST/ZMQ submission API

```
DECLARE ro BIT
H 0

CNOT 0 1
MEASURE 0

MEASURE 1 ro[0]
```

- Intermediate Languages defined by vendors
- Simple with limited control flow
- Processed in Python, `requests.post()`
- QASM 3 step in right direction (still py-gen)

```
def iqft qubit[n_counting]:qq {
    for i in [0:n_counting/2] {
        swap qq[i], qq[n_counting-i-1];
    }
    for i in [0:n_counting-1] {
        h qq[i];
        int j = i + 1;
        int y = i;
        while (y >= 0) {
            double theta = -pi / (2^(j-y));
            cphase(theta) qq[j], qq[y];
            y -= 1;
        }
    }
    h qq[n_counting-1];
}
```

(dynamic) Compiled Languages – Q#, Silq, and `qcor`

```
@EntryPoint()
operation MeasureOneQubit() : Result {
    // The following using block creates a fresh qubit and initializes it
    // in the |0〉 state.
    use qubit = Qubit();
    // We apply a Hadamard operation H to the state, thereby preparing the
    // state 1 / sqrt(2) (|0〉 + |1〉 ).
    H(qubit);
    // Now we measure the qubit in Z-basis.
    let result = M(qubit);
    // As the qubit is now in an eigenstate of the measurement operator,
    // we reset the qubit before releasing it.
    if result == One { X(qubit); }
    // Finally, we return the result of the measurement.
    return result;

}
```

- (dynamic) languages – feedback, sequential execution (no batch submission)
- We see languages like this as moving in the right direction.

```
def solve(k:!ℕ){
  // produce uniform superposition over k-bit uints
  i:=0:uint[k];
  for j in [0..k){ i[j]:=H(i[j]); }
  // invert i-th qubits (results in correct state, but entangled with i)
  qs:=vector(2^k,0:𝔹);
  qs[i]=X(qs[i]);
  // uncompute i
  forget(i=λ(qs:𝔹^(2^k))lifted{ // function to reconstruct i from qs
    i:=0:uint[k];
    for j in [0..2^k){
      if qs[j]{ // in the superposition's summand where qs[j]==1, i==j
        i=j as uint[k];
      }
    }
    return i;
  }(qs));
  // return result
  return qs;
}

// EXAMPLE CALL

def main(){
  // example usage for k=2
  return solve(2);
}
```

`qcor` – What is it and what are its design goals?

`qcor` is a language extension and associated compiler platform for heterogeneous quantum-classical computing in C++ and Python. – http://docs.aide-qc.org

QCOR

QUANTUM COMPILER

- Native Quantum Language Compiler
- Extensibility and Modularity
- Integration Platform
- Retargetable
- Extend C++ and Python
- Multi-modal (remote, local)
- Automated Placement
- Optimizing Compiler

qcor Quantum Kernel Expression

- Programming model: units of quantum execution decomposed into standalone functions – *quantum kernels*
  - Language extensibility, common patterns, unitary decomposition, kernel modifiers, functional programming, standard library development. C++ and Python supported.

Express Common Patterns

```
__qpu__ void amplification(qreg q) {
  // H q X q ctrl-ctrl-...-ctrl-Z H q Xq
  // compute - action - uncompute
  compute {
    H(q);
    X(q);
  }
  action {
    auto ctrl_bits = q.head(q.size() - 1);
    auto last_qubit = q.tail();
    Z::ctrl(ctrl_bits, last_qubit);
  }
}
```

Language Extensibility: Circuit Synthesis

```
@qjit
def ccnot(q : qreg):
    # create 111
    X(q)

    # decompose with qfactor
    with decompose(q, qfactor) as ccnot:
        ccnot = np.eye(8)
        ccnot[6,6] = 0.0
        ccnot[7,7] = 0.0
        ccnot[6,7] = 1.0
        ccnot[7,6] = 1.0

    # CCNOT should produce 110 (lsb)
    Measure(q)
```

Lambdas and Callables

```
using GroverOracle = KernelSignature<qreg>;
__qpu__ void run_grover(qreg q,
                 GroverOracle oracle) {
   ...
   oracle(q);
   ...
}
__qpu__ mark_states(qreg q) {
   ...
}
...
run_grover(q, mark_states);
auto mark_lambda
    = qpu_lambda([](qreg q) {...});
run_grover(q, mark_lambda);
```

Library Development

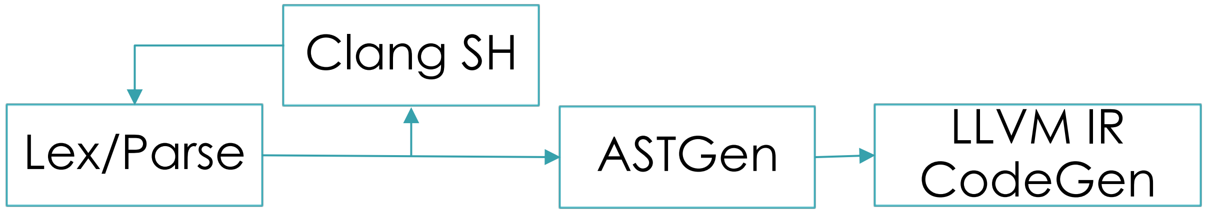
```
#include <qcor_qft>
#include <qcor_hadamard_test>
...
qft(q);
...

auto expectation =
          qcor::hadamard_test(x_gate,
            x_gate, n_state_qubits);
print("<X> = ", expectation);
...
auto workflow =
    QuaSiMo::getWorkflow("vqe",
            {{"optimizer", optimizer}});
result = workflow->execute(problemModel);
```

Extending Clang for DSL Processing

https://arxiv.org/abs/2010.08439
Finkel, McCaskey, Popoola, Liakh, Doerfert

- Goal: Leverage the Clang Plugin system
- `clang::SyntaxHandler`
  - Map invalid function bodies to valid ones
  - Annotate function with handler name
  - Custom DSL – to – valid C++ API calls
- Run after lexing, preprocessing, before AST Gen, lexing restarts with new code
  - Can add code after the function too

```
[[clang::syntax(sh_name)]] void foo() {
  ... Embedded DSL here...
  ... SyntaxHandler with name sh_name will
  ... translate this to standard C++ code
}

---------------------------------------------------

class MySyntaxHandler : public SyntaxHandler {
public:
  MySyntaxHandler() : SyntaxHandler("sh_name") {}
  void GetReplacement(Preprocessor &PP, Declarator &D,
                      CachedTokens &Toks,
                      raw_string_ostream &OS) override {
    ... analyze Toks, write new code to OS
  }
  void AddToPredefines(raw_string_ostream &OS) {
        ... add any #includes here
  }
};
```



Simple Example – `printf` some string

```
[[clang::syntax(tokens)]] void foo() {
   This is a test with a "string".
}
```

Processed with…

```
void GetReplacement(Preprocessor &PP, Declarator &D,
                    CachedTokens &Toks,
                    llvm::raw_string_ostream &OS) override {
  OS << "static const char* tokens = \"";
  for (auto &Tok : Toks) {
    OS << " ";
    OS.write_escaped(PP.getSpelling(Tok));
  }
  OS << "\";\n";
  // Rewrite syntax original function.
  OS << getDeclText(PP, D) << "{\n";
  OS << "printf(\"%s\",tokens);\n";
  OS << "}\n";
}
```

Clang SyntaxHandler Applied to `qcor`

```
#define __qpu__ [[clang::syntax(qcor)]]
```

```
__qpu__ void bell(qreg q, int shots) {
   ... quantum DSL here ...
}
```

`qcor` SyntaxHandler rewrites function to QuantumKernel subtype.

QuantumKernel

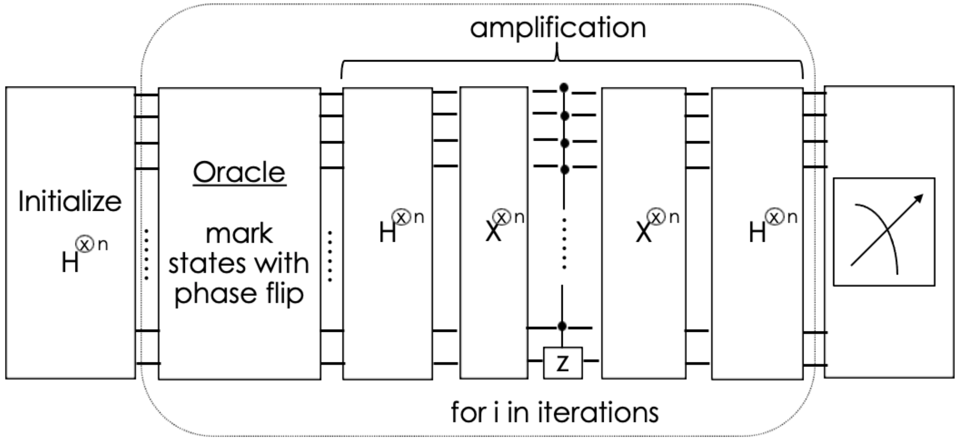
```
void bell(qreg q, int shots) {
   void internal_bell_call(qreg, int);
   internal_bell_call(q, shots);
}

class bell : public QuantumKernel<bell, qreg, int> {
public:
    bell(qreg q, int shots)
      : QuantumKernel<bell,qreg,int>(q, shots) {}
    ~bell() {
      ... execute on backend ....
    }
};

void internal_bell_call(qreg q, int shots) {
   class bell kernel(q, shots);
}
```

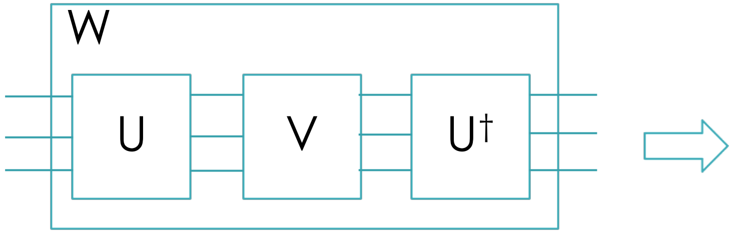
Runtime – libqrt.so

QuantumRuntime

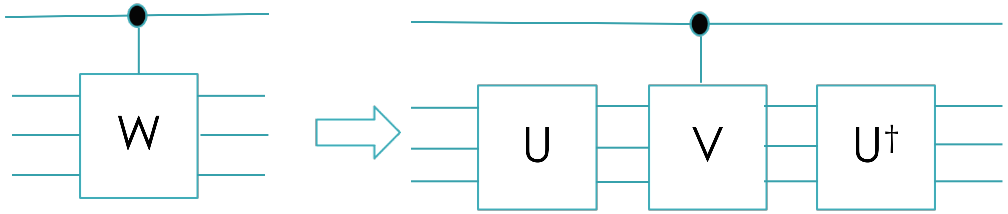
Program call to bell function is a call to another internal function that instantiates a temporary instance of the new QuantumKernel sub-type.

qcor Grover Example






```
compute {
... U ...
} action {
... V ...
}
```



Language extension helps compiler implementation

```
using GroveOracle = KernelSignature<qreg>;

__qpu__ void amplification(qreg q) {
  // H q X q ctrl-ctrl-...-ctrl-Z H q Xq
  // compute - action - uncompute

  compute {
    H(q);
    X(q);
  }
  action {
    auto ctrl_bits = q.head(q.size() - 1);
    auto last_qubit = q.tail();
    Z::ctrl(ctrl_bits, last_qubit);
  }
}

__qpu__ void run_grover(qreg q, GroverOracle oracle,
                        const int iterations) {
  H(q);

  for (int i = 0; i < iterations; i++) {
    oracle(q);
    amplification(q);
  }

  Measure(q);
}
```

```
__qpu__ void oracle(qreg q) {
  // Mark 101 and 011
  CZ(q[0], q[2]);
  CZ(q[1], q[2]);
}

int main() {
  const int N = 3;

  // Allocate some qubits
  auto q = qalloc(N);

  // Call grover given the oracle and n iterations
  run_grover(q, oracle, 1);

  // print the histogram
  q.print();
}
```

Features
- Passing callables
- 1-qubit gates auto ctrl
- Multi-qubit ctrl
- Kernel composition
- Instruction broadcast

```
$ qcor -qpu ibm:ibmq_paris -shots 1024 grover.cpp; ./a.out
```

Leveraging MLIR for Quantum Computing

Goal *– leverage lowering capabilities to take quantum languages to executable code, use QIR (LLVM IR)*

Languages *– provide mapping of languages to MLIR.*

Quantum Optimizations *– leverage pattern rewriting to optimize quantum code.*

Microsoft Quantum Intermediate Representation

- LLVM-based specification for quantum-classical computing
- Qubits and Measurement Results treated as opaque types
- Specification defines LLVM IR functions for
  - qubit register allocation/deallocation
  - qubit addressing
  - array handling
  - quantum instruction invocation
- Implement the specification to target real quantum co-processors
- Provide common representation for optimizations, transformations, and JIT

```
%Array = type opaque
%Result = type opaque
%Qubit = type opaque

declare %Result* @__quantum__qis__mz(%Qubit* %0)
declare void @__quantum__qis__cnot(%Qubit* %0, %Qubit* %1)
declare void @__quantum__qis__h(%Qubit* %0)

declare %Array* @__quantum__rt__qubit_allocate_array(i64 %0)
declare void @__quantum__rt__qubit_release_array(%Array* %0)
declare i8* @__quantum__rt__array_get_element_ptr_1d(%Array* %0, i64 %1)

define i32 @main(i32 %0, i8** %1)  {
  %4 = call %Array* @__quantum__rt__qubit_allocate_array(i64 2)
  %5 = call i8* @__quantum__rt__array_get_element_ptr_1d(%Array* %4, i64 0)
  %6 = bitcast i8* %5 to %Qubit**
  %7 = load %Qubit*, %Qubit** %6, align 8
  call void @__quantum__qis__h(%Qubit* %7)
  %8 = call i8* @__quantum__rt__array_get_element_ptr_1d(%Array* %4, i64 1)
  %9 = bitcast i8* %8 to %Qubit**
  %10 = load %Qubit*, %Qubit** %9, align 8
  call void @__quantum__qis__cnot(%Qubit* %7, %Qubit* %10)
  %11 = call %Result* @__quantum__qis__mz(%Qubit* %7)
  %12 = call %Result* @__quantum__qis__mz(%Qubit* %10)
  call void @__quantum__rt__qubit_release_array(%Array* %4)
  ret i32 0
}
```

QIR Basics - Types

Quantum data types:

- `%Qubit*` ➔ qubits
- `%Result*` ➔ measurement results

Data structures

- `%Array*` ➔ arrays/vectors of elements of the same type

e.g., qubit arrays, `vector<double>`, etc.

- `%Tuple*` ➔ user-defined tuple data

e.g. `{int, %Array*}` tuple of an integer and an array

- `%Callable*` ➔ generic function object (invoke with a `%Tuple*` and return a `%Tuple*`)

❖Key data structures are represented as **pointers** to **opaque** LLVM types.

❖Each implementation to provide concrete definitions.

➔ Representing quantum programs in QIR is both source-language- and runtime-independent.

QIR Basics - Quantum Instruction Set and Runtime

Standard Gate Operations

- Define a set of quantum operations (gates) as LLVM functions.

e.g., `__quantum__qis__h(%Qubit*)` for Hadamard gates.

- Implementations can provide targeting information: list of native operations (treated as runtime-provided `__qis__` functions) and an extended instruction set based on those `__qis__` operations.

➔ no need for new LLVM instructions for quantum operations.

Runtime Operations

- API functions to create, access, and manipulate basic data-types (`Array`, `Tuple`, `Callable`, etc.)
  - Allocate/deallocate qubit arrays.
  - Create Array, Tuple, Callable.
  - Access elements in Array/Tuple and invoke Callable.
  - Memory management of allocated objects.

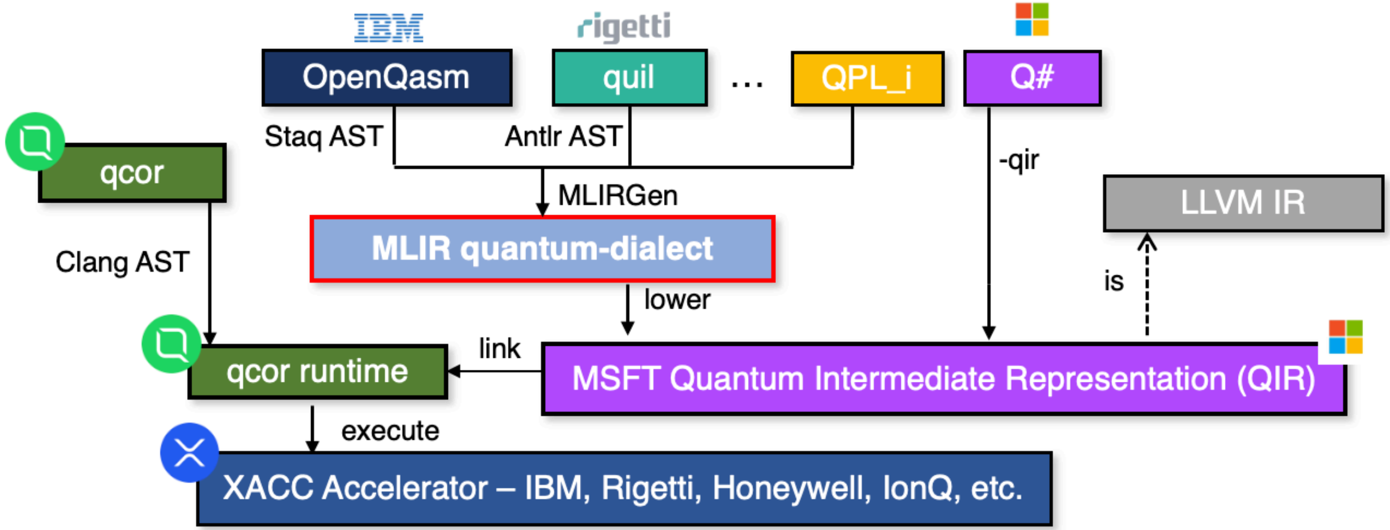
Extend `qcor` for existing quantum language compilation

- Current quantum kernel model is great, but it would be great if we could compile any stand-alone quantum representation to executable code
  - Take languages like OpenQASM (v3 or v2) and QUIL and generate executables targeting ANY quantum co-processor

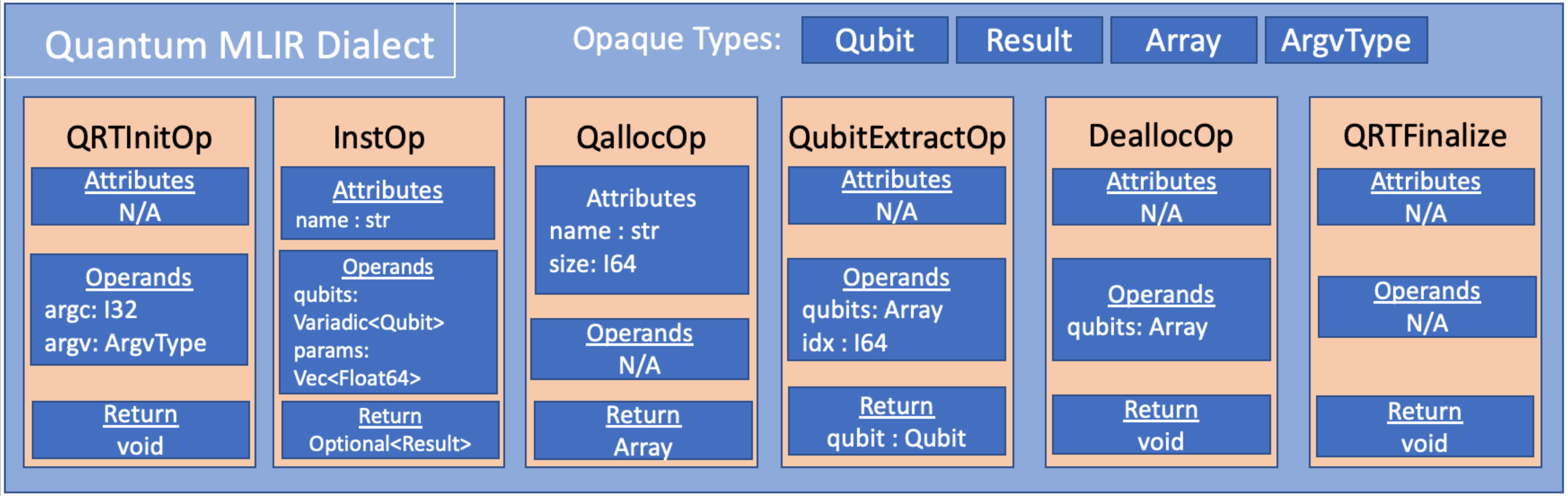
```
OPENQASM 3;
include "qelib1.inc";

const n = 10;

qubit q[n];

for i in [0:n] {
  x q[i];
}

bit c[n];
c = measure q;

for i in [0:n] {
  print("bit result", i, "=", c[i]);
}
```

```
$ qcor –qpu ibm:ibmq_vigo test.qasm
$ ./a.out

$ qcor –qpu qcs:Aspen-8 test.quil
$ ./a.out
```

We can achieve this via the integration of MLIR, `qcor`, and the QIR specification



Extending MLIR for Quantum Languages



Opaque types let the runtime library define the concrete structure

Each InstOp takes variadic list of qubits, and variadic list of double parameters

- Language features we enable as a foundation (basically follow the QIR):

1. Qubit register allocation and deallocation
2. Individual Qubit addressing / extraction from register
3. General quantum instruction modeling

3. Runtime library initialization and finalization

4. Standard Dialect for branching: if, for, while, etc…

ORNL: https://arxiv.org/pdf/2101.11365.pdf
MSFT/Zurich: https://arxiv.org/pdf/2101.11030.pdf

Lowering the Quantum Dialect to LLVM

- MLIR Pass Manager
  - Conversion Target set to LLVM Dialect
  - Quantum-to-LLVM lowering passes:
    - InstOp ---> __quantum__qis__INSTNAME(…) QIR API call
    - QallocOp ---> __quantum__rt__qubit_allocate_array(i64)
    - etc…

```
qubit q[2];
h q[0];
```

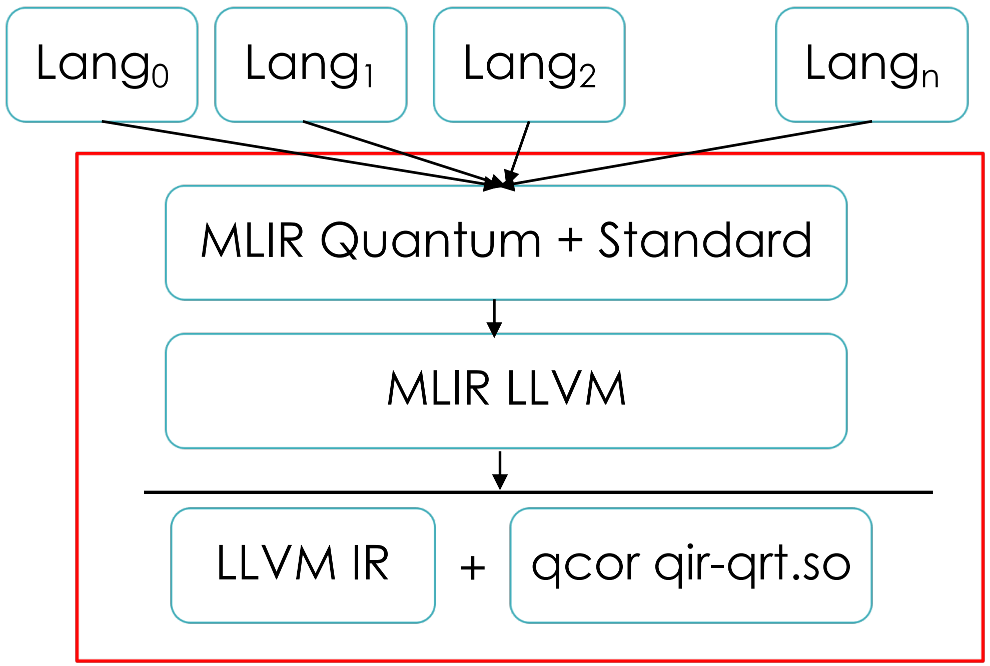
```
module  {
  func @simple() {
    %0 = "quantum.qalloc"() {name = "q", size = 2 : i64} : () -> !quantum.Array
    %c0_i64 = constant 0 : i64
    %1 = "quantum.qextract"(%0, %c0_i64) : (!quantum.Array, i64) -> !quantum.Qubit
    %2 = "quantum.inst"(%1) {name = "h", operand_segment_sizes = dense<[1, 0]> : vector<2xi32>} : (!quantum.Qubit) -> none
    "quantum.dealloc"(%0) : (!quantum.Array) -> ()
    return
  }
}
.... After LLVM IR lowering ....
define void @simple() local_unnamed_addr {
  %1 = tail call %Array* @__quantum__rt__qubit_allocate_array(i64 2)
  %2 = tail call i8* @__quantum__rt__array_get_element_ptr_1d(%Array* %1, i64 0)
  %3 = bitcast i8* %2 to %Qubit**
  %4 = load %Qubit*, %Qubit** %3, align 8
  tail call void @__quantum__qis__h(%Qubit* %4)
  tail call void @__quantum__rt__qubit_release_array(%Array* %1)
  ret void
}
```

```
// Create the PassManager for lowering
// to LLVM MLIR and run it
mlir::PassManager pm(&context);
auto q_to_llvm =
  std::make_unique<QuantumToLLVMLoweringPass>();
pm.addPass(q_to_llvm);
auto module_op = module.getOperation();
if (mlir::failed(pm.run(module_op))) {
  std::cout << "Pass Manager Failed\n";
  return 1;
}

//! module_op is now totally in LLVM Dialect !

// Now lower MLIR to LLVM IR
llvm::LLVMContext llvmContext;
auto llvmModule =
  mlir::translateModuleToLLVMIR(module,
                                llvmContext);
```

A workflow for rapid quantum compiler development via MLIR

- Steps required to map quantum languages to executable code

1. Define MLIR extensions to represent quantum language features in the MLIR IR
2. REQUIRED: Define language parsers that take source strings to an instance of the MLIR using the quantum extensions
3. Define a lowering mechanism to take quantum MLIR operations to LLVM representations – adherent with the QIR
4. Link with QIR runtime implementation



Bulk of the work is providing a general quantum dialect and a mechanism for lowering it to QIR

Rapid compiler prototyping implies just writing a parser to MLIR for they language

OpenQASM 2.0 Integration

OpenQASM 2.0 was our first prototype

```
OpenQASM 2.0;
include "qelib1.inc";
qreg q[2];
creg c[2];                  |
h q[0];                     | staq ast-gen
cx q[0], q[1];              V
measure q -> c;
```

```
|- Program
  ... qelib.inc (not included for brevity)...
  |- Register Decl(q[2], quantum)
  |- Register Decl(c[2])
  |- Declared(h)
    |- Var(q[0])
  |- Declared(cx)        key staq ast nodes:
    |- Var(q[0])            - RegisterDecl
    |- Var(q[1])            - DeclaredGate
  |- Measure                - CNOTGate
    |- Var(q[0])            - UGate
    |- Var(c[0])            - MeasureStmt
  |- Measure
    |- Var(q[1])
    |- Var(c[1])
```

- staq Clang-inspired AST
- Visitor pattern for AST nodes
- Map each node to MLIR

https://github.com/softwareQinc/staq

```
class OpenQasmMLIRGenerator : public staq::ast::Visitor {

 public:
  OpenQasmMLIRGenerator(mlir::MLIRContext &context)
      : QuantumMLIRGenerator(context) {}
  void visit(GateDecl &) override;
  void visit(RegisterDecl &) override;
  void visit(Program &prog) override;
  void visit(MeasureStmt &m) override;
  void visit(UGate &u) override;
  void visit(CNOTGate &cx) override;
  void visit(DeclaredGate &g) override;
};
```

```
void OpenQasmMLIRGenerator::visit(RegisterDecl &d) {
  auto size = d.size();
  auto name = d.id();

  auto reg_size =
    mlir::IntegerAttr::get(builder.getI64Type(), size);
  auto reg_name = builder.getStringAttr(name);

  auto allocation =
          builder.create<mlir::quantum::QallocOp>(
             location, array_type, reg_size, reg_name);
}
```

Visit nodes

```
$ qcor-mlir-tool -emit=mlir \
                          test.qasm
# lower to llvm ir, llc, and
# link to qir-qrt.so
$ qcor test.qasm
$ ./a.out -qrt nisq -shots 1000
Observed Counts:
00 : 494
11 : 506
```

```
module  {
  func @main(%arg0: i32, %arg1: !quantum.ArgvType) -> i32 {
    q.init(%arg0, %arg1)
    call @__internal_mlir_test() : () -> ()
    q.finalize()
    %c0_i32 = constant 0 : i32
    return %c0_i32 : i32
  }
  func @__internal_mlir_test() {
    %0 = q.qalloc(2) { name = q } : !quantum.Array
    %c0_i64 = constant 0 : i64
    %1 = q.extract(%0, %c0_i64) : !quantum.Qubit
    %2 = q.h(%1) : none
    %c1_i64 = constant 1 : i64
    %3 = q.extract(%0, %c1_i64) : !quantum.Qubit
    %4 = q.cx(%1, %3) : none
    %5 = q.mz(%1) : !quantum.Result
    %6 = q.mz(%3) : !quantum.Result
    q.dealloc(%0)
    return
  }
}
```

OpenQASM 3.0 Integration

https://github.com/qiskit/openqasm

QRNG in QASM 3

```
OPENQASM 3;

// Global constant, maximum bit size
// for the random integer
const max_bits = 4;

// Generate a superposition and
// measure to return a 50/50 random bit
def random_bit() qubit:a -> bit {
    h a;
    return measure a;
}

// Generate a random integer of max_bits bit width
// This will generate a random 0 or 1
// based on a single provided qubit put
// in a superposition
def generate_random_int() qubit:q -> int {
    // Create [0,0,0,...0] of size max_bits
    bit b[max_bits];

    // Set every bit as a random 0 or 1
    for i in [0:max_bits] {
        b[i] = random_bit() q;
        // reset qubit state for
        // next iteration
        reset q;
    }
    // Print the binary string
    print("random binary: ", b);

    // Cast to int and return
    return int[32](b);
}

// Allocate a single qubit
qubit a;

// Generate the random number
// using the allocated qubit
int n = generate_random_int() a;

// print the random number
print("Random int (lsb): ", n);
```

```
grammar qasm3;
program
    : header (globalStatement | statement)*
    ;
header
    : version? include*
    ;
version
    : 'OPENQASM' ( Integer | RealNumber ) SEMICOLON
    ;
...
globalStatement
    : subroutineDefinition
    | kernelDeclaration
    | quantumGateDefinition
    | calibration
    | quantumDeclarationStatement
    | pragma
    ;
statement
    : expressionStatement
    | assignmentStatement
    | classicalDeclarationStatement
    | branchingStatement
    | loopStatement
    | endStatement
    | aliasStatement
    | quantumStatement
    ;
...
```

QASM3 Features
- Classical control flow
- Subroutines and custom gates
- Variable declarations, casting
- Usual quantum operations

qcor QASM3 MLIRGen
- Provides Antlr Grammar
- We implement parse tree visitor that maps Antlr data structures to MLIR Ops
- Control flow handled with Standard Dialect (std.br)
- Variables represented as memrefs
- Lower Quantum Ops to LLVM Ops adherent to QIR
- Lower to executable code

Opportunity for Optimizations

- Concurrent MLIR work from researchers at ETH Zurich https://arxiv.org/pdf/2101.11030.pdf
  - Focus on data flow optimizations, not language lowering to executable code
  - Introduced 2 quantum dialects
    - Memory semantics, value semantics (quantumSSA)
- Quantum Optimizations via patterns in use-def chains
  - We add to our single dialect, `ValueSemanticsInstOp`
  - Easy example – ***identity pairs***

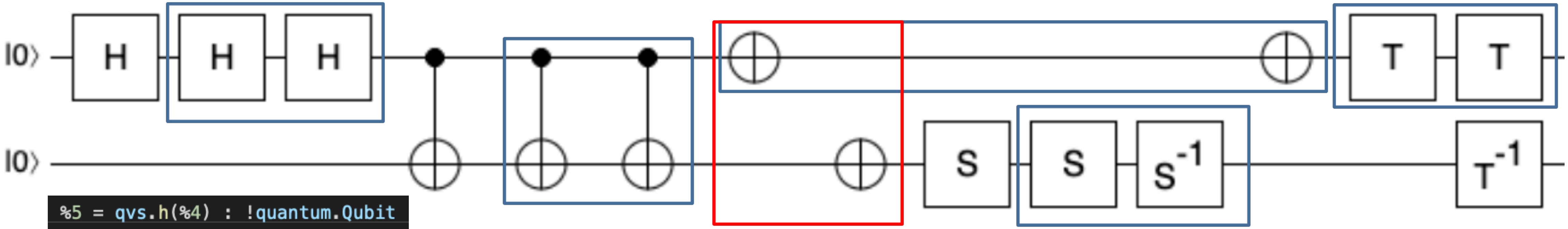
```
def ValueSemanticsInstOp : QuantumOp<"value_inst", [AttrSizedOperandSegments]> {
    let arguments = (ins StrAttr:$name, Variadic<QubitType>:$qubits, Variadic<F64>:$params);
    let results = (outs Variadic<AnyTypeOf<[ResultType, QubitType]>>:$result);

    let printer = [{  auto op = *this;
  p << "qvs." << op.name() << "(" << op.getOperands() << ") : " << op.result().getType(); }];
}
```

```
module  {
  func @__internal_mlir_small() -> i32 {
    %0 = q.qalloc(1) { name = q } : !quantum.Array
    %c0_i64 = constant 0 : i64
    %1 = q.extract(%0, %c0_i64) : !quantum.Qubit
    %2 = q.qalloc(1) { name = r } : !quantum.Array
    %3 = q.extract(%2, %c0_i64) : !quantum.Qubit
    %4 = qvs.h(%1) : !quantum.Qubit
    %5 = qvs.h(%4) : !quantum.Qubit
    %6 = qvs.h(%5) : !quantum.Qubit
    %7:2 = qvs.cnot(%6, %3) : !quantum.Qubit, !quantum.Qubit
    %8:2 = qvs.cnot(%7#0, %7#1) : !quantum.Qubit, !quantum.Qubit
    %9:2 = qvs.cnot(%8#0, %8#1) : !quantum.Qubit, !quantum.Qubit
    %10:2 = qvs.cnot(%9#1, %9#0) : !quantum.Qubit, !quantum.Qubit
    %11:2 = qvs.cnot(%10#0, %10#1) : !quantum.Qubit, !quantum.Qubit
    %12 = qvs.x(%11#1) : !quantum.Qubit
    %13 = qvs.x(%11#0) : !quantum.Qubit
    %14 = qvs.s(%13) : !quantum.Qubit
    %15 = qvs.s(%14) : !quantum.Qubit
    %16 = qvs.sdg(%15) : !quantum.Qubit
    %17 = qvs.x(%12) : !quantum.Qubit
    %18 = qvs.t(%17) : !quantum.Qubit
    %19 = qvs.tdg(%18) : !quantum.Qubit
    %20 = qvs.tdg(%16) : !quantum.Qubit
    q.dealloc(%0)
    q.dealloc(%2)
    %c0_i32 = constant 0 : i32
    return %c0_i32 : i32
  }
}
```

```
H H == I
X X == I
CX(a,b) CX(a,b) == I
```

Opportunity for Optimizations

Contrived example




```
%5 = qvs.h(%4) : !quantum.Qubit
%6 = qvs.h(%5) : !quantum.Qubit
```

Consumes %4 produces %5
Consumes %5 produces %6

Qubit lifeline is just following the user list for the operands and returns

```
%11:2 = qvs.cnot(%10#0, %10#1) : !quantum.Qubit, !quantum.Qubit
%12 = qvs.x(%11#1) : !quantum.Qubit
%13 = qvs.x(%11#0) : !quantum.Qubit
```

Identity pair (X X) but using different qubit results

MLIR Optimization: find users of result, if user + producer forms an identity pair, remove and replace result uses with first operand

Opportunity for Optimizations

```
...
    %4 = qvs.h(%1) : !quantum.Qubit
    %5 = qvs.h(%4) : !quantum.Qubit
    %6 = qvs.h(%5) : !quantum.Qubit
    %7:2 = qvs.cnot(%6, %3) : !quantum.Qubit, !quantum.Qubit
    %8:2 = qvs.cnot(%7#0, %7#1) : !quantum.Qubit, !quantum.Qubit
    %9:2 = qvs.cnot(%8#0, %8#1) : !quantum.Qubit, !quantum.Qubit
    %10:2 = qvs.cnot(%9#1, %9#0) : !quantum.Qubit, !quantum.Qubit
    %11:2 = qvs.cnot(%10#0, %10#1) : !quantum.Qubit, !quantum.Qubit
    %12 = qvs.x(%11#1) : !quantum.Qubit
    %13 = qvs.x(%11#0) : !quantum.Qubit
    %14 = qvs.s(%13) : !quantum.Qubit
    %15 = qvs.s(%14) : !quantum.Qubit
    %16 = qvs.sdg(%15) : !quantum.Qubit
…
```

```
$ qcor-mlir-tool -emit=llvm test.qasm -q-optimize
```

```
define i32 @__internal_mlir_small() local_unnamed_addr {
  %1 = tail call %Array* @__quantum__rt__qubit_allocate_array(i64 1)
  %2 = tail call i8* @__quantum__rt__array_get_element_ptr_1d(%Array* %1, i64 0)
  %3 = bitcast i8* %2 to %Qubit**
  %4 = load %Qubit*, %Qubit** %3, align 8
  %5 = tail call %Array* @__quantum__rt__qubit_allocate_array(i64 1)
  %6 = tail call i8* @__quantum__rt__array_get_element_ptr_1d(%Array* %5, i64 0)
  %7 = bitcast i8* %6 to %Qubit**
  %8 = load %Qubit*, %Qubit** %7, align 8
  tail call void @__quantum__qis__h(%Qubit* %4)
  tail call void @__quantum__qis__cnot(%Qubit* %4, %Qubit* %8)
  tail call void @__quantum__qis__x(%Qubit* %8)
  tail call void @__quantum__qis__s(%Qubit* %8)
  tail call void @__quantum__qis__tdg(%Qubit* %8)
  tail call void @__quantum__rt__qubit_release_array(%Array* %1)
  tail call void @__quantum__rt__qubit_release_array(%Array* %5)
  ret i32 0
}
```

Single Qubit Identity Pair Removal

```
mlir::LogicalResult matchAndRewrite(
    mlir::quantum::ValueSemanticsInstOp op,
    mlir::PatternRewriter& rewriter) const override {

  auto inst_name = op.name();
  auto return_value = *op.result().begin();
  if (return_value.hasOneUse()) {
    // get that one user
    auto user = *return_value.user_begin();
    // cast to a inst op
    if (auto next_inst =
            dyn_cast_or_null<mlir::quantum::ValueSemanticsInstOp>(user)) {
      // check that it is one of our known id pairs
      if (should_remove(next_inst.name().str(), inst_name.str())) {

        // need to get users of next_inst and point them to use
        // op.getOperands
        (*next_inst.result_begin()).replaceAllUsesWith(op.getOperand(0));

        rewriter.eraseOp(op);
        rewriter.eraseOp(next_inst);

        return success();
      }
    }
  }

  return failure();
}
```

The QIR enables integration of language approaches

- To get executable code, we implement the QIR specification API with `qcor`
  - Opaque `Qubits` and `Array<Qubit>` map to qcor qubits and qreg
  - Opaque `Results` map to `i1`
  - Instruction functions delegate to `qcor` Quantum Runtime
  - Can run in NISQ or FTQC mode
  - Can compile with or without `main()` entrypoint
  - Without entrypoint, one can include compiled libraries in existing C++ code

```
$ qcor bell.qasm
$ ./a.out -qrt nisq -qpu ibm:ibmq_paris
```

```
$ qcor -no-entrypoint bell.qasm
$ ls
   bell.o bell.qasm
----------------------------------------------
#include "qcor.hpp"

// Macro that maps to
// extern "C" void bell(qreg);
include_qcor_qasm(bell)

int main() {
   auto q = qalloc(2);
   // Function from bell.o
   bell(q)
   for (auto [bit, count] : q.counts()) {
      print(bit, ":", count);
   }
   return 0;
}
----------------------------------------------
$ qcor bell.o test.cpp -o test.x
$ ./test.x -qrt nisq -shots 2048 -qpu aer
00 : 1025
11 : 1023
```

Thanks!

We are focused on the development of frameworks and re-targetable compilers for near-term and future fault-tolerant heterogeneous quantum-classical computing.

Docs: http://docs.aide-qc.org

- Everything here is open source
  - https://github.com/ornl-qci/qcor
- Recent papers
  - mlir-quantum: https://arxiv.org/pdf/2101.11365.pdf
  - qcor: https://arxiv.org/abs/2010.03935
- Contact: mccaskeyaj@ornl.gov
- Funding Acknowledgment: DOE NQI/QSC, ARQC, QCAT programs