OAK RIDGE National Laboratory

Integrating an MLIR Pass into the Catalyst MLIR Compiler to Close the Loop for Qubit-Wise Commutativity

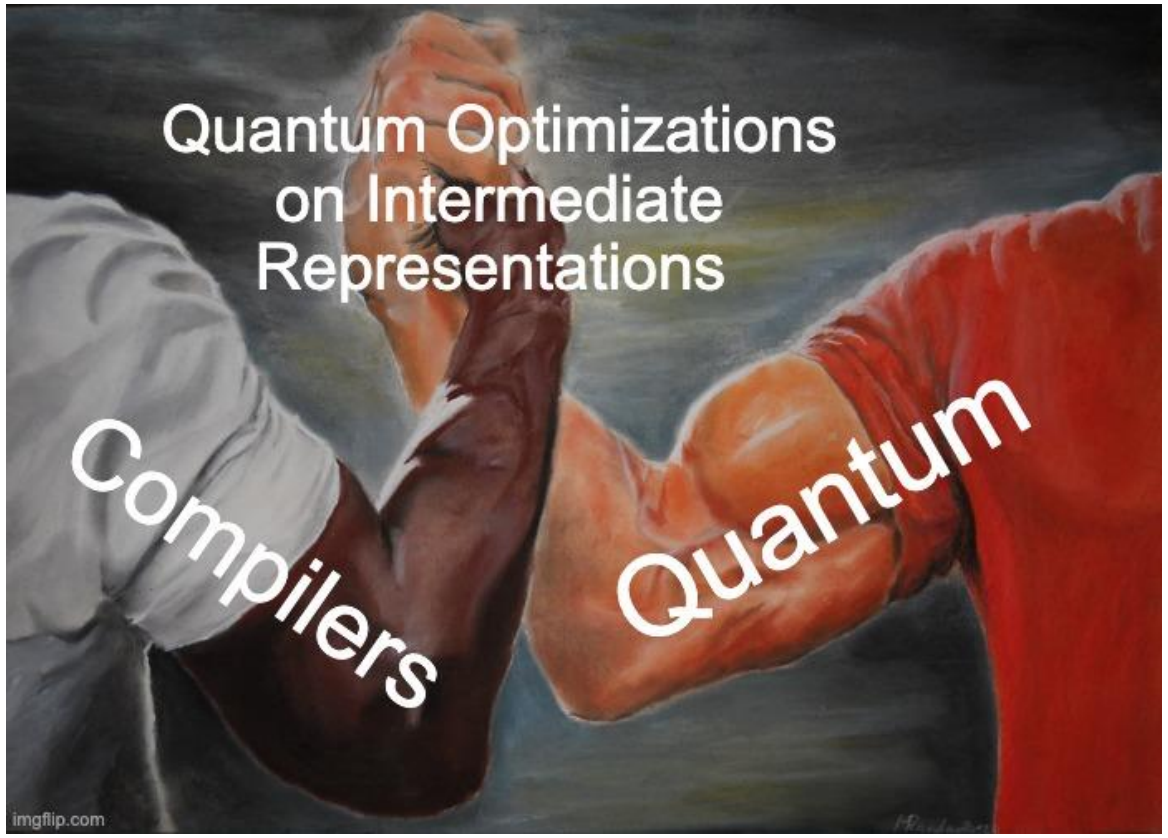
Anthony Cabrera*, Sharmin Afrose, Daniel Claudino, Travis Humble
*Research Scientist @ ORNL, cabreraam@ornl.gov

Quantum Software 2.0 Workshop
IEEE Quantum Week
September 19, 2024

Goal of this Talk

- Present a brief overview of MLIR as a framework for quantum compiler frontends
- Detail how we added a compiler pass into Catalyst MLIR quantum compiler frontend (and convince you that you can, too!)
- Elaborate on the importance and opportunities created by working at the IR level

Why work at the intersection of compilers and quantum computing, anyway?



- Intermediate representation mirrors ISA relationship between classical HW & SW
  - SW lowers to an IR and HW implements IR
- Leverage domain specific compilation applicable to *all* quantum programs
  - Lower all input languages to common IR then perform quantum-flavored optimizations
- Reason about optimizations at the level of quantum algorithm instead of quantum gates
  - Large semantic gap between quantum algorithm and gates closed by IR inserted in between

Quick MLIR Overview

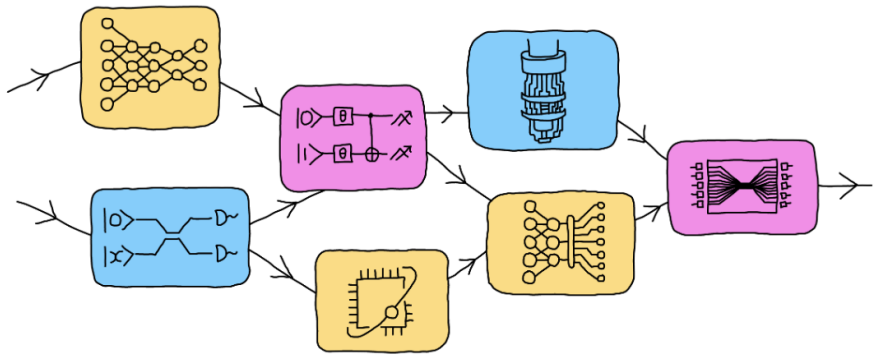
- A framework that provides the capability of authoring domain-specific intermediate representations
  - This allows us to reason about quantum at a higher abstraction level – MLIR Dialects – instead of quantum gates – e.g., QIR, QASM
- Relies on "progressive lowering"
  - Starting from a level of abstraction closer to a given domain and then lowering that representation into lower and lower levels of abstraction until we get to LLVM IR

Pennylane and Catalyst

- Two repositories of interest
  - PennyLane is the Quantum Python frontend from Xanadu
  - Catalyst is the JIT compiler built on MLIR that takes circuits defined in PennyLane

PennyLane is a cross-platform Python library for quantum computing, quantum machine learning, and quantum chemistry.

Train a quantum computer the same way as a neural network.



PENNYLANE

CATALYST BETA

Catalyst is an experimental package that enables just-in-time (JIT) compilation of hybrid quantum-classical programs.

Catalyst is currently under heavy development — if you have suggestions on the API or use-cases you'd like to be covered, please open an GitHub issue or reach out. We'd love to hear about how you're using the library, collaborate on development, or integrate additional devices and frontends.

The Problem:
Reduction of Observables in Hamiltonian

- Basing it off this PennyLane blog post that demonstrates Qubit-wise commutativity (QWC) as simple method of reducing measurements.

$$\text{cost}(\theta) = \langle 0|U(\theta)^\dagger \left(\sum_i c_i h_i\right) U(\theta)|0\rangle = \sum_i c_i \langle 0|U(\theta)^\dagger h_i U(\theta)|0\rangle.$$

Want to reduce number of terms

The Problem:
Reduction of Observables in Hamiltonian

- Basing it off this PennyLane blog post that demonstrates Qubit-wise commutativity (QWC) as simple method of reducing measurements.

$$\text{cost}(\theta) = \langle 0|U(\theta)^\dagger \left(\sum_i c_i h_i\right) U(\theta)|0\rangle = \sum_i c_i \langle 0|U(\theta)^\dagger h_i U(\theta)|0\rangle.$$

Want to reduce number of terms

- While PennyLane has Python libraries that can group observables via QWC, we would like embed this analysis into the compiler

The Problem:
Reduction of Observables in Hamiltonian

- Basing it off this PennyLane blog post that demonstrates Qubit-wise commutativity (QWC) as simple method of reducing measurements.

$$\text{cost}(\theta) = \langle 0|U(\theta)^\dagger \left(\sum_i c_i h_i\right) U(\theta)|0\rangle = \sum_i c_i \langle 0|U(\theta)^\dagger h_i U(\theta)|0\rangle.$$

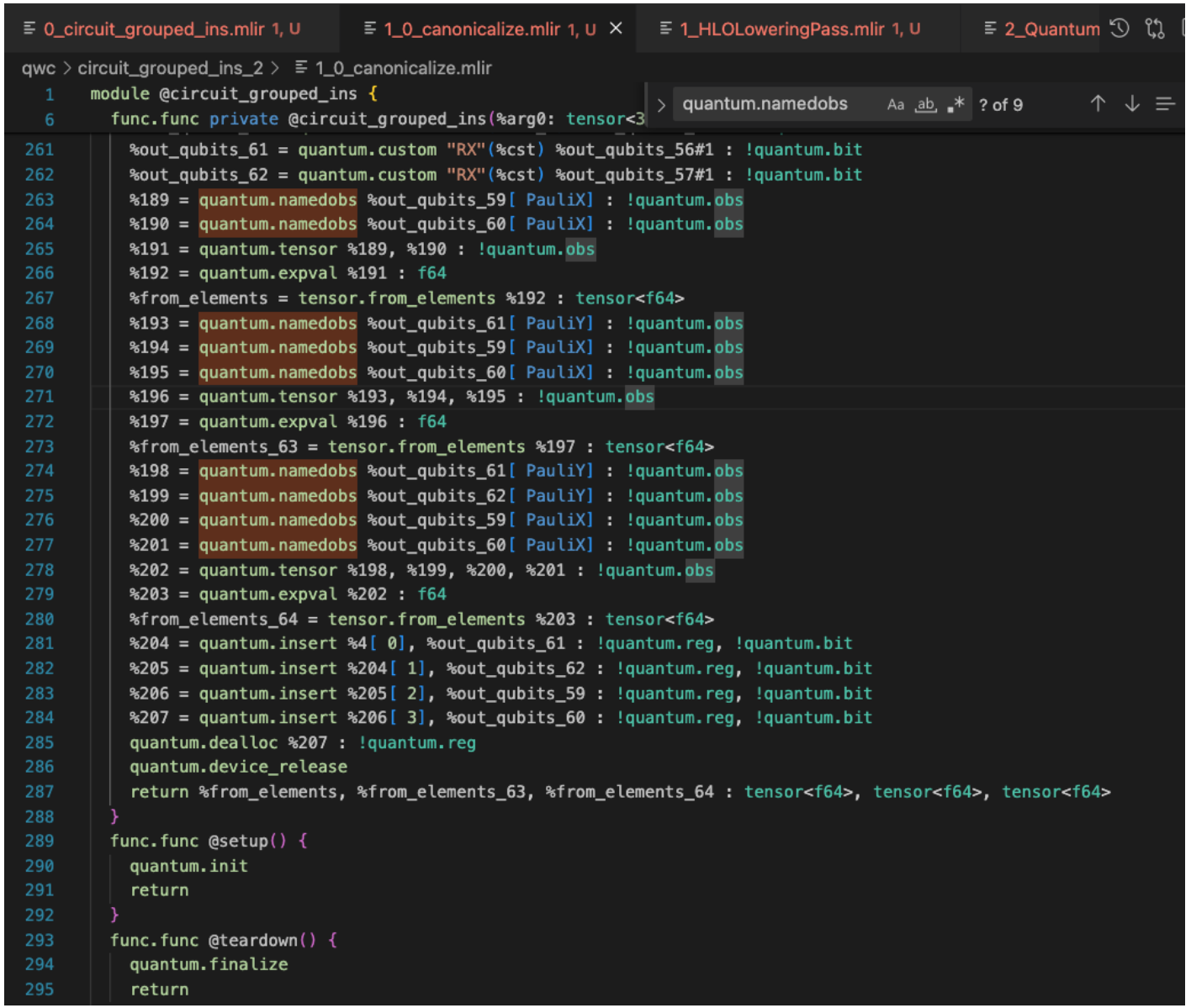
Want to reduce number of terms

- While PennyLane has Python libraries that can group observables via QWC, we would like embed this analysis into the compiler
- Intuition:
  - We want to solve an optimization problem using an iterative approach
  - Each iteration becomes more expensive as molecules get bigger
  - We want to make iterations cheaper by doing less work (i.e., compute)

What We Actually Did

- Before we proceeded, we wanted to make sure that the PennyLane/Catalyst flow didn't already do this
  - Ask me offline about my VSCode setup to walk through the PennyLane libraries and Catalyst C++ components!
- What we found was that the flow *almost* handles QWC correctly, but does not fully account for QWC
  - If number of commuting groups > 1, are all rotated into the correct shared eigenbasis, but the named measurement bases have not been changed to reflect the standard basis
  - If there's only one group in the set of observables, the measurement bases aren't correct **and** the correct rotations to the measurement basis are not inserted

Should be observations in the Z-basis…



What We Actually Did

- Before we proceeded, we wanted to make sure that the PennyLane/Catalyst flow didn't already do this
- What we found was that the flow *almost* handles QWC correctly, but does not fully account for QWC
  - If number of commuting groups > 1, are all rotated into the correct shared eigenbasis, but the named measurement bases have not been changed to reflect the standard basis
  - If there's only one group in the set of observables, the measurement bases aren't correct **and** the correct rotations to the measurement basis are not inserted
- To address this, **we introduce a compiler pass to detect these patterns and close the loop**

Determining how to proceed from prior pull requests

- To be fair, there is some documentation on writing the pattern matching and rewrites, but I want to show what files actually need to be modified/created

This file was added as part of a PR to add a new MLIR dialect and compiler pass. Maybe we can look at the PR to get a sense of what files we need to modify/create.

Check for Relevant Documentation

Look at Commit History

[MLIR] Mitigation dialect, ZNE op and lowering(global folding) #324

Code

Merged **rmoyard** merged 46 commits into `main` from `mitigation_dialect` on Dec 20, 2023

Conversation 67 Commits 46 Checks 20 Files changed 34 +955 −2

Changes from all commits File filter Conversations 0 / 34 files viewed Review in codespace Review changes

Filter changed files

- doc
  - changelog.md
- mlir
  - include
    - CAPI
      - Dialects.h
      - CMakeLists.txt
    - Mitigation
      - CMakeLists.txt
      - IR
        - CMakeLists.txt
        - MitigationDialect.h
        - MitigationDialect.td
        - MitigationOps.h
        - MitigationOps.td
      - Transforms
        - CMakeLists.txt
        - Passes.h
        - Passes.td
        - Patterns.h
  - lib
    - CAPI
      - CMakeLists.txt
      - Dialects.cpp

4 doc/changelog.md Viewed

```
@@ -2,6 +2,10 @@
 <h3>New features</h3>
+* A mitigation dialect (MLIR) was added. It initially contains a Zero Noise Extrapolation (ZNE) operation,
+  with a lowering to a global folded circuit.
+  [(#324)](https://github.com/PennyLaneAI/catalyst/pull/324)
+
 * Initial support for transforms. QFunc transforms are supported. QNode transforms have limited
   support. QNode transforms cannot be composed, and transforms are limited to what is currently
   available in PennyLane. This means that operations defined in Catalyst like `cond`, `for_loop`,
```

1 mlir/include/CAPI/Dialects.h Viewed

```
@@ -22,6 +22,7 @@ extern "C" {
 MLIR_DECLARE_CAPI_DIALECT_REGISTRATION(Quantum, quantum);
 MLIR_DECLARE_CAPI_DIALECT_REGISTRATION(Gradient, gradient);
+MLIR_DECLARE_CAPI_DIALECT_REGISTRATION(Mitigation, mitigation);
 MLIR_DECLARE_CAPI_DIALECT_REGISTRATION(Catalyst, catalyst);
 #ifdef __cplusplus
```

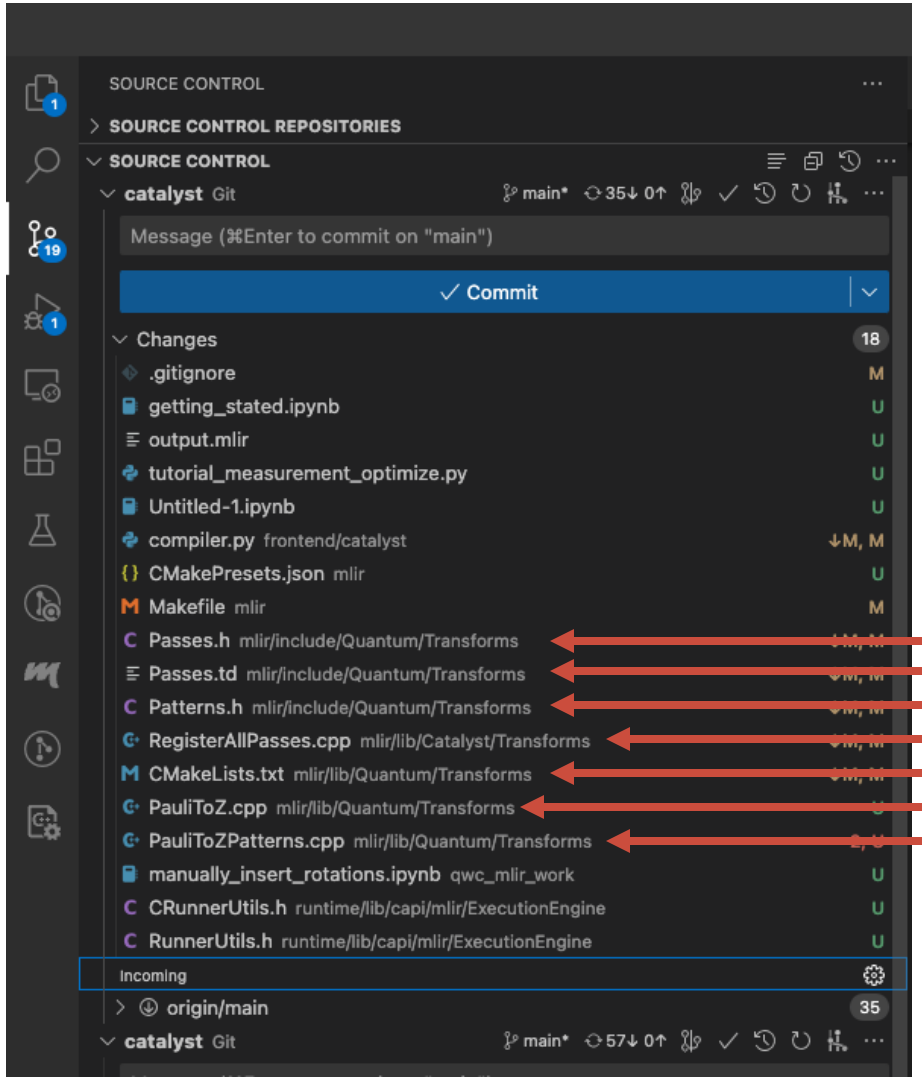
1 mlir/include/CMakeLists.txt Viewed

```
@@ -1,4 +1,5 @@
 add_subdirectory(Catalyst)
 add_subdirectory(Quantum)
 add_subdirectory(Gradient)
+add_subdirectory(Mitigation)
 add_subdirectory(Test)
```

2 mlir/include/Mitigation/CMakeLists.txt Viewed

```
@@ -0,0 +1,2 @@
```

What Files to Touch?



Create the pass:

- `PauliToZ.cpp`
- `PauliToZPatterns.cpp`

Make Pass visible:

- `RegisterAllPasses.cpp`

Add source files for building

- `mlir/lib/Quantum/Transforms/CMakeLists.txt`

Add pass boilerplate for tablegen:

- `Passes.h`
- `Passes.td`
- `Patterns.h`

OAK RIDGE National Laboratory

Adding the Pass

Writing the Pass (Core Logic)

Algorithm 1 Partial QWC Match and Rewrite Logic

Require: currOp = NamedObsOp and currOp.type $\notin \{Z, I\}$

1: inQubit $\leftarrow$ currOp.inQubit
2: parentOp $\leftarrow$ `definingOp`(inQubit)
3: **if** parentOp $\in \{\text{rotY}(\frac{-\pi}{2}), \text{rotX}(\frac{\pi}{2})\}$ **then**
4: | currOp.type $\leftarrow$ Z ▷ *already correctly rotated*
5: **else** ▷ *rotate into computational basis*
6: | rotOp $\leftarrow$ currOp.type = X ? $\text{rotY}(\frac{-\pi}{2})$: $\text{rotX}(\frac{\pi}{2})$
7: | `Insert`(rotOp)
8: | newOp $\leftarrow$ `Create`(NamedObsOp, rotOp.result, Z)
9: | `ReplaceWithNewOp`(currOp,newOp)

```
LogicalResult matchAndRewrite(NamedObsOp op,
    PatternRewriter& rewriter) const override {
  using NamedObsOp = _NOO_;
  auto namedObs = op.getTypeAttr().getValue();
  if (namedObs != NamedObservable::PauliZ
    && namedObs != NamedObservable::Identity) {
      auto arg = op.getQubit();
      auto defOp = arg.getdefOp();
      if (isa<CustomOp>(defOp)) {
        auto neg_pi_div2 = APFloat(-1.570796326794896);
        auto pos_pi_div2 = APFloat(1.5707963267948966);
          auto parentOp = cast<CustomOp>(defOp);
          auto gateName = parentOp.getGateName();
          if (gateName.equals("RX") ||
            gateName.equals("RY")) {
              auto rotAngleOp =
              parentOp.getParams()[0].getdefOp();
              if (isa<arith::ConstantFloatOp>
                  (rotAngleOp)) {
                auto arithConstFloatOp =
                  cast<arith::ConstantFloatOp>
                    (rotAngleOp);
                  auto rotAngle =
                    arithConstFloatOp.value();
                  if (namedObs ==
                    NamedObservable::PauliX &&
                    gateName.equals("RY") &&
                    rotAngle.compare(neg_pi_div2))
                  {
                    rewriter.replaceOpWithNewOp
                      <_NOO_>(op,
                      op.getResult().getType(),
                      op.getQubit(),
                      NamedObservable::PauliZ);
                    return success();
                  } else if (namedObs ==
                    NamedObservable::PauliY &&
                    gateName.equals("RX") &&
                    rotAngle.compare(pos_pi_div2))
                    {
                      rewriter.replaceOpWithNewOp
                      <_NOO_>(op,
                        op.getResult().getType(),
                        op.getQubit(),
                        NamedObservable::PauliZ);
                      return success();}}}
```

```
          else {
            Location loc = op.getLoc();
            auto rotAxis = namedObs
              == NamedObservable::PauliX ?
              StringRef{"RY"} : StringRef{"RX"};
              auto rotAngle = namedObs ==
                NamedObservable::PauliX ? neg_pi_div2 :
                  pos_pi_div2;
              auto rotAngleVal =
                rewriter.create
                  <arith::ConstantFloatOp>(
                  loc, rotAngle,
                  FloatType::getF64(
                    rewriter.getContext()));
              auto resultTy = arg.getType();
              auto rotOp = rewriter.create<CustomOp>(
                loc, TypeRange{resultTy}, TypeRange{},
                ValueRange{rotAngleVal}, ValueRange{arg},
                rotAxis, UnitAttr{}, ValueRange{},
                ValueRange{});
              auto finalTy = op.getResult().getType();
              rewriter.replaceOpWithNewOp<_NOO_>(
                op, finalTy,
                rotOp.getQubitResults().front(),
                NamedObservable::PauliZ);
        } return success();}}
return failure();}
```

Writing the Pass (Core Logic)

Algorithm 1 Partial QWC Match and Rewrite Logic

Require: currOp = NamedObsOp and currOp.type $\notin \{Z, I\}$

1: inQubit $\leftarrow$ currOp.inQubit
2: parentOp $\leftarrow$ `definingOp`(inQubit)
3: **if** parentOp $\in \{\text{rotY}(\frac{-\pi}{2}), \text{rotX}(\frac{\pi}{2})\}$ **then**
4: | currOp.type $\leftarrow$ Z ▷ *already correctly rotated*
5: **else** ▷ *rotate into computational basis*
6: | rotOp $\leftarrow$ currOp.type = X ? $\text{rotY}(\frac{-\pi}{2})$: $\text{rotX}(\frac{\pi}{2})$
7: | `Insert`(rotOp)
8: | newOp $\leftarrow$ `Create`(NamedObsOp, rotOp.result, Z)
9: | `ReplaceWithNewOp`(currOp,newOp)

```
LogicalResult matchAndRewrite(NamedObsOp op,
    PatternRewriter& rewriter) const override {
  using NamedObsOp = _NOO_;
  auto namedObs = op.getTypeAttr().getValue();
  if (namedObs != NamedObservable::PauliZ
    && namedObs != NamedObservable::Identity) {
      auto arg = op.getQubit();
      auto defOp = arg.getdefOp();
      if (isa<CustomOp>(defOp)) {
        auto neg_pi_div2 = APFloat(-1.570796326794896);
        auto pos_pi_div2 = APFloat(1.5707963267948966);
          auto parentOp = cast<CustomOp>(defOp);
          auto gateName = parentOp.getGateName();
          if (gateName.equals("RX") ||
            gateName.equals("RY")) {
              auto rotAngleOp =
              parentOp.getParams()[0].getdefOp();
              if (isa<arith::ConstantFloatOp>
                  (rotAngleOp)) {
                auto arithConstFloatOp =
                  cast<arith::ConstantFloatOp>
                    (rotAngleOp);
                  auto rotAngle =
                    arithConstFloatOp.value();
                  if (namedObs ==
                    NamedObservable::PauliX &&
                    gateName.equals("RY") &&
                    rotAngle.compare(neg_pi_div2))
                  {
                    rewriter.replaceOpWithNewOp
                      <_NOO_>(op,
                      op.getResult().getType(),
                      op.getQubit(),
                      NamedObservable::PauliZ);
                    return success();
                  } else if (namedObs ==
                    NamedObservable::PauliY &&
                    gateName.equals("RX") &&
                    rotAngle.compare(pos_pi_div2))
                    {
                      rewriter.replaceOpWithNewOp
                      <_NOO_>(op,
                        op.getResult().getType(),
                        op.getQubit(),
                        NamedObservable::PauliZ);
                      return success();}}}
```

```
          else {
            Location loc = op.getLoc();
            auto rotAxis = namedObs
              == NamedObservable::PauliX ?
              StringRef{"RY"} : StringRef{"RX"};
              auto rotAngle = namedObs ==
                NamedObservable::PauliX ? neg_pi_div2 :
                  pos_pi_div2;
              auto rotAngleVal =
                rewriter.create
                  <arith::ConstantFloatOp>(
                  loc, rotAngle,
                  FloatType::getF64(
                    rewriter.getContext()));
              auto resultTy = arg.getType();
              auto rotOp = rewriter.create<CustomOp>(
                loc, TypeRange{resultTy}, TypeRange{},
                ValueRange{rotAngleVal}, ValueRange{arg},
                rotAxis, UnitAttr{}, ValueRange{},
                ValueRange{});
              auto finalTy = op.getResult().getType();
              rewriter.replaceOpWithNewOp<_NOO_>(
                op, finalTy,
                rotOp.getQubitResults().front(),
                NamedObservable::PauliZ);
        } return success();}}
return failure();}
```

Writing the Pass (Core Logic)

Algorithm 1 Partial QWC Match and Rewrite Logic

Require: currOp = NamedObsOp and currOp.type $\notin \{Z, I\}$

1: inQubit $\leftarrow$ currOp.inQubit
2: parentOp $\leftarrow$ `definingOp`(inQubit)
3: **if** parentOp $\in \{\text{rotY}(\frac{-\pi}{2}), \text{rotX}(\frac{\pi}{2})\}$ **then**
4: | currOp.type $\leftarrow$ Z ▷ *already correctly rotated*
5: **else** ▷ *rotate into computational basis*
6: | rotOp $\leftarrow$ currOp.type = X ? $\text{rotY}(\frac{-\pi}{2})$: $\text{rotX}(\frac{\pi}{2})$
7: | `Insert`(rotOp)
8: | newOp $\leftarrow$ `Create`(NamedObsOp, rotOp.result, Z)
9: | `ReplaceWithNewOp`(currOp,newOp)

```
LogicalResult matchAndRewrite(NamedObsOp op,
    PatternRewriter& rewriter) const override {
  using NamedObsOp = _NOO_;
  auto namedObs = op.getTypeAttr().getValue();
  if (namedObs != NamedObservable::PauliZ
    && namedObs != NamedObservable::Identity) {
      auto arg = op.getQubit();
      auto defOp = arg.getdefOp();
      if (isa<CustomOp>(defOp)) {
        auto neg_pi_div2 = APFloat(-1.570796326794896);
        auto pos_pi_div2 = APFloat(1.5707963267948966);
          auto parentOp = cast<CustomOp>(defOp);
          auto gateName = parentOp.getGateName();
          if (gateName.equals("RX") ||
            gateName.equals("RY")) {
              auto rotAngleOp =
              parentOp.getParams()[0].getdefOp();
              if (isa<arith::ConstantFloatOp>
                  (rotAngleOp)) {
                auto arithConstFloatOp =
                  cast<arith::ConstantFloatOp>
                    (rotAngleOp);
                  auto rotAngle =
                    arithConstFloatOp.value();
                  if (namedObs ==
                    NamedObservable::PauliX &&
                    gateName.equals("RY") &&
                    rotAngle.compare(neg_pi_div2))
                  {
                    rewriter.replaceOpWithNewOp
                      <_NOO_>(op,
                      op.getResult().getType(),
                      op.getQubit(),
                      NamedObservable::PauliZ);
                    return success();
                  } else if (namedObs ==
                    NamedObservable::PauliY &&
                    gateName.equals("RX") &&
                    rotAngle.compare(pos_pi_div2))
                    {
                      rewriter.replaceOpWithNewOp
                      <_NOO_>(op,
                        op.getResult().getType(),
                        op.getQubit(),
                        NamedObservable::PauliZ);
                      return success();}}}
```

```
          else {
            Location loc = op.getLoc();
            auto rotAxis = namedObs
              == NamedObservable::PauliX ?
              StringRef{"RY"} : StringRef{"RX"};
              auto rotAngle = namedObs ==
                NamedObservable::PauliX ? neg_pi_div2 :
                  pos_pi_div2;
              auto rotAngleVal =
                rewriter.create
                  <arith::ConstantFloatOp>(
                  loc, rotAngle,
                  FloatType::getF64(
                    rewriter.getContext()));
              auto resultTy = arg.getType();
              auto rotOp = rewriter.create<CustomOp>(
                loc, TypeRange{resultTy}, TypeRange{},
                ValueRange{rotAngleVal}, ValueRange{arg},
                rotAxis, UnitAttr{}, ValueRange{},
                ValueRange{});
              auto finalTy = op.getResult().getType();
              rewriter.replaceOpWithNewOp<_NOO_>(
                op, finalTy,
                rotOp.getQubitResults().front(),
                NamedObservable::PauliZ);
        } return success();}}
return failure();}
```

Writing the Pass (Core Logic)

Algorithm 1 Partial QWC Match and Rewrite Logic

Require: currOp = NamedObsOp and currOp.type $\notin \{Z, I\}$

1: inQubit $\leftarrow$ currOp.inQubit
2: parentOp $\leftarrow$ `definingOp`(inQubit)
3: **if** parentOp $\in \{\text{rotY}(\frac{-\pi}{2}), \text{rotX}(\frac{\pi}{2})\}$ **then**
4: | currOp.type $\leftarrow$ Z ▷ *already correctly rotated*
5: **else** ▷ *rotate into computational basis*
6: | rotOp $\leftarrow$ currOp.type = X ? $\text{rotY}(\frac{-\pi}{2})$: $\text{rotX}(\frac{\pi}{2})$
7: | `Insert`(rotOp)
8: | newOp $\leftarrow$ `Create`(NamedObsOp, rotOp.result, Z)
9: | `ReplaceWithNewOp`(currOp,newOp)

```
LogicalResult matchAndRewrite(NamedObsOp op,
    PatternRewriter& rewriter) const override {
  using NamedObsOp = _NOO_;
  auto namedObs = op.getTypeAttr().getValue();
  if (namedObs != NamedObservable::PauliZ
    && namedObs != NamedObservable::Identity) {
      auto arg = op.getQubit();
      auto defOp = arg.getdefOp();
      if (isa<CustomOp>(defOp)) {
        auto neg_pi_div2 = APFloat(-1.570796326794896);
        auto pos_pi_div2 = APFloat(1.5707963267948966);
          auto parentOp = cast<CustomOp>(defOp);
          auto gateName = parentOp.getGateName();
          if (gateName.equals("RX") ||
            gateName.equals("RY")) {
              auto rotAngleOp =
              parentOp.getParams()[0].getdefOp();
              if (isa<arith::ConstantFloatOp>
                  (rotAngleOp)) {
                auto arithConstFloatOp =
                  cast<arith::ConstantFloatOp>
                    (rotAngleOp);
                  auto rotAngle =
                    arithConstFloatOp.value();
                  if (namedObs ==
                    NamedObservable::PauliX &&
                    gateName.equals("RY") &&
                    rotAngle.compare(neg_pi_div2))
                  {
                    rewriter.replaceOpWithNewOp
                      <_NOO_>(op,
                      op.getResult().getType(),
                      op.getQubit(),
                      NamedObservable::PauliZ);
                    return success();
                  } else if (namedObs ==
                    NamedObservable::PauliY &&
                    gateName.equals("RX") &&
                    rotAngle.compare(pos_pi_div2))
                    {
                      rewriter.replaceOpWithNewOp
                      <_NOO_>(op,
                        op.getResult().getType(),
                        op.getQubit(),
                        NamedObservable::PauliZ);
                    return success();}}}
          else {
            Location loc = op.getLoc();
            auto rotAxis = namedObs
              == NamedObservable::PauliX ?
              StringRef{"RY"} : StringRef{"RX"};
              auto rotAngle = namedObs ==
                NamedObservable::PauliX ? neg_pi_div2 :
                  pos_pi_div2;
              auto rotAngleVal =
                rewriter.create
                  <arith::ConstantFloatOp>(
                  loc, rotAngle,
                  FloatType::getF64(
                    rewriter.getContext()));
              auto resultTy = arg.getType();
              auto rotOp = rewriter.create<CustomOp>(
                loc, TypeRange{resultTy}, TypeRange{},
                ValueRange{rotAngleVal}, ValueRange{arg},
                rotAxis, UnitAttr{}, ValueRange{},
                ValueRange{});
              auto finalTy = op.getResult().getType();
              rewriter.replaceOpWithNewOp<_NOO_>(
                op, finalTy,
                rotOp.getQubitResults().front(),
                NamedObservable::PauliZ);
        } return success();}}
  return failure();}
```

Writing the Pass (Core Logic)

Algorithm 1 Partial QWC Match and Rewrite Logic

Require: currOp = NamedObsOp and currOp.type $\notin \{Z, I\}$

1: inQubit $\leftarrow$ currOp.inQubit
2: parentOp $\leftarrow$ `definingOp`(inQubit)
3: **if** parentOp $\in \{\text{rotY}(\frac{-\pi}{2}), \text{rotX}(\frac{\pi}{2})\}$ **then**
4: | currOp.type $\leftarrow$ Z ▷ *already correctly rotated*
5: **else** ▷ *rotate into computational basis*
6: | rotOp $\leftarrow$ currOp.type = X ? $\text{rotY}(\frac{-\pi}{2})$: $\text{rotX}(\frac{\pi}{2})$
7: | `Insert`(rotOp)
8: | newOp $\leftarrow$ `Create`(NamedObsOp, rotOp.result, Z)
9: | `ReplaceWithNewOp`(currOp,newOp)

```
LogicalResult matchAndRewrite(NamedObsOp op,
    PatternRewriter& rewriter) const override {
  using NamedObsOp = _NOO_;
  auto namedObs = op.getTypeAttr().getValue();
  if (namedObs != NamedObservable::PauliZ
    && namedObs != NamedObservable::Identity) {
      auto arg = op.getQubit();
      auto defOp = arg.getdefOp();
      if (isa<CustomOp>(defOp)) {
        auto neg_pi_div2 = APFloat(-1.570796326794896);
        auto pos_pi_div2 = APFloat(1.5707963267948966);
          auto parentOp = cast<CustomOp>(defOp);
          auto gateName = parentOp.getGateName();
          if (gateName.equals("RX") ||
            gateName.equals("RY")) {
              auto rotAngleOp =
              parentOp.getParams()[0].getdefOp();
              if (isa<arith::ConstantFloatOp>
                  (rotAngleOp)) {
                auto arithConstFloatOp =
                  cast<arith::ConstantFloatOp>
                    (rotAngleOp);
                  auto rotAngle =
                    arithConstFloatOp.value();
                  if (namedObs ==
                    NamedObservable::PauliX &&
                    gateName.equals("RY") &&
                    rotAngle.compare(neg_pi_div2))
                  {
                    rewriter.replaceOpWithNewOp
                      <_NOO_>(op,
                      op.getResult().getType(),
                      op.getQubit(),
                      NamedObservable::PauliZ);
                    return success();
                  } else if (namedObs ==
                    NamedObservable::PauliY &&
                    gateName.equals("RX") &&
                    rotAngle.compare(pos_pi_div2))
                    {
                      rewriter.replaceOpWithNewOp
                      <_NOO_>(op,
                        op.getResult().getType(),
                        op.getQubit(),
                        NamedObservable::PauliZ);
                      return success();}}}
```

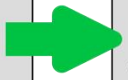
```
    else {
      Location loc = op.getLoc();
      auto rotAxis = namedObs
        == NamedObservable::PauliX ?
        StringRef{"RY"} : StringRef{"RX"};
        auto rotAngle = namedObs ==
          NamedObservable::PauliX ? neg_pi_div2 :
            pos_pi_div2;
        auto rotAngleVal =
          rewriter.create
            <arith::ConstantFloatOp>(
            loc, rotAngle,
            FloatType::getF64(
              rewriter.getContext()));
        auto resultTy = arg.getType();
        auto rotOp = rewriter.create<CustomOp>(
          loc, TypeRange{resultTy}, TypeRange{},
          ValueRange{rotAngleVal}, ValueRange{arg},
          rotAxis, UnitAttr{}, ValueRange{},
          ValueRange{});
        auto finalTy = op.getResult().getType();
        rewriter.replaceOpWithNewOp<_NOO_>(
          op, finalTy,
          rotOp.getQubitResults().front(),
          NamedObservable::PauliZ);
    } return success();}}
return failure();}
```

Catalyst Files Changed/Created

⊡	**frontend/catalyst/compiler.py** frontend/catalyst/compiler.py	+1 -0
⊡	**mlir/Makefile** mlir/Makefile	+8 -4
⊡	**mlir/include/Quantum/Transforms/Passes.h** mlir/include/Quantum/Transforms/Passes.h	+1 -0
⊡	**mlir/include/Quantum/Transforms/Passes.td** mlir/include/Quantum/Transforms/Passes.td	+6 -0
⊡	**mlir/include/Quantum/Transforms/Patterns.h** mlir/include/Quantum/Transforms/Patterns.h	+1 -0
⊡	**mlir/lib/Catalyst/Transforms/RegisterAllPasses.cpp** mlir/lib/Catalyst/Transforms/RegisterAllPasses.cpp	+1 -0
⊡	**mlir/lib/Quantum/Transforms/CMakeLists.txt** mlir/lib/Quantum/Transforms/CMakeLists.txt	+2 -0
⊞	**mlir/lib/Quantum/Transforms/PauliToZ.cpp** mlir/lib/Quantum/Transforms/PauliToZ.cpp	+64 -0
⊞	**mlir/lib/Quantum/Transforms/PauliToZPatterns.cpp** mlir/lib/Quantum/Transforms/PauliToZPatterns.cpp	+92 -0

Before and After Example

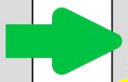
```
%o_qbits_55:2 = quantum.custom "CNOT"()
  %o_qbits_45, %o_qbits_54#0 :
  !quantum.bit, !quantum.bit
%o_qbits_56:2 = quantum.custom "CNOT"()
  %o_qbits_31, %o_qbits_55#0 :
  !quantum.bit, !quantum.bit
%o_qbits_57:2 = quantum.custom "CNOT"()
  %o_qbits_54#1, %o_qbits_56#0 :
  !quantum.bit, !quantum.bit
%79 = quantum.namedobs
  %o_qbits_55#1[ PauliY] : !quantum.obs
%80 = quantum.namedobs
  %o_qbits_56#1[ PauliY] : !quantum.obs
%81 = quantum.namedobs
  %o_qbits_57#1[ PauliX] : !quantum.obs
%82 = quantum.namedobs
  %o_qbits_57#0[ PauliX] : !quantum.obs
```



```
%npi2 = arith.constant 1.57079632 : f64
%ppi2 = arith.constant -1.5707794 : f64
%o_qbits_56:2 = quantum.custom "CNOT"()
  %o_qbits_46, %o_qbits_55#0 :
  !quantum.bit, !quantum.bit
%o_qbits_57:2 = quantum.custom "CNOT"()
  %o_qbits_32, %o_qbits_56#0 :
  !quantum.bit, !quantum.bit
%o_qbits_58:2 = quantum.custom "CNOT"()
  %o_qbits_55#1, %o_qbits_57#0 :
  !quantum.bit, !quantum.bit
%o_qbits_59 = quantum.custom "RX"(%ppi2)
  %o_qbits_56#1 : !quantum.bit
%79 = quantum.namedobs
  %o_qbits_59[ PauliZ] : !quantum.obs
%o_qbits_60 = quantum.custom "RX"(%ppi2)
  %o_qbits_57#1 : !quantum.bit
%80 = quantum.namedobs
  %o_qbits_60[ PauliZ] : !quantum.obs
%o_qbits_61 = quantum.custom "RY"(%npi2)
  %o_qbits_58#1 : !quantum.bit
%81 = quantum.namedobs
  %o_qbits_61[ PauliZ] : !quantum.obs
%o_qbits_62 = quantum.custom "RY"(%npi2)
  %o_qbits_58#0 : !quantum.bit
%82 = quantum.namedobs
  %o_qbits_62[ PauliZ] : !quantum.obs
```

Before and After Example

```
%o_qbits_55:2 = quantum.custom "CNOT"()
  %o_qbits_45, %o_qbits_54#0 :
  !quantum.bit, !quantum.bit
%o_qbits_56:2 = quantum.custom "CNOT"()
  %o_qbits_31, %o_qbits_55#0 :
  !quantum.bit, !quantum.bit
%o_qbits_57:2 = quantum.custom "CNOT"()
  %o_qbits_54#1, %o_qbits_56#0 :
  !quantum.bit, !quantum.bit
%79 = quantum.namedobs
  %o_qbits_55#1[ PauliY] : !quantum.obs
%80 = quantum.namedobs
  %o_qbits_56#1[ PauliY] : !quantum.obs
%81 = quantum.namedobs
  %o_qbits_57#1[ PauliX] : !quantum.obs
%82 = quantum.namedobs
  %o_qbits_57#0[ PauliX] : !quantum.obs
```



```
%npi2 = arith.constant 1.57079632 : f64
%ppi2 = arith.constant -1.5707794 : f64
%o_qbits_56:2 = quantum.custom "CNOT"()
  %o_qbits_46, %o_qbits_55#0 :
  !quantum.bit, !quantum.bit
%o_qbits_57:2 = quantum.custom "CNOT"()
  %o_qbits_32, %o_qbits_56#0 :
  !quantum.bit, !quantum.bit
%o_qbits_58:2 = quantum.custom "CNOT"()
  %o_qbits_55#1, %o_qbits_57#0 :
  !quantum.bit, !quantum.bit
%o_qbits_59 = quantum.custom "RX"(%ppi2)
  %o_qbits_56#1 : !quantum.bit
%79 = quantum.namedobs
  %o_qbits_59[ PauliZ] : !quantum.obs
%o_qbits_60 = quantum.custom "RX"(%ppi2)
  %o_qbits_57#1 : !quantum.bit
%80 = quantum.namedobs
  %o_qbits_60[ PauliZ] : !quantum.obs
%o_qbits_61 = quantum.custom "RY"(%npi2)
  %o_qbits_58#1 : !quantum.bit
%81 = quantum.namedobs
  %o_qbits_61[ PauliZ] : !quantum.obs
%o_qbits_62 = quantum.custom "RY"(%npi2)
  %o_qbits_58#0 : !quantum.bit
%82 = quantum.namedobs
  %o_qbits_62[ PauliZ] : !quantum.obs
```

Results on Synthetic and Real Molecules

Molecule	Measurements	After QWC	Reduction Factor
Synthetic_1	7	2	3.5×
Synthetic_2	8	3	2.67×
H_2	15	5	3×
HeH^+	27	9	3×
H_3^+	66	27	2.44×
He_2	181	63	2.87×
HF	631	151	4.18×
H_2O	1086	320	3.39×

TABLE I

THE NUMBER OF MEASUREMENTS FOR EACH HAMILTONIAN BEFORE AND AFTER THE QWC PASS.

Conclusion

- Intermediate representations lower the semantic gap between application and quantum backend target instructions
- Working at the intermediate representation level allows you to embed quantum domain knowledge into a compiler, allowing for quantum circuits from multiple inputs to be lowered to a common IR and then operated on by the same compiler passes
- It's not as daunting as it seems!
- Future Work
  - Make tree out of tree pass
  - More quantum domain compiler passes!

Thanks!

cabreraam@ornl.gov

Backups

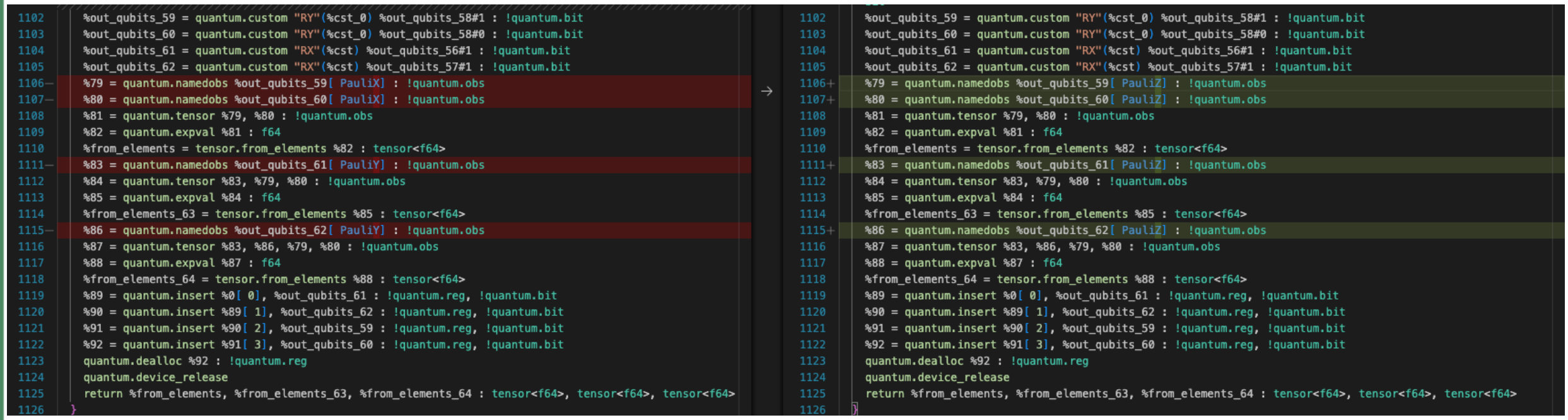
Step-through Python Execution
Useful VSCode Settings

- For Jupyter Notebook step-through
  - In settings.json: `"jupyter.debugJustMyCode": false`
- In a launch.json config for Python
  - ```
    {
      ...,
      "configurations": [
            {
                  "name": "Python: Current File",
                  "type": "python",
                  "request": "launch",
                  "program": "${file}",
                  "console": "integratedTerminal",
                  "justMyCode": false
            }
      ]
    }
    ```
    ```

Step through Python Execution

Demo

Before and After



```
%out_qubits_59 = quantum.custom "RY"(%cst_0) %out_qubits_58#1 : !quantum.bit
%out_qubits_60 = quantum.custom "RY"(%cst_0) %out_qubits_58#0 : !quantum.bit
%out_qubits_61 = quantum.custom "RX"(%cst) %out_qubits_56#1 : !quantum.bit
%out_qubits_62 = quantum.custom "RX"(%cst) %out_qubits_57#1 : !quantum.bit
%79 = quantum.namedobs %out_qubits_59[ PauliX] : !quantum.obs
%80 = quantum.namedobs %out_qubits_60[ PauliX] : !quantum.obs
%81 = quantum.tensor %79, %80 : !quantum.obs
%82 = quantum.expval %81 : f64
%from_elements = tensor.from_elements %82 : tensor<f64>
%83 = quantum.namedobs %out_qubits_61[ PauliY] : !quantum.obs
%84 = quantum.tensor %83, %79, %80 : !quantum.obs
%85 = quantum.expval %84 : f64
%from_elements_63 = tensor.from_elements %85 : tensor<f64>
%86 = quantum.namedobs %out_qubits_62[ PauliY] : !quantum.obs
%87 = quantum.tensor %83, %86, %79, %80 : !quantum.obs
%88 = quantum.expval %87 : f64
%from_elements_64 = tensor.from_elements %88 : tensor<f64>
%89 = quantum.insert %0[ 0], %out_qubits_61 : !quantum.reg, !quantum.bit
%90 = quantum.insert %89[ 1], %out_qubits_62 : !quantum.reg, !quantum.bit
%91 = quantum.insert %90[ 2], %out_qubits_59 : !quantum.reg, !quantum.bit
%92 = quantum.insert %91[ 3], %out_qubits_60 : !quantum.reg, !quantum.bit
quantum.dealloc %92 : !quantum.reg
quantum.device_release
return %from_elements, %from_elements_63, %from_elements_64 : tensor<f64>, tensor<f64>, tensor<f64>
}
```

```
%out_qubits_59 = quantum.custom "RY"(%cst_0) %out_qubits_58#1 : !quantum.bit
%out_qubits_60 = quantum.custom "RY"(%cst_0) %out_qubits_58#0 : !quantum.bit
%out_qubits_61 = quantum.custom "RX"(%cst) %out_qubits_56#1 : !quantum.bit
%out_qubits_62 = quantum.custom "RX"(%cst) %out_qubits_57#1 : !quantum.bit
%79 = quantum.namedobs %out_qubits_59[ PauliZ] : !quantum.obs
%80 = quantum.namedobs %out_qubits_60[ PauliZ] : !quantum.obs
%81 = quantum.tensor %79, %80 : !quantum.obs
%82 = quantum.expval %81 : f64
%from_elements = tensor.from_elements %82 : tensor<f64>
%83 = quantum.namedobs %out_qubits_61[ PauliZ] : !quantum.obs
%84 = quantum.tensor %83, %79, %80 : !quantum.obs
%85 = quantum.expval %84 : f64
%from_elements_63 = tensor.from_elements %85 : tensor<f64>
%86 = quantum.namedobs %out_qubits_62[ PauliZ] : !quantum.obs
%87 = quantum.tensor %83, %86, %79, %80 : !quantum.obs
%88 = quantum.expval %87 : f64
%from_elements_64 = tensor.from_elements %88 : tensor<f64>
%89 = quantum.insert %0[ 0], %out_qubits_61 : !quantum.reg, !quantum.bit
%90 = quantum.insert %89[ 1], %out_qubits_62 : !quantum.reg, !quantum.bit
%91 = quantum.insert %90[ 2], %out_qubits_59 : !quantum.reg, !quantum.bit
%92 = quantum.insert %91[ 3], %out_qubits_60 : !quantum.reg, !quantum.bit
quantum.dealloc %92 : !quantum.reg
quantum.device_release
return %from_elements, %from_elements_63, %from_elements_64 : tensor<f64>, tensor<f64>, tensor<f64>
}
```

Check for Relevant Documentation

Catalyst Compiler Passes

What does Catalyst's IR look like?

```python
def circuit(x: complex):
    qml.Hadamard(wires=0)

    A = np.array([1, 0], [0, np.exp(x)])
    qml.QubitUnitary(A, wires=0)

    B = np.array([1, 0], [0, np.exp(2*x)])
    qml.QubitUnitary(B, wires=0)

    return measure(0)
```

```
func.func @circuit(%arg0: complex<f64>) -> i1 {
    %c00 = complex.constant [0.0, 0.0] : complex<f64>
    %c10 = complex.constant [1.0, 0.0] : complex<f64>
    %c20 = complex.constant [2.0, 0.0] : complex<f64>

    %reg = quantum.alloc(1) : !quantum.reg
    %q0 = quantum.extract %reg[0] : !quantum.bit

    %q1 = quantum.custom "Hadamard"() %q0 : !quantum.bit

    %0 = complex.exp %arg0 : complex<f64>
    %A = tensor.from_elements %c10, %c00, %c00, %1 : tensor<2x2xcomplex<f64>>
    %q2 = quantum.unitary %A, %q1 : !quantum.bit

    %1 = complex.mul %arg0, %c20 : complex<f64>
    %2 = complex.exp %0 : complex<f64>
    %B = tensor.from_elements %c10, %c00, %c00, %2 : tensor<2x2xcomplex<f64>>
    %q3 = quantum.unitary %B, %q2 : !quantum.bit

    %m = quantum.measure %q3 : i1

    quantum.dealloc %r : !quantum.reg
    func.return %m : i1
}
```

Check for Relevant Documentation

Catalyst Compiler Passes

Writing transformations on Catalyst's IR

```
LogicalResult QubitUnitaryFusion::match(QubitUnitaryOp op)
{
    ValueRange qbs = op.getInQubits();
    Operation *parent = qbs[0].getDefiningOp();

    if (!isa<QubitUnitaryOp>(parent))
        return failure();

    QubitUnitaryOp parentOp = cast<QubitUnitaryOp>(parent);
    ValueRange parentQbs = parentOp.getOutQubits();

    if (qbs.size() != parentQbs.size())
        return failure();

    for (auto [qb1, qb2] : llvm::zip(qbs, parentQbs))
        if (qb1 != qb2)
            return failure();

    return success();
}
```

```
void QubitUnitaryFusion::rewrite(QubitUnitaryOp op, PatternRewriter &rewriter)
{
    ValueRange qbs = op.getInQubits();
    QubitUnitaryOp parentOp = cast<QubitUnitaryOp>(qbs[0].getDefiningOp());

    Value m1 = op.getMatrix();
    Value m2 = parentOp.getMatrix();

    linalg::MatmulOp matmul = rewriter.create<linalg::MatmulOp>(op.getLoc(), {m1, m2}, {});
    Value res = matmul.getResult(0);

    rewriter.updateRootInPlace(op, [&] {
        op->setOperand(0, res);
    });
    rewriter.replaceOp(parentOp, parentOp.getResults());
}
```