

# 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

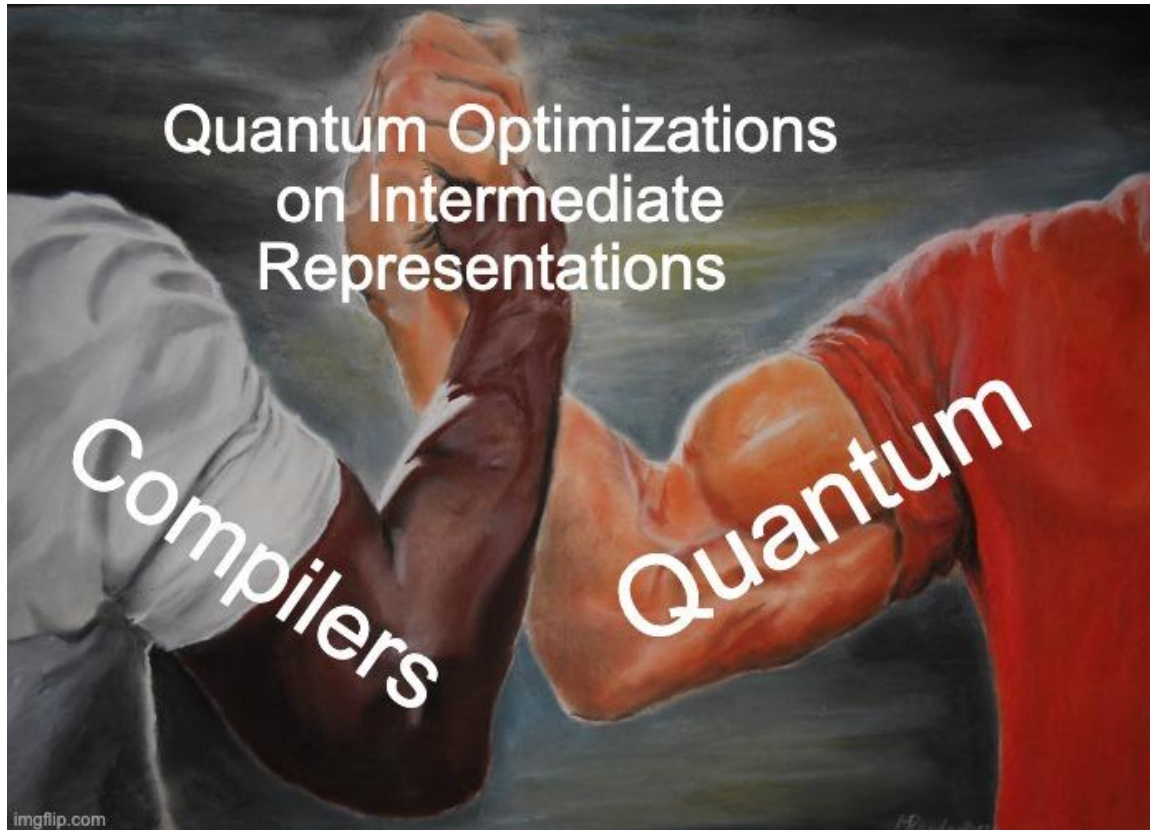
\*Research Scientist @ ORNL, [cabreraam@ornl.gov](mailto:cabreraam@ornl.gov)

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



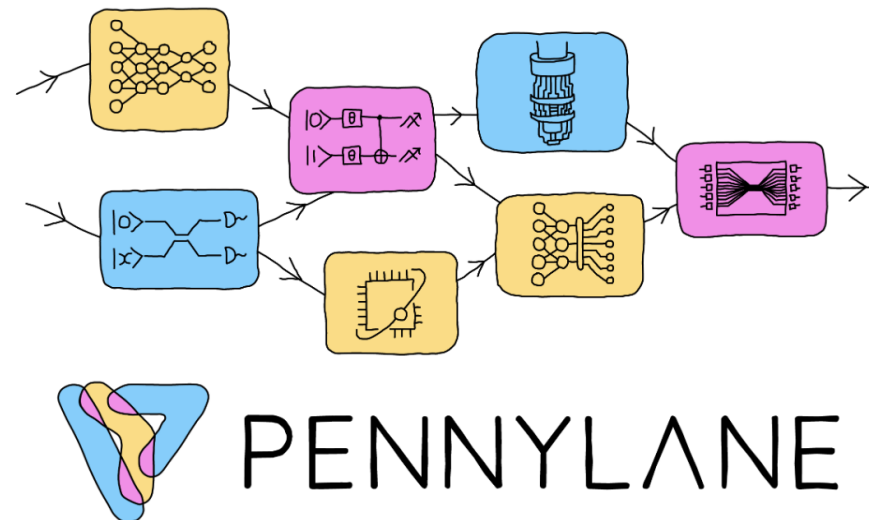
- A framework that provides the capability of authoring domain-specific intermediate representations
  - This allows us to reason about quantum at a higher abstraction level – MLIR Dialects – instead of quantum gates – e.g., QIR, QASM
- Relies on “progressive lowering”
  - Starting from a level of abstraction 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.



# 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 a 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...

```
0_circuit_grouped_ins.mlir 1, U 1_0_canonicalize.mlir 1, U X 1_HLOLoweringPass.mlir 1, U 2_Quantum ↻ 🔍
qwc > circuit_grouped_ins_2 > 1_0_canonicalize.mlir
1 module @circuit_grouped_ins {
6   func.func private @circuit_grouped_ins(%arg0: tensor<3> > quantum.namedobs Aa ab .* ? of 9 ↑ ↓ ≡
261   %out_qubits_61 = quantum.custom "RX"(%cst) %out_qubits_56#1 : !quantum.bit
262   %out_qubits_62 = quantum.custom "RX"(%cst) %out_qubits_57#1 : !quantum.bit
263   %189 = quantum.namedobs %out_qubits_59[ PauliX] : !quantum.obs
264   %190 = quantum.namedobs %out_qubits_60[ PauliX] : !quantum.obs
265   %191 = quantum.tensor %189, %190 : !quantum.obs
266   %192 = quantum.expval %191 : f64
267   %from_elements = tensor.from_elements %192 : tensor<f64>
268   %193 = quantum.namedobs %out_qubits_61[ PauliY] : !quantum.obs
269   %194 = quantum.namedobs %out_qubits_59[ PauliX] : !quantum.obs
270   %195 = quantum.namedobs %out_qubits_60[ PauliX] : !quantum.obs
271   %196 = quantum.tensor %193, %194, %195 : !quantum.obs
272   %197 = quantum.expval %196 : f64
273   %from_elements_63 = tensor.from_elements %197 : tensor<f64>
274   %198 = quantum.namedobs %out_qubits_61[ PauliY] : !quantum.obs
275   %199 = quantum.namedobs %out_qubits_62[ PauliY] : !quantum.obs
276   %200 = quantum.namedobs %out_qubits_59[ PauliX] : !quantum.obs
277   %201 = quantum.namedobs %out_qubits_60[ PauliX] : !quantum.obs
278   %202 = quantum.tensor %198, %199, %200, %201 : !quantum.obs
279   %203 = quantum.expval %202 : f64
280   %from_elements_64 = tensor.from_elements %203 : tensor<f64>
281   %204 = quantum.insert %4[ 0], %out_qubits_61 : !quantum.reg, !quantum.bit
282   %205 = quantum.insert %204[ 1], %out_qubits_62 : !quantum.reg, !quantum.bit
283   %206 = quantum.insert %205[ 2], %out_qubits_59 : !quantum.reg, !quantum.bit
284   %207 = quantum.insert %206[ 3], %out_qubits_60 : !quantum.reg, !quantum.bit
285   quantum.dealloc %207 : !quantum.reg
286   quantum.device_release
287   return %from_elements, %from_elements_63, %from_elements_64 : tensor<f64>, tensor<f64>, tensor<f64>
288 }
289 func.func @setup() {
290   quantum.init
291   return
292 }
293 func.func @teardown() {
294   quantum.finalize
295   return
```

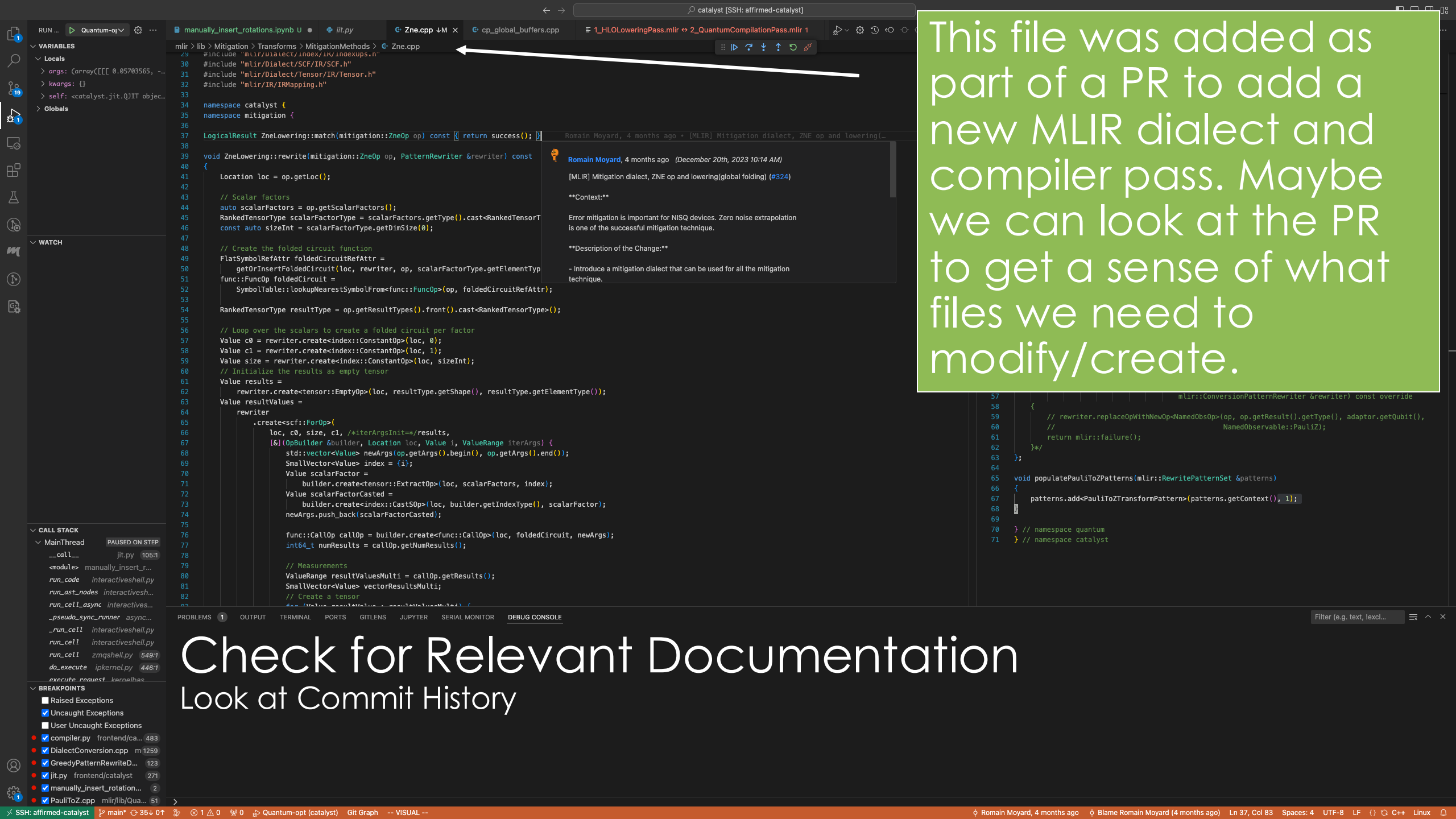
# What We Actually Did

- 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





# [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

doc/changelog.md

@@ -2,6 +2,10 @@

2

3

4

<h3>New features</h3>

5

6

7

8

+ \* 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>)

9

10

11

\* 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``,

mlir/include/CAPI/Dialects.h

@@ -22,6 +22,7 @@ extern "C" {

22

23

24

25

26

27

MLIR\_DECLARE\_CAPI\_DIALECT\_REGISTRATION(Quantum, quantum);

MLIR\_DECLARE\_CAPI\_DIALECT\_REGISTRATION(Gradient, gradient);

MLIR\_DECLARE\_CAPI\_DIALECT\_REGISTRATION(Catalyst, catalyst);

#ifdef \_\_cplusplus

22

23

24

25

26

27

28

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

mlir/include/CMakeLists.txt

@@ -1,4 +1,5 @@

1

2

3

4

add\_subdirectory(Catalyst)

add\_subdirectory(Quantum)

add\_subdirectory(Gradient)

add\_subdirectory(Test)

1

2

3

4

5

add\_subdirectory(Catalyst)

add\_subdirectory(Quantum)

add\_subdirectory(Gradient)

+ add\_subdirectory(Mitigation)

add\_subdirectory(Test)

mlir/include/Mitigation/CMakeLists.txt

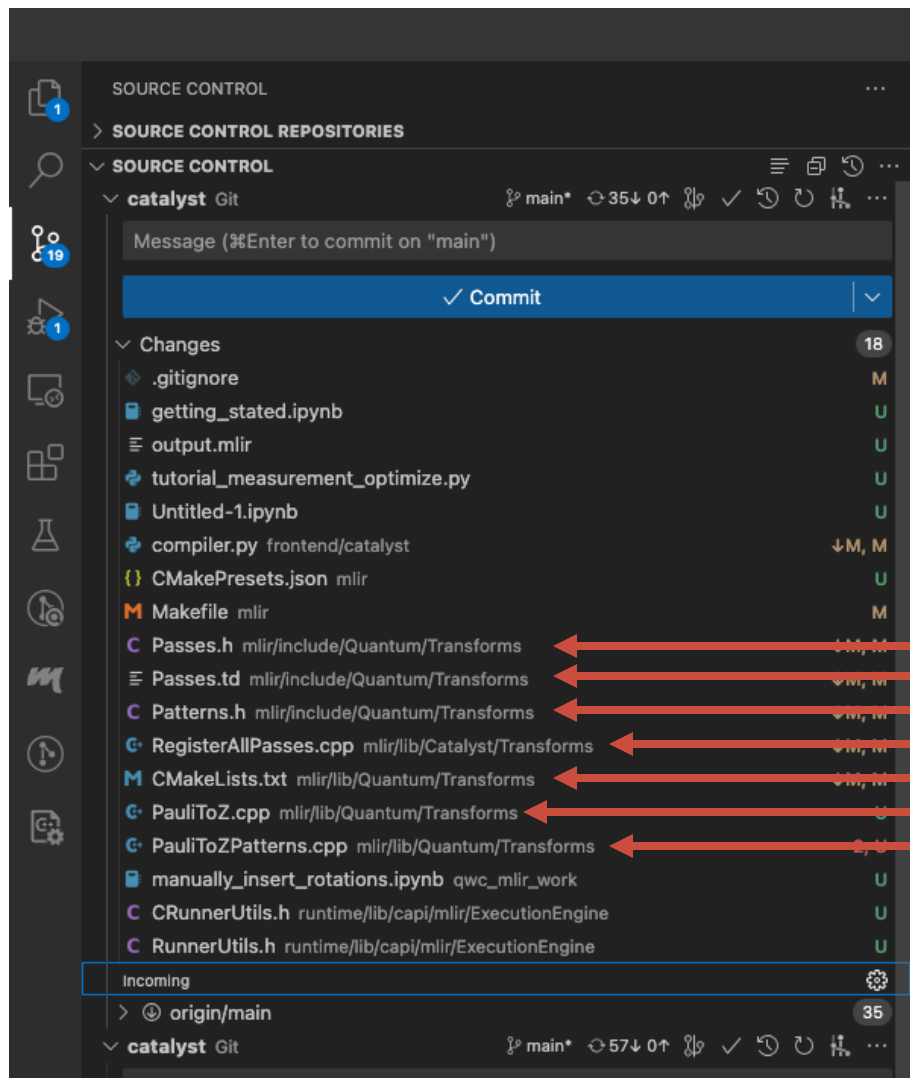
@@ -0,0 +1,2 @@

1

add\_subdirectory(IR)



# 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`



# 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 ? rotY( $\frac{-\pi}{2}$ ) : 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{-1.5707963267948966};
            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 ? rotY( $\frac{-\pi}{2}$ ) : 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 ? rotY( $\frac{-\pi}{2}$ ) : 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{-1.5707963267948966};
            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 ? rotY( $\frac{-\pi}{2}$ ) : 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{-1.5707963267948966};
            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 ? rotY( $\frac{-\pi}{2}$ ) : 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::PauliX)
        && namedObs != NamedObservable::Identity) {
        auto arg = op.getQubit();
        auto defOp = arg.getDefOp();
        if (isa<CustomOp>(defOp)) {
            auto neg_pi_div2 = APFloat{-1.5707963267948966};
            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();
}
```

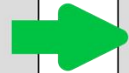
# Catalyst Files Changed/Created

▣	<b>frontend/catalyst/compiler.py</b> frontend/catalyst/compiler.py	+1 -0
▣	<b>mlir/Makefile</b> mlir/Makefile	+8 -4
▣	<b>mlir/include/Quantum/Transforms/Passes.h</b> mlir/include/Quantum/Transforms/Passes.h	+1 -0
▣	<b>mlir/include/Quantum/Transforms/Passes.td</b> mlir/include/Quantum/Transforms/Passes.td	+6 -0
▣	<b>mlir/include/Quantum/Transforms/Patterns.h</b> mlir/include/Quantum/Transforms/Patterns.h	+1 -0
▣	<b>mlir/lib/Catalyst/Transforms/RegisterAllPasses.cpp</b> mlir/lib/Catalyst/Transforms/RegisterAllPasses.cpp	+1 -0
▣	<b>mlir/lib/Quantum/Transforms/CMakeLists.txt</b> mlir/lib/Quantum/Transforms/CMakeLists.txt	+2 -0
+	<b>mlir/lib/Quantum/Transforms/PauliToZ.cpp</b> mlir/lib/Quantum/Transforms/PauliToZ.cpp	+64 -0
+	<b>mlir/lib/Quantum/Transforms/PauliToZPatterns.cpp</b> 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 <sub>1</sub>	7	2	3.5×
Synthetic <sub>2</sub>	8	3	2.67×
H <sub>2</sub>	15	5	3×
HeH <sup>+</sup>	27	9	3×
H <sub>3</sub> <sup>+</sup>	66	27	2.44×
He <sub>2</sub>	181	63	2.87×
HF	631	151	4.18×
H <sub>2</sub> O	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](mailto: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  
        }  
    ]  
}
```



Quantum-opt

manually\_insert\_rotations.ipynb

jit.py

cp\_global\_buffers.cpp

1\_HLOLoweringPass.mlir

2\_QuantumCompil

PauliToZ.cpp

PauliToZPatterns.cpp

APFloat.h

gradient\_to\_llvm.cpp

Arith.h

ArithOps.h.inc

QuantumAttributes.cf

VARIABLES

Locals

Globals

WATCH

CALL STACK

MainThread

\_\_call\_\_

\_\_module\_\_

run\_code

run\_ast\_nodes

run\_cell\_async

\_pseudo\_sync\_runner

\_run\_cell

run\_cell

run\_cell

do\_execute

execute\_request\_kernelhas

BREAKPOINTS

Raised Exceptions

Uncaught Exceptions

User Uncaught Exceptions

compiler.py

DialectConversion.cpp

GreedyPatternRewriteD...

jit.py

manually\_insert\_rotat...

PauliToZ.cpp

frontend > catalyst > jit.py > QJIT > \_\_call\_\_

class QJIT:

def \_\_init\_\_(self, fn, compile\_options):

compile\_options.static\_argnums, compile\_options.abstracted\_axes

# Active state of the compiler.

# TODO0: rework ownership of workspace, possibly CompiledFunction

self.workspace = None

self.c\_sig = None

self.out\_treedef = None

self.compiled\_function = None

self.jaxed\_function = None

# IRs are only available for the most recently traced function.

self.jaxpr = None

self.mlir = None # string form (historic presence)

self.mlir\_module = None

self.qir = None

functools.update\_wrapper(self, fn)

self.user\_sig = get\_type\_annotations(fn)

self.validate\_configuration()

self.user\_function = self.pre\_compilation()

# Static arguments require values, so we cannot AOT compile.

if self.user\_sig is not None and not self.compile\_options.static\_argnums:

self.aot\_compile()

def \_\_call\_\_(self, \*args, \*\*kwargs):

# Transparently call Python function in case of nested QJIT calls.

if EvaluationContext.is\_tracing():

return self.user\_function(\*args, \*\*kwargs)

requires\_promotion = self.jit\_compile(args)

# If we receive tracers as input, dispatch to the JAX integration.

if any(isinstance(arg, jax.core.Tracer) for arg in tree\_flatten(args)[0]):

if self.jaxed\_function is None:

self.jaxed\_function = JAX\_QJIT(self) # lazy gradient compilation

return self.jaxed\_function(\*args, \*\*kwargs)

elif requires\_promotion:

dynamic\_args = filter\_static\_args(args, self.compile\_options.static\_argnums)

args = promote\_arguments(self.c\_sig, dynamic\_args)

return self.run(args, kwargs)

def aot\_compile(self):

"""Compile Python function on initialization using the type hint signature."""

self.workspace = self.\_get\_workspace()

# TODO0: awkward, refactor or redesign the target feature

if self.compile\_options.target in ("jaxpr", "mlir", "binary"):

self.jaxpr, self.out\_treedef, self.c\_sig = self.capture(self.user\_sig or ())

mlir > lib > Quantum > Transforms > PauliToZPatterns.cpp

namespace catalyst {

namespace quantum {

struct PauliToZTransformPattern : public mlir::OpRewritePattern<NamedObsOp> {

mlir::LogicalResult match(NamedObsOp op) const override

auto rotationAngleOp = rotationOp.getParams()[0].getDefiningOp();

if (mlir::isa<mlir::arith::ConstantFloatOp>(rotationAngleOp)) {

auto neg\_pi\_div2 = llvm::APFloat(-1.5707963267948966);

auto pos\_pi\_div2 = llvm::APFloat(1.5707963267948966);

auto arithConstFloatOp =

mlir::cast<mlir::arith::ConstantFloatOp>(rotationAngleOp);

auto rotationAngle = arithConstFloatOp.value();

if (namedObs == NamedObservable::PauliX && rotationAxis.equals("RY") &&

rotationAngle.compare(neg\_pi\_div2))

return mlir::success();

else if (namedObs == NamedObservable::PauliY && rotationAxis.equals("RX") &&

rotationAngle.compare(pos\_pi\_div2))

return mlir::success();

}

return mlir::failure();

}

void rewrite(NamedObsOp op, mlir::PatternRewriter &rewriter) const override

// The "rewrite" method performs mutations on the IR rooted at "op" using

// the provided rewriter. All mutations must go through the provided rewriter.

// LLVM\_DEBUG(llvm::dbgs() << "rewrite pauli-to-z\n");

rewriter.replaceOpWithNewOp<NamedObsOp>(op, op.getResult().getType(), op.getQubit(),

NamedObservable::PauliZ);

}/

mlir::LogicalResult matchAndRewrite(NamedObsOp op, NamedObsOpAdaptor adaptor,

mlir::ConversionPatternRewriter &rewriter) const override

{

// rewriter.replaceOpWithNewOp<NamedObsOp>(op, op.getResult().getType(), adaptor.getQubit(),

// NamedObservable::PauliZ);

return mlir::failure();

}/

}

void populatePauliToZPatterns(mlir::RewritePatternSet &patterns)

{

patterns.add<PauliToZTransformPattern>(patterns.getContext(), 1);

}

// namespace quantum

// namespace catalyst

PROBLEMS

OUTPUT

TERMINAL

PORTS

GITLINS

JUPYTER

SERIAL MONITOR

DEBUG CONSOLE

SSH: affirmed-catalyst

main\*

354 0 T

Quantum-opt (catalyst)

Git Graph

-- NORMAL --

David Ittah, 2 months ago

Blame Sergei Mironov (7 months ago)

Ln 105, Col 1

Spaces: 4

UTF-8

LF

Python

3.12.2 (quantum: conda)

Linux

Step through Python Execution Demo



# Before and After

```
1102 %out_qubits_59 = quantum.custom "RY"(%cst_0) %out_qubits_58#1 : !quantum.bit
1103 %out_qubits_60 = quantum.custom "RY"(%cst_0) %out_qubits_58#0 : !quantum.bit
1104 %out_qubits_61 = quantum.custom "RX"(%cst) %out_qubits_56#1 : !quantum.bit
1105 %out_qubits_62 = quantum.custom "RX"(%cst) %out_qubits_57#1 : !quantum.bit
1106- %79 = quantum.namedobs %out_qubits_59[ PauliX] : !quantum.obs
1107- %80 = quantum.namedobs %out_qubits_60[ PauliX] : !quantum.obs
1108 %81 = quantum.tensor %79, %80 : !quantum.obs
1109 %82 = quantum.expval %81 : f64
1110 %from_elements = tensor.from_elements %82 : tensor<f64>
1111- %83 = quantum.namedobs %out_qubits_61[ PauliY] : !quantum.obs
1112 %84 = quantum.tensor %83, %79, %80 : !quantum.obs
1113 %85 = quantum.expval %84 : f64
1114 %from_elements_63 = tensor.from_elements %85 : tensor<f64>
1115- %86 = quantum.namedobs %out_qubits_62[ PauliY] : !quantum.obs
1116 %87 = quantum.tensor %83, %86, %79, %80 : !quantum.obs
1117 %88 = quantum.expval %87 : f64
1118 %from_elements_64 = tensor.from_elements %88 : tensor<f64>
1119 %89 = quantum.insert %0[ 0], %out_qubits_61 : !quantum.reg, !quantum.bit
1120 %90 = quantum.insert %89[ 1], %out_qubits_62 : !quantum.reg, !quantum.bit
1121 %91 = quantum.insert %90[ 2], %out_qubits_59 : !quantum.reg, !quantum.bit
1122 %92 = quantum.insert %91[ 3], %out_qubits_60 : !quantum.reg, !quantum.bit
1123 quantum.dealloc %92 : !quantum.reg
1124 quantum.device_release
1125 return %from_elements, %from_elements_63, %from_elements_64 : tensor<f64>, tensor<f64>, tensor<f64>
1126 }
```

→

```
1102 %out_qubits_59 = quantum.custom "RY"(%cst_0) %out_qubits_58#1 : !quantum.bit
1103 %out_qubits_60 = quantum.custom "RY"(%cst_0) %out_qubits_58#0 : !quantum.bit
1104 %out_qubits_61 = quantum.custom "RX"(%cst) %out_qubits_56#1 : !quantum.bit
1105 %out_qubits_62 = quantum.custom "RX"(%cst) %out_qubits_57#1 : !quantum.bit
1106+ %79 = quantum.namedobs %out_qubits_59[ PauliZ] : !quantum.obs
1107+ %80 = quantum.namedobs %out_qubits_60[ PauliZ] : !quantum.obs
1108 %81 = quantum.tensor %79, %80 : !quantum.obs
1109 %82 = quantum.expval %81 : f64
1110 %from_elements = tensor.from_elements %82 : tensor<f64>
1111+ %83 = quantum.namedobs %out_qubits_61[ PauliZ] : !quantum.obs
1112 %84 = quantum.tensor %83, %79, %80 : !quantum.obs
1113 %85 = quantum.expval %84 : f64
1114 %from_elements_63 = tensor.from_elements %85 : tensor<f64>
1115+ %86 = quantum.namedobs %out_qubits_62[ PauliZ] : !quantum.obs
1116 %87 = quantum.tensor %83, %86, %79, %80 : !quantum.obs
1117 %88 = quantum.expval %87 : f64
1118 %from_elements_64 = tensor.from_elements %88 : tensor<f64>
1119 %89 = quantum.insert %0[ 0], %out_qubits_61 : !quantum.reg, !quantum.bit
1120 %90 = quantum.insert %89[ 1], %out_qubits_62 : !quantum.reg, !quantum.bit
1121 %91 = quantum.insert %90[ 2], %out_qubits_59 : !quantum.reg, !quantum.bit
1122 %92 = quantum.insert %91[ 3], %out_qubits_60 : !quantum.reg, !quantum.bit
1123 quantum.dealloc %92 : !quantum.reg
1124 quantum.device_release
1125 return %from_elements, %from_elements_63, %from_elements_64 : tensor<f64>, tensor<f64>, tensor<f64>
1126 }
```

# Check for Relevant Documentation

## Catalyst Compiler Passes

### What does Catalyst's IR look like?

```
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());
}
```