CATALYST

Accelerating large-scale dynamic quantum algorithms with just-in-time compilation

XANADU

"Build quantum computers that are **useful and accessible** to people everywhere"

PennyLane is a Python library for programming quantum computers.

Catalyst is a JIT compiler for PennyLane programs.

01

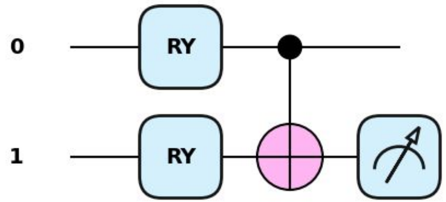
Towards a modern Quantum Compilation architecture

// How to program a quantum computer?

PennyLane is a Python library for programming quantum computers.

- Quantum **device** (hw or simulator)
- **Circuit** abstraction
  - Quantum **gates**
  - Quantum bits (**qubits**)
  - **Measurements**

```python
import pennylane as qml

dev = qml.device("default.qubit", wires=2)

@qml.qnode(dev)
def circuit(params):
    qml.RY(params[0], wires=0)
    qml.RY(params[1], wires=1)
    qml.CNOT(wires=[0, 1])
    return qml.expval(qml.Z(1))

params = [0.1, 0.3]
qml.draw_mpl(circuit)(params)
```



// PennyLane

Python user input

```python
import pennylane as qml
from catalyst import qjit

dev = qml.device("lightning.qubit",
wires=3)

@qml.qnode(device=dev)
def circuit(data):
    for i, d in enumerate(data):
        if i < 2:
            qml.RX(d, wires=i)
        else:
            qml.RY(d, wires=i)

    return qml.expval(qml.PauliZ(0))

data = jax.numpy.array([0.1, 0.2, 0.3])
circuit(data)
```

Python representation of the circuit

```
[RX(Array(0.1, dtype=float64), wires=[0]),
RX(Array(0.2, dtype=float64), wires=[1]),
RY(Array(0.3, dtype=float64), wires=[2]),
expval(Z(0))]
```

// What is quantum software today?

- Primarily Python-based software packages
- Programming at the level of **quantum circuits**
- **Execution** on **simulators (CPU/GPU)** and **hardware (QPU)**
- Execution on hardware involves:
  - **Optimizing** the circuit with **runtime parameters**
  - Serializing **just the quantum** component of the circuit via a human-readable intermediate representation
  - Submitting the circuit for execution via a **cloud REST API**

```python
import pennylane as qml
from catalyst import qjit

dev = qml.device("lightning.qubit",
wires=3)

@qml.qnode(device=dev)
def circuit(data):
    for i, d in enumerate(data):
        if i < 2:
            qml.RX(d, wires=i)
        else:
            qml.RY(d, wires=i)

    return qml.expval(qml.PauliZ(0))

data = jax.numpy.array([0.1, 0.2, 0.3])
circuit(data)
```

Catalyst is a JIT compiler for PennyLane programs.

```python
import pennylane as qml
from catalyst import qjit
import numpy as np

dev = qml.device("lightning.qubit", wires=2, shots=1000)

@qjit
@qml.qnode(dev)
def circuit(x, y, z):
    qml.RX(x, wires=[y + 1])
    qml.RY(x, wires=[z])
    qml.CNOT(wires=[y, z])
    return qml.probs(wires=[y + 1])

>>> circuit(np.pi / 3, 1, 2)
array([0.625, 0.375])
```

// Catalyst

Python user input

```python
import pennylane as qml
from catalyst import qjit

dev = qml.device("lightning.qubit", wires=3)

@qjit(keep_intermediate=True, autograph=True)
@qml.qnode(device=dev)
def circuit(data):
    for i, d in enumerate(data):
        if i < 2:
            qml.RX(d, wires=i)
        else:
            qml.RY(d, wires=i)

    return qml.expval(qml.PauliZ(0))

data = jax.numpy.array([0.1, 0.2, 0.3])

circuit(data)
```

Catalyst IR (MLIR with our own dialects)

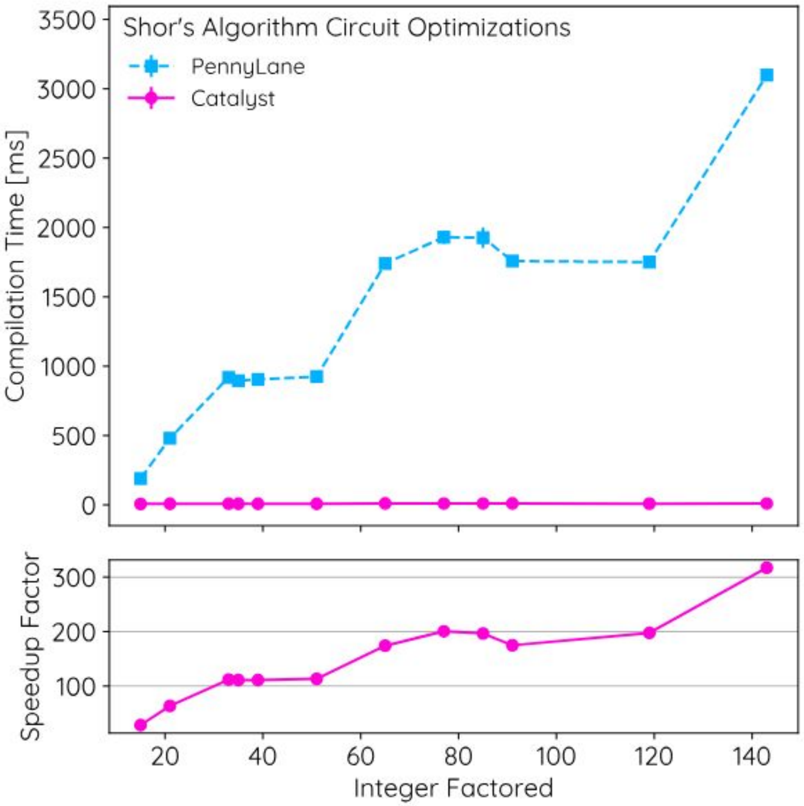
```
func.func public @circuit(%arg0: tensor<3xf64>) -> tensor<f64> attributes {diff_me
  quantum.device["/home/romain/Catalyst/catalyst/frontend/catalyst/utils/../../../
  ...
  %4 = scf.for %arg1 = %1 to %2 step %3 iter_args(%arg2 = %0) -> (!quantum.reg) {
    ...
    %17 = scf.if %extracted_12 -> (!quantum.reg) {
      ...
      %out_qubits = quantum.custom "RX"(%extracted_14) %18 : !quantum.bit
      ...
      scf.yield %19 : !quantum.reg
    } else {
      ...
      %out_qubits = quantum.custom "RY"(%extracted_14) %18 : !quantum.bit
      ...
      scf.yield %19 : !quantum.reg
    }
    scf.yield %17 : !quantum.reg
  }
  ...
  %7 = quantum.expval %6 : f64
  %from_elements = tensor.from_elements %7 : tensor<f64>
  quantum.dealloc %4 : !quantum.reg
  quantum.device_release
  return %from_elements : tensor<f64>
}
```

// PennyLane versus Catalyst?

- Static circuit (quantum only)
- The structure of the circuit is lost (for, while, cond)
- The circuit representation is recompiled for every different parameter
- Optimization is done at runtime (quantum only)

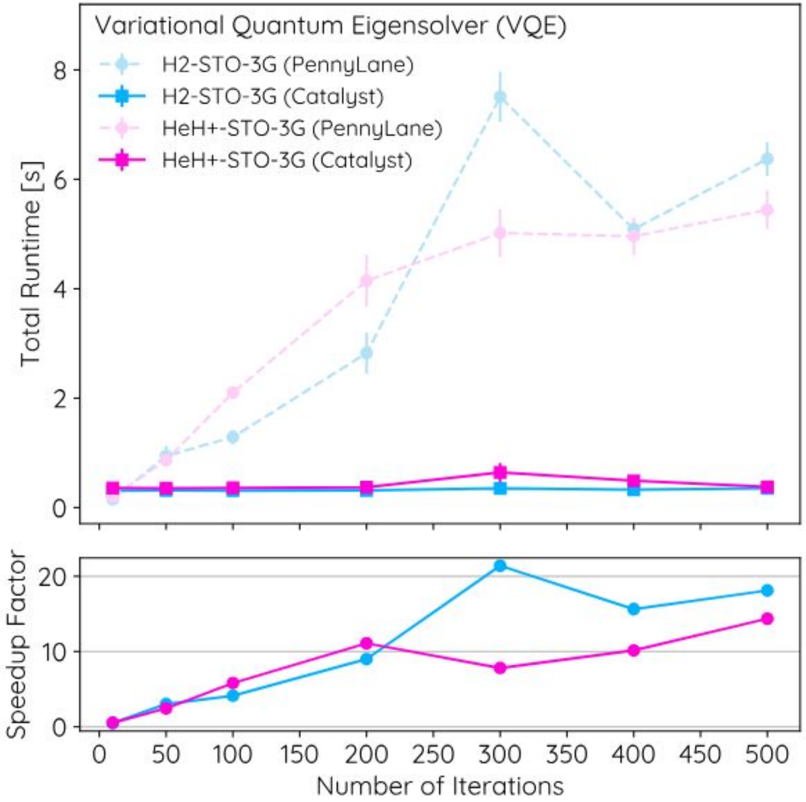
- Dynamic circuit (hybrid)
- The control flow is preserved
- The program is not recompiled when it does not need to.
- Optimization is done at compile time. (MLIR transformation passes)

// Compiling algorithms with structure



- Preservation of the control flow (for loop over qubits)
- Optimization at compile time on a compact IR.

// Parametric compilation (escaping Python speedup)

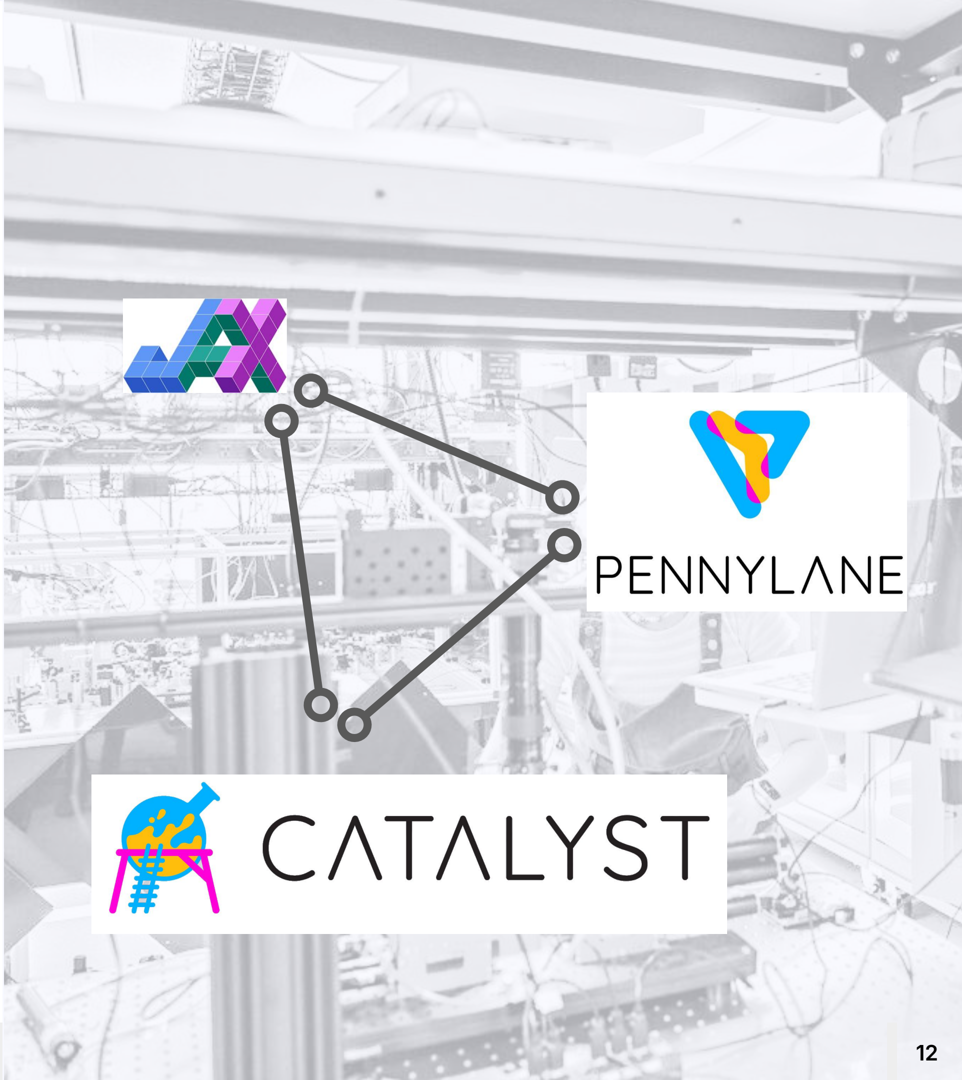


- The circuit is not recompiled because parameters are of the same type.
- VQE needs the same circuit to be executed for a lot of parameters.

02

Catalyst

Reimagining the quantum computing stack



// The Catalyst Stack

Frontend:

- PennyLane + Jax
- Dynamic programming model
- Python operator overloading
- Program capture

MLIR:

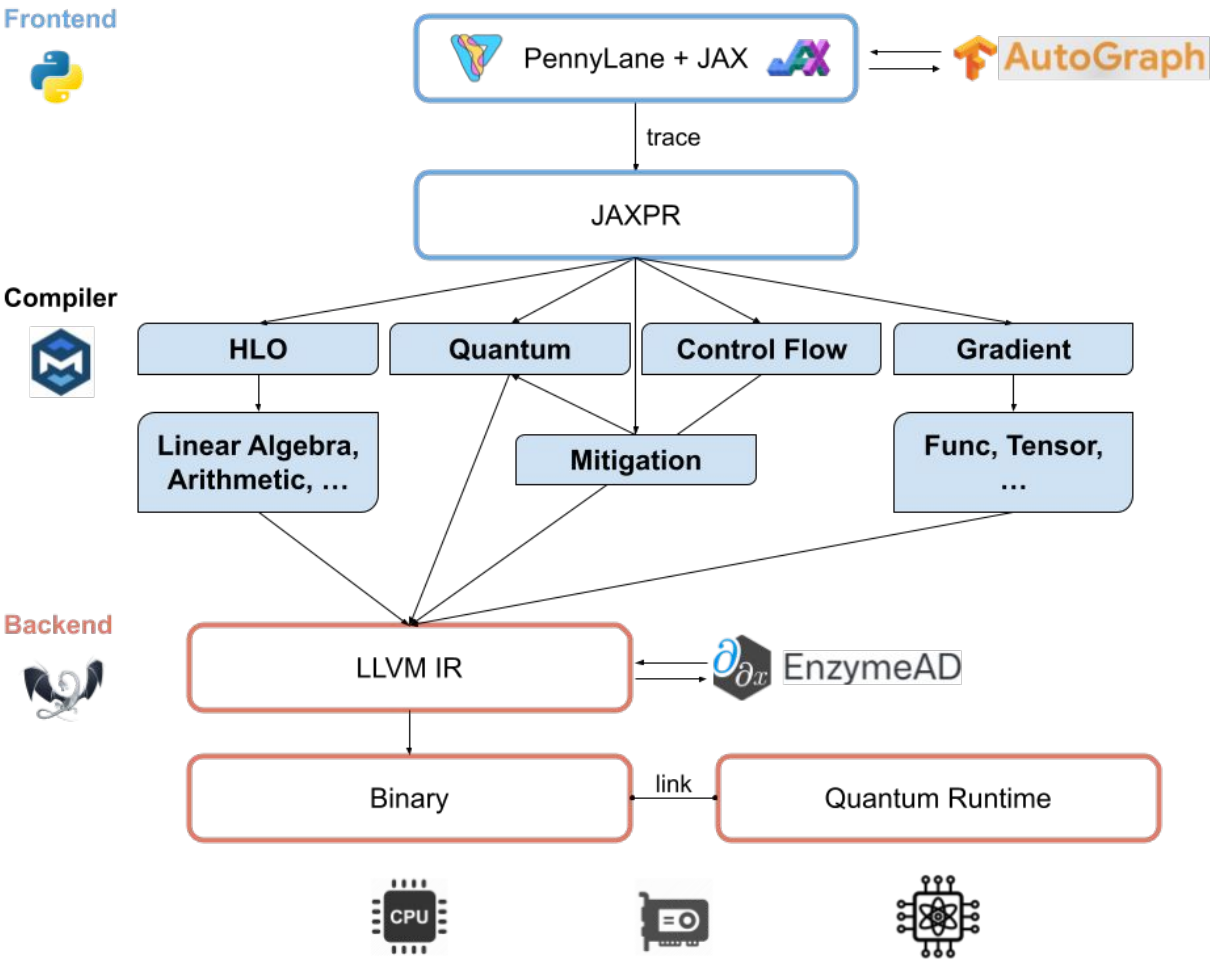
- Quantum autodiff
- Circuit optimizations
- Error mitigation

CodeGen:

- Leverage LLVM infrastructure
- Enzyme autodiff
- Binary code generation

Execution:

- Device-Host interactions
- Real-time classical processing
- Dynamic instruction dispatch
- Runtime circuit generation



// The Catalyst Stack

Frontend:

- PennyLane + Jax
- Dynamic programming model
- Python operator overloading
- Program capture

MLIR:

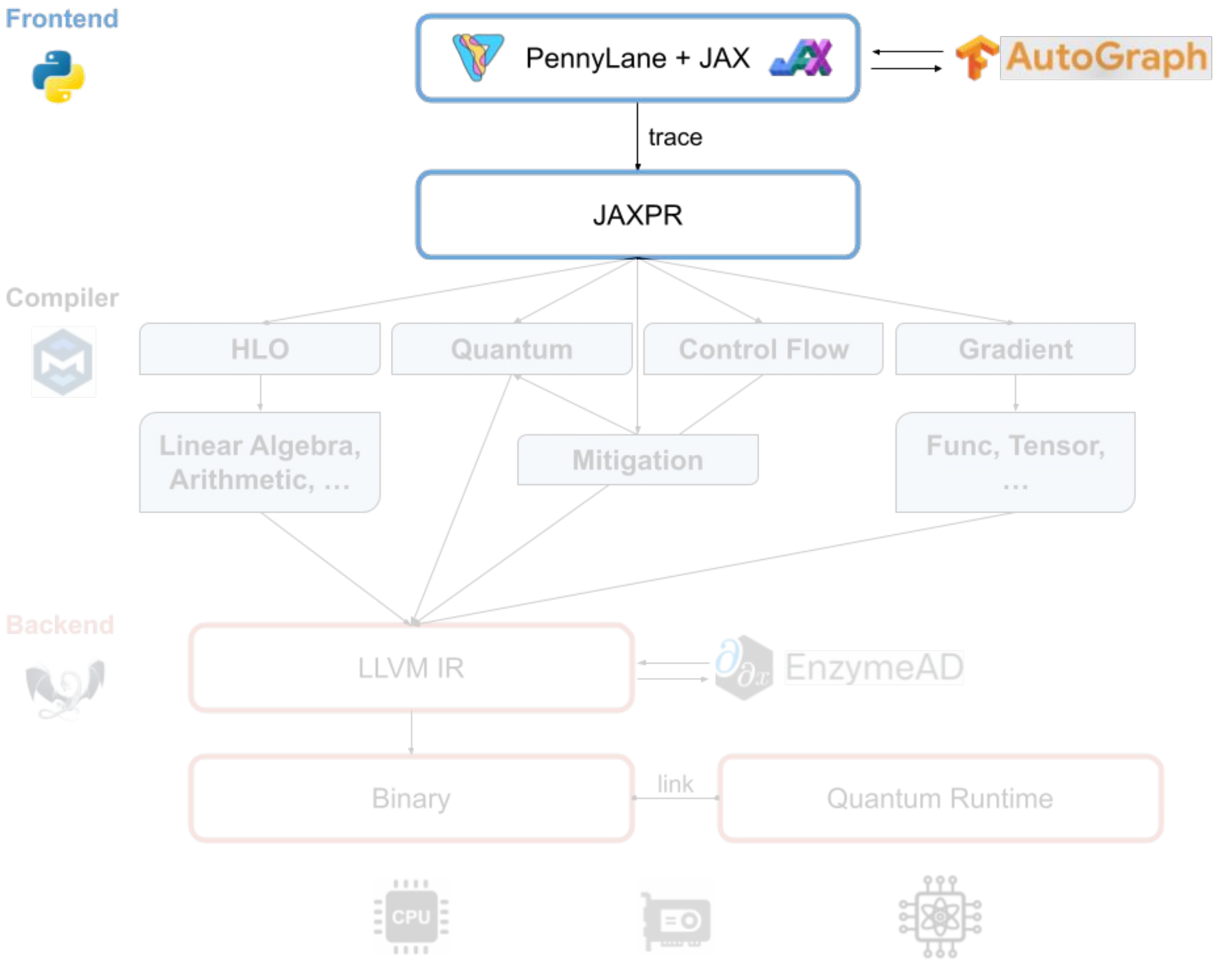
- Quantum autodiff
- Quantum circuit optimization
- Error mitigation

CodeGen:

- Leverage LLVM infrastructure
- Enzyme autodiff
- Binary code generation

Execution:

- Device-Host interactions
- Real-time classical processing
- Dynamic instruction dispatch
- Runtime circuit generation



// The Catalyst Stack

Frontend:

- PennyLane + Jax
- Dynamic programming model
- Python operator overloading
- Program capture

MLIR:

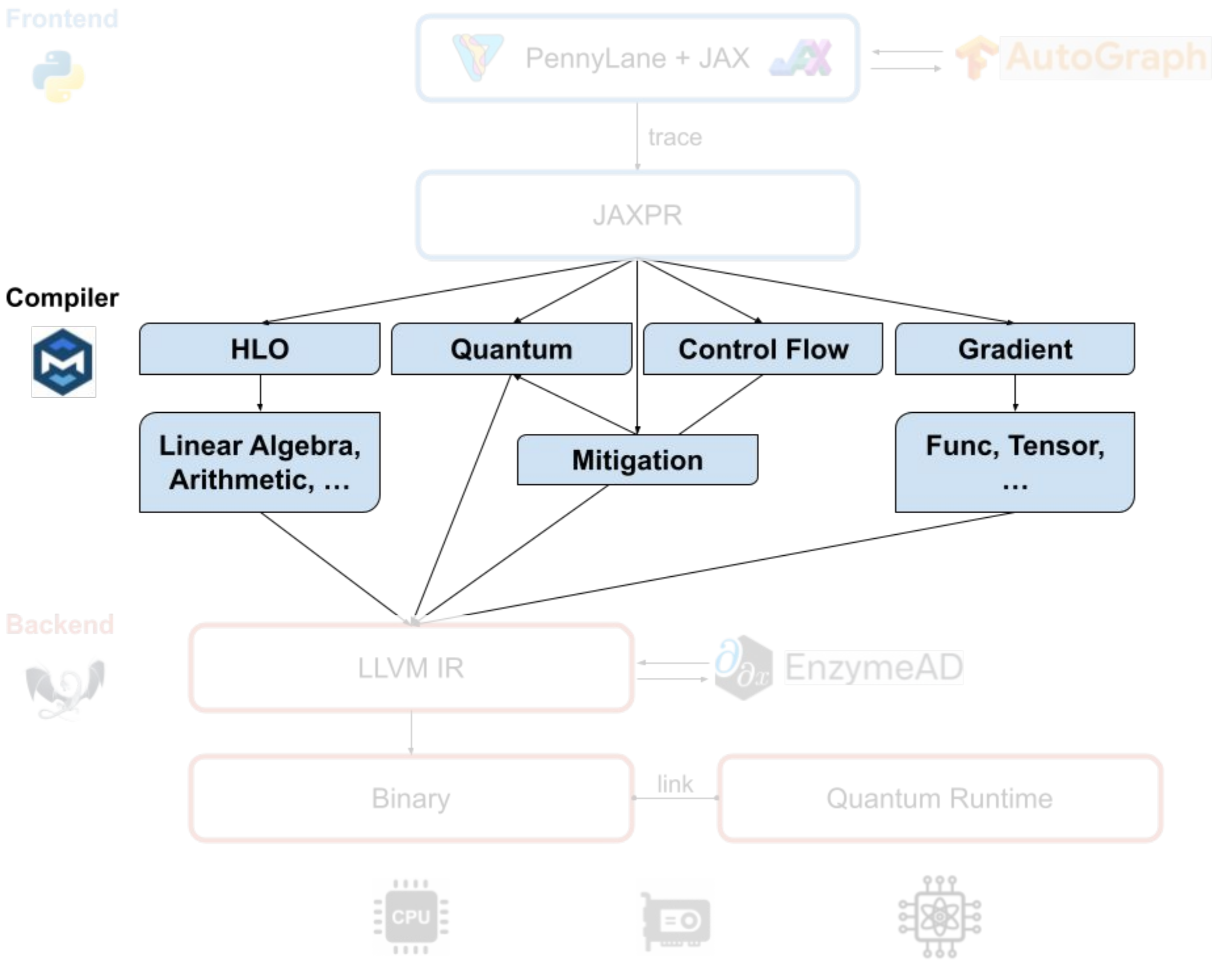
- Quantum autodiff
- Quantum circuit optimizations
- Error mitigation

CodeGen:

- Leverage LLVM infrastructure
- Enzyme autodiff
- Binary code generation

Execution:

- Device-Host interactions
- Real-time classical processing
- Dynamic instruction dispatch
- Runtime circuit generation



// Peephole Optimization example

Transformation pass of the quantum dialect

Match operations:

- Pattern rewriting framework
- `match` → `replace`

MLIR C++ →

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

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

    for (auto qb : qbs)
        if (qb.getDefiningOp() != parent)
            return failure();

    return success();
}
```

// Peephole Optimization example

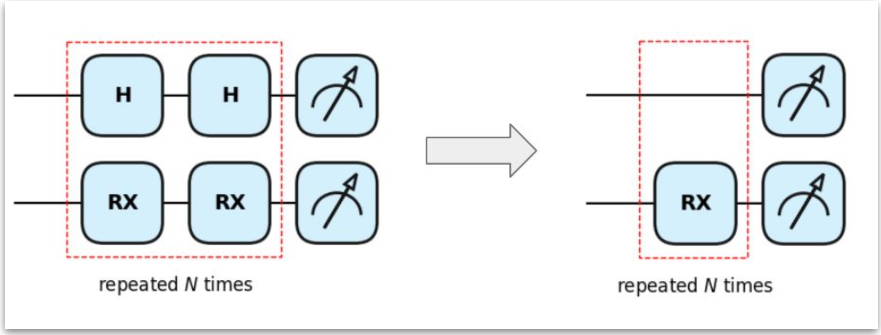
Rewrite operations:

- Graph traversal
- Qubit value semantics

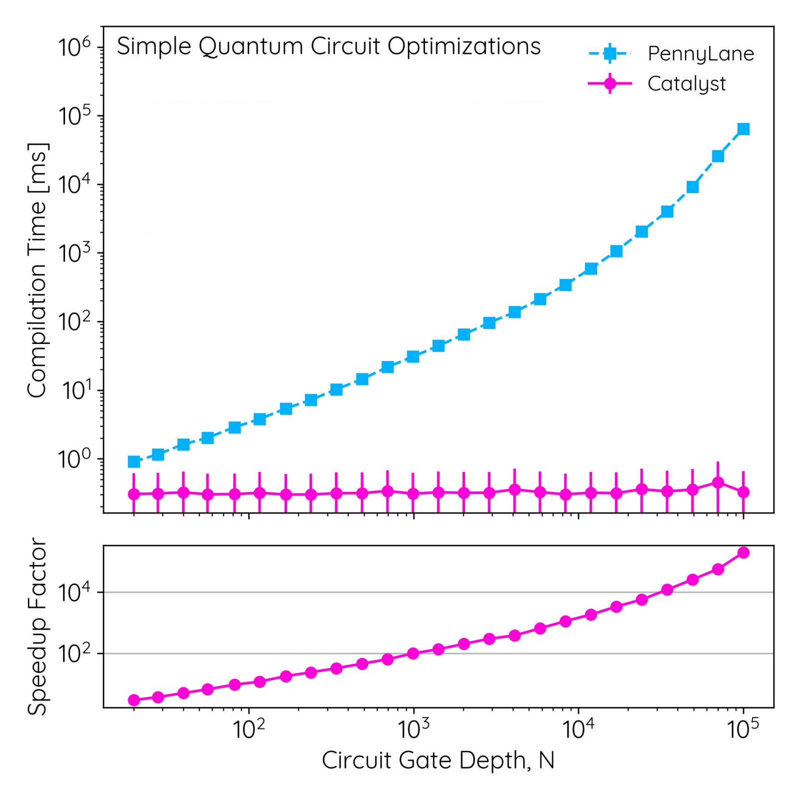
C++ for MLIR →

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

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

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

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

// Peephole optimization library



- Merge rotations pass
- Cancel inverses pass (hermitian gates)



// The Catalyst Stack

Frontend:

- PennyLane + Jax
- Dynamic programming model
- Python operator overloading
- Program capture

MLIR:

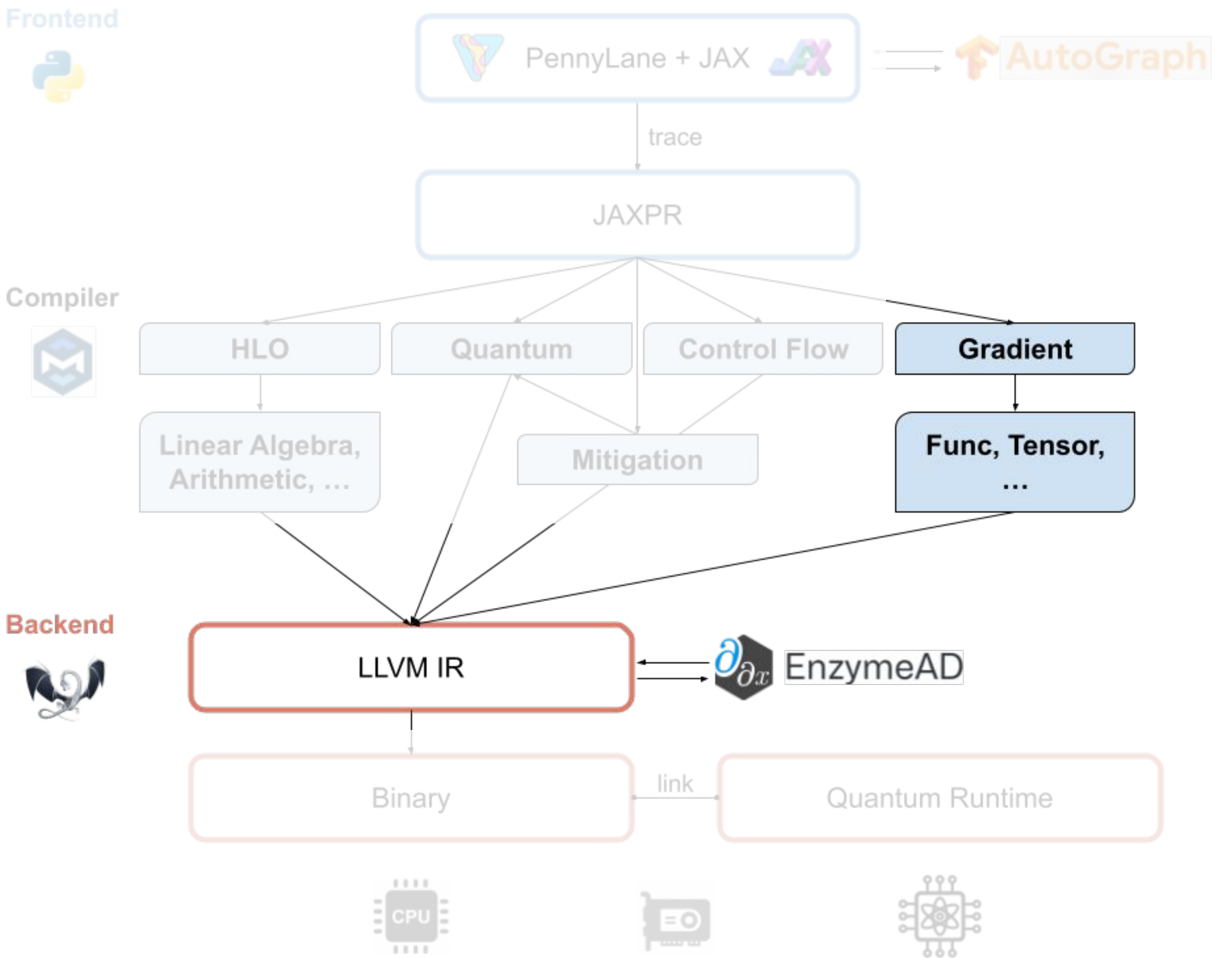
- Quantum autodiff
- Quantum circuit optimization
- Error mitigation

CodeGen:

- Leverage LLVM infrastructure
- Enzyme autodiff
- Binary code generation

Execution:

- Device-Host interactions
- Real-time classical processing
- Dynamic instruction dispatch
- Runtime circuit generation



// Derivatives of hybrid functions with Catalyst

Cost function with Catalyst gradient

JIT-compatible optimizer

Optimization loop

```
import jaxopt

dev = qml.device("lightning.qubit", wires=2)

@qml.qnode(dev)
def circuit(params):
    qml.Hadamard(0)
    qml.RX(jnp.sin(params[0]) ** 2, wires=1)
    qml.CRY(params[0], wires=[0, 1])
    qml.RX(jnp.sqrt(params[1]), wires=1)
    return qml.expval(qml.PauliZ(1))

@qjit
def cost(param):
    diff = grad(circuit, argnum=0)
    return circuit(param), diff(param)[0]

@qjit
def optimization():
    # initial parameter
    params = jnp.array([0.54, 0.3154])

    # define the optimizer
    opt = jaxopt.GradientDescent(cost, stepsize=0.4, value_and_grad=True)
    update = lambda i, args: tuple(opt.update(*args))

    # perform optimization loop
    state = opt.init_state(params)
    (params, _) = jax.lax.fori_loop(0, 100, update, (params, state))

    return params

>>> min_param = optimization()
>>> cost(min_param)
[array(-0.88926131), array([ 9.93427562e-06, -3.26427774e-05])]
```

// The gradient dialect

- All gradient operations lower to the **BackPropOp** in the gradient dialect.
- Enzyme: https://github.com/EnzymeAD/Enzyme
- The gradient dialect contains passes to lower our MLIR to **Enzyme** calls in LLVM.
  - Bufferization
  - Destination passing style
  - Register gradient rules for the quantum parts
  - Generate **__enzyme_autodiff** function calls
- Enzyme drives the generation of the derivative code in **LLVM**.

```
@__enzyme_register_gradient_circuit_0.quantum = global [3 x ptr] [ptr @circuit_0.quantum, ptr
@circuit_0.quantum.augfwd, ptr @circuit_0.quantum.customqgrad]

call void (...) @__enzyme_autodiff0(ptr @circuit_0.preprocess, ptr @enzyme_const, ptr %0, ptr
%1, ptr %19, i64 %2, i64 %3, i64 %4, ptr @enzyme_const, i64 %6, ptr @enzyme_const, ptr %25,
ptr @enzyme_dupnoneed, ptr %25, ptr %26, i64 0)
```

// The Catalyst Stack

Frontend:

- PennyLane + Jax
- Dynamic programming model
- Python operator overloading
- Program capture

MLIR:

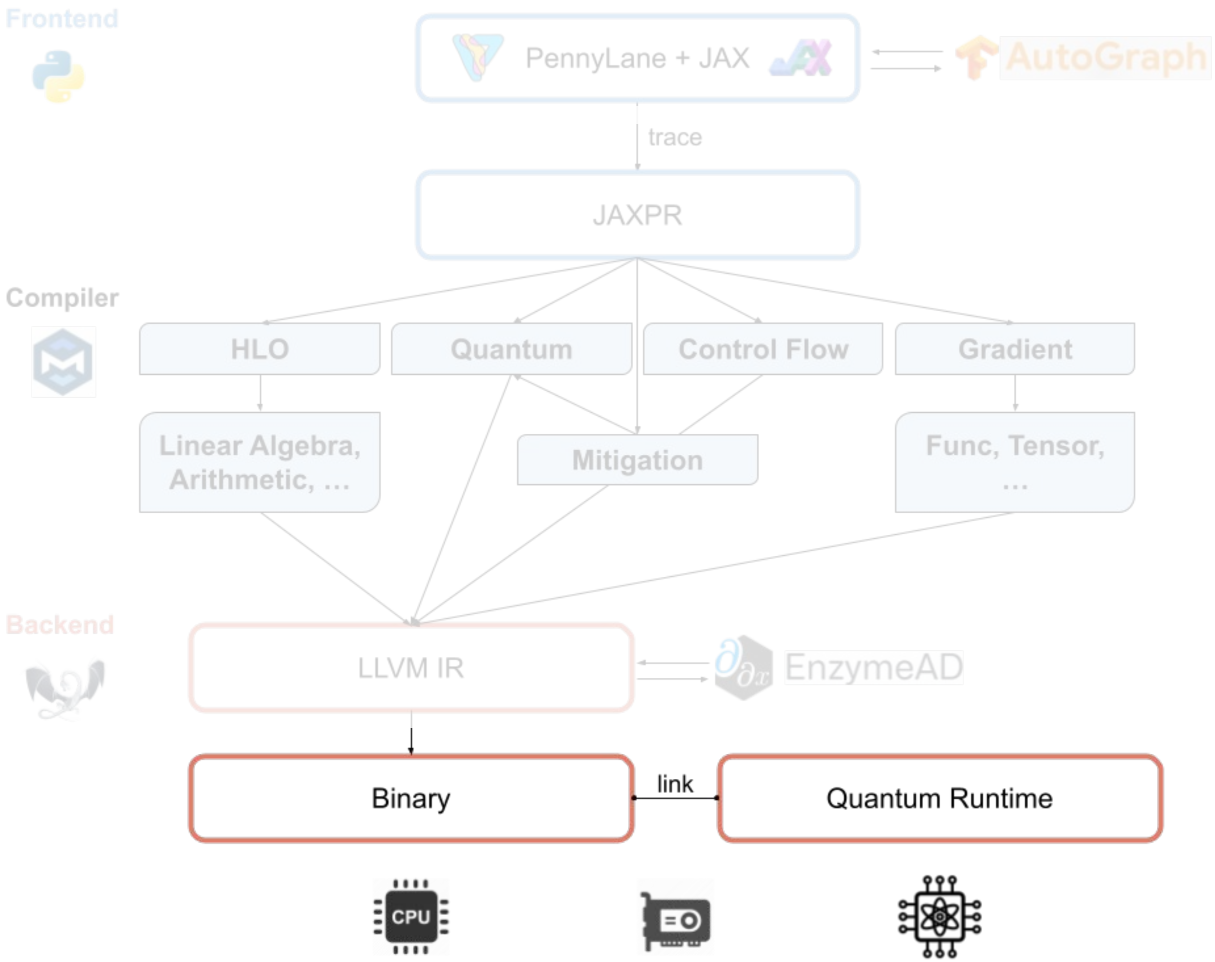
- Quantum autodiff
- Quantum circuit optimization
- Error mitigation

CodeGen:

- Leverage LLVM infrastructure
- Enzyme autodiff
- Binary code generation

Execution:

- Device-Host interactions
- Real-time classical processing
- Dynamic instruction dispatch
- Runtime circuit generation

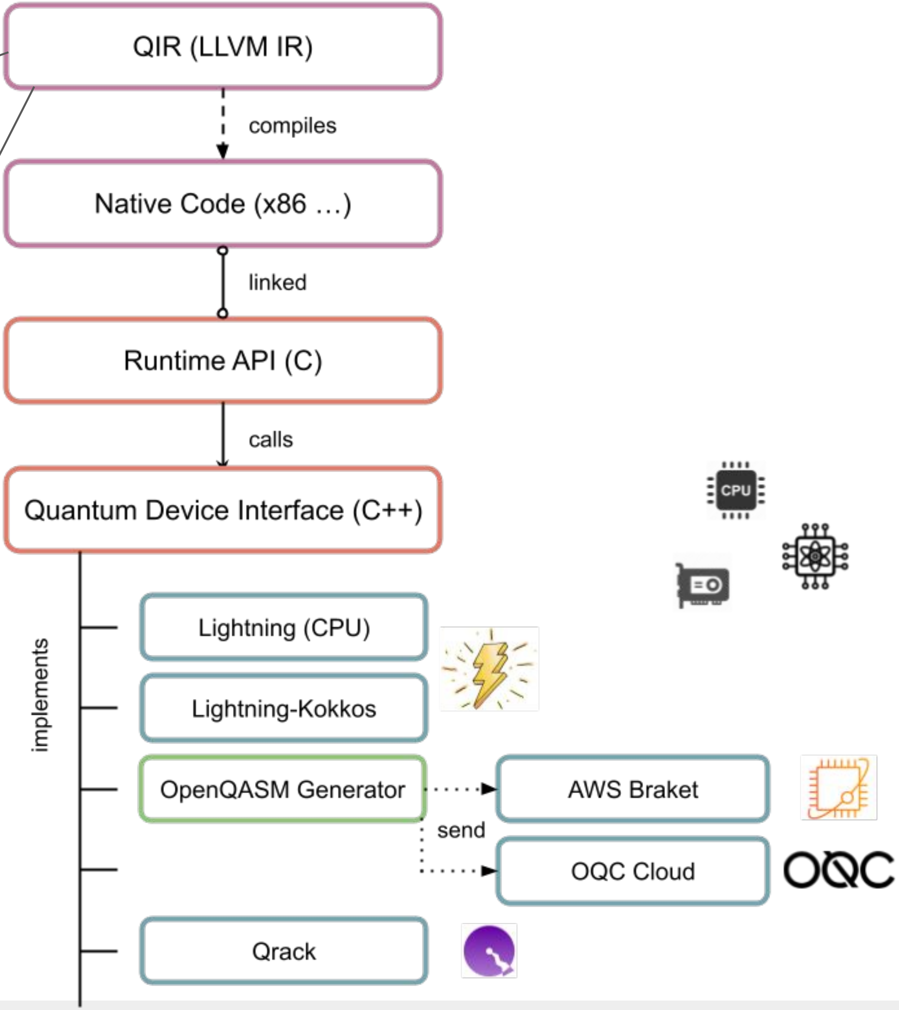


// The Execution Stack

```
void __catalyst__qis__Hadamard(QUBIT *)
void __catalyst__qis__CNOT(QUBIT *, QUBIT *)

RESULT *__catalyst__qis__Measure(QUBIT *)

ObsIdType __catalyst__qis__TensorObs(int64_t, ...)
double __catalyst__qis__Expval(ObsIdType)
```

Runtime Library:

- Thin layer between “QIR” and device backends
- Memory management & Error handling
- Quantum device instantiation and dispatching
- **Asynchronous** execution
- **Real-time** measurement feedback
- Runtime circuit generation for cloud execution



Thank you

XANADU

Romain Moyard

romain@xanadu.ai

Erick Ochoa Lopez

erick.ochoalopez@xanadu.ai

David Ittah

davidi@xanadu.ai

GitHub

https://github.com/PennyLaneAI/catalyst