// 5th International Workshop on Quantum Compilation

CATALYST

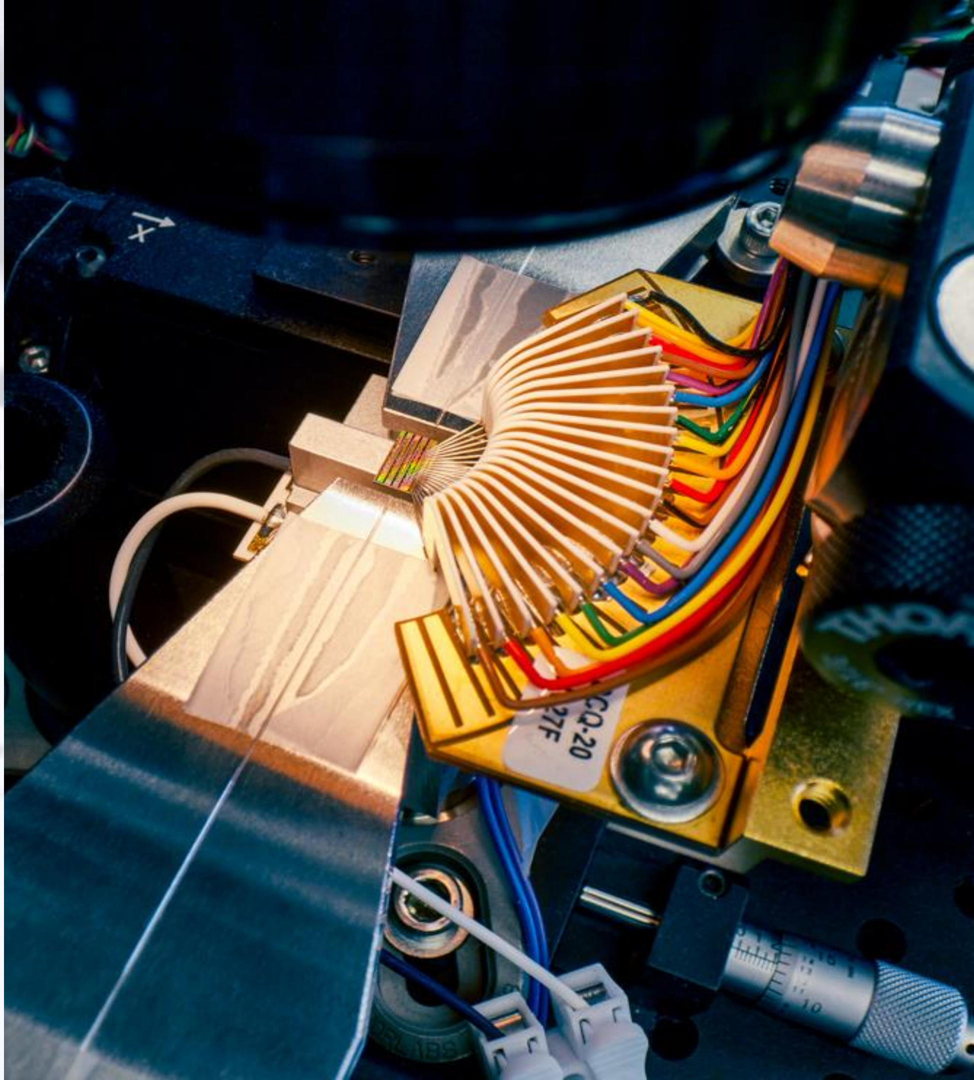
An AOT/JIT compiler for accelerated quantum computing

XANADU

// Our Mission

To build quantum computers that are useful and available to people everywhere

Founded	Headquarters	People	Funding ($US)
2016	**Toronto**	**170+**	**245M**



// The Catalyst Team

Erick Ochoa Lopez
(erick-xanadu)

David Ittah
(dime10)

Jacob Mai Peng
(pengmai)

Ali Asadi
(maliasadi)

Sergei Mironov
(grwlf)

01

Towards a modern Quantum Compilation architecture

// What are early frameworks doing wrong?

PennyLane

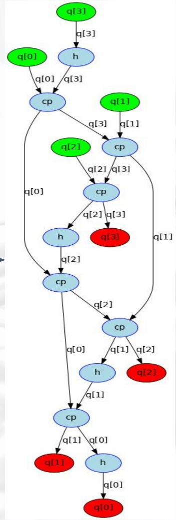
Qiskit

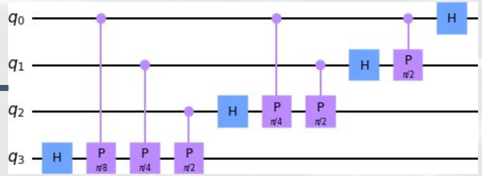
Cirq

ProjectQ

```python
def qft(n):
    circuit = QuantumCircuit(n)
    for k in range(n-1, -1, -1):
        circuit.h(k)
        for qb in range(k):
            circuit.cp(np.pi/2**(k-qb), qb, k)
    return circuit
```

n = 4





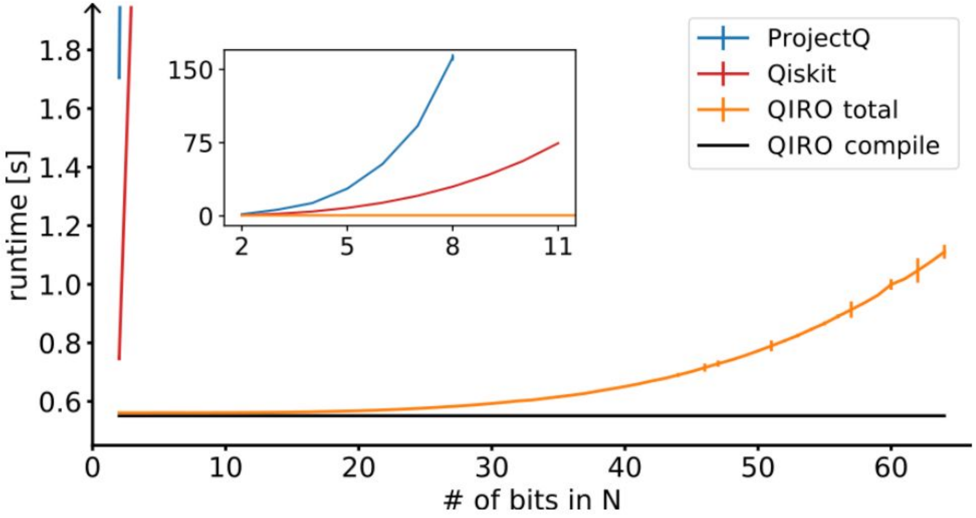
Optimization

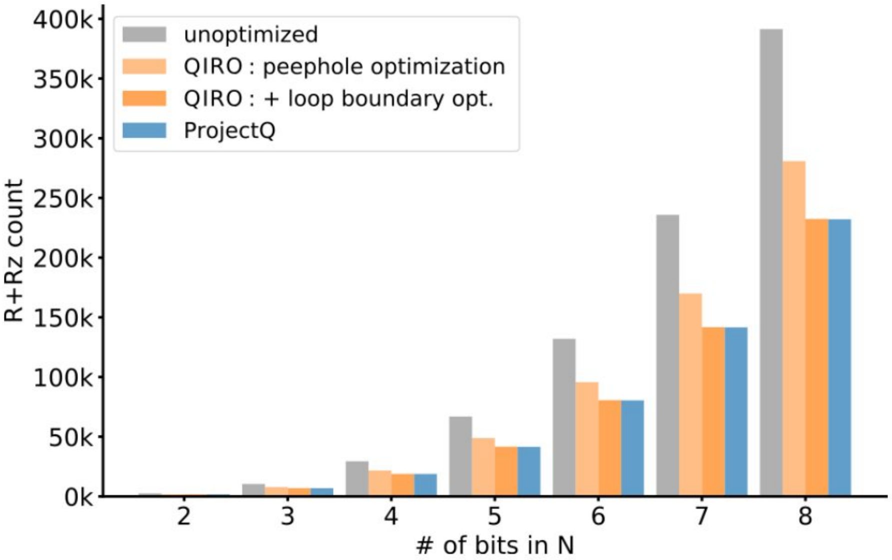
Device execution

// Scale it up

Compiling Shor's algorithm for large numbers

Modern RSA ~2000 bit keys





Flat representation of program grows ∝ n^4

→ order of 10^3 years in compile time

Dynamic representation of program

→ constant compile time (order of seconds)

David Ittah, Thomas Häner, Vadym Kliuchnikov, and Torsten Hoefler. 2022. *QIRO: A Static Single Assignment-based Quantum Program Representation for Optimization.* ACM Transactions on Quantum Computing 3, 3, Article 14. DOI

// Emergence of Quantum IRs

Early MLIR dialects

Ittah (21/01) arXiv:2101.11030
McKaskey (21/01) arXiv:2101.11365
McKaskey (21/09) arXiv:2109.00506
Peduri (21/09) arXiv:2109.02409
Guo (22/05) arXiv:2205.03866

Quantum Intermediate Representation (QIR)

Introducing QIR - Q# Blog (microsoft.com)

Industry adoption

QCOR (ornl.gov)

Introducing Catalyst (pennylane.ai)

CUDA Quantum (nvidia.com)

The future

...

Quantum Assembly or MLIR?

OpenQASM

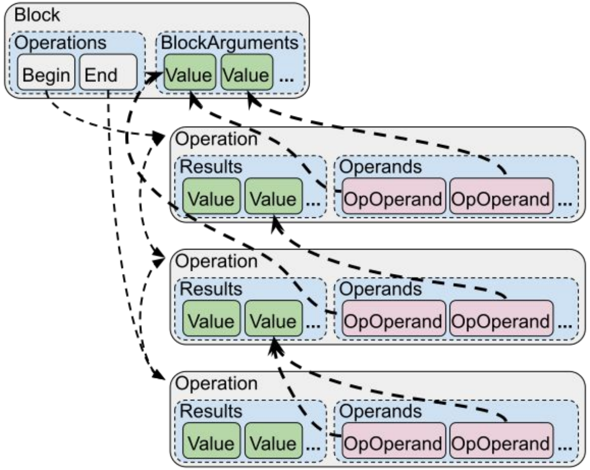
```
OPENQASM 2.0;
include "qelib1.inc";
qreg q[4];
h q[3];
cp(pi/8) q[0],q[3];
cp(pi/4) q[1],q[3];
```

2.0: Textual description of static circuit

3.0: Added (limited) classical instructions & dynamicism

No in-memory representation or transformation infrastructure

MLIR



Rich compilation ecosystem

Accommodate multiple domain-specific abstractions side-by-side

LLVM or MLIR?

QIR

```
define void @BellPair(%Qubit* %q1, %Qubit* %q2) {
entry:
  call void @__quantum__qis__h(%Qubit* %q1)
  call void @__quantum__qis__cnot(%Qubit* %q1,
                                  %Qubit* %q2)
  ret void
}
```

Opaque pointer types

Operations as functions

MLIR

```
func @BellPair(%q1: !quantum.bit, %q2: !quantum.bit)
{
    quantum.h %q1 : !quantum.bit
    quantum.cnot %q1, %q2 : !quantum.bit,
                            !quantum.bit
    return
}
```

Extensible types, operations, attributes, assembly, verifiers, structured ops, ...

What is MLIR? Structure!

LLVM IR

```
define double @func(double %arg) {
...
header:
  %i = phi i32 [ 0, %entry ], [ %ip1, %body ]
  %x = phi double [ %arg, %entry ], [ %xp2, %body ]

  %cmp = icmp ult i32 %i, 10
  br i1 %cmp, label %body, label %exit

body:
  %xp2 = fadd double %x, 2.000000e+00

  %ip1 = add i32 1, %i
  br label %header
...
}
```

MLIR

```
func @func(%arg: f64) {
  ...

  scf.for %i = 0 to 10 step 1 iter_args(%x = %arg) {
    %xp2 = arith.addf %x, 2 : f64
    scf.yield %xp2 : f64
  }

  ...
}
```

What is MLIR? Structure!

QIR

```
define void @region(%Qubit* %q1) {
entry:
  call void @__quantum__qis__h(%Qubit* %q1)
  call void @__quantum__qis__rz(double 0.1,
                               %Qubit* %q1)
  ret void
}
...

%f = call %Callable* @__quantum__rt__callable_create(
  [4 x void (%Tuple*, %Tuple*, %Tuple*)*]* @someOp,
  [2 x void (%Tuple*, i32)*]* null,
  %Tuple* null)

call @__quantum__rt__callable_make_adjoint(%f)
call @__quantum__rt__callable_invoke(%f)
```

MLIR

```
quantum.adjoint {
  quantum.h %q1 : !quantum.bit
  quantum.rz(0.1) %q1 : !quantum.bit
}
```

LLVM or MLIR?

QIR

```
define void @BellPair(%Qubit* %q1, %Qubit* %q2) {
entry:
  call void @__quantum__qis__h(%Qubit* %q1)
  call void @__quantum__qis__cnot(%Qubit* %q1,
                                  %Qubit* %q2)
  ret void
}
```

Opaque pointer types

Operations as functions

MLIR

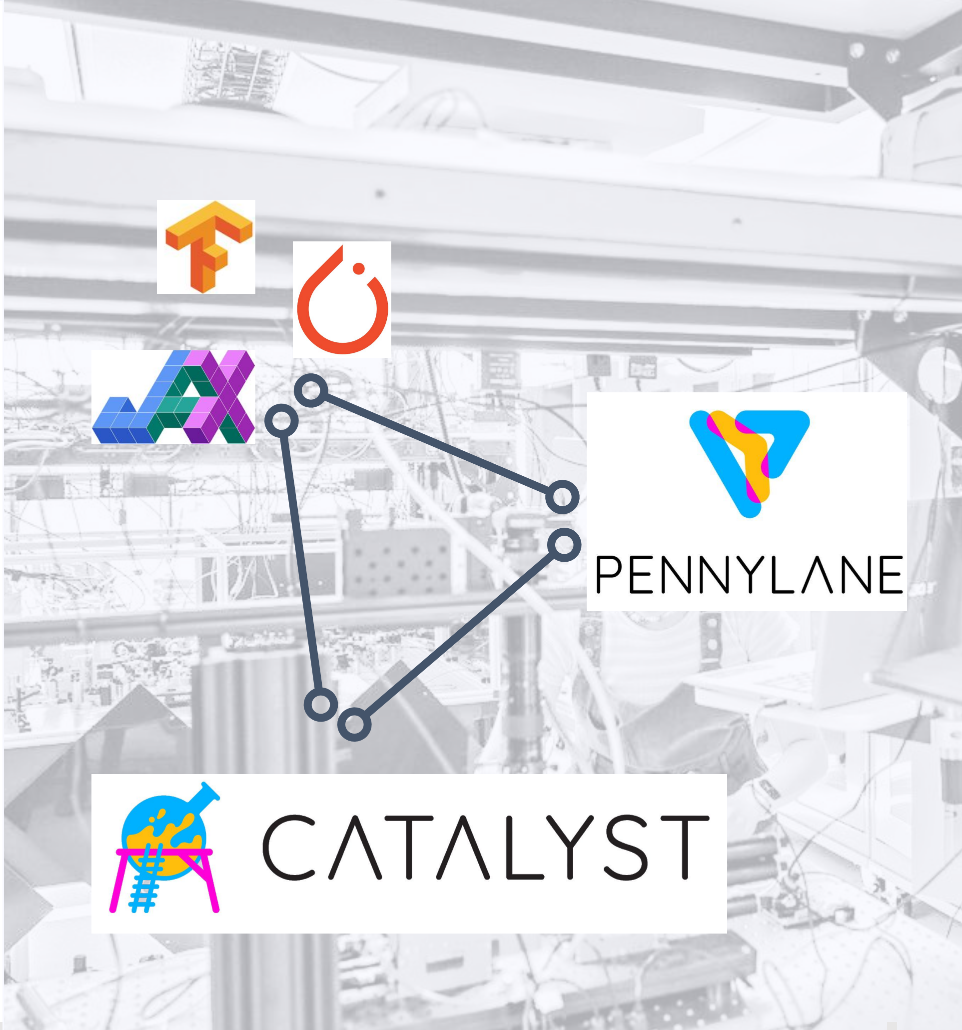
```
func @BellPair(%q1: !quantum.bit, %q2: !quantum.bit)
{
    quantum.h %q1 : !quantum.bit
    quantum.cnot %q1, %q2 : !quantum.bit,
                            !quantum.bit
    return
}
```

Extensible types, operations, attributes, assembly, verifiers, structured ops, ...

02

Catalyst

Reimagining the quantum computing stack



PennyLane

Hardware/simulator device

JAX just-in-time compilation

Quantum function/kernel

Call like you would a function. Executes on the quantum device, integrates with Python packages

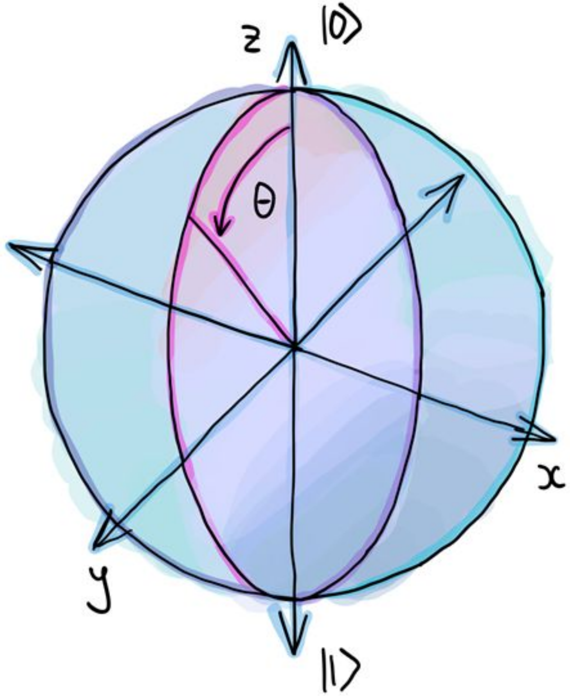
Hardware compatible automatic differentiation

Python control flow

```
import pennylane as qml
import jax
from jax import numpy as jnp

dev = qml.device('braket.aws.qubit', device_arn=...)

@jax.jit
@qml.qnode(dev, interface="jax")
def circuit(params):
    qml.RX(params[0], wires=0)

    for i in range(0, 3):
        qml.CRX(params[i + 1], wires=[i, i + 1])

    qml.Hadamard(wires=3)

    return qml.expval(qml.PauliZ(1) @ qml.PauliZ(2))

>>> params = jnp.array([[1.6, 1.2, -0.3, 1.0], [0.8, -0.543, 0.1, 0.6]]).T
>>> circuit(params)
DeviceArray([0.6791972, 0.9782419], dtype=float32)

>>> cost = lambda params: jnp.sum(jnp.sin(params))
>>> jax.grad(cost)(params)
DeviceArray([[-0.02919955,  0.6967067 ],
             [ 0.3623577 ,  0.8561624 ],
             [ 0.9553365 ,  0.9950042 ],
             [ 0.5403023 ,  0.8253356 ]], dtype=float32)
```

Current Issues

- Reducing latency between classical and quantum components
- Compilation bottlenecks - scaling to very large circuits
- Parametrized circuits to reuse compilation
- Dynamic quantum programs - quantum execution adapts to intermediate results, unbounded programs
- Heterogenous execution - distribute computation across host machine, accelerators, and quantum computers
- Unified architecture for compiling quantum & classical



Catalyst

- Simple UI
- Compile quantum kernel or entire workflow
- Fully differentiable

```python
import pennylane as qml
from catalyst import qjit, grad, cond, for_loop
from jax import numpy as jnp

dev = qml.device("lightning.qubit", wires=1)

@qml.qnode(dev)
def circuit(phi1, phi2):
    qml.RX(phi1, wires=0)
    qml.RY(phi2, wires=0)
    return qml.expval(qml.PauliZ(0))

@qjit
def cost(x, y):
    return jnp.sin(jnp.abs(circuit(x, y)[0])) - 1

>>> cost(0.53, 0.12)
-0.24437858702920867

>>> grad(cost, argnum=[0, 1])(0.53, 0.12)
(Array(-0.32874746), Array(-0.06765491))
```

Catalyst

- Simple UI
- Compile quantum kernel or entire workflow
- Fully differentiable
- JIT-compatible control flow
- Dynamic programs (mid-circuit measurements, adaptive circuits)

```python
# define a loop function
def loop_func(i: int, x: float):

    # define a conditional ansatz:
    def ansatz():
        qml.RX(x, wires=0)
        qml.Hadamard(wires=0)

    # apply the conditional quantum function
    cond(x > 1.4)(ansatz)()

    # update the value of x for the next iteration
    return x + jnp.pi / 4

# apply for loop n times
for_loop(0, n, 1)(loop_func)(init_x)
```

// The Catalyst Stack

Frontend:

- Program capture (tracing)
- Extend PL with dynamic elements

MLIR:

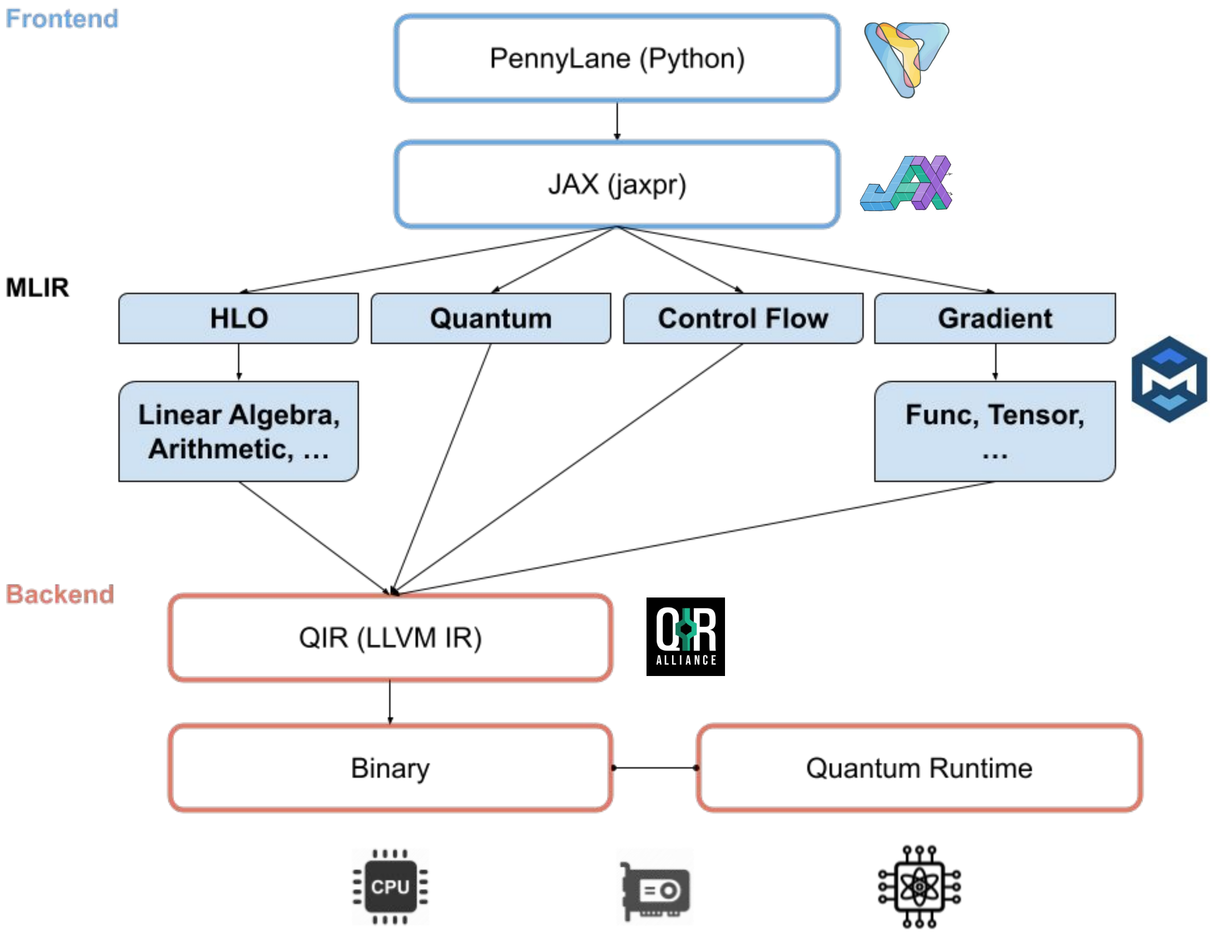
- Hybrid autodiff
- Circuit optimizations
- Classical optimization

CodeGen:

- Single source
- Leverage LLVM infrastructure

Execution:

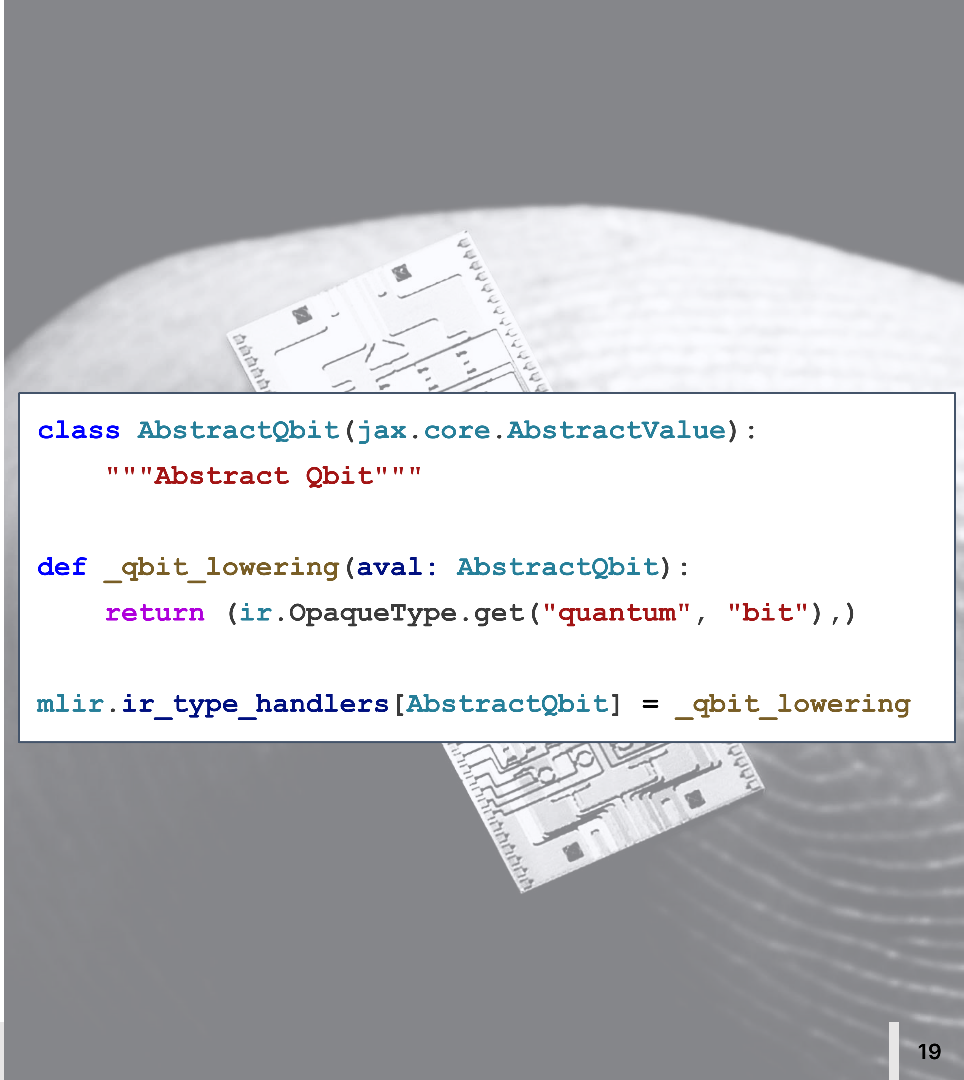
- Support Device-Host interactions
- Dynamic instruction dispatch
- Heterogeneous environment
- Runtime circuit generation + Remote execution



// PL + JAX = <3

Converting Python to MLIR

- ✓ Extending the JAX program representation



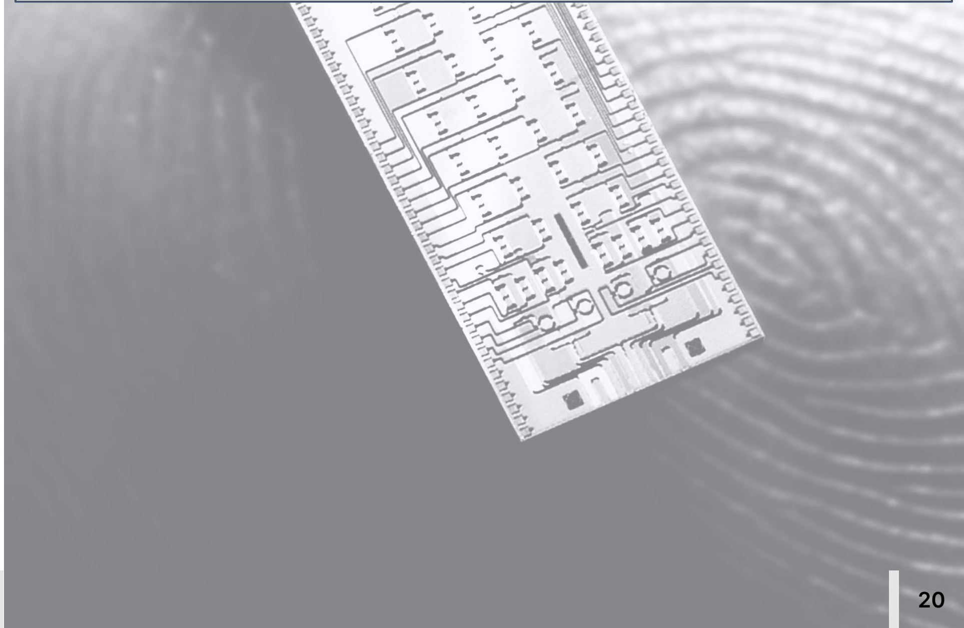
```python
class AbstractQbit(jax.core.AbstractValue):
    """Abstract Qbit"""

def _qbit_lowering(aval: AbstractQbit):
    return (ir.OpaqueType.get("quantum", "bit"),)

mlir.ir_type_handlers[AbstractQbit] = _qbit_lowering
```

// PL + JAX = <3

Converting Python to MLIR

- Extending the JAX program representation
- Conversion to value semantics

```python
unitary_p = jax.core.Primitive("unitary")
unitary_p.multiple_results = True

def unitary(matrix, *qubits):
    """Instantiate JAX primitive."""
    return unitary_p.bind(matrix, *qubits)

def _unitary_abstract_eval(matrix, *qubits):
    return (AbstractQbit(),) * len(qubits)
```



// PL + JAX = <3

Converting Python to MLIR

- ✓ Extending the JAX program representation
- ✓ Conversion to value semantics
- ✓ Custom MLIR lowerings

```
unitary_p = jax.core.Primitive("unitary")
unitary_p.multiple_results = True

def unitary(matrix, *qubits):
    """Instantiate JAX primitive."""
    return unitary_p.bind(matrix, *qubits)

def _unitary_abstract_eval(matrix, *qubits):
    return (AbstractQbit(),) * len(qubits)
```

```
def _unitary_lowering(ctx, matrix: ir.Value,
        *qubits: Tuple[ir.Value]):
    ctx.allow_unregistered_dialects = True

    mlir_op = QubitUnitaryOp([q.type for q in
qubits], matrix, qubits)

    return mlir_op.results

unitary_p.def_abstract_eval(_unitary_abstract_eval)
mlir.register_lowering(unitary_p, _unitary_lowering)
```

// MLIR

The Quantum IR

- ✓ Quantum dialect

```
def QubitUnitaryOp : Gate_Op<"unitary", [NoMemoryEffect]> {
    let summary = "Apply an arbitrary unitary matrix";

    let arguments = (ins
        AnyTypeOf<[
            2DTensorOf<[Complex<F64>]>, MemRefRankOf<[Complex<F64>], [2]>
        ]>:$matrix,
        Variadic<QubitType>:$in_qubits
    );

    let results = (outs
        Variadic<QubitType>:$out_qubits
    );

    let assemblyFormat = [{
        `(` $matrix `:` type($matrix) `)` $in_qubits attr-dict `:`
        type($out_qubits)
    }];
}
```

// MLIR

The Quantum IR

- ✓ Quantum dialect
- ✓ Optimizations

```
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();
}
```

// MLIR

The Quantum IR

- Quantum dialect
- Optimizations

```
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());
}
```

// MLIR

The Quantum IR

- ✓ Quantum dialect
- ✓ Optimizations
- ✓ Gradient dialect

```
def GradOp : Gradient_Op<"grad", [
        DeclareOpInterfaceMethods<CallOpInterface>,
        DeclareOpInterfaceMethods<SymbolUserOpInterface>]> {
    let summary = "Compute partial derivative tensors of a function.";

    let arguments = (ins
        StrAttr:$method,
        FlatSymbolRefAttr:$callee,
        Variadic<AnyType>:$operands,
        AnyIntElementsAttr:$diffArgIndices
    );

    let results = (outs
        Variadic<AnyTypeOf<[AnyFloat, RankedTensorOf<[AnyFloat]>]>>
    );

    let assemblyFormat = [{
        $method $callee `(` $operands `)` attr-dict `:`
        functional-type($operands, results)
    }];
}
```


// Quantum Autodiff

hybrid quantum function

$R^n \rightarrow R$

Real function computed via quantum execution

```
@qml.qnode(dev)
def circuit(phi):
    qml.RX(phi, wires=0)
    qml.RY(2 * phi, wires=1)
    qml.CNOT(wires=[1, 2])
    return qml.expval(qml.PauliZ(0))
```

// Quantum Autodiff

hybrid quantum function
$R^n \to R$

classical args processing
$R^n \to R^m$

quantum function
$R^m \to R$

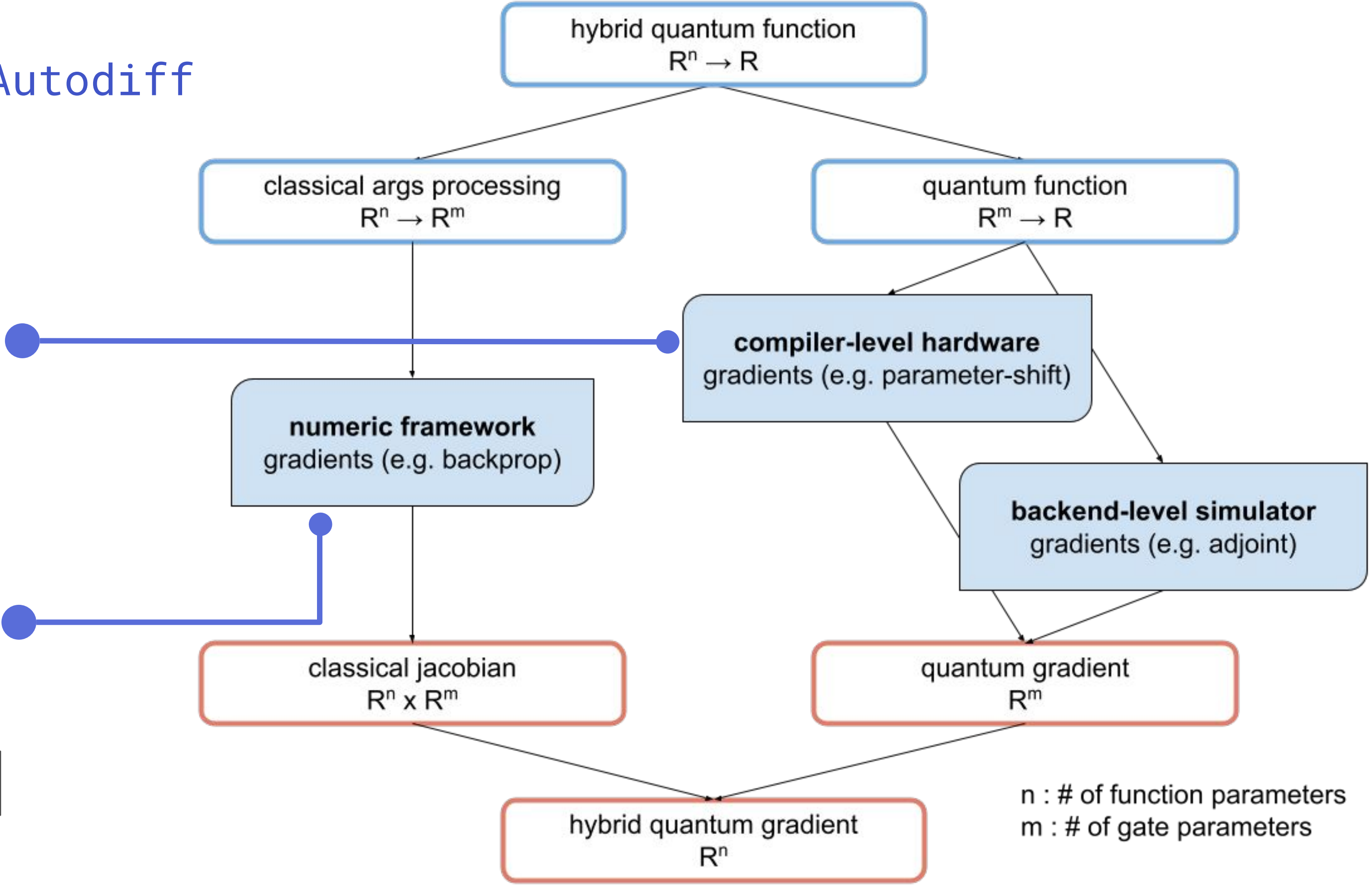
Classical function for argument processing

```
def gate_args(phi):
    return phi, 2 * phi
```

```
def circuit(arg1, arg2):
    qml.RX(arg1, wires=0)
    qml.RY(arg2, wires=1)
    qml.CNOT(wires=[1, 2])
    return qml.expval(qml.PauliZ(0))
```

Quantum execution, typically producing expectation values

// Quantum Autodiff



Perform shifting dynamically at runtime

Enzyme integration for hybrid gradient architecture

CodeGen

- ✓ Leverage built-in MLIR to LLVM IR conversion
- ✓ Add lowering rules from the quantum dialect to QIR

MLIR Dialects → LLVM Dialect → LLVM IR

```
func @BellPair(%q1: !quantum.bit, %q2: !quantum.bit)
{
    quantum.h %q1 : !quantum.bit
    quantum.cnot %q1, %q2 : !quantum.bit,
                            !quantum.bit
    return
}
```

```
define void @BellPair(%Qubit* %q1, %Qubit* %q2) {
entry:
  call void @__quantum__qis__h(%Qubit* %q1)
  call void @__quantum__qis__cnot(%Qubit* %q1,
                                  %Qubit* %q2)
  ret void
}
```

Extended QIR Target

- ✓ Usual Quantum Instruction Set

```
// Quantum Gates
void __quantum__qis__PauliX(QUBIT *)
void __quantum__qis__Hadamard(QUBIT *)
void __quantum__qis__S(QUBIT *)

void __quantum__qis__RX(double, QUBIT *)
void __quantum__qis__Rot(double, double, double, QUBIT *)

void __quantum__qis__CNOT(QUBIT *, QUBIT *)
void __quantum__qis__MultiRZ(double, int64_t, /*qubits*/...)

RESULT *__quantum__qis__Measure(QUBIT *)
```

// LLVM meets Quantum

Extended QIR Target

- Usual Quantum Instruction Set
- Observables
- Measurement statistics

```
// Observables
ObsIdType __quantum__qis__NamedObs(int64_t, QUBIT *)
ObsIdType __quantum__qis__HermitianObs(MemRefT_CplxT_double_2d *,
int64_t, /*qubits*/...)
ObsIdType __quantum__qis__TensorObs(int64_t, /*obsKeys*/...)
ObsIdType __quantum__qis__HamiltonianObs(MemRefT_double_1d *,
int64_t, /*obsKeys*/...)
```

```
// Measurement processes
double __quantum__qis__Expval(ObsIdType)
void __quantum__qis__Probs(MemRefT_double_1d *, int64_t,
/*qubits*/...)
void __quantum__qis__Sample(MemRefT_double_2d *, int64_t, int64_t,
/*qubits*/...)
void __quantum__qis__State(MemRefT_CplxT_double_1d *, int64_t,
/*qubits*/...)
```

// LLVM meets Quantum

Extended QIR Target

- ✓ Usual Quantum Instruction Set
- ✓ Observables
- ✓ Measurement statistics
- ✓ Device-based gradients

```
// Gradients
void __quantum__rt__toggle_recorder(bool)
void __quantum__qis__Gradient(int64_t, /*results*/...)
```

// The Execution Stack

User program:

- Compiled to native binary
- Linked against runtime library

Runtime Library:

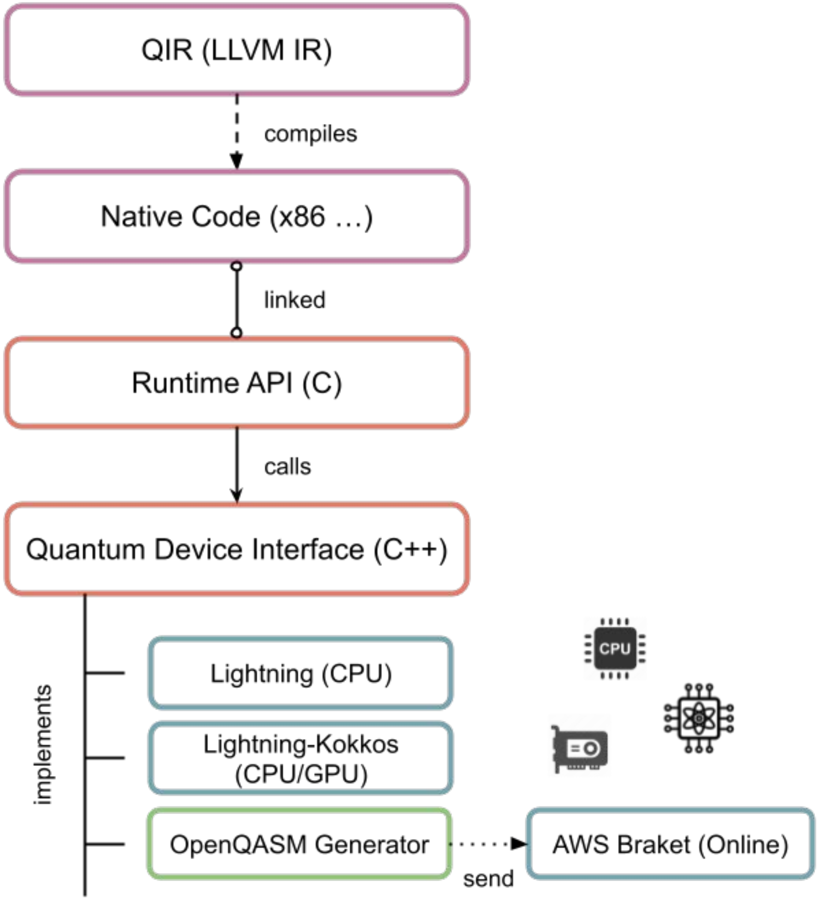
- Thin layer between QIR and device backends
- Memory management & Error handling
- Quantum Device instantiation and dispatching

Local devices:

- High-performance simulators
- Real-time measurement feedback
- Unbounded loops
- ...

Legacy execution mode:

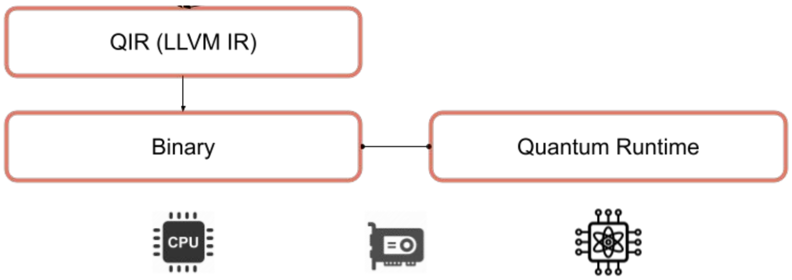
- OpenQASM generation at runtime
- Dispatch circuit to online providers
- Full hybrid workflows
- No feedback, High latency



What next?

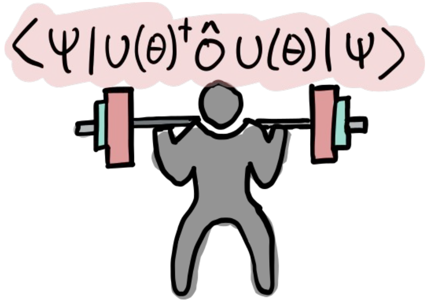
03

Device compilation & execution



Device-specific compilation, hardware execution, compilation for QPU - co-processor systems

Optimizations



Moving out of beta → optimizing for speed

Quantum compilation algorithms in MLIR

Thank you

pennylane.ai
Twitter → @PennyLane.ai

David Ittah
davidi@xanadu.ai

GitHub
https://github.com/PennyLaneAI/catalyst