The Journal of Open Source Software

Catalyst: a Python JIT compiler for auto-differentiable hybrid quantum programs

David Ittah [1*], **Ali Asadi**[1*], **Erick Ochoa Lopez**[1*], **Sergei Mironov**[1*], **Samuel Banning**[1*], **Romain Moyard** [1*], **Mai Jacob Peng** [1*], **and Josh Izaac** [1¶]

1 Xanadu, Toronto, ON, M5G 2C8, Canada ¶ Corresponding author * These authors contributed equally.

DOI: 10.21105/joss.06720

Software

- Review
- Repository
- Archive

Editor: Daniel S. Katz

Reviewers:

- @pmcao
- @otbrown

Submitted: 01 November 2023
Published: 09 July 2024

Summary

Catalyst is a software package for capturing Python-based hybrid quantum programs (that is, programs that contain both quantum and classical instructions), and just-in-time (JIT) compiling them down to an MLIR and LLVM representation and generating binary code. As a result, Catalyst enables the ability to rapidly prototype quantum algorithms in Python alongside efficient compilation, optimization, and execution of the program on classical and quantum accelerators. In addition, Catalyst allows for advanced quantum programming features essential for fault-tolerant hardware support and advanced algorithm design, such as mid-circuit measurement with arbitrary post-processing, support for classical control flow in and around quantum algorithms, built-in measurement statistics, and hardware-compatible automatic differentiation (AD).

Statement of need

The rapid development and availability of quantum software and hardware has had significant influence on quantum algorithm development and general quantum computing research. Through direct access to full-featured quantum programming SDKs (Bergholm et al., 2018; Google Inc., 2018; IBM Corporation, 2016b), high-performance simulators (Bergholm et al., 2018; NVIDIA cuQuantum team, 2022), and near-term noisy hardware (Amazon Web Services, 2020; IBM Corporation, 2016a; Madsen et al., 2022), researchers have new tools available to prototype, execute, analyze, and iterate during algorithm development. Notably, this has resulted in the development and exploration of new categories of quantum algorithms that take advantage of the strong integration with advanced classical tooling; an example being auto-differentiation and variational quantum algorithms (Delgado et al., 2021; Farhi et al., 2014; McClean et al., 2016).

However, as our hardware capabilities scale, it is becoming clear that we cannot separate classical and quantum parts of the program; classical processing remains essential for processing quantum input and output, as well as mid-circuit processing that must be performant enough to keep up with the quantum execution. Furthermore, we must support this while retaining the ability to rapidly prototype, integrate with classical tooling and accelerators, and provide efficient optimization and compilation of both quantum and classical instructions.

One of the core goals of Catalyst is to provide a unified representation for hybrid programs with which to drive optimization, device compilation, automatic differentiation, and many other types of transformations in a scalable way. Moreover, Catalyst is being developed to support next-generation quantum programming paradigms, such as dynamic circuit generation with classical control flow, real-time measurement feedback, qubit reuse, and dynamic quantum memory management. Most importantly, Catalyst provides a way to transform large scale user

workflows from Python into low-level binary code for accelerated execution in heterogeneous environments.

Catalyst is divided into three core components: a *frontend* that captures and lowers hybrid Python programs, a *compiler* that applies quantum and classical optimizations and transformations, and a *runtime* that allows the compiled binary to call into quantum devices for execution.

Frontend

The Catalyst frontend, built in Python and C++, directly integrates with both PennyLane (Bergholm et al., 2018), a Python framework for differentiable quantum programming, and JAX (Bradbury et al., 2018), a Python framework for accelerated auto-differentiation, to capture hybrid quantum programs. As a result, by decorating hybrid programs with the `@qjit` decorator, the Catalyst frontend is able to capture and ahead-of-time or just-in-time compile (from within Python) the quantum and classical instructions provided by PennyLane and JAX. In addition, Catalyst provides high-level functions for compact and dynamic circuit representation (`for_loop`, `cond`, `while_loop`, `measure`) as well as auto-differentiation (`grad`, `jacobian`, `vjp`, `jvp`). Preliminary support for AutoGraph (Moldovan et al., 2018) also allows users to write hybrid quantum programs using native Python control flow; all branches of the computation will be represented in the captured hybrid program.

Compiler

Building on the LLVM (Lattner & Adve, 2004) and MLIR (Lattner et al., 2021) compiler frameworks, and the QIR project (QIR Alliance, 2021), compilation is then performed on the MLIR-based representation for hybrid quantum programs defined by Catalyst. The compiler invokes a sequence of transformations that lowers the hybrid program to a lower level of abstraction, outputting LLVM IR with QIR syntax. In addition to the lowering process, various transformations take place, including quantum optimizations (adjoint cancellation, operator fusion), classical optimizations (code elimination), and automatic differentiation. In the latter case, classical auto-differentiation is provided via the Enzyme (Moses & Churavy, 2020) framework, while hardware-compatible quantum gradient methods (such as the parameter-shift rule (Schuld et al., 2019; Wierichs et al., 2022)) are provided as Catalyst compiler transforms.

Runtime

The Catalyst runtime is designed to enable Catalyst's highly dynamic execution model. As such, it generally assumes direct communication between a quantum device and its classical controller or host, although it also supports more restrictive execution models. Execution of the user program proceeds on the host's native architecture, while the runtime provides an abstract communication API for quantum devices that the user program is free to invoke at any time during its execution. Currently, Catalyst provides runtime integration for the high-performance PennyLane Lightning suite of simulators (*PennyLane Lightning*, 2023), as well as an OpenQASM3 pipeline with Amazon Braket simulator and hardware support.

Examples

The following example highlights the capabilities of the Catalyst frontend, enabling scalable and high-performance quantum computing from a feature-rich interactive Python environment.

First, we utilize Catalyst to just-in-time compile a complex function involving a mix of classical and quantum processing. Note that, through the AutoGraph feature, native Python control flow is automatically captured, allowing both branches to be represented in the compiled program.

```
import pennylane as qml
from catalyst import qjit, measure
from jax import numpy as jnp

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

@qjit(autograph=True)
def hybrid_function(x):

    @qml.qnode(dev, diff_method="parameter-shift")
    def circuit(x):
        qml.RX(x, wires=0)
        qml.RY(x ** 2, wires=1)
        qml.CNOT(wires=[0, 1])

        for i in range(0, 10):
            m = measure(wires=0)

            if m == 1:
                qml.CRX(x * jnp.exp(- x ** 2), wires=[0, 1])

            x = x * 0.2

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

    return jnp.sin(circuit(x)) ** 2
```

```
>>> hybrid_function(0.543)
array(0.70807342)
```

We can also consider an example that includes a classical optimization loop, such as optimizing a quantum computer to find the ground state energy of a molecule:

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

mol = qml.data.load("qchem", molname="H3+")[0]
n_qubits = len(mol.hamiltonian.wires)

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

@qjit
@qml.qnode(dev)
def cost(params):
    qml.BasisState(jnp.array(mol.hf_state), wires=range(n_qubits))
    qml.DoubleExcitation(params[0], wires=[0, 1, 2, 3])
    qml.DoubleExcitation(params[1], wires=[0, 1, 4, 5])
    return qml.expval(mol.hamiltonian)

@qjit
def optimization(params):
    loss = lambda x: (cost(x), grad(cost)(x))

    # set up optimizer and define optimization step
```

```
    opt = jaxopt.GradientDescent(loss, stepsize=0.3, value_and_grad=True)
    update_step = lambda step, args: tuple(opt.update(*args))

    # gradient descent parameter update loop using jit-compatible for-loop
    state = opt.init_state(params)
    (params, _) = for_loop(0, 10, step=1)(update_step)((params, state))
    return params

>>> params = jnp.array([0.54, 0.3154])
>>> final_params = optimization(params)
>>> cost(final_params)  # optimized energy of H3+
-1.2621179827928877
>>> mol.vqe_energy  # expected energy of H3+
-1.2613407428534986
```

Here, we are using the JAXopt gradient optimization library (Blondel et al., 2022) alongside the built-in auto-differentiation capabilities of Catalyst, to compile the entire optimization workflow. For this small toy example on 6 qubits, we can time the execution after compilation on the same system, as a non-rigorous demonstration of the advantage of performing this for loop outside of Python:

```
>>> %timeit optimization(params)
599 ms ± 96 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

Comparing this to a non-compiled workflow, where the `@qjit` decorator has been removed:

```
@qml.qnode(dev)
def no_qjit_cost(params):
    qml.BasisState(jnp.array(mol.hf_state), wires=range(n_qubits))
    qml.DoubleExcitation(params[0], wires=[0, 1, 2, 3])
    qml.DoubleExcitation(params[1], wires=[0, 1, 4, 5])
    return qml.expval(mol.hamiltonian)

def no_qjit_optimization(params):
    # set up optimizer
    opt = jaxopt.GradientDescent(no_qjit_cost, stepsize=0.3, jit=False)
    state = opt.init_state(params)

    for i in range(15):
        (params, state) = opt.update(params, state)

    return params
```

```
>>> %timeit no_qjit_optimization(params)
3.73 s ± 522 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

For more code examples, please see the Catalyst documentation[1].

Installation and contribution

The Catalyst source code is available under the Apache 2.0 license on GitHub[2], and binaries are available for `pip` installation on Linux and macOS[3]. Contributions to Catalyst — via feedback, issues, or pull requests on GitHub — are welcomed. Additional Catalyst documentation and tutorials are available on our online documentation[4].

[1] https://docs.pennylane.ai/projects/catalyst
[2] https://github.com/PennyLaneAI/catalyst
[3] https://pypi.org/project/pennylane-catalyst
[4] https://docs.pennylane.ai/projects/catalyst

Discussion and future work

The Catalyst hybrid compilation stack as presented here provides an end-to-end infrastructure to explore next-generation dynamic hybrid quantum-classical algorithms, by allowing for workflows that support compressed representation of large, highly structured quantum algorithms, as well as mid-circuit measurements with arbitrary classical processing and feedforward.

The Catalyst software stack will continue to be developed alongside research, algorith, and hardware needs, with potential future work including support for quantum hardware control systems, building out a library of MLIR quantum compilation passes for optimizing quantum circuits (without unrolling classical control structure), and explorations of dynamic quantum error mitigation and proof-of-concept error correction experiments.

Quantum software is driving many new results and ideas in quantum computing research, and the PennyLane framework has already been used in a number of scientific publications (Delgado et al., 2021; Wierichs et al., 2022) and educational materials (Xanadu, 2018). By enabling researchers to scale up their ideas and algorithms, and execute on both near-term and future quantum hardware, the software presented here will help drive future research in quantum computing.

Acknowledgements

We acknowledge contributions from Lee J O'Riordan, Nathan Killoran, and Olivia Di Matteo during the genesis of this project.

References

Amazon Web Services. (2020). *Amazon Braket*. https://aws.amazon.com/braket/

Bergholm, V., Izaac, J., Schuld, M., Gogolin, C., Ahmed, S., Ajith, V., Alam, M. S., Alonso-Linaje, G., AkashNarayanan, B., Asadi, A., & others. (2018). Pennylane: Automatic differentiation of hybrid quantum-classical computations. *arXiv Preprint arXiv:1811.04968*. https://doi.org/10.48550/arXiv.1811.04968

Blondel, M., Berthet, Q., Cuturi, M., Frostig, R., Hoyer, S., Llinares-López, F., Pedregosa, F., & Vert, J.-P. (2022). Efficient and modular implicit differentiation. *Advances in Neural Information Processing Systems*, *35*, 5230–5242.

Bradbury, J., Frostig, R., Hawkins, P., Johnson, M. J., Leary, C., Maclaurin, D., Necula, G., Paszke, A., VanderPlas, J., Wanderman-Milne, S., & Zhang, Q. (2018). *JAX: Composable transformations of Python+NumPy programs* (Version 0.3.13). http://github.com/google/jax

Delgado, A., Arrazola, J. M., Jahangiri, S., Niu, Z., Izaac, J., Roberts, C., & Killoran, N. (2021). Variational quantum algorithm for molecular geometry optimization. *Physical Review A*, *104*(5), 052402. https://doi.org/10.1103/physreva.104.052402

Farhi, E., Goldstone, J., & Gutmann, S. (2014). A quantum approximate optimization algorithm. *arXiv Preprint arXiv:1411.4028*. https://doi.org/10.48550/arXiv.1411.4028

Google Inc. (2018). *Cirq*. https://cirq.readthedocs.io/en/latest/

IBM Corporation. (2016a). *IBM Quantum Experience*. https://quantumexperience.ng.bluemix.net/

IBM Corporation. (2016b). *Qiskit*. https://qiskit.org/

Lattner, C., & Adve, V. (2004). LLVM: A compilation framework for lifelong program analysis and transformation. *2004 IEEE/ACM International Symposium on Code Generation and*

Optimization (CGO), 75–88. https://doi.org/10.1109/CGO.2004.1281665

Lattner, C., Amini, M., Bondhugula, U., Cohen, A., Davis, A., Pienaar, J., Riddle, R., Shpeisman, T., Vasilache, N., & Zinenko, O. (2021). MLIR: Scaling compiler infrastructure for domain specific computation. *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2–14. https://doi.org/10.1109/CGO51591.2021.9370308

Madsen, L. S., Laudenbach, F., Askarani, M. F., Rortais, F., Vincent, T., Bulmer, J. F., Miatto, F. M., Neuhaus, L., Helt, L. G., Collins, M. J., Lita, A. E., Gerrits, T., Nam, S. W., Vaidya, V. D., Menotti, M., Dhand, I., Vernon, Z., Quesada, N., & Lavoie, J. (2022). Quantum computational advantage with a programmable photonic processor. *Nature*, *606*(7912), 75–81. https://doi.org/10.1038/s41586-022-04725-x

McClean, J. R., Romero, J., Babbush, R., & Aspuru-Guzik, A. (2016). The theory of variational hybrid quantum-classical algorithms. *New Journal of Physics*, *18*(2), 023023. https://doi.org/10.1088/1367-2630/18/2/023023

Moldovan, D., Decker, J. M., Wang, F., Johnson, A. A., Lee, B. K., Nado, Z., Sculley, D., Rompf, T., & Wiltschko, A. B. (2018). AutoGraph: Imperative-style coding with graph-based performance. *arXiv Preprint arXiv:1810.08061*. https://doi.org/10.48550/arXiv.1810.08061

Moses, W., & Churavy, V. (2020). Instead of rewriting foreign code for machine learning, automatically synthesize fast gradients. In H. Larochelle, M. Ranzato, R. Hadsell, M. F. Balcan, & H. Lin (Eds.), *Advances in neural information processing systems* (Vol. 33, pp. 12472–12485). Curran Associates, Inc. https://proceedings.neurips.cc/paper/2020/file/9332c513ef44b682e9347822c2e457ac-Paper.pdf

NVIDIA cuQuantum team. (2022). *NVIDIA/cuQuantum: cuQuantum v22.03.0* (Version v22.03.0). Zenodo. https://doi.org/10.5281/zenodo.6385575

PennyLane Lightning: Fast state-vector simulators written in C++ (Version 0.33.1). (2023). http://github.com/pennylane/pennylane-lightning

QIR Alliance. (2021). *QIR Specification*. https://github.com/qir-alliance/qir-spec

Schuld, M., Bergholm, V., Gogolin, C., Izaac, J., & Killoran, N. (2019). Evaluating analytic gradients on quantum hardware. *Physical Review A*, *99*(3), 032331. https://doi.org/10.1103/physreva.99.032331

Wierichs, D., Izaac, J., Wang, C., & Lin, C. Y.-Y. (2022). General parameter-shift rules for quantum gradients. *Quantum*, *6*, 677. https://doi.org/10.22331/q-2022-03-30-677

Xanadu. (2018). *PennyLane Demos*. https://pennylane.ai/qml/demonstrations