PulseMLIR

An MLIR Dialect for Pulse Representations in Superconducting Quantum Computers

Anders Choi
Advisor: Pramod Bhatotia
Chair of Computer Systems
https://dse.in.tum.de/

15.04.2024 – 18.10.2024

Hybrid Quantum-Classical Programs

Noisy intermediate-scale quantum (NISQ) era: characterized by limited number of qubits (sub-1000) and high error rates

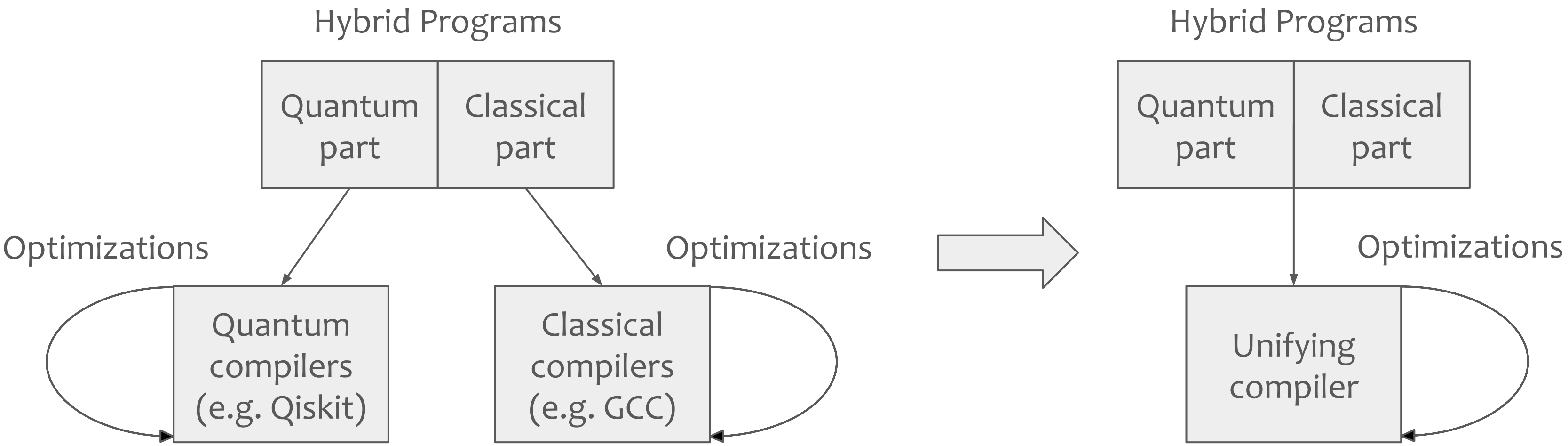
Necessary to delegate tasks to classical computers in many situations

Example NISQ algorithms:

- Variational quantum eigensolver (VQE)
- Quantum approximate optimization algorithm (QAOA)
- Quantum machine learning
- Surface code error correction

Compilation of Hybrid Quantum-Classical Programs

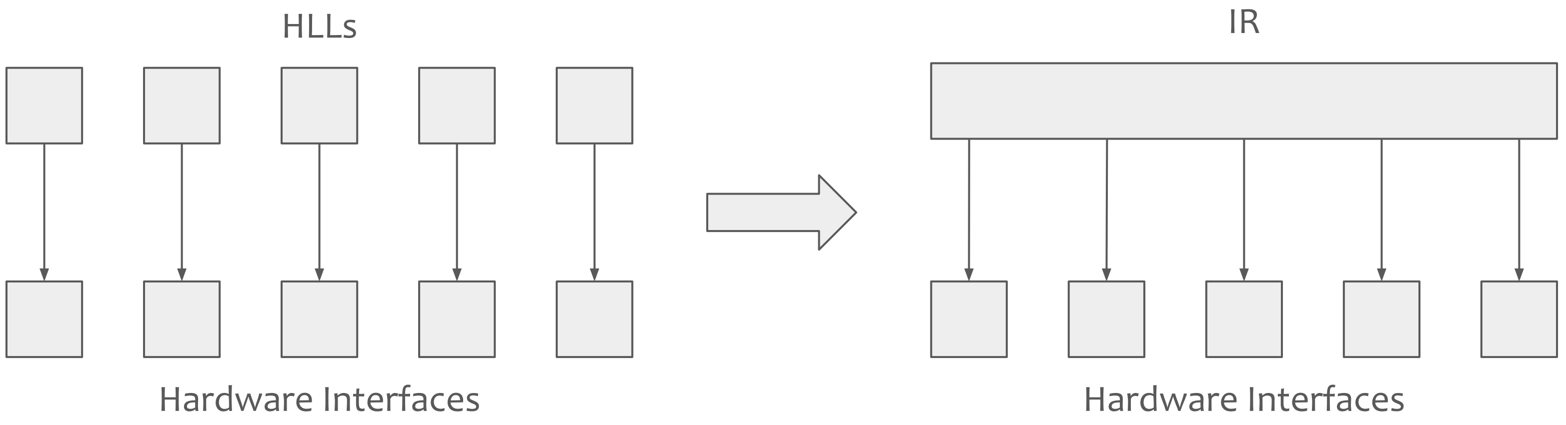
The quantum and classical parts of hybrid programs are normally compiled separately



A unifying IR - which can represent both quantum and classical programs - introduces possibilities for additional optimizations

Compilation of Pulse-Level Programs

There exists a variety of distinct high-level pulse representations for pulse generation hardware interfaces



A unifying IR - which can transform into multiple hardware interfaces - can serve as a common platform on which optimizations can be implemented

Research Gap

QIR[1]: an LLVM-based IR for hybrid quantum programs designed by Microsoft

- Qubit operations represented as loads/stores into memory (i.e. side effect)
  ∴ disables many classical optimizations

QSSA[2]: a gate-level MLIR dialect that uses value-semantics based operations rather than memory-based qubit operations

- LLVM adheres to single static assignment (SSA), and QSSA is designed to be side-effect free
  ∴ enables decades of research in compiler optimizations to be applied to quantum compilation

XACC[3] and pulselib[4]: quantum compilation framework/library with hardware-agnostic pulse-level control

- Not as well-known as LLVM/MLIR
- Violates SSA

qe-compiler[5]: a compiler that includes an MLIR dialect of pulse-level representation

- Not technology-agnostic or hardware-agnostic

[1] QIR: https://devblogs.microsoft.com/qsharp/introducing-quantum-intermediate-representation-qir/
[2] QSSA: https://arxiv.org/abs/2109.02409
[3] XACC: https://arxiv.org/abs/2003.11971
[4] pulselib: https://arxiv.org/abs/2409.08407
[5] qe-compiler: https://arxiv.org/abs/2408.06469v1

Problem Statement

Challenges:

1. How can we include quantum and classical control on the same IR?
2. Which instructions and organization should the IR provide such that it is SSA-compliant?
3. How to design an IR that can be used by different hardware interfaces?

How can we create a unifying IR to represent pulse information for superconducting quantum computers in hybrid quantum-classical programs?

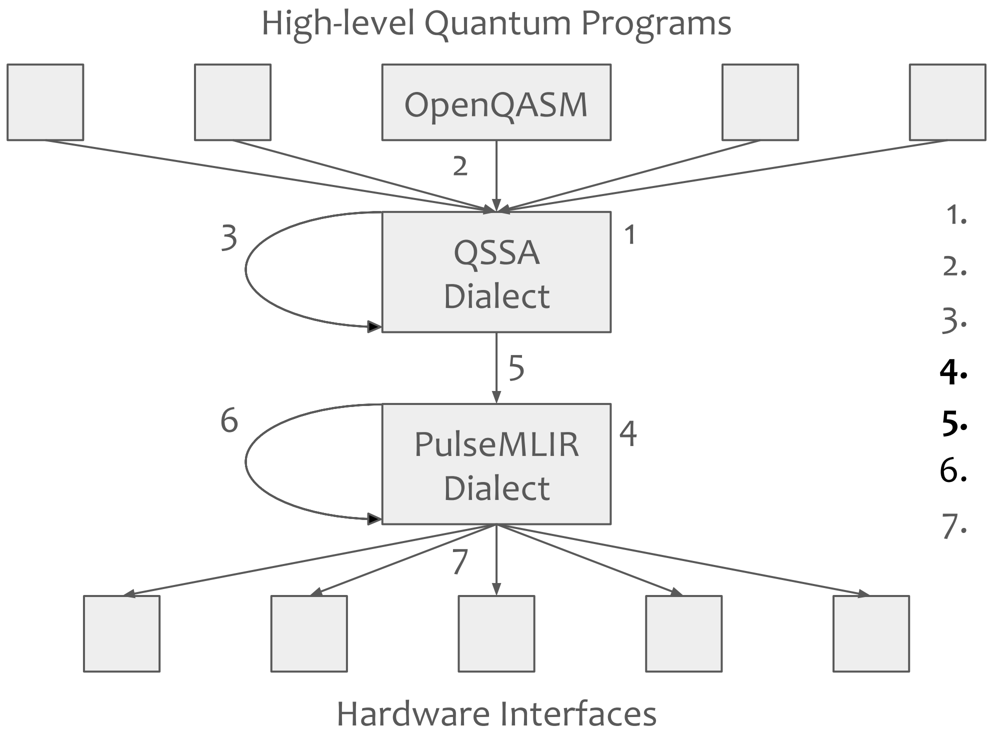
PulseMLIR: An MLIR Dialect for Pulse Representations

A unifying IR for hybrid quantum-classical programs on superconducting quantum computers

System design goals:

- Able to represent both quantum and classical programs
- Side-effect free and SSA-compliant
- Hardware-interface agnostic

Quantum Compilation Workflow



1. QSSA dialect creation
2. Lowering from high-level programs
3. Classical optimizations on QSSA
4. **PulseMLIR dialect creation**
5. **Lowering from QSSA dialect**
6. Classical optimizations on PulseMLIR
7. Lowering to hardware interfaces

PulseMLIR

Type	Description
waveform	Shape of a signal for manipulating a qubit
transmit_channel	Generic type for transmit channels
drive_channel	Transmit channel for driving qubit transitions
control_channel	Transmit channel for modulating control signals
measure_channel	Transmit channel for reading qubit states
acquire_channel	Receive channel for collecting measurement data

Operation	Operands	Returns	Description
initialize_channels	-	dc: drive_channel cc: control_channel mc: measure_channel ac: acquire_channel	Initialize channels for a single qubit
drag	duration: int sigma: int beta: float amplitude: float angle: float	wf: waveform	Create a drag waveform
gaussian_square	duration: int sigma: int width: int amplitude: float angle: float	wf: waveform	Create a Gaussian square waveform
play	waveform: waveform channel: transmit_channel	-	Play a waveform on a channel
delay	duration: int channel: transmit_channel	-	Delay for a duration on a channel
acquire	duration: int channel: transmit_channel	res: int	Acquire value into a register
shift_phase	phase: float channel: transmit_channel	-	Shift the phase of a channel
barrier	dc: drive_channel cc: control_channel mc: measure_channel ac: acquire_channel	-	Synchronize channels with delays

Gate operations supported by QSSA: CNOT, H, RX, RY, RZ, S, Sdg, T, Tdg, U, X, Y, and Z

These operations can be sufficiently represented by drag and Gaussian square waveforms

Example Transformation

```
func @qasm_main() {
        %a = qasm.allocate
        %b = qasm.allocate
        qasm.CX %a, %b
        %res = qasm.measure %a
        return
}
```

```
func @qasm_main() {
        %dc, %cc, %mc, %ac = pulse.initialize_channels(!pulse.drive_channel, !pulse.control_channel, !pulse.measure_channel, !pulse.acquire_channel)
        %dc_0, %cc_1, %mc_2, %ac_3 = pulse.initialize_channels(!pulse.drive_channel, !pulse.control_channel, !pulse.measure_channel, !pulse.acquire_channel)
        %cst = constant 1.5707963267948966 : f64
        pulse.shift_phase(%cst, %dc) : (f64, !pulse.drive_channel)
        pulse.shift_phase(%cst, %cc_1) : (f64, !pulse.control_channel)
        %c160_i32 = constant 160 : i32
        %c40_i32 = constant 40 : i32
        %cst_4 = constant 0.71712878400080549 : f64
        %cst_5 = constant 0.17973366371240579 : f64
        %cst_6 = constant -1.5707963267948968 : f64
        %0 = pulse.drag(%c160_i32, %c40_i32, %cst_4, %cst_5, %cst_6) : (i32, i32, f64, f64, f64) -> !pulse.waveform
        pulse.play(%0, %dc) : (!pulse.waveform, !pulse.drive_channel)
        %cst_7 = constant -0.66772675486091038 : f64
        %cst_8 = constant 0.084511795894186573 : f64
        %cst_9 = constant 0.00495799629351078 : f64
        %1 = pulse.drag(%c160_i32, %c40_i32, %cst_7, %cst_8, %cst_9) : (i32, i32, f64, f64, f64) -> !pulse.waveform
        pulse.play(%1, %dc_0) : (!pulse.waveform, !pulse.drive_channel)
        %c512_i32 = constant 512 : i32
        %c64_i32 = constant 64 : i32
        %c256_i32 = constant 256 : i32
        %cst_10 = constant 0.06111098055028464 : f64
        %cst_11 = constant 2.9065646597215022E-4 : f64
        %2 = pulse.gaussian_square(%c512_i32, %c64_i32, %c256_i32, %cst_10, %cst_11) : (i32, i32, i32, f64, f64) -> !pulse.waveform
        pulse.play(%2, %dc_0) : (!pulse.waveform, !pulse.drive_channel)
        %cst_12 = constant 0.39758741702842126 : f64
        %cst_13 = constant -2.1798559330788478 : f64
        %3 = pulse.gaussian_square(%c512_i32, %c64_i32, %c256_i32, %cst_12, %cst_13) : (i32, i32, i32, f64, f64) -> !pulse.waveform
        pulse.play(%3, %cc) : (!pulse.waveform, !pulse.control_channel)
        %cst_14 = constant 0.000000e+00 : f64
        %4 = pulse.drag(%c160_i32, %c40_i32, %cst_4, %cst_5, %cst_14) : (i32, i32, f64, f64, f64) -> !pulse.waveform
        pulse.play(%4, %dc) : (!pulse.waveform, !pulse.drive_channel)
        %cst_15 = constant -3.1413019971238212 : f64
        %5 = pulse.gaussian_square(%c512_i32, %c64_i32, %c256_i32, %cst_10, %cst_15) : (i32, i32, i32, f64, f64) -> !pulse.waveform
        pulse.play(%5, %dc_0) : (!pulse.waveform, !pulse.drive_channel)
        %cst_16 = constant 0.96173672051094527 : f64
        %6 = pulse.gaussian_square(%c512_i32, %c64_i32, %c256_i32, %cst_12, %cst_16) : (i32, i32, i32, f64, f64) -> !pulse.waveform
        pulse.play(%6, %cc) : (!pulse.waveform, !pulse.control_channel)
        %c1472_i32 = constant 1472 : i32
        %c64_i32_17 = constant 64 : i32
        %c1216_i32 = constant 1216 : i32
        %cst_18 = constant 2.356400e-01 : f64
        %cst_19 = constant -1.9104095842958464 : f64
        %7 = pulse.gaussian_square(%c1472_i32, %c64_i32_17, %c1216_i32, %cst_18, %cst_19) : (i32, i32, i32, f64, f64) -> !pulse.waveform
        pulse.play(%7, %mc) : (!pulse.waveform, !pulse.measure_channel)
        %c1568_i32 = constant 1568 : i32
        pulse.delay(%c1568_i32, %mc) : (i32, !pulse.measure_channel)
        %8 = pulse.acquire(%c1472_i32, %ac) : (i32, !pulse.acquire_channel)
        return
}
```

Future Work

- Extend support for additional gate operations
- Apply classical SSA optimizations on PulseMLIR
- Apply quantum-specific optimizations on PulseMLIR
- Implement transformations to lower PulseMLIR to various hardware interfaces
- Specify different dialects for other types of quantum computers, such as trapped-ion quantum computers

Summary

PulseMLIR is an MLIR pulse dialect for superconducting technology that innovates as being:

- MLIR-based dialect which:
    a. Allows reusing previously created optimizations
    b. Is implemented in a known environment making the creation of new optimizations easier
- Able to represent both quantum and classical operations
- Optimized to be both simple and complete