PulseMLIR: An MLIR Dialect for Pulse Representations in Superconducting Quantum Computers

Wonkeun Choi
TU Munich

Francisco Romão
TU Munich

Abstract

In the Noisy Intermediate-Scale Quantum (NISQ) era, quantum computing systems are constrained by limited qubit counts and high error rates. This has driven the rise of hybrid quantum-classical programs, combining classical and quantum tasks to improve computational efficiency. However, intermediate representations like Microsoft's QIR rely on memory-based semantics, which introduces side effects and disables many classical optimizations. Furthermore, gate-level programming abstractions commonly used today are limited in terms of hardware-specific optimizations. Pulse-level programming, which operates closer to the hardware, offers a path to more efficient quantum program execution but lacks a unified framework across different quantum architectures. This paper proposes PulseMLIR, an MLIR-based IR for pulse-level programming targeting superconducting quantum computers. PulseMLIR extends the existing QSSA dialect by introducing pulse-level operations, ensuring side-effect-free and SSA-compliant semantics. This approach enables classical optimizations on quantum programs and facilitates cross-platform compatibility between different pulse-generation hardware. We demonstrate the PulseMLIR workflow and outline potential future directions, including support for additional gate operations, classical and quantum optimizations, and compatibility with other quantum hardware, such as neutral atoms or trapped-ion systems. The source code is publicly available at `https://github.com/aehnh/PulseMLIR`.

1 Introduction

In the Noisy Intermediate-Scale Quantum (NISQ) era, quantum computing is characterized by low qubit counts and high error rates, making hybrid quantum-classical programs essential. These hybrid programs, which delegate tasks between quantum and classical systems, require efficient compilation pipelines. While quantum and classical parts are typically optimized separately, a unifying intermediate representation (IR) that can represent both offers new opportunities for cross-domain optimizations, improving the overall performance of hybrid programs.

At the same time, pulse-level programming provides a more direct way to control quantum hardware than traditional gate-level programming. By operating closer to the hardware, pulse-level programming allows for more aggressive optimizations. However, the current landscape of pulse-level programming is fragmented. Various high-level pulse representations exist, each tailored to specific pulse-generation hardware interfaces. A unifying IR that can translate between these diverse interfaces would create a common platform for researchers and compiler engineers, enabling optimizations and enhancing compatibility across different quantum devices.

Current solutions, such as QIR, an LLVM-based IR developed by Microsoft, represent quantum operations as memory-based loads and stores, which introduces side effects [2]. This design makes it difficult to apply many classical optimizations, which rely on side-effect-free semantics. On the other hand, QSSA, a gate-level MLIR dialect, overcomes this by using value-based semantics that eliminates side effects [5]. QSSA's single static assignment (SSA) compliance enables the use of well-established compiler optimizations, bringing decades of classical optimization techniques to quantum compilation. However, QSSA is limited to gate-level programming and does not extend to pulse-level control.

IBM Quantum Engine Compiler (qe-compiler) provides an MLIR dialect for pulse representation in superconducting quantum computers, but it lacks support for technology-agnostic or hardware-agnostic compilation, restricting its broader applicability across different quantum devices [3]. Both pulselib and XACC provide target-agnostic pulse-level IR, but neither are SSA-compliant, limiting their ability to leverage classical optimizations fully [1, 4]. This creates a gap in the quantum compilation landscape: no well-known, SSA-compliant, side-effect-free IR supports pulse-level programming for superconducting quantum computers.

This paper addresses How we can create a unifying IR to represent pulse information for superconducting quantum computers in hybrid quantum-classical programs. To answer this, we propose PulseMLIR, a unifying IR for pulse-level programming in superconducting quantum computers, built on a well-known compiler framework. PulseMLIR is side-effect-free and SSA-compliant, enabling a wide range of classical optimizations and providing a common platform for pulse-level control across different quantum devices.

2 Background

2.1 Quantum-Classical Hybrid Programs

In the NISQ era, quantum computers are constrained by their limited qubit count (typically fewer than 1000 qubits) and high error rates. As a result, quantum devices alone cannot efficiently execute large or complex computations. This necessitates using quantum-classical hybrid programs, where quantum computers handle the parts of the computation best suited to their strengths, while classical computers manage tasks better suited for classical processing.

Hybrid programs are central to many prominent NISQ algorithms, including the Variational Quantum Eigensolver (VQE), Quantum Approximate Optimization Algorithm (QAOA), Quantum Machine Learning, and Circuit Cutting and Knitting. For example, in VQE, the quantum computer prepares and measures quantum states, while a classical computer performs the parameter optimization needed to minimize the system's energy. In QAOA, the quantum part applies a series of gate operations, while the classical computer optimizes parameters to steer the evolution of the quantum state toward the desired outcome.

Another example is the execution of large quantum circuits, which cannot be executed on limited-qubit quantum hardware. These circuits are divided into smaller pieces (cutting) that quantum computers can handle. The results from the smaller circuits are later "knitted" together using classical post-processing, allowing for the simulation of larger quantum computations.

Since these algorithms' quantum and classical components are fundamentally different, they are typically compiled and optimized using separate toolchains. The quantum part is handled by quantum compilers such as Qiskit, while standard compilers like GCC or Clang process the classical part. This separation introduces challenges in optimization because the interaction between the quantum and classical parts must be coordinated during runtime.

A unifying IR that can model both quantum and classical operations presents new opportunities for cross-domain optimizations. By representing hybrid programs in a unified IR, developers can potentially reduce inefficiencies and improve performance through joint optimization strategies that are impossible in a separate compilation model.

2.2 Pulse-level Programming

Pulse-level programming in quantum computing refers to controlling the quantum hardware at a much lower level than the more common gate-level programming. In gate-level programming, quantum algorithms are abstracted into a series of quantum gates, such as the Pauli-X, Hadamard, or CNOT gates, which operate on qubits. These gates form the building blocks of quantum circuits. While this abstraction is convenient for designing algorithms and ensuring portability across different hardware platforms, it limits the ability to optimize for specific hardware characteristics.

Pulse-level programming, on the other hand, allows direct manipulation of the microwave pulses used to drive quantum operations. Rather than relying on predefined gates, users can precisely control these pulses' waveform, duration, and frequency. This closer-to-hardware control opens up opportunities for more aggressive optimizations, such as customizing pulses to minimize errors, reduce operation times, or even create entirely new types of operations tailored to the device's physical properties. As a result, pulse-level programming can offer significant performance advantages compared to gate-level programming, particularly in noise-sensitive systems typical of the NISQ era.

At a high level, pulse-level control can be abstracted into two layers: high-level pulse representations and pulse-generation hardware. High-level pulse representations are user interfaces that provide access to pulse-level programming without requiring users to work directly with hardware-specific details. Examples of such interfaces include Qiskit Pulse, Q-CTRL, Pulser, and JaqalPaw. These tools enable users to design pulses and experiment with pulse sequences at a higher level of abstraction while still benefiting from the optimization potential of pulse-level programming.

However, for the designed pulses to be executed on quantum hardware, they must be translated into signals that pulse-generation hardware can understand. This hardware includes devices like arbitrary waveform generators (AWG), arbitrary function generators (AFG), and radio frequency system-on-chip (RFSoC), which produce the electrical signals used to drive quantum operations. The challenge is that each high-level pulse representation is typically tied to a specific type of pulse-generation hardware. For instance, Qiskit Pulse can only be translated for IBM-specific quantum systems. This disconnect between high-level pulse representations and pulse generation hardware interfaces leads to fragmentation in the current pulse-level programming landscape.

To overcome this limitation, there is a growing need for a unifying IR that can standardize the translation pipeline between high-level pulse programming interfaces and diverse pulse-generation hardware. Such an IR would allow researchers to perform pulse-level optimizations without being restricted to specific hardware platforms or representations. By providing a common framework, this approach could significantly enhance the flexibility and portability of pulse-level programming across different quantum systems, driving progress in both algorithm design and hardware efficiency.

2.3 MLIR

MLIR (Multi-Level Intermediate Representation) is an extension of LLVM, a popular compiler framework that translates and optimizes code across various programming languages and hardware architectures. LLVM uses a low-level intermediate representation to enable efficient compilation for different systems, but it is primarily focused on traditional CPU and machine-level code. MLIR was developed to handle more complex compilation needs, especially for domains like machine learning and quantum computing, by supporting multiple levels of abstraction. This flexibility allows for optimizations at various stages of compilation beyond just machine code.

A key feature of MLIR is its dialect system, which defines specific operations and types tailored to different domains. Each dialect acts as a customized set of rules and structures, enabling developers to represent and optimize programs that suit their specific hardware or application. For example, a specialized dialect could represent quantum gates or pulses in quantum computing.

2.4 Static Single Assignment (SSA) Form

SSA form is a property of an IR in which every variable is assigned exactly once, and each variable is defined before it is used. In SSA form, if a variable needs to be updated or modified, a new version of that variable is created rather than overwriting the old value. This simplifies the flow of data through a program, making it easier for compilers to track variable usage and optimize the code.

SSA form is widely used in modern compilers because it enables a range of classical optimizations. For example, common optimizations applicable to SSA-compliant code include:

- Peephole optimizations look at small sections of code and replace inefficient instruction patterns with more efficient ones.
- Redundancy elimination, which removes calculations or memory accesses that are repeated unnecessarily.
- Deadcode elimination removes code that does not affect the program's output.

Because SSA makes the data flow explicit, it simplifies the process of applying these optimizations, allowing compilers to generate more efficient code.

The LLVM IR, used in the popular LLVM compiler framework, is inherently SSA-compliant, and since QIR is based on LLVM IR, it also adopts the SSA form. However, despite being SSA-compliant, QIR is unsuitable for classical optimizations like redundancy and dead code elimination. This is because QIR violates another key condition for these optimizations to be applicable, which will be discussed in the next subsection.

2.5 Value-based vs. Memory-based Semantics

For classical optimizations like redundancy elimination and dead code elimination to be applicable, the intermediate representation (IR) must satisfy an additional condition: it needs to use value-based semantics rather than memory-based semantics. In value-based semantics, data is passed directly between operations as explicit values, avoiding the need for loads and stores into memory. This simplifies the data flow and eliminates side effects, making it easier for the compiler to apply optimizations.

In the case of QIR, while it is SSA-compliant, it uses memory-based semantics for handling qubits. Specifically, qubit operations in QIR are represented as loads and stores into a special "qubit memory," as illustrated in the following code snippet:

```
%0 = call i8* @qntmGep(%Array* %qs, i64 0)
%1 = bitcast i8* %0 to %Qubit**
%2 = load %Qubit*, %Qubit** %1, align 8
%3 = call i8* @qntmGep(%Array* %qs, i64 1)
%4 = bitcast i8* %3 to %Qubit**
%5 = load %Qubit*, %Qubit** %4, align 8
call void @qntmIntrinsicCNOT(%Qubit* %2, %Qubit* %5)
```

In this representation, each instruction has a side effect—modifying the qubit memory—which prevents most classical optimizations from being applied. Because the contents of the qubit memory are treated as a "black box," the compiler cannot make useful inferences about them. As a result, even though QIR is SSA-compliant, its reliance on memory-based semantics disables key optimizations.

MLIR, however, allows for the conversion of this memory-based representation into value-based semantics, where the data is passed directly between operations without involving loads and stores. This makes the code side-effect-free, enabling the compiler to apply classical optimizations more easily. For example, QSSA represents the QIR code snippet above in a value-based form as:

```
%c1, %t1 = qssa.CNOT %c, %t
```

This simplified representation eliminates the need for loads and stores, making the code more amenable to classical optimizations. In a similar fashion, the MLIR dialect we present leverages value-based semantics, making quantum operations side-effect-free. This design allows for the application of decades of research in classical compiler optimizations, potentially improving the efficiency and performance of quantum programs.

3 Overview

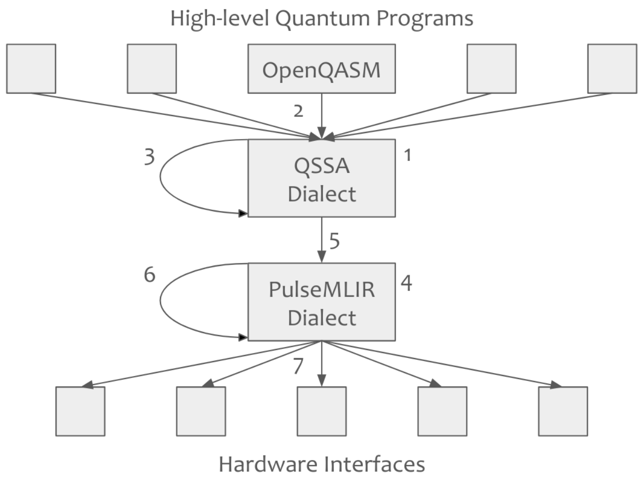

Figure 1: PulseMLIR compilation workflow

Figure 1 illustrates the PulseMLIR compilation workflow. PulseMLIR is built on top of QSSA, an SSA-compliant, side-effect-free MLIR dialect designed for quantum gate-level programs. The workflow is structured into a series of steps, labeled 1 through 9, which represent the following processes:

1. QSSA dialect creation
2. Transformation from high-level quantum programs down to QSSA dialect
3. Classical optimizations on the QSSA dialect

Next, the workflow continues with steps 4 through 5, which we focus on in this paper:

4. PulseMLIR dialect creation
5. Transformation from QSSA dialect down to PulseMLIR dialect

Steps 6 and 7 remain as directions for future work:

6. Classical and quantum optimizations on the PulseMLIR dialect
7. Transformation from PulseMLIR Dialect down to hardware interfaces

4 Design and Implementation

The PulseMLIR dialect is designed to represent pulse-level operations for superconducting quantum computers, as shown in Table 1. The dialect includes both types and operations essential for mapping higher-level quantum operations to pulse-level instructions.

The PulseMLIR dialect is lowered from the QSSA dialect, which only explicitly supports the following gate operations: CNOT, H, RX, RY, RZ, S, Sdg, T, Tdg, U, X, Y, and Z. Since these are the only gate operations supported, the PulseMLIR dialect is designed to provide corresponding pulse-level representations for each.

To determine the pulse representation of each gate operation, we generated the pulse sequences using Qiskit Pulse with optimizations disabled. This approach allowed us to closely match the pulse-level hardware instructions required for each gate.

In the current implementation, only two waveform types – *DRAG* and *Gaussian_square* – are provided, as they are sufficient to represent the supported gate operations. The drag waveform is used to handle qubit leakage errors, while the Gaussian square waveform serves broader purposes. New waveform types will be introduced whenever new gate operations requiring different pulse representations are added.

5 Future Work

One immediate area for future work is to extend the PulseMLIR dialect with additional gate operations and waveform types. We have implemented only a limited set of operations, including Gaussian-squared and DRAG waveforms, as these were the waveforms required for the gate operations currently implemented in QSSA. However, as we extend the compilation pipeline to support new gate operations, those operations must first be declared in the QSSA dialect. Subsequently, the transformations from OpenQASM to QSSA and then to the PulseMLIR dialect need to be implemented. This process will also require adding new waveforms to PulseMLIR as operations, depending on the needs of the additional gate operations. Extending the dialect to handle more diverse quantum operations will improve the flexibility and usefulness of PulseMLIR across a broader set of quantum applications.

Another task would be to implement classical SSA optimizations on the PulseMLIR dialect. Although the PulseMLIR dialect is SSA-compliant and side-effect-free, the benefits of these features can only be realized through classical optimizations. Optimizations such as inlining, loop-unrolling, and dead code elimination have the potential to significantly improve the performance and efficiency of quantum-classical hybrid programs compiled using PulseMLIR. These classical optimization techniques are well-established in traditional compilers and can be adapted to work within the 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

Table 1: Types and operations of PulseMLIR dialect

framework, allowing for a more effective use of hardware resources.

In addition to classical optimizations, we aim to implement quantum-specific optimizations on the PulseMLIR dialect. One of the key strengths of our approach is that it can apply both classical and quantum optimizations, giving it an advantage over pure quantum compilers, which lack this flexibility. Quantum optimizations, such as qubit reuse, gate commutation, and error mitigation techniques, are crucial for enhancing the fidelity and performance of quantum circuits. By integrating these quantum-specific optimizations into PulseMLIR, we can further improve the quality of the compiled quantum programs, making them more suitable for real-world quantum hardware with limited qubits and high error rates.

Another high-priority task is to write hardware interfaces to transform the PulseMLIR dialect into various pulse-generation hardware. So far, we have defined the PulseMLIR dialect, but the transformations that convert PulseMLIR instructions into signals that can be understood by specific hardware interfaces—such as arbitrary waveform generators (AWGs) or radio frequency system-on-chips (RFSoCs)—are still missing. These transformations are crucial for enabling the use of PulseMLIR across different quantum devices. The ability to target multiple hardware platforms is a key selling point of this unifying IR, so implementing these transformations is essential for making PulseMLIR a practical and versatile tool for pulse-level programming.

Finally, transformations from QSSA to PulseMLIR for other types of quantum computers, such as trapped-ion quantum computers, need to be created. While PulseMLIR was initially designed for superconducting quantum computers, it should be adaptable for other architectures that use pulses to manipulate qubits. However, the pulse representation of quantum gate operations for trapped-ion systems, for example, differs from that of superconducting systems. Thus, separate transformations will need to be implemented to translate gate operations from QSSA into the appropriate pulse representation for each type of quantum computer. By supporting multiple quantum technologies, PulseMLIR can become a more universal quantum compilation tool applicable to a broader range of hardware platforms.

6 Summary

In this work, we introduced PulseMLIR, a new intermediate representation for pulse-level programming of superconducting quantum computers built on the MLIR framework. Our approach extends the capabilities of existing gate-level dialects such as QSSA by providing fine-grained control over hardware-specific pulse sequences while maintaining SSA-compliance and side-effect-free semantics. This enables the application of classical compiler optimizations, which are typically unavailable in memory-based quantum IRs such as QIR. While we have implemented a core set of operations, including Gaussian-squared and DRAG waveforms, the framework is designed to be extensible, allowing future support for additional gate operations and waveform types.

References

[1] Aniket S. Dalvi, Leon Riesebos, Jacob Whitlow, and Kenneth R. Brown. Graph-based pulse representation for diverse quantum control hardware, 2024.

[2] Alan Geller. Introducing quantum intermediate representation (qir), 2020. https://devblogs.microsoft.com/qsharp/introducing-quantum-intermediate-representation-qir/.

[3] Michael B. Healy, Reza Jokar, Soolu Thomas, Vincent R. Pascuzzi, Kit Barton, Thomas A. Alexander, Roy Elkabetz, Brian C. Donovan, Hiroshi Horii, and Marius Hillenbrand. Design and architecture of the ibm quantum engine compiler, 2024.

[4] Thien Nguyen and Alexander McCaskey. Enabling pulse-level programming, compilation, and execution in xacc, 2020.

[5] Anurudh Peduri, Siddharth Bhat, and Tobias Grosser. Qssa: an ssa-based ir for quantum computing. In *Proceedings of the 31st ACM SIGPLAN International Conference on Compiler Construction*, CC '22. ACM, March 2022.