DIGITAL LIBRARY ACM DL

Association for Computing Machinery

acm open

Latest updates: https://dl.acm.org/doi/10.1145/3731599.3767552

RESEARCH-ARTICLE

Tackling the Challenges of Adding Pulse-level Support to a Heterogeneous HPCQC Software Stack: MQSS Pulse

JORGE ECHAVARRIA, Leibniz Supercomputing Center Munich, Garching bei Munchen, Bayern, Germany

MUHAMMAD NUFAIL FAROOQI, Leibniz Supercomputing Center Munich, Garching bei Munchen, Bayern, Germany

AMIT DEVRA, Technical University of Munich, Munich, Bayern, Germany

SANTANA LUJAN, German Aerospace Center, Koln, Nordrhein-Westfalen, Germany

LÉO VAN DAMME, Technical University of Munich, Munich, Bayern, Germany

HOSSAM AHMED, Leibniz Supercomputing Center Munich, Garching bei Munchen, Bayern, Germany

View all

Open Access Support provided by:

Technical University of Munich

IQM Quantum Computers

Alpine Quantum Technologies GmbH

PlanQC GmbH

Leibniz Supercomputing Center Munich

Argonne National Laboratory

View all

Tackling the Challenges of Adding Pulse-level Support to a Heterogeneous HPCQC Software Stack

MQSS Pulse

Jorge Echavarria
Leibniz Supercomputing Centre (LRZ)
Garching, Germany
jorge.echavarria@lrz.de

Muhammad Nufail Farooqi
Leibniz Supercomputing Centre (LRZ)
Garching, Germany
muhammad.farooqi@lrz.de

Amit Devra
Technical University of Munich
(TUM)
Munich, Germany
amit.devra@tum.de

Santana Lujan
German Aerospace Center (DLR)
Weßling, Germany
santana.lujan@dlr.de

Léo Van Damme
Technical University of Munich
(TUM)
Munich, Germany
leo.van-damme@tum.de

Hossam Ahmed
Leibniz Supercomputing Centre (LRZ)
Garching, Germany
hossam.ahmed@lrz.de

Martín Letras
Leibniz Supercomputing Centre (LRZ)
Garching, Germany
martin.letras@lrz.de

Ercüment Kaya
Leibniz Supercomputing Centre (LRZ)
Garching, Germany
ercuement.kaya@lrz.de

Adrian Vetter
planqc GmbH
Garching, Germany
adrian@planqc.eu

Max Werninghaus
Walther-Meißner-Institut (WMI)
Garching, Germany
max.werninghaus@wmi.badw.de

Martin Knudsen
Walther-Meißner-Institut (WMI)
Garching, Germany
martin.knudsen@wmi.badw.de

Felix Rohde
Alpine Quantum Technologies GmbH
(AQT)
Innsbruck, Austria
felix.rohde@aqt.eu

Albert Frisch
Alpine Quantum Technologies GmbH
(AQT)
Innsbruck, Austria
albert.frisch@aqt.eu

Eric Mansfield
IQM Quantum Computers
Munich, Germany
eric.mansfield@meetiqm.com

Rakhim Davletkaliyev
IQM Quantum Computers
Helsinki, Finland
rakhim.davletkaliyev@meetiqm.com

Vladimir Kukushkin
IQM Quantum Computers
Helsinki, Finland
vladimir.kukushkin@meetiqm.com

Noora Färkkilä
IQM Quantum Computers
Helsinki, Finland
noora.farkkila@meetiqm.com

Janne Mäntylä
IQM Quantum Computers
Helsinki, Finland
jmantyla@meetiqm.com

Nikolas Pomplun
German Aerospace Center (DLR)
Weßling, Germany
nikolas.pomplun@dlr.de

Andreas Spörl
German Aerospace Center (DLR)
Weßling, Germany
andreas.spoerl@dlr.de

Lukas Burgholzer
Technical University of Munich
(TUM)
Munich, Germany
Munich Quantum Software Company
GmbH (MQSC)
Garching, Germany
lukas.burgholzer@tum.de

Yannick Stade
Technical University of Munich
(TUM)
Munich, Germany
yannick.stade@tum.de

Robert Wille
Technical University of Munich
(TUM)
Munich, Germany
Munich Quantum Software Company
GmbH (MQSC)
Garching, Germany
robert.wille@tum.de

Laura B. Schulz
Argonne National Laboratory (ANL)
Lemont, IL, USA
schulz@anl.gov

Martin Schulz
Leibniz Supercomputing Centre (LRZ)
Garching, Germany
Technical University of Munich
(TUM)
Garching, Germany
martin.schulz@lrz.de

Abstract

We study the problem of adding native pulse-level control to heterogeneous High Performance Computing-Quantum Computing (HPCQC) software stacks, using the Munich Quantum Software Stack (MQSS) as a case study. The goal is to expand the capabilities of HPCQC environments by offering the ability for low-level access and control, currently typically not foreseen for such hybrid systems. For this, we need to establish new interfaces that integrate such pulse-level control into the lower layers of the software stack, including the need for proper representation.

Pulse-level quantum programs can be fully described with only three low-level abstractions: *ports* (input/output channels), *frames* (reference signals), and *waveforms* (pulse envelopes). We identify four key challenges to represent those pulse abstractions at: the user-interface level, at the compiler level (including the Intermediate Representation (IR)), and at the backend-interface level (including the appropriate exchange format). For each challenge, we propose concrete solutions in the context of MQSS. These include introducing a compiled (C/C++) pulse Application Programming Interface (API) to overcome Python runtime overhead, extending its LLVM support to include pulse-related instructions, using its C-based backend interface to query relevant hardware constraints, and designing a portable exchange format for pulse sequences. Our integrated approach provides an end-to-end path for pulse-aware compilation and runtime execution in HPCQC environments. This work lays out the architectural blueprint for extending HPCQC integration to support pulse-level quantum operations without disrupting state-of-the-art classical workflows.

CCS Concepts

• **Software and its engineering** → **Just-in-time compilers**; **Runtime environments**; **Formal language definitions**; **General programming languages**; • **Computer systems organization** → **Quantum computing**; • **Hardware** → **Quantum technologies**.

Keywords

HPCQC, JIT Compilation, Pulse-level Control, MLIR, QDMI, MQSS

1 Introduction

The convergence of classical High Performance Computing (HPC) and emerging Quantum Computing (QC) technology into a unified software stack presents unique opportunities—and challenges—for scientific applications. Classical HPC infrastructures offer mature tooling for orchestration, data movement, and fault-tolerance while delivering high performance on classical algorithms. Quantum hardware adds to this computational strength with novel computational capabilities for a specific class of quantum algorithms, implemented through coherent control of qubit systems to exploit their special properties, but in turn has to rely on the HPC infrastructure for seamless integration and operation. This includes the execution of the quantum software stack in general, and quantum compilers in particular. However, these have primarily focused on gate-level abstractions, in particular in the context of High Performance Computing-Quantum Computing (HPCQC), which may obscure the

potentially rich, low-level dynamics accessible on modern quantum devices, making them inaccessible to hybrid workflows.

Recent studies similarly emphasize the need to improve low-level quantum control to realize the full potential of quantum computing. Smith et al. [43], for example, point out that, in addition to better qubit fabrication and algorithms, scaling quantum devices toward fault-tolerance requires refined device-level control. This reinforces the consensus that enhancing hardware control mechanisms is critical as quantum systems grow in scale, and that this control must be available in modern QC software stacks beyond device-level experimental access: a quantum software stack must support it natively and across its entire functionality to enable effective HPCQC integration.

Providing direct pulse-level interfaces, for example, allows high-level classical orchestration to couple seamlessly with the hardware control layer. For instance, Delgado and Date [9] describe hybrid HPCQC workflows in which classical optimization algorithms compute optimal pulse sequences to steer quantum operations. In this way, exposing pulse-level control in the software stack effectively bridges the abstraction gap between classical orchestration and the low-level execution of quantum control pulses.

In this paper, we identify and address the key challenges of integrating this needed pulse-level support into a scalable, portable framework that spans both classical and quantum systems. We use Munich Quantum Valley (MQV)'s Munich Quantum Software Stack (MQSS) [6, 39] as a case study to demonstrate our approach.

2 Introducing Pulse-level Access

Pulse-level access enables direct control of the mechanisms used to control the qubits on the respective medium for the chosen quantum technology. This can have different physical manifestations (e.g., via microwaves or lasers), but most cases is the result of translating (higher-level) gate descriptions into some kind of pulse description.

2.1 Why is Pulse-level Helpful?

Low-level control of these pulses enables a more fine-grained control of the glsqc system, which can have several usage scenarios, including auotmated calibration and optimal control.

Automated Calibration: Calibration is the systematic, continuous, and iterative process of measuring and compensating for various sources of physical and control errors to ensure that the physical operations performed on qubits match their intended logical definitions. Establishing a unified abstraction layer that provides low-level access to diverse quantum hardware platforms for pulse-level control operations enables the management and scheduling of calibration routines. This allows QC service providers, like HPC centers, to monitor system usage patterns and dynamically schedule calibrations based on anticipated demand. This capability enables resource-aware calibration planning, which in turn helps tune the quantum hardware toward fidelity levels that meet operational requirements and align with user workloads as well as any additional priority criteria.

Note that automated calibration routines do not include the calibration of the broader QC environment (e.g., cryogenics, vacuum systems, or laser alignment) nor the physical initialization of qubits into a stable, ready state. Instead, they focus on fine-tuning system
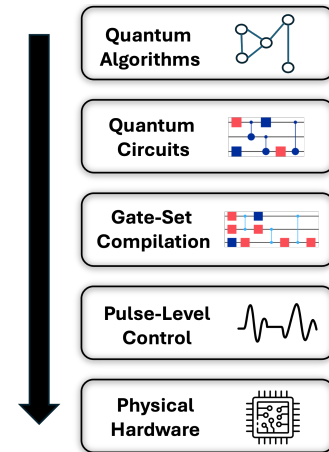

Figure 1: A top-down approach to QC: tracing the flow from quantum algorithms and their circuit representations to pulse-level control, i.e., electromagnetic waveforms on target hardware.

parameters such as control pulse amplitudes, durations, frequencies, and phase alignments to optimize gate fidelity and readout performance. These calibration procedures are platform-specific in implementation, but conceptually applicable across multiple QC technologies, including superconducting qubits, neutral atoms, and trapped ions.

For *superconducting qubits*, one of the parameters that demands frequent calibration is the qubit transition frequency, as it can drift on timescales of minutes to hours, therefore requiring continuous real-time tracking via Ramsey-based feedback loops, to ensure the accuracy of the microwave control pulses [4]. For *trapped ions systems*, a primary concern is the stability of the electromagnetic trap, with the motional modes frequencies experiencing hour-to-hour drifts of a few hundred hertz and day-to-day drifts of several kilohertz, requiring calibration on these respective schedules [25]. *Neutral atom systems* are dominated by the stability of their laser control systems and the physical integrity of the atom array, which requires calibration of parameters on a minute timescale [45].

In the Noisy Intermediate-Scale Quantum (NISQ) era, the calibration process is not merely for improving fidelity, but an essential prerequisite for the operation of the quantum accelerator. Even Quantum Error Correction (QEC) relies heavily on the physical error rates of the underlying physical components to be under a critical threshold of almost 99% fidelity for single and two-qubit gates to implement a surface code [35].

Pulse Engineering using Optimal-Control: Designing high-fidelity quantum gates for specific hardware platforms often relies on optimal control techniques to shape control pulses that precisely manipulate qubit dynamics while reducing the impact of errors [2, 17, 22]. These pulses are typically engineered to be robust against experimental noise, such as amplitude fluctuations and frequency detuning, which are common in quantum hardware [7, 23, 32, 33, 36]. In *open-loop* control, pulses are designed offline by simulating the dynamics under a Hamiltonian describing a quantum system, using optimization algorithms such as Gradient Ascent Pulse Engineering (GRAPE) [21]. While these methods can be highly effective, they rely on precise knowledge of the system Hamiltonian and may perform poorly if the model does
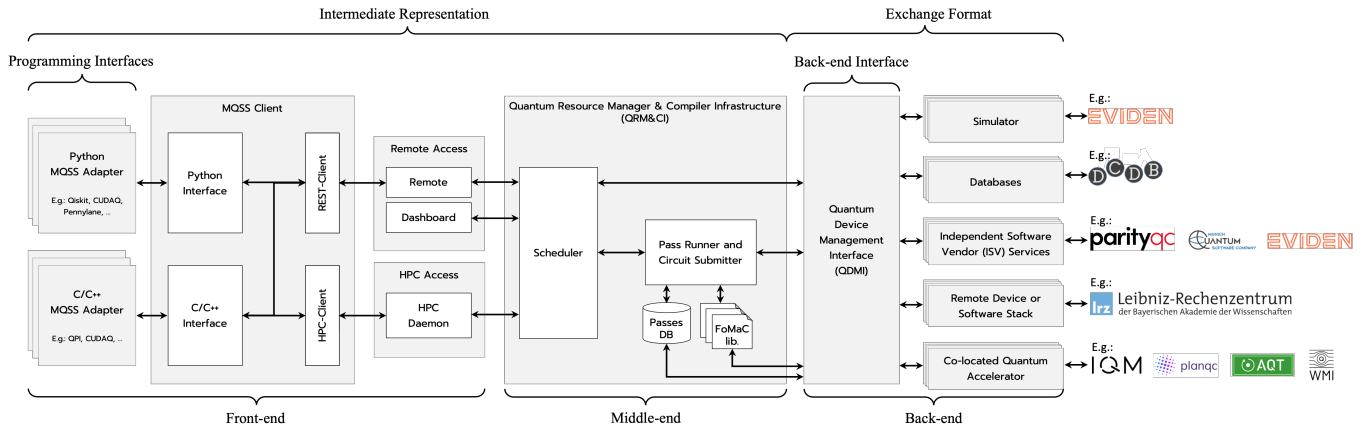
Figure 2: MQSS architecture overview: *MQSS Adapters* **(e.g., Qiskit, CUDAQ, PennyLane, and its native C-based QPI) submit gate- and pulse-based jobs to the** *MQSS Client,* **which handles automatic routing for both local HPC jobs and remote submissions. The** *Quantum Resource Manager & Compiler Infrastructure* **(QRM&CI) encompasses MQSS's second-level scheduler, its JIT LLVM-based compiler, and supporting libraries.** *QDMI* **exposes device capabilities (***ports, frames, waveforms,*** timing/granularity and constraints) to** *QRM&CI* **during JIT compilation and to the** *MQSS Adapters* **via the** *MQSS Client* **during runtime execution. Example** *QDMI Devices* **shown for illustrating target diversity: classical simulators, databases, ISVs, data centers, and superconducting, neutral atom, and trapped-ion quantum accelerators.**

not accurately reflect the true Hamiltonian governing the system dynamics [10, 14]. In contrast, *closed-loop* control incorporates experimental feedback to iteratively refine pulse parameters based on measured fidelities or system responses, enhancing robustness to hardware imperfections and unmodeled noise [15, 49]. A fundamental challenge in closed-loop quantum control is the inability to perform real-time feedback, as quantum measurements irreversibly collapse the system's state. As a result, optimizing control parameters typically requires a large number of repeated experiments, making the process resource-intensive. A hybrid approach, which combines open-loop pulse design with closed-loop calibration, is increasingly adopted for achieving near-optimal control on NISQ devices [3, 37, 38].

Pulse-level VQEs: Variational Quantum Eigensolvers (VQEs) are widely used hybrid quantum-classical algorithms for estimating ground state energies, but they face serious challenges in the NISQ era. These include decoherence from deep circuits, high classical overhead from frequent recompilation of parameterized Ansätze, and poor trainability due to barren plateaus, where the optimization landscape becomes exponentially flat as system size increases [27].

An emerging alternative is ctrl-VQE [11, 28], a pulse-level approach that bypasses traditional gate decomposition and instead optimizes the continuous control waveforms applied to the qubits. This can significantly reduce total circuit duration, thereby mitigating the impact of decoherence, and decrease the energy estimation error compared to gate-based methods. Unlike digital VQE, ctrl-VQE allows the variational optimizer to exploit the full analog nature of the hardware, including access to higher energy states in superconducting qubits (e.g., the $|2\rangle$ level in transmons), which can assist in faster state preparation and enhanced expressivity [26, 40]. These techniques are inspired by and connected to developments in quantum optimal control, where gradient-based pulse shaping is used to achieve high-fidelity target state preparation under physical constraints [16].

Although ctrl-VQE does not overcome all challenges on the path to quantum advantage, it represents a paradigm shift towards hardware-native quantum algorithm design and may extend to

other variational algorithms, such as Quantum Neural Networks (QNNs), especially in scenarios where shorter coherence times and pulse-level customization are critical.

2.2 Gaining Access to Pulse-level Programming

Recently, QC has seen rapid advancements, with several hardware platforms emerging as viable candidates, each with distinct advantages and limitations. As gate fidelities continue to improve, for approaching and even surpassing error-correction thresholds, pulse-level control is becoming increasingly crucial [1, 43]. It provides fine-grained access to the physical behavior of the qubits, allowing users to optimize performance beyond the limitations of gate-level abstraction. The top-down approach to QC shown in Fig. 1 underscores this point by highlighting the critical positioning of pulse-level control in the stack.

While gate-based programming simplifies algorithm development, it can obscure low-level imperfections such as crosstalk, decoherence, and calibration drift. Pulse-level access enables the implementation of a wide range of strategies from the field of quantum optimal control [2, 17, 22]. These include enhancing resilience to experimental imperfections through shaped pulses [23, 36, 51], applying dynamical decoupling techniques [13], achieving fast and high-fidelity gates [12, 19, 47], and leveraging machine learning–based pulse engineering [5, 37]. Pulse-level access also enables hardware-aware quantum algorithm development, which is essential for improving fidelity in near-term quantum devices. As quantum systems scale, pulse-level control will be critical for maximizing the capabilities of increasingly larger QPUs [8, 28, 41].

3 Case Study: Munich Quantum Software Stack

MQSS is an open-source [31], *runtime and compilation* full quantum software stack developed as part of the MQV initiative [6, 30, 39]. We chose MQSS, as it is openly available on GitHub, accessible to the broader research community, actively maintained via public repositories, as well as already clearly defined hardware/system interface that we can build upon [31]. As shown in Fig. 2, MQSS provides a

modular and extendable infrastructure for hybrid HPCQC workflows, offering various programming interfaces, compiler pipelines, and runtime integration layers.

Enabling pulse-level control in MQSS involves extending each layer of the stack to recognize and process pulse abstractions. At the front end, existing *programming interfaces* must be enhanced to accept pulse descriptions alongside traditional gate-based payloads. In the compiler, new Multi-Level Intermediate Representation (MLIR) pass pipelines[1] will lower gate-based dialects—e.g., Xanadu's Catalyst or NVIDIA's Quake—into a pulse-oriented dialect—e.g., IBM's *pulse* dialect or a custom equivalent—and subsequently transform that dialect into the proposed pulse *exchange format*. Finally, its novel *Quantum Device Management Interface* (*QDMI*) [50] must be updated to expose and manage the same pulse abstractions used by the *MQSS Adapters* and compiler passes.

4 Challenges of Adding Pulse-level Support

Built on top of the LLVM framework, MQSS already provides an extensible multi-dialect compiler infrastructure [24]. However, enabling pulse-level support requires further modifications of the following components:

- **Programming Interfaces**: Both human and automated tools need to have a mechanism to submit pulse-level jobs to the software stack. Supporting pulse-level payloads requires updates to both the *MQSS Adapters* and the *MQSS Client*.
- **Intermediate Representation**: A pulse-oriented MLIR dialect must be developed or adopted to enable the MQSS compiler to support LLVM compiler passes representing transformations at pulse level.
- **Backend Interface**: MQSS must be able to query quantum accelerators regarding their supported pulse implementations. The response will inform 1) end-user tools or automated systems via the *MQSS Client*—or any other arbitrary REST-like Application Programming Interface (API)—2) the compiler for translation and optimization decisions, and 3) runtime or Just-In-Time (JIT) compilation stages, enabling supportive tooling during compilation and execution.
- **Exchange Format**: The software stack must establish a mutual understanding with quantum accelerators regarding pulse-based payloads. The *QDMI* specification is flexible about the types of payloads it can support, however, we do need to make sure that at least one suitable format exists.

To achieve consistent pulse support across the stack's front-end, middle-end, and back-end, all components must share a precise, common definition of what constitutes a "pulse". We identify three essential abstractions:

- **Ports**: A software representation of the hardware input and output channels used to manipulate and read out qubits. It exposes vendor-defined actuation knobs for targeting user-accessible hardware components, such as drive or acquisition channels, while abstracting away device-specific complexity.
- **Waveforms**: A time-ordered array of samples, defining the amplitude envelope of a control signal. The amplitudes can

be provided either explicitly or by parametrized functions which, when assigned with specific parameter values, evaluate to a concrete array of samples.

- **Frames**: Stateful timing and carrier signal abstraction combining a reference clock, carrier frequency, and phase. It tracks the elapsed time and provides the timing, frequency, and phase context for playing *waveforms*, enabling precise carrier modulation and virtual phase rotations.

That is, in this paper we treat pulses as control signals with a shape defined by a *waveform*, modulated and timed with a carrier defined by a *frame*, and played on a device component defined by a *port*.

5 Tackling the Challenges

By systematically addressing the four challenges identified in Section 4, a full quantum software stack, such as the MQSS, can natively orchestrate pulse-level control across diverse platforms—including ion-trap, neutral-atom, and superconducting systems—thereby preparing the stack for future JIT compilation workflows and hardware-informed pulse-level optimizations.

In the following, each challenge is examined in its own subsection—programming interfaces, Intermediate Representation (IR), backend interface, and exchange format—where we detail our strategy for integrating pulse-level capabilities into the MQSS and our support of the above abstractions. Collectively, these efforts establish a coherent framework for extending the stack toward comprehensive quantum control.

5.1 Programming Interfaces

Fig. 2 shows the *MQSS Client*, which orchestrates quantum jobs via so-called *MQSS Adapters* such as, for example, Qiskit, CUDAQ, PennyLane, and its native *Quantum Programming Interface* (*QPI*)—a lightweight C-based library designed for HPCQC integration [20]. Because C/C++ are the dominant languages in HPC (powering, for example, CUDA and OpenMP), and because C implementations far less overhead compared to a scriping language like Python, we focus on C intefaces as the choice for large-scale HPCQC workloads. In practice, an HPC application invokes *QPI* C functions (embedded in the *MQSS QPI Adapter*) to construct and dispatch quantum programs/kernels. The *QPI* library compiles circuits into either an LLVM IR (e.g., Quantum Intermediate Representation (QIR)) or an MLIR dialect (e.g., NVIDIA's Quake, Xanadu's Catalyst, or IBM's pulse), and sends them via the *MQSS Client* to target a quantum accelerator. In this way, *QPI* can be enabled to submit HPCQC jobs using any kind of abstraction , including both gate- and potentially pulse-based abstractions, to local or remote quantum hardware through the MQSS framework.

To add pulse-level control to the enabled abstractions, we extend the *MQSS QPI Adapter* with constructs for *ports*, *waveforms*, and *frames*, the three pulse abstractions in our design. Listing 1 shows a simple quantum kernel defined in this extended *QPI* embodying the *pulse-level VQE* use case from Section 2.1 within function `pulse_vqe_quantum_kernel`. Inside an HPC loop *QPI* constructs and plays parameterized *waveforms*, collects measurements for energy estimation, and returns control to the classical optimizer for the next iteration. The snippet begins like a gate-based circuit before defining three *waveforms* (`qWaveform`) from input

---

[1]Note that additional pulse-specific transformations and optimizations can be added incrementally, just as one would extend a conventional LLVM-based gate-level compiler, such as the MQSS's [24, 46].

```
1  #include <mqss_qpi.h>
2  void pulse_vqe_quantum_kernel(void *results, int nshots,
       ↪Parameters *p) {
3      QCircuit circuit;
4      qCircuitBegin(&circuit)
5      QClassicalRegisters cr;
6      qInitClassicalRegisters(&cr, 2);
7      // we begin with X on both qubits
8      qX(0);
9      qX(1);
10     // define the waveforms
11     qWaveform(&waveform_1, p->amps_1);
12     qWaveform(&waveform_2, p->amps_2);
13     qWaveform(&waveform_3, p->amps_3);
14     // apply the waves
15     qPlayWaveform(qb1_drive_port, waveform_1);
16     qPlayWaveform(qb2_drive_port, waveform_2);
17     // do the frame changes
18     qFrameChange(qb1_drive_port, freq_qb1, p->phase_qb1);
19     qFrameChange(qb2_drive_port, freq_qb2, p->phase_qb2);
20     // apply the entangling pulse
21     qPlayWaveform(qb1_qb2_coupler_port, waveform_3);
22     // measure
23     qMeasure(0, 0);
24     qMeasure(1, 1);
25     qCircuitEnd();
26     if(!qExecute(dev, circuit, nshots))
27         QuantumResult* results = qRead(circuit);
28
29     qCircuitFree(circuit);
30 }
31 int main(){
32     do{
33         void* results = malloc(size);
34         pulse_vqe_quantum_kernel(&results, nshots, &parameters);
35         parameters = calculate_new_parameters(&results, parameters)
36     }while( stop_condition == false );
37     return 0;
38 }
```

Listing 1: Exemplary implementation of a simplified Hardware-efficient *Ansatz* [48], showcasing the proposed extension on pulse-level control to the *MQSS QPI Adapter* [20]. Note that the definition of the calculate_new_parameters routine is omitted here, its intended purpose is to represent a computationally expensive classical optimization routine. This listing is illustrative and shows only a subset of the proposed functionality. Listings 2 and 3 are intended to be equivalent representations of the same quantum kernel.

amplitudes, playing pulses on specific *ports* (qPlayWaveform), applying *frame* changes (qFrameChange), and finally measuring the results (qMeasure). The new three *QPI* primitives (i.e., qWaveform, qPlayWaveform, and qFrameChange) operate at native speed due to its C implementation. In detail:

- **qWaveform(waveform, amps)** creates a *waveform* object from given amplitudes (and, implicitly, duration/envelope) to use for subsequent pulses.
- **qPlayWaveform(port, waveform)** emits the specified waveform on a hardware *port* (e.g., a qubit drive or coupler channel), physically delivering the pulse to that qubit.
- **qFrameChange(port, frequency, phase)** adjusts the carrier *frame* of a qubit *port*, setting its drive frequency and phase offset for precise control.

These extensions satisfy our HPC-focused design requirements: by implementing them in C, we keep latency low and resource usage minimal, addressing the performance and integration challenges identified earlier. Importantly, our approach remains compatible

with other pulse-level APIs. For example, IBM has deprecated pulse-level support in Qiskit in early 2025, underscoring the need for alternative interfaces. In contrast, the *MQSS QPI Adapter* is expected to continue to support full analog control in HPC environments. Moreover, the *QPI* abstractions align closely with industry standards: Amazon Braket's Pulse feature, for example, exposes *ports*, *frames*, and *waveforms* through its SDK (including OpenQASM 3.0 support), while the OpenQASM 3.0 specification defines calibration (cal) blocks that explicitly use the same three abstractions. In practice, this means that a *QPI* pulse program could be translated or interfaced with Braket- or OpenQASM3-style schedules, if desired.

5.2 Intermediate Representation

As described in Section 3, the *MQSS Compiler* is fully based on LLVM-IR [46] and LLVM-MLIR [24], where all gate-based quantum circuit transformations are implemented as either QIR or MLIR (e.g., NVIDIA's Quake and Xanadu's Catalyst) passes. LLVM's built-in pass manager supports MLIR dialect-agnostic orchestration by allowing both operation-specific and operation-agnostic passes to be registered and executed on IR modules, regardless of the dialect they belong to—as long as the pass is targeted to the correct dialect context. Thus, any MLIR job loaded into memory can be processed by a pass suite appropriate for its dialect—simplifying support for new domains like pulse-level control. This design allows MQSS to handle multiple dialects—existing or future—without fundamental changes to the pass orchestration system.

For example, IBM's Quantum Engine defines a Pulse MLIR dialect, where each high-level gate or measurement is lowered into a pulse sequence via provided "calibration" *waveforms* [18]. In this dialect, every gate has an associated pulse *waveform*. Listing 2 illustrates an example with this MLIR dialect. Three *waveforms* are defined with pulse.def@waveform_*. The @pulse_vqe_quantum_kernel sequence then applies pulse.standard_x gates (X on each qubit), followed by pulse.play calls that apply the predefined *waveforms* on drive and coupler *frames*. It also utilizes pulse.frame_change to adjust phases/frequencies, and ends with the measurement: a readout pulse.play on each qubit followed by pulse.capture, returning the classical bits. As seen, LLVM-based quantum compilers, such as the one in MQSS, can natively support gate-level operations and pulse instructions even in one single IR.

This MLIR dialect is compatible with the three key abstractions—*ports*, *frames*, and *waveforms*—as MLIR constructs aligned with the three core definitions we introduced in Section 4. Specifically, *ports* model hardware-specific I/O channels—e.g., actuation or readout interfaces defined by the vendor. *Waveforms* describe signal envelopes created via create_waveform operations and emitted through play operations on mixed frames—structures mixing port channel and frame state. Further, *frames* combine a logical clock—time that increments with use—with stateful carrier signal parameters—frequency and phase.

Gate-level operations have direct pulse analogs that act on these mixed *frames*: for example, barrier, delay, shift_phase, set_phase, shift_frequency, set_frequency, and play are defined to sequence and modulate pulses instead of qubits. Readout is implemented by performing a play on a readout *frame* followed by a capture of that frame [18].

```
1  module {
2    pulse.def @waveform_1 { // Define waveforms
3      pulse.waveform amplitudes = %amplitudes_in : vector.vector<5
           ↪xi32>
4    }
5    pulse.def @waveform_2 { ... }
6    pulse.def @waveform_3 { ... }
7    // Main pulse-level kernel
8    pulse.sequence @pulse_vqe_quantum_kernel(
9    %drive0: !!pulse.mixed_frame, %drive1: !pulse.mixed_frame,
10   %coupler: !pulse.mixed_frame, %freq: f64,
11   %phase: f64) -> i1
12   attributes { pulse.argPorts = ["q0-drive-port",
13   "q1-drive-port", "q0q1-coupler-port", "", ""],
14   pulse.args =["q0-drive-frame", "q1-drive-frame",
15   "coupler-frame", "freq", "phase"]} {
16     // 1. Gate-level X on both qubits
17     pulse.standard_x(%drive0) : !pulse.mixed_frame
18     pulse.standard_x(%drive1) : !pulse.mixed_frame
19     // 2. Waveform constants
20     %wf1 = pulse.waveform.amplitudes @waveform_1
21     %wf2 = pulse.waveform.amplitudes @waveform_2
22     %wf3 = pulse.waveform.amplitudes @waveform_3
23     // 3. Apply single-qubit pulses
24     pulse.play(%drive0, %wf1): (!pulse.mixed_frame, !pulse.waveform
           ↪)
25     pulse.play(%drive1, %wf2): (!pulse.mixed_frame, !pulse.waveform
           ↪)
26     // 4. Frame changes
27     pulse.frame_change(%drive0, %freq, %phase) : (!pulse.
           ↪mixed_frame, f64, f64)
28     pulse.frame_change(%drive1, %freq, %phase) : (!pulse.
           ↪mixed_frame, f64, f64)
29     // 5. Entangling pulse
30     pulse.play(%coupler, %wf3)
31       : (!pulse.mixed_frame, !pulse.waveform)
32     // 6. Measurement on qubit0
33     %wf_r = pulse.waveform.constant @readout_pulse
34     pulse.play (%readout0, %wf_r) : (!pulse.mixed_frame, !pulse.
           ↪waveform)
35     %m0 = pulse.capture (%capture0): (!pulse.mixed_frame) -> i1
36     // 7. Measurement on qubit1
37     pulse.play {}(%readout1, %wf_r): (!pulse.mixed_frame, !pulse.
           ↪waveform)
38     %m1 = pulse.capture (%capture1): (!pulse.mixed_frame) -> i1
39     pulse.return %m0, %m1 : i1, i1
40   }
41 }
```

Listing 2: Pulse-level MLIR kernel (defining *waveforms*, applying pulses, *frame* changes, and captures) demonstrating how MQSS can represent and compile an entire pulse sequence. Listings 1 and 3 are intended to be equivalent representations of the same quantum kernel.

Note that LLVM's MLIR pass manager makes adding pulse-level support to the MQSS's compiler straightforward by simply extending the *MQSS Pass Suite* and register those passes against the appropriate dialect. Its existing pass runner infrastructure on the other hand, allows these new passes to be seamlessly combined with gate- and pulse-based pipelines[2].

Importantly, if, over time, the hardware modalities or pulse semantics require specialized behavior beyond what IBM's *pulse* MLIR dialect supports, one can define a custom *MQSS Pulse Dialect*. This dialect can then leverage the same pass manager framework and MLIR infrastructure to support richer or domain-specific pulse-level semantics.

---

[2]Note that by treating pulse constructs as first-class IR elements, LLVM frameworks like MQSS also make it possible to extend a quantum accelerator's native gate set. This means that an expert can define a new quantum gate by providing its pulse *waveform* on that hardware, and the compiler will lower it into the corresponding pulse operations, seamlessly integrating the new gate into the framework.
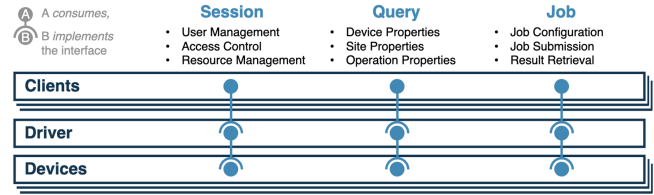
Figure 3: An overview of the QDMI interface, its components, and how they are connected across the three QDMI entities. Adding pulse-specific device, site, and operation properties will enable clients to retrieve information—such as the level of pulse access, available channels, and more—via the existing 'Query' interface. Note that submitting jobs with pulse-payload does not require modifications to the 'Job' interface; it only requires adding a single enumeration value.

5.3 Quantum Device Management Interface

QDMI is the hardware abstraction layer of MQSS, enabling seamless integration between software services—such as simulators, calibration tools, compilers, and telemetry-driven error mitigation—and quantum accelerators, including both physical quantum accelerators and third-party simulators [50]. As the *QDMI* interface specification is defined in C as a header-only library, it facilitates efficient use within HPC environments and supports services, like noise-aware simulation and automated calibration.

As seen in Fig. 3, *QDMI* defines three primary entities:

- **QDMI Clients**: The users of the QDMI library. For example, the *MQSS Client*, the *MQSS Compiler Passes*, or external tools. The clients do not have direct access to the devices but access through a *QDMI Driver*.
- **QDMI Driver**: A bespoke solution for orchestrating these interactions, managing available *QDMI Devices* and mediating client-side requests by implementing session and job control structures.
- **QDMI Devices**: Quantum hardware, quantum simulators, databases, and potentially remote services provided by, for example, Independent Software Vendors (ISVs), and even data centers.

The *QDMI* specification includes types and handlers for *sites*, *operations*, *sessions*, and *jobs*. This novel interface uses opaque pointers and enumeration values for data structures and operations on it, ensuring that new properties or operations can be added without breaking existing interfaces.

In *QDMI*, a *site* references a physical or logical qubit location—e.g., a superconducting qubit, an ion-trapped qubit, or a neutral-atom trap. *Operations* encompass, for example, quantum gates, measurements, and movement primitives. We will extend the *QDMI Operations* to support pulse primitives. In summary, to enable pulse-level support, the *QDMI* specification will be extended to provide:

- **Query capabilities** to see if the pulse interface is supported, and to query the types of supported pulses, their parameters and the allowed range of values. Pulse support can be provided at two levels of abstraction: *site* level and *port* level.
- **Data structures** added to represent pulse *waveforms*, pulse implementation and *ports*. The pulse representation can be used at both *site* and *port* levels.
- **Mechanisms** to query and set default pulse implementations for specific operations, as well as to add pulse implementation for custom operations.

- **A unified pulse submission interface**, to submit pulse programs in an exchange format supported by the device.

These enhancements ensure that MQSS can query the necessary information in order to prepare and submit pulse-level payloads to quantum hardware in a standardized, backward-compatible manner.

5.4 Exchange Format

QIR is a hardware-agnostic LLVM-compliant IR specification that many quantum toolchains already use as a universal exchange language [44].

Unlike textual languages such as, for example, OpenQASM, QIR can be directly compiled and linked with vendor-provided runtimes or libraries. For instance, a QIR *LLVM module* contains declared, but unimplemented quantum routines (e.g., `__quantum__qis__h__body`) that are resolved by linking in the hardware-specific definitions during execution. As a consequence, a QIR job can become an executable intermediate object, reducing end-to-end latency and keeping the format technology- and device-agnostic. By contrast, for example, the latest release of OpenQASM[3]—which does now include pulse constructs via `defcal` and `defcalgrammar`—is still a textual high-level format. Likewise, IBM has deprecated its old *Qiskit Pulse* schedules in favor of calibrated *fractional gates* and so-called *quantum dynamics* libraries.

We believe that relying on a mature, LLVM-based IR is a future-proof solution. Given the efficient and robust optimization routines brought by the LLVM ecosystem from the classical computing domain, state-of-the-art quantum stacks such as, for example, Quantinuum's, NVIDIA's, Rigetti's, and Oak Ridge National Laboratory (ORNL)'s are converging on QIR or other LLVM-based IRs [34].

We propose extending the QIR specification with a *Pulse Profile* to natively carry pulse-level abstractions, and using that QIR with pulse support as the default exchange format for pulses in MQSS and, consequently, the *QDMI* specification. QIR already defines the notion of *Profiles* to specialize this LLVM-compliant IR for certain hardware or use cases. Here, a so-called *Pulse Profile* would augment the *Base Profile* with the abstractions we introduced in Section 4, namely *port*, *frame*, and *waveform*. In practice, this could potentially mean adding new metadata and intrinsics to the QIR *LLVM modules*. For example, Listing 3 shows a human-readable QIR *LLVM module* with `attributes #0 = { "entry_point" "output_labeling_schema" "qir_profiles"="pulse" "required_num_ports"="1" }`. In this snippet:

- **Pulse Profile metadata**: Attribute `qir_profiles="pulse"` marks the *LLVM module*'s entry function as using the new *Pulse Profile*. The accompanying `output_labeling_schema` (and custom fields like `required_num_ports="1"`) signal that the program output should be formatted as a pulse job. These metadata align with QIR's extensibility points—for example, the QIR specification describes *Output Schemas* for calibrations or adaptive circuits—so we leverage the same mechanism to trigger a specialized pulse output format.

```
1  ; ModuleID = 'my_pulse'
2  %Qubit = type opaque
3  %Port = type opaque
4  %Waveform = type opaque
5  %Frame = type opaque
6  define void @my_pulse(float* %amps, float %freq, float %phase) #0
      ↪{
7    call void @__quantum__pulse__waveform__body(%Wave* %waveform0,
          ↪float* %amps)
8    call void @__quantum__pulse__waveform_play__body(%Port* %port0,
          ↪%Wave* waveform0)
9    call void @__quantum__pulse__frame_change__body(%Port* port0, %
          ↪freq, %phase)
10   call void @__quantum__pulse__delay__body(%Port* port0, 1024)
11   call void @__quantum__qis__mz__body(%Qubit* inttoptr (i64 0 to %
          ↪Qubit*), %Result* inttoptr (i64 0 to %Result*)) #1
12   call void @__quantum__qis__mz__body(%Qubit* inttoptr (i64 1 to %
          ↪Qubit*), %Result* inttoptr (i64 1 to %Result*)) #1
13   ret void
14 }
15 declare %Waveform* @__quantum__pulse__waveform__body(float, float
      ↪*)
16 declare void @__quantum__pulse__waveform_play__body(%Port*, %
      ↪Waveform*)
17 declare %Frame* @__quantum__pulse__frame_change__body(%Port*,
      ↪double)
18 declare void @__quantum__pulse__delay__body(%Frame*, int)
19 attributes #0 = { "entry_point" "output_labeling_schema" "
      ↪qir_profiles"="pulse" "required_num_ports"="1" }
```

Listing 3: Example QIR showing a potential mechanism to express a pulse job (with `qir_profiles="pulse"`): pulses are constructed via LLVM calls to `__quantum__pulse` intrinsics on opaque `%Port`, `%Waveform`, and `%Frame` types. Listings 1 and 2 are intended to be equivalent representations of the same quantum kernel.

- **Opaque types matching our abstractions**: We introduce QIR opaque types `%Port`, `%Waveform`, and `%Frame` (in addition to `%Qubit` and `%Result`) to represent the hardware elements of a pulse sequence. These types directly correspond to the MQSS abstractions of a control *port*, *frame*, and *waveform* (see Sec. 3). Given that an LLVM-compliant IR should support named opaque structs, QIR can carry these types without assuming any hardware details.
- **Pulse intrinsics (LLVM calls)**: Within the function (e.g. `@my_pulse`), the code calls new *pulse* intrinsics that mirror our pulse operations. For instance, `@__quantum__pulse__waveform__body(%Waveform* %waveform0, float* %amps)` creates a *waveform* object from amplitude samples, and `@__quantum__pulse__waveform_play__body(%Port* %port0, %Waveform* %waveform0)` plays it on a *port*. Similarly, `@__quantum__pulse__frame_change__body(%Port* %port0, double %freq, double %phase)` changes the frequency/phase on that *port*, and `@__quantum__pulse__delay__body(%Frame* %frame, int)` inserts a delay. These are declared (but not defined) in the *LLVM module*, just as the standard QIR calls are. At runtime, the hardware-specific *QDMI Device* layer would link these calls to the actual device APIs that implements *waveform* generation and scheduling.
- **Integration with gate-level operations**: The example also mixes in a standard QIR Quantum Instruction Set (QIS)[4] measurement. After constructing and sending pulses, the code calls `__quantum__qis__mz__body(%Qubit*, %Result*)` to measure each qubit. This shows that pulse-level instructions

---

[3]OpenQASM 3, at the moment of writing.

[4]That is, an instruction set compatible with the proposed *Pulse Profile*.

can seamlessly coexist with gate-level calls in the same QIR *LLVM module*. In MQSS, this means a single exchange file could contain the full hybrid sequence (pulses followed by measurements), preserving the program structure.

Overall, adopting a QIR-based exchange format for pulses leverages the robustness of LLVM (widely used in current stacks [46]) and the interoperability of a technology-neutral IR. QIR's LLVM roots mean we can apply standard compiler passes, optimizations, and linking workflows to pulse programs just as we do for classical code. By embedding pulse semantics at the IR level, MQSS becomes "pulse-aware" end-to-end while still retaining gate-level compatibility. This then ties together the entire MQSS.

In a typical workflow, *MQSS Adapters* can produce MLIR-pulse code, MQSS's MLIR-based compiler will then *lower* it to QIR with pulse support, and *QDMI* will submit it to the target quantum device for the hardware runtime to execute the linked *waveform* instructions.

5.5 Consistency Across the Stack

When combining the approach across all layers discussed above, the result is a unified design where *port*, *frame*, and *waveform* mean the same thing at every layer, and where pulse-level access is supported in both the high-level programming interfaces, MLIR-based JIT compilation, and the lower-level QIR exchange format supported by a backend interface like, for example, MQSS's *QDMI*.

6 Current Status

The work presented here is not a speculative vision but the result of an ongoing, coordinated effort across several fronts. To ground our design in hardware reality, we organized a series of day-long Technical Exchange Meetings (TEMs) with providers of neutral-atom, ion-trap, and superconducting devices, with upcoming workshops planned for photonic systems. These discussions allowed us to identify commonalities across technologies and feed those insights back into our proposal. Concretely, *QDMI* is already being extended through pull requests in its public GitHub repository [31], reflecting the lessons learned from these TEMs. In parallel, we are contributing at the ecosystem level: Leibniz Supercomputing Centre (LRZ) is part of the QIR Alliance steering committee and successfully proposed a new workstream to extend the QIR specification with pulse-level capabilities. This effort will build on the same hardware knowledge base while ensuring compatibility with the extensions under development for *QDMI*. On the other hand, we are analyzing how to evolve each *MQSS Adapter* [29] to accept pulse-level job definitions in a manner that remains compatible with existing pulse programming interfaces—thereby avoiding integration issues should MQSS add support for those platforms in the future as well. Once the *MQSS Adapters* are updated, the *MQSS Client* will be modified to forward this payload to the rest of the stack. Finally, because the MQSS compiler is LLVM/MLIR-based and dialect-agnostic, we expect only minimal core changes; nevertheless, we will empirically evaluate whether a dedicated MQSS pulse MLIR dialect is needed as implementation experience accumulates.

7 Related Work

Pulse-level control appears in several mainstream frameworks. Qiskit-Pulse (already deprecated) and Amazon Braket Pulse, for example, expose the same core abstractions we identified, that is, *ports*, *frames*, and *waveforms*, via Python APIs, providing convenient primitives for *waveform* construction, *frame* management, and scheduling. These interfaces are useful for experimentation, but their Pythonic nature and limit suitability for low-latency, tightly integrated HPC workflows.

On the IR and serialization side, OpenQASM 3 (with its calibration/`cal` blocks) and legacy formats such as `Qobj` demonstrate how pulse schedules can be described and exchanged, yet they remain tied to their originating ecosystems and are not designed as compiled, linkable IRs. Several groups (including at Quantinuum and ORNL [34]) are moving their compiler toolchains toward LLVM/MLIR-based designs to gain the benefits of compiled passes and tighter hardware integration; however, these projects typically address only parts of the stack (IR or runtime) rather than offering an end-to-end, HPC-centric solution.

For resource and device integration, the Quantum Resource Management Interface (QRMI) introduced in [42] and similar proposals address HPCQC integration, solving lifecycle and access-control problems, but not the compiler-level and pulse-format challenges required for native pulse programs. MQSS (with its *QPI*, LLVM/MLIR compiler, *QDMI*, and the proposed QIR with pulse support as exchange format) differs by targeting all four layers at once: a compiled, low-latency programming API, dialect-aware compilation, and a C/C++ backend query/management interface along with a QIR-based pulse exchange mechanism that align the same *port*, *frame*, and *waveform* abstractions across the entire pipeline.

That is, prior systems address important subproblems, but none yet combine compiler-aware IR, low-overhead programming API, backend interfaces, and a portable pulse exchange format in a single HPC-oriented stack, the gap our proposals is designed to fill.

8 Conclusions

We presented a detailed discussion of the obstacles to integrating pulse-level quantum control into an HPC stack and proposed solutions within MQV's MQSS. Our analysis identified three required pulse abstractions, namely, *ports*, *frames*, and *waveforms*, to be supported at programming interface, IR, backend interface, and exchange format levels. We also introduced the appropriate extensions to MQSS. Concretely, we 1) introduced an extension to its C-based programming API with pulse constructs to avoid Python runtime overhead, 2) illustrated the adoption of an MLIR pulse dialect to represent pulse commands alongside gate-level instructions, 3) leveraged its C/C++ quantum backend interface specification to query hardware pulse constraints during JIT compilation, and 4) defined an extension to add pulse-level support to the QIR specification and adopt it as an exchange format to leverage dynamic pulse constructs linking to accelerator implementations. These adaptations allow MQSS and other similar heterogeneous HPCQC software stacks to natively represent and compile pulse sequences, while remaining compatible with existing HPC scheduling and execution models. Our work establishes an end-to-end path

for pulse-aware hybrid quantum-classical workloads: by embedding the low-level pulse semantics into each layer of the software stack, we enable advanced control techniques (calibrations, custom *waveforms*, etc.) within HPC environments. This pulse-enabled HPCQC stack opens the door for new kinds of quantum-accelerated algorithms and more effective exploitation of near-term hardware.

Acknowledgments

References

[1] 2025. Quantum error correction below the surface code threshold. *Nature* 638, 8052 (2025), 920–926.
[2] Q Ansel, E Dionis, F Arrouas, B Peaudecerf, S Guérin, D Guéry-Odelin, and D Sugny. 2024. Introduction to theoretical and experimental aspects of quantum optimal control. *Journal of Physics B: Atomic, Molecular and Optical Physics* 57, 13 (jun 2024), 133001. doi:10.1088/1361-6455/ad46a5
[3] Yuval Baum, Mirko Amico, Sean Howell, Michael Hush, Maggie Liuzzi, Pranav Mundada, Thomas Merkh, Andre RR Carvalho, and Michael J Biercuk. 2021. Experimental deep reinforcement learning for error-robust gate-set design on a superconducting quantum computer. *PRX Quantum* 2, 4 (2021), 040324.
[4] Fabrizio Berritta, Jacob Benestad, Lukas Pahl, Melvin Mathews, Jan A. Krzywda, Réouven Assouly, Youngkyu Sung, David K. Kim, Bethany M. Niedzielski, Kyle Serniak, Mollie E. Schwartz, Jonilyn L. Yoder, Anasua Chatterjee, Jeffrey A. Grover, Jeroen Danon, David K. Oliver, and Ferdinand Kuemmeth. 2025. Efficient Qubit Calibration by Binary-Search Hamiltonian Tracking. *arXiv preprint arXiv:2501.05386* (2025). Describes real-time adaptive frequency calibration via feedback loops.
[5] Marin Bukov, Alexandre G. R. Day, Dries Sels, Phillip Weinberg, Anatoli Polkovnikov, and Pankaj Mehta. 2018. Reinforcement Learning in Different Phases of Quantum Control. *Phys. Rev. X* 8 (Sep 2018), 031086. Issue 3. doi:10.1103/PhysRevX.8.031086
[6] Lukas Burgholzer, Jorge Echavarria, Patrick Hopf, Yannick Stade, Damian Rovara, Ludwig Schmid, Ercüment Kaya, Burak Mete, Muhammad Nufail Farooqi, Minh Chung, Marco De Pascale, Laura Schulz, Martin Schulz, and Robert Wille. 2025. The Munich Quantum Software Stack: Connecting End Users, Integrating Diverse Quantum Technologies, Accelerating HPC. arXiv:2509.02674 [quant-ph] https://arxiv.org/abs/2509.02674
[7] D. Daems, A. Ruschhaupt, D. Sugny, and S. Guérin. 2013. Robust Quantum Control by a Single-Shot Shaped Pulse. *Phys. Rev. Lett.* 111 (Jul 2013), 050404. Issue 5. doi:10.1103/PhysRevLett.111.050404
[8] Robert De Keijzer, Oliver Tse, and Servaas Kokkelmans. 2023. Pulse based variational quantum optimal control for hybrid quantum computing. *Quantum* 7 (2023), 908.
[9] Andrea Delgado and Prasanna Date. 2025. Defining quantum-ready primitives for hybrid HPC-QC supercomputing: a case study in Hamiltonian simulation. *Frontiers in Computer Science* 7 (2025), 1528985. doi:10.3389/fcomp.2025.1528985
[10] D. Dong and I.R. Petersen. 2010. Quantum control theory and applications: a survey. *IET Control Theory & Applications* 4 (2010), 2651–2671. Issue 12. doi:10.1049/iet-cta.2009.0508 arXiv:https://digital-library.theiet.org/doi/pdf/10.1049/iet-cta.2009.0508
[11] Daniel J. Egger, Chiara Capecci, Bibek Pokharel, Panagiotis Kl Barkoutsos, Laurin E. Fischer, Leonardo Guidoni, and Ivano Tavernelli. 2023. Pulse Variational Quantum Eigensolver on Cross-Resonance Based Hardware. *Physical Review Research* 5, 3 (Sept. 2023). doi:10.1103/PhysRevResearch.5.033159 arXiv:2303.02410 [quant-ph]
[12] Simon J. Evered, Dolev Bluvstein, Marcin Kalinowski, Sepehr Ebadi, Tom Manovitz, Hengyun Zhou, Sophie H. Li, Alexandra A. Geim, Tout T. Wang, Nishad Maskara, Harry Levine, Giulia Semeghini, Markus Greiner, Vladan Vuletić, and Mikhail D. Lukin. 2023. High-fidelity parallel entangling gates on a neutral-atom quantum computer. *Nature* 622, 7982 (oct 2023), 268. doi:10.1038/s41586-023-06481-y arXiv:2304.05420 [quant-ph]
[13] Nic Ezzell, Bibek Pokharel, Lina Tewala, Gregory Quiroz, and Daniel A Lidar. 2023. Dynamical decoupling for superconducting qubits: A performance survey.

Physical Review Applied 20, 6 (2023), 064027.
[14] Guanru Feng, Franklin H. Cho, Hemant Katiyar, Jun Li, Dawei Lu, Jonathan Baugh, and Raymond Laflamme. 2018. Gradient-based closed-loop quantum optimal control in a solid-state two-qubit system. *Phys. Rev. A* 98 (Nov 2018), 052341. Issue 5. doi:10.1103/PhysRevA.98.052341
[15] Niklas J Glaser, Federico A Roy, Ivan Tsitsilin, Leon Koch, Niklas Bruckmoser, Johannes Schirk, João H Romeiro, Gerhard BP Huber, Florian Wallner, Malay Singh, et al. 2024. Sensitivity-adapted closed-loop optimization for high-fidelity controlled-z gates in superconducting qubits. *arXiv preprint arXiv:2412.17454* (2024).
[16] Steffen J Glaser, Ugo Boscain, Tommaso Calarco, Christiane P Koch, Walter Köckenberger, Ronnie Kosloff, Ilya Kuprov, Burkhard Luy, Sophie Schirmer, Thomas Schulte-Herbrüggen, et al. 2015. Training Schrödinger's cat: Quantum optimal control: Strategic report on current status, visions and goals for research in Europe. *The European Physical Journal D* 69, 12 (2015), 279.
[17] D. Guéry-Odelin, A. Ruschhaupt, A. Kiely, E. Torrontegui, S. Martínez-Garaot, and J. G. Muga. 2019. Shortcuts to adiabaticity: Concepts, methods, and applications. *Rev. Mod. Phys.* 91 (Oct 2019), 045001. Issue 4. doi:10.1103/RevModPhys.91.045001
[18] 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. 2024. Design and architecture of the IBM Quantum Engine Compiler. arXiv:2408.06469 [quant-ph] https://arxiv.org/abs/2408.06469
[19] Sven Jandura and Guido Pupillo. 2022. Time-Optimal Two- and Three-Qubit Gates for Rydberg Atoms. *Quantum* 6 (May 2022), 712. doi:10.22331/q-2022-05-13-712
[20] Ercüment Kaya, Burak Mete, Laura Brandon Schulz, Muhammad Nufail Farooqi, Jorge Echavarria, and Martin Schulz. 2024. QPI: A Programming Interface for Quantum Computers. In *IEEE International Conference on Quantum Computing and Engineering, QCE 2024, Montreal, QC, Canada, September 15-20, 2024*, Marek Osinski, Brian La Cour, and Lia Yeh (Eds.). IEEE, 286–291. doi:10.1109/QCE60285.2024.10293
[21] Navin Khaneja, Timo Reiss, Cindie Kehlet, Thomas Schulte-Herbrüggen, and Steffen J Glaser. 2005. Optimal control of coupled spin dynamics: design of NMR pulse sequences by gradient ascent algorithms. *Journal of magnetic resonance* 172, 2 (2005), 296–305.
[22] Christiane P Koch, Ugo Boscain, Tommaso Calarco, Gunther Dirr, Stefan Filipp, Steffen J Glaser, Ronnie Kosloff, Simone Montangero, Thomas Schulte-Herbrüggen, Dominique Sugny, et al. 2022. Quantum optimal control in quantum technologies. Strategic report on current status, visions and goals for research in Europe. *EPJ Quantum Technology* 9, 1 (2022), 19.
[23] Marko Kuzmanović, Isak Björkman, John J. McCord, Shruti Dogra, and Gheorghe Sorin Paraoanu. 2024. High-fidelity robust qubit control by phase-modulated pulses. *Phys. Rev. Res.* 6 (Feb 2024), 013188. Issue 1. doi:10.1103/PhysRevResearch.6.013188
[24] Martín Letras, Jorge Echavarria, Muhammad Nufail Farooqi, Marco De Pascale, Mario Hernández Vera, Nathaniel Tornow, Laura Schulz, and Martin Schulz. 2025. Towards a Unified Multi-Target MLIR-Based Compiler: A Heterogeneous Compilation Framework for High-Performance/Quantum Computing Integration. In *IEEE International Conference on Quantum Computing and Engineering, QCE 2025, Albuquerque, New Mexico, USA, August 31-September 5, 2025*.
[25] C. M. Löschnauer, J. Mosca Toba, A. C. Hughes, S. A. King, M. A. Weber, R. Srinivas, R. Matt, R. Nourshargh, D. T. C. Allcock, C. J. Ballance, C. Matthiesen, M. Malinowski, and T. P. Harty. 2024. Scalable, high-fidelity all-electronic control of trapped-ion qubits. arXiv:2407.07694 [quant-ph] https://arxiv.org/abs/2407.07694
[26] Alicia B Magann, Christian Arenz, Matthew D Grace, Tak-San Ho, Robert L Kosut, Jarrod R McClean, Herschel A Rabitz, and Mohan Sarovar. 2021. From pulses to circuits and back again: A quantum optimal control perspective on variational quantum algorithms. *PRX Quantum* 2, 1 (2021), 010101.
[27] Jarrod R McClean, Sergio Boixo, Vadim N Smelyanskiy, Ryan Babbush, and Hartmut Neven. 2018. Barren plateaus in quantum neural network training landscapes. *Nature communications* 9, 1 (2018), 4812.
[28] Oinam Romesh Meitei, Bryan T. Gard, George S. Barron, David P. Pappas, Sophia E. Economou, Edwin Barnes, and Nicholas J. Mayhall. 2021. Gate-Free State Preparation for Fast Variational Quantum Eigensolver Simulations: Ctrl-VQE. doi:10.48550/arXiv.2008.04302 arXiv:2008.04302 [quant-ph]
[29] Munich Quantum Software Stack (MQSS). 2025. MQSS Interfaces Documentation. Web page. https://munich-quantum-software-stack.github.io/MQSS-Interfaces/ Documentation of programming interfaces (Qiskit, CUDAQ, PennyLane) for MQSS.
[30] MQV. 2025. MQV's Munich Quantum Software Stack. https://www.munich-quantum-valley.de/research/research-areas/mqss.
[31] MQV. 2025. Munich Quantum Software Stack GitHub.com Organization. https://github.com/Munich-Quantum-Software-Stack.
[32] Matthias M Müller, Ressa S Said, Fedor Jelezko, Tommaso Calarco, and Simone Montangero. 2022. One decade of quantum optimal control in the chopped random basis. *Reports on Progress in Physics* 85, 7 (jun 2022), 076001. doi:10.1088/1361-6633/ac723c
[33] Hunter T. Nelson, Evangelos Piliouras, Kyle Connelly, and Edwin Barnes. 2023. Designing dynamically corrected gates robust to multiple noise sources using

geometric space curves. *Phys. Rev. A* 108, 1 (jul 2023), 012407. doi:10.1103/PhysRevA.108.012407 arXiv:2211.13248 [quant-ph]

[34] Thien Nguyen, Dmitry Lyakh, Raphael C. Pooser, Travis S. Humble, Timothy Proctor, and Mohan Sarovar. 2021. Quantum Circuit Transformations with a Multi-Level Intermediate Representation Compiler. arXiv:2112.10677 [quant-ph] https://arxiv.org/abs/2112.10677

[35] Josias Old, Stephan Tasler, Michael J. Hartmann, and Markus Müller. 2025. Fault-Tolerant Stabilizer Measurements in Surface Codes with Three-Qubit Gates. arXiv:2506.09029 [quant-ph] https://arxiv.org/abs/2506.09029

[36] Pablo M. Poggi, Gabriele De Chiara, Steve Campbell, and Anthony Kiely. 2024. Universally Robust Quantum Control. *Phys. Rev. Lett.* 132 (May 2024), 193801. Issue 19. doi:10.1103/PhysRevLett.132.193801

[37] Riccardo Porotti, Vittorio Peano, and Florian Marquardt. 2023. Gradient-Ascent Pulse Engineering with Feedback. *PRX Quantum* 4 (Jul 2023), 030305. Issue 3. doi:10.1103/PRXQuantum.4.030305

[38] Anurag Saha Roy, Kevin Pack, Nicolas Wittler, and Shai Machnes. 2025. Software tool-set for automated quantum system identification and device bring up. In *2025 17th International Conference on COMmunication Systems and NETworks (COMSNETS)*. IEEE, 1062–1067.

[39] Martin Schulz, Laura Brandon Schulz, Martin Ruefenacht, and Robert Wille. 2023. Towards the Munich Quantum Software Stack: Enabling Efficient Access and Tool Support for Quantum Computers. In *IEEE International Conference on Quantum Computing and Engineering, QCE 2023, Bellevue, WA, USA, September 17-22, 2023*, Brian La Cour, Lia Yeh, and Marek Osinski (Eds.). IEEE, 399–400. doi:10.1109/QCE57702.2023.10301

[40] Kyle M. Sherbert, Hisham Amer, Sophia E. Economou, Edwin Barnes, and Nicholas J. Mayhall. 2025. Parameterization and Optimizability of Pulse-Level VQEs. *Physical Review Applied* 23, 2 (Feb. 2025). doi:10.1103/PhysRevApplied.23.024036 arXiv:2405.15166 [quant-ph]

[41] Phattharaporn Singkanipa, Victor Kasatkin, Zeyuan Zhou, Gregory Quiroz, and Daniel A Lidar. 2025. Demonstration of algorithmic quantum speedup for an Abelian hidden subgroup problem. *Physical Review X* 15, 2 (2025), 021082.

[42] Iskandar Sitdikov, M. Emre Sahin, Utz Bacher, Aleksander Wennersteen, Andrew Damin, Mark Birmingham, Philippa Rubin, Stefano Mensa, Matthieu Moreau, Aurelien Nober, Hitomi Takahashi, and Munetaka Ohtani. 2025. Quantum resources in resource management systems. arXiv:2506.10052 [quant-ph] https://arxiv.org/abs/2506.10052

[43] Kaitlin N. Smith, Gokul Subramanian Ravi, Thomas Alexander, Nicholas T. Bronn, André R. R. Carvalho, Alba Cervera-Lierta, Frederic T. Chong, Jerry M. Chow, Michael Cubeddu, Akel Hashim, Liang Jiang, Olivia Lanes, Matthew J. Otten, David I. Schuster, Pranav Gokhale, Nathan Earnest, and Alexey Galda. 2022. Programming physical quantum systems with pulse-level control. *Frontiers in Physics* 10 (2022), 900099. doi:10.3389/fphy.2022.900099

[44] Yannick Stade, Lukas Burgholzer, and Robert Wille. 2024. Towards Supporting QIR: Thoughts on Adopting the Quantum Intermediate Representation. arXiv:2411.18682 [quant-ph] https://arxiv.org/abs/2411.18682

[45] Shinichi Sunami, Shiro Tamiya, Ryotaro Inoue, Hayata Yamasaki, and Akihisa Goban. 2025. Scalable Networking of Neutral-Atom Qubits: Nanofiber-Based Approach for Multiprocessor Fault-Tolerant Quantum Computers. *PRX Quantum* 6 (Feb 2025), 010101. Issue 1. doi:10.1103/PRXQuantum.6.010101

[46] Aleksandra Swierkowska, Jorge Echavarria, Laura Brandon Schulz, and Martin Schulz. 2024. Achieving Pareto-Optimality in Quantum Circuit Compilation via a Multi-Objective Heuristic Optimization Approach. In *IEEE International Conference on Quantum Computing and Engineering, QCE 2024, Montreal, QC, Canada, September 15-20, 2024*, Marek Osinski, Brian La Cour, and Lia Yeh (Eds.). IEEE, 306–310. doi:10.1109/QCE60285.2024.10297

[47] Léo Van Damme, Zhao Zhang, Amit Devra, Steffen J Glaser, and Andrea Alberti. 2025. Motion-insensitive time-optimal control of optical qubits. *QST* 10, 3 (may 2025), 035025. doi:10.1088/2058-9565/add61c

[48] Xin Wang, Bo Qi, Yabo Wang, and Daoyi Dong. 2024. EHA: Entanglement-variational Hardware-efficient Ansatz for Eigensolvers. *Physical Review Applied* 21, 3 (March 2024), 034059. doi:10.1103/PhysRevApplied.21.034059 arXiv:2311.01120 [quant-ph]

[49] Max Werninghaus, Daniel J Egger, Federico Roy, Shai Machnes, Frank K Wilhelm, and Stefan Filipp. 2021. Leakage reduction in fast superconducting qubit gates via optimal control. *npj Quantum Information* 7, 1 (2021), 14.

[50] Robert Wille, Ludwig Schmid, Yannick Stade, Jorge Echavarria, Martin Schulz, Laura Brandon Schulz, and Lukas Burgholzer. 2024. QDMI - Quantum Device Management Interface: Hardware-Software Interface for the Munich Quantum Software Stack. In *IEEE International Conference on Quantum Computing and Engineering, QCE 2024, Montreal, QC, Canada, September 15-20, 2024*, Marek Osinski, Brian La Cour, and Lia Yeh (Eds.). IEEE, 573–574. doi:10.1109/QCE60285.2024.10411

[51] Yuanjing Zhang, Tao Shang, Chenyi Zhang, and Xueyi Guo. 2025. Pulse-Level Quantum Robust Control with Diffusion-Based Reinforcement Learning. *Advanced Physics Research* 4, 5 (2025), 2400159. doi:10.1002/apxr.202400159 arXiv:https://advanced.onlinelibrary.wiley.com/doi/pdf/10.1002/apxr.202400159