



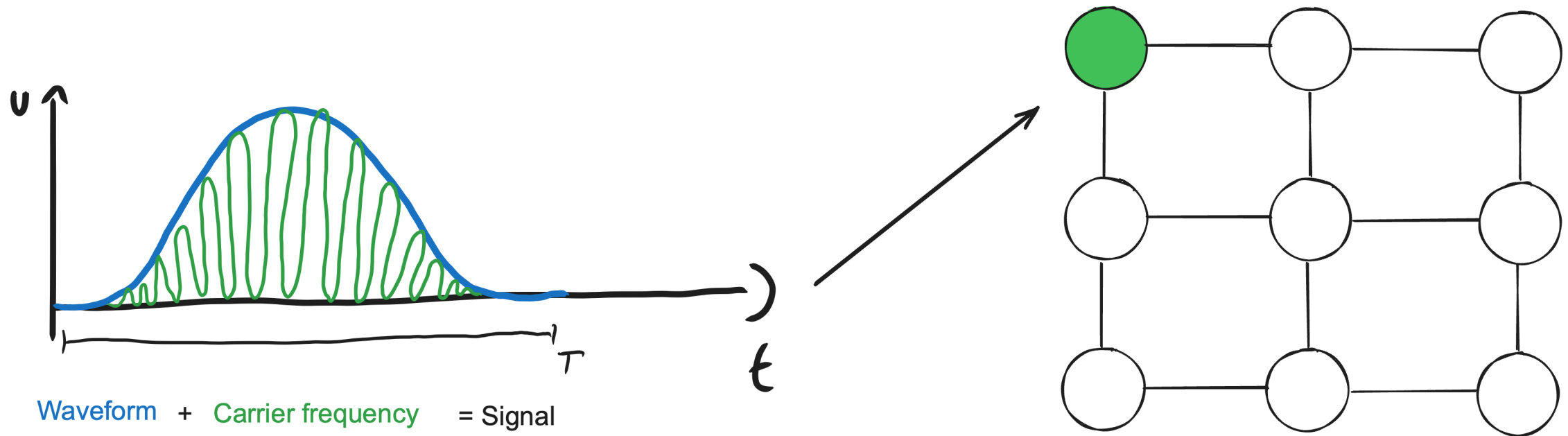
Leibniz Supercomputing Centre
of the Bavarian Academy of Sciences and Humanities

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

Case Study: The **Munich Quantum Software Stack - MQSS**

Jorge Echavarria

2025.09.10



Parameterization and optimizability of pulse-level VQEs

Kyle M. Sherbert,^{1,2,3} Hisham Amer,^{2,3} Sophia E. Economou,^{2,3} Edwin Barnes,^{2,3} and Nicholas J. Mayhall^{1,3,*}

¹*Department of Chemistry, Virginia Tech, Blacksburg, VA 24061*

²*Department of Physics, Virginia Tech, Blacksburg, VA 24061*

³*Virginia Tech Center for Quantum Information Science and Engineering, Blacksburg, VA 24061, USA*

In conventional variational quantum eigensolvers (VQEs), trial states are prepared by applying series of parameterized gates to a reference state, with the gate parameters being varied to minimize the energy of the target system. Recognizing that the gates are intermediates which are ultimately compiled into a set of control pulses to be applied to each qubit in the lab, the recently proposed ctrl-VQE algorithm takes the amplitudes, frequencies, and phases of the pulse as the variational parameters used to minimize the molecular energy. In this work, we explore how all three degrees of freedom interrelate with one another. To this end, we consider several distinct strategies to parameterize the control pulses, assessing each one through numerical simulations of a transmon-like device. For each parameterization, we contrast the pulse duration required to prepare a good ansatz, and the difficulty to optimize that ansatz from a well-defined initial state. We deduce several guiding heuristics to implement practical ctrl-VQE in hardware, which we anticipate will generalize for generic device architectures.

I. INTRODUCTION

Variational quantum eigensolvers (VQEs) are among the most promising candidates for achieving useful computations in chemistry on near-term quantum computers [1–6]. At their core, they are predict-and-test methods, where a quantum state, determined by a set of classical parameters as specified by an *ansatz*, is prepared on the quantum computer, and its energy measured. Then a

Three of the authors previously proposed the algorithm ctrl-VQE [15, 16], which takes the idea of a hardware-efficient ansatz to the extreme by parameterizing the actual physical control pulses used in the lab, bypassing the use of gates entirely. Designing the ansatz at the pulse level allows drastically shorter evolution times, even approaching the quantum speed limits imposed by the hardware [17], and hypothetically enabling the VQE to study much larger or more complex systems. The methods and

- Extended Supported Gate Sets
- Automated Calibration
- Pareto-optimal Gate Pulse Implementation
- Pulse-level VQEs
- Dynamical Decoupling
- Marketing

MQV's Munich Quantum Software Stack **MQSS**

The **MQSS** is the output of the **Q-DESSI**¹ (K5) consortium of the **Munich Quantum Valley** (MQV)

Our mission is to develop a:

- **comprehensive,**
- **extendable,** and
- **flexible open-source**

software for *full-stack* quantum computing systems.

MQSS Solutions

- Munich Quantum Portal
MQP
- Quantum Programming Interface
QPI
- Quantum Resource Manager
QRM
- Quantum Device Management Interface
QDMI
- ...

Source: planqc; Modality: Neutral Atoms System

MQSS Services

- Noiseless Simulation
- Noise-based Simulation
- JIT Compilation and Optimization
- Telemetry-based Error Mitigation
- Automated Calibration
- **Pulse-level Control**
- ...

Source: AQT; Modality: Ion-trap System

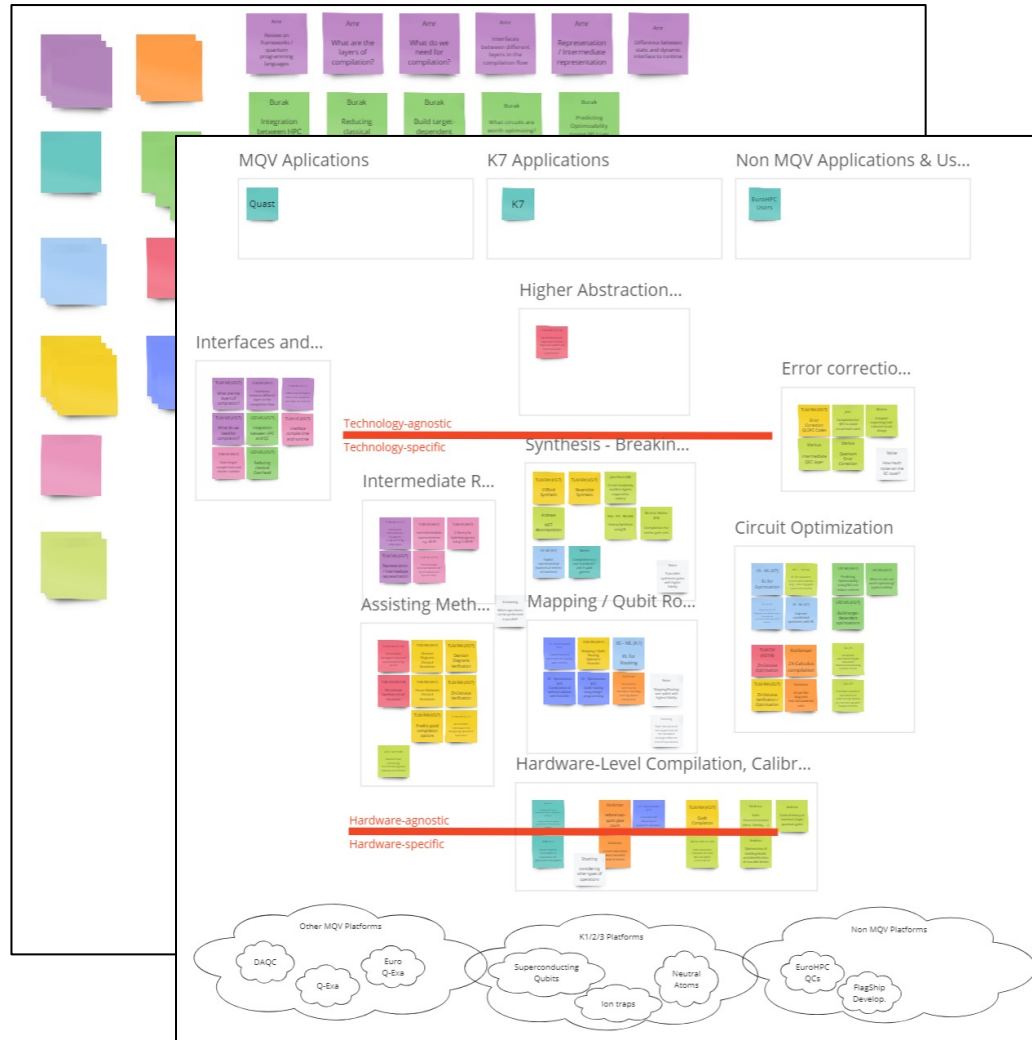
Munich Quantum Software Stack - MQSS

Integrating Pulse-level Support



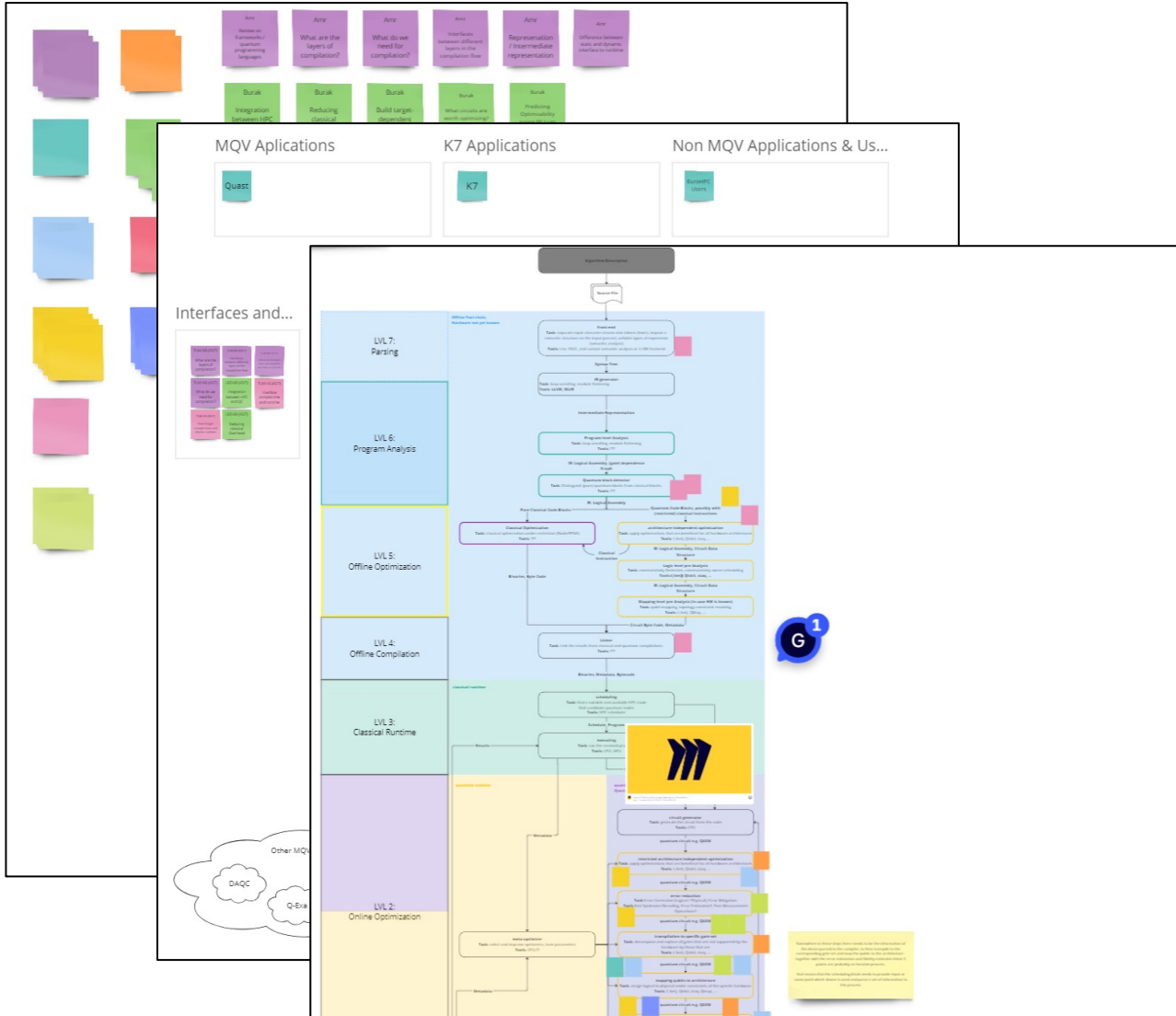
Munich Quantum Software Stack - MQSS

Integrating Pulse-level Support



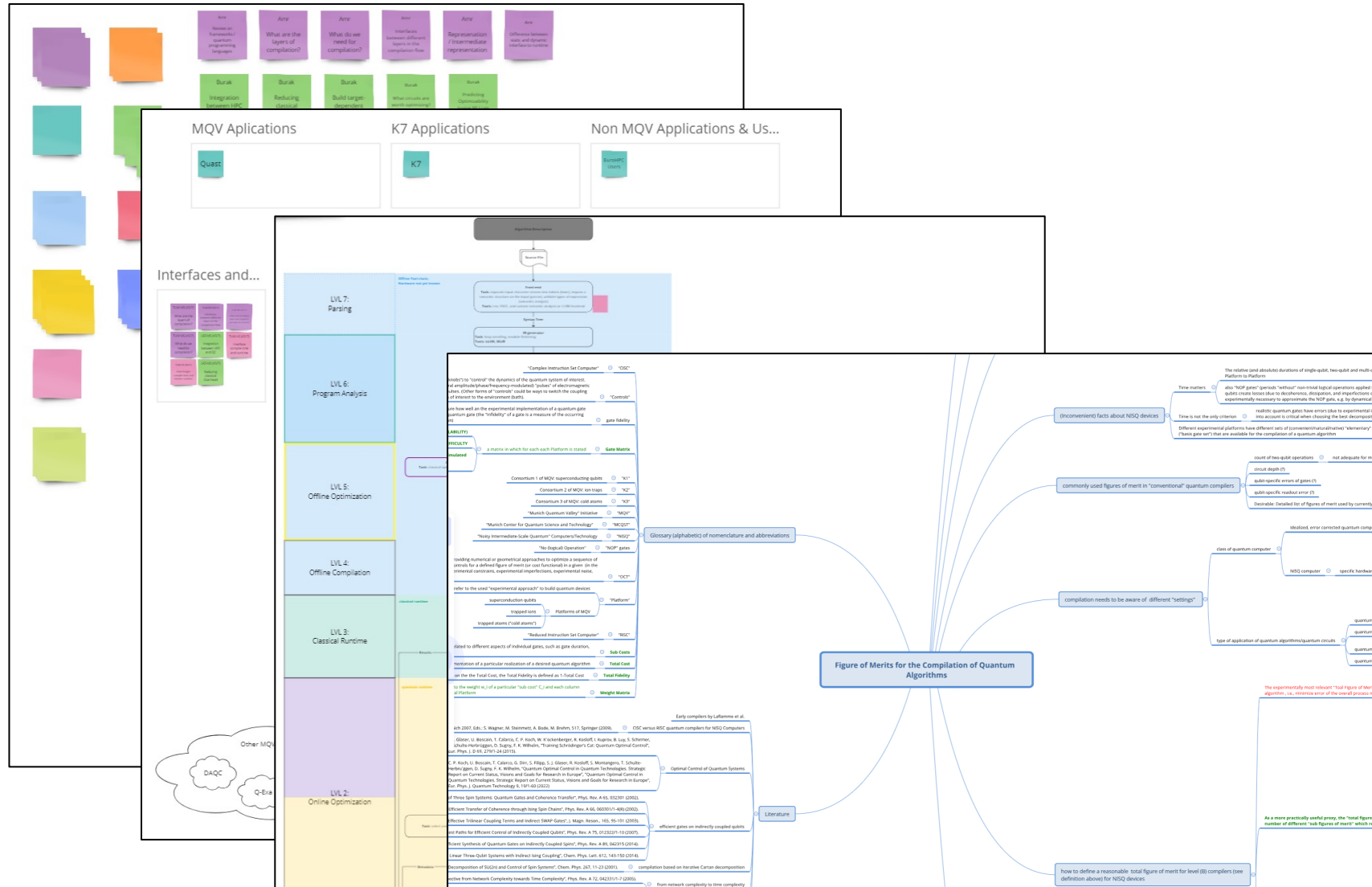
Munich Quantum Software Stack - MQSS

Integrating Pulse-level Support



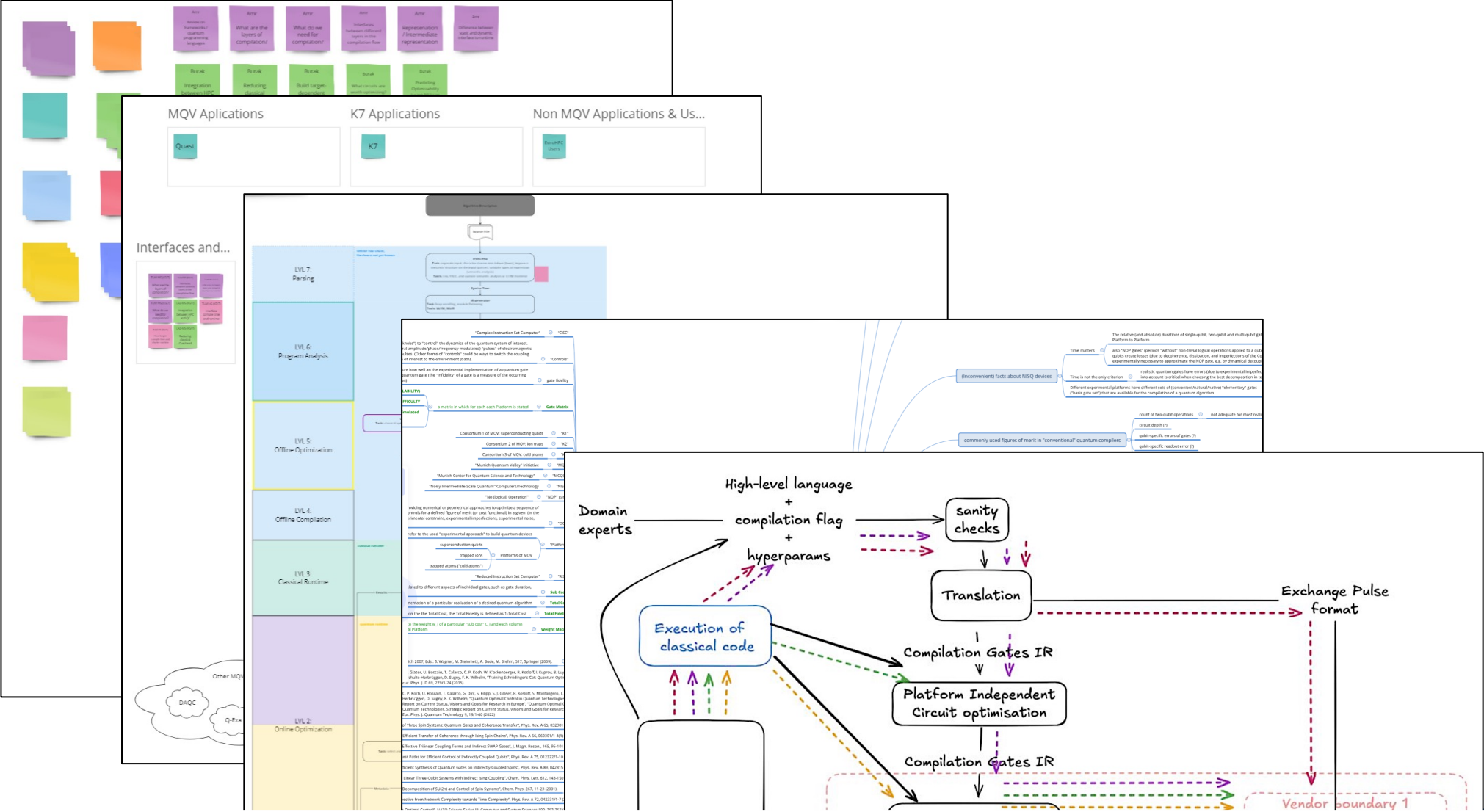
Munich Quantum Software Stack - MQSS

Integrating Pulse-level Support



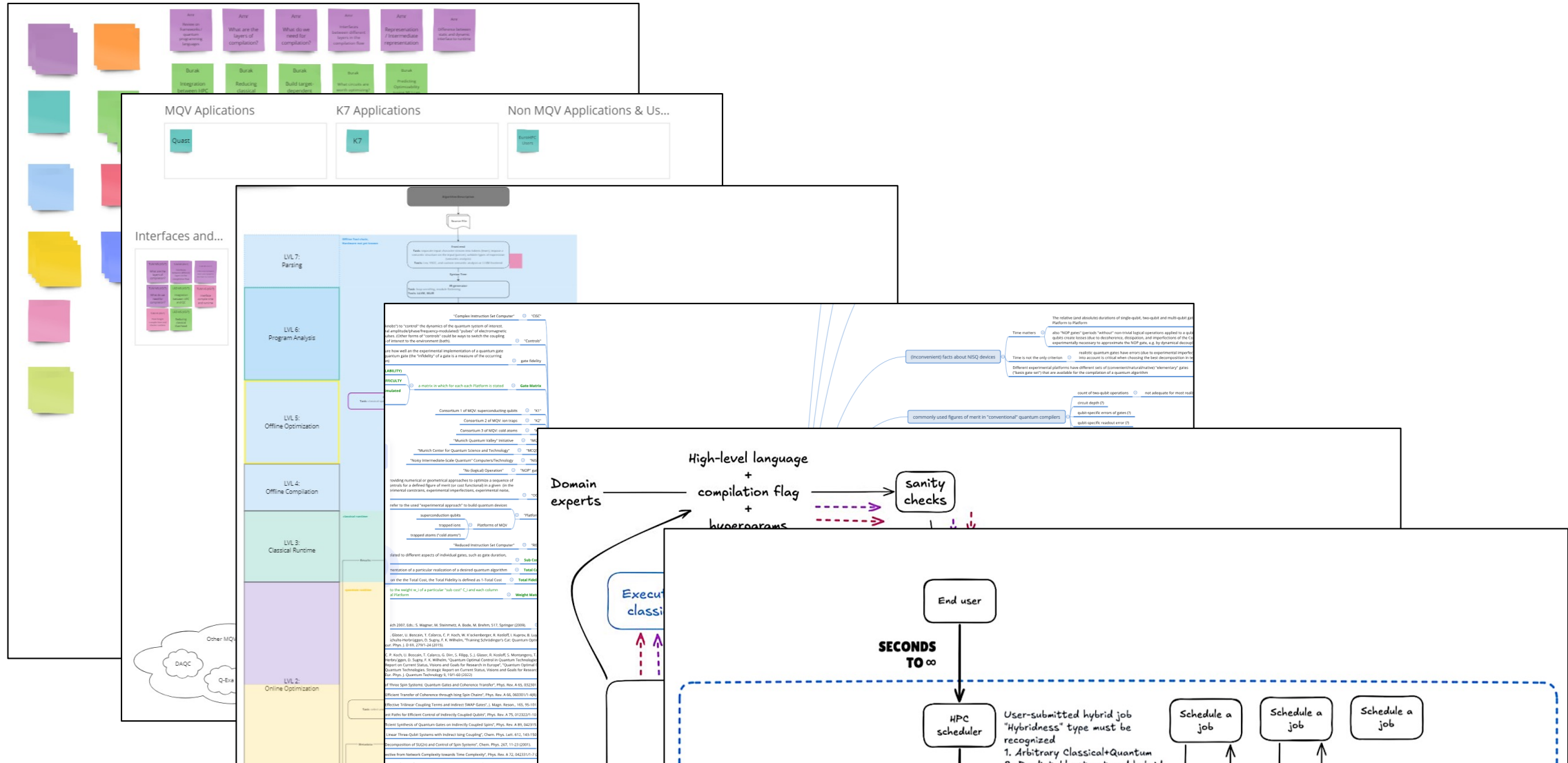
Munich Quantum Software Stack - MQSS

Integrating Pulse-level Support



Munich Quantum Software Stack - MQSS

Integrating Pulse-level Support



Advanced



Intermediate



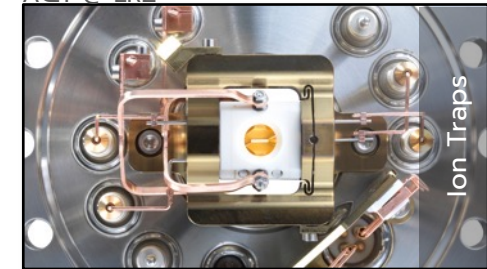
Basic



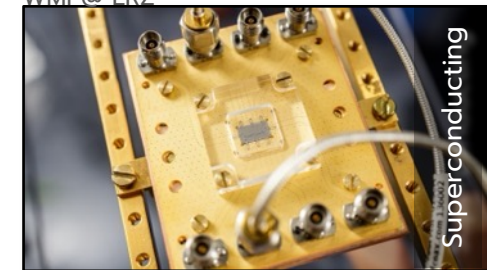
m types of users

n available devices

AQT @ LRZ



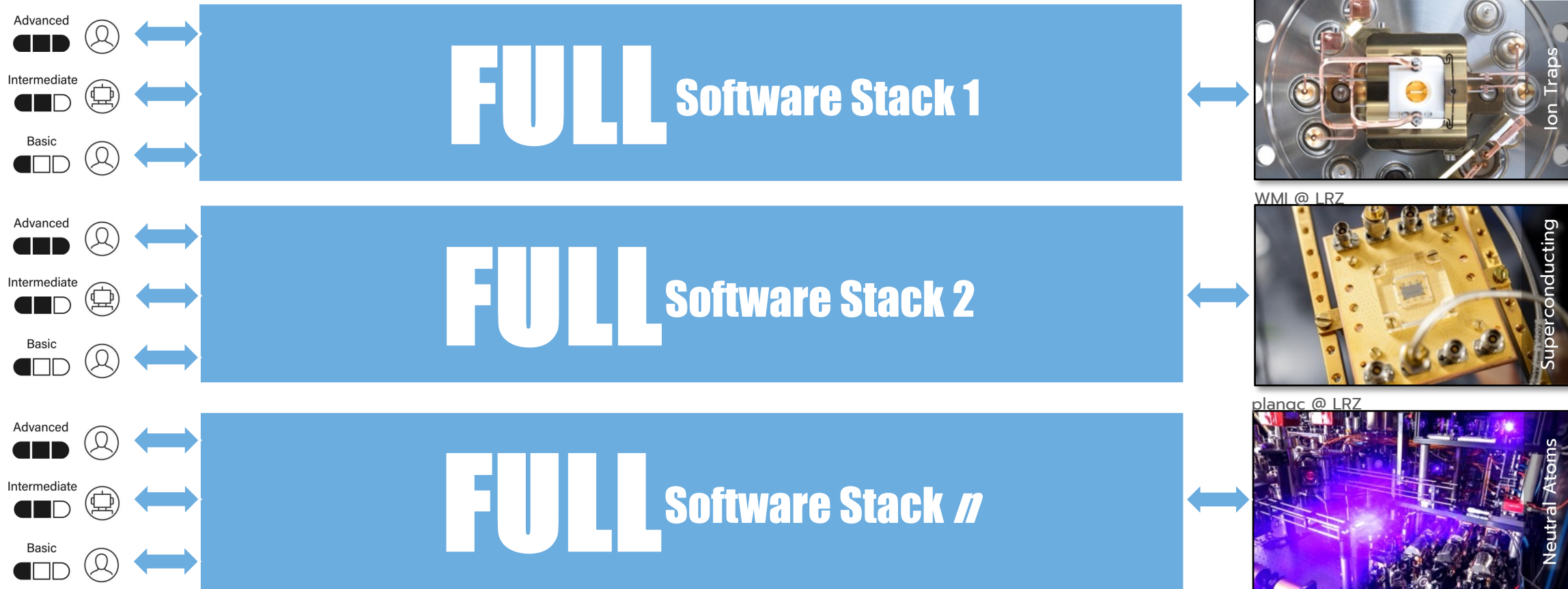
WMI @ LRZ



planqc @ LRZ







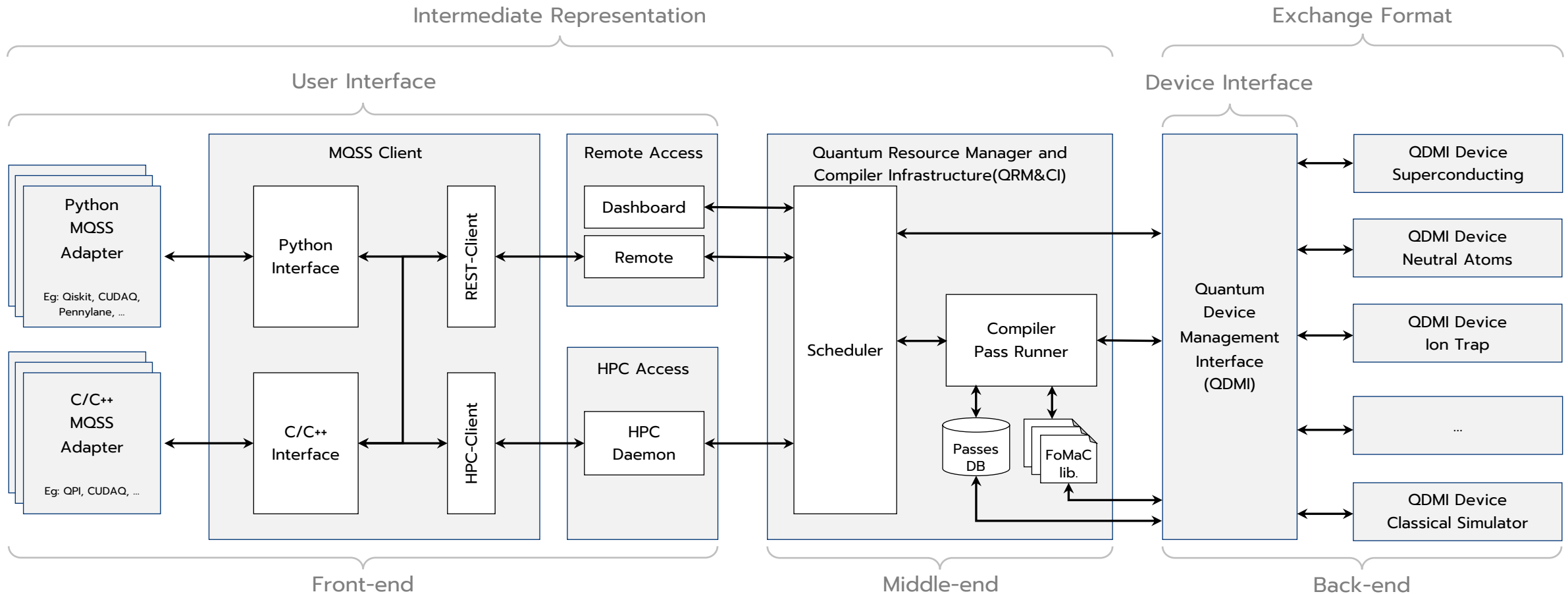
Munich Quantum Software Stack - MQSS

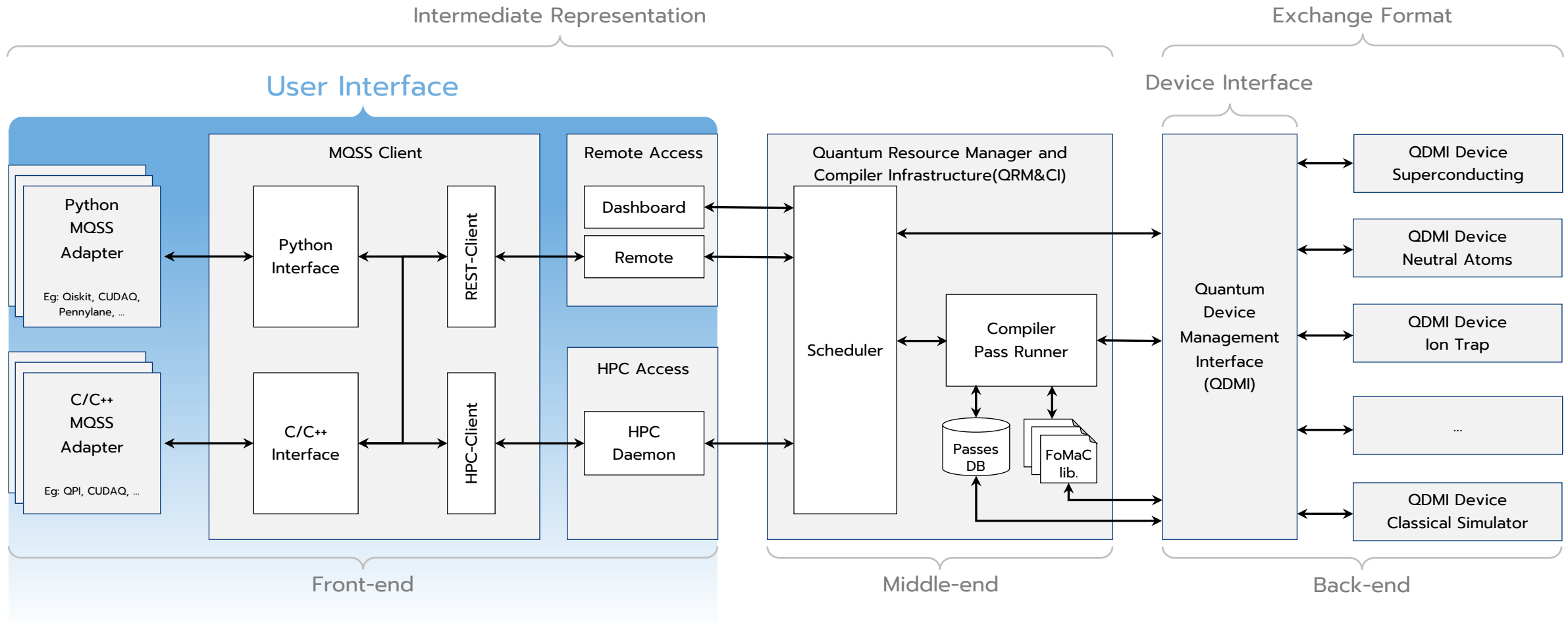
A True Full Software Stack

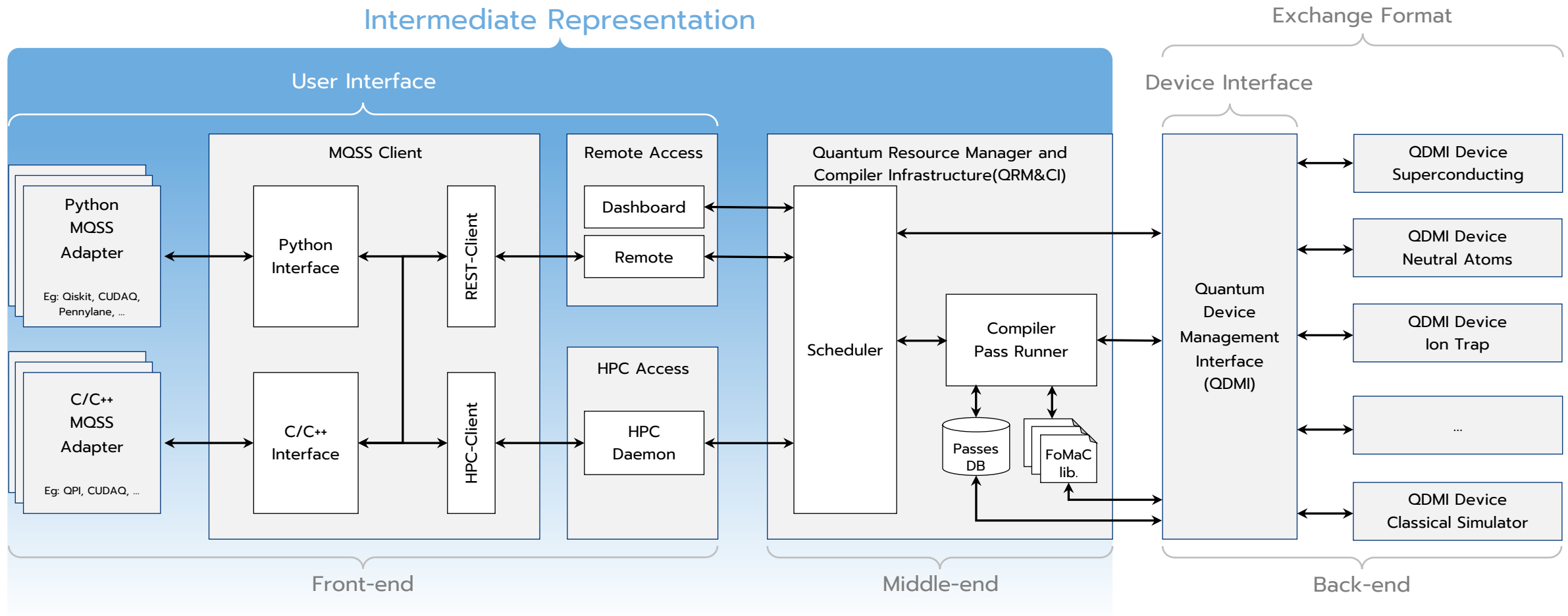


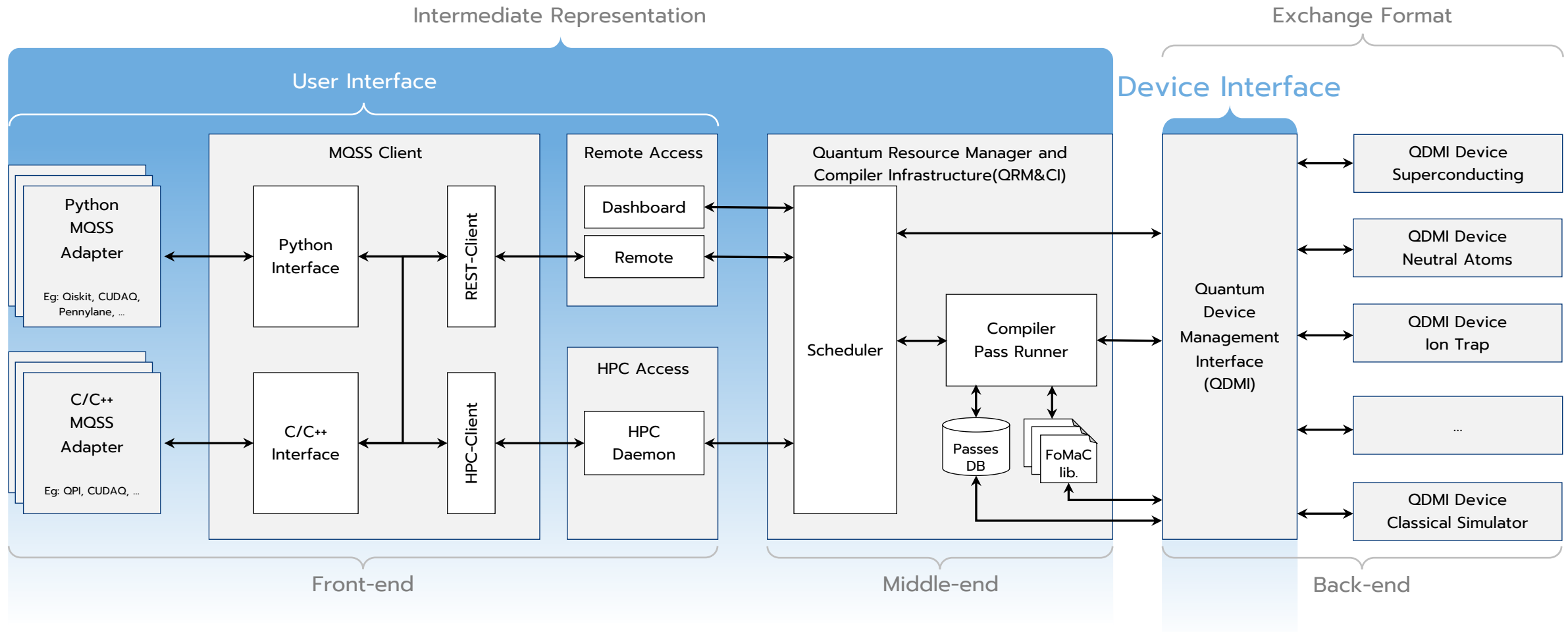
Munich Quantum Software Stack - MQSS

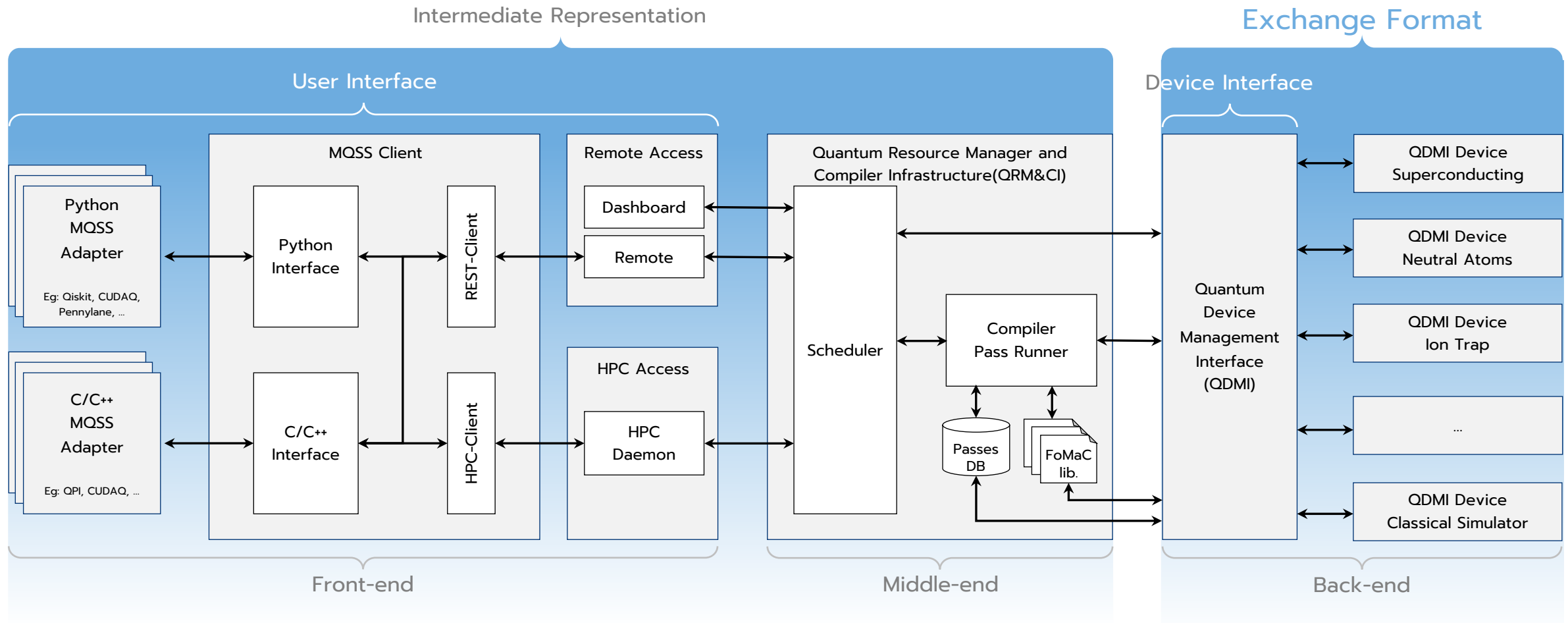
A True Full Software Stack











➤ Challenges

- ❖ User Interface
- ❖ Device Interface
- ❖ Intermediate Representation
- ❖ Exchange Format

C-based Quantum Programming Interface

2024 IEEE International Conference on Quantum Computing and Engineering (QCE)

QPI: A Programming Interface For Quantum Computers

Ercüment Kaya^{*†}, Burak Mete^{*†}, Laura Schulz^{*}, Muhammad Nufail Farooqi^{*}, Jorge Echavarria^{*}, Martin Schulz[†],
^{*}Leibniz Supercomputing Centre of the Bavarian Academy of Sciences and Humanities, Garching, Bavaria, Germany
{ercument.kaya, burak.mete, laura.schulz, muhammad.farooqi, jorge.echavarria}@lrz.de
[†]Technical University of Munich, Garching, Bavaria, Germany
{schulzm}@in.tum.de

Abstract—With the increasing maturity and accessibility of quantum computers, their alignment, integration, and use in the high-performance computing (HPC) ecosystem as a novel accelerator triggers a crucial new area of research. To address the demands for efficient and tightly coupled programming, we present the Quantum Programming Interface (QPI), a C-based library enabling the development of quantum tasks and submission to quantum resources.

I. INTRODUCTION

High-performance computing (HPC) systems offer enhanced performance and less resource consumption by integrating various devices and architectures. On the other hand, quantum computers (QC), with their unique computing paradigm, hold promises of being accelerators and stand-alone computational approaches, with their inherent capability of tackling problems that would require exponential resources to solve for their classical counterparts. However, it is essential to recognize that both computational paradigms can complement each other: QCs enable solving or accelerating intrinsic problems in quantum computing, while HPC systems pave the way for more optimal computing by handling operational control of QCs, the compilation of quantum circuits, and supporting the parameter optimization of quantum circuits in variational quantum algorithms.

Combining the radically different approaches of HPC systems with quantum computers presents a significant challenge at the software level. Beyond establishing a physical connection, the software stack development enables seamless user interaction between the two systems. Creating a hybrid application requires quantum programming tools (QPTs), which are designed to specify the interaction between the quantum computer and the HPC system. QPTs need to be abstracted from the quantum component at the application layer. Moreover, QPTs must be compatible with existing HPC tools and higher-level programming languages to create a better user experience and facilitate maintenance.

Quantum circuit compilation is another crucial step within the quantum integration software. It transforms high-level quantum algorithms into hardware-specific implementations, optimizing circuit efficiency, potentially reducing errors, and ensuring compatibility with diverse quantum hardware technologies, enabling seamless and efficient utilization of quan-

tum resources for real-world applications. Furthermore, given the possibility of multiple quantum backends employing various underlying technologies and features integrated into the software stack, it is crucial to abstract the compilation layer from both the application layer and the architecture. This is commonly achieved by describing the application in a so-called intermediate representation (IR), thereby adding support for various hardware configurations and addressing common programming requirements such as scheduling and further optimizations.

To tackle these challenges holistically, we present the Quantum Programming Interface (QPI), a lightweight library to embed quantum circuits in HPC applications. QPI enables the acceleration of HPC applications by allowing programmers to describe their quantum or classical-quantum programs within a common programming interface while efficiently leveraging quantum resources, regardless of the quantum device responsible for executing the job thereafter.

QPI is a C-programming interface that allows users to create quantum circuits at a high level of abstraction, which are then converted into an LLVM-compliant IR, allowing seamless communication and execution on various quantum computers and simulators.

Our main contributions are the following:

- We eliminate application and architecture dependencies from quantum circuits and HPC systems, simplifying the creation of quantum circuits
- We provide a holistic approach for hybrid quantum-classical applications
- We abstract the underlying technology of the target QPUs and expose them as local accelerators
- Overall, we offer a novel solution tailored for HPC ecosystems for 1) describing quantum circuits through an interface familiar to most researchers, 2) parsing the quantum algorithm's components into an LLVM-compliant IR, and 3) offloading it to the quantum compiler for its subsequent execution by the targeted quantum accelerator

These contributions are further described in section III.

C-based Quantum Programming Interface

LLVM/IR-based compilation

2024 IEEE International Conference on Quantum Computing and Engineering (QCE)

QPI: A Programming Interface For Quantum Computers

Ercüment Kaya*[†]
*Leibniz Supercomputing Centre of the Bavarian Academy of Sciences and Humanities, Garching, Bavaria, Germany

Abstract—With the quantum computers, the high-performance accelerator triggers the demands for efficient quantum circuit compilation. We present the Quantum Compiler Library (QCL) based library enabling submission to quantum hardware.

High-performance quantum computing requires integrating various disciplines. In quantum computing, the high-performance paradigm, hold promise computational approach, tackling problems that solve for their classical counterparts. Recognize that both quantum and classical problems in quantum computing for more optimal compilation of QCs, the compiler the parameter optimization algorithms.

Combining the quantum hardware with quantum software at the software connection, the software user interaction between application requirements which are designed quantum computer abstraction from the layer. Moreover, QP tools and higher-level better user experience. Quantum circuit of the quantum integration quantum algorithms optimizing circuit ensuring compatibility technologies, enabling s

979-8-3315-4137-8/24/\$31.00
DOI 10.1109/QCE60285.2024.10468888

2024 IEEE International Conference on Quantum Computing and Engineering (QCE)

Achieving Pareto-Optimality in Quantum Circuit Compilation via a Multi-Objective Heuristic Optimization Approach

Aleksandra Świerkowska*[†], Jorge Echavarria*, Laura Schulz*, Martin Schulz[‡],
*Leibniz Supercomputing Centre of the Bavarian Academy of Sciences and Humanities, Garching, Bavaria, Germany
{aleksandra.swierkowska, jorge.echavarria, laura.schulz}@lrz.de
[‡]Technical University of Munich, Garching, Bavaria, Germany
{martin.w.j.schulz}@tum.de

Abstract—High Performance Computing-Quantum Computing (HPCQC) integration presents a promising yet challenging opportunity, particularly in the area of quantum circuit compilation and optimization, requiring further advancements in the field of Quantum Computing (QC). To address this, we introduce the Munich Quantum Compiler, a key component of the Munich Quantum Software Stack (MQSS). This compiler employs a heuristic-based approach to select a Pareto-optimal subset of optimizations in the form of LLVM passes for quantum circuits described in an LLVM-compliant Intermediate Representation (IR).

Index Terms—Quantum Computing, Multi-objective Optimization, Quantum Compilation, LLVM, QIR, MOEA, Genetic Algorithm, NSGA-II

I. INTRODUCTION

In recent years, Quantum Computing (QC) has demonstrated significant potential for achieving exponential performance improvements over classical algorithms for certain classes of computational problems [1]–[5]. In order to enable wider growth and development of the quantum potential, efforts towards High Performance Computing-Quantum Computing (HPCQC) integration have been initiated [6], [7]. Reaching their goal of providing seamless cooperation between classical and quantum parts of the system would allow to reach a new territory of research, mainly in the form of hybrid algorithms.

However, to be able to achieve a hybrid software stack, it is necessary to design highly sophisticated compilers capable of both: 1) providing effective optimizations for quantum circuits, such as reducing their size to allow execution before significant coherence degradation, and 2) adapting the quantum circuits to the unique capabilities and limitations of the available quantum accelerators. Furthermore, a quantum compiler should be able to support the common software stack for classical and quantum applications. For that reason, we decided to utilize Quantum Intermediate Representation (QIR) [8], an LLVM-compliant Intermediate Representation (IR) supporting interleaving quantum and classical instructions within a single

transformations to the circuit through the application of custom LLVM passes. Nevertheless, employing compilation schemes akin to the classical world, such as iteratively applying LLVM optimization passes, introduces classical challenges into the quantum domain, as we will elaborate.

The challenges associated with finding Pareto-optimal optimization subsets [13] and the sequence in which to apply them [14], also known as phase ordering, have been extensively studied in the classical compilation field. These are well-known NP-Hard problems [15]. The proposed solutions range from utilizing Genetic Algorithms (GAs) [16]–[18], for years deemed as state-of-the-art, through more modern approaches based on, for example, Machine Learning (ML) [19] or Reinforcement Learning (RL) [15], [20], [21].

Similarly, ML and RL are usually utilized regarding quantum compilation [22]–[25], mainly due to their efficient execution times. However, most of the proposed approaches in the literature focus on optimizing a singular objective, usually a complex figure of merit consisting of a weighted sum of multiple different objectives, such as depth or number of gates. Although the importance of some quantum metrics is well-established, the development of effective quantum performance metrics remains an active area of research [26]. To provide a change of metrics in a figure of merit in model-based compilers, each time a tedious and time-consuming model retraining is necessary.

The novelty of the quantum optimization approach proposed in the Munich Quantum Compiler lies in providing a multi-objective optimization through the utilization of a GA, more specifically, a non-dominated sorting-based Multi-Objective Evolutionary Algorithm (MOEA) called Non-dominated Sorting Genetic Algorithm II (NSGA-II). This approach yields a set of solution candidates belonging to the Pareto frontier, none of which is fully dominated by any other solution found. While GAs have proven to be highly effective in the classical domain, to the best of our knowledge, they have not yet been applied to the quantum version of this problem.

C-based Quantum Programming Interface

LLVM/IR-based compilation

Technology-agnostic device-side interface

2024 IEEE International Conference on Quantum Computing and Engineering (QCE)

QPI: A Programming Interface For Quantum Computers

Ercüment Kaya*[†]
[†]Leibniz Supercomputing Centre

Abstract—With the quantum computers, the high-performance accelerator triggers the demands for efficient quantum programming. We present the Quantum Programming Interface (QPI) based library enabling submission to quantum hardware.

High-performance hardware performance integrating various quantum computing paradigms, hold promise computational approaches tackling problems that solve for their classical counterparts. We recognize that both quantum and classical problems in quantum computing for more optimal compilation of QCs, the compiler parameter optimization quantum algorithms.

Combining the requirements with quantum hardware at the software connection, the software user interaction between application requirements which are designed quantum computer abstraction from the layer. Moreover, QPI tools and higher-level better user experience. Quantum circuit of the quantum integration quantum algorithms optimizing circuit ensuring compatibility technologies, enabling submission to quantum hardware.

979-8-3315-4137-8/24/\$31.00 ©2024 IEEE
DOI: 10.1109/QCE60285.2024.10411

2024 IEEE International Conference on Quantum Computing and Engineering (QCE)

Achieving Pareto-Optimality in Quantum Circuit Compilation via a Multi-Objective Heuristic Optimization Approach

Aleksandra
^{*}Leibniz Supercomputing Centre

Abstract—High Performance (HPCQC) integration presents opportunity, particularly in the optimization, requiring Quantum Computing (QC), Munich Quantum Compiler, Quantum Software Stack (QSS) heuristic-based approach to optimizations in the form of described in an LLVM-compliant Intermediate Representation (IR).

Index Terms—Quantum compilation, Quantum Compiler Algorithm, NSGA-II

In recent years, Quantum computing has demonstrated significant performance improvements over classical counterparts. However, the integration of quantum computing into existing computational workflows remains a challenge. This paper presents a heuristic-based approach to quantum circuit compilation, aiming to achieve Pareto-optimality in the compilation process. The approach is implemented within the Quantum Software Stack (QSS) and leverages the LLVM-compliant Intermediate Representation (IR) for quantum circuit compilation.

However, to be able to achieve necessary design goals, such as reducing their size to coherence degradation, and to the unique capabilities of quantum accelerators. Furthermore, we are able to support the compilation and quantum applications, utilizing Quantum Intermediate Representation (QIR) and LLVM-compliant Intermediate Representation (IR) for quantum circuit compilation.

2024 IEEE International Conference on Quantum Computing and Engineering (QCE)

QDMI – Quantum Device Management Interface: Hardware-Software Interface for the Munich Quantum Software Stack

Robert Wille*[†], Ludwig Schmid[‡], Yannick Stadel[§], Jorge Echavarria[¶], Martin Schulz^{||}, Laura Schulz^{||}, Lukas Burgholzer*
^{*}Chair for Design Automation, Technical University of Munich, Munich, Germany
[†]Software Competence Center Hagenberg GmbH, Hagenberg, Austria
[‡]Leibniz Supercomputing Centre, Garching, Germany
[§]Chair of Computer Architecture and Parallel Systems, Technical University of Munich, Munich, Germany
{robert.wille, ludwig.s.schmid, yannick.stadel, martin.w.j.schulz, lukas.burgholzer}@tum.de
{laura.schulz, jorge.echavarria}@lrz.de

Abstract—Quantum computing is a promising technology that requires a sophisticated software stack to connect end users to the wide range of possible quantum backends. However, current software tools are usually hard-coded for single platforms and lack a dynamic interface that can automatically retrieve and adapt to changing physical characteristics and constraints of different platforms. With new hardware platforms frequently introduced and their performance changing on a daily basis, this constitutes a serious limitation. In this paper, we showcase a concept and a prototypical realization of an interface, called the *Quantum Device Management Interface* (QDMI), that addresses this problem by explicitly connecting the software and hardware developers, mediating between their competing interests. QDMI allows hardware platforms to provide their physical characteristics in a standardized way, and software tools to query that data to guide the compilation process accordingly. This enables software tools to automatically adapt to different platforms and to optimize the compilation process for the specific hardware constraints. QDMI is a central part of the Munich Quantum Software Stack (MQSS)—a sophisticated software stack to connect end users to the wide range of possible quantum backends. QDMI is publicly available as open source at <https://github.com/Munich-Quantum-Software-Stack/QDMI>.

I. MOTIVATION

Quantum utility—the ability to solve useful problems with quantum computing—crucially depends on the quality of the quantum software stack used to realize potential applications. Such a stack consists of various layers of software tools and must be able to connect the end users (usually domain experts from the respective application areas such as material simulation, machine learning or optimization) with the wide

Different quantum devices have different architectures, gate sets, error rates, topology, calibration, and noise models or provide fundamentally different operational capabilities such as qubit shuttling [7], [8].

Different quantum algorithms have different requirements, objectives, and trade-offs [9]–[12]. In addition, these factors can vary over time and depend on the environmental conditions as well as the state of the device. This needs a way to enable efficient communication and optimization between quantum compilers and quantum devices that encapsulates and reflects the knowledge base of the people developing said software and hardware. After all, quantum computers are likely to be used as accelerators for classical computing platforms and, hence, need to be tightly integrated into the rest of the ecosystem and workflows [13]. Such a communication and optimization process would require a common language and a standardized interface that both parties can understand and use. This would allow the people developing software tools to query relevant information and feedback about devices, and the people developing the hardware to provide guidance, express limitations, and offer suggestions in a standardized and automated machine-readable form.

In this paper, we showcase the *Quantum Device Management Interface* (QDMI) as a central part of the *Munich Quantum Software Stack* (MQSS) that addresses this problem. The MQSS is a project of the *Munich Quantum Valley* (MQV) initiative and is jointly developed by the *Leibniz Supercomputing Centre* (LRZ) and the *Chair for Design Automation*.

C-based Quantum Programming Interface

QPI: A Programming Interface For Quantum Computers

LLVM/IR-based compilation

Technology-agnostic device-side interface

HPCQC integration

Ercüment Kaya^{*†}
^{*Leibniz Supercomputing Center}

Abstract—With the quantum computers, the high-performance accelerator triggers the demands for efficient quantum programming. We present the Quantum Programming Interface (QPI) based library enabling submission to quantum hardware.

High-performance quantum computing requires integrating various disciplines. Quantum computing paradigm, hold promise computational approach, tackling problems that solve for their classical counterparts. We recognize that both quantum and classical problems in quantum computing for more optimal compilation of QCs, the compiler parameter optimization quantum algorithms.

Combining the requirements with quantum computing, the software connection, the software user interaction between application requirements, which are designed quantum computer abstraction from the layer. Moreover, QP tools and higher-level better user experience. Quantum circuit of the quantum integration algorithms optimizing circuit ensuring compatibility technologies, enabling simulation, machine learning, storage and analysis.

979-8-3315-4137-8/24/\$31.00 ©2024 IEEE
DOI: 10.1109/QCE60285.2024.10411

2024 IEEE International Conference on Quantum Computing and Engineering (QCE)

Achieving Pareto-Optimality in Quantum Circuit Compilation via a Multi-Objective Heuristic Optimization Approach

Aleksandra Swierkowska[†]
^{*Leibniz Supercomputing Center}

Abstract—High Performance Computing (HPCQC) integration presents opportunity, particularly in the optimization, requiring Quantum Computing (QC), Munich Quantum Compiler, Quantum Software Stack (MQSS) heuristic-based approach to optimizations in the form of described in an LLVM-compatible (IR).

Index Terms—Quantum compilation, Quantum Compiler, Algorithm, NSGA-II

I. INTRODUCTION
In recent years, Quantum computing has demonstrated significant performance improvements over classical computing. The integration of quantum computing (HPCQC) into high performance computing (HPC) environments, reaching their goal of providing a new territory of hybrid algorithms.

However, to be able to achieve this, it is necessary to design highly efficient quantum algorithms, such as reducing their size to coherence degradation, and to the unique capabilities of quantum accelerators. Furthermore, to be able to support the quantum applications, quantum software must be able to utilize Quantum Intermediate LLVM-compliant Intermediate Representation (IR) and classical computing, machine learning, storage and analysis.

979-8-3315-4137-8/24/\$31.00 ©2024 IEEE
DOI: 10.1109/QCE60285.2024.10411

2024 IEEE International Conference on Quantum Computing and Engineering (QCE)

QDMI – Quantum Device Management Interface: Hardware-Software Interface for the Munich Quantum Software Stack

Robert Wille[§]

[§]Chair of Quantum Computing

Abstract—Quantum computing requires a sophisticated software stack to support the wide range of quantum hardware. This paper introduces a dynamic adaptation to changing different platform introduced and this constitutes a case a concept called the Quantum Device Management Interface (QDMI) addresses this problem and hardware design interests. QDMI physical characteristics to query a quantum device accordingly. This addresses this problem for the specific hardware of the Munich Quantum Software Stack to quantum backend at <https://github.com/qdm-i/qdm-i>.

Quantum utilization of quantum computing. Such a stack of quantum software must be able to support the quantum applications, quantum software must be able to utilize Quantum Intermediate LLVM-compliant Intermediate Representation (IR) and classical computing, machine learning, storage and analysis.

979-8-3315-4137-8/24/\$31.00 ©2024 IEEE
DOI: 10.1109/QCE60285.2024.10411

2024 IEEE International Conference on Quantum Computing and Engineering (QCE)

A Software Platform to Support Disaggregated Quantum Accelerators

Ercüment Kaya^{*†}, Jorge Echavarria[‡], Muhammad Nufail Farooqi[‡], Aleksandra Swierkowska^{††}, Patrick Hopf^{††}, Burak Mete^{††}, Lukas Burgholzer[‡], Robert Wille[§], Laura Schulz[‡], Martin Schulz[‡]

^{*Leibniz Supercomputing Center of the Bavarian Academy of Sciences and Humanities, Garching, Germany}

[†] Technical University of Munich (TUM), Munich, Germany

[‡] Ludwig-Maximilians-Universität München (LMU), Munich, Germany

Email: {ercuemt.kaya, jorge.echavarria, muhammad.farooqi, aleksandra.swierkowska, patrick.hopf, burak.mete, laura.schulz}@lrz.de[†] {lukas.burgholzer, robert.wille, martin.wj.schulz}@tum.de[‡]

Abstract—Quantum computers are making their way into High Performance Computing (HPC) centers as next-generation accelerators. Due to their physical implementation as mostly large appliances in separate racks, their number in typical data centers is significantly lower than the number of nodes offloading work to them, unlike the case with GPU accelerators. As a consequence, they form large-scale disaggregated infrastructures that pose a number of integration challenges due to their diverse implementation technologies and their need to be used as a shared resource for optimal utilization. Running hybrid High Performance Computing-Quantum Computing (HPCQC) applications in HPC environments, where the quantum portion is offloaded to the quantum processing units (QPUs), requires sophisticated resource management strategies to optimize resource utilization and performance. In this paper, we present one aspect of the Munich Quantum Software Stack (MQSS) - a Just-In-Time (JIT) compilation and execution software stack for quantum and hybrid quantum-HPC workloads - beneficial for integrating disaggregated quantum accelerators into traditional HPC clusters.



Fig. 1: A view into the Quantum Integration Centre (QIC) at LRZ/Munich showing a superconducting system (left), an ion-trap system (middle) and HPC racks covering the classic compute. The result is a strongly disaggregated infrastructure combining classical HPC clusters with large-scale accelerator

C-based Quantum Programming Interface

QPI: A Programming Interface For Quantum Computers

LLVM/IR-based compilation

Achieving Pareto-Optimality in Quantum Circuit Compilation via a Multi-Objective Heuristic Optimization Approach

Technology-agnostic device-side interface

QDMI – Quantum Device Management Interface: Hardware-Software Interface for the Munich Quantum Software Stack

HPCQC integration

A Software Platform to Support Disaggregated Quantum Accelerators

LLVM/MLIR-based compilation

Towards a Unified Multi-Target MLIR-Based Compiler: A Heterogeneous Compilation Framework for High-Performance and Quantum Computing Integration

The MQSS

The Munich Quantum Software Stack: Connecting End Users, Integrating Diverse Quantum Technologies, Accelerating HPC

LUKAS BURGHOLZER*, Technical University of Munich, Germany and Munich Quantum Software Company GmbH, Germany

JORGE ECHAVARRIA*, Leibniz Supercomputing Centre, Germany

PATRICK HOFER*, Leibniz Supercomputing Centre, Germany

C-based Quantum Programming Interface

LLVM/IR-based compilation

Technology-agnostic device-side interface

HPCQC integration

LLVM/MLIR-based compilation

The MQSS

Pulse-support

2024 IEEE International Conference on Quantum Computing and Engineering (QCE)

QPI: A Programming Interface For Quantum Computers

Ercüment Kaya*
*Leibniz Supercomputing Center

Abstract—With the quantum computers, the high-performance accelerator triggers the demands for efficient quantum programming. We present the Quantum Programming Interface (QPI) based library enabling submission to quantum computers.

High-performance quantum computing requires integrating various devices, handling quantum computing paradigm, holding programming computational approach, tackling problems that solve for their classical counterparts. We recognize that both quantum and classical problems in quantum computing for more optimal compilation of QCs, the compiler parameter optimization quantum algorithms.

Combining the requirements with quantum computing, the software connection, the software user interaction between application requirements, which are designed quantum computer abstraction from the layer. Moreover, QPI tools and higher-level better user experience.

Quantum circuit compilation, the quantum integration quantum algorithms, optimizing circuit ensuring compatibility technologies, enabling simulation.

979-8-3315-4137-8/24/\$31.00 DOI: 10.1109/QCE60285.2024.10411

2024 IEEE International Conference on Quantum Computing and Engineering (QCE)

Achieving Pareto-Optimality in Quantum Circuit Compilation via a Multi-Objective Heuristic Optimization Approach

Aleksandr Kozlov
*Leibniz Supercomputing Center

Abstract—High Performance Computing (HPC) integration presents a significant opportunity, particularly in the context of quantum circuit compilation and optimization, requiring efficient quantum computing (QC). The Munich Quantum Compiler, Quantum Software Stack (MQSS) presents a heuristic-based approach to optimizations in the form of a multi-objective heuristic optimization described in an LLVM-compliant IR.

Index Terms—Quantum compilation, Quantum Compiler, Algorithm, NSGA-II

I. INTRODUCTION

In recent years, Quantum computing has demonstrated significant potential for performance improvements over classical counterparts. However, the integration of quantum computing into HPC requires addressing several challenges, including the need for efficient compilation and optimization techniques. This paper presents a multi-objective heuristic optimization approach to achieve Pareto-optimality in quantum circuit compilation, enabling the simultaneous optimization of multiple objectives such as execution time, resource usage, and circuit complexity.

2024 IEEE International Conference on Quantum Computing and Engineering (QCE)

QDMI – Quantum Device Management Interface: Hardware-Software Interface for the Munich Quantum Software Stack

Robert Wille
§Chair of Quantum Computing

Abstract—Quantum computing requires a sophisticated hardware-software interface to manage the diverse range of quantum devices and software tools. This paper introduces the Quantum Device Management Interface (QDMI), a hardware-software interface designed to facilitate the integration of quantum devices into the Munich Quantum Software Stack. QDMI provides a unified interface for managing quantum devices, enabling the efficient execution of quantum circuits and the optimization of quantum resources.

2024 IEEE International Conference on Quantum Computing and Engineering (QCE)

A Software Platform to Support Disaggregated Quantum Accelerators

Ercüment Kaya
*Leibniz Supercomputing Center

Abstract—Quantum computing requires a sophisticated hardware-software interface to manage the diverse range of quantum devices and software tools. This paper introduces the Quantum Device Management Interface (QDMI), a hardware-software interface designed to facilitate the integration of quantum devices into the Munich Quantum Software Stack. QDMI provides a unified interface for managing quantum devices, enabling the efficient execution of quantum circuits and the optimization of quantum resources.

2024 IEEE International Conference on Quantum Computing and Engineering (QCE)

Towards a Unified Multi-Target MLIR-Based Compiler: A Heterogeneous Compilation Framework for High-Performance and Quantum Computing Integration

Martín Letras, Jorge Echavarría
Leibniz Supercomputing Center

Abstract—The Munich Quantum Software Stack (MQSS) is a compilation and runtime framework for high-performance computing (HPC) and quantum computing (QC). A unified compilation framework becomes necessary to support the integration of HPC and QC applications. This paper presents a Multi-Level Intermediate Representation (MLIR) framework for the compilation and execution of HPC and QC applications. The framework is designed to support the integration of HPC and QC applications, enabling the efficient execution of quantum circuits and the optimization of quantum resources.

2024 IEEE International Conference on Quantum Computing and Engineering (QCE)

The Munich Quantum Software Stack: Connecting End Users, Integrating Diverse Quantum Technologies, Accelerating HPC

LUKAS
GmbH, Chair of Quantum Computing

Abstract—The Munich Quantum Software Stack (MQSS) is a compilation and runtime framework for high-performance computing (HPC) and quantum computing (QC). A unified compilation framework becomes necessary to support the integration of HPC and QC applications. This paper presents a Multi-Level Intermediate Representation (MLIR) framework for the compilation and execution of HPC and QC applications. The framework is designed to support the integration of HPC and QC applications, enabling the efficient execution of quantum circuits and the optimization of quantum resources.

2024 IEEE International Conference on Quantum Computing and Engineering (QCE)

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

Jorge Echavarría
Leibniz Supercomputing Center

Abstract—The Munich Quantum Software Stack (MQSS) is a compilation and runtime framework for high-performance computing (HPC) and quantum computing (QC). A unified compilation framework becomes necessary to support the integration of HPC and QC applications. This paper presents a Multi-Level Intermediate Representation (MLIR) framework for the compilation and execution of HPC and QC applications. The framework is designed to support the integration of HPC and QC applications, enabling the efficient execution of quantum circuits and the optimization of quantum resources.

➤ Port

- ❖ Software abstraction representing any **input** or **output** component controlling qubits
- ❖ It allows a **hardware vendor** to provide relevant actuation **knobs they wish to expose** to the user in order to manipulate and observe qubits³

³ While hiding the complexities of the device's underlying technology

➤ Waveform

- ❖ **Time-dependent envelope** that can be used to **emit** signals on an output port or **receive** signals from an input port

➤ Frame

❖ A software abstraction that acts as^{4,5}:

- **Clock** within the quantum program with its time being incremented on each usage
- A stateful carrier signal defined by a **frequency** and **phase**

⁴ When **transmitting** signals to the qubit, a frame determines: a) **time** at which the waveform envelope is emitted, b) its carrier **frequency**, and c) its **phase** offset

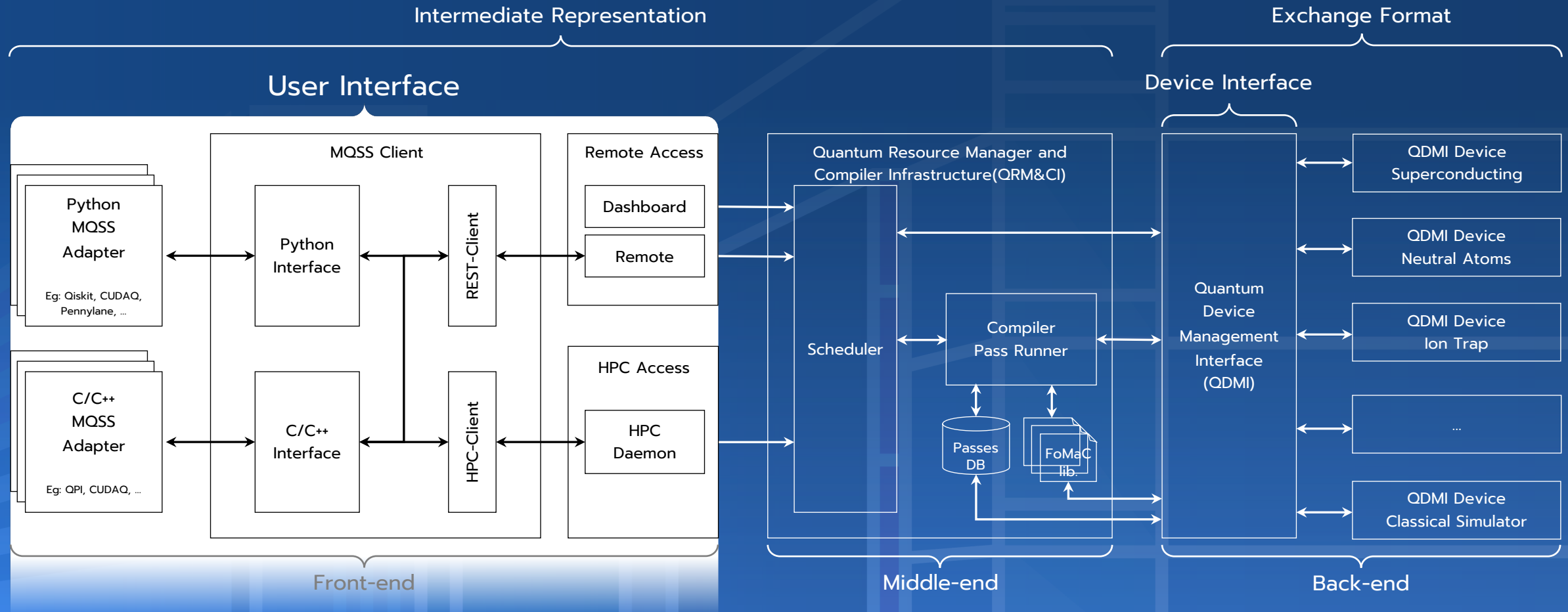
⁵ When **capturing** signals from a qubit, at minimum a frame determines the time at which the signal is captured

➤ Challenges

- ❖ User Interface
- ❖ Device Interface
- ❖ Intermediate Representation
- ❖ Exchange Format

➤ Abstractions

- ❖ Frame
- ❖ Waveform
- ❖ Port



Current MQSS Adapters



Upcoming MQSS Adapters with pulse-level support



Calibration block

```
1 from braket.aws import AwsDevice
2 from braket.devices import Devices
3 from braket.ir.openqasm import Program
4
5 openpulse_script = """
6 OPENQASM 3.0;
7 cal {
8     bit[1] psb;
9     waveform my_waveform = gaussian(12.0ns, 3.0ns, 0.2, false);
10    play(Transmon_25_charge_tx, my_waveform);
11    psb[0] = capture_v0(Transmon_25_readout_rx);
12 }
13 """
14
15 program = Program(source=openpulse_script)
16
17 device = AwsDevice(Devices.Rigetti.Ankaa3)
18 task = device.run(program, shots=100)
```

cal { ... }

- The **cal** (calibration) block is where you describe pulses and measurements directly (rather than high-level gates)
- Anything inside here is interpreted as a pulse program

Calibration block (OPENQASM 3.0 `cal { ... }`)

```
1 from braket.aws import AwsDevice
2 from braket.devices import Devices
3 from braket.ir.openqasm import Program
4
5 openpulse_script = ""
6 OPENQASM 3.0;
7 cal {
8     bit[1] psb;
9     waveform my_waveform = gaussian(12.0ns, 3.0ns, 0.2, false);
10    play(Transmon_25_charge_tx, my_waveform);
11    psb[0] = capture_v0(Transmon_25_readout_rx);
12 }
13 ""
14
15 program = Program(source=openpulse_script)
16
17 device = AwsDevice(Devices.Rigetti.Ankaa3)
18 task = device.run(program, shots=100)
```

`cal { ... }`

- The **cal** (calibration) block is where you describe pulses and measurements directly (rather than high-level gates)
- Anything inside here is interpreted as a pulse program

✓ OPENQASM 3.0 `cal { ... }` → OpenPulse (waveform)

```
1 {
2   "openpulse_version": "1.0",
3   "backend": "quantum_accelerator_v1",
4   "pulse_library": [
5     {
6       "name": "my_waveform",
7       "samples": [
8         [0.000, 0.000],
9         .
10        .
11        .
12        [0.000, 0.000]
13      ]
14    }
15  ],
```

Calibration block (OPENQASM 3.0 `cal { ... }`)

```
1 from braket.aws import AwsDevice
2 from braket.devices import Devices
3 from braket.ir.openqasm import Program
4
5 openpulse_script = ""
6 OPENQASM 3.0;
7 cal {
8     bit[1] psb;
9     waveform my_waveform = gaussian(12.0ns, 3.0ns, 0.2, false);
10    play(Transmon_25_charge_tx, my_waveform);
11    psb[0] = capture_v0(Transmon_25_readout_rx);
12 }
13 ""
14
15 program = Program(source=openpulse_script)
16
17 device = AwsDevice(Devices.Rigetti.Ankaa3)
18 task = device.run(program, shots=100)
```

`cal { ... }`

- The **cal** (calibration) block is where you describe pulses and measurements directly (rather than high-level gates)
- Anything inside here is interpreted as a pulse program

✓ OPENQASM 3.0 `cal { ... }` → OpenPulse (frame)

```
19  "frames": [
20    {
21      "name": "q25_tx_frame",
22      "frame": {
23        "port": "Transmon_25_charge_tx",
24        "frequency": 5.0e9,
25        "phase": 0.0
26      }
27    },
28    {
29      "name": "q25_rx_frame",
30      "frame": {
31        "port": "Transmon_25_readout_rx",
32        "frequency": 6.5e9,
33        "phase": 0.0
34      }
35    }
36  ],
```

Calibration block (OPENQASM 3.0 `cal { ... }`)

```
1 from braket.aws import AwsDevice
2 from braket.devices import Devices
3 from braket.ir.openqasm import Program
4
5 openpulse_script = ""
6 OPENQASM 3.0;
7 cal {
8     bit[1] psb;
9     waveform my_waveform = gaussian(12.0ns, 3.0ns, 0.2, false);
10    play(Transmon_25_charge_tx, my_waveform);
11    psb[0] = capture_v0(Transmon_25_readout_rx);
12 }
13 ""
14
15 program = Program(source=openpulse_script)
16
17 device = AwsDevice(Devices.Rigetti.Ankaa3)
18 task = device.run(program, shots=100)
```

`cal { ... }`

- The **cal** (calibration) block is where you describe pulses and measurements directly (rather than high-level gates)
- Anything inside here is interpreted as a pulse program

✓ OPENQASM 3.0 `cal { ... }` → OpenPulse (port)

```
38 "schedule": [
39   {
40     "name": "drive_q25",
41     "t0": 0,
42     "port": "Transmon_25_charge_tx",
43     "waveform": "my_waveform",
44     "frame": "q25_tx_frame"
45   },
46   {
47     "name": "acquire_q25",
48     "t0": 120,
49     "duration": 240,
50     "port": "Transmon_25_readout_rx",
51     "frame": "q25_rx_frame",
52     "memory_slot": 0,
53     "mode": "v0"
54   }
55 ]
56 }
```

Native

```
1 from bracket.pulse import ArbitraryWaveform, ConstantWaveform
2
3 cst_wfm = ConstantWaveform(length=1e-7, iq=0.1)
4 arb_wf = ArbitraryWaveform(amplitudes=np.linspace(0, 100))
5 gaussian_waveform = GaussianWaveform(1e-7, 25e-9, 0.1)
6
7 pulse_sequence = (
8     PulseSequence()
9     .play(drive_frame, waveform)
10    .capture_v0(readout_frame)
11 )
12
13 start_length=12e-9
14 end_length=2e-7
15 lengths = np.arange(start_length, end_length, 12e-9)
16
17 tasks = [
18     device.run(pulse_sequence, shots=100, inputs={"length": length})
19     for length in lengths
20 ]
```

Native

```
1 from bracket.pulse import ArbitraryWaveform, ConstantWaveform
2
3 cst_wfm = ConstantWaveform(length=1e-7, iq=0.1)
4 arb_wf = ArbitraryWaveform(amplitudes=np.linspace(0, 100))
5 gaussian_waveform = GaussianWaveform(1e-7, 25e-9, 0.1)
6
7 pulse_sequence = (
8     PulseSequence()
9     .play(drive_frame, waveform)
10    .capture_v0(readout_frame)
11 )
12
13 start_length=12e-9
14 end_length=2e-7
15 lengths = np.arange(start_length, end_length, 12e-9)
16
17 tasks = [
18     device.run(pulse_sequence, shots=100, inputs={"length": length})
19     for length in lengths
20 ]
```



```
#include <qpi.h>

int main(){
    do{
        void* results = malloc(size);
        pulse_vqe_quantum_kernel(&results, nshots, &
                                ↪parameters);
        parameters = calculate_new_parameters(&results,
                                ↪ parameters)
    }while( stop_condition == false );
    return 0;
}
```

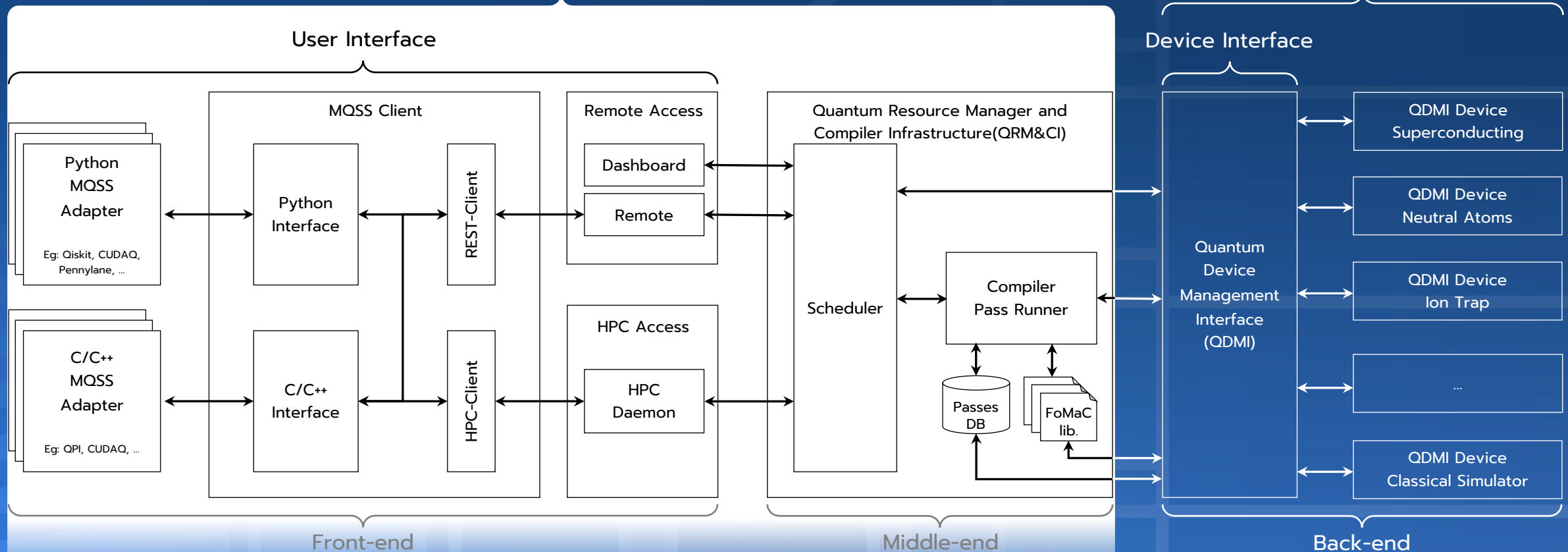
```
void pulse_vqe_quantum_kernel(void *results, int nshots
    ↪, Parameters *p) {

    QCircuit circuit;
    qCircuitBegin(&circuit)
    QClassicalRegisters cr;
    qInitClassicalRegisters(&cr, 2);
    // we begin with X on both qubits
    qX(0);
    qX(1);
    // define the waveforms
    qWaveform(&waveform_1, p->amps_1);
    qWaveform(&waveform_2, p->amps_2);
    qWaveform(&waveform_3, p->amps_3);
    // apply the waves
    qPlayWaveform(qb1_drive_port, waveform_1);
    qPlayWaveform(qb2_drive_port, waveform_2);
    // do the frame changes
    qFrameChange(qb1_drive_port, freq_qb1, p->phase_qb1
    ↪);
    qFrameChange(qb2_drive_port, freq_qb2, p->phase_qb2
    ↪);
    // apply the entangling pulse
    qPlayWaveform(qb1_qb2_coupler_port, waveform_3);
    // measure
    qMeasure(0, 0);
    qMeasure(1, 1);
    qCircuitEnd();

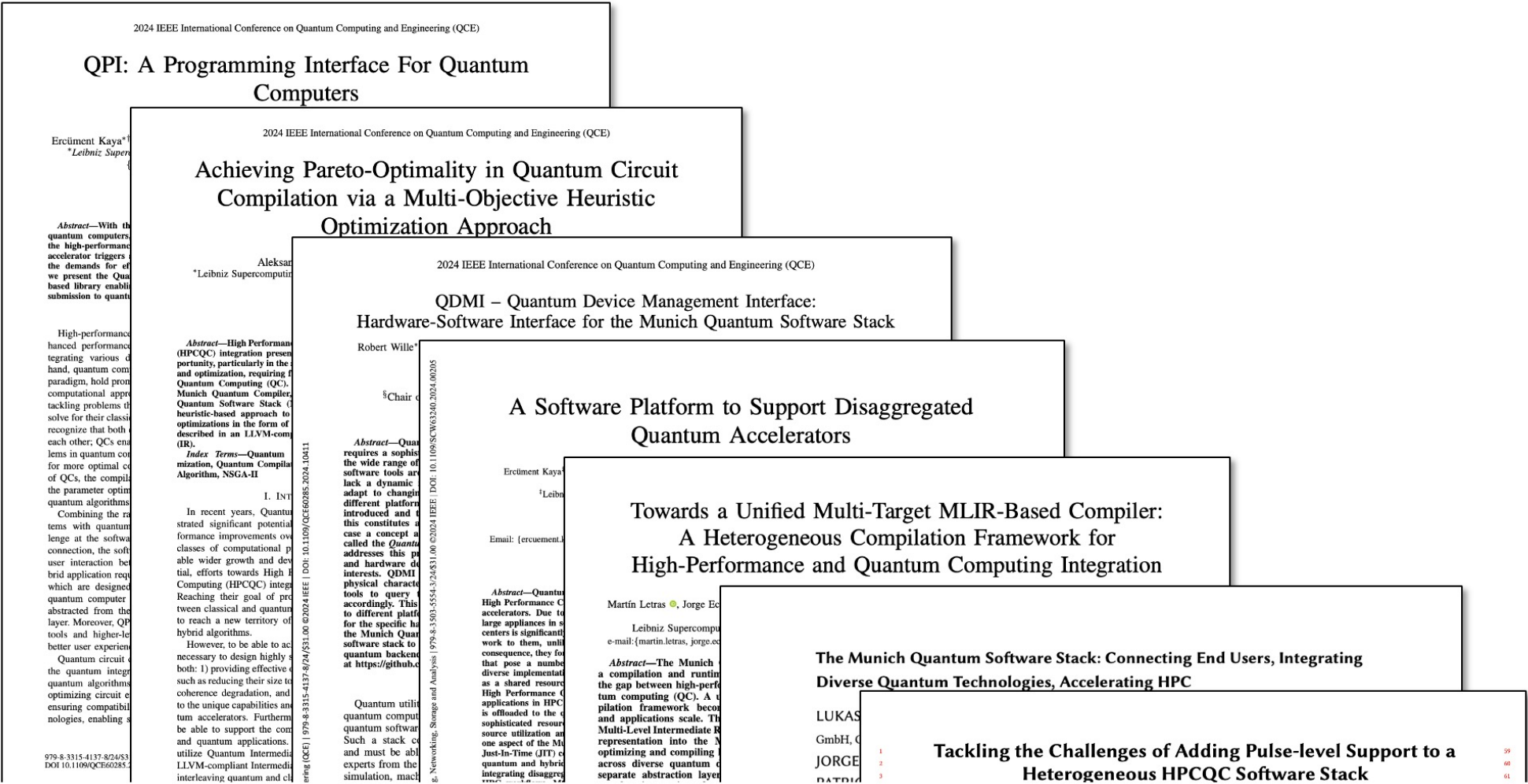
    if(!qExecute(dev, circuit, nshots))
        QuantumResult* results = qRead(circuit);

    qCircuitFree(circuit);
}
```

Intermediate Representation

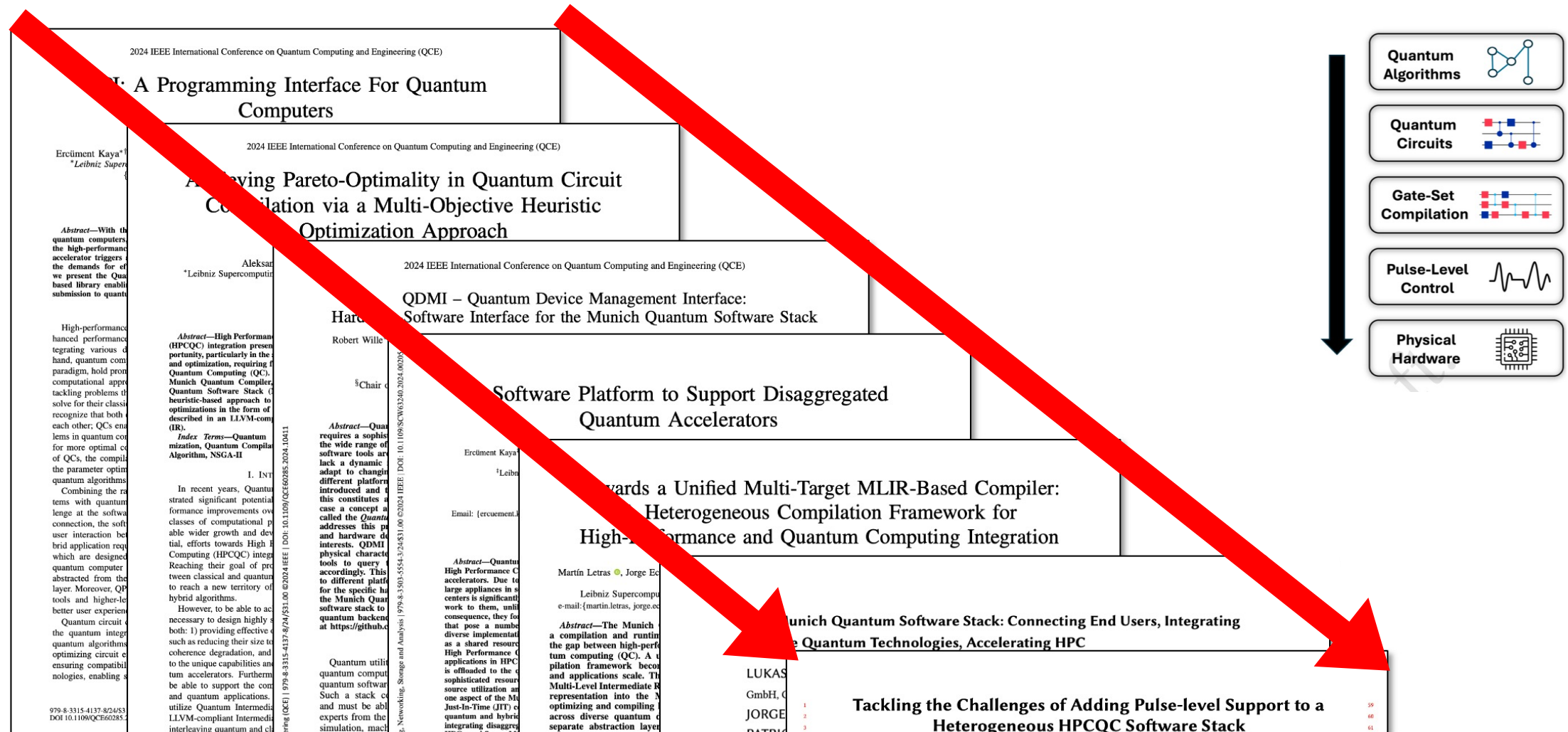


MQSS Pulse
Intermediate Representation – The Path to Pulse-level Control



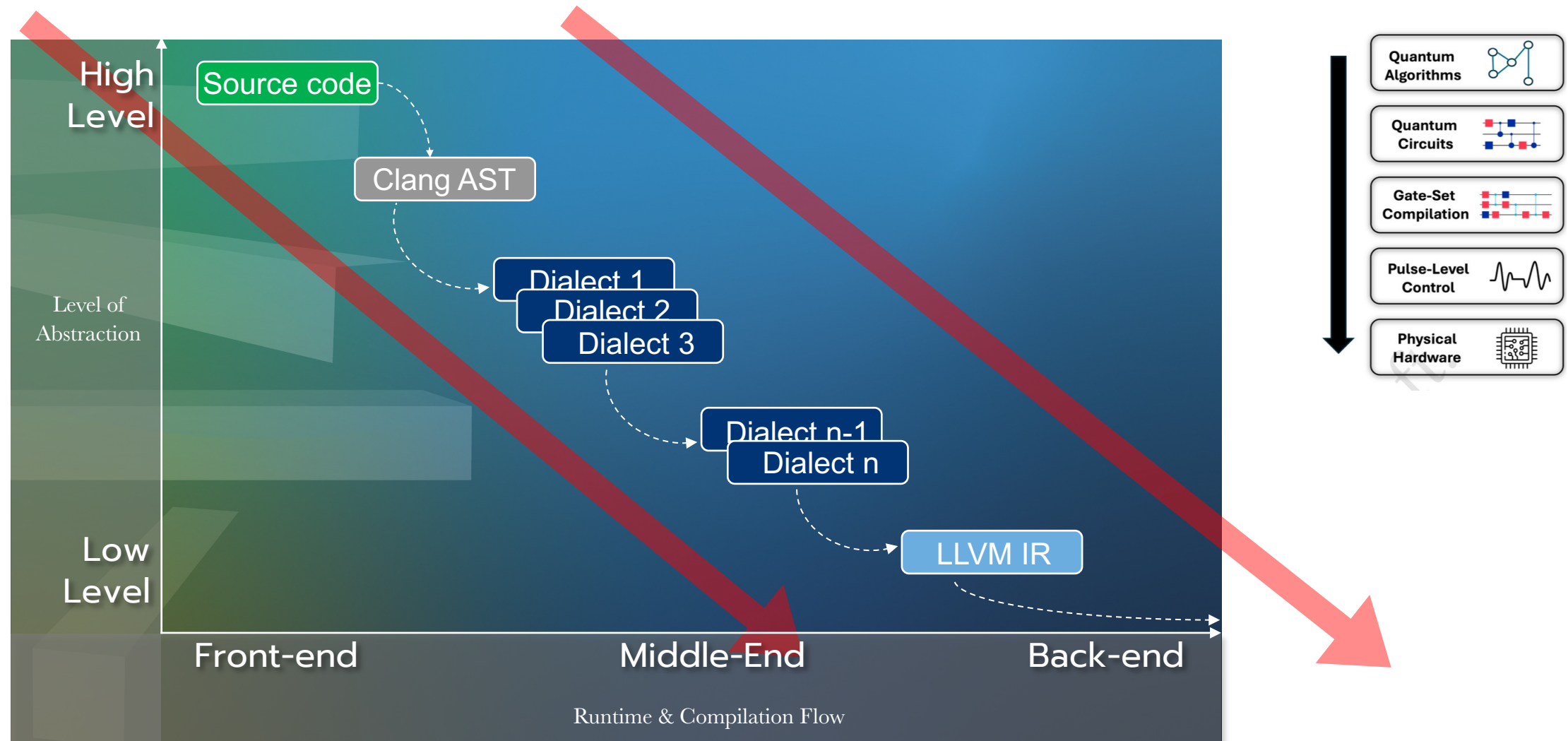
MQSS Pulse

Intermediate Representation – The Path to Pulse-level Control



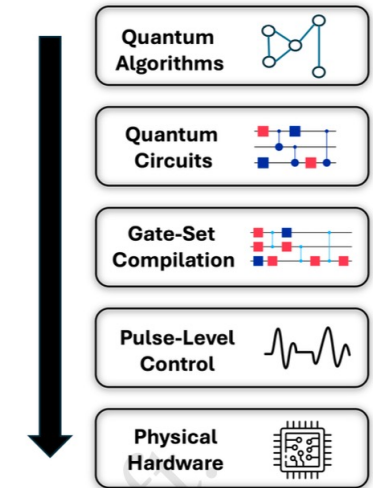
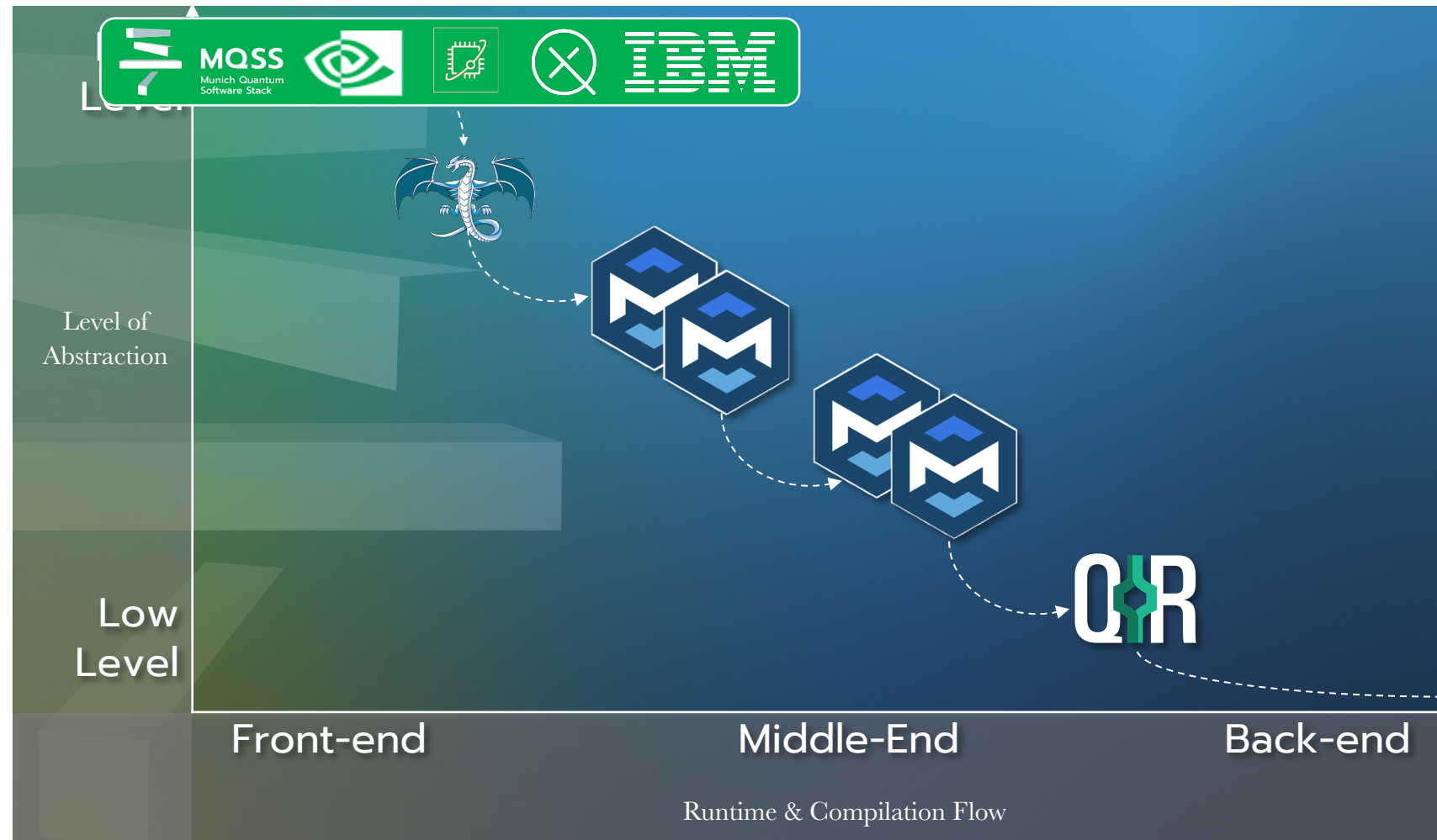
MQSS Pulse

Intermediate Representation – The Path to Pulse-level Control



MQSS Pulse

Intermediate Representation – The Path to Pulse-level Control



➤ “Traditional” pulse compilation workflow:

1. Quantum operations are translated into pulse-level operations
2. Pulse-level operations are optimized and scheduled
3. Optimized and scheduled pulse-level operations are lowered to hardware primitives

IBM's pulse

- IQM's Quantum Engine Compiler (`qe_compiler`) supports the following MLIR dialects:
 - ❖ OpenQASM3 IR (`oq3`)
 - ❖ Quantum IR/dialect (`quir`)
 - ✓ *Consistent with QDMI Operations²*
 - ❖ Pulse IR (`pulse`)
 - ✓ *Consistent with OpenPulse*
- Seamless translation from gate-level quantum circuits into sequences of pulse operations on frames using MLIR pulse calibrations that the compiler receives as input

² Quantum Operations \equiv Gates & Measurements

IBM's pulse

```
module {
  pulse.def @waveform_1 { // Define waveforms
    pulse.waveform amplitudes = %amplitudes_in : vector.vector<5
      ↪xi32>
  }
  pulse.def @waveform_2 { ... }
  pulse.def @waveform_3 { ... }

  // Main pulse-level kernel
  pulse.sequence @pulse_vqe_quantum_kernel(
    %drive0: !!pulse.mixed_frame, %drive1: !pulse.mixed_frame,
    %coupler: !pulse.mixed_frame, %freq: f64,
    %phase: f64) -> i1
  attributes { pulse.argPorts = ["q0-drive-port",
    "q1-drive-port", "q0q1-coupler-port", "", ""],
    pulse.args = ["q0-drive-frame", "q1-drive-frame",
    "coupler-frame", "freq", "phase"]} {

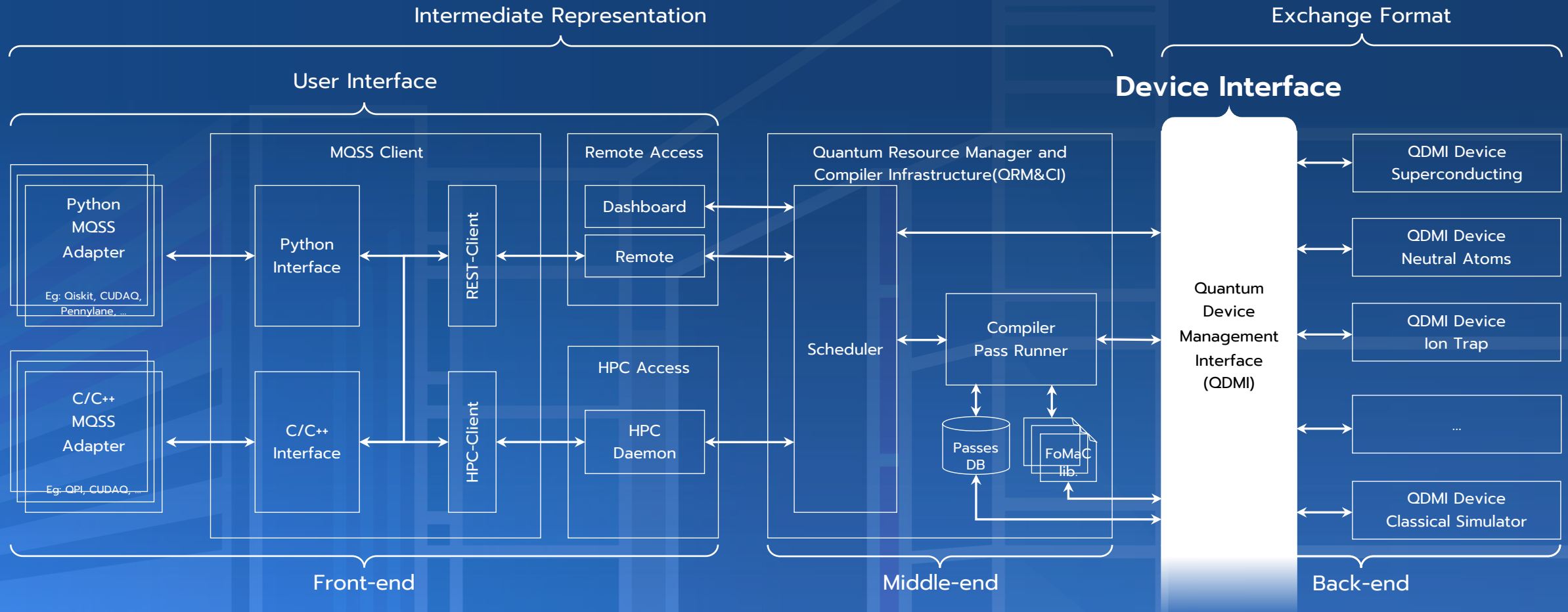
    // 2. Waveform constants
    %wf1 = pulse.waveform.amplitudes @waveform_1
    %wf2 = pulse.waveform.amplitudes @waveform_2
    %wf3 = pulse.waveform.amplitudes @waveform_3

    // 3. Apply single-qubit pulses
    pulse.play(%drive0, %wf1): (!pulse.mixed_frame, !pulse.waveform
      ↪)
    pulse.play(%drive1, %wf2): (!pulse.mixed_frame, !pulse.waveform
      ↪)
```

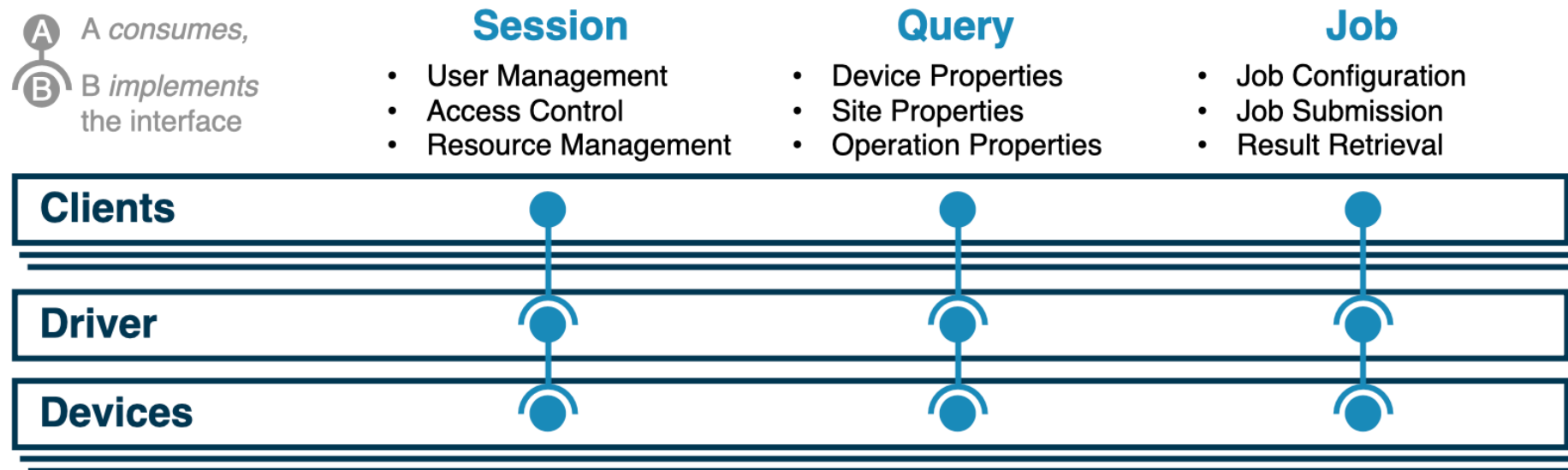
```
// 4. Frame changes
pulse.frame_change(%drive0, %freq, %phase) : (!pulse.
  ↪mixed_frame, f64, f64)
pulse.frame_change(%drive1, %freq, %phase) : (!pulse.
  ↪mixed_frame, f64, f64)
```

```
// 5. Entangling pulse
pulse.play(%coupler, %wf3)
: (!pulse.mixed_frame, !pulse.waveform)
```

```
// 6. Measurement on qubit0
%wf_r = pulse.waveform.constant @readout_pulse
pulse.play (%readout0, %wf_r) : (!pulse.mixed_frame, !pulse.
  ↪waveform)
%m0 = pulse.capture (%capture0): (!pulse.mixed_frame) -> i1
pulse.return %m0, %m1 : i1, i1
}
```



The Structure of the QDMI



Opaque Pointers in QDMI

- Objects such as **sessions**, **jobs**, **devices**, **sites**, and **operations** are opaque pointers
- Pointers to a data structure that is not defined in the header file
 - ❖ The actual implementation is only known to the ~~entity~~^{hardware provider} that defines the object
 - ❖ They allow changing the internal representation of the object without breaking the client code
 - ❖ Opaque pointers effectively serve as type-safe IDs that are checked statically by the compiler

→ *More stable and easier to maintain interface*

Opaque Pointers in QDMI

◆ QDMI_Site

```
typedef struct QDMI_Site_impl_d* QDMI_Site
```

A handle for a site.

An opaque pointer to an implementation of the QDMI site concept. A site is a place that can potentially hold a qubit. In case of superconducting qubits, sites can be used synonymously with qubits. In case of neutral atoms, sites represent individual traps that can confine atoms. Those atoms are then used as qubits. To this end, sites are generalizations of qubits that denote locations where qubits can be placed on a device. Each implementation of the [QDMI Device Interface](#) defines the actual implementation of the concept.

A simple example of an implementation is a struct that merely contains the site ID, which can be used to identify the site.

```
struct QDMI_Site_impl_d {  
    size_t id;  
};
```


Opaque Pointers in QDMI

◆ QDMI_Site

```
typedef struct QDMI_Site_impl_d* QDMI_Site
```

A handle for

An opaque

qubits, site

atoms are

implement

A simple e

◆ QDMI_Operation

```
typedef struct QDMI_Operation_impl_d* QDMI_Operation
```

A handle for an operation.

An opaque pointer to an implementation of the QDMI operation concept. An operation generally represents any instruction that can be executed on a device. This includes gates, measurements, classical control flow elements, movement of qubits, pulse-level instructions, etc. Each implementation of the [QDMI Device Interface](#) defines the actual implementation of the concept.

A simple example of an implementation is a struct that merely contains the name of the operation, which can be used to identify the operation.

```
struct  
size_  
};
```

```
struct QDMI_Operation_impl_d {  
    std::string name;  
};
```

Opaque Pointers in QDMI

◆ QDMI_Site

```
typedef struct QDMI_Site_impl_d* QDMI_Site
```

A handle f

◆ QDMI_Operation

An opaque

qubits, site

atoms are

implement

A simple e

```
typedef struct QDMI_Operation_impl_d* QDMI_Operation
```

A handle f

An opaque

on a device

implement

A simple e

◆ QDMI_Job

```
typedef struct QDMI_Job_impl_d* QDMI_Job
```

A handle for a client-side job.

An opaque pointer to a type defined by the driver that encapsulates all information about a job submitted to a device by a client.

Remarks

Implementations of the underlying type will want to store the device handle used to create the job in the job handle to be able to access the device when needed.

```
struct  
size_t  
};
```

```
struct  
std:  
};
```

Opaque Pointers in QDMI

◆ QDMI_Site

```
typedef struct QDMI_Site_impl_d* QDMI_Site
```

A handle for

◆ QDMI_Operation

An opaque

qubits, site

atoms are

implement

```
typedef struct QDMI_Operation_impl_d* QDMI_Operation
```

A handle for

◆ QDMI_Job

An opaque

on a device

implement

```
typedef struct QDMI_Job_impl_d* QDMI_Job
```

A handle for

◆ QDMI_Device_Job

A simple e

```
typedef struct QDMI_Device_Job_impl_d* QDMI_Device_Job
```

Remark

A handle for a device job.

Implem

device \

An opaque pointer to a type defined by the device that encapsulates all information about a job on a device.

```
struct  
size_t  
};
```

```
struct  
std:  
};
```

Opaque Pointers in QDMI

QDMI_PULSE_CHANNEL

- ChannelType: e.g., DriveChannel, ReadoutChannel, ...
- Size_t: Id
- Constraints

QDMI_PULSE_PARAMETER

- Name
- Permission (Read-only, R/W)
- Range

QDMI_PULSE_SHAPE: Definition?

Predefined pulse shape:

- Name: "Gaussian"
- formula: str -> "ax²+bx+c"
- parameters: List[QDMI_PULSE_PARAMETER]

QDMI_PULSE_GATE_IMPLEMENTATION

- Pulse program Intermediate Representation (e.g., OpenPulse)
- OR QDMI_PULSE_SHAPE with parameter values set

New potential candidates

Enums in QDMI

- QDMI relies on enumerations to define properties for **sessions**, **jobs**, **devices**, **sites**, and **operations**
- For each type of property, a corresponding enumeration is defined
- We do not define a separate function for each property → the value of a property is retrieved by calling a single function with the property enumeration as an argument
- QDMI's enumerations allow adding new properties without breaking the interface
- **If a new property is added, the corresponding enumeration can be added to the interface without changing the existing functions**

→ *More compact, extensible, and predictable interface*

Enums in QDMI

QDMI_DEVICE_PROPERTY_NAME	0	<code>char*</code> (string) The name of the device.
QDMI_DEVICE_PROPERTY_VERSION	1	<code>char*</code> (string) The version of the device.
QDMI_DEVICE_PROPERTY_STATUS	2	QDMI_Device_Status The status of the device.
QDMI_DEVICE_PROPERTY_LIBRARYVERSION	3	<code>char*</code> (string) The implemented version of QDMI.
QDMI_DEVICE_PROPERTY_QUBITSNUM	4	<code>size_t</code> The number of qubits in the device.
QDMI_DEVICE_PROPERTY_SITES	5	<p><code>QDMI_Site*</code> (QDMI_Site list) The sites of the device.</p> <p>The returned QDMI_Site handles may be used to query site and operation properties. The list need not be sorted based on the QDMI_SITE_PROPERTY_ID.</p>
QDMI_DEVICE_PROPERTY_OPERATIONS	6	<p><code>QDMI_Operation*</code> (QDMI_Operation list) The operations supported by the device.</p> <p>The returned QDMI_Operation handles may be used to query operation properties.</p>

⋮

Pulse-related Enums in QDMI

QDMI_DEVICE_PROPERTY_T

- QDMI_DEVICE_PROPERTY_PULSE_SUPPORT
 - 0: No pulse support
 - 1: Site level (QDMI site)
 - 2: Channel (readout, global, qubit drive, coupler drive, etc.)
- QDMI_DEVICE_PROPERTY_SUPPORTED_PULSE_SHAPE_TYPE
 - 0: Standard (well-known predefined shapes, e.g., Gaussian, parameterized, etc. – defined by a formula definition of parameters: see above)
 - 1: Arbitrary pulse shapes (these are arbitrary-shaped pulses, not defined in a standard way, e.g., a list of pulse amplitudes and phases; see above)
- QDMI_DEVICE_PROPERTY_AVAILABLE_PULSE_SHAPES
 - List(QDMI_PULSE_SHAPE, List(QDMI_SITE or QDMI_CHANNEL)) and the corresponding channel (e.g., drive channel can have Gaussian, readout does not support Gaussian)
- ⋮

QDMI_PROGRAM_FORMAT_T

- QDMI_PROGRAM_FORMAT_OPENPULSE
- QDMI_PROGRAM_FORMAT_QIRPULSE

Non-exhaustive list¹

¹ Property types not mentioned: a) Pulse channel properties, b) Pulse operation properties, and c) Site properties.

Routines in QDMI

```
int QDMI_device_query_device_property ( QDMI_Device      device ,
                                       QDMI_Device_Property prop ,
                                       size_t             size ,
                                       void *            value ,
                                       size_t *          size_ret )
```

Query a device property.

Parameters

- [in] **device** The device to query. Must not be `NULL`.
- [in] **prop** The property to query. Must be one of the values specified for `QDMI_Device_Property`.
- [in] **size** The size of the memory pointed to by `value` in bytes. Must be greater or equal to the size of the return type specified for `prop`, except when `value` is `NULL`, in which case it is ignored.
- [out] **value** A pointer to the memory location where the value of the property will be stored. If this is `NULL`, it is ignored.
- [out] **size_ret** The actual size of the data being queried in bytes. If this is `NULL`, it is ignored.

⋮

Pulse-related Routines in QDMI

```
int QDMI_device_query_device_property ( QDMI_Device      device,
                                       QDMI_Device_Property prop,
                                       size_t             size,
                                       void *             value,
                                       size_t *           size_ret )
```

```
// First call to the function to get the size of memory required for all the sites
size_t size_ret;
QDMI_device_query_device_property(
    device,
    QDMI_DEVICE_PROPERTY_SITES, /* QDMI enum value */
    NULL,
    NULL,
    &size_ret
);
```

```
// Second call to the function to get the QDMI_Sites
void* value = malloc(size);
QDMI_device_query_device_property(
    device,
    QDMI_DEVICE_PROPERTY_SITES, /* QDMI enum value */
    size,
    value,
    NULL
);
```


Join the conversation

➤ <https://tiny.badw.de/gSkYAK>



✨ Pulse-Level Support #171

OpenFeature1 / 6



mnfarooqi opened on Jun 5 · edited by mnfarooqi

Edits Collaborator

What's the problem this feature will solve?

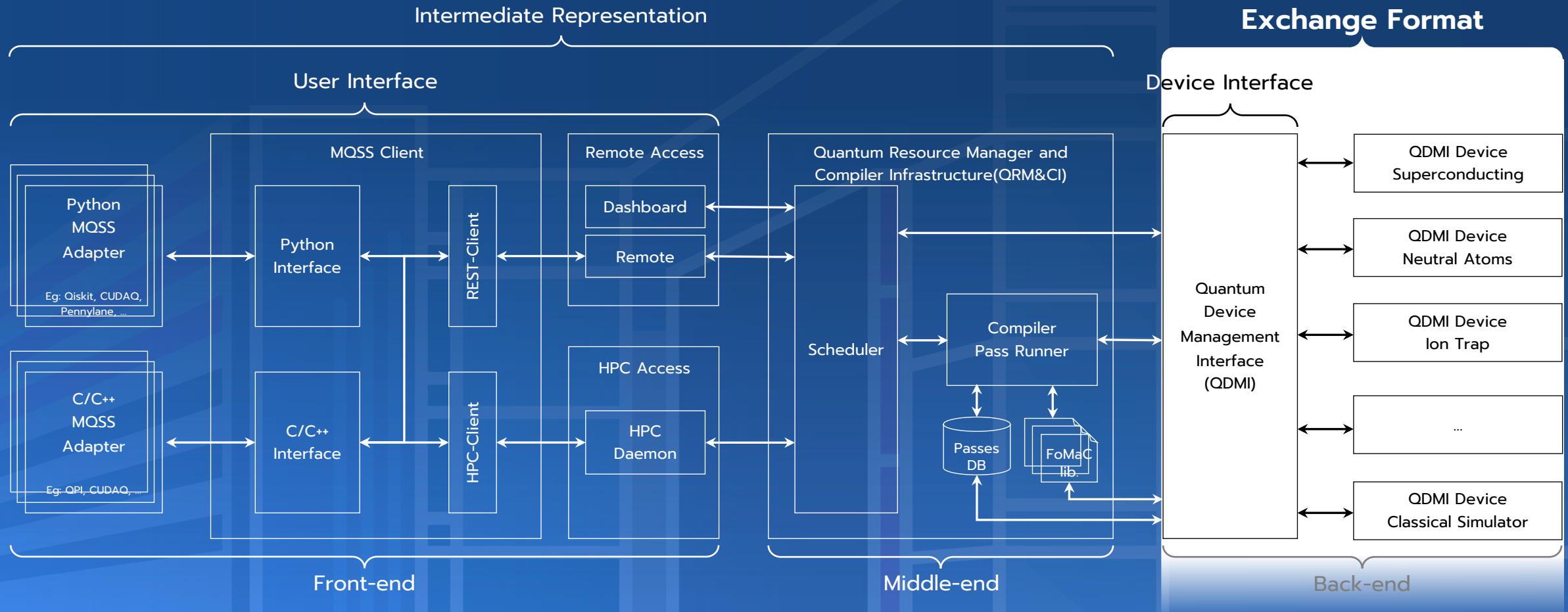
This issue focuses on extending the QDMI to include pulse level support, as discussed offline. Exposing the pulse-level support will open up a range of new opportunities, such as calibration and enabling a pulse interface for advanced users. It will also enable the development of software that can optimise user programs at a lower level of abstraction.

Based on some preliminary discussions, the issue will be divided into the following sub-issues to facilitate subsequent discussions and PRs.

- 👉 ✨ [Querying if the pulse interface is supported #172](#)
- 👉 ✨ [Data types for pulse representation #173](#)
- 👉 ✨ [Query interface to support pulses at the QDMI Site abstraction level #174](#)
- 👉 ✨ [Interface to set default pulse implementation for operations #175](#)
- 👉 ✨ [Query interface to support pulses at the abstraction level of a channel #176](#)
- 👉 ✨ [Pulse Submission Interface #177](#)

Open questions:

Do the supported pulse shapes on a device change from one site/channel to another, or are they global to the device?



QIR-Pulse

- The LRZ as a member of the steering committee of the **QIR Alliance** will lead a workstream for creating **QIR-Pulse**
 - ❖ We suggest creating **Pulse Profile** and modify the output schemas accordingly
 - ❖ The **QDMI** specification will adopt **QIR-Pulse** as the default pulse exchange format but it will also support OpenPulse and IQM-Pulse

Prototypical extension
to the QIR specification
enabling pulse-level
support

```
; ModuleID = 'my_pulse'

%Qubit = type opaque
%Port = type opaque
%Waveform = type opaque
%Frame = type opaque

define void @my_pulse(float* %amps, float %freq, float %phase) #0
    ↪{
    call void @__quantum__pulse__waveform__body(%Wave* %waveform0,
        ↪float* %amps)
    call void @__quantum__pulse__waveform_play__body(%Port* %port0,
        ↪%Wave* waveform0)
    call void @__quantum__pulse__frame_change__body(%Port* port0, %
        ↪freq, %phase)
    call void @__quantum__pulse__delay__body(%Port* port0, 1024)
    call void @__quantum__qis__mz__body(%Qubit* inttoptr (i64 0 to %
        ↪Qubit*), %Result* inttoptr (i64 0 to %Result*)) #1
    call void @__quantum__qis__mz__body(%Qubit* inttoptr (i64 1 to %
        ↪Qubit*), %Result* inttoptr (i64 1 to %Result*)) #1
    ret void
    }

declare %Waveform* @__quantum__pulse__waveform__body(float, float
    ↪*)
declare void @__quantum__pulse__waveform_play__body(%Port*, %
    ↪Waveform*)
declare %Frame* @__quantum__pulse__frame_change__body(%Port*,
    ↪double)
declare void @__quantum__pulse__delay__body(%Frame*, int)

attributes #0 = { "entry_point" "output_labeling_schema" "
    ↪qir_profiles"="pulse" "required_num_ports"="1" }
```

- **Goal:** Remove obstacles to pulse-level quantum control in HPCQC integration with the MQSS
- **Pulse abstractions:** Ports, frames, and waveforms supported at front-end, middle-end, and back-end of a heterogeneous HPCQC software stack similar to MQSS
- **Challenges:** User interface, device interface, intermediate representation, and exchange formats require pulse abstractions support
- **Compatibility:** Native pulse representation across the stack preserving HPC scheduling/execution models
- **Impact:** Enables pulse-aware hybrid workloads (calibration, custom waveforms) and new quantum-accelerated algorithms for near-term hardware

MQSS Pulse

Academic and Industry Leaders Supporting MQSS Pulse

