LAMINA: An MLIR-Based Translation Library for Heterogeneous Quantum-Classical Compilation

Marco De Pascale[1,2]
Leibniz Supercomputing Centre, München, Germany

Mario Hernández Vera[1,2]
Leibniz Supercomputing Centre, München, Germany

Jorge Echavarria
Leibniz Supercomputing Centre, München, Germany

Muhammad Nufail Farooqi
Leibniz Supercomputing Centre, München, Germany

Martin Schulz
Technical University of Munich, Germany

Laura Schulz
Argonne National Laboratory, Lemont, IL, USA

Abstract

Quantum computing is increasingly integrated into High-Performance Computing (HPC) environments, where quantum processors act as specialized accelerators within hybrid workflows. The Munich Quantum Software Stack (MQSS) – a unified compilation and runtime framework for hybrid quantum–classical computing – provides the foundation for this integration. However, the growing heterogeneity of applications demands more flexible compilation tools. This work introduces an Multi-Level Intermediate Representation (MLIR)-based translation library that extends MQSS by enabling the conversion of CUDA-Quantum (CUDA-Q) (`quake`) dialects into machine learning–oriented MLIR representations compatible with modern compiler ecosystems. Leveraging MLIR's dialect-driven design, the library enables hardware-agnostic transformations, device-specific optimizations, and seamless integration with MQSS components. The proposed approach bridges quantum compilation and contemporary machine learning frameworks, facilitating GPU-accelerated circuit simulation, hybrid quantum–classical workflows, and heterogeneous execution, thereby advancing a unified compiler infrastructure for quantum computing.

2012 ACM Subject Classification Computer systems organization → Quantum computing; Software and its engineering → Just-in-time compilers; Software and its engineering → Retargetable compilers

Keywords and phrases HPCQC, MLIR, Quantum Computing, Heterogeneous Computing

Digital Object Identifier 10.4230/OASIcs.PARMA-DITAM.2026.5

Supplementary Material *Software (Source Code)*: `https://github.com/QCT-UEA-management/quake2iree.git` [7], archived at `swh:1:snp:c7b8f26bbeb0273ac7e029a8bba5845db219bdec`

Funding This work is supported by the German Federal Ministry of Research, Technology and Space (BMFTR) with the grants 13N15689 (DAQC), 13N16063 (Q-Exa), 13N16078 (MUNIQC-Atoms), 13N16187 (MUNIQC-SC), 13N16690 (Euro-Q-Exa), 13N16894 (MAQCS), European fundings 101136607 (CLARA), 101114305 (Millenion), 10111394-6 (OpenSuperQPlus), 101194491 (QEX), and the Bavarian State Ministry of Science and the Arts (StMWK) through funding, as part of MQV, Q-DESSI.

[1] Corresponding authors
[2] These authors contributed equally to this work.

Acknowledgements The authors acknowledge Fabian Mora Cordero and Sunita Chandrasekaran from the University of Delaware for the initial discussion on this topic. The authors thank the anonymous reviewers for their constructive feedback and insightful suggestions.

1 Introduction

Quantum Computing (QC) is rapidly evolving from experimental systems into a critical component of modern HPC ecosystems. In this paradigm, quantum processors are envisioned as specialized accelerators that complement classical compute nodes, much like Graphical Processing Units (GPUs) or FPGAs, to solve computationally demanding subproblems in scientific and industrial applications. Realizing this hybrid model requires a robust software infrastructure that bridges high-level quantum programming frameworks with diverse quantum and classical hardware back-ends. The MQSS [4] has been developed to address this need. It provides a modular, open-source software ecosystem that integrates quantum devices into HPC environments and enables users to execute hybrid quantum-classical workflows efficiently through unified compilation, scheduling, and runtime management. Building on this foundation, the MQSS is evolving toward a fully dialect-agnostic, MLIR-based compilation framework that can integrate multiple quantum programming front-ends within a unified infrastructure. At present, MQSS supports the `quake` and `catalyst` dialects through its Quantum Resource Manager & Compiler Infrastructure (QRM & CI) component, which hosts the current `quake`-based compiler implementation. The QRM & CI repository provides the foundation for this compiler and demonstrates the ongoing transition toward a more general, extensible MLIR-driven design.

Despite this progress, the increasing heterogeneity of both hardware and software in quantum computing poses new challenges. Current compilers and runtime systems must not only support a variety of quantum technologies – such as superconducting qubits, trapped ions, and neutral atoms – but also interface with heterogeneous classical resources, such as CPUs, GPUs, and distributed HPC systems. Moreover, quantum circuit simulation and hybrid quantum–classical workflows typically require high-performance classical acceleration, particularly on GPU-based systems [13]. Existing compilation approaches, however, are typically tailored to specific platforms and lack the flexibility to dynamically target or optimize for different architectures and technologies. This limits their capacity to adapt dynamically to heterogeneous HPC infrastructures, preventing the efficient utilization of GPUs and other accelerators essential for quantum circuit simulation.

In addition to software and architectural heterogeneity, practical constraints from the physics of quantum simulation must be considered when targeting CPU or GPU back-ends. For state-vector simulators, the memory requirement scales as $M \approx 16 \times 2^n$ bytes for n qubits in double precision, limiting a 40 GB GPU to about 31 qubits and requiring distributed memory for larger systems; scalability here is bounded mainly by qubit count and inter-device bandwidth. Tensor-network methods become advantageous when the simulated states exhibit limited entanglement. Quantum states obeying the so-called *area law* for entanglement entropy – where subsystem entropy grows at most with boundary size – can be efficiently approximated by tensor-network states with moderate bond dimension [6]. In this regime, the memory cost of a matrix product state simulator scales as $O(nd\chi^2)$, with n the number of qubits, $d = 2$ the local dimension, and χ the bond dimension controlling retained singular values. For fixed χ, the memory grows linearly with n, enabling simulations of around 60 qubits on a single high-end GPU – such as an NVIDIA H100 with 96 GB of memory – compared to roughly 33 qubits for state-vector methods [13]. Because χ and accuracy depend

on circuit entanglement, highly entangled states still demand exponentially larger resources. Thus, circuits that generate weak or localized entanglement can be efficiently simulated on GPUs or CPUs using tensor-network techniques. In contrast, strongly correlated circuits remain limited by memory and communication costs.

To address these challenges, we observe that both Machine Learning (ML)-based models and quantum algorithms can be expressed in terms of linear-algebraic operations. This shared mathematical structure enables the use of those machine learning compilers such as Intermediate Representation Execution Environment (IREE) [15], which take tensor-based intermediate representations of ML models and, through successive MLIR passes, automatically map its execution to different hardware back-ends, including GPUs. Leveraging this commonality, we propose a new MLIR-based translation library for MQSS with the capability to translate quantum programs expressed in the `quake` dialect into a combination of MLIR dialects compatible with the IREE compiler. This extension leverages the modular, dialect-driven design of MLIR to enable hardware-agnostic transformations and produce intermediate representations compatible with IREE, which in turn performs device-specific optimizations and code generation. By integrating seamlessly with MQSS components such as the QRM & CI and the Quantum Device Management Interface (QDMI), the library extends the existing support of MQSS – which already interfaces with Quantum Processing Units (QPUs) based on superconducting, ion-trap, and neutral-atom technologies, as well as emulation platforms – by enabling workloads also to leverage GPU acceleration. In doing so, it enhances MQSS's ability to efficiently parallelize and accelerate quantum circuit simulation across heterogeneous compute resources, paving the way for a unified heterogeneous compiler that seamlessly combines quantum and classical compilation paths within a single software framework. Through the integration of MLIR-based lowering into the IREE compiler and the Open Accelerated Linear Algebra (OpenXLA) [16] ecosystem within the MQSS, this work advances the broader vision of building an interoperable, and extensible software framework that connects end users, HPC infrastructures, and diverse quantum technologies.

The remainder of this paper is organized as follows. Section 2 provides the necessary technical background, introducing the key compiler technologies – LLVM, MLIR, and the `quake` dialect – and describing the architecture of the MQSS together with the roles of IREE and OpenXLA in heterogeneous compilation. Section 3 presents the design goals and architecture of the proposed Linear Algebra MLIR Interface for Neutral Acceleration (LAMINA) translation library, explaining its integration within the MQSS framework and illustrating its operation through an example workflow. Finally, Section 4 (current implementation) details the `quake2iree` prototype module that realizes the cross-dialect translation from quantum to standard MLIR dialects.

2 Technical Background

2.1 LLVM, MLIR, and `quake`: Foundational Technologies for Hybrid Compilation

The design of our translation library builds on modular compiler infrastructure that underpins modern heterogeneous compilation workflows. To situate our contribution within this landscape, we briefly introduce three key components: LLVM, MLIR, and the `quake` dialect.

The LLVM project [11] established the foundation for reusable, language-agnostic compiler technology. At its core lies a typed, language-independent Intermediate Representation (IR) that enables extensive compile-time and runtime optimizations across diverse targets. LLVM's modular architecture allows the coexistence of multiple front-ends and back-ends, while its

static single-assignment (SSA) form and rich type system facilitate precise program analysis and transformation. Over time, LLVM has become the *de facto* substrate for modern compiler ecosystems, serving as the foundational back-end for languages such as C/C++/Objective-C (via the Clang frontend), Rust, and Swift; moreover, its concept of IR as a common language for all middle-layers is so powerful that it has been extended to the Quantum Itermediate Representation (QIR) [19] to support QC. QIR is used in MQSS as an exchange format between MLIR-based quantum dialects such as `quake` or `catalyst` and hardware-specific software back-ends, since it provides a hardware-agnostic format for expressing quantum operations.

While LLVM excels at low-level optimizations, it lacks mechanisms for representing higher-level program structure or domain-specific semantics. The MLIR framework [12] extends LLVM by introducing a multi-level approach that supports multiple abstraction layers within one infrastructure. MLIR defines dialects – domain-specific IR extensions that capture specialized semantics while remaining interoperable. This enables progressive lowering, where high-level representations are incrementally transformed into lower-level forms (e.g., LLVM IR) while preserving source structure and metadata. MLIR's design thus allows compiler developers to compose transformations that span from algorithmic descriptions to device-specific instructions, drastically reducing engineering effort and improving maintainability across heterogeneous targets.

Within the MLIR ecosystem, the `quake` dialect – developed as part of NVIDIA's CUDA-Q framework [14] – extends MLIR with a unified representation for quantum-classical programs. Serving as the quantum kernel IR in CUDA-Q, `quake` bridges high-level constructs (from C++ and Python) with low-level execution through a dual semantic model: *memory semantics* for runtime-managed qubits and *value semantics* where qubits behave as first-class SSA values. This structure enables efficient analysis of data dependencies and supports transformations such as circuit optimization and hybrid control flow integration. As an MLIR dialect abstracting quantum operations and types, `quake` has been adopted by MQSS as one of its primary intermediate representations for quantum kernels in its current implementation, forming the basis for target-independent optimization and lowering.

2.2 The Munich Quantum Software Stack

The MQSS constitutes the core software infrastructure of the Munich Quantum Valley (MQV) initiative, aiming to bridge quantum and classical computing within a unified high-performance environment. Developed jointly by the Leibniz Supercomputing Centre (LRZ) and the Technical University of Munich (TUM), the MQSS provides a modular, extensible, and open-source framework that connects users, compilers, and hardware through a common software layer. Its architecture is designed to integrate a diverse range of quantum back-ends, including superconducting, ion-trap, and neutral-atom devices – alongside high-performance classical systems, enabling the integration of QC into HPC, which is referred to as HPCQC. This allows running hybrid quantum-classical workflows that are tightly coupled to existing supercomputing infrastructures such as LRZ's SuperMUC-NG cluster. The MQSS embodies the broader MQV vision of integrating quantum computers as disaggregated accelerators within HPC facilities, moving beyond the current paradigm of isolated, cloud-based quantum resources.

At its core, the MQSS comprises three tightly interconnected layers, as illustrated in Fig. 1. The front-end layer currently provides adapters for major quantum programming frameworks such as Qiskit, Pennylane, CUDA-Q, and its own C/C++-based QPI [10], while its modular design allows additional front-ends to be integrated in the future. These

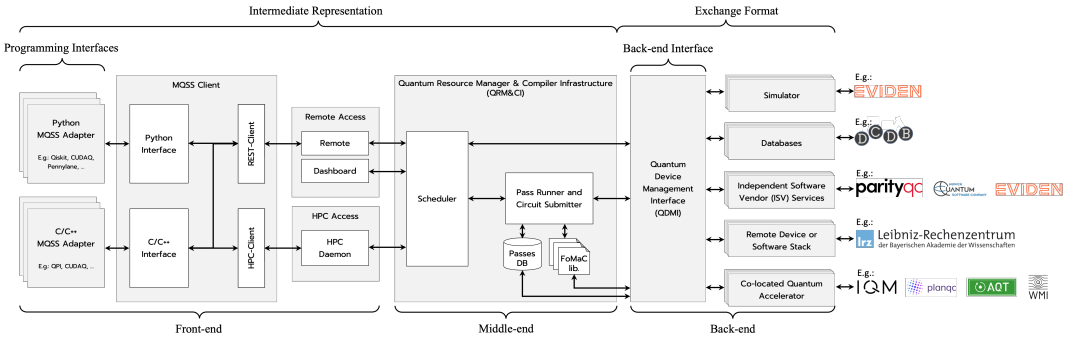

Figure 1 MQSS architecture overview: *MQSS Adapters* (e.g., Qiskit, CUDA-Q, PennyLane, and its native C-based Quantum Programming Interface (QPI)) submit gate- and pulse-based jobs to the *MQSS Client*, which handles automatic routing for both local HPC jobs and remote submissions. The QRM & CI encompasses MQSS's second-level scheduler, its Just In Time (JIT) LLVM-based compiler, and supporting libraries. QDMI exposes device capabilities, 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.

MQSS Adapters decouple programming models from the underlying runtime, allowing users to describe quantum circuits and hybrid workflows without depending on specific hardware or compilation paths. The middle-end layer – implemented through the QRM & CI – handles the quantum portion of submitted workloads, performing their compilation, optimization, and scheduling. It leverages the MLIR framework to represent quantum programs at multiple abstraction levels, thereby separating target-agnostic optimizations (such as circuit simplification, decomposition, and layout-independent rewriting) from target-specific transformations that adapt circuits to individual hardware back-ends. In its current form, the QRM & CI relies primarily on receiving from the Adapters the quantum circuits expressed in `quake` and `catalyst` [8] MLIR Dialects; QRM & CI lowers that representation to QIR or to device-oriented exchange format (such as OpenQASM). The back-end layer, implemented via the hardware-agnostic QDMI [18], is responsible for submitting the optimized circuits to the selected hardware executor and managing device interaction and feedback. The development of the present LAMINA library builds directly upon this foundation, extending the QRM & CI with new dialect conversion passes that connect the MQSS quantum compilation pipeline to heterogeneous classical back-ends through IREE and OpenXLA; see Sec. 2.3 for a description of both.

Finally, QDMI, as MQSS back-end layer provides a unified communication channel, with real hardware back-ends, made available as abstract devices, abstracting device-specific details through standardized queries, job, and session interface, and is complemented by the Figures-of-Merit and Constraints (FoMaC) libraries, which expose live device characteristics and support hardware-aware compilation.

This layered architecture enables MQSS to act as a flexible bridge between diverse programming frameworks and an expanding ecosystem of quantum technologies. However, hybrid workloads increasingly depend on heterogeneous compute resources. Large-scale quantum circuit simulations, variational hybrid quantum-classical algorithms, and Quantum Machine Learning (QML) applications rely heavily on GPU-accelerated classical computation. GPUs are particularly well-suited for state-vector evolution and tensor-network contraction,

where high-throughput linear algebra operations dominate runtime complexity. For example, simulators such as NVIDIA cuQuantum [1] and Google's TensorFlow Quantum [3] demonstrate that GPU-based acceleration can outperform CPU-only implementations by orders of magnitude for moderate circuit sizes, making them essential for algorithm benchmarking and noise modeling in near-term quantum research. Likewise, hybrid optimization loops – as used in algorithms such as the Variational Quantum Eigensolver (VQE) or the Quantum Approximate Optimization Algorithm (QAOA) – require tight coupling between classical optimizers running on HPC nodes or GPUs and quantum kernels executed on physical or emulated quantum devices. Furthermore, QC simulation remains a critical component for the quantum computing community at this stage of developmental maturity. These workloads highlight the growing need for a compilation and execution framework that can transparently target both quantum and classical accelerators while maintaining semantic consistency across architectures.

2.3 The IREE and OpenXLA Ecosystem

The IREE compiler and runtime framework, together with the OpenXLA ecosystem – which encompasses components such as `XLA` and the `StableHLO` dialect – are designed to enable efficient deployment of compute-intensive ML workloads across heterogeneous hardware. Both frameworks are built on the MLIR infrastructure and share a design philosophy centered on modularity, retargetability across heterogeneous hardware, and efficient execution. Initially developed for ML workloads, they provide an ideal foundation for compiling and executing quantum–classical hybrid applications, thanks to the fact that both ML models and quantum circuits can be expressed in terms of linear algebra.

IREE is an MLIR-native compiler and runtime environment that unifies compilation, optimization, and execution across diverse hardware targets. It supports both ahead-of-time (AOT) and JIT compilation, enabling flexible deployment on CPUs, GPUs, and dedicated accelerators such as Tensor Processing Unit (TPU) or emerging AI cores. At its core, IREE lowers MLIR modules through a sequence of dialect transformations and back-end-specific passes that progressively translate high-level operations into executable machine code for the selected target. IREE supports several input dialects, including `TOSA`, `MHLO`, and the standards `linalg`, `arith`, `math`, `tensor`, and `scf`, which form a rich foundation for representing numerical and control-flow operations common to both machine-learning and quantum workloads. Its modular runtime enables seamless integration with existing HPC infrastructures and supports heterogeneous CPU/GPU architectures – including NVIDIA and AMD – while providing low-latency execution via dynamic scheduling of overlapping compute and data-transfer workloads across devices.

Complementary to IREE, OpenXLA is an open ecosystem of compiler and runtime components designed for large-scale ML workloads, providing a bridge between MLIR-based front-ends and high-performance back-ends such as Accelerated Linear Algebra (XLA) and StableHLO. OpenXLA provides a rich ecosystem of optimizers for tensor and matrix operations, enabling sophisticated graph-level optimizations, automatic differentiation, and device-specific tuning. While initially developed for frameworks such as TensorFlow, PyTorch, and JAX, OpenXLA's modularity and MLIR-based design make it equally suitable for quantum pipelines. In particular, it enables seamless integration between parameterized quantum circuits and classical ML models, treating quantum kernels as differentiable computational primitives within machine-learning workflows. This is a key enabler for QML and the Variational Quantum Algorithm (VQA) [2], where optimization and training loops require tight coupling between quantum and classical computations.

3 Library Design and MLIR Translation

3.1 Design Goals

The LAMINA library is conceived as a modular bridge between quantum-oriented MLIR dialects and heterogeneous classical back-ends within the MQSS. Its principal goal is to enable the translation of CUDA-Q's `quake` dialect operations into MLIR representations compatible with IREE and OpenXLA. This allows quantum kernels to leverage the same optimization and deployment pipelines such as those provided by IREE and OpenXLA.

The design of LAMINA follows three main principles:

1. **Portability**: Support multiple compute architectures (Central Processing Unit (CPU), GPU, Tensor Processing Unit (TPU), or other accelerators) while remaining agnostic to vendor-specific details. The library should allow the same quantum kernel to be compiled and executed efficiently across heterogeneous HPC environments.
2. **Modularity and Extensibility**: Use MLIR's dialect-based design to implement a clean separation between front-end quantum dialects, intermediate transformation passes, and back-end lowering stages. This enables incremental extension to dialects such as `XQSS Pulse` [5], and supports external optimization passes developed by the community.
3. **Interoperability and Integration**: Ensure tight coupling with existing MQSS components – particularly the QRM & CI and QDMI – so that hardware-aware compilation and runtime control remain consistent across quantum and classical targets.

Through these design principles, LAMINA aims to expand the MQSS compilation pipeline to support targeting both quantum hardware and high-performance classical accelerators under a unified MLIR-based abstraction.

3.2 Architecture of the Translation Library

At a high level, LAMINA operates as a translation and optimization layer that consumes MLIR modules expressed in the `quake` dialect, produced by the MQSS CUDA-Q Adapter, the extended MQSS's QPI, or by translation passes applied to circuits expressed in the Catalyst MLIR dialect. This `quake` representation is then progressively lowered to MLIR dialects compatible with IREE or OpenXLA back-ends. The quantum compiler in QRM & CI applies translation passes as the final lowering before submission via QDMI; this allows all optimization and transpilation passes to be applied beforehand, which is helpful in transpiling the circuit to a specific gate set or for benchmarking use cases. The translation process consists of two stages:

1. **Cross-Dialect Mapping**: LAMINA performs a dialect-to-dialect transformation that maps `quake` operations to a composition of MLIR dialects. When targeting IREE, the set of MLIR dialects is composed of the standards `arith`, `math`, `tensor`, `scf`, and `linalg`. When targeting OpenXLA, `quake` is translated into first `scf`, `arith`, `memref` and `linalg`, the latter then lowered to the Stable High-Level Optimizer (`stablehlo`) Dialect [17]. Both mappings express quantum gates and measurement operations as tensor or linear-algebra primitives suitable for execution on classical hardware.
2. **Back-end Lowering and Code Generation**: then, LAMINA invokes IREE's or OpenXLA's compilation pipelines to lower the translated MLIR module to target-specific code. Depending on the execution context, this may result in GPU kernels or vectorized CPU instructions. The binaries are generated by the dedicated LAMINA MQSS component (See 3.3), which also handles submission via the HPC cluster scheduler.

This architecture ensures that the translation path from quantum kernels to classical accelerators remains transparent, verifiable, and reusable, enabling reproducible hybrid workflows where identical quantum programs can be benchmarked or simulated across different hardware back-ends.

3.3 Integration with MQSS Components

LAMINA integrates seamlessly into the existing MQSS infrustructure as three separate components, as a set of MLIR translation passes, by implementing the QDMI specifincation into the so called LAMINA QDMI Device and as a Figures-of-Merit and Constraints (FoMaC) library. As translation passes, LAMINA is available as a pass the MLIR dialect agnostic pass runner within QRM & CI that can apply to a circuit, alternatively to lower `quake` to QIR or to the device-native exchange format (see Fig 2).

During compilation QRM & CI communicates with the LAMINA FoMaC library to know whether or not there is a GPU with enough memory to fit the number of qubits in the circuit, and the estimated time to access it; the first information is dispatched to the LAMINA QDMI Device, the second back to the MQSS Client.

The LAMINA QDMI Device receives the translated circuit, the target GPU architecture, and cluster partition to which it belongs (both provided by the LAMINA FoMaC). It subsequently compiles to the target architecture using IREE or OpenXLA, then submits the resulting binary for execution via the available HPC resource scheduler. The LAMINA QDMI Device is also responsible for routing the results back to the MQSS Client that requested execution.

Via MLIR passes, FoMaC library and the LAMINA QDMI Device, LAMINA enables QDMI to serve as a unified interface for communicating with both quantum and classical execution resources. In the traditional MQSS workflow, quantum kernels compiled through the standard pipelines are dispatched to physical quantum devices via QDMI. LAMINA enriches this pipeline by interfacing to it classical compute resources in the same way as quantum resources. The powerful abstraction layer provided by QDMI handles both real and simulated execution targets through a consistent API. Users and system components can therefore submit, monitor, and retrieve results from quantum processors and classical accelerators uniformly, without requiring separate interfaces or modification to their source code. From an operational perspective, this allows classical HPC resources to serve as drop-in replacements for real devices, enabling reproducible benchmarking, regression testing, and performance studies under controlled simulation conditions.

LAMINA integration with MQSS is hence made in a way to reinforce MQSS philosophy of hardware-agnostic design, ensuring that all computational resources – quantum or classical can be orchestrated coherently within the same ecosystem.

3.4 Example Workflow: Pure Quantum Algorithm

Let us illustrate the ideas above with a simple example: submitting a pure quantum algorithm to a single GPU without the need to sync with classical computation, which we would call a hybrid algorithm. In the following, we assume the access to classical resources is managed by Slurm [9], a resource scheduler widely used in HPC centers. In our simple example, the workflow proceeds as follows.

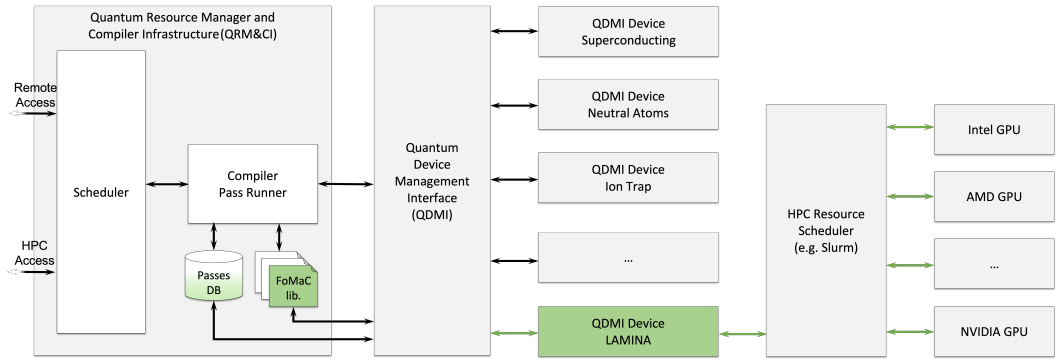

Figure 2 LAMINA integration within MQSS. The circuit reaches the Scheduler in QRM & CI via one submission channel. The dedicated translation passes in the Pass DB are applied by the Compiler Pass Runner to translate from `quake` to the input dialects for the LAMINA QDMI Device; the dedicated FoMaC gets information on the better GPU for the circuit at hand. Translated circuit and GPU details are dispatched to the LAMINA QDMI Device, to perform compilation and submission to the HPC resource scheduler.

1. **Front-end to `Quake`**: the user submits a quantum algorithm (quantum job), composed of a single quantum circuit, via one of the MQSS Adapters; as part of its responsibility, the MQSS Adapter lowers the circuit to `quake` representation and passes it to the QRM & CI component.
2. **Optimization and MLIR Dialect Translation**: in QRM & CI the quantum compiler optimizes the circuit. It applies all the passes required by the use-case, and as the latest MLIR-pass, it applies the translation from `quake` to the set of standard MLIR dialects compatible with IREE. During this step, thanks to the LAMINA FoMaC library which exploits the HPC cluster scheduler functionalities (e.g., the Slurm REST API), the target GPU and how to reach it via the HPC resource scheduler is identified.
3. **Loading the LAMINA QDMI Device and dispatching**: at this stage QRM & CI loads LAMINA QDMI Device related library and dispatches the MLIR representation of the circuit to it for further compilation to machine code.
4. **Target Architecture Identification and Compilation**: on receipt of MLIR circuit representation and target GPU architecture, the LAMINA QDMI Device executes the IREE compilation pipeline.
5. **Job Submission**: once the binary file has been prepared by IREE, it is ready for execution; the LAMINA QDMI Device submits its execution to the HPC cluster scheduler, specifying the resource selected for compilation.
6. **Returning results**: much like all the QPUs dedicated QDMI devices, the LAMINA QDMI Device waits for the job to complete; once that happens, it reports the results back to the QDMI Client which submitted the job (QRM & CI in this example), or, in case of errors, the related error messages.

4 Current Implementation: Cross-Dialect Mapping and the `Quake2IREE` Module

At the current stage of development, the implementation of the **cross-dialect mapping layer** within LAMINA is being prototyped through the standalone module `Quake2IREE`. This module provides a minimal but functional infrastructure to transform quantum programs expressed in the `quake` dialect – originating from the MQSS Adapters – into standard MLIR dialects that can participate in IREE's compilation and optimization pipeline.

The `Quake2IREE` project builds directly on the MLIR/LLVM framework and defines a complete conversion pass (`QuakeToStandard`) that lowers quantum-specific constructs to conventional MLIR representations. The pass is implemented as an MLIR *Conversion Pattern* combined with a dedicated *Type Converter* and is designed to integrate seamlessly into the LAMINA pipeline as its first transformation stage. Its purpose is to decouple quantum-specific abstractions from their physical semantics and re-express them in terms of tensor-based operations compatible with IREE's back-ends.

The core of the implementation resides in three modules that correspond to the standard structure of MLIR-based compiler projects:

- **Dialect Module:** Defines the intermediate representations used in the pipeline. This includes the `quake` dialect for quantum computation and the `Classical Compute` (`CC`) dialect for classical control and data structures, defined by NVIDIA as part of the CUDA-Q project. The module also provides shared utilities for type handling, attribute registration, and cross-dialect interoperability.
- **Conversion Module:** Implements the `QuakeToStandard` conversion pass, which performs type conversion and operation rewriting. The `QuakeToStandardTypeConverter` maps quantum-specific types such as `!quake.ref` (the type for a single qubit reference) and `!quake.veq` (the type for a vector of qubits) to tensor-based representations (e.g., `tensor<i1>` for the first and `tensor<?xi1>` for the second, using the MLIR `Tensor` Dialect). Conversion patterns such as `ConvertAlloca`, `ConvertVeqSize`, and `ConvertInitState` translate allocation (`quake.alloca`), size query (`quake.veq_size`), and initialization (`quake.init_state`) operations into their classical analogues.
- **Tool Module:** Provides the command-line utility `quake-to-standard-opt`, which serves as the standalone driver for the conversion pass and allows developers to test the transformation pipeline on individual MLIR modules. This component will later be incorporated into the LAMINA toolchain as the entry point for classical–quantum lowering workflows.

The transformation realized by the `Quake2IREE` module enables quantum programs to be represented entirely in terms of standard MLIR dialects (`arith`, `tensor`, `complex`, `func`, `scf`) and the `CC` dialect. By doing so, the resulting IR becomes compatible with existing MLIR optimization passes, facilitating the use of IREE's lowering infrastructure for heterogeneous targets. Conceptually, this step transforms quantum IR into a form that can be reasoned about using the same optimization strategies applied to classical ML workloads, paving the way for unified compilation across quantum and classical domains.

To ensure the correctness and robustness of the current implementation, a lightweight Python-based test suite has been developed. The script collects per-test results and reports aggregated statistics, automatically failing the CI pipeline if any individual test fails. This testing framework provides an early validation layer for new conversion patterns and will later be extended with semantic checks comparing `quake` and lowered IR behaviors. It is integrated into the project's build system and serves as the foundation for continuous integration testing as additional dialect mappings are added.

Although `Quake2IREE` currently operates as an independent prototype, it represents the foundational stage of LAMINA's cross-dialect translation layer. The next development milestone will integrate this module directly into the MQSS compilation pipeline via the QRM & CI, enabling end-to-end workflows from CUDA-Q front ends to IREE-compatible MLIR modules.

5 Conclusions and Outlook

This work presented the design and early implementation of LAMINA, a new MLIR-based translation library that extends the MQSS toward heterogeneous quantum-classical compilation. By introducing a translation path from CUDA-Q (`quake`) dialects to MLIR representations compatible with IREE and OpenXLA, LAMINA enables hybrid workloads to target both quantum and classical accelerators through a unified compiler infrastructure. The approach leverages MLIR's modular dialect architecture to bridge quantum abstractions with mature machine-learning compiler ecosystems, fostering cross-domain interoperability between quantum computing, classical HPC, and AI-oriented hardware.

The integration of LAMINA within MQSS represents an essential step toward realizing the broader vision of the MQV initiative: building a software ecosystem in which quantum devices function as integral components of HPC infrastructures. Functionally, LAMINA extends the existing QRM & CI of MQSS, introducing cross-dialect translation capabilities that connect its MLIR-based compilation pipeline with classical back-ends and ML-oriented runtimes. Through its coupling with the QRM & CI and QDMI, LAMINA provides a coherent runtime model where real and virtual devices can coexist, enabling developers to benchmark, simulate, and deploy quantum workloads on heterogeneous resources such as CPUs and GPUs without changing workflow semantics. The resulting interoperability creates opportunities for accelerating quantum circuit simulation, hybrid algorithm optimization, and QML applications on classical accelerators, while maintaining full compatibility with MQSS's existing runtime ecosystem.

From a developer's perspective, current quantum software workflows are often fragmented and tightly coupled to specific execution back-ends. In practice, developers frequently rely on different simulators for CPU and GPU execution, use back-end-specific intermediate representations, or maintain separate code paths for simulation and hardware execution. This fragmentation complicates rapid prototyping and benchmarking, as developers often need to adapt or rewrite quantum circuits when moving from local CPU-based simulation to GPU-accelerated simulation or to execution on physical QPUs. The approach presented in this work directly addresses these challenges by providing a unified compilation path for quantum circuits that is agnostic to the underlying classical execution backend. By lowering quantum kernels to a common MLIR-based representation and leveraging on ML compilers, developers can target CPU- or GPU-based simulation through the same interface and reuse the same workflow within the broader MQSS stack.

At the current stage, the `quake2iree` prototype demonstrates the feasibility of translating quantum-specific constructs into standard MLIR dialects, allowing their integration into IREE's and OpenXLA's compilation pipelines. Future work will focus on several key directions. First, the ongoing integration of `quake2iree` into the MQSS compilation pipeline will enable automated end-to-end workflows from front-end quantum languages to executable IREE modules. Second, LAMINA will be extended with hardware-aware optimization passes that exploit information from FoMaC and QDMI to perform dynamic scheduling and device-specific tuning. Third, we will extend IREE to support a hardware back-end for which it can compile, including the Intel Data Center GPU Max 1550 (formerly known as the Intel Ponte Vecchio GPU), thereby furthering its vendor-agnostic characteristic. Benchmarks will accompany each of the directions listed above to assess how well the MQSS with LAMINA performs in comparison with other frameworks accelerating quantum simulation on GPUs, such as NVIDIA's CUDA-Q. We will publish the results of these benchmarks in follow-up

papers. Finally, as part of MQV's broader roadmap, LAMINA will serve as a foundation for exploring unified compiler techniques that couple quantum kernels with AI-oriented computation graphs, contributing to the convergence of quantum, HPC, and ML ecosystems.

References

1 Harun Bayraktar, Ali Charara, David Clark, Saul Cohen, Timothy Costa, Yao-Lung L. Fang, Yang Gao, Jack Guan, John Gunnels, Azzam Haidar, Andreas Hehn, Markus Hohnerbach, Matthew Jones, Tom Lubowe, Dmitry Lyakh, Shinya Morino, Paul Springer, Sam Stanwyck, Igor Terentyev, Satya Varadhan, Jonathan Wong, and Takuma Yamaguchi. cuquantum sdk: A high-performance library for accelerating quantum science. In *2023 IEEE International Conference on Quantum Computing and Engineering (QCE)*, volume 01, pages 1050–1061, 2023. doi:10.1109/QCE57702.2023.00119.

2 Kishor Bharti, Alba Cervera-Lierta, Thi Ha Kyaw, Tobias Haug, Sumner Alperin-Lea, Abhinav Anand, Matthias Degroote, Hermanni Heimonen, Jakob S. Kottmann, Tim Menke, Wai-Keong Mok, Sukin Sim, Leong-Chuan Kwek, and Alán Aspuru-Guzik. Noisy intermediate-scale quantum algorithms. *Rev. Mod. Phys.*, 94:015004, February 2022. doi:10.1103/RevModPhys.94.015004.

3 Michael Broughton, Guillaume Verdon, Trevor McCourt, Antonio J. Martinez, Jae Hyeon Yoo, Sergei V. Isakov, Philip Massey, Ramin Halavati, Murphy Yuezhen Niu, Alexander Zlokapa, Evan Peters, Owen Lockwood, Andrea Skolik, Sofiene Jerbi, Vedran Dunjko, Martin Leib, Michael Streif, David Von Dollen, Hongxiang Chen, Shuxiang Cao, Roeland Wiersema, Hsin-Yuan Huang, Jarrod R. McClean, Ryan Babbush, Sergio Boixo, Dave Bacon, Alan K. Ho, Hartmut Neven, and Masoud Mohseni. Tensorflow quantum: A software framework for quantum machine learning. *arXiv preprint arXiv:2003.02989*, 2020. last revised 26 Aug 2021 (v2). arXiv:2003.02989.

4 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. The xqss: Connecting end users, integrating diverse quantum technologies, accelerating hpc, 2025. arXiv:2509.02674.

5 Jorge Echavarria, Muhammad Nufail Farooqi, Amit Devra, Santana Lujan, Léo Van Damme, Hossam Ahmed, Martín Letras, Ercüment Kaya, Adrian Vetter, Max Werninghaus, Martin Knudsen, Felix Rohde, Albert Frisch, Eric Mansfield, Rakhim Davletkaliyev, Vladimir Kukushkin, Noora Färkkilä, Janne Mäntylä, Nikolas Pomplun, Andreas Spörl, Lukas Burgholzer, Yannick Stade, Robert Wille, Laura B. Schulz, and Martin Schulz. Tackling the challenges of adding pulse-level support to a heterogeneous hpcqc software stack: Xqss pulse. *arXiv e-prints*, page arXiv:2510.26565, October 2025. doi:10.48550/arXiv.2510.26565.

6 J. Eisert, M. Cramer, and M. B. Plenio. Colloquium: Area laws for the entanglement entropy. *Rev. Mod. Phys.*, 82:277–306, February 2010. doi:10.1103/RevModPhys.82.277.

7 Mario Hernandez Vera and Marco De Pascale. QCT-UEA-management/quake2iree. Software, swhId: swh:1:snp:c7b8f26bbeb0273ac7e029a8bba5845db219bdec (visited on 2026-03-17). URL: https://github.com/QCT-UEA-management/quake2iree.git, doi:10.4230/artifacts.25678.

8 David Ittah, Ali Asadi, Erick Ochoa Lopez, Sergei Mironov, Samuel Banning, Romain Moyard, Mai Jacob Peng, and Josh Izaac. Catalyst: a python jit compiler for auto-differentiable hybrid quantum programs. *Journal of Open Source Software*, 9(99):6720, 2024. doi:10.21105/joss.06720.

9 Morris A. Jette and Tim Wickberg. Architecture of the slurm workload manager. In Dalibor Klusáček, Julita Corbalán, and Gonzalo P. Rodrigo, editors, *Job Scheduling Strategies for Parallel Processing*, pages 3–23, Cham, 2023. Springer Nature Switzerland. doi:10.1007/978-3-031-43943-8_1.

10 Ercüment Kaya, Burak Mete, Laura Schulz, Muhammad Nufail Farooqi, Jorge Echavarria, and Martin Schulz. QPI: A programming interface for quantum computers. In *2024 IEEE International Conference on Quantum Computing and Engineering (QCE)*, volume 02, 2024. doi:10.1109/QCE60285.2024.10293.

11 C. Lattner and V. Adve. Llvm: a compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, pages 75–86, 2004. doi:10.1109/CGO.2004.1281665.

12 Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. Mlir: Scaling compiler infrastructure for domain specific computation. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 2–14, 2021. doi:10.1109/CGO51591.2021.9370308.

13 Gabin Schieffer, Stefano Markidis, and Ivy Peng. Harnessing CUDA-Q's MPS for Tensor Network Simulations of Large-Scale Quantum Circuits . In *2025 33rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP)*, pages 94–103, Los Alamitos, CA, USA, March 2025. IEEE Computer Society. doi:10.1109/PDP66500.2025.00022.

14 The CUDA-Q development team. CUDA-Q. URL: https://github.com/NVIDIA/cuda-quantum.

15 The IREE Authors. IREE, September 2019. URL: https://github.com/iree-org/iree.

16 The OpenXLA Community. The OpenXLA Ecosystem. URL: https://openxla.org.

17 The StableHLO Community. The StableHLO Operation Set. URL: https://openxla.org/stablehlo.

18 Robert Wille, Ludwig Schmid, Yannick Stade, Jorge Echavarria, Martin Schulz, Laura Schulz, and Lukas Burgholzer. Qdmi – quantum device management interface: A standardized interface for quantum computing platforms. In *IEEE International Conference on Quantum Computing and Engineering (QCE)*, 2024.

19 Yannick Stade and Lukas Burgholzer and Robert Wille. Towards Supporting QIR: Steps for Adopting the Quantum Intermediate Representation, 2025.