Bridging Paradigms: Designing for HPC-Quantum Convergence

Amir Shehata[a,*], Peter Groszkowski[a], Thomas Naughton[b], Muralikrishnan Gopalakrishnan Meena[a], Elaine Wong[b], Daniel Claudino[c], Rafael Ferreira da Silva[a], Thomas Beck[a]

[a] *National Center for Computational Sciences, Oak Ridge National Laboratory, Oak Ridge, TN, USA*
[b] *Computer Science and Mathematics, Oak Ridge National Laboratory, Oak Ridge, TN, USA*
[c] *Computational Sciences and Engineering, Oak Ridge National Laboratory, Oak Ridge, TN, USA*

Abstract

This paper presents a comprehensive software stack architecture for integrating quantum computing (QC) capabilities with High-Performance Computing (HPC) environments. While quantum computers show promise as specialized accelerators for scientific computing, their effective integration with classical HPC systems presents significant technical challenges. We propose a hardware-agnostic software framework that supports both current noisy intermediate-scale quantum devices and future fault-tolerant quantum computers, while maintaining compatibility with existing HPC workflows. The architecture includes a quantum gateway interface, standardized APIs for resource management, and robust scheduling mechanisms to handle both simultaneous and interleaved quantum-classical workloads. Key innovations include: (1) a unified resource management system that efficiently coordinates quantum and classical resources, (2) a flexible quantum programming interface that abstracts hardware-specific details, (3) A Quantum Platform Manager API that simplifies the integration of various quantum hardware systems, and (4) a comprehensive tool chain for quantum circuit optimization and execution. We demonstrate our architecture through implementation of quantum-classical algorithms, including the variational quantum linear solver, showcasing the framework's ability to handle complex hybrid workflows while maximizing resource utilization. This work provides a foundational blueprint for integrating QC capabilities into existing HPC infrastructures, addressing critical challenges in resource management, job scheduling, and efficient data movement between classical and quantum resources.

Keywords: Quantum Computing, High-Performance Computing, System Integration, Quantum Applications.

Sections

1. Introduction

Quantum Computing (QC) holds immense promise for accelerating scientific discovery across multiple domains, including quantum chemistry, materials science, optimization, and artificial intelligence (AI). However, QC is not expected to replace classical computing but rather serve as an accelerator for tasks best suited to its capabilities, necessitating their integration with their classical counterparts, particularly High Performance Computing (HPC). Similar to how GPUs enhance classical computing through heterogeneous integration, quantum processing units (QPUs) could accelerate specific quantum advantaged algorithms while leveraging classical systems for other computations within larger scientific workflows.

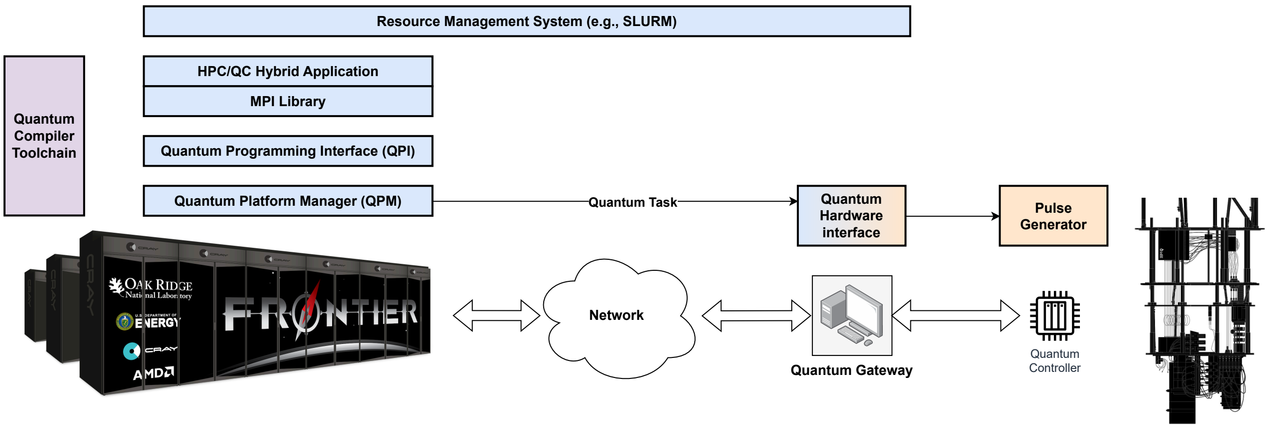
Currently, several major research centers worldwide are actively pursuing QC/HPC integration initiatives. The European Union's EuroHPC project has designated multiple centers for quantum computer installation, including facilities in Czechia, Finland, France, Germany, Italy, Poland, and Spain. Similarly, significant efforts are underway at various national research laboratories in the United States. These initiatives primarily focus on developing optimized software stacks and workflows for various quantum technologies, preparing for the eventual integration of fault-tolerant quantum computers (FTQC).

The Oak Ridge Leadership Computing Facility (OLCF) at Oak Ridge National Laboratory (ORNL) has established considerable expertise in deploying world-leading supercomputers that utilize accelerator technologies, as demonstrated by systems like Frontier and Summit. Through the Quantum Computing User Program (QCUP), ORNL has gained valuable experience in providing access to diverse quantum resources and managing hybrid quantum-classical workflows. In our previous work [5], we outlined a comprehensive framework for integrating QC capabilities into HPC environments and detailed the infrastructure requirements for supporting quantum-classical workflows. This experience, combined with ORNL's established processes for supercomputer lifecycle management, provides a

*Corresponding address: National Center for Computational Sciences, Oak Ridge National Laboratory, Oak Ridge, TN, 37831, USA

**This manuscript has been authored by UT-Battelle, LLC, under contract DE-AC05-00OR22725 with the US Department of Energy (DOE). The publisher acknowledges the US government license to provide public access under the DOE Public Access Plan (`http://energy.gov/downloads/doe-public-access-plan`).

Email addresses: `shehataa@ornl.gov` (Amir Shehata), `groszkowskip@ornl.gov` (Peter Groszkowski), `naughtont@ornl.gov` (Thomas Naughton), `gopalakrishm@ornl.gov` (Muralikrishnan Gopalakrishnan Meena), `wongey@ornl.gov` (Elaine Wong), `claudinodc@ornl.gov` (Daniel Claudino), `silvarf@ornl.gov` (Rafael Ferreira da Silva), `becktl@ornl.gov` (Thomas Beck)



Resource Management System (e.g., SLURM)
Quantum Compiler Toolchain
HPC/QC Hybrid Application
MPI Library
Quantum Programming Interface (QPI)
Quantum Platform Manager (QPM)
Quantum Task
Quantum Hardware interface
Pulse Generator
Network
Quantum Gateway
Quantum Controller
FRONTIER

Figure 1: A high-level physical view of QC/HPC integration, overlaid with a simplified representation of the quantum software stack. The quantum gateway can interface with multiple quantum hardware systems of the same type. Additionally, multiple gateways, including virtualized ones, can be configured to communicate with different types of hardware. This separation is designed to minimize workload on the gateway, reducing congestion and preventing over-subscription.

strong foundation for developing comprehensive QC/HPC integration strategies.

Building upon our previous integration framework [5], this paper advances the state of the art by presenting a detailed software stack design and implementation strategies for quantum-HPC integration. While our previous work established the foundational requirements, this paper focuses on the practical software architecture and design aspects, as well as validation of the software stack in real-world scenarios. Our key contributions include:

1. A hardware-agnostic software stack design that supports both current noisy intermediate-scale quantum-era (NISQ) devices and future fault-tolerant quantum computers, while maintaining compatibility with existing HPC workflows and resource management systems.
2. A comprehensive software stack architecture that addresses critical challenges in quantum-HPC integration, including resource scheduling, job management, and efficient data movement between classical and quantum resources.
3. An analysis of the proposed software architecture, including its interaction with scientific applications and its integration with external software components such as workflow management systems.
4. Implementation strategies for supporting both on-premises quantum hardware and cloud-based quantum resources, enabling flexible deployment options for HPC centers.

2. Framework Overview

A high-level physical view of the integration of an on-premises quantum machine is illustrated in Figure 1. In the figure, the quantum machine is connected to a quantum controller. The quantum controller acts as the interface between classical control hardware and quantum processors (qubits). The main function of the controller is to generate, manipulate, and read out signals that control the quantum computation.

A classical compute node is directly connected to the quantum controller and acts as a "quantum gateway" for access to the quantum resource. The quantum gateway computer runs classical services, which include resource management software to efficiently allocate/reserve the quantum hardware. The gateway computer also provides dedicated, low-latency computational resources for any classical operations that are needed by the quantum controller to satisfy timing constraints. Lastly, the gateway can host portions of the resource management plugins that are hardware-aware and used to interface resources elsewhere on the network (e.g., *Frontier* HPC supercomputer).

The bulk of the QC/HPC software stack resides on traditional classical HPC resources, e.g., service nodes, compute nodes, etc. The resource manager (e.g., SLURM [21]) runs on the HPC service nodes and the compute node where application processes are launched. The hybrid application spans the HPC and QC resources. Figure 1 overlays a notional mapping of where the different services and capabilities reside. The Quantum Programming Interface (QPI) is the application facing layer that is used by the hybrid application on the HPC compute nodes to interact with the quantum resources. The Quantum Platform Manager (QPM) is the hardware facing layer that provides access to underlying quantum resources from the HPC compute nodes. These software layers are described in more detail in Section 7.

3. Traditional HPC Accelerators: GPU Example

For the foreseeable future, QC will likely be regarded as a specialized computational accelerator for tasks particularly suited to this paradigm. Therefore, it is instructive to analyze the integration of GPUs in classical computing to draw parallels and lessons from GPU/CPU integration. ORNL's Frontier [28] supercomputer utilizes AMD GPUs and has further emphasized the need for specialized accelerators in computer architecture.

3.1. GPU Control Interfaces

Developing efficient GPU programs requires mastering two distinct phases: (1) Developing and compiling GPU-enabled code, and (2) Running the resulting binary on the target system. Typically, GPU kernels are annotated with specific syntax, such as `__global__`, to signify that the function is intended for execution on the GPU. GPU APIs, like CUDA [19, 23] or HIP [18], provide a method to interact with the GPU and can be categorized into several groups. For example, Initialization and Version Management APIs manage the setup and version checking of the GPU environment. Device Management APIs handle device selection, querying device attributes, and setting various device configuration. Execution Control APIs govern how kernels are launched and executed on the GPU. Memory Management APIs handle GPU memory management, including allocation, deallocation, and copying. These APIs are employed within host code, running on the CPU, to set up the essential components needed for GPU kernel execution. For example, device management APIs enable the selection of a specific GPU for executing GPU code, while execution control APIs facilitate launching GPU kernels on the chosen GPU. These APIs are typically encapsulated within user-space GPU libraries, which interact with the GPU kernel driver to perform their functions.

3.2. GPU Compilation

A specialized compiler is utilized to process GPU-enabled source code. It first pre-processes directives, macros, and includes. It also splits the code into parts that will run on the device (i.e., the GPU) and those that will execute on the host (i.e., the CPU). Host code is compiled using standard compilers such as gcc or clang. GPU code is lowered into an intermediate representation suitable for further compilation. HIP for example uses LLVM IR. During this process, any optimization passes are carried out on the compiled code. If the GPU target architecture is known, the code is lowered down to a format suitable for execution on the GPU. Otherwise, the intermediate representation is stored in the combined host and GPU binary.

3.3. GPU Scheduling and Execution

When the program is executed, the operating system loads the binary into memory and begins executing the host code. Any invocation of the GPU API results in the execution of the GPU library code, which interacts with the GPU device driver to perform operations such as GPU memory allocation, context creation, and device management. When the host code launches a GPU kernel, the GPU API looks up the function symbol in the kernel symbol table within the ELF binary. If the GPU function has already been compiled to the target GPU architecture, it is queued on the GPU scheduler. If it is still in an intermediate representation (IR) format the GPU API compiles it down to the specific machine code for the GPU architecture present in the system. This Just-In-Time (JIT) compilation is optimized for the current hardware and the compiled code is often cached to avoid recompilation in future executions, significantly improving performance.

Once compiled, the kernel is queued on the GPU scheduler, which manages the execution order of multiple kernels to efficiently utilize the GPU resources. On program completion, the GPU library performs finalization, ensuring that all GPU operations are completed, memory is deallocated, and other resources are cleaned up gracefully. This includes releasing any device contexts, freeing allocated memory, and ensuring that the GPU is in a clean state for future operations.

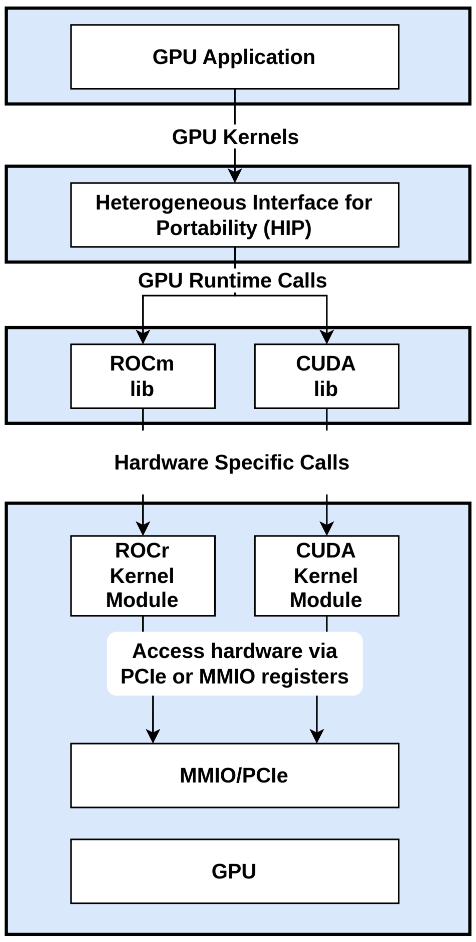


Figure 2: GPU software stack showing a number of abstraction layers that separate the application interface from low level tools that may be hardware-specific. A similar general approach will be employed with working with quantum hardware based accelerators.

3.4. Discussion

The GPU software stack described in this section serves as a good template for the QC software stack, refer to Figure 7. Similar to the control interfaces shown in Figure 2, QC will require a comparable set of interface APIs that provide the appropriate level of abstraction for applications. The GPU compilation process shares many parallels with quantum circuit compilation, where circuits are lowered to an intermediate representation and further optimized into a hardware-specific format, following the JIT compilation process. Finally, circuits will need to be scheduled and executed on the target platform raising similar challenges to host/GPU allocation and coordination. These points will be elaborated on further in Sections 5, 6, and 7.

4. Application Patterns

Although hybrid classical-quantum algorithms inherently rely on both types of hardware, it is useful to classify application patterns based on their varying computational demands. Such distinction is important, given the significant asymmetry in the availability of quantum computers in the foreseeable future. In both current, and even next generation of QC/HPC ecosystems, quantum hardware deployment will be limited (i.e., only a single or few quantum devices will likely be available), in contrast to classical compute infrastructure, which will continue to be the dominant computational resource. Below, we distinguish between different application patterns based on their need to utilize quantum and classical hardware. These approaches span from those requiring minimal QC resources to implementations where quantum computation dominates the total resource allocation[1]. While it is expected that all these application types are treated as first-class citizens by the underlying software tools (e.g., ORNL's QFw), explicitly identifying the core differences between them, can help optimize resource allocation and improve overall runtime efficiency in hybrid computing environments.

1. *High quantum, low classical resource usage:* This category includes applications where the quantum resource requirements dominate the computational workload. Here the classical resources are primarily used for (usually fast, and requiring a minimal HPC node count to execute) pre- and post-processing tasks that may include circuit generation or transpilation, and classical analysis of measurement results.

2. *Low quantum, high classical resource usage:* In this category, the core of computation is classical in nature and only a small amount of quantum resources are required. This pattern mainly expands on applications currently encountered in HPC ecosystems, where many HPC nodes are utilized over long times, but where quantum resources may be invoked very selectively, only for specific parts of the total computational task.

3. *Roughly equal quantum and classical resource usage:* Here, quantum and classical computations are more balanced, each playing a substantial role in the total computational task, and neither dominating the total resource requirements.

Having a framework (and crucially a scheduler) that is able to understand these different application patterns, can help minimize resource idle time, and thus maximize overall hybrid system performance of the full QC/HPC system. For example, in the case of applications that adhere to Pattern 2 above, in order to optimally utilize the (highly scarce) quantum hardware, the scheduler may need to interleave quantum execution of multiple hybrid jobs, whereas in the case of jobs that mainly adhere to Pattern 1, the scheduling becomes more straightforward, as jobs can be largely scheduled sequentially. See Section 5 for further discussion.

4.1. Time-Sensitive Classical-Quantum Computation

Some hybrid applications may also have a hard constraint on the latency between interleaved classical and quantum parts of the computation. One example of such a scenario includes quantum circuits with mid-circuit measurements (see, e.g., [15]), where parts of the algorithm can be conditionally executed depending on measurement results of a qubit (or many qubits) *during* (i.e., mid-way through) the quantum evolution. In the simplest variant of above paradigm, the classical processing involved can be fairly minimal, and in practice can be often done on the close-to-quantum hardware electronics (e.g., on a FPGA- or ASIC-based devices, with minimal communication latency to the quantum hardware). One can, however, also imagine more involved scenarios, where such mid-circuit measurements spawn complicated, even HPC-based computations, which in turn determine what further quantum operations should be executed. This latter operational pattern is currently not practical, due to the very limited coherence times[2] of the available, even state-of-the-art, quantum computers. As quantum hardware evolves, however, and some level of fault-tolerance is achieved, the timescales on which quantum devices will stay coherent will be extended dramatically (to seconds, minutes, or beyond), thus reducing the hard-time constraints on how long hybrid computations (and related communications) that require time-sensitive processing, can take. It is worth pointing out that the inability of a hybrid QC/HPC systems to keep pace, will generally result in a reduction of the fidelity of obtained results (formally, as coherence is gradually lost, quantum states become more mixed, thus introducing classical uncertainty in the qubit measurements, see e.g., [16]).

Another important example where time-sensitive classical-quantum interactions are required is syndrome-decoding (a mechanism by which one can understand how errors can be corrected in a quantum computer, when quantum error correcting codes are being utilized). This process usually involves continually obtaining selective measurements, which then need to be classically processed (i.e., decoded) before future quantum operations are applied, see e.g., [29] – each iteration of this protocol, must happen in a well defined time (that depend on various implementation details such as the hardware being used, or the specific error correction code being utilized). Although researchers previously believed that the decoding process would require substantial classical resources, recent studies [17, 30] suggest that dedicated FPGA/ASIC hardware may suffice, and HPC-level ecosystems will likely not be directly utilized for

[1] Although often the classical HPC compute requirements are expressed in terms of a measure such as *node-hours*, and equivalent can be used for quantum devices, due to the generally very limited availability of the latter, it may initially be more useful to discuss the (quantum especially) requirements in terms of a fraction-of-the-total that are available.

[2] In simple terms, a "coherence time" of a quantum system defines a timescale beyond which the environmental noise can destroy a quantum state, thus making a quantum computation no longer quantum.

syndrome decoding. Nevertheless, given the algorithmic usefulness of time-sensitive computations and communications we outline above, it is important for the software stack and tooling to be able to fully support this useful mode of operations.

4.2. Special Cases and Scheduling Flexibility

The categorization of hybrid applications based on the relative demand for quantum and classical resources provides a useful starting point for designing resource allocation strategies and guiding scheduler behavior. However, in practice, more fine-grained classifications may also be warranted. These could account for factors such as algorithmic structure, tolerance to execution delays or job duration.

Moreover, certain applications may require exceptions to general scheduling policies. Examples include benchmarking jobs, debugging or testing runs, and workflows tied to time-sensitive experimental campaigns or calibration events. Such cases may not align neatly with the high/low resource usage patterns defined earlier, and instead demand more dynamic treatment within the scheduler. These scenarios can be accommodated by incorporating metadata into job submissions—such as user—defined priority levels or job types—that enable policy overrides or custom handling.

While the current work does not aim to implement these mechanisms directly, we recognize their potential value and consider them an important direction for future extension of this effort.

5. Resource Management

The effective management of resources is an essential component in a QC/HPC software stack. A balance must be met between resource utilization (*platform viewpoint*) and application productivity (*application viewpoint*). The platform viewpoint seeks to maximize utilization of the resources, which keeps the available resources as busy as possible. The application may have a different objective, whereby the focus is on minimizing time to solution. This latter viewpoint may care less about overall utilization if it simplifies the programming experience or provides benefits strictly to the singular application (i.e., ignoring consequences to other users of the shared resource). These resource management viewpoints are not unique to the QC/HPC context, but do help guide choices for interfaces and policies to support coordinated use cases that emerge in coupled QC/HPC systems.

5.1. Allocation

The allocation of computational assets is driven by the application usage patterns, which involve quantum and classical steps that determine the allocation strategies (Figure 3). The application patterns outlined in Section 4 identify three general usage modes, characterized by the application's balance of quantum and classical computing (i.e., *High QC/Low classical*, *Low QC/High classical*, and *roughly equal*).

When quantum and classical computing power must be allocated concurrently and remain reserved for the same duration, we refer to this as *simultaneous allocations* (Figure 3a). An *interleaved allocation* allows for independent reservations, potentially overlapping or forming a chained sequence (see Figure 3b). For example, HPC capabilities might be used first for pre-processing, followed by quantum systems for execution, with results stored for later post-processing on HPC infrastructure. We use “HPC" instead of simply “classical" in this example to emphasize the allocation dynamics of hybrid QC/HPC applications.

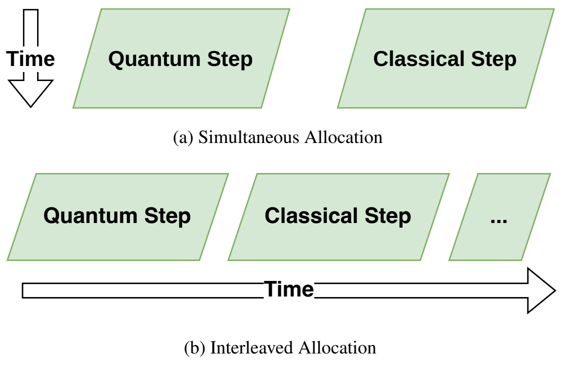


Figure 3: Allocation strategies to support hybrid QC/HPC applications.

The steps in the hybrid application correspond to work that must be managed by the quantum and classical reservation systems, as shown in Figure 4. The typical HPC approach has a single work queue that is used to submit jobs for execution on the HPC compute nodes. The HPC scheduler selects the appropriate time and location for the job and grants exclusive access to a set of compute nodes for a fixed time period. This access is governed by policies that help balance system usage. Similar to the HPC scenario, the individual quantum computational devices are used exclusively for the duration of a given computation (i.e., circuit execution) with appropriately defined policies.

5.2. Scheduling

Currently, we are exploring the use of the existing HPC workload manager, SLURM, to manage both HPC and Quantum resources. We use the heterogeneous job (hetjob) capability in SLURM to enable different resource specifications to be given for the HPC and QC (job) portions of a hybrid application. This allows us to support the simultaneous allocation mode.

We are currently investigating an approach that would leverage SLURM's features to define generic resources that would represent a QC device. For example, the Generic RESource [22] functionality allows for defining a named resource and quantity that can be used to track availability of quantum resources. This will allow us to support both the simultaneous and interleaved allocation strategies. This integration of both HPC and QC devices under a single resource management system will enable more detailed investigations into different scheduling policies and prioritization schemes.

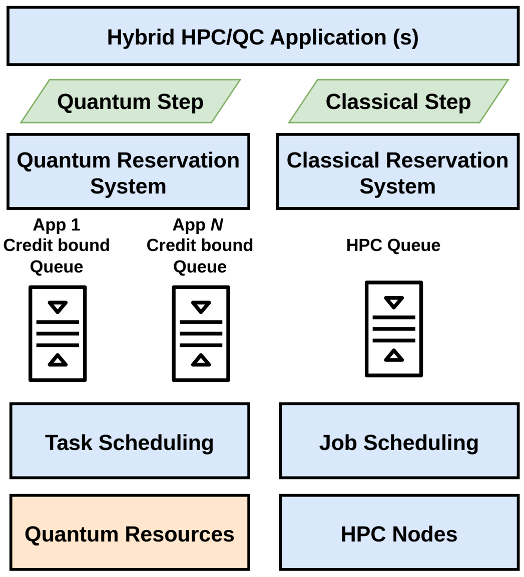


Figure 4: Illustration of hybrid QC/HPC application resource management. The classical reservation system uses a system wide queue. The quantum resources are exposed via a reservation system that determines the scheduling of tasks (ciruits). The illustration highlights our envisioned approach to the quantum resource management whereby credit-based application queues help provide practical time bounds needed for hybrid applications.

5.3. Multi-resource Scheduling

Hybrid QC/HPC applications leverage multiple resources to solve problems that are computationally expensive. The resource requirements are provided to help guide scheduling decisions for allocating access to the computing devices. As mentioned previously, the traditional HPC approach is to block allocate a set of compute nodes for exclusive (single user) access. Whether using a simultaneous or interleaved approach, the QC resources must be usable during the time window that aligns with the HPC compute allocation. This introduces timing constraints for the hybrid applications, which involve coordination among different scheduling and resource management systems.

The quantum resource manager is driven by requests to execute tasks (i.e., quantum circuits). The system tracks what quantum devices are in-use, and allocates available devices to run tasks (quantum circuits). The tasks run exclusively on the quantum devices. The allocation of the quantum devices is governed entirely by the resource manager to maximize utilization of the scarce quantum resources.

The coordinated scheduling of the HPC and QC resources introduces several interesting problems. Chief among these is the fact that task duration is not uniform across the HPC and QC resources. In the case of a simultaneous allocation, the HPC portion uses the QC hardware and must have some quality of service on quantum task activation (i.e., bounded request time). Otherwise, the HPC resource will idle waiting for results from the QC resource. Therefore some form of "*soft allocation*" is needed to constrain the activation time for quantum task submissions. A credit system can be used to provide these service agreements in order to meet general bounds on execution.

The QC/HPC software stack needs to provide support for this two-level scheduling problem to manage quality of service constraints for the hybrid applications using combined resources. The current plans are to explore the creation of dynamic work queues for the quantum application contexts, as illustrated in Figure 4 (e.g., *App N credit bound queue*). These hybrid application queues would help control time bounding via a credit system to help throttle requests to the quantum resource. This is similar to other techniques employed in HPC where shared resources need to be used during phases of the application and need some degree of quality of service guarantees (e.g., network latency/bandwidth, parallel filesystems). The credit-based approach is often useful to ensure provisioning of the shared resource. The hybrid application requires access to quantum resources that is managed via a "*soft reservation*" to ensure bound on quantum task activation time. This will require development of new mechanisms and scheduling capabilities to expose quantum resource load to enable more advanced coordinated scheduling of hybrid applications.

6. Hybrid QC/HPC Application Preparation

Having examined various application types and resource management approaches for hybrid quantum-classical computing, we now turn our attention to the critical preprocessing steps needed to prepare these hybrid applications for successful execution. A hybrid application consists of two segments: classical and quantum code. The classical code follows standard handling procedures, while the quantum code undergoes several compiler passes before reaching the hardware, as illustrated in Figure 5. Typically, a quantum circuit is programmed using generic gates, then lowered to an intermediate representation (IR). User-specified transformations, such as circuit cutting and gate reduction, are applied to optimize the circuit. The circuit is then transpiled into hardware-compatible gates, which are ultimately converted into pulses by the quantum controller for execution.

Hybrid applications can be either interpreted or compiled, requiring a unified software stack to support both approaches. In the interpreted case, all outlined steps are executed on demand once the application builds and runs the circuit, with classical processing up to the pulse generation stage. The proposed software stack abstracts the steps beyond circuit construction while providing configurable controls for quantum circuit transformations, aligning with familiar frameworks like Qiskit [2].

The compiled case differs in that some of the quantum processing passes can occur while compiling the hybrid application. The GPU compilation process described in Section 3 offers a valuable framework that can serve as a model for developing the analogous process in QC. Figure 6 illustrates a high-level view of a quantum enabled compilation flow.

A hybrid application delineates quantum tasks using specialized syntax. A quantum-enabled compiler is then able to pre-process and split the code into host and quantum segments. The quantum segments are lowered into an IR, where compiler passes optimize the quantum operations further. These compiler passes can be configured by the user. If the target quan-

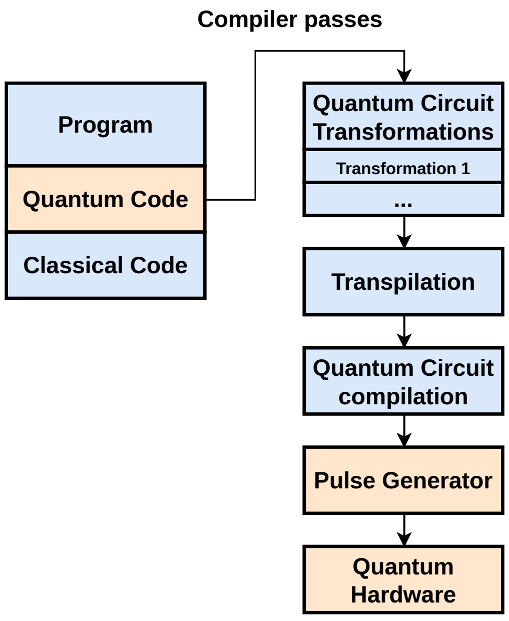


Figure 5: Quantum circuit processing passes.

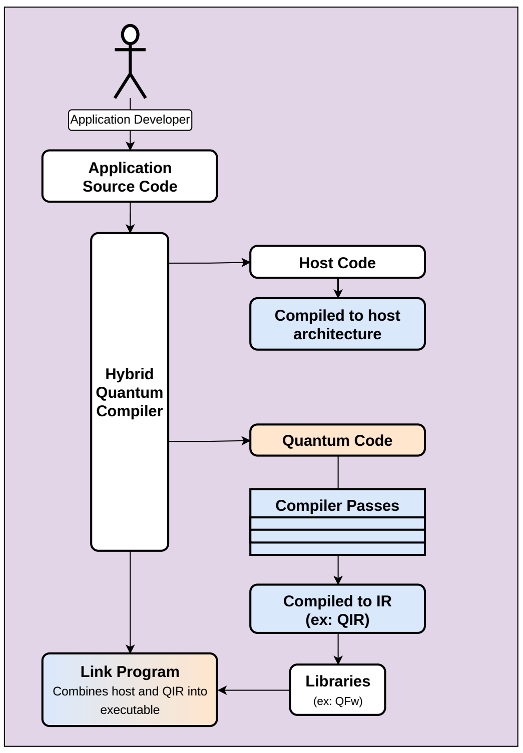


Figure 6: Quantum compiler toolchain.

tum hardware is known, the circuit can be lowered further into hardware specific IR; otherwise, it remains in the generic IR form.

The host code is compiled into the target architecture using standard compilers, e.g., `gcc` or `clang`. Both the IR and the compiled host code are then linked against the necessary

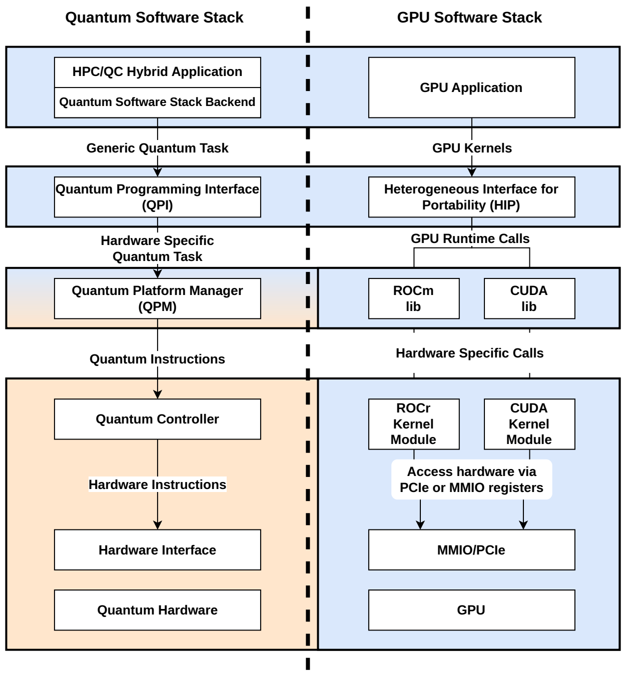


Figure 7: Quantum accelerator software stack layers in contrast to a typical GPU accelerator software stack.

libraries into a binary that can be executed on classical computing resources.

The target quantum hardware may not be known at the time of the hybrid application compilation. Even if it is known, the dynamic nature of quantum circuit formulation and ongoing hardware calibration, which can alter specific hardware attributes, necessitate an additional transpilation pass followed by translation into pulses. The same infrastructure that processes quantum code, from lowering it to a generic IR to generating pulses, can be utilized both during compilation and at runtime when the hybrid application requests a quantum circuit execution. Therefore, it is essential to formalize the interfaces to that infrastructure, henceforth referred to as the Quantum Toolchain, as shown in Figure 8.

7. QC/HPC Architecture

7.1. Software Layer View

To achieve the described functionality, we propose dividing the software stack into distinct software layers, with each layer providing an appropriate abstraction level to the layer above it as shown in Figure 7. In that diagram, we draw parallels between the GPU software stack layers and the proposed quantum software stack layers.

The highest layer of the software stack is the hybrid application itself, which can run either through an interpreter or as a compiled executable. Similar to how an MPI application initializes the MPI library, the hybrid application must initialize the software stack. This initialization process enables the software stack to discover available quantum resources and execute the necessary setup procedures. Once initialized, the application can leverage the most appropriate circuit-building package for its needs. For interpreted applications, Qiskit is one example of such a package.

The proposed software stack provides a backend for a select set of circuit-building packages. The purpose of this backend is to allow the application to use these packages as it normally would, with the only difference being the selection of the backend. This approach enables application developers to build and test their code locally using simulators before seamlessly transitioning to a QC/HPC environment when ready. The backend communicates with the rest of the software stack through the Quantum Programming Interface (QPI) layer.

The QPI layer offers several categories of APIs designed to simplify interactions with the software stack. These include initialization and finalization of the software stack, device management for discovering and configuring quantum devices, tool management for defining and configuring compilation passes, execution control for launching quantum tasks, resource management for configuring how resources are allocated, and result and error handling. The QPI is implemented as a library linked against the hybrid application and also provides Python bindings to support Python-based applications. The QPI can be accessed directly by the hybrid application or, when used with a package like Qiskit, through the provided backend.

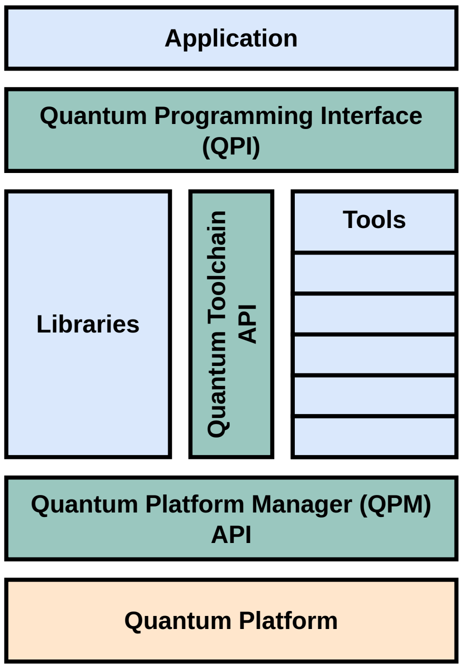
To submit a quantum task for execution, the application uses the execution control APIs provided by the QPI layer. After processing the task, the QPI layer hands off a hardware-specific quantum task to the Quantum Platform Manager (QPM). The QPM serves as a hardware abstraction layer, offering APIs to facilitate quantum task submission, result retrieval, and queries regarding device status, attributes, and configuration.

The QPM is implemented as a plugin-enabled library that includes a set of utility operations for all hardware providers. The plugin architecture allows each quantum hardware provider to develop their own plugin that implements the QPM API. Common utilities like an RPC communication layer or a scheduling system will be provided by the QPM library. These are intended to ease plugin development. This modular design enables the software stack to support multiple quantum hardware platforms without requiring modifications to the higher layers.

7.2. *Interface Normalization*

Building on the software layer view introduced in Section 7.1, we further define a set of normalized interfaces, as illustrated in Figure 8. These interfaces standardize interactions across different hardware and software layers, reducing complexity for hardware vendors and enhancing interoperability. By formalizing these interfaces, different entities can independently develop various components of the software stack while ensuring seamless integration between them.

Two key interfaces, the Quantum Programming Interface (QPI) and the Quantum Platform Manager (QPM), were mentioned earlier and will be discussed in more details in the following sections. Positioned at the top and bottom of the stack, respectively, their primary role is to standardize interactions between the application layer and the underlying quantum hardware. The first providing an application-friendly interface and the latter a hardware-friendly interface.



Application
Quantum Programming Interface (QPI)
Libraries
Quantum Toolchain API
Tools
Quantum Platform Manager (QPM) API
Quantum Platform

Figure 8: The diagram illustrates the proposed common interfaces that enable interoperability across different implementations. The hybrid application interacts with the software stack through the Quantum Programming Interface. Quantum circuit transformation tools can be integrated via the Quantum Toolchain Interface, while quantum hardware connects to the software stack by implementing the Quantum Platform Manager API.

7.3. *Quantum Platform Manager*

The Quantum Platform Manager (QPM) API provides a unified abstraction layer for quantum hardware, enabling the software stack to remain hardware-agnostic by exposing hardware capabilities and features consistently. However, given the diverse modalities and varying hardware capabilities, a single generic API risks being overly broad, making it challenging to express hardware configurations and features effectively. To address this, the QPM API is divided into three categories. The first, Resource Management APIs, allows the resource manager to allocate resources for hybrid applications. The second, Runtime APIs, enables hybrid applications to execute quantum tasks during operation. The third category, still under investigation, includes hardware-specific APIs that focus on low-level primitives, such as gate and pulse level control, tailored to unique hardware features. These hardware-specific APIs are designed to map directly to hardware capabilities, while more complex logical operations are handled by higher-level layers, such as the Quantum Programming Interface (QPI). A listing of the QPM APIs is provided in the supplementary material.

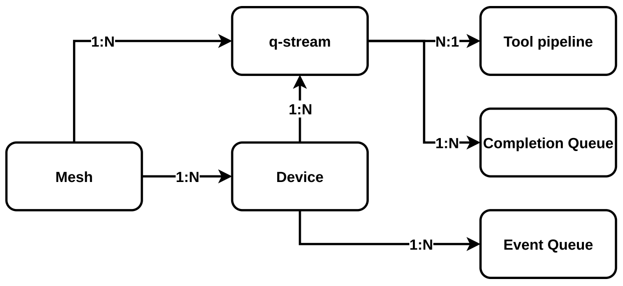


Figure 9: Quantum Programming Interface (QPI) object relationship diagram.

7.4. *Quantum Programming Interface*

The Quantum Programming Interface (QPI) acts as a high-level abstraction layer that simplifies how applications interact with diverse quantum hardware. It provides a consistent, application-friendly API that presents a unified view of available resources, allowing applications to configure their execution environment according to their specific needs. This high-level perspective contrasts with the low-level, hardware-oriented API of the Quantum Platform Manager (QPM), which is designed to be closer to kernel driver interfaces and more amenable to vendor implementation. By handling common resource orchestration tasks at the QPI level, the framework avoids redundant implementations across different QPMs. Through the standardization of essential constructs, the QPI promotes consistency across varied quantum hardware platforms, even when individual QPMs expose differing capabilities. Unlike comprehensive quantum programming frameworks such as Qiskit or PennyLane, the QPI does not define a programming paradigm. Instead, it focuses on enabling efficient integration and execution of quantum tasks on heterogeneous backends. At the same time, it supports access to hardware-specific optimizations via QPM-provided APIs, striking a balance between portability and performance.

Beyond abstraction, the QPI introduces a mechanism for configuring and managing the quantum software toolchain. Through its tool management APIs, users can define compiler passes, including circuit cutting or gate reduction, and delegate task execution across available high-performance computing (HPC) resources. The QPI is responsible for orchestrating the software stack, selecting appropriate quantum backends, and tailoring the final transpilation stage based on the targeted hardware. It also supports event-driven execution by offering constructs for monitoring task completion and system-level events such as noise fluctuations, calibration updates, or hardware failures, thereby improving operational transparency. To support efficient quantum task execution, the QPI provides a q-stream construct that groups quantum tasks for sequential or parallel execution. These tasks can be explicitly assigned to resources by the user or dynamically scheduled by the QPI based on resource availability. Together, these features position the QPI as a key architectural component for enabling scalable, hybrid quantum-classical workflows.

7.4.1. *QPI Object View*

The QPI design, illustrated in Figure 9, abstracts access to one or more quantum resources through a set of logical constructs. At its core, the Mesh object encapsulates one or more quantum devices, with operations applied to the Mesh affecting all contained devices. These operations are managed by a q-stream object, a container for a sequence of operations—ranging from simple device queries to full quantum task execution, that can be executed together. Multiple q-streams can be bound to a single Mesh, each representing a distinct set of operations configured to run sequentially or in parallel. A Mesh can also distribute operations across its devices, supporting either embarrassingly parallel execution or complex entanglement procedures, such as distributing a quantum circuit across multiple devices within the same Mesh.

Each quantum resource is represented as a device object within a Mesh, and individual devices can also have one or more q-streams bound to them, with operations restricted to the specific device. An event queue object, bound to a device, captures events such as noise incidents, calibration updates, or hardware faults; multiple event queues can be created to handle different event types, but events are ignored if no queues are bound. Similarly, a completion queue, bound to a q-stream, receives notifications when operations in the q-stream complete, such as quantum task results or calibration outcomes. If a q-stream runs operations in parallel, notifications may arrive out of order; otherwise, the QPI ensures ordered delivery. A q-stream can have multiple completion queues, each dedicated to specific completion types.

Finally, the tool pipeline object manages tools applied to quantum tasks, such as compiler passes for circuit cutting or gate reduction. A single tool pipeline can be bound to multiple q-streams, ensuring consistent tool application across all quantum tasks originating from those q-streams. These constructs collectively enable the QPI to streamline interaction with quantum hardware, monitor operation completion and system events, and optimize hybrid quantum-classical workflows. The QPI module is currently under active development. The proposed API is included in the supplementary materials.

7.5. *Quantum Toolchain*

Another critical interface is the Quantum Toolchain API. As discussed in Section 6, quantum tasks require transpilation at runtime, even if they have already undergone compilation at compile-time. This transpilation process is not limited to converting a high-level quantum task into a hardware-compatible form; it also includes essential optimizations such as, gate reduction and circuit cutting. These operations refine quantum tasks to maximize efficiency and adapt them to the constraints of specific hardware backends.

The objective of this software stack proposal is to formalize the interface to these tools, enabling tool developers to extend the stack without requiring modifications to other components. By establishing a well-defined Quantum Toolchain API as in Figure 8, the stack ensures flexibility, allowing new tools and optimizations to be integrated seamlessly while preserving interoperability with existing software and hardware layers.

Each tool will take quantum programs expressed in QIR [32] or OpenQASM [35], perform circuit transformation, and produce a modified QIR or OpenQASM representation that can be passed to the next tool in the pipeline. This process forms a toolchain of operations, where a quantum task is introduced at the top of the pipeline and progressively refined through a series of transformations. At the end of this pipeline, the final version of the quantum task is fully optimized and lowered into a hardware-compatible format to be accelerated to the corresponding backend [31, 34, 25]. For example, the execution engine for QIR [31] (QIR-EE) can be invoked for the purposes of parsing, interpreting and executing QIR across multiple hardware platforms. The software stack, through the QPI, provides applications with the ability to configure this pipeline to suit their specific needs, ensuring flexibility in how quantum tasks are processed and optimized before execution.

Additionally, the tools themselves can leverage HPC resources for execution, as certain transformations, such as large scale circuit optimization or gate reduction, can be computationally intensive and time-consuming. By utilizing the available HPC allocation, these operations can be distributed across multiple compute nodes, significantly reducing processing time and enabling efficient handling of complex quantum tasks.

7.6. Architectural Overview

The architecture depicted in Figure 10 outlines a more detailed breakdown of the software stack. The Quantum Framework (QFw) includes the quantum software stack backend, the QPI Library implementation, the quantum toolchain implementation and the simulation environment, which includes multiple simulator support, e.g., TNQVM [27] and NWQ-Sim [26]. It also includes a set of scripts which give the user a familiar HPC like interface. The prototype we have implemented thus far realizes key components of the architecture and serves as a means to validate our conceptual designs. It has been described in greater detail in our previous work [5] as well as in the supplementary material.

We will analyze the architecture with a top-down approach. Typically, users would compose an sbatch script for submission as a SLURM job on a supercomputer like Frontier, where the script specifies the resources needed for the job, which could be HPC resources, QC resources, or a combination thereof. The QFw provides a set of startup scripts that initialize its infrastructure. These scripts should be incorporated into the sbatch script when launching a job. The design capitalize on two SLURM features, Generic RESource (GRES) and Heterogeneous Job features. The latter is used to identify two sets of resources which need to be allocated at the same time, as shown in Figure 11.

The first component indicates a request for ten nodes from the compute cluster while the second component requests one QC node. GRES allows the specification of criteria which is passed down to the GRES plugin aiding in the selection of the quantum resource. As described in Section 5, during this step the QC reservation system is engaged to determine if the QC resource can accommodate the job. Only when the QC and HPC reservation is successful will the hybrid application reservation

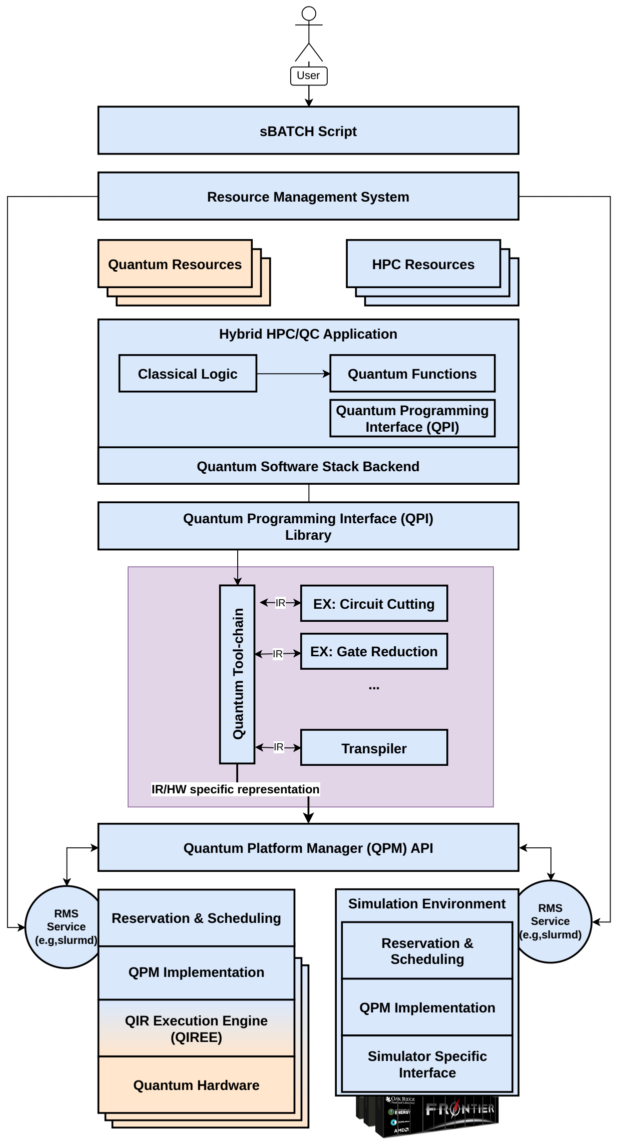


Figure 10: QC/HPC detailed software stack architecture

```
#!/bin/bash

# job component 1
#SBATCH -A stf008
#SBATCH -N 10
#SBATCH --partition=compute
#SBATCH -t 1:00:00

#SBATCH hetjob

# Heterogeneous job definition
# for the QC node
#SBATCH --partition=quantum
#SBATCH --nodes=1
#SBATCH --ntasks=1
#SBATCH --gres=qc:QC:1
#SBATCH --time=01:00:00
```

Figure 11: Example of a SLURM heterogenous job that includes two parts representing the HPC and QC sides of the hybrid application; 10 nodes allocated for the HPC portion, and 1 node for the QC portion.

```
# Setup the QFw infrastructure
qfw_setup.sh
# Run the application
qfw_srun.sh python3 application.py
# Teardown the QFw infrastructure
qfw_teardown.sh
```

Figure 12: Example of QFw helper scripts for framework setup, application execution wrapper and framework teardown.

be granted and the rest of the script execute. The sbatch script can then use the provided scripts to setup the QFw infrastructure, run the application and teardown the QFw infrastructure as shown in Figure 12.

As discussed in Section 5, ensuring the most efficient use of quantum hardware is paramount. Given the high demand and limited availability of QPUs, an effective resource management strategy is essential to maximize utilization and minimize idle time. To address this, the QFw will provide a reservation and scheduling library. This library will be responsible for handling job reservations, ensuring that quantum resources are allocated efficiently. During job reservation, this library will be used to request and secure access to the quantum hardware before the job runs.

The cluster SLURM controller will interact with SLURMd to determine the availability of quantum resources. As illustrated in the corresponding diagram, the reservation library is designed to integrate with the SLURMd plugin running on the quantum gateway via a GRES plugin. The reservation library will leverage the QPM API to query quantum hardware for resource-specific details, such as qubit availability, calibration status, and expected circuit runtime constraints. Based on these criteria, the library will make an informed reservation decision, ensuring that job scheduling aligns with the current state of the quantum hardware. Once a reservation decision is made, the result is communicated back to the SLURM controller, which finalizes the scheduling process and allocates the necessary resources accordingly.

Once the resources are granted, the application begins execution on the classical compute portion of the allocation. This may include HPC nodes or, in cases where only quantum resources are requested, the quantum gateway. The application first calls the initialization routine of the QPI, setting up the necessary components before proceeding with its computational tasks. During execution, the application configures the tool pipeline and launches quantum tasks through appropriate QPI calls. The QPI library ensures that each quantum task is processed through the tool pipeline and ultimately lowered down to the target hardware. To achieve this, the QPI interacts with the QPM API, querying hardware-specific details and selecting the appropriate transpiler based on the returned information. Once the quantum task is processed through the toolchain, the QPI then invokes the QPM APIs responsible for executing the quantum task on the hardware. Since the hardware could be managing multiple jobs, the QPM utility layer, provides a scheduling library, as outlined in the system architecture, to coordinate between the different jobs ensuring tasks are completed within a time bound as discussed in Section 5.

Once execution is complete, the results are propagated back up the stack to the application. From the application's perspective, this process appears as a standard synchronous or asynchronous function call, abstracting the complexities of quantum execution. Finally, after completing all quantum and classical computations, the application calls the QPI finalization routines, which gracefully clean up allocated resources and terminate the session.

7.7. *Simulation Environment*

Quantum computers are expected to remain a scarce resource for the foreseeable future. As a result, classical quantum simulators will continue to play a crucial role in aiding researchers with testing and debugging their applications before deploying them on actual quantum hardware. To support this need, the QFw infrastructure provides a simulation environment designed to simplify the integration and management of multiple simulator backends.

The simulation environment can be assigned multiple HPC nodes, allowing it to fit seamlessly into the overall software architecture described thus far. It achieves this integration by implementing a QPM plugin, similar to how real quantum hardware interfaces with the software stack. Since each simulator may have unique characteristics and requirements, it is possible for each simulator backend to have its own QPM implementation, ensuring compatibility with the broader quantum execution framework.

Managing the various simulator instances is a key function of the simulation environment. This is accomplished using the PMIx Reference Run-Time Environment (PRTE), a well-established library designed to handle parallel process execution on HPC clusters. A single PRTE instance runs within the simulation environment and manages all allocated nodes. This setup allows applications to utilize multiple types of quantum simulators simultaneously, with PRTE orchestrating their execution.

Since the simulation environment spans multiple HPC nodes, it can distribute quantum task simulations across available resources by launching multiple simulator instances concurrently. If a simulator supports MPI operations, the environment can take advantage of all allocated nodes to run a single large-scale simulation, significantly enhancing circuit simulation capabilities. However, the scalability of this approach is constrained by factors such as the simulator type, the number of available nodes, and the total memory capacity.

By providing a flexible and scalable simulation environment, the QFw infrastructure facilitates experimentation, benchmarking, and debugging. Researchers can test their algorithms in a stable environment before deploying them on quantum hardware, improving efficiency and reducing the need for costly quantum computing time.

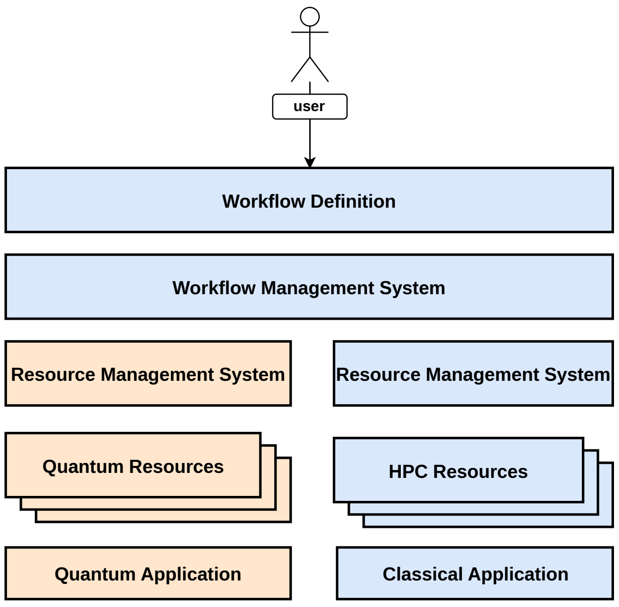


Figure 13: Illustrate the integration of the quantum software stack described in this paper with workflow management systems

8. Architecture Validation

8.1. *Workflow Integration and Service Management Architecture*

The integration of quantum computing within HPC environments requires robust workflow management and service infrastructure to support diverse operational patterns. Our architecture addresses this through three complementary components: (1) distributed workflow management, (2) secure service integration, and (3) comprehensive system telemetry. The software stack architecture proposed in this paper is designed to work seamlessly with workflow orchestration frameworks that manage distributed quantum-classical workflows across heterogeneous resources.

Traditional HPC ecosystems typically rely on direct SSH access, batch schedulers like SLURM, and manual job submission processes. However, quantum-HPC integration requires more sophisticated infrastructure capabilities including: (1) programmatic API access for automated systems, (2) real-time data streaming between experimental facilities and compute resources, (3) fine-grained authorization and policy enforcement, (4) workflow orchestration across heterogeneous resources, and (5) secure communication channels for sensitive quantum algorithms. A service mesh architecture addresses these requirements by providing an infrastructure layer that facilitates secure and efficient communication between distributed services while abstracting networking complexity and enforcing authentication, authorization, and traffic management policies.

The proposed software stack provides key abstractions for resource allocation, and task decomposition, enabling efficient coordination between quantum and classical computing elements. As mentioned in Section 5, it will also ensure quantum resources are utilized optimally, by supporting dynamic resource allocation and intelligent task scheduling. These features can be further leveraged by workflow management systems such as Pilot-Quantum [13] to handle complex dependency patterns such as variational quantum algorithms and circuit cutting workflows.

OLCF's Secure Scientific Service Mesh (S3M) [14] will provide the foundational infrastructure for secure and flexible integration of quantum resources within the broader HPC ecosystem. S3M will enable controlled access to quantum and classical resources through policy-driven interfaces, supporting both traditional HPC workloads and emerging quantum-classical hybrid applications. The framework implements sophisticated access control, rate limiting, and centralized authentication, while providing a secure bridge between external workflow systems and internal HPC resources. This architecture will allow quantum applications to leverage HPC resources as computational accelerators within larger scientific workflows, while maintaining operational security and trust boundaries.

Other HPC centers can implement similar capabilities through various approaches:

1. *API Gateway Solutions:* Centers using traditional batch schedulers can implement RESTful API layers that provide programmatic access to SLURM/PBS systems while maintaining existing authentication mechanisms.

2. *Container Orchestration Platforms:* Sites with Kubernetes or OpenShift deployments can leverage service mesh technologies to provide similar traffic management, security, and observability features.

3. *Workflow Management Integration:* Existing workflow systems can be extended with quantum-aware scheduling and resource management capabilities.

4. *Cloud-Native Approaches:* Centers adopting cloud-native architectures can utilize managed services from cloud service providers to achieve comparable functionality.

The key architectural requirements (secure API access, real-time streaming workflow orchestration, and policy reinforcement) can be satisfied through different technological implementations while maintaining the same functional capabilities for quantum-HPC integration.

The system also incorporates comprehensive telemetry capabilities that collect operational metrics across both quantum and classical resources. A dedicated telemetry service utilizes the QPM APIs to gather quantum hardware-specific data, including qubit calibration status, error rates, and queue depths. This data is stored in a persistent database alongside classical system metrics, enabling correlation analysis and performance optimization. The telemetry infrastructure supports both real-time monitoring for operational decision-making and historical analysis for resource utilization optimization. Through integration with S3M, the telemetry data could be securely accessed by authorized workflow systems and analysis tools, facilitating informed scheduling decisions and hardware-aware optimization of quantum-classical workflows.

```
# Initial implementation
dev_mu = qml.device("lightning.qubit",
        wires=tot_qubits, shots=n_shots)
```

(a) Initial implementation

```
# QFw porting
from qfw_qiskit import QFWBackend,
  ↪QFwBackendType, QFwBackendCapability

nwqsim = QFWBackend(
    betype=QFwBackendType.QFW_TYPE_NWQSIM,
    capability=QFwBackendCapability.
      ↪QFW_CAP_STATEVECTOR)

dev_mu = qml.device("qiskit.remote",
        wires=tot_qubits, backend=nwqsim,
          ↪shots=n_shots)
```

(b) QFw port

Figure 14: Listing showing required changes for port of VQLS using Xanadu's Lightning simulator [48] to the QFw backend.

8.2. *Application Experience*

We tested the QFw framework with an NWQ-Sim backend to tackle a hybrid quantum-classical application: the variational quantum linear solver (VQLS) [6]. VQLS is a variational approach for solving linear systems of equations of the form $A\mathbf{x} = \mathbf{b}$, offering a scalable solution for quantum linear solvers in existing NISQ devices. It has practical applications in various fields, including fluid dynamics [12, 10], which is our specific focus–solving canonical fluid flow problems using NISQ devices [8, 1].

The algorithm formulates an optimization problem where a quantum circuit represents the solution vector $\mathbf{x}$, denoted as $\hat{\mathbf{x}}$. The objective is to minimize the error in reconstructing $\mathbf{b}$ using $\hat{\mathbf{x}}$, which defines the cost function to be minimized. The optimization is performed iteratively (over multiple epochs), with a classical optimization technique adjusting the parameters of the quantum circuit at each epoch. The cost function is computed using a QPU.

We implemented VQLS using Pennylane [4]. The integration with QFw is seamless from the application user's perspective–the only modification required is defining the Pennylane `device` variable, as shown in Figure 14. For the specific test that we performed, we solved a 2×2 matrix problem, resulting in a circuit with 3 qubits. We used a simple ansatz involving Hadamard and RY rotation gates on each qubit, giving 3 parameters to be optimized. One function evaluation of this problem for optimization requires execution of 72 circuits.

As we continue exploring different ansatzes for our application, porting our VQLS implementation to the QFw revealed challenges related to optimizer selection. The initial PennyLane implementation was validated using local simulators to assess both accuracy and computational speed. The QFw framework provided easier access to more detailed insights, including a log of executed circuits, queuing times before execution, execution times per circuit, and other performance metrics. For example, the total time per epoch for our use case was 342 sec, with an average time of ≈ 0.4 sec per circuit evaluation. Using the Lightning simulator, running on a local node, the time per circuit evaluation was ≈ 0.003 sec. This profiling information revealed that a large number of circuits were being submitted to optimize the circuit parameters at each epoch - 480 circuit evaluations per epoch. This stemmed from the use of a gradient-based optimization method that requires large number of circuit evaluations per parameter update. Initially, we hypothesized that due to the complexity of our end application, a gradient-based strategy, such as stochastic gradient descent, might be necessary to traverse the optimization landscape.

To address this, we replaced the optimizer with a gradient-free optimization approach: the Constrained Optimization by Linear Approximation (COBYLA) method [9, 11], which performs only 1 function evaluation per epoch. This modification reduced the optimization time for our application due to lower number of function (circuit) evaluations - only 1 function evaluation per epoch, resulting in 72 circuit evaluations per epoch. Gradient-based optimization can be useful to tackle certain scenarios with complex function evaluations to compute the gradients. The analysis using QFw denotes that the choice and applicability of the optimizer for our application warrants further exploration. The original Pennylane implementation did not reveal this inefficiency, as local simulators have negligible overhead in circuit execution and parameter updates for such small test cases. Similarly, running on a real QPU through cloud services need not expose this bottleneck either, since the connection latency to the quantum cloud service would overshadow the computational overhead of evaluating numerous circuits. The profiling and debugging capabilities enabled by QFw for simulators can be invaluable for identifying such inefficiencies in hybrid applications.

Looking toward the near future, quantum computers will likely remain standalone machines interfaced with HPC systems via high-speed networks, adding layers of complexity to hybrid quantum-classical workflows. These systems will incur overheads from the surrounding infrastructure—such as latency from framework layers, communication delays between HPC and quantum subsystems, queuing delays, or device—specific I/O bottlenecks. The software stack can often mitigate framework latency by maximizing the use of HPC resources for processing quantum tasks and optimizing network communications. However, these optimizations will not eliminate latency limitations entirely.

Applications will also need to be mindful of circuit generation and execution. As described above, applications like VQLS—or other variational algorithms—when scaled to larger problem sizes, may generate thousands of quantum circuits. Moreover, some of this circuit generation may be inherently sequential, as the structure of a given circuit can depend on results obtained from previously executed ones, thereby limiting opportunities for parallelism and making the overall workflow more sensitive to latency-related overheads.

Nevertheless, efforts should be made to parallelize circuit generation whenever possible. For example, Lu et al. [3] demonstrate a workflow (not involving hybrid Q-HPC execution) that enables parallel circuit generation for the Quantum Phase Estimation (QPE) algorithm. More broadly, application developers should remain mindful of the structure and dependencies within their workflows. Latency constraints that arise from sequential circuit generation or tight classical-quantum feedback loops are often difficult to mitigate implicitly at the framework level. Designing algorithms and workflows that expose opportunities for concurrency, where possible, can significantly reduce end-to-end execution time in hybrid settings.

9. Related Work

The space exploring QC integration with existing HPC resources is relatively nascent, with a flurry of activity and collaborations surfacing only within the last few years. Some examples include conceptual renderings of QHPC middleware [47], an effort at ORNL [37] integrating quantum runtimes in parallel with other accelerators onto CPUs and GPUs within an established task-based kernel framework IRIS [38], and the Munich Quantum Valley [40], a consortium of research institutions and universities in Bavaria focused on a unified software stack for the HPC-QC ecosystem [39, 49]. However, other international teaming underscores the importance of pursuing robust integration efforts and and providing frameworks for effectively enhancing conventional computing. For example, HPC-QS [41], funded by the European Union, has intentions to integrate and couple two quantum simulators, with two existing European Tier-0 supercomputers, and to deploy an open European federated hybrid infrastructure. More recently, IBM joined forces with the Riken Quantum Computing Center [42], where the IBM system powered by a 133-qubit IBM Quantum Heron processor, would be co-located and integrated with the Fugaku supercomputer. Additionally, with Pasqal [43], they aim to develop a unified programming model built on Qiskit, aiming to integrate quantum and classical computing resources for HPC workflows. Lastly, IQM have partnered with Hewlett Packard Enterprise (HPE) for their version of quantum-HPC integration [44]. Other industry developments that can potentially enable effective hybrid QC-HPC include Nvidia's Cuda-Q [45] and Riverlane's Deltaflow [46].

10. Conclusion

This paper has developed a framework for the integration of Quantum Computing (QC) with High-Performance Computing (HPC) architectures. Building upon previous work that integrated GPU acceleration hardware into HPC, the paper details the design and functionality of a multi-layered software stack that aims to bridge these two computational paradigms in a seamless and hardware agnostic way. Our approach is to maintain the familiar HPC user experience while introducing quantum acceleration capabilities, addressing a range of application motifs that includes simultaneous and interleaved QC/HPC workflows.

Our framework emphasizes the necessity for low-latency interaction between classical and quantum resources. We integrate quantum hardware with classical HPC by developing standardized APIs such as the Quantum Platform Manager (QPM) that manages resource utilization effectively. At the top of the stack, the Quantum Programming Interface (QPI) abstracts the quantum resources, providing a unified interface for applications. The design accommodates resource sharing, batch processing of quantum tasks, and the use of simulation backends for testing and development, all of which contribute to maximizing the utilization of scarce quantum resources.

The introduction of a Workflow Management System for the QC/HPC interleaved workflows allows for optimized resource allocation and usage, ensuring that neither quantum nor classical resources remain idle. Moreover, we highlight the importance of telemetry and result persistence through a database system, which not only aids in machine health monitoring but also enhances the utility of results in both internal and external workflow contexts, particularly with OLCF's Secure Scientific Service Mesh (S3M).

The methodologies presented in this paper provide a blueprint for integrating quantum computing into existing HPC infrastructures. They address critical challenges such as latency, resource management, and workflow optimization, paving the way for a future where quantum computing can be seamlessly leveraged to accelerate scientific and computational tasks. As quantum technologies evolve, these frameworks will need to adapt, but the foundational concepts introduced here offer a robust starting point for this integration, promising to unlock new computational possibilities in scientific research and beyond.

The strategies outlined here begin to expose the complexities involved in advancing an adaptive ecosystem model for the next generation of HPC. Extensive progress has already been made during the transition from CPU-based to GPU-based parallel supercomputing over the last two decades, culminating in the recent crossing of the exascale barrier (see Section 3). The application-dependent challenges presented by integrating quantum resources, however, are apparent in the detailed development of the quantum framework (QFw) outlined above. The goals of balancing the demands of resource allocation/management, scheduling, data movement between heterogeneous processors, and the development of complex workflows that optimally coordinate the multi-user computing environment in a hardware-agnostic way present a computing grand challenge for the next decade. The long-term aim for the convergence of modeling and simulation, artificial intelligence, and quantum computing into powerful and adaptive tools that significantly accelerate scientific discovery is certainly a worthy goal for the computational science community.

Acknowledgments

This research used resources of the Oak Ridge Leadership Computing Facility at the Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725. Research sponsored by the Laboratory Directed Research and Development Program of Oak Ridge National Laboratory, managed by UT-Battelle, LLC, for the US Department of Energy. This work was also partially supported by Office of Advanced Scientific Computing Research's Accelerated Research in Quantum Computing Program MACH-Q project.

We thank Dr. Chao Lu (ORNL) for his contributions to the development of the VQLS application, which was used as a representative example in this work.

References

[1] M. Gopalakrishnan Meena, Y. Zhang, W. Jiang, Y. Lin, S. Günther, X. Gao, Towards a quantum algorithm for the incompressible nonlinear navier-stokes equations, in: 2024 IEEE International Conference on Quantum Computing and Engineering (QCE), Vol. 1, IEEE, 2024, pp. 662–668.

[2] A. Javadi-Abhari, M. Treinish, K. Krsulich, C. J. Wood, J. Lishman, J. Gacon, S. Martiel, P. D. Nation, L. S. Bishop, A. W. Cross, B. R. Johnson, J. M. Gambetta, Quantum computing with Qiskit (2024). `arXiv:2405.08810`, `doi:10.48550/arXiv.2405.08810`.

[3] C. Lu, M. G. Meena, K. C. Gottiparthi, Lugo: an enhanced quantum phase estimation implementation, arXiv preprint arXiv:2503.15439 (2025).

[4] V. Bergholm, J. Izaac, M. Schuld, C. Gogolin, S. Ahmed, V. Ajith, M. S. Alam, G. Alonso-Linaje, B. AkashNarayanan, A. Asadi, et al., Pennylane: Automatic differentiation of hybrid quantum-classical computations, arXiv preprint arXiv:1811.04968 (2018).

[5] T. Beck, A. Baroni, R. Bennink, G. Buchs, E. A. Coello Perez, M. Eisenbach, R. Ferreira da Silva, M. Gopalakrishnan Meena, K. Gottiparthi, P. Groszkowski, T. S. Humble, R. Landfield, K. Maheshwari, S. Oral, M. A. Sandoval, A. Shehata, I.-S. Suh, C. Zimmer, Integrating quantum computing resources into scientific hpc ecosystems, Future Generation Computer Systems 161 (2024) 11–25. `doi:10.1016/j.future.2024.06.058`.

[6] C. Bravo-Prieto, R. LaRose, M. Cerezo, Y. Subasi, L. Cincio, P. J. Coles, Variational quantum linear solver, Quantum 7 (2023) 1188.

[7] L. Lamport, LaTeX: A document preparation system, 2nd Edition, Addison Wesley, Massachusetts, 1994.

[8] M. Gopalakrishnan Meena, K. C. Gottiparthi, J. G. Lietz, A. Georgiadou, E. A. Coello Pérez, Solving the hele–shaw flow using the harrow–hassidim–lloyd algorithm on superconducting devices: A study of efficiency and challenges, Physics of Fluids 36 (10) (2024).

[9] M. J. Powell, A direct search optimization method that models the objective and constraint functions by linear interpolation, Springer, 1994.

[10] C.-C. Ye, N.-B. An, T.-Y. Ma, M.-H. Dou, W. Bai, D.-J. Sun, Z.-Y. Chen, G.-P. Guo, A hybrid quantum-classical framework for computational fluid dynamics, Physics of Fluids 36 (12) (2024).

[11] A. Pellow-Jarman, I. Sinayskiy, A. Pillay, F. Petruccione, A comparison of various classical optimizers for a variational quantum linear solver, Quantum Information Processing 20 (6) (2021) 202.

[12] R. Demirdjian, D. Gunlycke, C. A. Reynolds, J. D. Doyle, S. Tafur, Variational quantum solutions to the advection–diffusion equation for applications in fluid dynamics, Quantum Information Processing 21 (9) (2022) 322.

[13] P. Mantha, F. J. Kiwit, N. Saurabh, S. Jha, A. Luckow, Pilot-Quantum: A Quantum-HPC Middleware for Resource, Workload and Task Management, arXiv preprint arXiv:2412.18519 (2024).

[14] S. Oral, R. Ferreira Da Silva, S. Abraham, R. Adamson, V. Anantharaj, T. Beck, A. Barker, K. Bethea, J. Brown, M. Brim, et al., OLCF's Advanced Computing Ecosystem (ACE): FY24 Efforts for the DOE Integrated Research Infrastructure (IRI) Program, Tech. rep., Oak Ridge National Laboratory (ORNL), Oak Ridge, TN (United States) (2024).

[15] K. Rudinger, G. J. Ribeill, L. C. Govia, M. Ware, E. Nielsen, K. Young, T. A. Ohki, R. Blume-Kohout, T. Proctor, Characterizing midcircuit measurements on a superconducting qubit using gate set tomography, Phys. Rev. Appl. 17 (2022) 014014. `doi:10.1103/PhysRevApplied.17.014014`.
URL `https://link.aps.org/doi/10.1103/PhysRevApplied.17.014014`

[16] M. A. Nielsen, I. L. Chuang, Quantum computation and quantum information, Cambridge university press, 2010.

[17] B. Barber, K. M. Barnes, T. Bialas, O. Buğdaycı, E. T. Campbell, N. I. Gillespie, K. Johar, R. Rajan, A. W. Richardson, L. Skoric, et al., A real-time, scalable, fast and resource-efficient decoder for a quantum computer, Nature Electronics (2025) 1–8.

[18] HIP Documentation, Advanced Micro Devices, Inc. Accessed: 2025-02-15 (2025).
URL `https://rocm.docs.amd.com/projects/HIP/`

[19] CUDA Toolkit Documentation, NVIDIA, Inc. Accessed: 2025-02-15 (2025).
URL `https://docs.nvidia.com/cuda/`

[20] Qristal – The Quantum Brilliance SDK, Quantum Brilliance, Inc. Accessed: 2025-02-15 (2025).
URL `https://qristal.readthedocs.io/en/latest/`

[21] SLURM Work Manager, SchedMD, Inc. Accessed: 2025-02-15 (2025).
URL `https://slurm.schedmd.com/overview.html`

[22] SLURM: Generic RESource Scheduling, SchedMD, Inc. Accessed: 2025-06-04 (2025).
URL `https://slurm.schedmd.com/gres.html`

[23] J. Nickolls, I. Buck, M. Garland, K. Skadron, Scalable parallel programming with CUDA: Is CUDA the parallel programming model that application developers have been waiting for?, Queue 6 (2) (2008) 40–53. `doi:10.1145/1365490.1365500`.
URL `https://doi.org/10.1145/1365490.1365500`

[24] C. Lattner, V. Adve, LLVM: A compilation framework for lifelong program analysis & transformation, in: Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization, CGO '04, IEEE Computer Society, USA, 2004, p. 75.

[25] A. J. McCaskey, D. I. Lyakh, E. F. Dumitrescu, S. S. Powers, T. S. Humble, Xacc: a system-level software infrastructure for heterogeneous quantum–classical computing, Quantum Science and Technology 5 (2) (2020) 024002.

[26] A. Li, B. Fang, C. Granade, G. Prawiroatmodjo, B. Hein, M. Rotteler, S. Krishnamoorthy, Sv-sim: Scalable pgas-based state vector simulation of quantum circuits, in: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, 2021.

[27] T. Nguyen, D. I. Lyakh, E. F. Dumitrescu, D. Clark, J. Larkin, A. J. McCaskey, Tensor network quantum virtual machine for simulating quantum circuits at exascale, ACM Transactions on Quantum Computing 4 (2021) 1 – 21.
URL `https://api.semanticscholar.org/CorpusID:233324255`

[28] S. Atchley, C. Zimmer, J. Lange, D. Bernholdt, V. Melesse Vergara, T. Beck, M. Brim, R. Budiardja, S. Chandrasekaran, M. Eisenbach, T. Evans, M. Ezell, N. Frontiere, A. Georgiadou, J. Glenski, P. Grete, S. Hamilton, J. Holmen, A. Huebl, D. Jacobson, W. Joubert, K. Mcmahon, E. Merzari, S. Moore, A. Myers, S. Nichols, S. Oral, T. Papatheodore, D. Perez, D. M. Rogers, E. Schneider, J.-L. Vay, P. K. Yeung, Frontier: Exploring exascale, in: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC '23, Association for Computing Machinery, New York, NY, USA, 2023. `doi:10.1145/3581784.3607089`.
URL `https://doi.org/10.1145/3581784.3607089`

[29] D. Gottesman, An introduction to quantum error correction and fault-tolerant quantum computation, in: Quantum information science and its contributions to mathematics, Proceedings of Symposia in Applied Mathematics, Vol. 68, 2010, pp. 13–58.

[30] J. Bausch, A. W. Senior, F. J. Heras, T. Edlich, A. Davies, M. Newman, C. Jones, K. Satzinger, M. Y. Niu, S. Blackwell, et al., Learning high-accuracy error decoding for quantum processors, Nature (2024) 1–7.

[31] QIR-EE Developers, QIR-EE, [Computer Software] `https://doi.org/10.11578/qiree/dc.20250114.1` (2025).
URL `https://github.com/ORNL-QCI/qiree`

[32] QIR Alliance, QIR Specification, accessed: 2025-01-13 (2025).
URL `https://github.com/qir-alliance/qir-spec`

[33] QIR Alliance, The QIR Alliance, `https://www.qir-alliance.org/alliance/`, accessed: 2025-02-05 (2025).

[34] E. Wong, V. L. Ortega, D. Claudino, S. Johnson, S. Afrose, M. Gowrishankar, A. M. Cabrera, T. S. Humble, A cross-platform execution engine for the quantum intermediate representation (2024). `arXiv:2404.14299`.
URL `https://arxiv.org/abs/2404.14299`

[35] A. W. Cross, L. S. Bishop, J. A. Smolin, J. M. Gambetta, Open quantum assembly language (2017). `arXiv:1707.03429`.
URL `https://arxiv.org/abs/1707.03429`

[36] N. Ma, H. Li, Understanding and estimating the execution time of quantum programs (2024). `arXiv:2411.15631`.
URL `https://arxiv.org/abs/2411.15631`

[37] M. A. H. Monil, E. Wong, N. R. Miniskar, V. Leyton-Ortega, A. Cabrera, J. S. Vetter, T. S. Humble, Q-IRIS: The Evolution of the IRIS Task-Based Runtime to Enable Classical-Quantum Workflows, In preparation. (2025).

[38] J. Kim, S. Lee, B. Johnston, J. S. Vetter, IRIS: A portable runtime system exploiting multiple heterogeneous programming systems, in: 2021 IEEE High Performance Extreme Computing Conference (HPEC), IEEE, 2021, pp. 1–8.

[39] M. Schulz, M. Ruefenacht, D. Kranzlmüller, L. B. Schulz, Accelerating HPC With Quantum Computing: It Is a Software Challenge Too, Computing in Science & Engineering 24 (4) (2022) 60–64. `doi:10.1109/MCSE.2022.3221845`.

[40] Munich Quantum Valley, HPC-QC, `https://www.hpcqc.org/home`, accessed: 2025-02-24 (2025).

[41] European High-Performance Computing Joint Undertaking (JU), HPC-QS, `https://www.hpcqs.eu/`, accessed: 2025-02-24 (2025).

[42] IBM Newsroom, RIKEN Selects IBM's Next-Generation Quantum System to be Integrated with the Supercomputer Fugaku, `https://tinyurl.com/ibmriken`, accessed: 2025-02-24 (2024).

[43] IBM Newsroom, IBM and Pasqal Plan to Expand Quantum-Centric Supercomputing Initiative, `https://tinyurl.com/ibmpasqal`, accessed: 2025-02-24 (2024).

[44] IQM Newsroom, IQM Quantum Computers collaborates with Hewlett Packard Enterprise and demonstrates quantum-HPC integration at ISC 2024, `https://tinyurl.com/iqmhpenews`, accessed: 2025-02-24 (2024).

[45] Nvidia, CudaQ, `https://developer.nvidia.com/cuda-q`, accessed: 2025-02-24 (2025).

[46] Riverlane, Deltaflow: The Quantum Error Correction Stack, `https://www.riverlane.com/quantum-error-correction-stack`, accessed: 2025-02-24 (2025).

[47] N. Saurabh, S. Jha, A. Luckow, A Conceptual Architecture for a Quantum-HPC Middleware, in: 2023 IEEE International Conference on Quantum Software (QSW), IEEE Computer Society, Los Alamitos, CA, USA, 2023, pp. 116–127. `doi:10.1109/QSW59989.2023.00023`.
URL `https://doi.ieeecomputersociety.org/10.1109/QSW59989.2023.00023`

[48] Xanadu, Lightning Simulator, `https://www.xanadu.ai/products/lightning/`, accessed: 2025-02-05 (2025).

[49] E. Kaya, J. Echavarria, M. N. Farooqi, A. Swierkowska, P. Hopf, B. Mete, L. Burgholzer, R. Wille, L. Schulz, M. Schulz, A software platform to support disaggregated quantum accelerators, in: SC24-W: Workshops of the International Conference for High Performance Computing, Networking, Storage and Analysis, IEEE, 2024, pp. 1646–1653.