Platform Architecture for Tight Coupling of High-Performance Computing with Quantum Processors

SHANE A. CALDWELL, MOEIN KHAZRAEE, ELENA AGOSTINI, TOM LASSITER, COREY SIMPSON, OMRI KAHALON, MRUDULA KANURI, JIN-SUNG KIM, SAM STANWYCK, MUYUAN LI, JAN OLLE, CHRISTOPHER CHAMBERLAND, BEN HOWE, BRUNO SCHMITT, JUSTIN G. LIETZ, and ALEX MCCASKEY, NVIDIA Corporation, USA

JUN YE, Institute of High Performance Computing (IHPC), Agency for Science, Technology and Research (A*STAR), Singapore

ANG LI, Pacific Northwest National Laboratory, USA and University of Washington, USA

ALICIA B. MAGANN, COREY I. OSTROVE, KENNETH RUDINGER, ROBIN BLUME-KOHOUT, and KEVIN YOUNG, Quantum Performance Laboratory, Sandia National Laboratories, USA

NATHAN E. MILLER, Lincoln Laboratory, Massachusetts Institute of Technology, USA

YILUN XU and GANG HUANG, Lawrence Berkeley National Laboratory, USA

IRFAN SIDDIQI, University of California, Berkeley, USA and Lawrence Berkeley National Laboratory, USA

JOHN LANGE, Oak Ridge National Laboratory, USA and University of Pittsburgh, USA

CHRISTOPHER ZIMMER and TRAVIS HUMBLE, Oak Ridge National Laboratory, USA

We propose an architecture, called NVQLink, for connecting high-performance computing (HPC) resources to the control system of a quantum processing unit (QPU) to accelerate workloads necessary to the operation of the QPU. We aim to support every physical modality of QPU and every type of QPU system controller (QSC). The HPC resource is optimized for real-time (latency-bounded) processing on tasks with latency tolerances of tens of microseconds. The network connecting the HPC and QSC is implemented on commercially available Ethernet and can be adopted relatively easily by QPU and QSC builders, and we report a round-trip latency measurement of 3.96 µs (max) with prospects of further optimization. We describe an extension to the CUDA-Q programming model and runtime architecture to support real-time callbacks and data marshaling between the HPC and QSC. By doing so, NVQLink extends heterogeneous, kernel-based programming to the QSC, allowing the programmer to address CPU, GPU, and FPGA subsystems in the QSC, all in the same C++ program, avoiding the use of a performance-limiting HTTP interface. We provide a pattern for QSC builders to integrate with this architecture by making use of multi-level intermediate representation dialects and progressive lowering to encapsulate QSC code.

Additional Key Words and Phrases: Quantum Computing, High-performance computing, System Architecture

Authors' Contact Information: Shane A. Caldwell, scaldwell@nvidia.com; Moein Khazraee; Elena Agostini; Tom Lassiter; Corey Simpson; Omri Kahalon; Mrudula Kanuri; Jin-Sung Kim; Sam Stanwyck; Muyuan Li; Jan Olle; Christopher Chamberland; Ben Howe; Bruno Schmitt; Justin G. Lietz; Alex McCaskey, NVIDIA Corporation, Santa Clara, California, USA; Jun Ye, Institute of High Performance Computing (IHPC), Agency for Science, Technology and Research (A*STAR), Singapore, Singapore, Singapore; Ang Li, Pacific Northwest National Laboratory, Richland, Washington, USA and University of Washington, Seattle, Washington, USA; Alicia B. Magann; Corey I. Ostrove; Kenneth Rudinger; Robin Blume-Kohout; Kevin Young, Quantum Performance Laboratory, Sandia National Laboratories, Albuquerque, New Mexico, USA; Nathan E. Miller, Lincoln Laboratory, Massachusetts Institute of Technology, Lexington, Massachusetts, USA; Yilun Xu; Gang Huang, Lawrence Berkeley National Laboratory, Berkeley, California, USA; Irfan Siddiqi, University of California, Berkeley, Berkeley, California, USA and Lawrence Berkeley National Laboratory, Berkeley, California, USA; John Lange, Oak Ridge National Laboratory, Oak Ridge, Tennessee, USA and University of Pittsburgh, Pittsburgh, Pennsylvania, USA; Christopher Zimmer; Travis Humble, Oak Ridge National Laboratory, Oak Ridge, Tennessee, USA.

1 Introduction

Quantum computing is one of the most promising technologies of our era, and its invention is a great challenge in science and engineering. There is a worldwide, multi-decade research effort going on to meet that challenge. While there is much exciting progress in that effort, today's quantum processing units (QPUs) are somewhat wild objects belonging to an industry engaged in deep and rapidly evolving research. Many uncertainties must be resolved on our way to a robust and compelling example of useful quantum computation.

1.1 Two regimes of HPC integration

It is widely anticipated, though not a position of this paper, that unlocking the potential of quantum computing will require integrating the QPU within the traditional high-performance computing (HPC) environment of a supercomputer [1–4]. In this model, the QPU acts as a specialized node within the supercomputer. The QPU augments the supercomputer, and we may think of the new machine as a quantum-accelerated supercomputer. There may be many quantum nodes in such a machine, so as to implement circuit knitting or similar protocols to divide work among the nodes. The QPUs may also be linked to each other in a quantum network to distribute entanglement, in which case their behavior may be like a larger quantum computer with its own heterogeneous quantum resources [5]. The supercomputer is expected to be running its own compute-intensive part of the application, and it may be free to treat the QPU as a black box. If the QPU is operating with some combination of quantum error correction (QEC), quantum error mitigation (QEM), and quantum ensemble sampling, the data throughput and latency requirements between the supercomputer and the QPU may be very modest relative to typical node-to-node interconnects in the supercomputer.

We expect a second type of HPC integration will be required, where the HPC resource must be *tightly coupled* to the control system of the QPU to perform online workloads of the QPU itself. In this scenario, which is the subject of this paper, the HPC resource is probably of a smaller scale than a supercomputer and is optimized for real-time computation in the critical path of QPU operation, serving functionally as a part of the QPU. The best known of these workloads is QEC decoding, which in several QPU modalities requires the lowest possible latency while sustaining network throughput of $\sim 1\,\mathrm{Tb/s}$ and compute of up to $\sim 1\,\mathrm{PFLOP/s}$ [6]. We expect QPUs will also require continual recalibration to keep quantum operational fidelities within tolerances, and the computational demands of calibration procedures may rise as tolerances shrink, as longer-range errors have to be suppressed [7], and as machine learning is increasingly used to improve automation and accuracy. These tasks require specialized hardware and programming that are not present in the supercomputing environment, and they rightly belong to the QPU as conditions of its operation.

1.2 Requirements from QEC

Tight coupling is critical for QEC decoding. Consider a QPU running a program encoded with a stabilizer code. The code's stabilizer measurements produce a continuous stream of *syndrome* data emitted by the QSC at the QEC cycle rate, which is up to $\sim 1\,\mathrm{MHz}$ in systems planned today [8–15]. Each syndrome measurement is 1 bit per ancilla qubit per QEC cycle, though this would increase to perhaps 16 bits per ancilla in the case that continuously varying ("soft") readout data is used as input [16–19]. The QEC decoder must process all of the syndrome data to maintain a record of the current state of the QPU, and information from this record is accessed for feedforward operations at specific points in the program. A useful and common paradigm for determining the compiled structure of such a program on a topological code is found in *Pauli-based computation* [20] with *lattice surgery* [21] and compilation into a sequence of concurrent Pauli measurements with non-Clifford gates at each layer [22].

There are two timing requirements on the decoder performance:

(1) Decoder throughput, measured as syndrome cycles decoded per unit of time, must keep up with the continuous stream of syndrome data. At each feedforward event, the decoder's record will be delayed from the stabilizer rounds by some amount of real time and some number of stabilizer cycles which constitute a backlog for the decoder. If syndrome processing falls short of the streaming rate over one feedforward period, the backlog grows proportionally to that shortfall which leads to exponential growth of the shortfall and eventual stalling of the QPU [23].
(2) Reaction time, measured as time elapsed between the last syndrome measurement acquisition on the QSC to the application of a feedforward action derived from syndrome data up to that point, directly impacts both QPU clock speed and fidelity.

We emphasize that additional latency in the reaction time impacts the correctness of a quantum program's execution, even in a fault-tolerant quantum processor. Consider a fault-tolerant QPU that makes effective use of QEC to reduce its per-instruction error probability ε by many orders of magnitude below the error rates of its physical operations. A program on this QPU will be free of any logical error with probability $P = (1-\varepsilon)^N$, implying an exponential cutoff in success probability at a program of size $N_{\text{limit}} \sim 1/\varepsilon$. The same argument applies if we count only instructions which perform non-Clifford gates and require feed-forward events in the QEC protocol. While waiting for the feed-forward operation, the target logical qubit can run syndrome extraction cycles to remain coherent, and a key metric of interest in QEC is the logical error rate per cycle. This means that the logical error incurred by the reaction time can be estimated with the logical error rate per cycle along with the number of cycles spent idle. At these events, which are also used to estimate program runtimes (for example in [24]), the instruction error probability increases linearly with respect to the feed-forward reaction time of the QEC decoding system [25–27]. So we see that system latencies affect not only the rate of processing speed but the tradeoff between a program's execution time and the correctness of its execution.

Decoder throughput and reaction time are similar to what is seen in superscalar processors that contain multiple execution units in a single core. Due to parallel resources (execution units), the total number of instructions executed per second is not simply the inverse of the end-to-end execution time of a single instruction. In this analogy, the decoder throughput (syndrome extraction cycles decoded per second) is analogous to the number of instructions retired per second, and the reaction time is analogous to the end-to-end instruction execution time.

There is currently no experimentally demonstrated solution for QEC decoding at scale, ie. in systems that effectively operate $\sim$ 100 logical qubits or more in a fault tolerant quantum program containing $\sim$ 1 million logical instructions. There is very active ongoing research in QEC codes as well as algorithmic [28–41] and AI-based [18, 42–49] decoders. Today's state of the art in applied QEC comprises early demonstrations of partial functionality: single-qubit state preservation convincingly below threshold [9]; specialized demonstrations of quantum logic on a few logical qubits [11, 50–58]; and demonstrations of decoding on FPGA and CPU with sufficient throughput on small codes [9, 59, 60]. A number of analyses of requirements for the quantum system controller are available [61–63], and a viewpoint on the tradespace of specialized QEC decoding hardware (GPU, FPGA, ASIC) is available in Ref. [64]. Architectures based entirely in firmware (FPGA) or hardware (ASIC), where decoding graphs are physically instantiated in hardware, contemplate purpose-built networks for distributed computation at ultralow latency [8, 65].

To achieve fault tolerant quantum computing at scale, the throughput requirement on QEC decoding will necessitate heavy use of parallel processing. For example, a fault-tolerant algorithm using lattice surgery with topological codes requires operations with very large effective code distances. In addition to processing multiple syndrome measurement

rounds for all logical qubits, large code patches need to be processed per round in parallel over both space and time [27, 66]. By partitioning the lattice into blocks of spatial dimensions $O(d)$ (for a distance d code) and temporal dimensions $O(d_m)$ (where d_m is the number of syndrome measurement rounds), all such blocks can be decoded in parallel. In subsection 6.2 we discuss such requirements in more detail to achieve scalable real-time decoding algorithms. We also discuss the important role GPUs can play for block-wise decoding, especially when using AI-based pre-decoders [26, 49].

In contrast to topological codes, quantum low-density parity check (qLDPC) codes such as bivariate bicycle codes [67] offer lower qubit overhead. The modular decoding structure and parallelization strategies for topological codes cannot directly be ported to qLDPC decoding, but novel decoding strategies are actively being investigated [8, 36, 60].

There are a number of practical considerations that favor a heterogeneous architecture that admits accelerated computing hardware and a programming model that support rapid iteration.

(1) Within the execution of one fault-tolerant quantum program, it is necessary to dynamically refer to the relevant detector error matrix (DEM) [68] for the logical operation at hand. Logical gates require dynamic reconfigurations of their Tanner graph, and each *applied* logical gate (ie. logical gate as applied to particular logical qubits in the QPU) requires its own DEM values for decoding. Thus, strategies for handling dynamical changes to DEM structure and values, or DEM modularization, are likely necessary.
(2) QPUs are subject to physical drift, such that the noise model informing the DEM changes over time. The decoder configuration must be maintained continuously to preserve high logical fidelity.
(3) In a high distance code, it will generally be impossible to align the readout hardware with every DEM in the needed set of DEMs. Therefore a network and syndrome aggregation mechanism are required.
(4) The rate of research in the QEC space is such that many researchers and builders are likely to innovate on QEC codes and decoders and want to share solutions. An open platform supporting such sharing and shortened time to solution for new implementations will be valuable.

The total compute required for decoding at scale can be estimated from Figure 10 of Ref. [6], which finds that a 100-qubit QPU can run a depth-10^6 program encoded by a surface code if it has Fusion Blossom running at a compute intensity of 200 TFLOP/s. At 1 PFLOP/s, Fusion Blossom supports a depth-10^9 program on a 1000-qubit QPU. If we instead assume the use of an AI decoder, we would estimate needing 2 FLOP per logical qubit, per model parameter, and per inference. Assuming we use a 25-million parameter model (5× the size of the AlphaQubit model [18]) and decode at a 1 MHz QEC cycle, we would need 50 TFLOP/s per logical qubit to run the decoder. We may add a factor of ten to this to ensure we have headroom for dynamic compilation and realtime updating of QPU control parameters and the decoder noise model and conclude that 50 PFLOP/s provides a conservative estimate for the necessary compute intensity at the scale of 100 logical qubits.

1.3 Challenges

Introducing tightly coupled HPC to the QPU environment presents a number of challenges that have emerged as priorities within the quantum-HPC community, which has started seeking solutions that scale up with increasing capabilities of quantum computing systems [69]. Most QPU system controllers (QSCs) are implemented using field programmable gate array (FPGA) or radio-frequency system-on-chip (RFSoC) devices, which run a firmware-defined pulse processor unit (PPU) [70, 71]. (Please look ahead to Figure 1 for a system diagram identifying components discussed here.) There are dozens of PPU implementations in use in the quantum computing industry, most of which are closed-source and proprietary. Each PPU has its own features, instruction set, and compiler toolchain that lowers

from programs expressed in gate-level or pulse-level quantum instruction languages such as CUDA-Q, QASM, Cirq, Quil, and others, as well as lower-level languages that introduce control over pulse scheduling, waveform definitions, and other digital signal processing features common in modern software-defined radio technology. The whole QSC system integrates an array of PPUs with state-of-the-art synchronization of all PPU clocks (250 MHz typ.), microwave phase coherence, and often an ultra-low latency $\sim$ 300 ns network among the PPUs.

Many features and requirements of the QSC differ among the QPU modalities as well as the various implementations available from QSC vendors. To give several examples:

- Some modalities achieve all-to-all qubit connectivity within the QPU by dynamically shuttling trapped ions or atoms in space, either to interact with neighbors or to enter zones of the trap where specific control and readout functions are performed. Other modalities have fixed qubit topologies, and no dynamic routing constraints impact the PPU schedule.
- QPU modalities differ in the presence or absence of optical carrier waves for control and readout.
- QSCs have variable level of dynamism and programmability. Some QSCs support only static sequences or nonparameterized arbitrary waveforms, while others support fully dynamic pulse pipelines and real-time-patchable parameterized waveforms.
- QSCs have variable level of sophistication in compiler toolchain. The ability to mix gate-level and pulse-level programming paradigms, and to mingle runtime concerns such as QEC encoding, is still in a nascent stage and not standardized.
- QSCs have variable network topologies and sophistication of pulse scheduling functionality within the QSC. Some QSCs may support a global pulse scheduler that maintains resolution of every clock tick of every PPU, while others do not.
- Physical qubit state readout mechanisms are very different from one modality to another, which entails very different hardware scaling properties with respect to physical qubit counts and very different data reduction procedures that are typically encoded in PPU firmware.
- Some QSCs will place control and readout electronics into the physical QPU environment, including deep cryogenic environments with extreme power limitations [72–74].

1.4 Goals and outline

To summarize, our view is that a program defined on a QPU is part of a program defined on a supercomputer. The quantum portion of the program must be compiled down through quantum instructions and eventually to pulses scheduled across every PPU in the QSC, taking on sensitivity to the PPU architecture, the physical QPU modality and instance, and the current calibrated state of the physical QPU. Yet from within the PPU program running in the we must send data to a tightly coupled resource for QEC decoding. We want to be able to write everything from the parent application and the body of the decoding function in common programming languages such as Python, `C++`, and CUDA. We want the resulting program to be orchestrated on a distributed real-time system that does not rely on the use of web-oriented networking technology such as HTTP transport. And we want to do this while allowing every QSC builder to achieve this integration with minimal effort and changes to their toolchain.

By defining the NVQLink architecture, we aim to advance the integration of quantum processors into the supercomputing environment by the following means.

(1) Introducing an industrial grade, real-time (deterministic latency) interface between a QSC and HPC resources for online quantum error correction and all other compute-intensive online workloads of the QPU.
(2) Providing programming mechanisms and efficient data marshaling among many heterogeneous devices within a Logical QPU.
(3) Defining a platform that scales from current-generation QPU devices to future fault-tolerant systems.
(4) Enabling offline program development and validation through substitutable PPU emulation (VPPU) to reduce dependencies on physical hardware.
(5) Offering steps toward standardization in parts of the system architecture where innovation is not needed or desired.
(6) Doing the above in a way that enables all QPU and QSC builders to take advantage of the platform to advance their own successes.

In section 2 we describe the hardware architecture of a Logical QPU. In section 3 we introduce proposed CUDA-Q extensions for real-time device callbacks and heterogeneous memory. In section 4 we propose a new trait-based runtime, compiled-kernel format, executors, and a Logical QPU Driver API. In section 5 we suggest preliminary requirements in an effort to work in the open and gather feedback from interested readers. In section 6 we describe in greater detail the QPU-level workloads that we expect will benefit from tight coupling. In section 7 we discuss the use of development and simulation tool (PPU emulation, Physical QPU simulation, and QEC simulation).

2 System Architecture

The NVQLink system comprises the real-time execution environment in which an HPC system is tightly coupled to a QPU control system. The system is represented in Figure 1.

2.1 System components

The NVQLink architecture defines set of hardware components and their relationships that constitute a *machine model* for tightly coupled HPC and quantum programming. This machine model corresponds to a subset of the systems that can be defined in the CUDA-Q programming model, which is described in more detail in section 3.

- **Physical QPU** (PQPU) Quantum system whose physical state is described by a vector in a computational Hilbert space. This is the quantum object in the system and the key resource that enables quantum computation. We refer to it independently of its control and readout electronics. Examples include superconducting electrical circuits, spin qubits, trapped atoms and ions, flying photon arrays, etc.
- **Logical QPU** (LQPU) System capable of quantum computation, necessarily including the control and readout electronics.
- **Quantum System Controller** (QSC) System that implements quantum coherent control and readout on the Physical QPU. This typically contains an array of PPUs comprising all the analog I/O channels to and from the Physical QPU. We are abstracting over the choice of whether to group PPUs into smaller chassis or assemble them into a single-layer array.
- **Pulse Processing Unit** (PPU) Unit of control and readout electronics. Analog control unit containing one or more Pulse Processors, typically implemented on one or more FPGAs. The PPUs perform control and readout on physical qubits. Modern PPUs typically use Software Defined Radio (SDR) techniques relying on Numerically Controlled Oscillators (NCO) for microwave carrier frequencies. Figure 1 identifies four components of the PPU

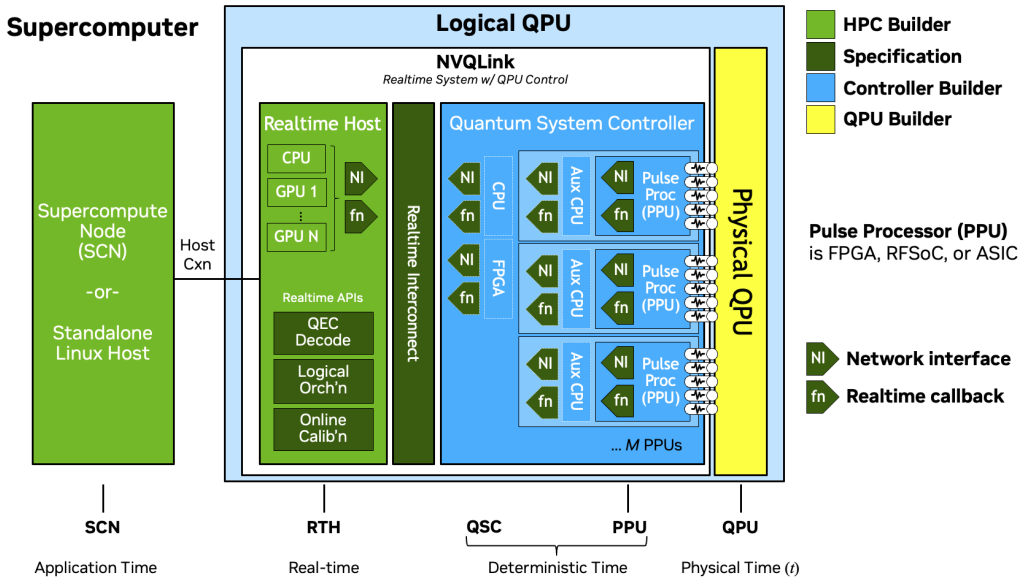


Fig. 1. Machine model of the Logical QPU. The NVQLink architecture comprises the Real-time Host (RTH) and QPU Control System (QSC) connected by a low-latency, scalable Real-time Interconnect joining them into a network capable of handling the runtime workloads of a Fault Tolerant Quantum Computer. The RTH contains traditional HPC compute resources (CPUs and GPUs and perhaps other specialized hardware), while the QSC contains the Pulse Processing Units controlling the QPU. These compute resources also comprise the key memory and storage resources for the application to consider and orchestrate including GPU memory, RTH system main memory, etc. The programming model for this system is built to recognize all CPUs, GPUs, and resources within the QSC including CPUs, PPUs, and other specialized FPGA resources, as targetable Devices and enable Real-time Callback functions (fn) among them to support distributed processing and data marshaling. To support this, a small and optional Network Interface (NI) is provided that enables unilateral and private adoption by the QSC builder. This construction affords flexibility in the value chain of Physical QPU, QSC, and runtime protocols such as QEC Decoding and Online Calibration: each of these components is provided by a third party who may build or integrate some or all of the system and who may share their implementation of each component or keep it proprietary at their discretion. The purpose of this architecture is support such flexibility while enabling every implementation to achieve state-of-the-art HPC performance at minimal cost and time to solution.

(1) Network Interface (NI)
(2) Pulse Processor (typically implement in FPGA or RFSoC firmware)
(3) Analog/Digital Converters (A/D)
(4) Analog Frontend, which comes with electrical isolation effective at microwave frequencies and has analog I/O ports in a format such as SMA.

This architecture attempts to be agnostic to all behaviors and implementation details of the PPU except its means of communicating with the Real-time Host.

- **Virtual Pulse Processing Unit** (VPPU) A PPU emulator that can be substituted for the actual PPU of the NVQLink system for offline development and system simulation. VPPU implementations must fully implement the instruction set of the target PPU and maintain compiler compatibility.
- **Real-time Host** (RTH) High-performance computing resource used for workloads in the Real-time Domain (see below). This will be a GPU node or multi-node cluster programmable by CUDA-Q, and its CPU a Host in terms of the CUDA Programming Model [75].

- **Real-time Interconnect** Network switch connecting the Real-time Host to the QSC in some way and perhaps to every PPU directly. The network architecture of NVQLink, described further in subsection 2.3, is optimized for low latency performance on a static network and supporting easy integration into proprietary firmware on the QSC.
- **Access Node** A traditional (non-real-time) compute node that serves remote clients, communicates with local and online storage, processes requests, etc. In a standalone configuration typical of usage by a QPU builder, it is usually a server running a common Linux distribution on a local network with the QSC. In a supercomputing integration it may be a node within the supercomputer. The only assumption we make on the Access Node is that it should connect to the Real-time Host with performance sufficient to run the application.

The primary hypothesis motivating the inclusion of the Real-time Host in the Logical QPU in the NVQLink model is the expectation that useful quantum computation will require tight coupling. We envision a future in which the Logical QPU is the *de facto* meaning of the term "QPU," and in CUDA-Q this is the meaning of the kernel attribute `__qpu__`. The Logical QPU is a heterogeneous device comprising other processors, some of which are also programmable using CUDA. The Logical QPU itself is programmable using CUDA-Q.

While work is ongoing to realize this vision, we expect the Access Node to continue to connect to the QSC by whatever means exist in prior integrations. This configuration must be left to the QPU and QSC builder who maintain these prior integrations. The Real-time Host can be introduced gradually in a time-sharing mode where its functions are brought online with mixed usage.

The QSC is subject to its own set of requirements which vary by Physical QPU modality, architecture, and version. This architecture is intentionally mute on the QSC's requirements, so that it can vary maximally in response to the needs of the Physical QPU.

2.2 Time domains

We identify and distinguish four time domains that are required in the system.

- **Physical Time Domain** (PTD) The time domain describing the Physical QPU in the lab frame, ie. the continuous variable t in the control Hamiltonian of the PQPU.
- **Deterministic Time Domain** (DTD) Time domain in which quantum-coherent control and readout operations are scheduled. Typically this domain is defined discretized by the synchronized clock of the FPGAs in the QSC.
- **Real-time Domain** (RTD) Time domain in which tasks are performed that either require data that cannot be localized within the QSC or require more compute intensity than the QSC can support, yet nevertheless require some contract on overall latency. The clearest requirements for this domain come from running online quantum error correction, but we anticipate that other valuable uses of the RTD will be found before fault tolerant quantum computing is fully realized.
- **Application Time Domain** (ATD) Time domain in which the application of the Logical QPU is executed. This is in all respects a traditional HPC domain, and the latency of operations affects application performance in ways that are no different from traditional HPC applications.

The DTD and RTD operate under real-time contraints due to the latency requirements with which they must produce and return results to the QSC. In order for the system to ensure correct operation these constraints can be specified using a set of inputs to the DTD and RTD that define the real-time requirements that the DTD and RTD must meet.

These constraints will enable the DTD and RTD to determine the satisfiability of the requested operations as well as to ensure that the system schedule will meet the requirements. As in other real-time systems we expect these requirments to be expressed either as individual deadlines for each requested operation, or a periodicity requirment that constrains the processing time for a recurring set of operations. These inputs to the DTD and RTD will allow the resources to internally manage the workload schedule and provide feedback to indicate a violation of the real-time requirements. Additionally, it will enable the DTD and RTD to independently adjust the quality of results to potentially provide a tradeoff between accuracy and timeliness, for instance an operator could return early with intermediate/sub-optimal results if its determined that fully computing the operation would result in a missed deadline.

2.3 Network architecture

Implementations of the Real-time Interconnect in an NVQLink system are open to third parties with their own requirements. Therefore we do not stipulate such requirements as a maximum acceptable latency, minimum throughput, or even a topology.

We instead provide open access to a highly performant and scalable reference implementation that aims to satisfy the following criteria.

(1) A standard protocol optimized for a static, point-to-point network.
(2) A freely available FPGA IP core implementing the protocol that QSC builders may integrate privately into their proprietary firmwares.
(3) Apart from FPGA IP core, reliance only on widely available networking equipment.
(4) Latency as low as possible, subject to the constraints above.
(5) Throughput and network radix well in excess of current QPU scales.

To achieve this goal, the number of components involved in the communication should be minimized, while scaling the system to a high number of PPUs is desired. Even though putting the QSC system as a Peripheral Component Interconnect Express (PCIe) card can enable direct communication between QSC and GPU, it encounters the scaling challenges of PCIe. Main challenge being limited number of PCIe slots to insert such cards in commodity equipment, as well as challenges of increasing the number of slots. Also from a development point of view, developing such cards for newer versions of PCIe bus, or maintaining the driver and software can be challenging.

The alternative is to use a network interface card (NIC) and connect the QSC to the HPC system through commodity Ethernet or InfiniBand physical links. The increased latency between QSC and NIC from packet generation and going across the transceivers and the physical link is in order of nanoseconds, with minimal jitter. Benefiting from the hardware acceleration within the NIC, the added latency to get to the PCIe bus is also low with low jitter. If desired, Ethernet or InfiniBand switches can be added, which also have accelerated hardware. This alternative enables easier scaling, both in terms of link speed to transfer more data per PPU, as well as use of network switches to scale to more PPUs and aggregating the data. Therefore, we proceeded with this approach, as the added latencies and jitter were acceptable, especially compared to typical higher latency and jitter considerations for the PCIe bus.

Note that the number of components in the processing of the packet is still important and should be minimized. Therefore, we used NIC offload features such as Remote Direct Memory Access (RDMA) to bypass the host processor and kernel, as well as DOCA GPUNetIO library [76] to enable direct packet handling from the GPU instead of the host processor. In other words, benefiting from these two technologies, only the NIC and GPU are involved during the processing of packets coming from and going to the QSC, without any host involvement. Also by using an HPC system

that has a dedicated PCIe switch between the NIC and GPU slots, the data does not traverse to the host processor socket and stays between the NIC and GPU with a single hop over the switch. This avoids contention from other components on the shared PCIe bus.

Another point to consider is when latency and jitter is the main objective, using a reliable connection becomes less desirable, as that reliability translates into packet retransmission in case of dropped packets due to errors such as checksum error. Such retransmissions usually happen after some timeout, and even if a specific system has features to notify the sender of such a drop, the added latency of this retransmission can throw off the timing of the consecutive packets, with minimal control from the software side. On the other hand, for this specific use case, the size of the data sent per transmission is really small, and using a certified Ethernet connection with low bit error rate (BER), such as NVIDIA's 100G Ethernet cables engineered for a BER of less than 10^{-15}, makes the probability of such drops minuscule.

If detection of such drops are desired, we can include a packet number in each packet and expose it to the software. If a higher reliability is desired, we can replicate the small packets, which achieves a more predictable latency and jitter. For this use case with periodic small packets, typically the bandwidth is not a bottleneck, and if the rate and size of data becomes a bottleneck, a higher speed Ethernet/InfiniBand link can be used. Therefore, we opted for an unreliable connection between the QSC and the NIC, and kept the flexibility of handling such rare occurrences in the hand of the user and software, if necessary.

2.4 Network proof of concept

To show a Proof of Concept (PoC), we built on top of NVIDIA Holoscan Sensor Bridge (HSB) [77] ecosystem, which provides means to send data between an FPGA and NIC using the RDMA over Converged Ethernet (RoCE) protocol, as well handling the enumeration steps and the control signals. We made a setup with an ARM system hosting a NVIDIA RTX PRO 6000 Blackwell GPU and a NVIDIA ConnectX-7 Network Interface Card (NIC), as well as an AMD RFSoC FPGA, shown in Fig. 2. We also use a separate laptop to connect to the Integrated Logic Analyzer (ILA) within the FPGA to read out the results, not to have any impact on the latency measurements. This setup is using off the shelf components, and thanks to utilizing the the mature software stack of RoCE, it can be ported to other GPUs, such as NVIDIA GB300, and this GPU was only chosen for the PoC purposes. Clearly, the network card and GPU performance, and their connection over PCIe will have impacts on the latency.

To measure the latency, we designed a system where FPGA creates packets, host loops them back, and FPGA measures the time difference. To achieve high accuracy time measurements, a Precision Time Protocol (PTP) time stamp generator was used, to produce nanosecond level time values. The 96-bit value of PTP alongside a 16-bit value of packet number, along 18 bytes of 0 form the 32-byte payload for the RoCE packets. Upon arrival of the looped-back packets, the current time stamp and the timestamp in the packet, alongside its packet number, are sent to a laptop through the ILA. On the laptop, packets are checked to be consecutive, and per packet the time difference is measured.

The main logical IP on the FPGA is the HSB IP, which is configured through our software and creates standard RoCE packets. This IP is available for several FPGA vendors, and follows the standard interfaces for the Ethernet MAC IPs. Moreover, on the receive side, it detaches the RoCE headers and delivers the payload to the FPGA. Additionally, this IP provides control signals for the FPGA, and we use them to set the time between packets, as well as start or stop the data stream from the FPGA. For a final design, the data stream can be changed from the packet number and time stamp to the proper data based on the application needs.

Fig. 4a shows the raw end-to-end latency captured by the ILA, and Fig. 4b shows the distribution of this data. This is for the beginning of the run, as there is a need for a short warm-up period, which incurs higher latencies. This warm-up

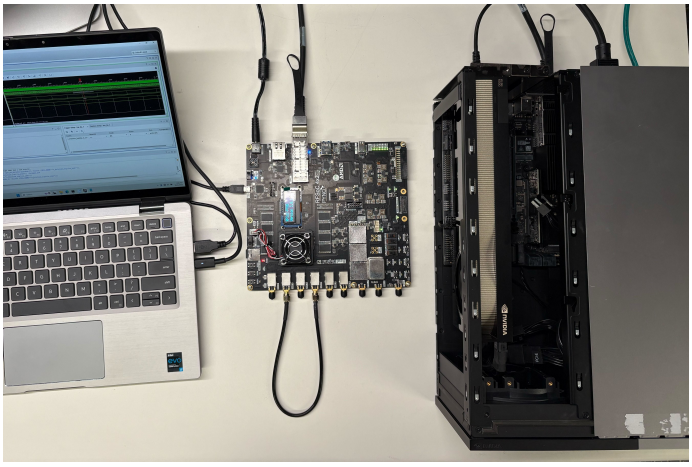


Fig. 2. Proof of Concept setup

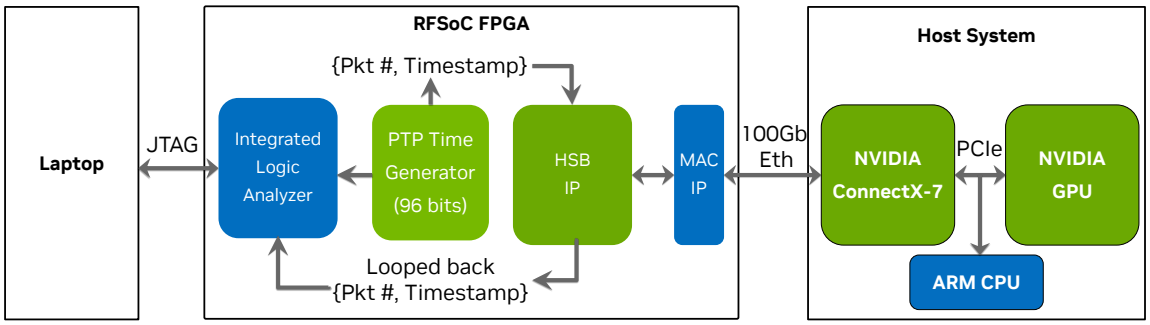
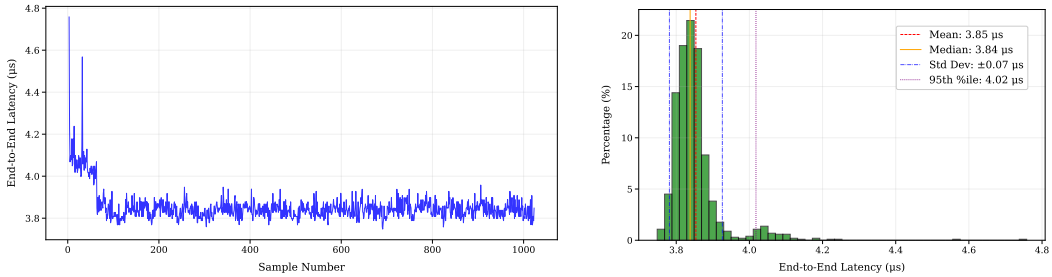


Fig. 3. Proof of Concept flow



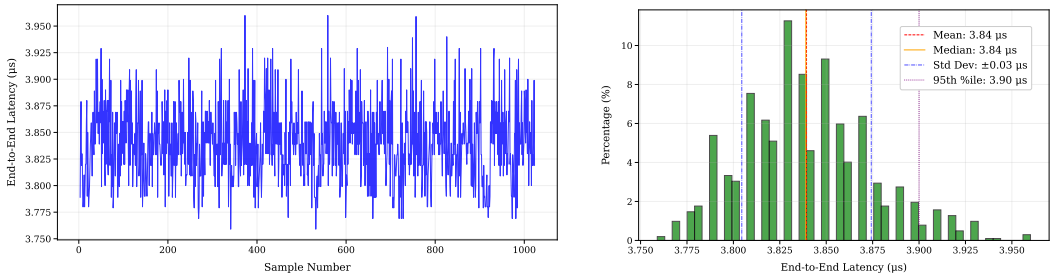
(a) End-to-end latency over time

(b) Histogram of end-to-end latency

Fig. 4. Observing slight increased warm-up latency at the beginning of a run

period is expected, due to factors such as some queue elements in the NIC being populated for the first time, or the cache allocation on NIC or GPU side. If desired, this can be removed through an initialization process with some dummy packets to cover this warm-up period. After that, the system becomes stable, and as shown in Fig. 5a and Fig. 5b for

an example run, where we observe very little variance and end-to-end latencies below 4 μs. The mean and median latencies are measured at 3.839 μs, with a standard deviation of 35 ns and a sample maximum of 3.96 μs.



(a) End-to-end latency over time

(b) Histogram of end-to-end latency

Fig. 5. Steady state latency during the same run as Fig. 4

The loop-back software achieving this latency is comprised of two main parts, a CPU side that enumerates the FPGA and initializes the connection between the HSB IP and the host, as well as sending the extra desired configurations to the FPGA such as the data generation rate, and finally launches a persistent kernel on the GPU. The other part of the software is this persistent GPU kernel which takes over the processing on the host side, and waits for packet arrival, and loops them back. Note that benefiting from the RDMA technology, the data arrives at the GPU memory directly from the NIC, and benefiting from the GPUNetIO library, GPU directly sends the control commands and output packet to the NIC, without the host processor or its memory getting involved in any of these interactions.

3 Programming Model

Programming NVQLink systems is enabled by extensions to the open-source CUDA-Q platform that expose the heterogeneous quantum-classical architecture through a unified programming interface. This programming model extends CUDA's proven heterogeneous computing paradigm to encompass quantum processing units, enabling developers to orchestrate computation across CPUs, GPUs, and quantum control systems within a single application framework.

The cornerstone of this approach is the quantum kernel — a C++ function annotated with `__qpu__` that seamlessly integrates quantum operations with classical control flow. These kernels are compiled by the `nvq++` compiler into Multi-Level Intermediate Representation (MLIR) dialects: Quake for quantum instructions and CC for classical operations. This intermediate representation enables sophisticated optimization and provides the foundation for real-time quantum-classical communication through device callbacks.

Definition 3.1 (Quantum Kernel). A C++ function annotated with the `__qpu__` attribute that encapsulates quantum operations within standard C++ control flow. Quantum kernels may contain quantum gates, measurements, and classical control logic, enabling the expression of complex quantum algorithms with familiar programming constructs.

Definition 3.2 (Device). Within the CUDA-Q machine model, a device represents any computational resource within the logical QPU capable of executing library functions and kernels. Devices include traditional processors (CPUs, GPUs), quantum control systems (PPUs, Real-time Hosts), and composite systems (supercomputing clusters). Each device is assigned a unique identifier and can be composed hierarchically from other devices.

3.1 Heterogeneous Machine Model

The CUDA-Q machine model for NVQLink extends the familiar CUDA host-device paradigm across three computational domains, as illustrated in Table 1. This hierarchical abstraction enables programmers to reason about quantum-classical systems using established parallel programming concepts while accommodating the unique requirements of quantum control systems.

Domain	Supercomputer	Real-time Host	QSC
Programming Model	CUDA-Q	CUDA	CUDA-Q
Host	Access Node CPU	Real-time Host CPU	Real-time Host CPU[†]
Device	Logical QPU	GPU	PPU or VPPU (emulator)
Kernel Type	Quantum kernel: `__qpu__`	CUDA kernel: `__global__`	Device callback: `cudaq::device_call`
Compiler	`nvq++`	`nvcc`	QSC-specific

Table 1. Programming model hierarchy spanning supercomputing, real-time control, and quantum control domains. [†]PPUs are addressable by the Real-time Host; QSCs may include dedicated control processors.

This three-tier architecture provides natural abstraction boundaries: the supercomputer manages high-level algorithm orchestration, the Real-time Host coordinates real-time data management and error correction, and the QSC executes fine-grained quantum control. Device unique identifiers (UIDs) enable direct addressing across all tiers, while the Real-time Host maintains a registry of available devices and their capabilities for efficient computation routing.

As indicated in Table 1, the QSC domain supports both physical PPUs and Virtual Pulse Processing Units (VPPUs) as device types. The VPPU provides a substitutable PPU emulator for offline development and testing (subsection 7.1). From the programmer's perspective, quantum kernels compiled by `nvq++` produce identical results when executed on VPPU or physical PPU, as both implement the same `quantum_control_trait` interface (subsection 4.1). This substitutability enables developers to iterate on quantum programs and real-time protocols without requiring continuous physical hardware access, accelerating development workflows before deployment to production systems.

3.2 Real-time Device Callbacks

Central to enabling tight quantum-classical integration for CUDA-Q programmers is the `cudaq::device_call` mechanism — a templated function intrinsic that enables quantum kernels to invoke computations on classical processing resources with automated data marshaling and minimal latency. This capability is essential for implementing real-time quantum error correction, adaptive algorithms, and measurement-based quantum computing protocols. Listing 1 demonstrates what `device_call` usage may look like to a CUDA-Q programmer.

The `device_call` intrinsic supports multiple invocation patterns optimized for different computational scenarios. These templates signatures provide several key capabilities:

- **Device Selection**: Explicit targeting of computational resources by unique identifier
- **Execution Flexibility**: Support for both CPU functions and GPU kernel launches
- **Automatic Marshaling**: Compiler-managed data movement between heterogeneous devices
- **Type Safety**: Template-based interface preserving C++ type semantics and enabling compile-time error checking

Function signatures for device callbacks must conform to CUDA-Q type constraints. Arguments passed to device callbacks are considered immutable. Argument types must therefore be pass by value or based by constant reference

Listing 1. Real-time quantum error correction using device callbacks.

```
__qpu__ void adaptive_qec_kernel(cudaq::qvector<>& data_qubits,
                                 cudaq::qvector<>& ancilla_qubits,
                                 int cycles) {
  for(int = 0; i < cycles; ++i){
    // Stabilizer circuits here
    ...
    // Execute syndrome extraction measurements
    auto syndrome = mz(ancilla_qubits);

    // Real-time streaming to dedicated GPU
    cudaq::device_call(/*gpu_id=*/1,
                       surface_code_enqueue,
                       syndrome);
    // Repeat
  }

  // Real-time decode on dedicated GPU
  auto correction = cudaq::device_call(/*gpu_id=*/1,
                                       surface_code_decode);

  // Apply corrections physically if desired (typically tracked in software)
  if (correction.x_errors.any())
    apply_pauli_x_corrections(data_qubits, correction.x_errors);
  if (correction.z_errors.any())
    apply_pauli_z_corrections(data_qubits, correction.z_errors);
}
```

Listing 2. Device callback template signatures.

```
// CPU function call on specified device
template <typename Callable, typename... Args>
auto device_call(std::size_t device_id, Callable&& func, Args&&... args)
    -> decltype(func(std::forward<Args>(args)...));

// CPU function call on default device
template <typename Callable, typename... Args>
auto device_call(Callable&& func, Args&&... args)
    -> decltype(func(std::forward<Args>(args)...));

// CUDA kernel launch with compile-time grid specification
template <std::size_t NumBlocks, std::size_t NumThreadsPerBlock,
          typename Kernel, typename... Args>
auto device_call(std::size_t device_id, Kernel&& kernel, Args&&... args)
    -> decltype(kernel(std::forward<Args>(args)...));
```

(`const T&`). Device functions must complete execution before quantum operations can resume, ensuring deterministic program semantics. Device callback library develpers are free to launch asynchronous threads within callback functions that return `void`.

3.3 Heterogeneous Memory Abstraction Layer

Given the heterogeneous logical QPU architecture in 1, the programming model requires a mechanism for reasoning about data allocated across inherent classical devices. Therefore, the programming model abstracts heterogeneous memory systems through the `cudaq::device_ptr<T>` type, which encapsulates device-specific memory handles while maintaining type safety and automatic lifetime management. This abstraction enables transparent data movement optimized for the underlying hardware topology. To meet these requirements, we specify a `device_ptr` type in Listing 3.

Listing 3. Device pointer abstraction for heterogeneous memory management.

```
template <typename T>
struct device_ptr {
  std::size_t handle = std::numeric_limits<std::size_t>::max();
  std::size_t size = 0;
  std::size_t device_id = std::numeric_limits<std::size_t>::max();
  void* host_shadow = nullptr;

  device_ptr(T* host_data) : host_shadow(host_data) {
    handle = register_device_memory(host_data, sizeof(T));
    size = sizeof(T);
  }
};
```

The memory model supports both explicit control for performance-critical applications and implicit management for programmer productivity. Compiler implementations should analyze data usage patterns to optimize transfers, employing high-bandwidth RDMA for capable devices and efficient batching for network-attached resources. Memory coherency is maintained through a combination of compiler analysis and runtime synchronization, ensuring program correctness across the heterogeneous system.

`device_ptr<T>` types provide a unique handle to allocated logical QPU device data, in a similar way that CUDA exposes device pointers for handling data on GPU device. These instances should be usable within CUDA-Q kernel code (i.e. quantum kernel signatures may contain `device_ptr` types), enabling sophisticated quantum-classical workflows that mutate data on classical devices within a logical QPU via the `device_call` intrinsic.

This device data modeling also requires that NVQLink expose an API for creation, manipulation, and destruction of these device pointer types. In the same way that CUDA provides a device driver API (e.g. `cudaMalloc`, `cudaMemcpy`, `cudaFree`, `cuLaunchKernel`, etc.), we require a logical QPU device driver API, which we elaborate on in Section 4.4.

3.4 Compilation Implications

The NVQLink concepts proposed here for the CUDA-Q programming model have direct ramifications on compiler implementations. Key to this is how one handles lowering the proposed `device_call` intrinsic. It is expected that quantum operations naturally lower to pulse level representations, followed by lowering to subsequent operations that drive the dynamics of the quantum register via a distributed set of FPGAs or equivalent System on Chip (SoC) instances. Note we are intentionally generic in this last statement due to the potential spectrum quantum execution timing domains one may encounter. Pulse representations may lower directly to FPGA softcore-processor Instruction Set Architecture (ISA) code, or it may lower to standard CPU code that mediates pulse emission. Moreover, it is unclear how `device_call` should lower to quantum control systems, and how modality-specific timing constraints should influence that code generation. Here we try to elucidate some of these finer points.

For the task of quantum kernel compilation, we identify two end points of a system latency sensitivity spectrum that is dependent on the underlying system latency tolerance and associated control requirements - we classify this as high latency sensitivity vs low latency sensitivity. Systems with high latency sensitivity are those that exhibit shorter qubit coherence times, thereby requiring lower real-time feedback latency. Systems with low latency sensitivity exhibit longer qubit coherence times, and can therefore tolerate slower real-time feedback. Each domain imposes different constraints on the communication patterns between the Real-time Host and the Quantum System Controller (QSC), fundamentally altering the kernel code compilation strategy.

3.4.1 High Latency Sensitivity. Quantum systems with stringent real-time requirements operate with extremely tight latency budgets that impose the most restrictive timing constraints on the control system, therefore we enumerate the following requirements for kernel compiler implementations:

- **No Real-time Host Mediation**: The control driver cannot mediate any data marshaling or function invocation during quantum operations due to latency constraints that would exceed decoherence timescales.
- **Remote Direct Memory Access (DMA) Required**: All real-time classical processing must occur through Remote Direct Memory Access (RDMA) paths that bypass traditional network stacks and CPU involvement, enabling sub-microsecond data transfer.
- **Pre-Compiled ISA Programs**: Complete ISA programs must be uploaded to FPGAs in advance and triggered atomically, with minimal interactive communication with the Real-time Host during execution. Control must be inherent to this pre-compiled ISA representation. We allow for dynamic instruction queuing, whereby FPGAs consuming ISA instructions pull the next instruction from a queue that is updated in real-time by the Real-time Host. The requirement then is that the instruction queue remains non-empty until program termination.

The compilation model for high latency sensitivity systems must perform aggressive ahead-of-time optimization and leverage pre-initialized, asynchronous data processing threads (e.g. CUDA persistent kernels) for real-time callback functionality. Figure 6 demonstrates this asynchronous workflow, and it is envisioned that compiler implementations will lower to this type of workflow for latency-critical modalities.

3.4.2 Low Latency Sensitivity. Quantum systems with more relaxed timing requirements exhibit higher tolerance for processing delays, enabling more flexible control architectures. Here we enumerate the requirements for kernel code lowering for this timing domain:

- **Real-time Host Mediation Permitted**: The Real-time Host can mediate data marshaling and function invocation between quantum operations without violating coherence constraints.
- **RDMA Beneficial but Optional**: While RDMA provides performance benefits, standard network communication paths remain viable for many applications.
- **Runtime Operation Streaming**: The Real-time Host can stream operations at runtime.
- **Interactive Execution Model**: Quantum and classical operations can be interleaved with bidirectional communication between the Real-time Host and QSC.

The compilation model for low latency sensitivity systems clearly supports just-in-time (JIT) compilation, optimization, and runtime adaptation. Figure 7 demonstrates the envisioned execution workflow for these systems. It is possible for compiler implementations to lower to Real-time Host library calls that mediate and drive kernel execution, including real-time data marshaling and device callback invocation.

3.4.3 Lowering Workflows. It will be helpful to elucidate what compiler lowering may look like in a typical implementation of an NVQLink implementation. Figure 8 demonstrates the envisioned compiler code generation workflow for an NVQLink architecture. Here one can see the quantum IR nodes (e.g. `quake.h`, etc.) as well as the critical `cc.device_call` operation. It is expected that quantum operations are subsequently lowered to pulse level representations (e.g. Pulse MLIR Dialects) in tandem with dataflow into and out of this `device_call` operation. The next phase of compilation is dictated by the inherent system timing domain exposed. Long coherence time domain systems may subsequently lower the code in Figure 8 (top right) directly to a library of QSC-specific intrinsics on the Real-time Host. It is envisioned that the `device_call` operation here will be lowered to a specific library function that takes as input callback arguments,

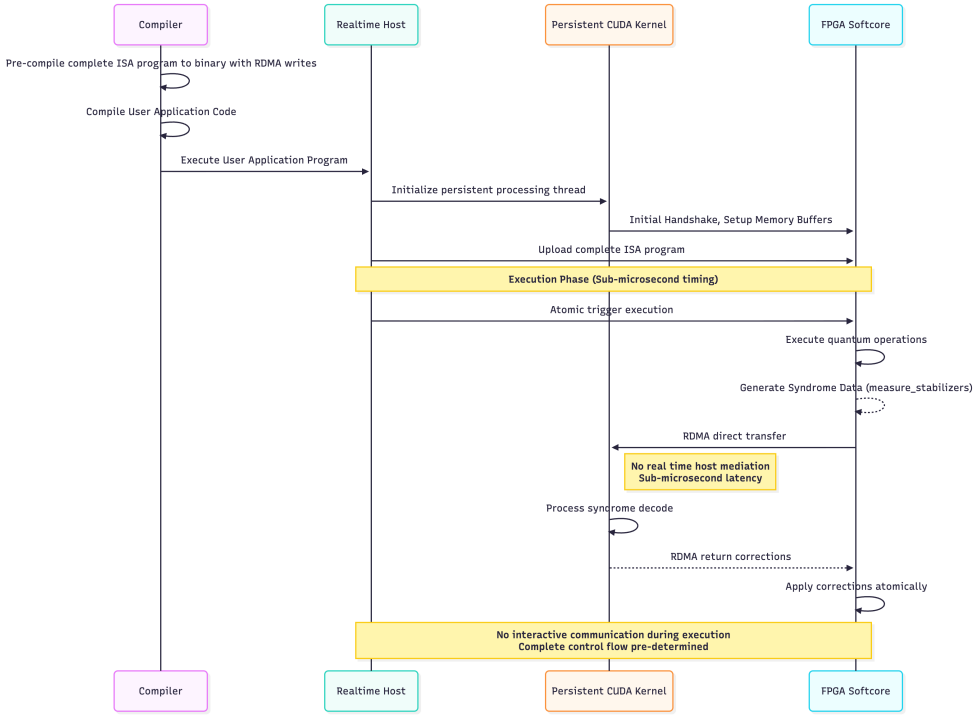


Fig. 6. Sequence diagram describing the interaction between Real-time Host, CUDA Device, and FPGA control to enable realtime data marshaling and classical device callback invocation for systems with short coherence time requirements. Quantum kernels must be compiled ahead-of-time to FPGA-control-specific ISA binary code. The Real-time Host must initialize the total system - launching persistent CUDA threads that wait for FPGA-delivered data for processing.

the callback name, and a callback return value pointer and leverages the NVQLink runtime API (see 4) to generically affect callback invocation on a designated device within the Logical QPU.

Latency-critical systems will require further lowering of pulse representations down to FPGA ISA code, with the understanding that this is likely a distributed FPGA system, with specific physical qubits assigned to specific FPGAs. Hence, the original unified kernel will need to be broken up into unique programs targeting specific FPGAs. Moreover, implementations for this timing domain will require lowering of `device_call` to FPGA operations that trigger direct data marshaling from FPGA memory to GPU memory (RDMA). GPU device code will need to be listening for data events asynchronously and apply requested callbacks once all data is received from executing FPGA ISA code.

These compilation workflow concepts will span available NVQLink architectural implementations due to the enhancements we propose to the CUDA-Q programming model. Next, we turn our attention to necessary runtime support for describing these compositional architectures and data types required for real-time classical coprocessing during critical system timing windows.

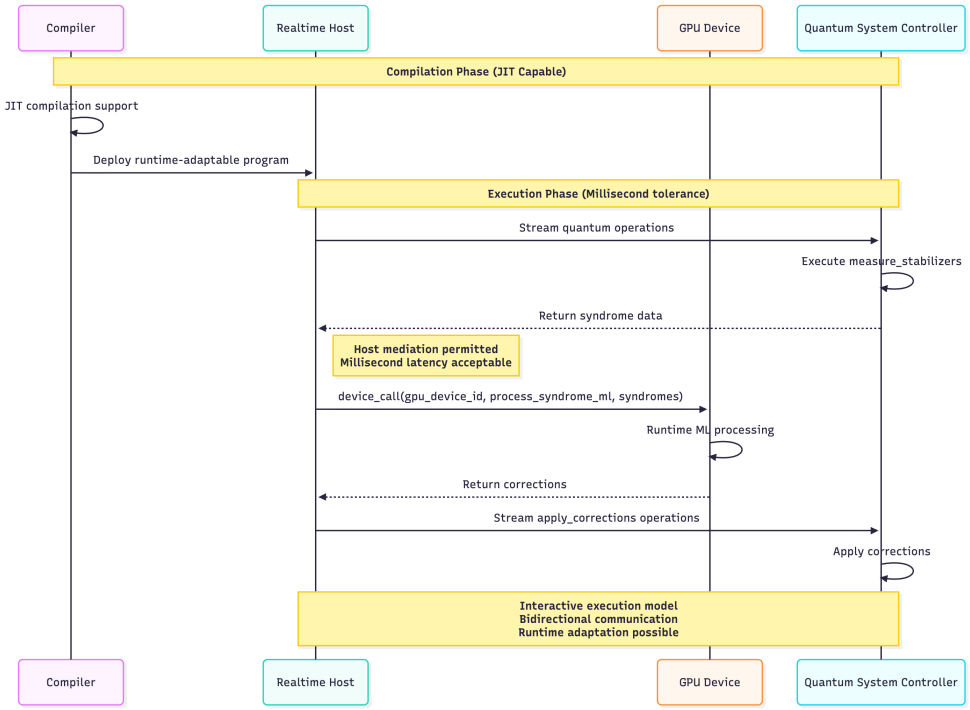


Fig. 7. Sequence diagram describing the interaction between Real-time Host, CUDA Device, and FPGA control to enable realtime data marshaling and classical device callback invocation for systems with long coherence time requirements. Here there is more flexibility for the Real-time Host to mediate interactions between FPGA control and GPU coprocessing. Quantum kernels can be compiled ahead-of-time or just-in-time, and pulse level instructions can be dynamically generated and emitted. Qubit measurement feedback can be mediated via the Real-time Host and marshaled manually to available classical processing devices.

4 Runtime Architecture

The NVQLink runtime architecture promotes a high performance, zero-overhead abstraction model built on *static polymorphism* and *trait-based composition*. These choices are made to serve the goals of minimizing real-time callback latency and maximizing configurability and extensibility of device types within the Logical QPU.

In this section, we fully specify required class types that enable expression of real-time data marshaling, data management, device callback processing for our envisioned logical QPU. Ultimately we propose interfaces for defining *devices* and their unique behaviours or capabilities, as well as a general device driver API for the logical QPU.

4.1 Devices and their traits

To start, NVQLink proposes a generic `cudaq::device` type, shown in Listing 4. This type serves as the fundamental abstraction for all processing resources within the Logical QPU. Each device retains its own UID and a registry of its functions that can be invoked under a real-time callback during quantum kernel execution. Each device receives its UID from an atomic counter during system initialization to ensure uniqueness. Devices are intended to maintain a mapping from library locations to callable functions, enabling dynamic function resolution. Devices expose `connect()` and `disconnect()` methods to enable one-time initialization and finalization of every device instance.

```cpp
// User code for a quantum kernel that
// requires realtime device callback (add).

// Declare device function
int add(int, int);

// Define CUDA-Q quantum kernel
__qpu__ int maybe_increment(int i) {
  cudaq::qubit q;
  h(q);
  return cudaq::device_call(2, add, i, mz(q));
}
```

```
func.func @maybe_increment(%arg0: i32) -> i32 {
  %0 = quake.null_wire
  %1 = quake.h %0 : (!quake.wire) -> !quake.wire
  %measOut, %wires = quake.mz %1 : (!quake.wire) -> (!
     quake.measure, !quake.wire)
  %2 = quake.discriminate %measOut : (!quake.measure) ->
      i1
  %3 = cc.cast unsigned %2 : (i1) -> i32
  %4 = cc.device_call @add on 2 (%arg0, %3) : (i32, i32)
      -> i32
  return %4 : i32
}
```

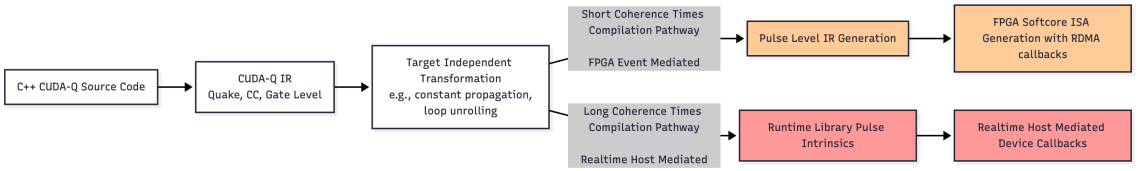


Fig. 8. (top left) Prototypical CUDA-Q quantum kernel code demonstrating quantum code interleaved with realtime classical processing on an NVQLink device. This code executes on a Logical QPU and compiler code generation depends on the timing domain inherent to the targeted architecture. Compiler implementations, as a first step, should lower to a unified representation that is amenable for further target-specific lowering (top right). Below these snippets (bottom), we see how compiler workflows should approach the proposed system timing domains.

Listing 4. Base class `cudaq::device` for devices managed by the NVQLink runtime.

```cpp
template<typename Derived, typename... Traits>
class device {
private:
  std::size_t device_id = 0;
public:
  void connect();
  void disconnect();
  std::size_t get_id() const;
};
```

The runtime expresses device functionality through a trait system that composes device-specific capabilities at compile time. Each trait is a set of behaviors that can be implemented on an object and is expressed by a class whose members are all public methods. We propose here a set of traits to be built into CUDA-Q, and this system supports straightforward extensibility to new traits defined by third parties.

We enumerate the following traits:

- **`explicit_data_marshalling_trait`**: Provides realtime host mediated memory management with methods for allocation, deallocation, and data transfer (see 5). This trait enables real-time data processing for slow timing modalities. It exposes the familiar API for data allocation, deallocation, and mutation.
- **`device_callback_trait`**: Enables realtime host mediated device function invocation (see 6). This trait enables real-time device callback capabilities for slow timing modalities. It exposes an API for applying a specified callback function (device-specific) on provided function arguments represented as pre-allocated `device_ptr` instances. It also allows for function result return via a separate pre-allocated `device_ptr` instance.

- **`quantum_control_trait`**: Specializes quantum control system functionality including program upload and trigger mechanisms. Devices that inherit this trait present a unified view of QCS devices for the realtime host (see 7). Programs are represented as compiled kernel binaries. Execution is triggered with kernel arguments provided as `device_ptr` instances. Kernel results are returned via a pre-allocated `device_ptr` instance.
- **`rdma_trait`**: Supports high-performance, low-latency data transfer through remote direct memory access (see 8). This trait is designed for fast architecture modalities where real-time host mediation of data marshaling and callback invocation is not possible. It exposes a minimal interface allowing one to extract memory buffer data for one-time initialization and handshake mechanisms with designated `quantum_control_trait` device instances. Concrete `rdma_trait` types are required to perform all pertinent RDMA connection tasks within the sub-type specific `connect` method implementation.

Listing 5. Trait supporting data movement controlled by the Real-time Host.

```
template <typename Derived>
class explicit_data_marshaling_trait {
public:
  void *resolve_pointer(device_ptr &devPtr);

  device_ptr malloc(std::size_t size) const;

  template <typename... Sizes,
            std::enable_if_t<(std::conjunction_v<std::is_integral<Sizes>...>),
                             int> = 0>
  auto malloc(Sizes... szs) {
    return std::make_tuple(static_cast<Derived *>(this)->malloc(szs)...);
  }

  void free(device_ptr &d);

  template <typename... Ptrs,
            typename = std::enable_if_t<
                (std::conjunction_v<std::is_same<
                     std::remove_cv_t<std::remove_reference_t<Ptrs>>,
                     device_ptr>...>)>>
  void free(Ptrs &&...d) {
    (free(d), ...);
  }

  void send(device_ptr &dest, const void *src);
  void recv(void *dest, const device_ptr &src);
};
```

Listing 6. Trait supporting real-time device callback functions.

```
template <typename Derived>
class device_callback_trait {
public:
  void launch_callback(const std::string &callbackName,
                       const std::vector<device_ptr> &args);
  void launch_callback(const std::string &callbackName, device_ptr &result,
                       const std::vector<device_ptr> &args);
};
```

This device architecture is designed to support various timing modalities and real-time data processing capabilities. Instantiation of concrete device types allows users to define unique logical QPU architectures programmatically. Their

Listing 7. Trait supporting program binary upload and execution on QCS devices.

```
template <typename Derived>
class quantum_control_trait {
public:
  void upload_program(const std::vector<std::byte> program_data);
  void trigger(device_ptr &result, const std::vector<device_ptr> &args, const cudaqx::heterogeneous_map& config);
};
```

Listing 8. Trait supporting data movement by remote direct memory access (RDMA).

```
template <typename Derived, typename RDMADataType>
class rdma_trait {
public:
  RDMADataType& get_rdma_connection_data() const;
};
```

use alongside architecture-specific quantum kernel compilation allows the development of real-time quantum-classical workflows that leverage GPU coprocessing natively.

4.2 Compiled Quantum Kernels

The NVQLink architecture considers general quantum kernel compilation as the means for mapping user-intent to executable instructions on pulse processors. We do not limit ourselves to static circuit compilation, but instead consider fully parameterized functional representations that leverage general classical control flow. Moreover, we allow for control flow that is non-deterministic (e.g. dictated by quantum measurement results). We take a CUDA-Q kernel centric approach, but seek to enable 3rd party compilation and binary loading as well, as long as 3rd party kernels support our proposed `device_call` capability.

We promote an ahead-of-time compilation strategy and rely on standard code linking techniques to construct hybrid quantum-classical executables and libraries. As such, the NVQLink architecture proposes an extension point for the library that enables compiled kernel generation. We represent compiled kernel code as in 9.

Listing 9. The NVQLink representation for compiled quantum kernels.

```
struct qcontrol_program {
  std::vector<std::byte> binary;
  std::size_t qdevice_id;
};
class compiled_kernel {
protected:
  std::string kernel_name;
  std::vector<qcontrol_program> control_programs;
public:
  const std::string &name() const;
  const std::vector<qcontrol_program> &get_programs() const;
};
```

We represent the compiled quantum kernel as a singular unit, but composed of potentially many individual binary programs (the `qcontrol_program`). Logical QPUs are expected to be composed of many `quantum_control_trait` devices (e.g. many pulse processors) and this design allows for the expression of binary programs for each constituent control device. Each program tracks that it is intended to run on. Quantum kernels in CUDA-Q are assigned a kernel name via its function name and signature. We retain this information and tag it to the `compiled_kernel` instance.

Listing 10. The NVQLink representation for architecture-specific compilers.

```
struct qcontrol_program {
  std::vector<std::byte> binary;
  std::size_t qdevice_id;
};
class compiled_kernel {
protected:
  std::string kernel_name;
  std::vector<qcontrol_program> control_programs;
public:
  const std::string &name() const;
  const std::vector<qcontrol_program> &get_programs() const;
};
```

To generate compiled kernel instances, NVQLink proposes a `compiler` interface that is meant for architecture-specific compilation extensibility. The class is defined as defined in 10

This API enables architecture-specific pre-compiled code loading (e.g. from object files) as well as JIT code compilation. Each capability takes as input the architecture-defined `quantum_control_trait` devices so that the compiled kernel is fully architecture-aware.

4.3 Holistic Kernel Execution

The NVQLink architecture promotes a hardware-aware capability for triggering kernel launches on available and specified `quantum_control_trait` devices.

Listing 11. The NVQLink representation for architecture-specific kernel executors.

```
template <typename Derived>
class executor {
public:
  void
  trigger_execution(std::unordered_map<std::size_t, any_device> &qcs_devices,
                    device_ptr &res, const std::vector<device_ptr> &args);
};
```

The intention for kernel launching is that the NVQLink implementation will upload all required programs to specified `quantum_control_trait` devices. Once all binary programs have been uploaded, an architecture-specific `executor` will be used to trigger the synchronous execution of all binary pulse programs.

The `executor` provides a singular view on the collective pulse processing units, effectively modeling a single virtual pulse processor. This type may be leveraged for more than just hardware-specific program triggering. As an effective adaptor on quantum code execution, one could leverage this extension point to provide real-time instruction streaming and scheduling.

4.4 Logical QPU Driver API

To this point, we have proposed an architecture for constructing Logical QPU instances that enable real-time (within coherence times) data marshaling and processing. For envisioned advanced use cases, it will prove beneficial to define a library API on top of the NVQLink proposed types enabling efficient Logical QPU data management, mutation, and kernel and device callback invocation. To this end, we propose an NVQLink Driver API exposing a familiar interface for coprocessor data allocation, deallocation, mutation, and kernel loading and launching.

Listing 12. The NVQLink Logical QPU Driver API.

```
template <typename... DeviceTypes>
void initialize(DeviceTypes &&...in_devices);
void shutdown();

std::size_t get_num_qcs_devices();
std::size_t get_num_classical_devices();

device_ptr malloc(std::size_t size);
device_ptr malloc(std::size_t size, std::size_t devId);
template <typename T>
device_ptr malloc(std::optional<std::size_t> device_it = std::nullopt);

void free(device_ptr &d);
template <
    typename... Ptrs,
    typename = std::enable_if_t<
        (std::conjunction_v<std::is_same<
             std::remove_cv_t<std::remove_reference_t<Ptrs>>, device_ptr>...>)>>
void free(Ptrs &&...d) {
  (free(d), ...);
}

void memcpy_to_qpu(device_ptr &arg, const void *src);
void memcpy_from_qpu(void *dest, const device_ptr &src);
template <typename T>
T memcpy_from_qpu(const device_ptr &src) ;

handle load_kernel(const std::string &location,
                   const std::string &kernel_name);
handle load_kernel_from_code(const std::string &code,
                             const std::string &kernel_name);

void launch_kernel(handle kernelHandle, device_ptr &result,
                   const std::vector<device_ptr> &args);
void launch_kernel(handle kernelHandle, const std::vector<device_ptr> &args);
template <typename Ret, typename... Args>
Ret launch_kernel(handle kernelHandle, Args &&...args);
```

NVQLink proposes the function API described in 12. The library specification starts with general initialization and shutdown functions. Initialization takes as input the user-specified concrete devices that compose the Logical QPU.

The library proposes the familiar user-facing set of data allocation and deallocation functions (`malloc` and `free`). These functions allow one to specify a target device with the Logical QPU where the data resides. If one does not specify a concrete device ID, the real-time host memory space is targetd. These functions return and operate on the aforementioned `device_ptr` instances. The library promotes a device-specific data mutation API (`memcpy`) allowing one to move data to and from the Logical QPU.

The API exposes a mechanism for loading and launching quantum kernels. The library should be configurable in this regard, allowing one to specify architecture-specific compiler and executor instances. Loading a pre-compiled kernel from file, or JIT compiling from source string, delegate to the specified compiler implementation and return a handle to the compiled kernel. This handle is used for subsequent calls to `launch_kernel`. Kernel launching takes as input the kernel arguments, pre-allocated on the real-time host via this API. For kernels that return a value, one can supply an appropriately sized `device_ptr` to `launch_kernel`.

5 Preliminary Specification

We present here a prospective set of requirements here in an effort to telegraph future work toward a hardened specification and inspire readers to provide feedback and input on that specification.

(1) The firmware definition of the PPU MUST remain reprogrammable by the QSC builder in the field.
(2) The NVQLink system MUST support an operating mode in which PPU instructions are opaque to the Real-time Host. In particular, the operator must have the option of encrypting these instructions without losing functionality in the Real-time Domain.
(3) The Network Interface MUST be guaranteed to interoperate with the Real-time Host and Interconnect, and the performance characteristics of the network SHOULD be publicly documented and readily measurable by end users.
(4) The Real-time Host MUST be programmable by `C++`, CUDA, and CUDA-Q.
(5) The Real-time Host MAY include other processing resources, including CPU-GPU systems-on-chip (SoCs), FPGAs, or ASICs.
(6) The QSC provider MAY provide a Virtual Pulse Processing Unit (VPPU) that emulates its PPU instruction set for offline development and testing (subsection 7.1).
(7) Round trip data latency from QSC to Real-time Host and back MUST be measurable by a CUDA-Q library function, and the results of this function invocation MUST return latency of every measured sample over its set of input (to be determined).

6 QPU Level Workloads

This section translates NVQLink abstractions into concrete QPU-level workloads. We first work through a minimal but representative QEC subroutine, T-state teleportation, to illustrate kernel-embedded device callbacks and decoder interaction (subsection 6.1). We then analyze scalable fault-tolerant execution under lattice surgery, connecting decoder throughput and reaction time to parallel-window strategies and GPU batch execution (subsection 6.2). Finally, we describe how the availability of HPC with tight coupling can benefit calibration and QCVV workloads with fast, parameterized control in (subsection 6.3 and subsection 6.4).

6.1 Example: T gate teleportation in CUDA-Q

Experimental demonstrations of fault-tolerant quantum computing subroutines are an active research topic in quantum computing and an essential step toward practical quantum computation. To move beyond experimentation to production, developers need access to new primitives in quantum-capable programming languages, including real-time processing of syndrome data and low-latency data exchange between FPGAs and GPUs.

A canonical subroutine that depends on these real-time capabilities is magic state distillation and teleportation. Once a magic state is prepared on a logical ancilla qubit through magic state distillation, cultivation, or similar methods, it can be applied to a target qubit via teleportation [78, 79]. The logical measurement outcome of the ancilla qubit determines whether a conditional S gate must be applied to complete the teleportation protocol. This measurement step is where the decoder may block execution, since the syndrome history from both the target and ancilla qubits is analyzed to predict fault locations and determine whether the logical observable was flipped.

The following example in Listing 13 illustrates the key components involved in calling out to a decoder from CUDA-Q to perform T gate teleportation.

The decoder is first initialized and waits for data, while the CUDA-Q kernel is then launched, sending calls to the decoder kernel during execution. Much of the complexity lies in configuring the decoder server to understand what data to expect, how to convert raw stabilizer measurements into detector events, and which logical observables need to be decoded.

Listing 13. CUDA-Q C++ main for a T gate teleportation example.

```cpp
// Client/Server decoder example
int main(){
  // Now configure the decoders
  // Setting up the decoder properly requires information about the structure of the quantum kernel
  auto config = config_from_kernel(example_kernel, nData, nAncx, nAncz, x_schedule, z_schedule);
  cudaq::qec::decoding::config::configure_decoders(config);

  // Set up experiment parameters
  // ...
  // Here an example kernel is parametrized on physical qubits in a logical qubit.

  auto run_result = cudaq::run(numShots, example_kernel, nData, nAncx, nAncz, x_schedule, z_schedule);

  parse_logical_results(run_result);
  cudaq::qec::decoding::config::finalize_decoders();
}
```

Listing 14 shows the example kernel which has the main logic for the teleportation routine. The T gate production itself can be any routine, but many have a repeat-until-success component. We do not implement `t_distill_attempt`, but show and example which sits under a while loop until it returns that the protocol has been successful. In this case, simple flag checking is done under the hood, but more generally this can require calls to a decoder as well.

The bulk of gates executed in this example come from the rounds of stabilizer measurements. This is where most of the syndrome data is passed to the decoder.

In this example, the T gate is teleported onto a freshly initialized logical qubit, but the routine also demonstrates how teleportation can be applied to a logical qubit in a general quantum state. In such cases, the qubit persists beyond this step and participates in subsequent Clifford and non-Clifford gate operations, rather than being immediately measured as shown here.

What is not shown in the example are the explicit `cudaq::device_call` statements. This omission highlights an important aspect of the approach: different hardware platforms have varying requirements for classical interactions, and a flexible interface for invoking classical resources from within a quantum kernel is therefore essential. The `cudaq::device_call` mechanism enables such flexibility. In general, CUDA-Q provides high-level APIs designed to support multiple qubit technologies, while still allowing a modular, library-centric framework in which hardware developers can integrate their own solutions through `cudaq::device_call`.

In latency-sensitive regimes, the quantum kernels may resemble the example shown above, where only the minimum dynamism required—such as repeat-until-success logic and conditional S gates—is enabled. On hardware with greater tolerance for latency, the design can permit more frequent device calls. For instance, the decoder could be reconfigured dynamically based on intermediate computation results. Just-in-time compilation for lattice surgery routines is another promising direction that could lead to more efficient routing of logical two-qubit operations.

Listing 14. CUDA-Q kernel body, setting up and performing the teleportation.

```
// Client/Server style decoding
void __qpu__ example_kernel(int nData, int nAncx, int nAncz,
           const std::vector<std::size_t> &x_schedule,
           const std::vector<std::size_t> &z_schedule,
           int nRounds){
  // Allocate logical qubit 0
  cudaq::qvector data_q(nData);
  cudaq::qvector anc_x(nAncx);
  cudaq::qvector anc_z(nAncz);

  // lQ0 will distill t state
  cudaq::qec::patch logicalQ0(nData, nAncx, nAncz);

  // Example repeat-until-success distillation protocol
  bool success = false;
  while(success == false){
    success = t_distill_attempt(logicalQ0);
  }

  // lQ1 will be target of teleportation
  cudaq::qec::patch logicalQ1(nData, nAncx, nAncz);

  // 1 device in this example.
  int deviceID = 0;
  // 1 decoder in this example.
  int decoderID = 0;
  // Certain systems may require data tagging
  int tagID = 0;

  // Run stabilizer rounds to initialize into lQ1 into logical |0> state,
  // and preserve lQ0
  for(int r = 0; r < nRounds; ++r){
    // Specify how syndrome bits are sent to decoders
    stabilizer_round(deviceID, tagID, logicalQ0, x_schedule, z_schedule);
    tagID += 1;
    stabilizer_round(deviceID, tagID, logicalQ1, x_schedule, z_schedule);
    tagID += 1;
  }

  // transversal CX
  cx(logicalQ1, logicalQ0);

  // Additional QEC cycles
  for(int r = 0; r < nRounds; ++r){
    // Specify how syndrome bits are sent to decoders
    stabilizer_round(deviceID, tagID, logicalQ0, x_schedule, z_schedule);
    tagID += 1;
    stabilizer_round(deviceID, tagID, logicalQ1, x_schedule, z_schedule);
    tagID += 1;
  }

  auto data_readout0 = mz(logicalQ0.data);
  auto readout_packed = cudaq::to_integer(data_readout0);

  cudaq::device_call(custom_library::enqueue_syndromes_ui64,
      deviceID, readout_packed, decoder_id, tagID);

  bool correction = cudaq::device_call(custom_library::get_correction,
                       deviceID, readout_packed, decoder_id);

  // Conditional s gate to correct t state teleportation
  if(correction){
    s(logicalQ1);
  }

  // Depending on the system, may need additional rounds of QEC while waiting on decoder.
  // Here we simply readout.
  auto data_readout1 = mz(logicalQ1.data);
  return data_readout1;
}
```

Listing 15. CUDA-Q sub-kernel performing stabilizer rounds with data transfers to a decoder.

```
// X stabilizers and Z stabilizers specified via cnot schedules
__qpu__ std::vector<cudaq::measure_result>
stabilizer_round(int deviceID, int tagID, patch logicalQubit, const std::vector<std::size_t> &x_schedule,
           const std::vector<std::size_t> &z_schedule) {
  for (std::size_t i = 0; i < logicalQubit.ancx.size(); i++)
    reset(logicalQubit.ancx[i]);
  for (std::size_t i = 0; i < logicalQubit.ancz.size(); i++)
    reset(logicalQubit.ancz[i]);

  // x_stabilizer circuit
  h(logicalQubit.ancx);
  for (std::size_t xi = 0; xi < logicalQubit.ancx.size(); ++xi)
    for (std::size_t di = 0; di < logicalQubit.data.size(); ++di)
      if (x_schedule[xi * logicalQubit.data.size() + di] == 1)
        cudaq::x<cudaq::ctrl>(logicalQubit.ancx[xi], logicalQubit.data[di]);
  h(logicalQubit.ancx);

  // z_stabilizer circuit
  for (size_t zi = 0; zi < logicalQubit.ancz.size(); ++zi)
    for (size_t di = 0; di < logicalQubit.data.size(); ++di)
      if (z_schedule[zi * logicalQubit.data.size() + di] == 1)
        cudaq::x<cudaq::ctrl>(logicalQubit.data[di], logicalQubit.ancz[zi]);

  // Decoder convention dependent:
  // S = (S_X, S_Z), (x flip syndromes, z flip syndrones).
  // x flips are trigger z-stabilizers (ancz)
  // z flips are trigger x-stabilizers (ancx)
  auto syndrome_bits = mz(logicalQubit.ancz, logicalQubit.ancx);

  // Can use CUDA-Q wrapped call for device portability
  // int decoderID = deviceID;
  // cudaq::qec::decoding::enqueue_syndromes(decoderID, syndrome_bits);

  // Or call custom library implementation
  auto syndrome_packed = cudaq::to_integer(syndrome);
  cudaq::device_call(custom_library::enqueue_syndromes_ui64,
                     deviceID, syndrome_packed, decoder_id, tagID);
  return syndrome_bits;
}
```

6.2 Scalable fault tolerant quantum programs

Given access to magic states used as resource states, any universal quantum algorithm can be compiled in a sequence of multi-qubit Pauli measurements [22]. The fault-tolerant execution of multi-qubit Pauli measurements when using topological codes (such as the surface code) in hardware architectures constrained by nearest neighbor interactions requires a technique called lattice surgery [22, 25, 80]. In lattice surgery experiments, there can be logical spacelike failures (such as logical X, Y or Z errors occurring on logical qubits) or logical timelike failures, which occurs when the wrong parity of a multi-qubit Pauli measurement is obtained. Spacelike failures are exponentially suppressed with increasing code distance d, whereas timelike failures are exponentially suppressed with the number of syndrome measurement rounds d_m performed during the measurement of a multi-qubit Pauli operator [25, 81]. As such the runtime of an algorithm depends on both d and d_m (see for instance Eqs. (C7)-(C9) in Ref. [25]). An example of a $Y \otimes Y \otimes X$ measurement is shown in fig. 9a.

In what follows, we assume that the depth of a quantum algorithm corresponds to the number of sequential non-Clifford gates (for instance T gates or Toffoli gates). See Fig.6 in Ref. [22] for an example (note that multiple non-Cliffords such as T gates can be done in parallel, the depth of the circuit results from non-Clifford gates which cannot be parallelized). Before implementing the j'th sequential non-Clifford gate, the Pauli frame prior to the application of

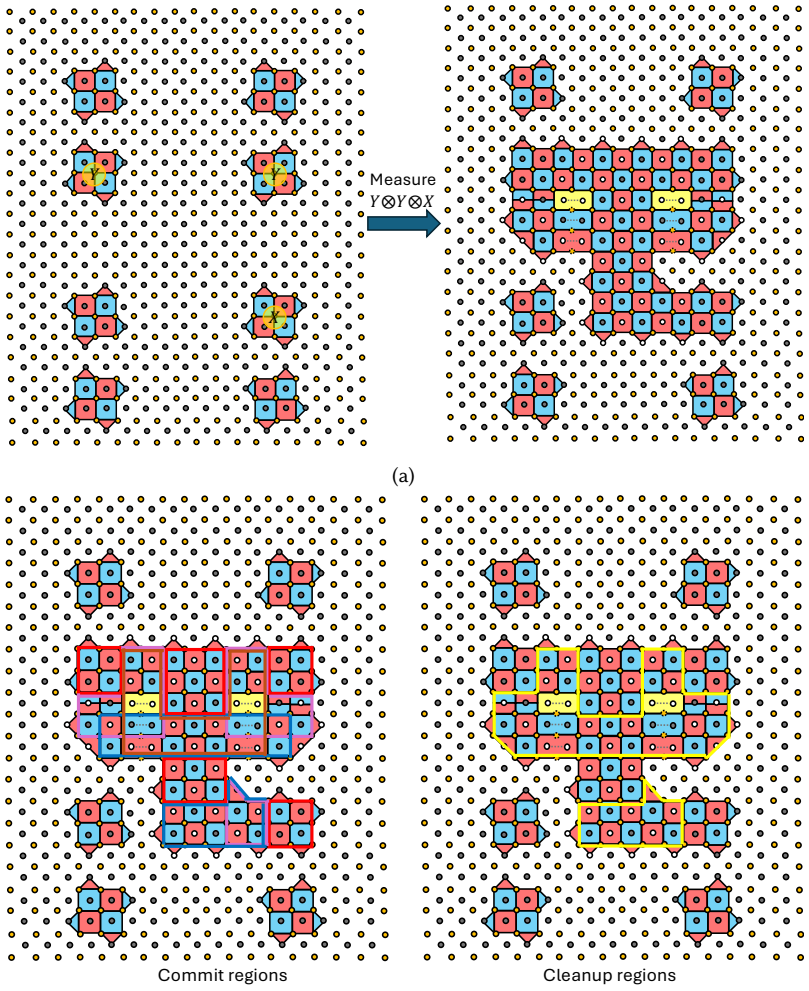


Fig. 9. (a) Example of a logical $Y \otimes Y \otimes X$ measurement implemented via lattice surgery for surface code patches with $d = 3$. The yellow vertices correspond to data qubits and grey vertices to ancillas. Red (blue) plaquettes correspond to X (Z)-type stabilizers. Twist defects are represented by yellow plaquettes. White vertices represent random stabilizer measurements in the first syndrome measurement round when the patches are merged. However the product of all stabilizers with white vertices corresponds to the parity of the $Y \otimes Y \otimes X$ measurement. Stabilizer measurements of the merged surface code patch are measured over d_m rounds, after which data qubits in the routing space region are measured in the X-basis which splits the surface code patches back to their original form. (b) Partitioning of a 2D slice of the merged lattice into commit (red boxes) regions and cleanup regions (yellow boxes). Errors in all commit regions are decoded in parallel. Likewise, errors in all cleanup regions are decoded in parallel. For commit regions, errors are corrected using stabilizer measurements from the red boxes, as well as surrounding buffer regions (shown by blue, purple and brown boxes).

the gate must be known [23, 82]. As such, all syndrome measurement rounds up to the consumption of the magic state must be processed. As was shown in Ref. [26], when decoding syndrome measurement rounds using a sliding window approach, the wait time needed to decode all syndrome measurement rounds for consumption of the magic state of the

j-th sequential non-Clifford gate is

$$T^{b_j} = \frac{c^j r}{T_s^{j-1}} + T_l \Big[\frac{T_s^{1-j}(c^j - T_s^j)}{c - T_s} \Big], \tag{1}$$

where we assume a linear time decoder that takes time $T_{\text{DEC}}^{(r)} = cr$ to decode r syndrome measurement rounds, and where c is a constant. In eq. (1), T_s is the time taken to perform one round of stabilizer measurements and T_l is the round-trip communication latency of the Real-time Interconnect. For a PQPU with two-qubit gate times of 100ns and a $1\mu s$ measurement + reset time for ancillas, we can set $T_s = 1.4\mu s$ for the surface code. If we assume that the number of syndrome measurement rounds prior to the consumption of the magic state used to implement the first non-Clifford gate is 33 (which applies to $d \approx 20$ surface codes when using lattice surgery to perform a $Z \otimes Z$ measurement) and take an inbound latency of $T_l = 20\mu s$ (note that in this work, a $4\mu s$ round trip latency was demonstrated), Ref. [26], showed that $c \lesssim T_s$ results in wait times T^{b_j} that grow sub-exponentially. The requirement that $c \lesssim T_s$ be challenging to achieve with realistic hardware and current state of the art QEC decoders. In Ref. [27, 66], an improvement was obtained by replacing a sliding window decoder with a parallel window decoder. Suppose that $L \geq r$ rounds need to be decoded for consuming a magic state and obtaining the current state of the Pauli frame. Instead of decoding the L rounds in sequential batches of r rounds, the L rounds are partitioned into commit and buffer regions (where each contains $O(d)$ rounds). Corrections in a given commit region using syndromes from the commit region as well as syndromes from buffer regions both immediately before and after the commit region. After performing such corrections, corrections in cleanup regions are performed to remove any remaining residual errors. See fig. 3 in Ref. [27]. Note that all commit and cleanup regions can be decoded in parallel which is a critical assumption for achieving real-time decoding in what follows. Let N_{par} be the number of parallel resources, n_{com} the number of rounds in a commit region and n_W the number of rounds in a cleanup region. Let $2T_{\text{DEC}}$ be the time to decode the commit regions and cleanup regions. Ref. [27] showed that the exponential backlog problem can be avoided if N_{par} satisfies

$$N_{\text{par}} \geq \frac{2T_{\text{DEC}}}{(n_{\text{com}} + n_W)(T_l + T_s)}. \tag{2}$$

The above discussion of parallel window decoding focused purely on the time domain. When performing lattice surgery, the effective distance d_{eff} of merged surface code patches can be very large (see for instance fig. 9a). Decoding a surface code patch with $d_{\text{eff}} \gg d$ can result in T_{DEC} times which are too large for practical considerations. However, commit/buffer and cleanup regions can also be used in the space domain in addition to the time domain. An example is provided in fig. 9b. In such a setting, commit regions and their corresponding buffer regions have both a spatial and temporal component and can all be decoded in parallel. Spatial cleanup regions correspond to residual portions of the lattice that need to be corrected after performing corrections in the commit regions. Similarly to commit regions, cleanup regions have both a spatial and temporal component and are also decoded in parallel. With enough parallel resources for spatial commit/cleanup regions, T_{DEC} for lattice surgery can be made commensurate to T_{DEC} times for pure memory settings with codes of distance d even when $d_{\text{eff}} \gg d$.

We conclude this section by remarking that GPUs are particularly well suited for parallel window decoding protocols. Decoding can be viewed as an inference problem where each commit/buffer region corresponds to one element of the input batch size (and similarly for cleanup regions). In the context of AI-based decoders, if a GPU has enough memory to store the model, each batch element can (in principle) be processed in parallel. Although AI-based decoders are difficult to scale to large code distances [18], AI-based pre-decoders have been shown to scale to very large code distances [26, 49]. Pre-decoders correct most of the local error chains (which are the dominant sources of errors in topological

codes) and thus can be used to significantly accelerate a global algorithmic decoder. Current work at NVIDIA is being done to build pre-decoders that can accelerate algorithmic decoders by more than an order of magnitude with the AI model having efficient implementations on a GPU [83]. Since modern GPUs can efficiently process large batch sizes, obtaining an N_{par} that satisfies eq. (2) becomes more straightforward.

6.3 QPU Calibration

Across the hardware stack of a future large scale quantum computer, there will be many compile-time and run-time decisions which must be made on-line without user intervention to decide things like the traversal direction of a calibration workflow given measured data, whether a particular characterization protocol should be mapped to a local control FPGA or out to a host CPU, and how to decode a set of error syndromes and with which computational resource. In particular, control systems with tightly coupled CPU and GPU compute will significantly improve the development and implementation of conditional calibration workflows which take advantage of in-depth characterization routines for the bring-up and maintenance of physical qubits in the presence of many sources of noise and time-dependent drift.

Recent work details methods for bringing up a set of qubits from an uncalibrated state and then maintaining their performance using a conditional, directed, acyclic calibration graph [84]. In this premise, QPU developers design a calibration flow described by a graph structure that runs experiments, analyzes their measurement outputs to determine whether the parameters under test are within their desired specification, and then determines the next calibration step to run based on the current state of the system. The individual calibration steps can be simple, from sweeping a control frequency to determine a qubit's resonance, to much more complicated routines which utilize characterization techniques like gate set tomography [85] or randomized benchmarking [86, 87] to generate metrics which inform parameter updates. These characterization routines (discussed more thoroughly in Section 6.4) can introduce much more computational complexity to design robust experiments, run complex circuits on the QPU quickly and efficiently, and fit their results. An architecture like the NVQLink which enables easier programming of these calibration graph structures will enable QPU development teams to bring up their devices faster and maintain high performance without having to invest as much overhead developing the firmware and software systems to enable this functionality.

For simple calibration routines, the analysis required may involve simple peak detection or curve fitting, while more complex routines may require complex characterization. For example, running gate set tomography for even two qubits can require running thousands of quantum circuits along with computationally expensive optimization routines which take on the time scale of hours to solve on a desktop CPU. This generally disallows rapid feedback which would be useful for calibration. There have been demonstrations of hardware-accelerated implementations of smaller characterization routines like quantum state tomography in FPGA [88], but even this protocol runs into limits of a small FPGA's LUT and BRAM resources for anything greater than six qubits. In practice, many research groups utilize faster but less informative characterization routines like randomized benchmarking (RB) to benchmark errors in their systems and determine if a calibration step improves or worsens the RB metric. For example, a laboratory experiment with CPU-hosted, AWG-based control for characterizing single qubit leakage rates using leakage RB [89] takes about ten minutes (dominated by circuit runtime), while running GST and extracting leakage rates for a single qubit takes on the order of an hour. Both circuit runtime and fitting the measurement results to useful metrics can be sped up by compute tightly coupled to the control system, enabling calibration routines which use more complex characterization for highly accurate parameter estimation.

Additionally, stabilizing time-dependent drift is an important part of calibration workflows for quantum devices. Recent work has demonstrated single- and few-shot calibration protocols which operate very quickly to track and

calibrate for drift in the parameter space of a quantum system [90, 91]. These sorts of protocols can be integrated towards the end of calibration graph structures, where after a QPU has been brought to a stable operating point, low-latency calibration protocols take over to stabilize drift between running application circuits. However, even these intentionally lightweight protocols can require significant overhead depending on the control system implementing them and the number of qubits under calibration, and control systems must be able to implement corrections on a very fast time scale to track drift closely. For example, these protocols were demonstrated in an FPGA implementation and a cryogenic-CMOS ASIC simulation to be able to improve gate fidelities by orders of magnitude depending on the time between parameter updates [91]. In this demonstration, the steady state infidelity of a simulated single-qubit gate with drifting optimal parameters reduced from roughly 10^{-4} to 10^{-5} when the calibration update time reduced from 10 microseconds to 1 microsecond, with further improvements achievable with faster updates. This would require fast pulse generation and classical compute which would be enabled by systems such as the NVQLink. The example in [91] demonstrated the computational cost of tuning a single-qubit gate to be quite low, but scaling to the simultaneous drift control of hundreds to thousands of qubits and their gate operations will pose a challenge for the control system which would be aided by tightly coupled classical compute. Further, in the case of error-corrected logical qubits, the few-shot calibration protocols presented in [90] can be extended to use syndrome data to inform calibration. This capability would be advanced by having the decoding systems and calibration systems present within the same tightly coupled control interface, such as in NVQLink.

Overall, we see a wide variety of application spaces for low-latency calibration of quantum devices when tightly coupled with CPU and GPU compute. Scheduling calibration routines via an automated graph-like system, performing complex characterization routines in order to inform calibration steps, and maintaining low-latency communication for time-dependent drift control of a rapidly scaling parameter space all become significantly easier with tightly coupled compute which can quickly inform the control systems of the QPU. We see the NVQLink fitting well into this specific space to enable the creation of stable, scalable logical qubits.

6.4 QCVV and benchmarking

The computational power of quantum computers today is severely limited by noise and control errors. Noise characterization and calibration experiments are critically important to understanding this noise and maximizing these devices' capabilities. Quantum characterization, verification, and validation (QCVV) experiments, such as gate set tomography [85] or randomized benchmarking [86, 87], are valuable tools improving hardware performance, but are experimentally challenging and can easily outmatch the abilities of the classical control systems. These protocols can comprise hundreds or thousands of unique quantum circuits, can sometimes utilize random gates [87, 92], and can, independently, come with specifications to execute the circuits in strict order, returning time-stamped measurement results [93]. Circuits may need to be designed adaptively based on measurement results [94] and dispatched to the QPU with low latency. Calibration experiments must further modify low-level pulse parameters [95], as frequently as after each circuit execution or even within individual circuits. Existing classical control infrastructure is often not up to the task.

A major barrier to meeting these demanding requirements is the reliance on host-driven arbitrary waveform playback for pulse generation. In this conventional architecture, each control channel is driven by an arbitrary waveform generator (AWG) whose output is pre-computed on a host computer, uploaded to the instrument's onboard memory, and played back verbatim during circuit execution. Because AWG memory is limited, the complete set of QCVV circuits often cannot be stored at once, requiring experiments to be divided into small batches. For each batch, all waveforms must be

compiled, transferred to the instrument, and executed sequentially. This batching prohibits *rastering* through the full circuit list, wherein one shot is taken from each circuit in turn, repeatedly, until the total number of desired shots is reached. Rastering of the data acquisition reveals time correlation and can reduce bias in error estimates due to drift, but batching makes it impossible. These disadvantages are compounded by the lengthy upload times involved. Transferring waveforms for batches of circuits, which can number in the hundreds or thousands, takes anywhere from minutes to hours. Such delays not only incur significant quantum downtime but also fragment the experimental process, making it difficult to capture precise snapshots of device performance at a particular point in time.

Table 2 summarizes the controller resource requirements for four classes of QCVV experiment: traditional “static" QCVV, adaptive QCVV, single-shot calibration, and calibration using deep reinforcement learning. Each stresses different combinations of the controller subsystems.

Application	Resource Requirements					
	Controller Memory	Computational Throughput	Latency (Real-time Interconnect)	Latency (Host Connection)	Real-time Pulse Updates	Control Flow Flexibility
Traditional “Static" QCVV	High	Low	Low	Medium	None	Medium
Adaptive/Online QCVV	High	High	High	Medium	Low	High
Calibration (Single-Shot)	Low	Medium	High	Low	High	Medium
Calibration (RL)	High	High	High	Low	High	High

Table 2. Classical controller resource requirements for four representative classes of QCVV experiments/protocols. Different classes of QCVV protocol can vary widely in their overall resource requirements and the specific subsystems they stress.

Traditional “static" QCVV protocols include common experiments such as randomized benchmarking and gate set tomography (GST) and involve the construction of experiments consisting of possibly hundreds or even thousands of circuits specially structured to learn one of more properties of a quantum processor, *e.g.*, gate or SPAM infidelities, with most computational analysis performed on a host system in post-processing. While these experiments don’t rely heavily on access to realtime compute resources, they very frequently stress available memory resources, particularly in older AWG-based controllers. Real-time controllers based on FPGAs or RFSoCs are able to generate waveforms on the fly based on few-bit keywords. In these cases, increased flexibility in control workflows significantly reduces both controller memory requirements and the latency requirements with a host system [71].

Adaptive QCVV experiments—ones which aim to learn the characteristics of a quantum processor in real-time with dynamically constructed experiment designs—and real-time calibration protocols demand much more from the control hardware. They require nearly real-time interconnects between the controller and host, to enable shot-by-shot updates to control parameters, or even within-shot updates. Even slight delays can hinder the performance of feedback loops and complicate the execution of branching control logic. Meanwhile, the computational throughput needed for on-the-fly decision making can vary widely among these tasks. For instance, simple fast-feedback routines may require only basic arithmetic with minimal memory, but when controllers integrate complex algorithms, such as those based on deep reinforcement learning or requiring decoding of QEC syndrome data, they increasingly stress every aspect of the control system.

Modern real-time controllers based on FPGAs or RFSoCs integrate digital waveform synthesis and sequencing directly in hardware. Rather than uploading large waveform buffers, they generate parameterized pulses in real time, enabling adaptive control, feedback, and fine-grained synchronization without the bottlenecks of host-side compilation and transfer, or the memory requirements of storing precompiled waveforms. This increased flexibility in control workflows

significantly reduces both controller memory requirements and the latency requirements with a host system [71]. Further incorporating low-latency interconnects to fast classical co-processing can enable even the most demanding adaptive calibration and characterization routines. Tiered heterogeneous control architectures, like NVQLink, are wellsuited to meet these needs by offloading light computational tasks, such as basic arithmetic or inference on small neural networks, to FPGA hardware for rapid processing. More complex tasks, including training and inference of statistical models or larger neural networks, are handled on CPU or GPU nodes via a slightly higher-latency connection.

Transitioning away from host-controlled AWG controllers in favor of these lower-latency architectures will not just allow existing QCVV and calibration routines to run as intended, but will also encourage development and deployment of new classes protocols that will enable hardware developers and users to probe and mitigate errors with ever finer resolution and speed.

7 Development and Simulation

Although the primary motivation for the NVQLink architecture is to support quantum computing at scale, we recognize that the utility of any platform depends strongly on the ability to develop and maintain software on that platform. This section describes offline tools that support quantum program development and validation outside the real-time execution path, enabling developers to test and refine quantum programs before deployment on physical hardware.

CUDA-Q [96] provides logical-level circuit simulators (state vector, density matrix, tensor network) [97] to support research in quantum computing and application development [98]. Beyond this logical layer, we distinguish two complementary capabilities for physical hardware emulation: the VPPU, which emulates PPU instruction-level behavior to enable offline testing of compiled quantum programs, and PQPU simulators, which model quantum dynamics at the Hamiltonian level to provide insights into physical implementation fidelity. These tools operate at different architectural layers—VPPU at the PPU compiler and instruction set architecture (ISA) level, and PQPU simulators at the quantum state evolution level. The VPPU abstraction is introduced in subsection 7.1, and we include a few notes on the role of PQPU simulation in subsection 7.2. A particularly important use case in focus here is the simulation of QEC encoded programs, which we discuss in subsection 7.3.

7.1 Virtual Pulse Processing Unit

Developing and validating quantum programs for tightly coupled systems presents a fundamental challenge: compiled programs must target PPU-specific instruction sets with precise timing constraints, yet validating these programs traditionally requires deploying to physical hardware. This creates development bottlenecks, as iterations on quantum control logic, real-time protocols (QEC decoding, adaptive calibration), and pulse sequences require continuous hardware access. The VPPU addresses this challenge by providing a substitutable PPU emulator that enables offline program validation in the PTD. A recent prototyping attempt [99] has shown promise in achieving full pipeline validation. By transforming ISA instructions into an appropriate signal representation for physical QPU simulation (eg. $V(t)$ in the control Hamiltonian of the PQPU, where t evolves in PTD), the VPPU allows developers to test instruction sequences, validate timing constraints, and debug pulse schedules before committing to physical execution, thereby accelerating development cycles for both quantum programs and real-time classical protocols.

As defined in subsection 2.1, the VPPU must be substitutable for the physical PPU. This substitutability ensures that quantum programs compiled for physical hardware can be tested and validated offline without modification. VPPU implementations achieve this by implementing the `quantum_control_trait` interface defined in the runtime architecture (subsection 4.1), presenting the same programmatic interface as their corresponding physical PPU devices.

If offered, VPPU implementations must:

- Fully implement target PPU's instruction set (ISA)
- Transform ISA control instructions to signal representations usable in PQPU dynamical simulation
- Emulatie PQPU readout: convert signal representations in a PQPU simulation into correct PPU data formats
- Maintain compatibility with the PPU compiler
- Support the same trait interface as its physical PPU or QSC

The signal representations produced by VPPU enable offline inspection and visualization of compiled pulse sequences. Developers can analyze timing relationships, identify potential resource conflicts, and validate pulse scheduling logic before deploying to physical hardware. This capability is particularly valuable for debugging complex multi-qubit operations and verifying that compiled programs meet timing constraints of the target QPU modality.

VPPU implementations maintain an interface that transforms ISA instructions into signal representations in the PTD. To achieve this output, implementers may choose various approaches—from lookup tables for simple waveforms to full quantum dynamics simulation. Some VPPU implementations may use Physical QPU simulators (subsection 7.2) as computational backends for computing eg. $V(t)$ from ISA instructions, but this is entirely an implementation choice invisible to calling code. The VPPU interface remains strictly at the PPU instruction level (ISA $\rightarrow V(t)$), maintaining clear architectural separation from Hamiltonian-level simulation. Higher-level optimization tasks such as reinforcement learning-based calibration (subsection 6.3), or QEC decoder training (subsection 7.3) are independent tools that may use VPPU as a library component.

7.2 Physical QPU Simulation

Physical QPU simulators are computational tools that model quantum dynamics at the Hamiltonian level, often operating on quantum state vectors or density matrices. While a VPPU emulates PPU instruction execution (ISA $\rightarrow V(t)$), a PQPU simulator models the fundamental quantum state evolution (Hamiltonian $\rightarrow |\psi\rangle$), where the time variable t in the control Hamiltonian evolves in the PTD.

PQPU simulators can function as backends for VPPU implementations to provide quantum state fidelity modeling under physical noise, as tools for physical parameter characterization and optimization, or as While they need not run in the RTD as realtime applications, we expect that in practice they will usefully be supported on the Real-time Host in a timesharing mode.

7.3 QEC simulation

Simulation of quantum programs is fundamental to quantum algorithm research and development, and this remains true in the study of quantum error correction (QEC). However, QEC protocols impose unique demands on simulation, emphasizing different dimensions compared to traditional algorithmic workloads. Like quantum algorithms, simulating QEC routines allows researchers to validate and optimize new proposals.

A key focus in these studies is the logical error rate of a specific protocol. For instance, a quantum memory experiment may involve preparing a logical quantum state and performing repeated rounds of error correction to extend the logical qubit's lifetime beyond that of its constituent physical qubits. Such processes can be simulated, where the resulting logical error rate depends on three main factors: the quantum circuit, the noise model, and the decoder. Understanding how each of these influences logical qubit fidelity is a central task for researchers seeking to design the next generation of fault-tolerant quantum architectures.

Although the utility of QEC simulation is considerable, the structure of these protocols poses challenges for conventional approaches. Because a logical qubit is encoded into many physical qubits, QEC simulation scales poorly with techniques such as density matrix (DM) or statevector (SV) simulation. Tensor network (TN) simulators can handle larger qubit counts when entanglement remains sparse, but QEC circuits typically employ multiple layers of two-qubit gates, quickly reaching the limits of TN efficiency. Moreover, frequent mid-circuit measurements common in QEC workflows further complicate these simulation methods.

One major opportunity lies in the fact that much of QEC involves Clifford operations, which can be simulated efficiently using stabilizer simulators such as Stim [100]. Stim enables researchers to explore the interplay of circuits, noise models, and decoders: three essential components of QEC studies. Its design excels for offline decoding workloads, where large batches of simulated shots are generated and later decoded to evaluate protocol performance. Stim's circuit language is optimized for this case and prioritizes execution speed, though it does not support conditional gate application.

The gap across these simulation approaches is the inability to model full QEC workflows, including scenarios such as magic state distillation and conditional gate execution based on mid-circuit decoding results. Addressing this gap is a key motivation behind the NVQLink architecture, which supports such workflows through the `cudaq::device_call` interface. Library code can be written to switch seamlessly between real QPU execution and an emulated mode for simulation.

Listing 16. Enqueue syndromes wrapper function.

```
__qpu__ void
enqueue_syndromes(std::uint64_t decoder_id,
                  const std::vector<cudaq::measure_result> &syndromes,
                  std::uint64_t tag) {
#ifdef LIBSIM
  cudaq::device_call(enqueue_syndromes_simulation, decoder_id, syndromes, tag);
#else
  uint64_t syndrome_size = syndromes.size();
  uint64_t syndrome = cudaq::to_integer(syndromes);
  cudaq::device_call(enqueue_syndromes_ui64, decoder_id, syndrome_size,
                     syndrome, tag);
#endif
}
```

In Listing 16, we show an example of sending measurement data to a decoder. When the library is compiled with `-DLIBSIM`, the `enqueue_syndromes` function produces a binary optimized for simulation, eliminating the need to pack measurement data into integers by directly using the size field of the `std::vector`. When targeting real hardware, the developer instead packs the measurement results explicitly and specifies the number of syndrome bits. Both cases are represented in the example, enabling the same application code to be validated under simulation before being re-targeted to an experimental control system.

By incorporating decoder calls and conditional gate logic directly in application code, this design also generalizes across simulation strategies. For example, extended stabilizer simulation for scenarios which are very nearly Clifford, or back to state vector simulation when qubit counts are low but advanced noise modeling is needed. Stabilizer-based simulation will serve as a natural starting point for many QEC workloads, yet CUDA-Q kernels written in this style can retarget to any simulation strategy, or hardware backend, so long as the necessary data transfer semantics are defined via the appropriate `cudaq::device_call` bindings.

8 Conclusion

We have presented NVQLink, a platform architecture that tightly couples high-performance classical compute to quantum-processor control systems. The architecture defines a model of a Logical QPU comprising CPUs, GPUs, and PPUs connected by a low-latency Real-time Interconnect, and it distinguishes four time domains—Physical (PTD), Deterministic (DTD), Real-time (RTD), and Application (ATD)—to reason about correctness, latency budgets, and performance requirements end-to-end. A RoCE-based proof of concept demonstrates sub-4 μs steady-state round-trip latency with commodity networking equipment and a GPU-resident loopback path, indicating a practical route to scaling radix and bandwidth while keeping jitter low.

We presented proposed extensions to CUDA-Q with device-addressable real-time callbacks (`cudaq::device_call`) and a heterogeneous memory abstraction (`cudaq::device_ptr<T>`), enabling quantum kernels to invoke deterministic classical computation with compiler-managed marshaling. We outlined compilation strategies across latency regimes—AOT lowering to pulse/ISA with RDMA and persistent kernels for short-timescale modalities, and host-mediated streaming with JIT for longer-timescale modalities—expressed via Quake/CC MLIR and unified lowering.

Our proposed runtime architecture offers a zero-overhead, trait-based model to compose device capabilities—explicit data marshaling, device callbacks, quantum control (`quantum_control_trait`), and RDMA—plus representations for compiled quantum kernels, pluggable compilers/executors, and a Logical QPU Driver API for allocation, transfer, and launching on both physical PPUs and substitutable VPPUs.

We surveyed QPU-level workloads, including an example of T state distillation, showing how GPU batch inference and pre-decoders reduce backlog and align with RTD constraints. For calibration and QCVV, we showed how moving beyond host-driven AWG playback to parameterized, real-time control with GPU/CPU co-processing enables adaptive protocols, short-cycle drift mitigation, and within-shot feed-forward.

Finally, we described development tools: VPPU as a drop-in emulator at the PPU ISA boundary, PQPU simulators for Hamiltonian-level studies, and CUDA-Q simulators for logical-level testing—allowing the same source to retarget between emulation and hardware via `device_call`.

We invite QPU and QSC builders, HPC centers, and researchers to evaluate and engage with our proposal and communicate with us about requirements. Our intent is a pragmatic, open path to real-time accelerated computing in the QPU domain that scales from today's devices to fault-tolerant systems.

Acknowledgments

The authors would like thank the following individuals for helpful conversations and feedback in preparing the material for this manuscript: Owen Arnold, Simeon Baker-Finch, Francesco Battistel, Gilan Ben-Shach, Matthew Bradley, Gustavo Cancelo, Sergio Cantu, Arnaud Carignan-Dugas, Yonatan Cohen, Coleman Collins, Niccolo Coppola, Erik Davis, Patrick Deuley, Nicolas Didier, Mengke Feng, Brett Freeman, Louis Fry-Bouriaux, Joanna Fulton, Pranav Gokhale, Eric Holland, Andrew Houck, Matthew Hutchings, Jerome Javelle, Ronan Jezequel, Gwen Johnson, Glenn Jones, Ryan Jones, William Kindel, Jeffrey Marshall, Josh Moles, Yasunobu Nakamura, Tom Noel, Thomas Ohki, Simon Philips, Laurent Prost, Matthew Reagor, David Reilly, David Rivas, Yoav Romach, Christoph Rühle, Colm Ryan, Kentaro Sano, Andre Saraiva, Mitsuhisa Sato, Laura Schulz, Michael Sorenson, Ramon Szmuk, Ryousei Takano, Brian Tarasinski, Jeff Thompson, Sho Uemura, David van Zanten, Oded Wertheim, Evan Zalys-Geller, Avishai Ziv.

Support is also acknowledged from the U.S. Department of Energy, Office of Science, National Quantum Information Science Research Centers, Quantum Systems Accelerator under Air Force Contract No. FA8702-15-D-0001. These results are based in part upon work supported by the U.S. Department of Energy, Office of Science, National Quantum Information Science Research Centers, Quantum Science Center. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the Dept. of Energy. Y.J. would like to thank the National Research Foundation, Singapore, National Quantum Office under its National Quantum Computing Hub and Hybrid Quantum-Classical Computing 1.0 programmes. A*STAR under C23091703 and Q.InC Strategic Research and Translational Thrust for the funding support. Y.X., G.H., I.S. would like to thank the support from the U.S. Department of Energy, Office of Science, Advanced Scientific Computing Research (ASCR) Quantum Testbed Program under Contract No. DE-AC02-05CH11231.

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology & Engineering Solutions of Sandia, LLC, a wholly owned subsidiary of Honeywell International Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525. This paper describes objective technical results and analysis. Any subjective views or opinions that might be expressed in the paper do not necessarily represent the views of the U.S. Department of Energy or the United States Government.

References

[1] Masoud Mohseni, Artur Scherer, K. Grace Johnson, Oded Wertheim, Matthew Otten, Navid Anjum Aadit, Kirk M. Bresniker, Kerem Y. Camsari, Barbara Chapman, Soumitra Chatterjee, Gebremedhin A. Dagnew, Aniello Esposito, Farah Fahim, Marco Fiorentino, Abdullah Khalid, Xiangzhou Kong, Bohdan Kulchytskyy, Ruoyu Li, P. Aaron Lott, Igor L. Markov, Robert F. McDermott, Giacomo Pedretti, Archit Gajjar, Allyson Silva, John Sorebo, Panagiotis Spentzouris, Ziv Steiner, Boyan Torosov, Davide Venturelli, Robert J. Visser, Zak Webb, Xin Zhan, Yonatan Cohen, Pooya Ronagh, Alan Ho, Raymond G. Beausoleil, and John M. Martinis. How to Build a Quantum Supercomputer: Scaling Challenges and Opportunities, November 2024.

[2] 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 Munich Quantum Software Stack: Connecting End Users, Integrating Diverse Quantum Technologies, Accelerating HPC, September 2025.

[3] Eric Mansfield, Stefan Seegerer, Panu Vesanen, Jorge Echavarria, Burak Mete, Muhammad Nufail Farooqi, and Laura Schulz. First Practical Experiences Integrating Quantum Computers with HPC Resources: A Case Study With a 20-qubit Superconducting Quantum Computer. URL http://arxiv.org/abs/2509.12949.

[4] Gilles Buchs, Thomas Beck, Ryan Bennink, Daniel Claudino, Andrea Delgado, Nur Aiman Fadel, Peter Groszkowski, Kathleen Hamilton, Travis Humble, Neeraj Kumar, et al. The role of quantum computing in advancing scientific high-performance computing: A perspective from the adac institute. *arXiv preprint arXiv:2508.11765*, 2025.

[5] Hassan Shapourian, Eneet Kaur, Troy Sewell, Jiapeng Zhao, Michael Kilzer, Ramana Kompella, and Reza Nejabati. Quantum Data Center Infrastructures: A Scalable Architectural Design Perspective, January 2025.

[6] Daan Camps, Ermal Rrapaj, Katherine Klymko, Brian Austin, and Nicholas J. Wright. Evaluation of the Classical Hardware Requirements for Large-Scale Quantum Computations. In *ISC High Performance 2024 Research Paper Proceedings (39th International Conference)*, pages 1–12, May 2024. doi: 10.23919/ISC.2024.10528937.

[7] Paul V. Klimov, Andreas Bengtsson, Chris Quintana, Alexandre Bourassa, Sabrina Hong, Andrew Dunsworth, Kevin J. Satzinger, William P. Livingston, Volodymyr Sivak, Murphy Yuezhen Niu, Trond I. Andersen, Yaxing Zhang, Desmond Chik, Zijun Chen, Charles Neill, Catherine Erickson, Alejandro Grajales Dau, Anthony Megrant, Pedram Roushan, Alexander N. Korotkov, Julian Kelly, Vadim Smelyanskiy, Yu Chen, and Hartmut Neven. Optimizing quantum gates towards the scale of logical qubits. *Nature Communications*, 15(1):2442, March 2024. ISSN 2041-1723. doi: 10.1038/s41467-024-46623-y.

[8] Theodore J. Yoder, Eddie Schoute, Patrick Rall, Emily Pritchett, Jay M. Gambetta, Andrew W. Cross, Malcolm Carroll, and Michael E. Beverland. Tour de gross: A modular quantum computer based on bivariate bicycle codes. *arXiv preprint arXiv:2506.03094*, 2025. In preparation.

[9] Google Quantum AI and Collaborators, Rajeev Acharya, Dmitry A. Abanin, Laleh Aghababaie-Beni, Igor Aleiner, Trond I. Andersen, Markus Ansmann, Frank Arute, Kunal Arya, Abraham Asfaw, Nikita Astrakhantsev, Juan Atalaya, Ryan Babbush, Dave Bacon, Brian Ballard, Joseph C. Bardin, Johannes Bausch, Andreas Bengtsson, Alexander Bilmes, Sam Blackwell, Sergio Boixo, Gina Bortoli, Alexandre Bourassa, Jenna Bovaird, Leon Brill, Michael Broughton, David A. Browne, Brett Buchea, Bob B. Buckley, David A. Buell, Tim Burger, Brian Burkett, Nicholas Bushnell, Anthony Cabrera, Juan Campero, Hung-Shen Chang, Yu Chen, Zijun Chen, Ben Chiaro, Desmond Chik, Charina Chou, Jahan Claes, Agnetta Y. Cleland, Josh Cogan, Roberto Collins, Paul Conner, William Courtney, Alexander L. Crook, Ben Curtin, Sayan Das, Alex Davies, Laura De Lorenzo, Dripto M. Debroy, Sean Demura, Michel Devoret, Agustin Di Paolo, Paul Donohoe, Ilya Drozdov, Andrew Dunsworth, Clint Earle, Thomas Edlich, Alec Eickbusch, Aviv Moshe Elbag, Mahmoud Elzouka, Catherine Erickson, Lara Faoro, Edward Farhi, Vinicius S. Ferreira, Leslie Flores Burgos, Ebrahim Forati, Austin G. Fowler, Brooks Foxen, Suhas Ganjam, Gonzalo Garcia, Robert Gasca, Élie Genois, William Giang, Craig Gidney, Dar Gilboa, Raja Gosula, Alejandro Grajales Dau, Dietrich Graumann, Alex Greene, Jonathan A. Gross, Steve Habegger, John Hall, Michael C. Hamilton, Monica Hansen, Matthew P. Harrigan, Sean D. Harrington, Francisco J. H. Heras, Stephen Heslin, Paula Heu, Oscar Higgott, Gordon Hill, Jeremy Hilton, George Holland, Sabrina Hong, Hsin-Yuan Huang, Ashley Huff, William J. Huggins, Lev B. Ioffe, Sergei V. Isakov, Justin Iveland, Evan Jeffrey, Zhang Jiang, Cody Jones, Stephen Jordan, Chaitali Joshi, Pavol Juhas, Dvir Kafri, Hui Kang, Amir H. Karamlou, Kostyantyn Kechedzhi, Julian Kelly, Trupti Khaire, Tanuj Khattar, Mostafa Khezri, Seon Kim, Paul V. Klimov, Andrey R. Klots, Bryce Kobrin, Pushmeet Kohli, Alexander N. Korotkov, Fedor Kostritsa, Robin Kothari, Borislav Kozlovskii, John Mark Kreikebaum, Vladislav D. Kurilovich, Nathan Lacroix, David Landhuis, Tiano Lange-Dei, Brandon W. Langley, Pavel Laptev, Kim-Ming Lau, Loïck Le Guevel, Justin Ledford, Joonho Lee, Kenny Lee, Yuri D. Lensky, Shannon Leon, Brian J. Lester, Wing Yan Li, Yin Li, Alexander T. Lill, Wayne Liu, William P. Livingston, Aditya Locharla, Erik Lucero, Daniel Lundahl, Aaron Lunt, Sid Madhuk, Fionn D. Malone, Ashley Maloney, Salvatore Mandrà, James Manyika, Leigh S. Martin, Orion Martin, Steven Martin, Cameron Maxfield, Jarrod R. McClean, Matt McEwen, Seneca Meeks, Anthony Megrant, Xiao Mi, Kevin C. Miao, Amanda Mieszala, Reza Molavi, Sebastian Molina, Shirin Montazeri, Alexis Morvan, Ramis Movassagh, Wojciech Mruczkiewicz, Ofer Naaman, Matthew Neeley, Charles Neill, Ani Nersisyan, Hartmut Neven, Michael Newman, Jiun How Ng, Anthony Nguyen, Murray Nguyen, Chia-Hung Ni, Murphy Yuezhen Niu, Thomas E. O'Brien, William D. Oliver, Alex Opremcak, Kristoffer Ottosson, Andre Petukhov, Alex Pizzuto, John Platt, Rebecca Potter, Orion Pritchard, Leonid P. Pryadko, Chris Quintana, Ganesh Ramachandran, Matthew J. Reagor, John Redding, David M. Rhodes, Gabrielle Roberts, Eliott Rosenberg, Emma Rosenfeld, Pedram Roushan, Nicholas C. Rubin, Negar Saei, Daniel Sank, Kannan Sankaragomathi, Kevin J. Satzinger, Henry F. Schurkus, Christopher Schuster, Andrew W. Senior, Michael J. Shearn, Aaron Shorter, Noah Shutty, Vladimir Shvarts, Shraddha Singh, Volodymyr Sivak, Jindra Skruzny, Spencer Small, Vadim Smelyanskiy, W. Clarke Smith, Rolando D. Somma, Sofia Springer, George Sterling, Doug Strain, Jordan Suchard, Aaron Szasz, Alex Sztein, Douglas Thor, Alfredo Torres, M. Mert Torunbalci, Abeer Vaishnav, Justin Vargas, Sergey Vdovichev, Guifre Vidal, Benjamin Villalonga, Catherine Vollgraff Heidweiller, Steven Waltman, Shannon X. Wang, Brayden Ware, Kate Weber, Travis Weidel, Theodore White, Kristi Wong, Bryan W. K. Woo, Cheng Xing, Z. Jamie Yao, Ping Yeh, Bicheng Ying, Juhwan Yoo, Noureldin Yosri, Grayson Young, Adam Zalcman, Yaxing Zhang, Ningfeng Zhu, and Nicholas Zobrist. Quantum error correction below the surface code threshold. *Nature*, December 2024. ISSN 0028-0836, 1476-4687. doi: 10.1038/s41586-024-08449-y.

[10] J. M. Pino, J. M. Dreiling, C. Figgatt, J. P. Gaebler, S. A. Moses, M. S. Allman, C. H. Baldwin, M. Foss-Feig, D. Hayes, K. Mayer, C. Ryan-Anderson, and B. Neyenhuis. Demonstration of the trapped-ion quantum-CCD computer architecture. *Nature*, 592(7853):209–213, April 2021. ISSN 0028-0836, 1476-4687. doi: 10.1038/s41586-021-03318-4.

[11] Dolev Bluvstein, Simon J. Evered, Alexandra A. Geim, Sophie H. Li, Hengyun Zhou, Tom Manovitz, Sepehr Ebadi, Madelyn Cain, Marcin Kalinowski, Dominik Hangleiter, J. Pablo Bonilla Ataides, Nishad Maskara, Iris Cong, Xun Gao, Pedro Sales Rodriguez, Thomas Karolyshyn, Giulia Semeghini, Michael J. Gullans, Markus Greiner, Vladan Vuletić, and Mikhail D. Lukin. Logical quantum processor based on reconfigurable atom arrays. *Nature*, 626(7997):58–65, February 2024. ISSN 1476-4687. doi: 10.1038/s41586-023-06927-3.

[12] A.G. Radnaev, W.C. Chung, D.C. Cole, D. Mason, T.G. Ballance, M.J. Bedalov, D.A. Belknap, M.R. Berman, M. Blakely, I.L. Bloomfield, P.D. Buttler, C. Campbell, A. Chopinaud, E. Copenhaver, M.K. Dawes, S.Y. Eubanks, A.J. Friss, D.M. Garcia, J. Gilbert, M. Gillette, P. Goiporia, P. Gokhale, J. Goldwin, D. Goodwin, T.M. Graham, C.J. Guttormsson, G.T. Hickman, L. Hurtley, M. Iliev, E.B. Jones, R.A. Jones, K.W. Kuper, T.B. Lewis, M.T. Lichtman, F. Majdeteimouri, J.J. Mason, J.K. McMaster, J.A. Miles, P.T. Mitchell, J.D. Murphree, N.A. Neff-Mallon, T. Oh, V. Omole, C. Parlo Simon, N. Pederson, M.A. Perlin, A. Reiter, R. Rines, P. Romlow, A.M. Scott, D. Stiefvater, J.R. Tanner, A.K. Tucker, I.V. Vinogradov, M.L. Warter, M. Yeo, M. Saffman, and T.W. Noel. Universal Neutral-Atom Quantum Computer with Individual Optical Addressing and Nondestructive Readout. *PRX Quantum*, 6(3):030334, August 2025. ISSN 2691-3399. doi: 10.1103/66s8-jj18.

[13] M. Veldhorst, H. G. J. Eenink, C. H. Yang, and A. S. Dzurak. Silicon CMOS architecture for a spin-based quantum computer. *Nature Communications*, 8(1):1766, December 2017. ISSN 2041-1723. doi: 10.1038/s41467-017-01905-6.

[14] Jelmer M. Boter. Spiderweb Array: A Sparse Spin-Qubit Array. *Physical Review Applied*, 18(2), 2022. doi: 10.1103/PhysRevApplied.18.024053.

[15] Sara Bartolucci, Patrick Birchall, Hector Bombín, Hugo Cable, Chris Dawson, Mercedes Gimeno-Segovia, Eric Johnston, Konrad Kieling, Naomi Nickerson, Mihir Pant, Fernando Pastawski, Terry Rudolph, and Chris Sparrow. Fusion-based quantum computation. *Nature Communications*, 14 (1):912, February 2023. ISSN 2041-1723. doi: 10.1038/s41467-023-36493-1.

[16] Christopher A. Pattison, Michael E. Beverland, Marcus P. da Silva, and Nicolas Delfosse. Improved quantum error correction using soft information, July 2021.

[17] Maurice D. Hanisch, Bence Hetényi, and James R. Wootton. Soft information decoding with superconducting qubits, November 2024.

[18] Johannes Bausch, Andrew W. Senior, Francisco J. H. Heras, Thomas Edlich, Alex Davies, Michael Newman, Cody Jones, Kevin Satzinger, Murphy Yuezhen Niu, Sam Blackwell, George Holland, Dvir Kafri, Juan Atalaya, Craig Gidney, Demis Hassabis, Sergio Boixo, Hartmut Neven, and Pushmeet Kohli. Learning high-accuracy error decoding for quantum processors. *Nature*, 635(8040):834–840, 2024. doi: 10.1038/s41586-024-08148-8. URL https://doi.org/10.1038/s41586-024-08148-8.

[19] Yutaro Akahoshi, Riki Toshio, Jun Fujisaki, Hirotaka Oshima, Shintaro Sato, and Keisuke Fujii. Runtime reduction in lattice surgery utilizing time-like soft information, October 2025.

[20] Sergey Bravyi, Graeme Smith, and John A. Smolin. Trading Classical and Quantum Computational Resources. *Physical Review X*, 6(2):021043, June 2016. doi: 10.1103/PhysRevX.6.021043.

[21] Dominic Horsman, Austin G. Fowler, Simon Devitt, and Rodney Van Meter. Surface code quantum computing by lattice surgery. *New Journal of Physics*, 14(12):123011, December 2012. ISSN 1367-2630. doi: 10.1088/1367-2630/14/12/123011.

[22] Daniel Litinski. A Game of Surface Codes: Large-Scale Quantum Computing with Lattice Surgery. *Quantum*, 3:128, March 2019. ISSN 2521-327X. doi: 10.22331/q-2019-03-05-128. URL https://doi.org/10.22331/q-2019-03-05-128.

[23] Barbara M. Terhal. Quantum error correction for quantum memories. *Rev. Mod. Phys.*, 87:307–346, Apr 2015. doi: 10.1103/RevModPhys.87.307. URL https://link.aps.org/doi/10.1103/RevModPhys.87.307.

[24] Craig Gidney. How to factor 2048 bit RSA integers with less than a million noisy qubits, May 2025.

[25] Christopher Chamberland and Earl T. Campbell. Universal quantum computing with twist-free and temporally encoded lattice surgery. *PRX Quantum*, 3(1):010331, February 2022. ISSN 2691-3399. doi: 10.1103/PRXQuantum.3.010331.

[26] Christopher Chamberland, Luis Goncalves, Prasahnt Sivarajah, Eric Peterson, and Sebastian Grimberg. Techniques for combining fast local decoders with global decoders under circuit-level noise. *Quantum Science and Technology*, 8(4):045011, jul 2023. doi: 10.1088/2058-9565/ace64d. URL https://doi.org/10.1088/2058-9565/ace64d.

[27] Luka Skoric, Dan E. Browne, Kenton M. Barnes, Neil I. Gillespie, and Earl T. Campbell. Parallel window decoding enables scalable fault tolerant quantum computation. *Nature Communications*, 14(1):7040, November 2023. ISSN 2041-1723. doi: 10.1038/s41467-023-42482-1.

[28] Pavel Panteleev and Gleb Kalachev. Degenerate quantum ldpc codes with good finite length performance. *Quantum*, 5:585, 2021.

[29] Yue Wu and Lin Zhong. Fusion blossom: Fast mwpm decoders for qec. In *2023 IEEE International Conference on Quantum Computing and Engineering (QCE)*, volume 1, pages 928–938. IEEE, 2023.

[30] Oscar Higgott and Craig Gidney. Sparse blossom: correcting a million errors per core second with minimum-weight matching. *Quantum*, 9:1600, 2025.

[31] Tim Chan. Snowflake: A Distributed Streaming Decoder, December 2024.

[32] Timo Hillmann, Lucas Berent, Armanda O. Quintavalle, Jens Eisert, Robert Wille, and Joschka Roffe. Localized statistics decoding: A parallel decoding algorithm for quantum low-density parity-check codes, June 2024.

[33] Abbas B. Ziad, Ankit Zalawadiya, Canberk Topal, Joan Camps, György P. Gehér, Matthew P. Stafford, and Mark L. Turner. Local Clustering Decoder: A fast and adaptive hardware decoder for the surface code, November 2024.

[34] Anqi Gong, Sebastian Cammerer, and Joseph M. Renes. Toward Low-latency Iterative Decoding of QLDPC Codes Under Circuit-Level Noise, March 2024.

[35] Laleh Aghababaie Beni, Oscar Higgott, and Noah Shutty. Tesseract: A search-based decoder for quantum error correction, 2025. URL https://arxiv.org/abs/2503.10988.

[36] Tristan Müller, Thomas Alexander, Michael E. Beverland, Markus Bühler, Blake R. Johnson, Thilo Maurer, and Drew Vandeth. Improved belief propagation is sufficient for real-time decoding of quantum memory. *arXiv preprint arXiv:2506.01779*, 2025. IBM Quantum.

[37] Ming Wang, Ang Li, and Frank Mueller. Fully Parallelized BP Decoding for Quantum LDPC Codes Can Outperform BP-OSD, July 2025.

[38] Stasiu Wolanski and Ben Barber. Ambiguity Clustering: An accurate and efficient decoder for qLDPC codes, January 2025.

[39] Kai Zhang, Jubo Xu, Fang Zhang, Linghang Kong, Zhengfeng Ji, and Jianxin Chen. LATTE: A Decoding Architecture for Quantum Computing with Temporal and Spatial Scalability, September 2025.

[40] Mark L. Turner, Earl T. Campbell, Ophelia Crawford, Neil I. Gillespie, and Joan Camps. Scalable decoding protocols for fast transversal logic in the surface code, May 2025.

[41] Ben Barber, Kenton M. Barnes, Tomasz Bialas, Okan Buğdaycı, Earl T. Campbell, Neil I. Gillespie, Kauser Johar, Ram Rajan, Adam W. Richardson, Luka Skoric, Canberk Topal, Mark L. Turner, and Abbas B. Ziad. A real-time, scalable, fast and highly resource efficient decoder for a quantum computer, January 2025.

[42] John Blue, Harshil Avlani, Zhiyang He, Liu Ziyin, and Isaac L. Chuang. Machine Learning Decoding of Circuit-Level Noise for Bivariate Bicycle Codes, April 2025.

[43] Moritz Lange. Data-driven decoding of quantum error correcting codes using graph neural networks. *Physical Review Research*, 7(2), 2025. doi: 10.1103/PhysRevResearch.7.023181.

[44] Xiangjun Mi and Frank Mueller. Toward Uncertainty-Aware and Generalizable Neural Decoding for Quantum LDPC Codes, October 2025.

[45] Gengyuan Hu, Wanli Ouyang, Chao-Yang Lu, Chen Lin, and Han-Sen Zhong. Efficient and Universal Neural-Network Decoder for Stabilizer-Based Quantum Error Correction, February 2025.

[46] Hanyan Cao, Feng Pan, Yijia Wang, and Pan Zhang. qecGPT: Decoding Quantum Error-correcting Codes with Generative Pre-trained Transformers, July 2023.

[47] Zejun Liu, Anqi Gong, and Bryan K. Clark. Decoding quantum low density parity check codes with diffusion, September 2025.

[48] Boris M. Varbanov, Marc Serra-Peralta, David Byfield, and Barbara M. Terhal. Neural network decoder for near-term surface-code experiments. *Physical Review Research*, 7(1):013029, January 2025. doi: 10.1103/PhysRevResearch.7.013029.

[49] Spiro Gicev, Lloyd C. L. Hollenberg, and Muhammad Usman. Fully convolutional 3D neural network decoders for surface codes with syndrome circuit noise. *arXiv e-prints*, art. arXiv:2506.16113, June 2025. doi: 10.48550/arXiv.2506.16113. URL https://arxiv.org/abs/2506.16113.

[50] C. Ryan-Anderson, J. G. Bohnet, K. Lee, D. Gresh, A. Hankin, J. P. Gaebler, D. Francois, A. Chernoguzov, D. Lucchetti, N. C. Brown, T. M. Gatterman, S. K. Halit, K. Gilmore, J. A. Gerber, B. Neyenhuis, D. Hayes, and R. P. Stutz. Realization of Real-Time Fault-Tolerant Quantum Error Correction. *Physical Review X*, 11(4):041058, December 2021. ISSN 2160-3308. doi: 10.1103/PhysRevX.11.041058.

[51] M. P. da Silva, C. Ryan-Anderson, J. M. Bello-Rivas, A. Chernoguzov, J. M. Dreiling, C. Foltz, J. P. Gaebler, T. M. Gatterman, D. Hayes, N. Hewitt, J. Johansen, D. Lucchetti, M. Mills, S. A. Moses, B. Neyenhuis, A. Paz, J. Pino, P. Siegfried, J. Strabley, S. J. Wernli, R. P. Stutz, and K. M. Svore. Demonstration of logical qubits and repeated error correction with better-than-physical error rates, April 2024.

[52] Shival Dasu, Simon Burton, Karl Mayer, David Amaro, Justin A. Gerber, Kevin Gilmore, Dan Gresh, Davide DelVento, Andrew C. Potter, and David Hayes. Breaking even with magic: Demonstration of a high-fidelity logical non-Clifford gate, June 2025.

[53] Kentaro Yamamoto, Yuta Kikuchi, David Amaro, Ben Criger, Silas Dilkes, Ciarán Ryan-Anderson, Andrew Tranter, Joan M. Dreiling, Dan Gresh, Cameron Foltz, Michael Mills, Steven A. Moses, Peter E. Siegfried, Maxwell D. Urmey, Justin J. Burau, Aaron Hankin, Dominic Lucchetti, John P. Gaebler, Natalie C. Brown, Brian Neyenhuis, and David Muñoz Ramo. Quantum Error-Corrected Computation of Molecular Energies, May 2025.

[54] Harald Putterman, Kyungjoo Noh, Connor T. Hann, Gregory S. MacCabe, Shahriar Aghaeimeibodi, Rishi N. Patel, Menyoung Lee, William M. Jones, Hesam Moradinejad, Roberto Rodriguez, Neha Mahuli, Jefferson Rose, John Clai Owens, Harry Levine, Emma Rosenfeld, Philip Reinhold, Lorenzo Moncelsi, Joshua Ari Alcid, Nasser Alidoust, Patricio Arrangoiz-Arriola, James Barnett, Przemyslaw Bienias, Hugh A. Carson, Cliff Chen, Li Chen, Harutiun Chinkezian, Eric M. Chisholm, Ming-Han Chou, Aashish Clerk, Andrew Clifford, R. Cosmic, Ana Valdes Curiel, Erik Davis, Laura DeLorenzo, J. Mitchell D'Ewart, Art Diky, Nathan D'Souza, Philipp T. Dumitrescu, Shmuel Eisenmann, Essam Elkhouly, Glen Evenbly, Michael T. Fang, Yawen Fang, Matthew J. Fling, Warren Fon, Gabriel Garcia, Alexey V. Gorshkov, Julia A. Grant, Mason J. Gray, Sebastian Grimberg, Arne L. Grimsmo, Arbel Haim, Justin Hand, Yuan He, Mike Hernandez, David Hover, Jimmy S. C. Hung, Matthew Hunt, Joe Iverson, Ignace Jarrige, Jean-Christophe Jaskula, Liang Jiang, Mahmoud Kalaee, Rassul Karabalin, Peter J. Karalekas, Andrew J. Keller, Amirhossein Khalajhedayati, Aleksander Kubica, Hanho Lee, Catherine Leroux, Simon Lieu, Victor Ly, Keven Villegas Madrigal, Guillaume Marcaud, Gavin McCabe, Cody Miles, Ashley Milsted, Joaquin Minguzzi, Anurag Mishra, Biswaroop Mukherjee, Mahdi Naghiloo, Eric Oblepias, Gerson Ortuno, Jason Pagdilao, Nicola Pancotti, Ashley Panduro, J. P. Paquette, Minje Park, Gregory A. Peairs, David Perello, Eric C. Peterson, Sophia Ponte, John Preskill, Johnson Qiao, Gil Refael, Rachel Resnick, Alex Retzker, Omar A. Reyna, Marc Runyan, Colm A. Ryan, Abdulrahman Sahmoud, Ernesto Sanchez, Rohan Sanil, Krishanu Sankar, Yuki Sato, Thomas Scaffidi, Salome Siavoshi, Prasahnt Sivarajah, Trenton Skogland, Chun-Ju Su, Loren J. Swenson, Stephanie M. Teo, Astrid Tomada, Giacomo Torlai, E. Alex Wollack, Yufeng Ye, Jessica A. Zerrudo, Kailing Zhang, Fernando G. S. L. Brandão, Matthew H. Matheny, and Oskar Painter. Hardware-efficient quantum error correction via concatenated bosonic qubits. *Nature*, 638(8052):927–934, February 2025. ISSN 1476-4687. doi: 10.1038/s41586-025-08642-7.

[55] Matt J. Bedalov, Matt Blakely, Peter D. Buttler, Caitlin Carnahan, Frederic T. Chong, Woo Chang Chung, Dan C. Cole, Palash Goiporia, Pranav Gokhale, Bettina Heim, Garrett T. Hickman, Eric B. Jones, Ryan A. Jones, Pradnya Khalate, Jin-Sung Kim, Kevin W. Kuper, Martin T. Lichtman, Stephanie Lee, David Mason, Nathan A. Neff-Mallon, Thomas W. Noel, Victory Omole, Alexander G. Radnaev, Rich Rines, Mark Saffman, Efrat Shabtai, Mariesa H. Teo, Bharath Thotakura, Teague Tomesh, and Angela K. Tucker. Fault-Tolerant Operation and Materials Science with Neutral Atom Logical Qubits, December 2024.

[56] Rich Rines, Benjamin Hall, Mariesa H. Teo, Joshua Viszlai, Daniel C. Cole, David Mason, Cameron Barker, Matt J. Bedalov, Matt Blakely, Tobias Bothwell, Caitlin Carnahan, Frederic T. Chong, Samuel Y. Eubanks, Brian Fields, Matthew Gillette, Palash Goiporia, Pranav Gokhale, Garrett T. Hickman, Marin Iliev, Eric B. Jones, Ryan A. Jones, Kevin W. Kuper, Stephanie Lee, Martin T. Lichtman, Kevin Loeffler, Nate Mackintosh, Farhad Majdeteimouri, Peter T. Mitchell, Thomas W. Noel, Ely Novakoski, Victory Omole, David Owusu-Antwi, Alexander G. Radnaev, Anthony Reiter, Mark Saffman, Bharath Thotakura, Teague Tomesh, and Ilya Vinogradov. Demonstration of a Logical Architecture Uniting Motion and In-Place Entanglement: Shor's Algorithm, Constant-Depth CNOT Ladder, and Many-Hypercube Code, September 2025.

[57] Ben W. Reichardt, Adam Paetznick, David Aasen, Ivan Basov, Juan M. Bello-Rivas, Parsa Bonderson, Rui Chao, Wim van Dam, Matthew B. Hastings, Ryan V. Mishmash, Andres Paz, Marcus P. da Silva, Aarthi Sundaram, Krysta M. Svore, Alexander Vaschillo, Zhenghan Wang, Matt Zanner, William B. Cairncross, Cheng-An Chen, Daniel Crow, Hyosub Kim, Jonathan M. Kindem, Jonathan King, Michael McDonald, Matthew A. Norcia, Albert Ryou, Mark Stone, Laura Wadleigh, Katrina Barnes, Peter Battaglino, Thomas C. Bohdanowicz, Graham Booth, Andrew Brown, Mark O.

Brown, Kayleigh Cassella, Robin Coxe, Jeffrey M. Epstein, Max Feldkamp, Christopher Griger, Eli Halperin, Andre Heinz, Frederic Hummel, Matthew Jaffe, Antonia M. W. Jones, Eliot Kapit, Krish Kotru, Joseph Lauigan, Ming Li, Jan Marjanovic, Eli Megidish, Matthew Meredith, Ryan Morshead, Juan A. Muniz, Sandeep Narayanaswami, Ciro Nishiguchi, Timothy Paule, Kelly A. Pawlak, Kristen L. Pudenz, David Rodríguez Pérez, Jon Simon, Aaron Smull, Daniel Stack, Miroslav Urbanek, René J. M. van de Veerdonk, Zachary Vendeiro, Robert T. Weverka, Thomas Wilkason, Tsung-Yao Wu, Xin Xie, Evan Zalys-Geller, Xiaogang Zhang, and Benjamin J. Bloom. Fault-tolerant quantum computation with a neutral atom processor, June 2025.

[58] J. A. Muniz, D. Crow, H. Kim, J. M. Kindem, W. B. Cairncross, A. Ryou, T. C. Bohdanowicz, C.-A. Chen, Y. Ji, A. M. W. Jones, E. Megidish, C. Nishiguchi, M. Urbanek, L. Wadleigh, T. Wilkason, D. Aasen, K. Barnes, J. M. Bello-Rivas, I. Bloomfield, G. Booth, A. Brown, M. O. Brown, K. Cassella, G. Cowan, J. Epstein, M. Feldkamp, C. Griger, Y. Hassan, A. Heinz, E. Halperin, T. Hofler, F. Hummel, M. Jaffe, E. Kapit, K. Kotru, J. Lauigan, J. Marjanovic, M. Meredith, M. McDonald, R. Morshead, S. Narayanaswami, K. A. Pawlak, K. L. Pudenz, D. Rodríguez Pérez, P. Sabharwal, J. Simon, A. Smull, M. Sorensen, D. T. Stack, M. Stone, L. Taneja, R. J. M. van de Veerdonk, Z. Vendeiro, R. T. Weverka, K. White, T.-Y. Wu, X. Xie, E. Zalys-Geller, X. Zhang, J. King, B. J. Bloom, and M. A. Norcia. Repeated ancilla reuse for logical computation on a neutral atom quantum computer, June 2025.

[59] Laura Caune, Luka Skoric, Nick S. Blunt, Archibald Ruban, Jimmy McDaniel, Joseph A. Valery, Andrew D. Patterson, Alexander V. Gramolin, Joonas Majaniemi, Kenton M. Barnes, Tomasz Bialas, Okan Buğdaycı, Ophelia Crawford, György P. Gehér, Hari Krovi, Elisha Matekole, Canberk Topal, Stefano Poletto, Michael Bryant, Kalan Snyder, Neil I. Gillespie, Glenn Jones, Kauser Johar, Earl T. Campbell, and Alexander D. Hill. Demonstrating real-time and low-latency quantum error correction with superconducting qubits, October 2024.

[60] Markus Bühler, Michael E. Beverland, Frank Haverkamp, Thilo Maurer, Drew Vandeth, and Matthew Walther. Real-time decoding of the gross code memory with fpgas. *arXiv preprint arXiv:2510.21600*, 2025. IBM Quantum, manuscript in preparation.

[61] Francesco Battistel, Christopher Chamberland, Kauser Johar, Ramon W. J. Overwater, Fabio Sebastiano, Luka Skoric, Yosuke Ueno, and Muhammad Usman. Real-Time Decoding for Fault-Tolerant Quantum Computing: Progress, Challenges and Outlook. *Nano Futures*, 7(3):032003, September 2023. ISSN 2399-1984. doi: 10.1088/2399-1984/aceba6.

[62] Yaniv Kurman, Lior Ella, Nir Halay, Oded Wertheim, and Yonatan Cohen. Controller-decoder system requirements derived by implementing Shor's algorithm with surface code, November 2024.

[63] Yaniv Kurman, Lior Ella, Ramon Szmuk, Oded Wertheim, Benedikt Dorschner, Sam Stanwyck, and Yonatan Cohen. Benchmarking the ability of a controller to execute quantum error corrected non-Clifford circuits, November 2024.

[64] Riverlane. Gpus, asics or fpgas? here's how they measure up for quantum error correction. https://www.riverlane.com/blog/gpus-asics-or-fpgas-here-s-how-they-measure-up-for-quantum-error-correction, 2024. Riverlane Blog.

[65] Namitha Liyanage, Yue Wu, Siona Tagare, and Lin Zhong. Fpga-based distributed union-find decoder for surface codes. *IEEE Transactions on Quantum Engineering*, 2024.

[66] Xinyu Tan, Fang Zhang, Rui Chao, Yaoyun Shi, and Jianxin Chen. Scalable Surface-Code Decoders with Parallelization in Time. *PRX Quantum*, 4 (4):040344, December 2023. doi: 10.1103/PRXQuantum.4.040344. URL https://arxiv.org/abs/2209.09219.

[67] Sergey Bravyi, Andrew W Cross, Jay M Gambetta, Dmitri Maslov, Patrick Rall, and Theodore J Yoder. High-threshold and low-overhead fault-tolerant quantum memory. *Nature*, 627(8005):778–782, 2024.

[68] Peter-Jan H. S. Derks, Alex Townsend-Teague, Ansgar G. Burchards, and Jens Eisert. Designing fault-tolerant circuits using detector error models, December 2024.

[69] Open qhpc software ecosystem project.

[70] Yilun Xu, Gang Huang, Jan Balewski, Ravi Naik, Alexis Morvan, Bradley Mitchell, Kasra Nowrouzi, David I Santiago, and Irfan Siddiqi. QubiC: An open-source fpga-based control and measurement system for superconducting quantum information processors. *IEEE Transactions on Quantum Engineering*, 2:1–11, 2021.

[71] Yilun Xu, Gang Huang, Neelay Fruitwala, Abhi Rajagopala, Ravi K Naik, Kasra Nowrouzi, David I Santiago, and Irfan Siddiqi. QubiC 2.0: An extensible open-source qubit control system capable of mid-circuit measurement and feed-forward. *arXiv preprint arXiv:2309.10333*, 2023.

[72] C.H. Liu. Single Flux Quantum-Based Digital Control of Superconducting Qubits in a Multichip Module. *PRX Quantum*, 4(3), 2023. doi: 10.1103/PRXQuantum.4.030310.

[73] Devin L. Underwood, Joseph A. Glick, Ken Inoue, David J. Frank, John Timmerwilke, Emily Pritchett, Sudipto Chakraborty, Kevin Tien, Mark Yeck, John F. Bulzacchelli, Chris Baks, Pat Rosno, Raphael Robertazzi, Matthew Beck, Rajiv V. Joshi, Dorothy Wisnieff, Daniel Ramirez, Jeff Ruedinger, Scott Lekuch, Brian P. Gaucher, and Daniel J. Friedman. Using Cryogenic CMOS Control Electronics To Enable A Two-Qubit Cross-Resonance Gate, December 2023.

[74] Samuel K. Bartee, Will Gilbert, Kun Zuo, Kushal Das, Tuomo Tanttu, Chih Hwan Yang, Nard Dumoulin Stuyck, Sebastian J. Pauka, Rocky Y. Su, Wee Han Lim, Santiago Serrano, Christopher C. Escott, Fay E. Hudson, Kohei M. Itoh, Arne Laucht, Andrew S. Dzurak, and David J. Reilly. Spin-qubit control with a milli-kelvin CMOS chip. *Nature*, 643(8071):382–387, July 2025. ISSN 1476-4687. doi: 10.1038/s41586-025-09157-x.

[75] NVIDIA Corporation. CUDA C++ Programming Guide: 5.4 Heterogeneous Programming, 2025. URL https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html?highlight=device#heterogeneous-programming. Accessed: 2025-08-19.

[76] Doca gpunetio.

[77] Nvidia holoscan sensor bridge.

[78] Sergey Bravyi and Alexei Kitaev. Universal quantum computation with ideal clifford gates and noisy ancillas. *Phys. Rev. A*, 71:022316, Feb 2005. doi: 10.1103/PhysRevA.71.022316. URL https://link.aps.org/doi/10.1103/PhysRevA.71.022316.

[79] Craig Gidney, Noah Shutty, and Cody Jones. Magic state cultivation: growing t states as cheap as cnot gates, 2024. URL https://arxiv.org/abs/2409.17595.

[80] Austin G. Fowler and Craig Gidney. Low overhead quantum computation using lattice surgery. *arXiv e-prints*, art. arXiv:1808.06709, August 2018. doi: 10.48550/arXiv.1808.06709. URL https://arxiv.org/abs/1808.06709.

[81] Christopher Chamberland and Earl T. Campbell. Circuit-level protocol and analysis for twist-based lattice surgery. *Phys. Rev. Res.*, 4:023090, May 2022. doi: 10.1103/PhysRevResearch.4.023090. URL https://link.aps.org/doi/10.1103/PhysRevResearch.4.023090.

[82] Christopher Chamberland, Pavithran Iyer, and David Poulin. Fault-tolerant quantum computing in the Pauli or Clifford frame with slow error diagnostics. *Quantum*, 2:43, January 2018. ISSN 2521-327X. doi: 10.22331/q-2018-01-04-43. URL https://doi.org/10.22331/q-2018-01-04-43.

[83] Jan Olle, Christopher Chamberland, Muyuan Li, and Scott Thornton. AI-based pre-decoder for accelerating quantum error correction. *Work in preparation*.

[84] Julian Kelly, Peter O'Malley, Matthew Neeley, Hartmut Neven, and John M. Martinis. Physical qubit calibration on a directed acyclic graph, 2018. URL https://arxiv.org/abs/1803.03226.

[85] Erik Nielsen, John King Gamble, Kenneth Rudinger, Travis Scholten, Kevin Young, and Robin Blume-Kohout. Gate set tomography. *Quantum*, 5: 557, 2021.

[86] E. Knill, D. Leibfried, R. Reichle, J. Britton, R. B. Blakestad, J. D. Jost, C. Langer, R. Ozeri, S. Seidelin, and D. J. Wineland. Randomized benchmarking of quantum gates. *Phys. Rev. A*, 77:012307, Jan 2008. doi: 10.1103/PhysRevA.77.012307. URL https://link.aps.org/doi/10.1103/PhysRevA.77.012307.

[87] Easwar Magesan, J. M. Gambetta, and Joseph Emerson. Scalable and robust randomized benchmarking of quantum processes. *Phys. Rev. Lett.*, 106: 180504, May 2011. doi: 10.1103/PhysRevLett.106.180504. URL https://link.aps.org/doi/10.1103/PhysRevLett.106.180504.

[88] Nathan Eli Miller, Biswadeep Chakraborty, and Saibal Mukhopadhyay. A reconfigurable quantum state tomography solver in fpga. In *2023 IEEE International Conference on Quantum Computing and Engineering (QCE)*, volume 01, pages 1412–1421, 2023. doi: 10.1109/QCE57702.2023.00160.

[89] Joel J Wallman, Marie Barnhill, and Joseph Emerson. Robust characterization of leakage errors. *New Journal of Physics*, 18(4):043021, April 2016. ISSN 1367-2630. doi: 10.1088/1367-2630/18/4/043021. URL http://dx.doi.org/10.1088/1367-2630/18/4/043021.

[90] Alicia B. Magann, Nathan E. Miller, Robin Blume-Kohout, Peter Maunz, and Kevin C. Young. In preparation: Single- and few-shot protocols for quantum device calibration and drift mitigation, 2025.

[91] Nathan Eli Miller, Laith A. Shamieh, and Saibal Mukhopadhyay. Low-latency digital feedback for stochastic quantum calibration using cryogenic cmos. In *2025 Design, Automation & Test in Europe Conference (DATE)*, pages 1–7, 2025. doi: 10.23919/DATE64628.2025.10992779.

[92] Joel J Wallman and Joseph Emerson. Noise tailoring for scalable quantum computation via randomized compiling. *Physical Review A*, 94(5):052325, 2016.

[93] Timothy Proctor, Melissa Revelle, Erik Nielsen, Kenneth Rudinger, Daniel Lobser, Peter Maunz, Robin Blume-Kohout, and Kevin Young. Detecting and tracking drift in quantum information processors. *Nature communications*, 11(1):5396, 2020.

[94] JP Marceaux and Kevin Young. Streaming quantum gate set tomography using the extended kalman filter. In *2023 IEEE International Conference on Quantum Computing and Engineering (QCE)*, volume 1, pages 1401–1411. IEEE, 2023.

[95] Kenneth Rudinger, JP Marceaux, Akel Hashim, David I Santiago, Irfan Siddiqi, and Kevin C Young. Heisenberg-limited calibration of entangling gates with robust phase estimation. *arXiv preprint arXiv:2502.06698*, 2025.

[96] The CUDA-Q development team. CUDA-Q. https://github.com/NVIDIA/cuda-quantum, 2025. URL https://github.com/NVIDIA/cuda-quantum. Available at https://github.com/NVIDIA/cuda-quantum.

[97] The cuQuantum Development Team. Nvidia cuquantum sdk. https://github.com/NVIDIA/cuQuantum, 2024. URL https://github.com/NVIDIA/cuQuantum. Available at https://github.com/NVIDIA/cuQuantum.

[98] NVIDIA Corporation, CUDA-QX Development Team. CUDA-QX. https://github.com/NVIDIA/cudaqx, 2025. URL https://nvidia.github.io/cudaqx/. Available at https://nvidia.github.io/cudaqx/.

[99] Jun Ye and Jun Yong Khoo. Emuplat: A framework-agnostic platform for quantum hardware emulation with validated transpiler-to-pulse pipeline, 2025. URL https://arxiv.org/abs/2509.12639.

[100] Craig Gidney. Stim: a fast stabilizer circuit simulator. *Quantum*, 5:497, July 2021. ISSN 2521-327X. doi: 10.22331/q-2021-07-06-497. URL https://doi.org/10.22331/q-2021-07-06-497.