A Full Stack Framework for High Performance Quantum-Classical Computing

Xin Zhan
Hewlett Packard Enterprise
Houston, USA
xin.zhan@hpe.com

K. Grace Johnson
Hewlett Packard Enterprise
Milpitas, USA
grace.johnson@hpe.com

Aniello Esposito
Hewlett Packard Enterprise
Basel, Switzerland
aniello.esposito@hpe.com

Soumitra Chatterjee
Hewlett Packard Enterprise
Bengaluru, India
soumitra@hpe.com

Barbara Chapman
Hewlett Packard Enterprise
New York, USA
barbara.chapman@hpe.com

Marco Fiorentino
Hewlett Packard Enterprise
Milpitas, USA
marco.fiorentino@hpe.com

Kirk M. Bresniker
Hewlett Packard Enterprise
Milpitas, USA
kirk.bresniker@hpe.com

Raymond G. Beausoleil
Hewlett Packard Enterprise
Milpitas, USA
ray.beausoleil@hpe.com

Masoud Mohseni
Hewlett Packard Enterprise
Milpitas, USA
masoud.mohseni@hpe.com

Abstract

To address the growing needs for scalable High Performance Computing (HPC) and Quantum Computing (QC) integration, we present our HPC-QC full stack framework and its hybrid workload development capability with modular hardware/device-agnostic software integration approach. The latest development in extensible interfaces for quantum programming, dispatching, and compilation within existing mature HPC programming environment are demonstrated. Our HPC-QC full stack enables high-level, portable invocation of quantum kernels from commercial quantum SDKs within HPC meta-program in compiled languages (C/C++ and Fortran) as well as Python through a quantum programming interface library extension. An adaptive circuit knitting hypervisor is being developed to partition large quantum circuits into sub-circuits that fit on smaller noisy quantum devices and classical simulators. At the lower-level, we leverage Cray LLVM-based compilation framework to transform and consume LLVM IR and Quantum IR (QIR) from commercial quantum software front-ends in a retargetable fashion to different hardware architectures. Several hybrid HPC-QC multi-node multi-CPU and GPU workloads (including solving linear system of equations, quantum optimization, and simulating quantum phase transitions) have been demonstrated on HPE EX supercomputers to illustrate functionality and execution viability for all three components developed so far. This work provides the framework for a unified quantum-classical programming environment built upon classical HPC software stack (compilers, libraries, parallel runtime and process scheduling).

CUG 2025 | ©HPE , New York, NY

ACM ISBN 978-1-4503-XXXX-X/2025/06
https://doi.org/XXXXXXX.XXXXXXX

CCS Concepts

• **Software and its engineering** → **Development frameworks and environments**; • **Computing methodologies** → *Distributed computing methodologies*; • **Theory of computation** → Quantum computation theory.

Keywords

quantum-HPC integration, heterogeneous systems, distributed programming models, quantum computing, hybrid quantum-classical programming,

ACM Reference Format:
Xin Zhan, K. Grace Johnson, Aniello Esposito, Soumitra Chatterjee, Barbara Chapman, Marco Fiorentino, Kirk M. Bresniker, Raymond G. Beausoleil, and Masoud Mohseni. 2025. A Full Stack Framework for High Performance Quantum-Classical Computing . In *Proceedings of Make sure to enter the correct conference title from your rights confirmation email (CUG 2025 | ©HPE)*. ACM, New York, NY, USA, 7 pages. https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

A hybrid HPC-QC computation paradigm, where quantum processing units (QPUs) are integrated into heterogeneous HPC infrastructures as additional accelerators besides GPUs and FPGAs, will maximize the optimal utilization of both paradigms for massive parallel processing and enable quantum computing to operate at scale [10]. The synergy between HPC and QC provides a unique advantage to tackle challenges such as complex memory requirements for QC and computationally intractable problems in HPC that neither paradigm can efficiently solve on its own. Due to potential exponential reductions in time or space complexity for certain problems, QC has attracted significant interest as an alternative computational model for performance beyond exascale. The desire for HPC-QC hybrid systems provides natural and essential demands for integrating quantum programming capability (to program, compile, distribute, and execute quantum circuits) into existing classical HPC programming environments.

A robust quantum classical full stack solution is the key to enable scalable, distributed, and hybrid HPC-QC workload development. With diverse quantum hardware such as superconducting qubits, trapped ions, neutral atoms, and photonic qubits [4, 12], various quantum software packages focus on different aspects of quantum computing, e.g. circuit synthesis or optimizing complex design processes including qubit allocation, auxiliary qubit reuse, error mitigation, and quantum error correction. The majority of these software packages adopt declarative frameworks—Python-based Domain-Specific-Languages (DSL)—which facilitate fast experimentation and have built-in REST client support [11]. When scaling to large circuit sizes with ~100-qubits or greater, declarative frameworks struggle to handle compilation bottlenecks and latency between classical and quantum components in the program, even in the current era of Noisy Intermediate Scale Quantum (NISQ) systems [16]. Leveraging classical compilation tool chains such as Clang/LLVM is crucial for developing a large-scale hybrid HPC-QC workload with research efforts moving in this direction [5, 13]. In addition, the vast majority of science and math libraries that support massively parallel distributed computing used to accelerate large-scale classical HPC applications adopts compiled languages such as C, C++, and Fortran for optimal performance. Hence, integrating quantum kernels within HPC applications will need to take this into consideration. For scalable quantum computing in both the NISQ era (to study systems larger than ~100-qubits) and future fault-tolerant era (to parallelize quantum error correction), efficient partitioning and distribution of large quantum workloads across QPUs for parallel execution is needed. With a good understanding of these challenges and the diversity of current quantum software landscape, we are addressing three main aspects in our design: comparability, performance and scalability.

2 Methodology

We adopt a modular hardware/device-agnostic approach towards the development of a full quantum-classical stack. Within HPC programming environment, we decompose the QC extension into three tracks: *A.*) quantum application programming interface library extension, *B.*) adaptive circuit knitting hypervisor for quantum workload distribution, and *C.*) quantum compilation and runtime extension. Figure 1 demonstrates the hierarchical architecture diagram of proposed quantum-classical full stack solution with all three components highlighted.

At the highest level is hybrid application development, including but not limited to quantum many body physics and chemistry simulations, quantum machine learning, and optimization. Hybrid HPC-QC applications of varying algorithmic complexity and scale covering these different areas are developed to demonstrate functionality and validate execution viability for all three extensions.

To enable seamless invocation of quantum kernels from vendor-specific quantum SDKs within HPC applications developed in C/C++ and Fortran, we developed a quantum interface library. Circuit simulation is replaced with Quantum API calls. With this approach, minimal changes to the programming and compilation of massively parallel classical HPC applications (combined with science and math libraries) are needed for application developers, facilitating experimentation with different quantum SDKs.

In parallel with the quantum API extension, a novel adaptive circuit knitting (ACK) toolkit is being developed as a hypervisor for quantum workload distribution over multi-QPU systems. The ACK algorithm finds efficient partitions of quantum circuits as they evolve by cutting the circuits (in space and time) in locations that minimize entanglement between partitions, overcoming the exponential classical post-processing of standard circuit knitting. It is utilized to manage classical and quantum communication between compute nodes tasked with sub-circuit execution and measurement reconstruction, dispatching large-scale quantum system simulation or quantum machine learning workloads onto smaller QPUs for parallel execution.

At the lower level, quantum compiler and runtime extensions are being developed to enable performant language-level support for quantum constructs and to address the bottlenecks in compile-time with increasing circuit size.

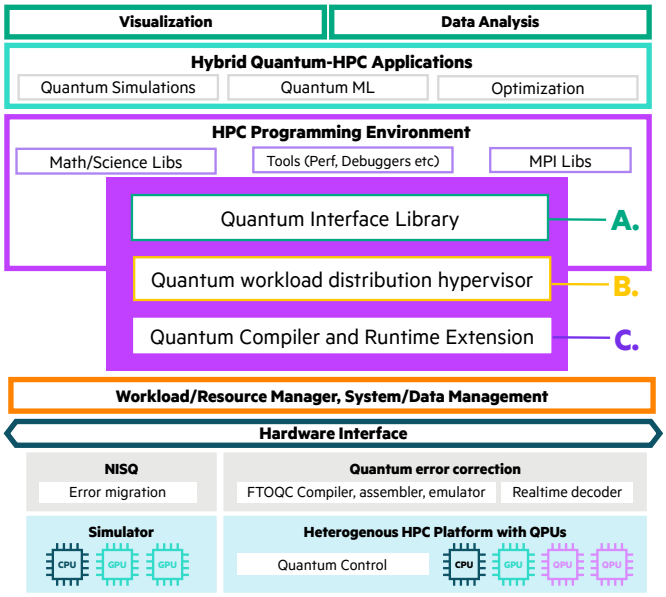

Figure 1: Quantum-classical full stack architecture with extensions for a quantum API, adaptive circuit knitting hypervisor, and quantum compilation and runtime within HPC Programming Environment.

2.1 Quantum Interface Library

The quantum interface library acts as a bridge between traditional HPC applications developed in C/C++ and Fortran and advanced quantum algorithms provided in Python-based DSL, facilitating the integration of quantum computing within existing classical workloads. It provides callable Application Programming Interfaces (APIs) that allow users to access popular quantum algorithms implemented by various quantum SDKs with different programming models, ensuring that HPC applications can leverage quantum resources without requiring fundamental changes to their existing codebase.

2.1.1 Design of Quantum interface library. Quantum Interface Library is designed to expose a set of generic, high-level APIs that abstract away the specifics of quantum SDKs. This design enables the main application logic to invoke quantum kernels without being tightly coupled to the underlying SDK-specific interfaces, functions, or runtime implementations. It also enables hybrid workloads to be submitted to a variety of quantum hardware platforms and simulators through supported SDKs. Different SDKs may interface with multiple types of physical QPUs, such as superconducting qubits, trapped ions, etc.

This design ensures portability across heterogeneous quantum platforms by allowing a single quantum-classical application to execute on different quantum devices with minimal or no code changes. It also promotes extensibility, enabling the framework to integrate emerging quantum SDKs and backends with minimal disruption to the user-facing programming model. By leveraging these abstractions, the current framework can support rapid prototyping and deployment of hybrid quantum-classical workloads, while remaining adaptable to the evolving quantum software and hardware landscape.

The interface library itself is implemented in C, providing low-level, efficient, and stable APIs that are compatible with C/C++ and Fortran applications. Rather than embedding the quantum logic directly into the library, the interface library routes quantum API calls to/from quantum SDKs within the application meta-program, with appropriate data handling and transfer during runtime. HPC applications can continue to be built with classical CPE C/C++ and Fortran compilers and linked with other libraries such as Cray Science and Math Libraries (CSML)[2] as well as the quantum interface library, as illustrated in Figure 2. Dynamic linking allows quantum SDKs to be updated independently without recompiling the interface library or hybrid application. The interface library calls vendor-specific quantum SDK routines to compile quantum circuits into standard quantum assembly languages such as OpenQASM [6] and subsequently execute on various supported gate-based quantum hardware and simulators hosted remotely on cloud or on-premise.

A hybrid MPI execution model is adopted wherein classical computation and quantum simulation are executed as separate MPI processes but communicate via MPI messaging. The classical process is responsible for handling classical computations, setting up and invoking the quantum algorithm, and sending inputs to and receiving outputs from the quantum simulation. Quantum process is responsible for circuit synthesis and execution. Using MPI to separate these tasks, as shown in Figure 3, allows:

- Efficient parallel execution for performance: Quantum processing can be offloaded to the simulator or hardware device asynchronously.
- Scalability: Classical and quantum processes can be expanded independently across multiple CPU, GPU, and QPU nodes.
- Flexibility and modularity: Keeping the classical and quantum parts separate allows easier debugging and swapping out different quantum backends.

Developed quantum programming interface library is utilized to deliver two MPI enabled hybrid HPC-QC applications in Python and C/C++. Quantum kernels for circuit synthesis and execution are provided by Classiq's Python based quantum SDK [1].

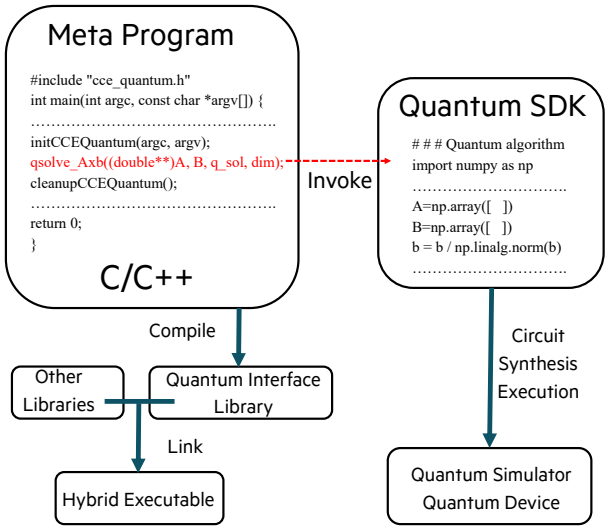

Figure 2: Quantum interface library replaces circuit simulation with API calls to enable invocation of quantum kernel from classical HPC program.

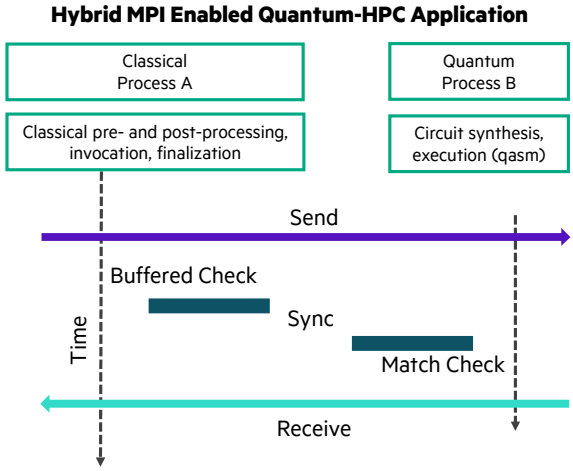

Figure 3: Hybrid MPI enabled Quantum-HPC application scheme.

2.1.2 Case A: Quantum linear solver. The first use case is to solve linear systems of equations with the Harrow-Hassidim-Lloyd (HHL) algorithm[9]. HHL algorithm is a basic quantum algorithm for solving a set of linear equations: $A\vec{x} = \vec{b}$, represented by an $N \times N$ matrix A and a vector $\vec{b}$ of size $N = 2^n$. The solution to the problem is designated by variable $\vec{x}$. In the quantum setting, we reformulate this task by encoding $\vec{b}$ as a normalized quantum state $|b\rangle \in \mathbb{C}^N$, and seek to prepare a quantum state $|x\rangle \propto A^{-1}|b\rangle$ that is proportional to the classical solution $\vec{x}$, i.e.,

$$|x\rangle \propto A^{-1}|b\rangle.$$

The workflow for implementing HHL involves classical pre-processing, quantum circuit synthesis, execution, and result validation. The classical pre-processing we need to perform is to decompose matrix A into a sum of Pauli Strings. The efficiency of decomposition can vary depending on the choice of naive or optimized decomposition and the number of Pauli terms, which is dependent on the structure of matrix A. Here we use a naive decomposition of matrix A into 10 Pauli strings, where each matrix entry contributes independently to the Pauli expansion, as shown in the run log (top image of Figure 4).

```
#########################################################
Cray C/C++ Compiler // Cray MPICH // Cray Math Library-BLAS
#########################################################
Solving a linear system of equations:
Synthesized the full circuit. Sent circuit and metadata to simulator.
Received circuit and metadata from classical application.
Terminate simulator.

A = [[ 0.28 -0.01  0.02 -0.1 ]
 [-0.01  0.5  -0.22 -0.07]
 [ 0.02 -0.22  0.43 -0.05]
 [-0.1  -0.07 -0.05  0.42]]
b = [0.18257419 0.36514837 0.54772256 0.73029674]

Pauli strings list:
II :  (0.408+0j)
IZ :  (-0.052+0j)
IX :  (-0.03+0j)
ZI :  (-0.017+0j)
ZZ :  (-0.057+0j)
ZX :  (0.02+0j)
XI :  (-0.025+0j)
XZ :  (0.045+0j)
XX :  (-0.16+0j)
YY :  (-0.06+0j)

Comparing solutions:
classical:   [1.55071576 2.36243885 2.73915645 2.82784967]
HHL:         [1.60021777 2.40844196 2.77379842 2.8671254 ]
relative distance:  1.8 %
```

```
Job id    Name          Username   Time Use   NODES  S  Queue
--------------------------------------------------------------------
358737    interactive   user1      00:01:02   1      R  hpcnode[25]
358738    qc_hybrid     qc_user1   00:00:05   2      R  hpc_qcnode[1-2]
358739    HPC_workload1 user2      00:01:10   10     R  hpcnode[28-37]
358740    HPC_workload2 user3      00:00:02   10     R  hpcnode[10-15]
```

Figure 4: Hybrid quantum linear solver run log and results comparison with classical solution from BLAS (top). Schematic SLURM job status query (bottom).

The algorithm estimates a quantum state from the initial quantum state corresponding to n-dimensional vector. Estimation is done by inverting eigenvalues encoded in phases of eigenstates with use of Quantum Phase Estimation (QPE) and amplitude encoding. Now that we have finished with pre-processing we can turn to defining the HHL algorithm. The algorithm consists of 4 steps: 1) State preparation of the RHS vector $\vec{b}$. 2) QPE for the unitary matrix $e^{2\pi iA}$, which encodes eigenvalues on a quantum register of size m. 3) An inversion algorithm which loads amplitudes according to the inverse of the eigenvalue registers. 4) An inverse QPE with the parameters in (2). The resulting synthesized circuit is executed on state vector simulator by *Qiskit Aer* [3] in the interest of obtaining an exact solution. Hybrid executable can be submitted and scheduled for parallel execution as regular HPC workloads by a workload manager such as Slurm or PBS as shown in the bottom image of Figure 4.

The final quantum state must be processed to obtain a classical solution for comparison. Post-processing is necessary to reconstruct the classical solution by computing the expectation values of Pauli measurements and normalizing. The HHL solution is compared with the solution from the BLAS (Basic Linear Algebra Subroutines) library provided in Cray's scientific and mathematics library, and demonstrated less than 2% deviation for a 4x4 matrix, as shown in the top image of Figure 4. This is a small hybrid workload tested up to a 64x64 matrix on a HPE Cray EX system on two nodes with two AMD EPYC 7763 (Milan) CPUs per node. The primary objective of this first test case is to validate the functional correctness of the quantum interface library and establish the integrated workflow for a hybrid quantum-HPC workload.

2.1.3 Case B: Quantum optimization. The second workload consists of solving MaxCut problems with the quantum approximate optimization algorithm (QAOA) [8]. Synthesized circuits are provided in gate-level quantum assembly language OpenQASM3 with more complex control flows and parametric circuits. Such circuits are needed for variational algorithms like QAOA, which rely on quantum circuits (ansatz) parameterized by real-valued parameters and classical optimization that updates these parameters iteratively.

QAOA stands out as a prominent hybrid quantum-classical technique for addressing challenging combinatorial optimization problems, including the MaxCut problem, which involves finding the best partition of a graph's vertices into two sets to maximize the number of edges between them. The QAOA variational quantum state is given by:

$$|\psi(\boldsymbol{\gamma}, \boldsymbol{\beta})\rangle = \prod_{j=1}^{p} e^{-i\beta_j H_M} e^{-i\gamma_j H_C} |+\rangle^{\otimes n},$$

where:

- $|\psi(\boldsymbol{\gamma}, \boldsymbol{\beta})\rangle$ is the QAOA ansatz after p layers,
- $\boldsymbol{\gamma} = (\gamma_1, \ldots, \gamma_p)$ and $\boldsymbol{\beta} = (\beta_1, \ldots, \beta_p)$ are variational parameters,
- H_C is the cost Hamiltonian encoding the optimization problem,
- $H_M = \sum_{i=1}^{n} X_i$ is the mixer Hamiltonian (sum of Pauli-X operators),
- $|+\rangle^{\otimes n}$ is the initial equal superposition state.

The goal is to find $\boldsymbol{\gamma}, \boldsymbol{\beta}$ that maximize the expected cost:

$$\max_{\boldsymbol{\gamma}, \boldsymbol{\beta}} \langle \psi(\boldsymbol{\gamma}, \boldsymbol{\beta}) | H_C | \psi(\boldsymbol{\gamma}, \boldsymbol{\beta}) \rangle.$$

To scale this approach for large MaxCut instances (e.g., 500-10000 nodes), we explore the QAOA-in-QAOA (QAOA^2)[18] framework in this study. QAOA^2 employs a divide-and-conquer strategy: the original graph is decomposed into smaller subgraphs, each of which can be independently solved—often in parallel—using QAOA, enabling efficient utilization of multiple quantum resources.This is especially useful for current NISQ devices with limited number of qubits (~100-qubits).

The hybrid classical-quantum MaxCut workflow using QAOA^2 is illustrated in Figure 5 with results collected and compared on classical rank 0. The approach partitions a large input graph into smaller subgraphs that can be mapped to NISQ-capable circuits. The classical graph partitioning and community-level Maxcut merging are managed by classical HPC process, while quantum simulation of subgraphs is simulated using distributed memory parallelism over MPI. Each quantum process simulates a quantum device and performs QAOA optimization on its assigned subgraph. Results are sent back to rank 0 and gathered asynchronously, and subgraph results are combined into a unified solution. The QAOA^2 results are benchmarked against classical approaches (Goemans-Williamson, Greedy, Random) for node counts up to 2500 using up to 512 CPU nodes on the same HPE Cray EX system as described in the previous case. Greedy and GW methods can still yield strong results, especially on small or moderately sized graphs.QAOA^2 gives a reasonable approximation and performs better on larger structured graphs, as expected. This application was demonstrated live at ISC24 using a 20-qubits IQM quantum device accessed from the LUMI supercomputer in Finland [7].

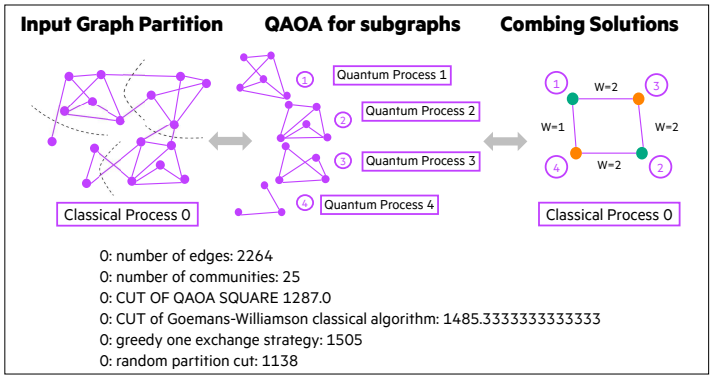

Figure 5: Hybrid distributed Quantum MaxCut with QAOA^2 scheme (top) and results comparison with classical algorithms (bottom).

2.2 Adaptive Circuit Knitting Hypervisor

For quantum computing to operate at scale, it will be necessary to efficiently parallelize over many QPUs, possibly of different modalities, integrating them as co-processors within an HPC framework. Quantum interconnects between QPUs may not be available in the near term. We will need to rely on classical HPC interconnects.

For hybrid quantum-classical computing to scale, efficient partitioning and distribution of quantum workloads across quantum processing units (QPUs) is crucial. To address the scalability of QC, a novel adaptive circuit knitting (ACK) toolkit has been developed with initial testing conducted on 1D quantum spin chains.

ACK can efficiently partition a larger quantum circuit into smaller sub-circuits by finding the optimal cuts locations of minimal entanglement on quantum gates between qubits. Especially, our method decreases the sampling overhead of circuit knitting by cutting gates in locations that minimize entanglement between partitions.

Distributing quantum computation is a highly non-trivial task and presents fundamentally different challenges and opportunities compared to classical HPC parallelization models. This is primarily due to the presence of entanglement, a non-local quantum correlation between qubits that cannot be cleanly partitioned in space or logic. Entanglement is essential to quantum advantage but also introduces strong coupling between distant parts of a quantum circuit. This makes naive partitioning ineffective: if two highly entangled qubits are placed on different QPUs, then inter-device communication or post-processing is required to reconstruct their joint behavior. Complicating this further, the structure of entanglement can be dynamic and problem-dependent — it is often not known in advance which qubits are most entangled and where circuit cuts can be made with minimal loss of quantum coherence. To address this, the circuit knitting technique [15] has been recently developed, where large quantum circuits are split into subcircuits, executed independently, and their observables recombined via classical post-processing. While this allows for some form of parallel execution and even distribution across classical HPC nodes, it incurs an exponential sampling overhead if entanglement between partitions is high.

To mitigate this cost, a more sophisticated adaptive circuit knitting method has been developed. In this approach, the quantum circuit is analyzed — and partitioned — in a way that explicitly minimizes the entanglement between subcircuits. One enabling tool is the use of tensor networks (TNs), such as matrix product states (MPS), which represent quantum states in a compressed form that naturally encodes the entanglement structure. These representations allow for an adaptive algorithm to select optimal cut points that minimize entanglement entropy, reducing the number of samples needed to reconstruct observables. As shown in the top panel of Figure 6, the process involves an inner loop where subcircuits (inspired by TN structure) are optimized in parallel and an outer loop where the cutting strategy is refined based on entanglement minimization. This fusion of quantum simulation with classical optimization mirrors how GPUs accelerate linear algebra, suggesting a future where QPUs act as accelerators for classically structured quantum workloads.

We demonstrate the potential of this method by applying adaptive circuit knitting to simulate strongly disordered 1D spin chains evolving under an Ising Hamiltonian with both transverse and longitudinal fields. Systems like these are of practical interest in condensed matter physics, and are known to exhibit complex quantum phenomena such as many-body localization, which can be difficult to simulate classically. Using an entropy-guided cutting

strategy, circuits of up to 32 qubits can be simulated on a single node with 4 A100 GPUs with 40GB HBM2 DRAM. For 40-qubits simulation, 256 nodes on Perlmutter with 4 A100 GPUs each were utilized using CUDA-Q SDK with GPU-accelerated quantum simulator backends [14]. We observed significant reductions in sampling overhead compared to baseline load-balanced cuts — most cases with 10-100x improvement and for a few cases over 1000x improvement, as shown in the lower panel of Figure 6.

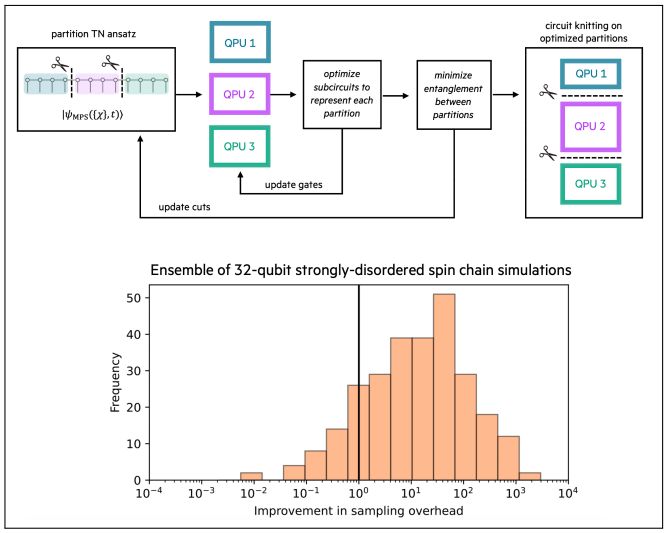

Figure 6: Adaptive circuit knitting hypervisor scheme (top) and sampling overhead reduction for 32-qubits strongly disordered spin chains (bottom).

2.3 Quantum Compiler Extension

At the lower level, quantum compiler and runtime extension is under development, to enable performant language-level support for quantum constructs and quantum co-processor programming capabilities. Good compatibility with existing classical compilers, libraries, and utilities that perform code analysis during compilation and automatically generate highly optimized code can be achieved by this direct integration. The main goal is to address the bottleneck in compile time for increasing circuit size (with increasing gate counts, gate depth, and number of qubits) and minimize latency between classical and quantum components in the program.

The framework of quantum compiler and runtime extensions is grounded in the following design principles:

- Integrated Quantum Co-Processor Programming Model: QPUs are integrated as accelerators alongside CPUs and GPUs within a heterogeneous HPC system. Rather than framing this solely as a runtime coordination of a single application workflow, the model is fundamentally enabling compilation of hybrid applications that offload selected quantum kernels to QPUs. These kernels may include quantum circuit synthesis, parametric ansatz, or iterative quantum-classical routines.
- Compiler-Level Interoperability Across Frontends: Frontend-agnosticism is emphasized by supporting LLVM IR extended with Quantum Intermediate Representation (QIR). QIR was developed by Microsoft and intended to serve as a common interface between many languages and target quantum hardwares. It enables the ingestion of IR emitted by a variety of quantum compilers, such as CUDA-Q (NVQ++), Q#, and OpenQASM-based toolchains.
- Hardware-Retargetable Code Generation and Linking: We leverage and build upon existing classical LLVM compilation framework to consume LLVM IR and QIR emitted from different quantum compilers/frontends. Assembly code for specific target architecture is generated from input IR with no non-standard extension. By linking with the proper runtime libraries provided by different quantum hardware vendors, execution on target quantum processors can be achieved in a quantum-backend-retargetable manner.

Figure 7 shows a partial fragment of the full LLVM IR with QIR emitted from a hybrid quantum-classical program. In a single-source context, a CUDA-Q quantum kernel for the creation and measurement of a 30-qubits GHZ state [17] is embedded in a C++ hybrid application with QIR emitted. LLVM codegen in Cray compiler is utilized to lower input IRs for binary object creation and linked with CUDA-Q runtime libraries for hybrid executable generation, which is executed on a single A100 GPU. It demonstrates how classical control flow, quantum memory management, and quantum instruction emission are interleaved via a unified intermediate representation. Semantic boundaries between quantum and classical computation is retained.

The modular, extensible design that builds on established Cray compiler infrastructure ensures that as quantum hardware evolves, the same compilation strategy can be retargeted, optimized, and deployed across future heterogeneous systems with QPUs constructed from diverse qubit modalities.

```
## QIR types and intrinsics
%Array = type opaque
%Qubit = type opaque
%Result = type opaque
declare %Array* @__quantum__rt__qubit_allocate_array(i64)
declare void @__quantum__rt__qubit_release_array(%Array*)
declare i64 @__quantum__rt__array_get_size_1d(%Array*)
declare i8* @__quantum__rt__array_get_element_ptr_1d(%Array*, i64)
declare void @__quantum__qis__h(%Qubit*)
declare void @__quantum__qis__x__ctl(%Array*, %Qubit*)
declare %Result* @__quantum__qis__mz(%Qubit*)
--------------------------------------------------------------------------------------------------------
## GHZ kernel (partial)
 {
entry:
 %qvec = call %Array* @__quantum__rt__qubit_allocate_array(i64 %n_qubits)
 %zero_idx = call i8* @__quantum__rt__array_get_element_ptr_1d(%Array* %qvec, i64 0)
 %q0 = bitcast i8* %zero_idx to %Qubit*
 call void @__quantum__qis__h(%Qubit* %q0)
 br label %loop
 %i = phi i64 [0, entry], [%next_i, loop_body]
 %cond = icmp ult i64 %i, %n_qubits_minus1
 br i1 %cond, label %loop_body, label %measure
--------------------------------
}
```

Figure 7: Partial IR fragment (with QIR) for hybrid applicaiton with GHZ kernel.

3 Summary

We have presented our quantum classical full stack solution and modular hardware/device-agnostic approach to address the unique challenges in pairing quantum computing with modern HPC infrastructure. To enable and facilitate the development of distributed, scalable hybrid HPC-QC workloads, the three main aspects we address are compatibility, performance, and scalability. The architecture design is defined as the development of extensible interfaces for quantum programming (bridging classical applications and various quantum SDKs), dispatching (sub-circuit partitioning for distributing quantum workloads across multiple QPUs), and compilation (portable quantum compilation and retargetability) within exsiting HPC programming environment.

At this stage, practical outcomes have been demonstrated on a range of hybrid HPC-QC workloads. Distributed hybrid HPC-QC applications including a quantum liner solver, quantum optimization using QAOA, and 1D spin chain simulation from smaller scale (2 AMD EPYC 7763 (Milan) CPU nodes) to large scale (up to 1024 A100 GPUs across 256 GPU nodes) have been constructed to validate the functionalities of quantum extensions developed by far. Existing infrastructure for HPC can be leveraged for future HPC-Quantum hybrid system in data and user management, process scheduling, control and networking to a large extent. Building on top of existing HPC toolchain with broad architecture support can significantly reduce development time and avoid complexity associated with compatibility issues. To ensure effective implementation and testing of QC extensions under development, we will continue to collaborate with multiple quantum software and hardware partners, and obtain valuable feedback from broader Quantum and HPC communities.

4 ACKNOWLEDGEMENTS

We would like to express our sincere gratitude to our collaborators at Classiq (Tamuz Danzig) and NVIDIA (Pooja Rao, Yuri Alexeev) for insightful discussions on their quantum algorithms and software packages. Special thanks to HPE Cray Compilation Environment Team for all the support.

References

[1] 2024. *Classiq platform.* https://platform.classiq.io/
[2] 2024. *Cray Scientific and Math Libraries.* https://cpe.ext.hpe.com/docs/24.07/csml/index.html
[3] 2024. *Introduction to qiskit.* https://www.ibm.com/quantum/qiskit
[4] M. Akhtar, F. Bonus, F. R. Lebrun-Gallagher, N. I. Johnson, M. Siegele-Brown, S. Hong, S. J. Hile, S. A. Kulmiya, S. Weidt, and W. K. Hensinger. 2023. A high-fidelity quantum matter-link between ion-trap microchip modules. *Nature Communications* 14, 531 (Feb. 2023), 1–8. doi:10.1038/s41467-022-35285-3
[5] Frederic T Chong, Diana Franklin, and Margaret Martonosi. 2017. Programming languages and compiler design for realistic quantum hardware. *Nature* 549 (Sept. 2017), 180–187. doi:10.1038/nature23459
[6] Andrew W Cross, Lev S Bishop, John A. Smolin, and Jay M. Gambetta. 2017. Open Quantum Assembly Language. *arXiv:* (2017), 1–24. doi:10.48550/arXiv.1707.03429
[7] Aniello Esposito and Tamuz Danzig. 2024. Hybrid Classical-Quantum Simulation of MaxCut using QAOA-in-QAOA. In *2024 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, Vol. 3. 1088–1094. doi:10.1109/IPDPSW63119.2024.00180
[8] Edward Farhi, Jeffrey Goldstone, and Sam. Gutmann. 2014. A Quantum Approximate Optimization Algorithm. *arXiv:* (Nov. 2014), 1–36. doi:10.48550/arXiv.1411.4028
[9] Aram W Harrow, Avinatan Hassidim, and Seth. Lloyd. 2009. Quantum algorithm for linear systems of equations. *Physics Review Letters* 103, 15 (Feb. 2009), 150502–150516. doi:10.1103/PhysRevLett.103.150502
[10] Travis S. Humble, Alexander McCaskey, Dmitry I. Lyakh, Meenambika. Gowrishankar, Albert. Frisch, and Thomas Monz. 2021. Quantum Computers for High-Performance Computing. *IEEE Micro* 41, 5 (Sept. 2021), 15–23. doi:10.1109/MM.2021.3099140
[11] Michael Kifer and Annie Yanhong Liu. 2018. *Declarative Logic Programming: Theory, Systems, and Applications* (1st. ed.). ACM Books, New York, NY.
[12] P. Krantz, M. Kjaergaard, F. Yan, T. P. Orlando, S. Gustavsson, and W.D. Oliver. 2019. A quantum engineer's guide to superconducting qubits. *Applied Physics Reviews* 6, 2 (June 2019), 021318–1–021318–57. doi:10.1063/1.5089550
[13] Andrew Litteken, Yung-Ching Fan, Devina Singh, Margaret Martonosi, and Frederic T Chong. 2023. An updated LLVM-based quantum research compiler with further OpenQASM support. *Quantum Science and Technology* 5, 3 (Feb. 2023), 1–19. doi:10.1088/2058-9565/ab8c2c
[14] Masoud Mohseni. 2025. How to Build a Distributed Quantum Computer: Adaptive Circuit Knitting. In *NVIDIA GTC 2025.* Nvidia corporation, Santa Clara, CA. https://www.nvidia.com/en-us/on-demand/session/gtc25-dd73669/
[15] Tianyi Peng, Aram W Harrow, Maris Ozols, and Xiaodi. Wu. 2020. Simulating Large Quantum Circuits on a Small Quantum Computer. *Physics Review Letters* 125 (Oct. 2020), 150504–150523. doi:10.1103/PhysRevLett.125.150504
[16] John Preskill. 2018. Quantum Computing in the NISQ era and beyond. *Quantum* 2, 79 (Aug. 2018), 1–8. doi:10.22331/q-2018-08-06-79
[17] Yajuan Zhao, Rui Zhang, Wulan Chen, Xian-Bin Wang, and Jiazhong Hu. 2021. Creation of Greenberger-Horne-Zeilinger states with thousands of atoms by entanglement amplification. *npj Quantum Inf* 7, 24 (Feb. 2021), 1–6. doi:10.1038/s41534-021-00364-8
[18] Z Zhou, Y Du, X Tian, and D. Tao. 2023. Qaoa-in-qaoa: solving large-scale maxcut problems on small quantum machines. *Physics Review Applied* 19, 15 (Feb. 2023), 024027–024042. doi:10.1103/PhysRevApplied.19.024027