OpenMP-Q: Quantum Task Offloading in OpenMP

Swastik Mittal[1][0000−0001−6418−8741], Atulya Mahesh[2], and Frank Mueller[3][0000−0002−0258−0294]

North Carolina State University, Raleigh, NC, USA

Abstract. Quantum computing devices are being considered for accelerating workloads such as combinatorial optimization, chemistry simulation, and hybrid machine-learning kernels, yet today's quantum SDKs (e.g., Qiskit, Cirq) are almost exclusively exposed through Python. This creates a gap for C/C++ applications, many of which already orchestrate heterogeneous CPU/GPU work with **OpenMP**, because they lack a clean path to invoke quantum kernels or iterate on their results.
We present **OpenMP-Q** to raise a discussion for inclusion in the OpenMP standard. OpenMP-Q is an extension to the OpenMP 5.x target model that treats a quantum processor as a first-class device and enables bidirectional communication between C++ and Python. OpenMP-Q adds (i) a quantum device type, (ii) a mechanism that bridges OpenMP's offload API to the Python quantum stack via **POSIX pipes**, and (iii) an extended offloading framework that communicates variables via map clauses to a Python script, executes the quantum task, and reinserts the results into the same mapped variables. The pipe mechanism also supports ***reverse-offload-style*** workflows, e.g., variational algorithms where C++ supplies initial rotation angles, the quantum backend returns qubit statistics, and the host refines those angles for subsequent iterations, without breaking OpenMP's asynchronous execution model or requiring extra device kernels. Moreover, by combining standard OpenMP parallel regions with per-thread quantum offload, OpenMP-Q scales natively across multiple QPUs, dispatching independent circuit evaluations in parallel.

Keywords: OpenMP · Quantum Computers · HPC

1 Introduction

Quantum computing has progressed from laboratory prototypes to cloud-hosted commercial devices that can outperform the fastest classical supercomputers on narrowly scoped tasks [18]. Google's 53-qubit Sycamore processor, e.g., needed just 200s to sample a random circuit that classical estimates place at $\approx 10,000$ years compute cost [4], though some debate remains surrounding this claim [29, 3]. Variational algorithms such as QAOA and VQE extend this promise to domains with realistic applications including combinatorial optimization, quantum chemistry, and machine-learning. The set of problems for which quantum computing promises significant speedups over classical approaches is only growing.

A broad spectrum of quantum hardware is now reachable via cloud. Superconducting qubit platforms (IBM-Q, Rigetti, QCI, OQC, Google, Amazon), trapped-ion systems (IonQ, Quantinuum), and neutral-atom arrays (QuEra, Pasqal) can be accessed through Amazon Braket [1], Azure Quantum [26], qBraid [17] or directly from the vendors. Software libraries like Qiskit [20], Cirq [13], Tket [33] and Pennylane [6] enable users to design quantum workloads while lower-level control of the hardware is enabled by interfacing through proprietary software layers, or open-source research infrastructure such as Conqure [24], QisDax [5], DAX [30], ARTIQ [8], Qubic [40, 41], and QICK [34]. This ecosystem of hardware and software technologies supports a growing interest in quantum computing and is evolving rapidly.

Traditionally, many of these applications are solved using conventional high-performance computing (HPC) systems with algorithmic solutions that have progressed rapidly through decades of hardware and software improvements [19]. High-performance computing (HPC) systems define the pinnacle of modern computing by relying on massively parallel processing. Specialized accelerators and nodes supporting highly concurrent execution with multi-threaded processing are utilized to achieve performance requirements. With the growing potential of quantum computers and strong overlapping with the existing HPC applications, requirements for integrating such devices as accelerators have increased. There have been significant contributions of integrating classical HPC with quantum computers [19, 9, 38, 32]).

This integration introduces middleware as software will be needed at the interface between classical HPC and low-level controlled quantum execution on quantum processing units (QPUs), i.e., for connectivity and communication between classical and quantum devices. Quantum-HPC middleware systems that facilitate the efficient coupling of quantum-classical computing are becoming increasingly important [31]. OpenMP [11, 25] is one of the most widely used framework for parallelizing classical HPC algorithms via multi-threading [21, 37]. Its *target* directives also enable seamless offloading to conventional accelerators such as GPUs. By contrast, modern quantum software stacks are defined almost entirely through Python-based SDKs, leaving C/C++ OpenMP applications without a straightforward way to call quantum kernels.

We introduce OpenMP-Q, a novel task-offloading extension to the OpenMP 5.x target model that enables quantum-accelerated HPC workloads to be dispatched to quantum devices directly from OpenMP programs. The extension adds two new clauses to the target directive, namely **circuit**, which holds information about the quantum circuit (sequence of quantum gates), and **iteration**, which specifies the number of workload iterations. These parameters are conveyed to the OpenMP runtime to describe the circuit and its execution frequency. A lightweight bridge based on POSIX pipes launches the requisite Python code and establishes bi-directional communication between the C/C++ host and the quantum task executed through the Python quantum SDK.

OpenMP-Q unifies three key offload modes under the familiar OpenMP API, namely single-QPU, multi-QPU, and in-region reverse-offload. In reverse-offload

mode, a quantum kernel returns measurement statistics, the CPU immediately updates the rotation angles, and the revised parameters stream back into the same long-lived target region, thereby avoiding repeated kernel launches. Meanwhile, wrapping the offload directive in an `omp parallel` region lets each thread bind to its own QPU for concurrent circuit execution. On the host side, programmers can overlap classical work with GPU offload or call MPI collectives, yielding a tightly-coupled quantum-classical pipeline that scales from on-node accelerators to multi-node clusters and cloud deployments. This paper, in contrast to Conqure [24], aims to spark a discussion on potential inclusion of OpenMP-Q into the OpenMP standard and makes the following contributions:

- We design and implement OpenMP-Q, an OpenMP 5.x extension that treats a quantum processor as a first-class target device, adding only two new clauses — `circuit` and `iteration` — to the standard API and discuss compatibility with the OpenMP standard.
- We build a lightweight runtime bridge using POSIX pipes that gives C/C++ developers the illusion of native quantum-kernel launches while actually invoking Python-based quantum software.
- We demonstrate how OpenMP-Q naturally supports both reverse-offload (in-region iterations without re-launching) and multi-QPU parallelism simply by mixing `iteration(k)` and `omp parallel`.
- We validate our approach with a scale-out VQE experiment, achieving up to 2.9× end-to-end speedup by dispatching multiple variational runs concurrently and a 126× speedup for a single long-running target region (reverse-offload style execution) for the same workload.

2 Related Work

2.1 Middleware for Hybrid QC-HPC Workloads

Quantum computing (QC) introduces a novel mode of computation with the possibility of greater computational power that remains to be exploited, presenting exciting opportunities for HPC applications [14]. Yet, only limited effort has been invested into coupling quantum offloading with classical HPC. Elsharkawy et al. [14] review possible QC-HPC integration scenarios and identify key features quantum-programming tools must provide. Bertels et al. [7] propose OpenQL, a quantum classical hybrid computational language whose compiler translates the program to a common assembly language called cQASM, which can be executed on a quantum simulator. Britt et al.[10] develop a quantum-accelerator framework that uses specialized kernels to offload select workloads while integrating into existing computing infrastructure. These studies underscore the growing demand for Quantum-HPC integration and contribute valuable tooling for both domains. However, they do not articulate a general middleware solution that abstracts the interface between classical and quantum resources. Saurabh et al. [31] provide a conceptual middleware to facilitate reasoning about quantum-classical integration, which serves as the basis for future middleware.

2.2 OpenMP-Centric Quantum Offloading

MPI and OpenMP are the de factor standards for distributed and shared-memory parallelism, respectively. We target OpenMP as the standard classical HPC C/C++ framework to integrate with Python-based quantum APIs. Because OpenMP already supports offloading to accelerators such as GPUs, it offers a natural foundation for treating a quantum processor as yet another target device. Prior work has extended the OpenMP target directive to accommodate emerging accelerators [15]. We propose **OpenMP-Q**, extending the OpenMP 5.x target specification to provide additional support for quantum offloading following the OpenMP standards while coexisting seamlessly with existing GPUs and other accelerators.

[22] presents the closest related effort in terms of the OpenMP-Q contribution. Their work extends OpenMP to support quantum offloading through function calls that create and measure quantum registers and apply a fixed set of single- and two-qubit gates. These circuits are then transpiled into QASM or QIR for execution. However, that poster offers no evaluation results and falls short on the generality and scalability needed for broad quantum-classical integration. OpenMP-Q, by contrast, stays fully compliant with the OpenMP 5.x target-offload standard, adds only a few new clauses, and embeds a pipe-based bridge inside the existing target-runtime path without altering its default behavior. It cooperates with other accelerators and still supports performance features such as reverse offloading.

Conqure [24] introduced an open-source infrastructure for quantum job scheduling and integration with both HPC and existing quantum frameworks such as Qiskit, including an early version of OpenMP-Q. Our current submission differs in that it focuses on OpenMP compatibility, semantic challenges and novel features such as the `iteration` clause to facilitate QPU parallelism and iterative kernel execution within a single job.

3 Design

The device clause in OpenMP, which enables device offload, was introduced in OpenMP 4.0 [27], giving C/C++ and Fortran codes a standard **target** directive for launching kernels on accelerators. Subsequent releases [28] added teams, nowait, unified shared memory, and improved data-mapping clauses, enabling performance on par with hand-coded CUDA/HIP kernels. Chunhua et al. [23] examined the newly released accelerator directives and created an initial reference implementation, referred to as HOMP (Heterogeneous OpenMP), focused on targeting NVIDIA GPUs. Furthermore, multiple works demonstrated significant speedups utilizing the target offload on different accelerators [36, 16] or extended this support using compiler techniques to support new accelerators [12]. Similarly, we propose **OpenMP-Q**, a quantum accelerator extension for OpenMP.

3.1 OpenMP-Q: Single QPU

Quantum programs consist of a sequence of gates performing individual operations on one or multiple qubits [22]. Quantum programs usually run the same gate sequence many times. If several quantum accelerators are available, these sequences can be split and dispatched to different devices according to the data partitions.

Listing 1.1: OpenMP-Q: Single Quantum Offload

```
void VQE(QuantumWrapper *qc, int num_qubits, double angles[]) {
    //add series of quantum gates, here: VQE
    qc->h(0); // Hadamard gate
    for (int i = 0 ; i < num_qubits; i++)
        qc->ry(angles[i], i); // Y Rotation
    for (int i = 0 ; i < num_qubits-1; i++)
        qc->cx(i, i+1); // CNOT gate
    for (int i = 0 ; i < num_qubits; i++)
        qc->ry(angles[num_qubits+i], i); // Y Rotation
    qc->measure();
}

void main() {
    double angles[num_qubits] = init_angles(); // angles per VQE qubit
    QuantumWrapper *qc = new QuantumWrapper; // QuantumWrapper Class
        Object
    for (i = 0; i < MAX_ITERS ; i++) {
        # pragma omp target device(Quantum) circuit(qc) map(to: angles)
            map(from: qubit_stats) // 1
        {
            VQE(qc, num_qubits, angles); // 2. Quantum Gate Sequence
            qubit_stats = qc->execute_quantum_task(); // 3.
        }

        MPI_Broadcast(... qubit_stats ...); // distribute over nodes
        angles = update_angles(qubit_stats); // classical
        MPI_Allreduce(0 , ... angles ...);
        pick_best_angles(angles);
    }
}
```

Listing 1.1 presents a basic hybrid quantum-classical variational loop (e.g., VQE or QAOA) implemented with OpenMP. The code first initializes the array of rotation angles, then offloads a quantum-measurement kernel to the accelerator. After the kernel returns, the qubit measurement statistics of multiple shots are copied back to the host, where a classical optimizer computes revised angle values. This sequence repeats for successive iterations until the algorithm converges on a final result or some upper bound on iterations is reached.

The routine that refines the rotation angles can run as a GPU kernel or be spread across the CPU cores of a node — even a multi-node MPI kernel could used if computational needs require it. After each quantum evaluation, the locally produced values are combined to generate the updated parameter set

for the next iteration. MPI is optional: Without it, the computation remains on a single node, whereas enabling MPI lets the workflow span nodes, each with their own QPU, and merges their results via collective operations.

To execute the offloaded task on quantum devices, we extend the OpenMP 5.x target specification with a quantum-circuit library that integrates seamlessly with existing OpenMP runtime. The numbered comments in Listing 1.1 include the following:

1. `device(quantum)` tells OpenMP to target a quantum accelerator. The newly introduced *circuit* clause passes the programmer-initialized *QuantumWrapper* object as a pointer to the OpenMP runtime.
2. When the region begins, the programmer populates the information about the quantum sequence within the object by invoking library methods for qubit gates, such as *h(), ry(), and cx()*.
3. The OpenMP runtime inspects the object, translates the stored gate list along with quantum task execution information into a Python script that targets the selected quantum software tool, and launches the quantum task on the designated accelerator (see Section 3.2).

3.2 OpenMP-Q: Reverse Offload

Listing 1.2: OpenMP-Q: Reverse Offload Compatibility

```
#pragma omp requires reverse_offload

// Same host arrays as before
double angles[num_qubits], qubit_stats[num_qubits];

// One long-lived target region
QuantumCircuit *qc = VQE(); //(1) quantum gate sequence
#pragma omp target circuit(qc) device(quantum) iteration(MAX_ITERS) map(
    to: angles map(from: qubit_stats)
{
    //(2) Invoke quantum task
    for (int iter = 0; iter < MAX_ITERS; ++iter) {
        qubit_stats = qc->execute_quantum_task(); // Executes
        /*
            qc->write_to_device(angles); //(3) Write angles to python
            qc->run(); //(4) Execute QPU circuit
            qc->qubit_stats = read_from_device(); //(5) Relay results to C
                /C++
        */

        #pragma omp target(ancestor: 1) map(always, to: angles)
            angles = update_angles(qubit_stats)
    }
}
```

The OpenMP 5.x target specification provides the reverse-offload capability within the target region. Reverse-offload lets device code invoke a host function

through a shared-memory RPC channel without exiting the surrounding target region. Eliminating repeated target launches significantly reduces the per-iteration overhead. It also reduces memory overhead by only copying the required or modified data between device-host when necessary.

Listing 1.2 demonstrates OpenMP-Q's compatibility with reverse-offload. To capitalize on the benefits of reverse-offload, we expect the programmer to invoke the quantum gate sequences outside the target region; otherwise, with execution inside, it would lead to re-initialization of the same quantum gate at each iteration. The sequence of iterations executed is also pushed within the target to execute a single long-running target region. We introduce a new `iteration` clause that specifies the exact number of host-device exchanges to the OpenMP runtime, thereby establishing a deterministic bi-directional communication channel between the C/C++ host and the python-quantum task.

1. The user constructs and populates the `QuantumCircuit` object and hands its pointer to the runtime through `circuit(qc)`.
2. On entering the long-lived `target` region, OpenMP-Q reads both the `circuit` and `iteration` clauses, spawns the Python bridge, and initializes the quantum task context.
3. At each iteration, the runtime updates the device-side buffer with the current angle values that were designated by the original map clause.
4. The python-quantum task executes task using the quantum software library, runs the circuit with the supplied angles, and obtains qubit statistics.
5. The script writes the measurement data back through the pipe, where they populate `qubit_stats` on the OpenMP side.

The `iteration` value tells the script how many such rounds to perform before terminating. After each round, the host updates the angle array, optionally in parallel via MPI or GPU offload, and the cycle repeats until all iterations complete. This avoids the overhead of creating separate offload tasks for every quantum call.

3.3 OpenMP-Q: Multi-Qpu

Listing 1.3: OpenMP-Q: Multiple Quantum Offload

```
double angles[num_threads][num_qubits] = init_angles();

#pragma omp parallel
{
    qdev = omp_get_thread_num();
    for(int i = 0 ; i < MAX_ITERS ; i++) {
        device_angles = angles[qdev];
        QuantumCircuitWrapper *qc = new QuantumCWrapper(qubits);
        # pragma omp target device(Quantum) circuit(qc) map(to:
            device_angles)
        {
            VQE(c, num_qubits, device_angles);
            qubit_stats = qc->execute_quantum_task();
        }
```

```
        angles[qdev] = update_angles(qubit_stats);
    }
}
```

Listing 1.3 illustrates how to launch several quantum tasks in parallel. Using OpenMP, we spawn as many tasks as there are available threads, each executing a different quantum circuit on a separate QPU (or on distinct slots of a simulator). To remain consistent with OpenMP's offload rules, every thread first copies its own row from the global angles array into a private qpu_angles buffer, so the parameters inside the target region are thread-specific.

The same mechanism scales naturally when multiple QPUs exist. For instance, a variational algorithm that explores different Ansatz templates [39] can assign one QPU per node, feed each with its own set of initial angles, and evaluate several parameterized Ansatz strategies at once.

4 Implementation

OpenMP-Q provides a shared embedding library that integrates directly into LLVM frontends [2], enabling the generation of Quantum Intermediate Representation (QIR) for quantum gates via our OpenMP-Q extension.

- Listings 1.1, 1.2, and 1.3 demonstrate how the user invokes our embedded quantum library to instantiate and initialize a `QuantumWrapper` object.
- The user may build the gate sequence either inside the `target` region (Listing 1.3) or beforehand (Listing 1.2). Each library call (e.g., `h()`, `ry()`, `cx()`) appends the corresponding gate (as a Python-ready string) to the circuit object.
- Listing 1.4 demonstrates OpenMP offload runtime modifications to consider quantum as an offload device with two different execution styles (Listing 1.2 & 1.3).
- The `circuit` and `iteration` clauses forward the circuit pointer and iteration count into the OpenMP runtime's `targetKernel` entry (see Listing 1.4). Here, the runtime unpacks the object pointer to recover the gate list and, in the reverse-offload case, spawns the Python bridge at region entry by invoking the python execution process. In the multi-QPU pattern, region entry merely marshals the initial `angles[]` into the circuit object.
- OpenMP-Q uses a pair of Linux POSIX pipes to connect the C/C++ runtime and the Python process, streaming the `angles[]` vector into Python and returning `qubit_stats[]` back to the host.
- In the multi-QPU variant (Listing 1.3), the gate sequence lives in the offload region so each thread can target a distinct circuit/device. OpenMP-Q can also support a reverse-offload version of this pattern.
- When the user calls `qc->execute_quantum_task()`, the library writes the current `angles[]` into the device buffer, invokes the Python-based quantum software, and populates `qubit_stats[]` with the returned measurements.
- In the reverse-offload loop (Listing 1.2), these steps occur back-and-forth inside a single, long-lived `target` region: send angles, run the circuit, receive statistics, update angles, and repeat—avoiding repeated region entry

for each iteration. To enable this bidirectional exchange via our quantum-circuit library, we create two POSIX pipes to form a bidirectional channel between C++ and Python. One pipe carries data from the host into the Python subprocess, the other carries results back, similar to device copy-in and copy-out. We then *fork()* a child process, *dup2()* its stdin/stdout to the appropriate pipe ends, and exec() the Python interpreter. This allows the parent to write angle vectors to the child's stdin and read qubit statistics from its stdout seamlessly.

- In Listing 1.4, the OpenMP runtime retrieves the `num_iteration` value from the `iteration` clause. This value defaults to 0 when the clause is absent. If `num_iteration`> 0, it performs a single, long-lived offload region that loops `num_iteration` times (reverse-offload style); otherwise, it issues separate offload calls each time an iteration completes (e.g., the multi-QPU pattern).
- Listing 1.5 shows the Python script that will be generated and executed by listing 1.2: It reads the iteration count from its command-line arguments, then loops that many times, reading angles from the input pipe, executing the quantum task via the software library, and writing the resulting statistics (qubits frequency) back to the output pipe.

Listing 1.4: OpenMP-Q, LLVM OpenMP Offload Updates

```
targetKernel(Int64_t DeviceId, KernelArgsTy *KernelArgs) {
    if(DeviceId == Quantum) {
        // (1) Retrieve quantum circuit object pointer
        c = (QuantumCircuitWrapper*) KernelArgs->ArgBasePtrs[n]; // n
             = 0 (serial) or 1 (parallel)
        c->num_iterations = KernelArgs->num_iterations;
        for (int32_t I = 0; I < KernelArgs->NumArgs; ++I){
           if (KernelArgs->ArgTypes[I] & OMP_TGT_MAPTYPE_TO) {
               // (2) Parse mapped device angles from ‘map(to:
                    device_angles)‘ and serialize
               processDataMapTo(KernelArgs->ArgBasePtrs[I])
            }
        }

        if(c->num_iterations != 0)
           c->invoke_python();
    }
    // (3) Execute user-defined offload region -- populate quantum
        gate sequence
    target(...)
    if (DeviceId == Quantum) {
        // (4) Generate and execute Python script
        if(c->num_iterations == 0)
            c->invoke_python();
        // (5) Write back result from python to device_angles via ‘map
             (from: device_angles)‘
        processDataMapFrom(KernelArgs->ArgBasePtrs[I]))
    }
}
```

Listing 1.5: Generated Python Script: Circuit Execution

```
if __name__ == "__main__":
    input_itr = json.loads(sys.argv[2])
    circuit = QuantumCircuit(4)
    params = []
    params.append(Parameter('p0'))
    params.append(Parameter('p1'))
    params.append(Parameter('p2'))
    params.append(Parameter('p3'))
    params.append(Parameter('p4'))
    params.append(Parameter('p5'))
    params.append(Parameter('p6'))
    params.append(Parameter('p7'))
    circuit.ry(params[0], 0)
    circuit.ry(params[1], 1)
    circuit.ry(params[2], 2)
    circuit.ry(params[3], 3)
    circuit.cx(0, 1)
    circuit.cx(1, 2)
    circuit.cx(2, 3)
    circuit.ry(params[4], 0)
    circuit.ry(params[5], 1)
    circuit.ry(params[6], 2)
    circuit.ry(params[7], 3)
    circuit.measure_all()
    simulator = AerSimulator()
    for i in range(input_itr):
        resp = sys.stdin.readline().strip()
        resp = re.sub(r',\s*]', ']', resp)
        response_data = json.loads(resp)
        user_params = response_data
        vals = [float(p) for p in user_params]
        bound_qc = circuit.assign_parameters({ param: value for param,
            value in zip(circuit.parameters, vals)})
        job = simulator.run(bound_qc, shots=1024)
        result = job.result()
        counts = result.get_counts()
        counts = json.dumps(counts)
        sys.stdout.write(counts)
        sys.stdout.flush()
```

Discussion: The QPU kernel offloading via a Python script provides a viable path for current quantum environments and can easily be retargeted from one QPU vendor to another. Our OpenMP specification is more abstract though and can be backed by non-Python interfaces. In the future, a native C/C++ interface could be used to lower-level controls, such as ARTIQ [8], Qubic [40, 41], or QICK [34]. We are currently investigating future applications in quantum that may require such tight binding. The near-term applications, however, can cope easily with the described Python interface.

5 Results

To perform evaluations, we programmed a variational quantum eigensolver (VQE) to approximate the ground-state energy of a Hamiltonian. VQE proceeds by: (1)

Ansatz preparation: A parameterized quantum circuit encodes a trial wavefunction. (2) **Expectation estimation**: The Hamiltonian is decomposed into simpler terms (e.g., Pauli strings), each measured on the quantum device to compute the energy. (3)**Classical optimization**: A classical solver updates the circuit parameters and iterates until converged. Because VQE can stall in local minima or barren plateaus, practitioners often launch multiple trials with different random initial angles and select the lowest-energy result. Traditionally, these runs execute one after another even though they are pairwise independent.

We evaluate OpenMP-Q using a VQE implementation that optionally parallelizes over different random angles. The VQE problem is a 7-vertex Max-Cut instance, following the setup in [35]. Due to space constraints, we omit the quantum accuracy results and instead compare runtimes for 1–6 VQE runs, executed (a) serially (Listing 1.1) and (b) in parallel via our Multi-QPU offload extension (Listing 1.3). Figure 1 plots the median total time (y-axis) over 200 trials (whiskers show 1st and 3rd quartiles, barely visible due to low variations) as the number of concurrent runs increases (x-axis).

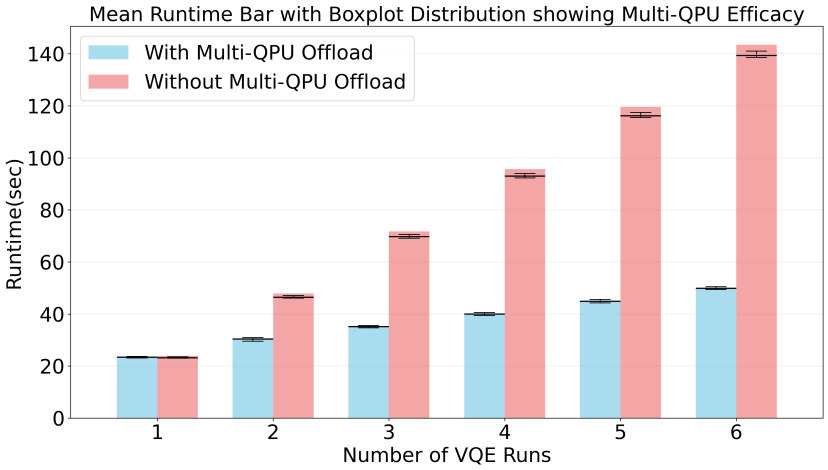

Fig. 1: Comparison of runtimes when running VQE runs with and without Multi-QPU Offloading

A single VQE execution takes 23 seconds on average under OpenMP-Q. Six runs back-to-back require 143 seconds on average, whereas six simultaneous runs complete in just 49 seconds, which is a 2.9× overall speedup. These results confirm that integrating quantum offload into the OpenMP parallel execution model yields substantial performance gains and that OpenMP-Q effectively extends OpenMP's accelerator framework to quantum devices.

We also evaluated OpenMP-Q in reverse-offload mode as shown in Listing 1.2. By entering the target region only once, Python is launched a single time and a bidirectional pipe connection is established for all six iterations. Figure 2 depicts the median wall-clock time over 200 trials: Six iterations complete in approxi-

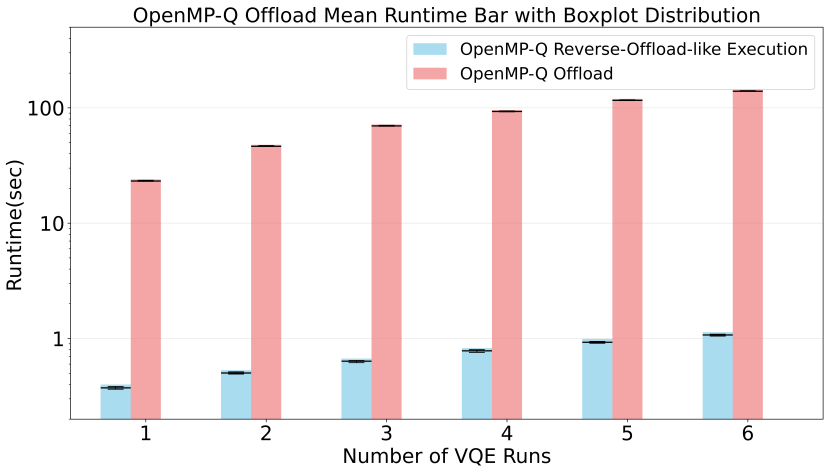

Fig. 2: Comparison of runtimes when running VQE runes with and without Reverse-Offload-like Functionality

mately 1.13 seconds, a 126× speedup compared to naive back-to-back offloads. This dramatic improvement highlights the cost of repeated execution of the OpenMP target directive, reconstructing the circuit and re-invoking Python (including pipe setup and library imports) on every iteration, which demonstrates the need for reverse-offload style execution for quantum task offloading. Since LLVM's OpenMP currently does not support reverse-offload, our experiment emulates such behavior by using x86 as the "device" and running the simulator in Python, i.e., the host-only execution without reverse-offload is effectively equivalent. OpenMP-Q's reverse-offload capability eliminates this overhead, which is essential for efficient hybrid workflows. A native Multi-QPU experiment executed with the reverse-offload target would further improve performance over all iterations.

6 Conclusion

We presented **OpenMP-Q**, a lightweight extension to the OpenMP 5.x **target** directive that treats quantum processors in much same manner as any other offload devices, and discuss its compatibility challenges with the existing OpenMP standard. OpenMP-Q's capabilities are realized by adding only two clauses, `circuit` to pass a user-built quantum-circuit object and `iteration` to enable single long target region execution, along with a POSIX-pipe bridge to existing Python-based quantum software libraries. OpenMP-Q preserves the familiar OpenMP programming model while transparently invoking quantum kernels. Our design naturally supports **reverse-offload** without repeated region re-entry and scales across multiple QPUs via standard **omp parallel** directives. A case study on a Max-Cut VQE problem demonstrated up to a 2.9× speedup when dispatching six variational runs in parallel and a 126× speedup when dispatching a single long target region for the same workload. We hope our presentation will spark a discussion on suitability of quantum acceleration support as an extension of the OpenMP standard.

References

1. AWS. Amazon braket., https://aws.amazon.com/braket/
2. Clang: Compiler Front-end, https://clang.llvm.org/
3. Aaronson, S.: Shtetl-optimized (Sep 2019), https://www.scottaaronson.com/blog/
4. Arute, F., Arya, K., Babbush, R., Bacon, D., Bardin, J.C., Barends, R., Biswas, R., Boixo, S., Brandao, F.G., Buell, D.A., et al.: Quantum supremacy using a programmable superconducting processor. Nature **574**(7779), 505–510 (2019)
5. Badrike, K., Dalvi, A.S., Mazurek, F., D'Onofrio, M., Whitlow, J., Chen, T., Phiri, S., Riesebos, L., Brown, K.R., Mueller, F.: Qisdax: An open source bridge from qiskit to ion trap quantum devices. In: IEEE International Conference on Quantum Computing and Engineering (QCE) (Sep 2023)
6. Bergholm, V., Izaac, J., Schuld, M., Gogolin, C., Ahmed, S., Ajith, V., Alam, M.S., Alonso-Linaje, G., AkashNarayanan, B., Asadi, A., Arrazola, J.M., Azad, U., Banning, S., Blank, C., Bromley, T.R., Cordier, B.A., Ceroni, J., Delgado, A., Matteo, O.D., Dusko, A., Garg, T., Guala, D., Hayes, A., Hill, R., Ijaz, A., Isacsson, T., Ittah, D., Jahangiri, S., Jain, P., Jiang, E., Khandelwal, A., Kottmann, K., Lang, R.A., Lee, C., Loke, T., Lowe, A., McKiernan, K., Meyer, J.J., Montañez-Barrera, J.A., Moyard, R., Niu, Z., O'Riordan, L.J., Oud, S., Panigrahi, A., Park, C.Y., Polatajko, D., Quesada, N., Roberts, C., Sá, N., Schoch, I., Shi, B., Shu, S., Sim, S., Singh, A., Strandberg, I., Soni, J., Száva, A., Thabet, S., Vargas-Hernández, R.A., Vincent, T., Vitucci, N., Weber, M., Wierichs, D., Wiersema, R., Willmann, M., Wong, V., Zhang, S., Killoran, N.: Pennylane: Automatic differentiation of hybrid quantum-classical computations (2022), https://arxiv.org/abs/1811.04968
7. Bertels, K., Sarkar, A., Hubregtsen, T., Serrao, M., Mouedenne, A.A., Yadav, A., Krol, A., Ashraf, I., Almudever, C.G.: Quantum computer architecture toward full-stack quantum accelerators. IEEE Transactions on Quantum Engineering **1**, 1–17 (2020)
8. Bourdeauducq, S., Jördens, R., Zotov, P., Britton, J., Slichter, D., Leibrandt, D., Allcock, D., Hankin, A., Kermarrec, F., Sionneau, Y., Srinivas, R., Tan, T.R., Bohnet, J.: Artiq 1.0 (May 2016). https://doi.org/10.5281/zenodo.51303, https://doi.org/10.5281/zenodo.51303
9. Britt, K.A., Humble, T.S.: High-performance computing with quantum processing units. ACM Journal on Emerging Technologies in Computing Systems (JETC) **13**(3), 1–13 (2017)
10. Britt, K.A., Mohiyaddin, F.A., Humble, T.S.: Quantum accelerators for high-performance computing systems. In: 2017 IEEE International Conference on Rebooting Computing (ICRC). pp. 1–7. IEEE (2017)
11. Clark, D.: Openmp: a parallel standard for the masses. IEEE Concurrency **6**(1), 10–12 (1998). https://doi.org/10.1109/4434.656771
12. Cramer, T., Römmer, M., Kosmynin, B., Focht, E., Müller, M.S.: Openmp target device offloading for the sx-aurora tsubasa vector engine. In: International Conference on Parallel Processing and Applied Mathematics. pp. 237–249. Springer (2019)
13. Developers, C.: Cirq. Zenodo (Apr 2025). https://doi.org/10.5281/ZENODO.4062499, https://zenodo.org/doi/10.5281/zenodo.4062499
14. Elsharkawy, A., To, X.T.M., Seitz, P., Chen, Y., Stade, Y., Geiger, M., Huang, Q., Guo, X., Ansari, M.A., Mendl, C.B., et al.: Integration of quantum accelerators with high performance computing–a review of quantum programming tools. arXiv preprint arXiv:2309.06167 (2023)

15. Espinosa, A., Klemm, M., de Supinski, B.R., Cytowski, M., Klinkenberg, J.: Advancing OpenMP for Future Accelerators. Springer (2024)
16. Gammelmark, M., Rydahl, A., Karlsson, S.: Openmp target offload utilizing gpu shared memory. In: International Workshop on OpenMP. pp. 114–128. Springer (2023)
17. Hill, R.J., Jain, R., Gupta, H., Jun Liang, T., Louamri, M.M., Young, R., Weis, E., Tsuoka, K., Jacobson, G., McIrvin, C., Weinberg, P., Purohit, S., Necaise, J., Vara, E.A., Chakraborty, P., Liu, J., Coladangelo, A.W., Kakhandiki, P., Makhanov, H., Sharma, P., Arulandu, A., Cosentino, A., Setia, K.: qBraid-SDK: Platform-agnostic quantum runtime framework. (Mar 2025). https://doi.org/10.5281/zenodo.12627596, https://github.com/qBraid/qBraid
18. Horowitz, M., Grumbling, E.: Quantum computing: progress and prospects (2019)
19. Humble, T.S., McCaskey, A., Lyakh, D.I., Gowrishankar, M., Frisch, A., Monz, T.: Quantum computers for high-performance computing. IEEE Micro **41**(5), 15–23 (2021)
20. Javadi-Abhari, A., Treinish, M., Krsulich, K., Wood, C.J., Lishman, J., Gacon, J., Martiel, S., Nation, P.D., Bishop, L.S., Cross, A.W., Johnson, B.R., Gambetta, J.M.: Quantum computing with qiskit (2024), https://arxiv.org/abs/2405.08810
21. Jin, H., Jespersen, D., Mehrotra, P., Biswas, R., Huang, L., Chapman, B.: High performance computing using mpi and openmp on multi-core parallel systems. Parallel Computing **37**(9), 562–575 (2011)
22. Lee, J.K., Brown, O.T., Bull, M., Ruefenacht, M., Doerfert, J., Klemm, M., Schulz, M.: Quantum task offloading with the openmp api. arXiv preprint arXiv:2311.03210 (2023)
23. Liao, C., Yan, Y., De Supinski, B.R., Quinlan, D.J., Chapman, B.: Early experiences with the openmp accelerator model. In: OpenMP in the Era of Low Power Devices and Accelerators: 9th International Workshop on OpenMP, IWOMP 2013, Canberra, ACT, Australia, September 16-18, 2013. Proceedings 9. pp. 84–98. Springer (2013)
24. Mahesh, A., Mittal, S., Mueller, F.: Conqure: A co-execution environment for quantum and classical resources (2025), https://arxiv.org/abs/2505.02241
25. Mattson, T.G.: An introduction to openmp. In: ccgrid. pp. 3–5 (2001)
26. Microsoft: Azure quantum development kit, https://github.com/microsoft/qsharp
27. OpenMP Architecture Review Board: OpenMP Application Programming Interface Version 4.0 (2013), https://www.openmp.org/wp-content/uploads/OpenMP4.0.0.pdf
28. OpenMP Architecture Review Board: OpenMP Application Programming Interface Version 5.2 (2022), https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5-2.pdf
29. Pednault, E., Gunnels, J., Maslov, D., Gambetta, J.: On "quantum supremacy" (Oct 2019), https://www.ibm.com/blogs/research/2019/10/on-quantum-supremacy/
30. Riesebos, L., Bondurant, B., Whitlow, J., Kim, J., Kuzyk, M., Chen, T., Phiri, S., Wang, Y., Fang, C., Horn, A.V., Kim, J., Brown, K.R.: Modular software for real-time quantum control systems. In: 2022 IEEE International Conference on Quantum Computing and Engineering (QCE). p. 545–555. IEEE (Sep 2022). https://doi.org/10.1109/qce53715.2022.00077, http://dx.doi.org/10.1109/QCE53715.2022.00077
31. Saurabh, N., Jha, S., Luckow, A.: A conceptual architecture for a quantum-hpc middleware. In: 2023 IEEE international conference on quantum software (QSW). pp. 116–127. IEEE (2023)

32. Schulz, M., Ruefenacht, M., Kranzlmüller, D., Schulz, L.B.: Accelerating hpc with quantum computing: It is a software challenge too. Computing in Science & Engineering **24**(4), 60–64 (2022). https://doi.org/10.1109/MCSE.2022.3221845
33. Sivarajah, S., Dilkes, S., Cowtan, A., Simmons, W., Edgington, A., Duncan, R.: t|ket⟩: a retargetable compiler for nisq devices. Quantum Science and Technology **6**(1), 014003 (nov 2020). https://doi.org/10.1088/2058-9565/ab8e92, https://dx.doi.org/10.1088/2058-9565/ab8e92
34. Stefanazzi, L., Treptow, K., Wilcer, N., Stoughton, C., Bradford, C., Uemura, S., Zorzetti, S., Montella, S., Cancelo, G., Sussman, S., Houck, A., Saxena, S., Arnaldi, H., Agrawal, A., Zhang, H., Ding, C., Schuster, D.I.: The QICK (quantum instrumentation control kit): Readout and control for qubits and detectors. Rev. Sci. Instrum. **93**(4), 044709 (Apr 2022)
35. Tsukayama, D., Shirakashi, J.i., Shibuya, T., Imai, H.: Enhancing computational accuracy with parallel parameter optimization in variational quantum eigensolver. AIP Advances **15**(1), 015226 (01 2025). https://doi.org/10.1063/5.0236028, https://doi.org/10.1063/5.0236028
36. Valero-Lara, P., Kim, J., Hernandez, O., Vetter, J.: Openmp target task: Tasking and target offloading on heterogeneous systems. In: European Conference on Parallel Processing. pp. 445–455. Springer (2021)
37. Vitali, E., Gadioli, D., Palermo, G., Beccari, A., Cavazzoni, C., Silvano, C.: Exploiting openmp and openacc to accelerate a geometric approach to molecular docking in heterogeneous hpc nodes. The Journal of Supercomputing **75**, 3374–3396 (2019)
38. Wintersperger, K., Safi, H., Mauerer, W.: Qpu-system co-design for quantum hpc accelerators. In: International Conference on Architecture of Computing Systems. pp. 100–114. Springer (2022)
39. Wu, A., Li, G., Wang, Y., Feng, B., Ding, Y., Xie, Y.: Towards efficient ansatz architecture for variational quantum algorithms. arXiv preprint arXiv:2111.13730 (2021)
40. Xu, Y., Huang, G., Balewski, J., Naik, R., Morvan, A., Mitchell, B., Nowrouzi, K., Santiago, D.I., Siddiqi, I.: Qubic: An open-source fpga-based control and measurement system for superconducting quantum information processors. IEEE Transactions on Quantum Engineering **2**, 1–11 (2021). https://doi.org/10.1109/tqe.2021.3116540, http://dx.doi.org/10.1109/TQE.2021.3116540
41. Xu, Y., Huang, G., Fruitwala, N., Rajagopala, A., Naik, R.K., Nowrouzi, K., Santiago, D.I., Siddiqi, I.: Qubic 2.0: An extensible open-source qubit control system capable of mid-circuit measurement and feed-forward (2023), https://arxiv.org/abs/2309.10333