QPI: A Programming Interface For Quantum Computers

Ercüment Kaya*†, Burak Mete*†, Laura Schulz*, Muhammad Nufail Farooqi*, Jorge Echavarria*, Martin Schulz†,
*Leibniz Supercomputing Centre of the Bavarian Academy of Sciences and Humanities, Garching, Bavaria, Germany
{ercuement.kaya, burak.mete, laura.schulz, muhammad.farooqi, jorge.echavarria}@lrz.de
†Technical University of Munich, Garching, Bavaria, Germany
{schulzm}@in.tum.de

***Abstract*—With the increasing maturity and accessibility of quantum computers, their alignment, integration, and use in the high-performance computing (HPC) ecosystem as a novel accelerator triggers a crucial new area of research. To address the demands for efficient and tightly coupled programming, we present the Quantum Programming Interface (QPI), a C-based library enabling the development of quantum tasks and submission to quantum resources.**

I. Introduction

High-performance computing (HPC) systems offer enhanced performance and less resource consumption by integrating various devices and architectures. On the other hand, quantum computers (QC), with their unique computing paradigm, hold promises of being accelerators and stand-alone computational approaches, with their inherent capability of tackling problems that would require exponential resources to solve for their classical counterparts. However, it is essential to recognize that both computational paradigms can complement each other; QCs enable solving or accelerating intrinsic problems in quantum computing, while HPC systems pave the way for more optimal computing by handling operational control of QCs, the compilation of quantum circuits, and supporting the parameter optimization of quantum circuits in variational quantum algorithms.

Combining the radically different approaches of HPC systems with quantum computers presents a significant challenge at the software level. Beyond establishing a physical connection, the software stack development enables seamless user interaction between the two systems. Creating a hybrid application requires quantum programming tools (QPTs), which are designed to specify the interaction between the quantum computer and the HPC system. QPTs need to be abstracted from the quantum component at the application layer. Moreover, QPTs must be compatible with existing HPC tools and higher-level programming languages to create a better user experience and facilitate maintenance.

Quantum circuit compilation is another crucial step within the quantum integration software. It transforms high-level quantum algorithms into hardware-specific implementations, optimizing circuit efficiency, potentially reducing errors, and ensuring compatibility with diverse quantum hardware technologies, enabling seamless and efficient utilization of quantum resources for real-world applications. Furthermore, given the possibility of multiple quantum backends employing various underlying technologies and features integrated into the software stack, it is crucial to abstract the compilation layer from both the application layer and the architecture. This is commonly achieved by describing the application in a so-called intermediate representation (IR), thereby adding support for various hardware configurations and addressing common programming requirements such as scheduling and further optimizations.

To tackle these challenges holistically, we present the Quantum Programming Interface (QPI), a lightweight library to embed quantum circuits in HPC applications. QPI enables the acceleration of HPC applications by allowing programmers to describe their quantum or classical-quantum programs within a common programming interface while efficiently leveraging quantum resources, regardless of the quantum device responsible for executing the job thereafter.

QPI is a C-programming interface that allows users to create quantum circuits at a high level of abstraction, which are then converted into an LLVM-compliant IR, allowing seamless communication and execution on various quantum computers and simulators.

Our main contributions are the following:

- We eliminate application and architecture dependencies from quantum circuits and HPC systems, simplifying the creation of quantum circuits
- We provide a holistic approach for hybrid quantum-classical applications
- We abstract the underlying technology of the target QPUs and expose them as local accelerators
- Overall, we offer a novel solution tailored for HPC ecosystems for 1) describing quantum circuits through an interface familiar to most researchers, 2) parsing the quantum algorithm's components into an LLVM-compliant IR, and 3) offloading it to the quantum compiler for its subsequent execution by the targeted quantum accelerator

These contributions are further described in section III.

979-8-3315-4137-8/24/$31.00 ©2024 IEEE
DOI 10.1109/QCE60285.2024.10293

II. Background and Related Work

With the rising adoption of quantum accelerators in HPC ecosystems, there is a growing need for new quantum programming interfaces and frameworks tailored to HPC environments. These interfaces and frameworks aim to enable users to efficiently access and perform computations on such accelerators, thereby promoting the exploration and utilization of quantum capabilities across various applications.

Significant efforts have been dedicated towards the development of quantum programming frameworks [1] to effectively bridge the gap between high-level quantum algorithms, the underlying quantum hardware, and their seamless integration within the context of HPC systems.

Take, for example, ProjectQ [2], an open-source project that uses a Python-embedded language incorporating both quantum compilation tools and the ability to target IBM quantum devices as well as quantum simulators. OpenQL [3], a framework with compilation tools and a high-level description of quantum circuits that could be mapped in various quantum hardware types. Quilc [4], yet another framework providing a target-specific platform explicitly focusing on NISQ-era algorithms. Or ScaffCC [5], a framework that is based on LLVM and designed primarily for quantum circuit compilation and scheduling of quantum jobs.

III. Quantum Programming Interface

Quantum Programming Interface (QPI) is a lightweight C-based library that abstracts hardware dependencies from quantum accelerators and is tailored for HPC systems. Hence, HPC applications can exploit radically different computational paradigms of quantum computers.

C is the dominant programming language for the HPC domain [6], and various HPC development platforms and application programming interfaces (API), such as CUDA and OpenMP, are built in C. Using those APIs makes QPI more tailored for HPC systems.

QPI supports 24 quantum gates, named parameters, and circuit fusing. Named parameters are crucial for HPCQC applications since most of them execute the circuit in a loop and update the parameters based on post-measurement evaluation.

QPI acts as a front-end for the software stack shown in Figure 1. It is integrated into the Munich Quantum Software Stack (MQSS), a comprehensive framework aimed at seamlessly incorporating quantum acceleration into HPC ecosystems [7]. As an open-source component of the MQSS, QPI is accessible at [8]. It supports multiple intermediate representations (IR), facilitating integration with various backends. QPI translates the circuit into target IR, which can be QIR [9] or OpenQASM 2.0 [10]. After the circuit is translated, the generated code is offloaded to a so-called *Offload Listener*.

Bell state, also known as EPR pairs, represents one of the basic examples of quantum entanglement. It is widely used in quantum teleportation and quantum cryptography. Hence, Bell State is one of the cornerstones of quantum information science.

Algorithm 1 shows Bell State's implementation with QPI. The implementation consists of three phases as follows:

1) Application: The users include the library, as demonstrated in Line 1, allowing them to use the additional features. The variable named *circuit* represents a quantum circuit (Line 4). Before inserting the instructions, the circuit needs to be initiated by calling the *qCircuitBegin* function (Line 6). It performs all the required memory allocations and saves the circuit into the global list. The variable named *cr* represents classical registers (Line 8), and the users call the *qInitClassicalRegisters* function to initiate the classical registers (Line 9). The function *qH* applies a Hadamard gate to a given target, which is *qubit 0* for the given example (Line 11). The function *qCX* applies a controlled gate where the first argument is the control, and the second is the target, qubit 0 and 1, respectively, for the given example (Line 12). The *qMeasure* function adds a measurement instruction in the Z-basis for a given qubit specified in the first argument and saves the result in the classical register at the index specified in the third argument (Line 13 and 14). The function *qCircuitEnd* indicates that the circuit is completed and ready to execute (Line 16).

2) Execution: The *qExecute* function mainly handles the execution phase. This function takes the circuit and the number of shots as arguments. The circuit is translated into target representation such as OpenQASM 2.0 [10] and QIR [9]. The circuit is offloaded to the backend through a lightweight daemon process named *Offload Listener*, as shown in Figure 1. The circuit is executed on the available quantum accelerator, and the *Offload Listener* sends the results back to QPI.

3) Result Interpretation QPI represents results in a linked list typed *QuantumResult*. A single element of the list contains the state and the count, as shown in Lines 21 to 25.

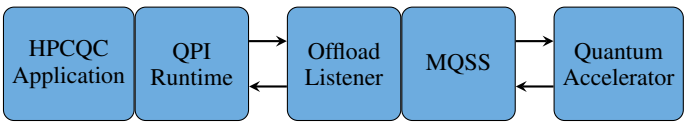

Fig. 1: Illustration of the QPI's communication with interconnected components of the software stack.

A. Ease of Access for Variational Algorithms

Variational quantum algorithms, one of the most promising use cases in quantum computing in NISQ-era, might require high-level and problem-specific details, along with the fundamental requirements for defining a quantum circuit. For instance, Variational Quantum Eigensolver [11], [12] requires an expectation value calculation of a certain molecular Hamiltonian within each iteration. To execute such a circuit on quantum hardware, one must decompose the Hamiltonian operator and perform several measurements equivalent to the total count of Pauli words present in the Hamiltonian operator [13]. However, this poses a significant challenge for programming frameworks since multiple measurements create numerous intermediate representations of the quantum job. Consequently, this complexity may require users to generate

Algorithm 1 Bell State Implementation with QPI.

```
#include <qpi.h>

int main(){
  QCircuit circuit;
  int numberOfShots = 100;
  qCircuitBegin(&circuit);

  QClassicalRegisters cr;
  qInitClassicalRegisters(&cr, 2);

  qH(0);
  qCX(0, 1);
  qMeasure(0, cr, 0);
  qMeasure(1, cr, 1);

  qCircuitEnd();

  int isErr = qExecute(circuit, numberOfShoots);
  if(!isErr) {
    QuantumResult* results = qRead(circuit);
    while(results){
       printf("%s %d\n", results->state, results->count);
       results = results->next;
    }
  }

  qCircuitFree(circuit);

  return 0
}
```

Algorithm 2 Bell State Implementation with Qiskit Provider.

```
def bell_state():
   _provider = HPCOffloadProvider()
   _backend = _provider.get_backend("Q5")

   _circuit = QuantumCircuit(2, 2)
   _circuit.h(0)
   _circuit.cx(0, 1)
   _circuit.measure_all()

   _job = _backend.run(_circuit, shots=1000)
   print(_job.result().get_counts())
```

multiple instances of the same circuit on their end, adding to the overall intricacy of the computational process. The QPI implementation addresses this challenge through its compact format. It enables users to define the ansatz only once and efficiently execute measurements along different bases, ultimately combining them into a single expectation value.

Another aspect of QPI support for variational algorithms is that it supports various options for parameter optimization. It includes the so-called *quantum gradients* [14], through routines such as parameter-shift [15] or SPSA [16], which are used to calculate the optimal parameters in a variational approach without having to define their gradient. They essentially generate multiple quantum circuits whose parameters are slightly perturbed in different directions and calculate the gradients of the original circuit using a finite-difference-like method. QPI natively supports these methods and enables users to send hybrid quantum jobs.

Algorithm 3 Psuedocode for *QAOA*.

```
def qaoa(parameters, H_C, H_B, p):
   # Apply Hadamard to all qubits
   apply_hadamard(qubits)
   # Apply alternating layers of unitaries
   for i in range(p):
      # Apply e^{-iγH_C} = U(C,γ)
      apply_layer(H_C, parameters[2*i], qubits)
      # Apply e^{-iβH_B} = U(C,β)
      apply_layer(H_B, parameters[2*i+1], qubits)
   # Measure the quantum state
   results = measure(qubits)

   # Postprocesss bitstrings into exp. values
   exp_val = convert_counts_to_exp_val(results)
   return exp_val

def optimize_qaoa(H_C, H_B, p):
   parameters = random(p)
   for i in num_iters:
      exp_val = qaoa(parameters, H_C, H_B)
      parameters = optimize(exp_val, parameters)
```

IV. Experimental Study

A. Experimental Setup

Qiskit is a widely adopted software development kit (SDK) for writing quantum circuits in Python, renowned for its accessibility and robust capabilities. To assist the Qiskit user community in transitioning to high-performance computing quantum computing (HPCQC), the MQSS offers the *HPC Offload Provider*. This custom provider plays a crucial role by enabling HPC users to access quantum accelerators, facilitating offloading quantum circuits within HPC environments utilizing Pythonic programming interfaces like Qiskit.

MQSS includes several tools to complement the quantum computation, such as the compiler and scheduler. Therefore, the computation of these tasks is not a part of the benchmarks, as they are all excluded from the QPTs. MQSS has two versions that support and use *QASM 2.0* and *QIR*, respectively. As our *HPC Offload Provider* and QPI are tightly integrated within the MQSS, we will utilize the same provider in the following comparative analyses to ensure fair comparisons.

To evaluate our implementations, we create two applications: Bell State and Quantum Approximate Optimization Algorithm (QAOA) [17] for solving the Max-Cut problem. We compare generated *QASM 2.0* and *QIR* code to measure their accuracy in *Bell State* and the execution times of QAOA.

The implementation of the Bell State is given in Algorithms 6 and 7 using QPI and Qiskit Provider, respectively. The pseudocode for the QAOA is provided in Algorithm 3.

The experiments used the HPC resources at the Leibniz Supercomputing Centre (LRZ), which are integrated with the QC systems. The jobs are submitted to an HPC node; then the quantum parts are offloaded to the quantum accelerator. The targeted quantum accelerator is a 20-qubit superconducting quantum device from IQM. The classical node has a 256 GB memory and features 2 Intel CPU sockets (Intel Xeon Platinum 8360Y CPU @ 2.40GHz), each with 36 physical cores.

Algorithm 4 *QASM 2.0* code generated by QPI.

```
OPENQASM 2.0;
include "qelib1.inc";
qreg q[2];
creg c0[2];
h q[0];
cx q[0], q[1];
barrier q[0],q[1];
measure q[0] -> c0[0];
measure q[1] -> c0[1];
```

Algorithm 5 *QASM 2.0* code generated by Qiskit Provider.

```
OPENQASM 2.0;
include "qelib1.inc";
qreg q[2];
creg meas[2];
h q[0];
cx q[0],q[1];
barrier q[0],q[1];
measure q[0] -> meas[0];
measure q[1] -> meas[1];
```

B. Experimental Results

1) Bell State: The generated *QASM 2.0* code from QPI and Qiskit are given in Algorithms 4 and 5, respectively.

As observed, the only difference is in the classical registers of the *measure* instruction, which does not affect the outcome of the application.

While QPI supports QIR natively, we resorted to *qiskit-qir* [18] for parsing to QIR the results obtained by the Qiskit Provider. The generated QIR codes are given in Algorithms 6 and 7, respectively.

The difference in the classical registers also affects the QIR outcome. Besides that, the function name is also different, and it is by design and does not change the outcome of the circuit. As can be seen, QPI is fully capable of generating accurate codes.

2) QAOA: Quantum Approximate Optimization Algorithm (QAOA) constitutes an example of a hybrid quantum-classical algorithm due to its inherent structure requiring classical computation along with quantum workloads. It is mainly used for solving combinatorial optimization problems, whose classical problem description can be converted into a quantum Hamiltonian [19]. The algorithm starts by initializing all the qubits into an equal superposition state, allowing all the bitstrings to be a valid solution to the original problem. Then, the time evolution w.r.t. the calculated Hamiltonian is parameterized and applied to the principle system, called the *cost unitary*. This part is interleaved by the *mixing unitary*, which allows the full Hilbert space exploration. These two unitaries can be repeated across several layers for more accurate results. A classical optimizer is run to find parameters as close as possible to the optimal. The optimizer can be gradient-based, which requires considerably heavy classical computation, or it can also be calculated numerically by putting the load onto the quantum computer by spawning multiple quantum circuits [16] [15].

Algorithm 6 *QIR* code generated by QPI.

```
define void @EntryPoint() #0 {
entry:
  ;...
  call void @__quantum__qis__h__body(%Qubit* null)
  call void @__quantum__qis__cnot__body(%Qubit* null, %
      ↪ Qubit* inttoptr (i64 1 to %Qubit*))
  call void @__quantum__qis__mz__body(%Qubit* null, %
      ↪ Result* null)
  call void @__quantum__qis__mz__body(%Qubit* inttoptr (
      ↪ i64 1 to %Qubit*), %Result* inttoptr (i64 1 to
      ↪ %Result*))
  ;...
}
```

Algorithm 7 *QIR* code generated by Qiskit Provider.

```
define void @circuit-120() #0 {
entry:
  ;...
  call void @__quantum__qis__h__body(%Qubit* null)
  call void @__quantum__qis__cnot__body(%Qubit* null, %
      ↪ Qubit* inttoptr (i64 1 to %Qubit*))
  call void @__quantum__qis__mz__body(%Qubit* null, %
      ↪ Result* inttoptr (i64 2 to %Result*))
  call void @__quantum__qis__mz__body(%Qubit* inttoptr (
      ↪ i64 1 to %Qubit*), %Result* inttoptr (i64 3 to
      ↪ %Result*))
  ;...
}
```

The experiments involving QAOA are executed over eighteen variations, based on different combinations of *number of nodes*, *number of layers*, and *number of optimization iterations*. We conclude the experiments by measuring the different sections of the application.

The sections are *circuit creation*, *circuit execution*, and *post quantum optimization*. While the first and third sections are entirely classical, the second section mainly depends on the software stack and quantum accelerator. Consequently, during our experiments, we got outlier results, errors that indicate *time out* which arise after waiting 1000 seconds for the results, and warnings about there being no available QPUs. To have more accurate results, we exclude such executions from the experiment.

Table Ia and Ib shows the average execution times of the selections based on different combinations of *number of nodes* (Column 1), *number of optimization iterations*(Column 2), and *number of layers* (Column 3). All results are in seconds and round up to 6-digit precision. Figure 2 visualizes the benchmarks for a various number of qubits, and a fixed number of layers (4) and iterations (50).

We observe that the execution time of a single *circuit creation* increases as the number of qubits and the number of layers is increased in Qiskit Provider and QPI, and the increase in Qiskit Provider is more aggressive compared to QPI. Besides, we observed the average execution time of a single *circuit creation* in QPI is 97% less than Qiskit Provider. However, the average difference between QPI and Qiskit Provider is 0.052 seconds. Even though the improvement rate is high, a significant amount of circuits need to be created to observe the effect. The maximum difference of all *circuit*

N	I	L	Circuit Creation	Circuit Execution	Post Quantum Execution
4	30	3	5.65×10^{-4}	415.4384	4.3×10^{-5}
		4	6.52×10^{-4}	415.2457	3.9×10^{-5}
	40	3	7.59×10^{-4}	552.1834	4.7×10^{-5}
		4	8.77×10^{-4}	552.5055	4.8×10^{-5}
	50	3	9.07×10^{-4}	686.9105	6.4×10^{-5}
		4	1.111×10^{-3}	686.3051	6.6×10^{-5}
6	30	3	8.06×10^{-4}	442.3336	2.94×10^{-4}
		4	1.024×10^{-3}	444.0678	2.9×10^{-4}
	40	3	9.95×10^{-4}	587.9037	3.76×10^{-4}
		4	1.434×10^{-3}	588.1279	3.98×10^{-4}
	50	3	1.211×10^{-3}	732.3930	4.77×10^{-4}
		4	1.434×10^{-3}	737.8179	3.98×10^{-4}
8	30	3	1.151×10^{-3}	472.4381	2.132×10^{-3}
		4	1.393×10^{-3}	474.9329	2.116×10^{-3}
	40	3	1.548×10^{-3}	638.6727	2.848×10^{-3}
		4	1.904×10^{-3}	641.1524	2.838×10^{-3}
	50	3	1.834×10^{-3}	786.6663	3.476×10^{-3}
		4	2.407×10^{-3}	794.3176	3.548×10^{-3}

(a)

N	I	L	Circuit Creation	Circuit Execution	Post Quantum Execution
4	30	3	2.164×10^{-2}	421.2956	1.496×10^{-3}
		4	2.841×10^{-2}	423.2209	2.332×10^{-3}
	40	3	3.155×10^{-2}	560.1304	2.734×10^{-3}
		4	3.784×10^{-2}	560.6924	2.936×10^{-3}
	50	3	3.92×10^{-2}	700.512	3.74×10^{-3}
		4	4.93×10^{-2}	698.699	5.02×10^{-3}
6	30	3	3.382×10^{-2}	446.9151	1.776×10^{-2}
		4	4.375×10^{-2}	447.1684	1.251×10^{-2}
	40	3	4.458×10^{-2}	595.545	1.999×10^{-2}
		4	5.353×10^{-2}	596.5857	2.059×10^{-2}
	50	3	5.579×10^{-2}	736.2345	2.542×10^{-2}
		4	6.808×10^{-2}	748.882	2.774×10^{-2}
8	30	3	5.039×10^{-2}	478.4504	9.374×10^{-2}
		4	6.279×10^{-2}	479.1915	9.281×10^{-2}
	40	3	6.639×10^{-2}	639.8411	1.259×10^{-1}
		4	8.182×10^{-2}	645.4425	1.223×10^{-1}
	50	3	8.34×10^{-2}	802.6442	1.577×10^{-1}
		4	1.038×10^{-1}	803.4487	1.532×10^{-1}

(b)

TABLE I: Average creation and execution time of the sections using (a) QPI and (b) Qiskit Provider. Note that 1) N, I, and L correspond to the number of nodes, number of iterations, and number of layers, respectively, and that 2) the reported creation and execution times are expressed in seconds.

creation executions is 0.1 seconds, which occurred during eight nodes, four layers, and fifty iterations.

The most intensive section is the *circuit execution*. It contains offloading to the *Offload Listener*, waiting for the execution on the QPU to be completed, parsing the response from *Offload Listener*, and returning the results to the HPCQC application. Qiskit Provider and QPI use the same protocol to utilize the quantum accelerators. The differences that cause

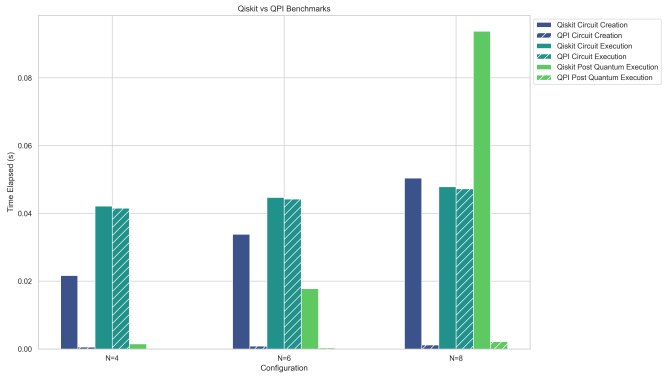

Fig. 2: Comparison of Qiskit and QPI benchmarks for different values of N. N represents the number of nodes in the problem, or similarly the number of qubits for the algorithm. The plot shows the time elapsed for Circuit Creation, Circuit Execution, and Post Quantum Execution for different qubit numbers, with 50 iterations and 4 layers. The values for Circuit Execution have been scaled down by a factor of 10,000 for better readability. The same colors are used for corresponding categories in both Qiskit and QPI, with different hatching patterns to differentiate between them.

execution time difference are serialization of the *QuantumTask* and deserialization of *QuantumResult*.

We observe that the execution time of a single *circuit execution* increases as the number of qubits and layers increases in Qiskit Provider and QPI. The increase rate caused by the increase in the number of qubits is more aggressive than the increase in the number of layers. Besides that, even though the average execution time of a single *circuit execution* in QPI is 1.2% less than Qiskit Provider, the average execution time of this section takes 7.4 seconds using Qiskit Provider, and it has called 61 to 101 times per execution. Therefore, it can significantly affect the total execution time of the application. In our experiments, it affects up to 15.97 seconds.

The section *post quantum optimization* is similar to *circuit creation*. The improvement rate in *post quantum optimization* is as high as *circuit creation*. The section is not compute-intensive to show a difference more prominent than 0.1 seconds. However, it is clear that as compute-intensive increases in the section, the difference would be more significant.

V. Conclusion and Future Works

This work presents QPI, a C-based library for developing and targeting quantum jobs into quantum resources. Unlike other programming interfaces and libraries, we aim to design it to be tailored for HPC systems. Thanks to its lightweight nature, our experiment shows notable improvement in the classical and quantum sides. However, there is room for improvement.

Our future works are described as follows:

- The main known bottleneck of QPI is string manipulations to create OpenQASM 2.0 code. We wish to

improve on that by providing full support for the non-string manipulation-based intermediate representations.
- We wish to implement known algorithms such as QFT and Shor's as off-to-shelf solutions to provide a better development kit.
- To capture more complex quantum circuit generation dynamics, one of the next goals is to implement and benchmark methods where the circuit grows dynamically [20].
- Generating MLIR code as an intermediate representation.
- Currently, the target QPU cannot be specified by the interface and is assigned by the inherent scheduling mechanism of the MQSS, but we wish to allow the users to assign the target QPU among the available devices

VI. Acknowledgement

This work is funded by the German Federal Ministry for Education and Research under grants 13N15689 (DAQC), 13N16063 (Q-Exa), 13N16188 (MUNIQC-SC), and 13N16078 (MUNIQC-ATOMS), and the Bavarian State Ministry of Science and the Arts as part of Munich Quantum Valley (MQV). Further, we want to acknowledge the contribution done by Dr. Martin Ruefenacht while he was at LRZ.

References

[1] R. LaRose, "Overview and comparison of gate level quantum software platforms," *Quantum*, vol. 3, p. 130, 2019.

[2] D. S. Steiger, T. Häner, and M. Troyer, "Projectq: an open source software framework for quantum computing," *Quantum*, vol. 2, p. 49, 2018.

[3] N. Khammassi, I. Ashraf, J. Someren, R. Nane, A. Krol, M. A. Rol, L. Lao, K. Bertels, and C. G. Almudever, "Openql: A portable quantum programming framework for quantum accelerators," *ACM Journal on Emerging Technologies in Computing Systems (JETC)*, vol. 18, no. 1, pp. 1–24, 2021.

[4] R. S. Smith, E. C. Peterson, M. G. Skilbeck, and E. J. Davis, "An open-source, industrial-strength optimizing compiler for quantum programs," *Quantum Science and Technology*, vol. 5, no. 4, p. 044001, 2020.

[5] A. JavadiAbhari, S. Patil, D. Kudrow, J. Heckey, A. Lvov, F. T. Chong, and M. Martonosi, "Scaffcc: A framework for compilation and analysis of quantum computing programs," in *Proceedings of the 11th ACM Conference on Computing Frontiers*, 2014, pp. 1–10.

[6] I. Laguna, R. Marshall, K. Mohror, M. Ruefenacht, A. Skjellum, and N. Sultana, "A large-scale study of mpi usage in open-source hpc applications," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2019, pp. 1–14.

[7] M. Schulz, H. Ahmed, X. Deng, J. Echavarria, M. Gammelmark, S. Karlsson, E. Kaya, M. Reznak, L. B. Schulz, and M. Tovey, "From the physics lab to the computer lab: Towards flexible and comprehensive devops for quantum computing," in *Proceedings of the 21st ACM International Conference on Computing Frontiers: Workshops and Special Sessions*, ser. CF '24 Companion. New York, NY, USA: Association for Computing Machinery, 2024, p. 139–143. [Online]. Available: https://doi.org/10.1145/3637543.3653432

[8] Munich Quantum Software Stack. (2024) Munich Quantum Software Stack. [Online]. Available: https://github.com/Munich-Quantum-Software-Stack

[9] QIR Alliance, *QIR Specification*, 2021, also see https://qir-alliance.org. [Online]. Available: https://github.com/qir-alliance/qir-spec

[10] A. W. Cross, L. S. Bishop, J. A. Smolin, and J. M. Gambetta, "Open quantum assembly language," *arXiv preprint arXiv:1707.03429*, 2017.

[11] A. Peruzzo, J. McClean, P. Shadbolt, M.-H. Yung, X.-Q. Zhou, P. J. Love, A. Aspuru-Guzik, and J. L. O'brien, "A variational eigenvalue solver on a photonic quantum processor," *Nature communications*, vol. 5, no. 1, p. 4213, 2014.

[12] J. Tilly, H. Chen, S. Cao, D. Picozzi, K. Setia, Y. Li, E. Grant, L. Wossnig, I. Rungger, G. H. Booth *et al.*, "The variational quantum eigensolver: a review of methods and best practices," *Physics Reports*, vol. 986, pp. 1–128, 2022.

[13] A. Jena, S. N. Genin, and M. Mosca, "Optimization of variational-quantum-eigensolver measurement by partitioning pauli operators using multiqubit clifford gates on noisy intermediate-scale quantum hardware," *Physical Review A*, vol. 106, no. 4, p. 042443, 2022.

[14] M. Schuld, V. Bergholm, C. Gogolin, J. Izaac, and N. Killoran, "Evaluating analytic gradients on quantum hardware," *Physical Review A*, vol. 99, no. 3, p. 032331, 2019.

[15] D. Wierichs, J. Izaac, C. Wang, and C. Y.-Y. Lin, "General parameter-shift rules for quantum gradients," *Quantum*, vol. 6, p. 677, 2022.

[16] J. C. Spall, "A one-measurement form of simultaneous perturbation stochastic approximation," *Automatica*, vol. 33, no. 1, pp. 109–112, 1997.

[17] E. Farhi, J. Goldstone, and S. Gutmann, "A quantum approximate optimization algorithm," *arXiv preprint arXiv:1411.4028*, 2014.

[18] Microsoft, "qiskit-qir." [Online]. Available: https://github.com/microsoft/qiskit-qir

[19] W. M. Kirby and P. J. Love, "Variational quantum eigensolvers for sparse hamiltonians," *Physical review letters*, vol. 127, no. 11, p. 110503, 2021.

[20] H. R. Grimsley, S. E. Economou, E. Barnes, and N. J. Mayhall, "An adaptive variational algorithm for exact molecular simulations on a quantum computer," *Nature communications*, vol. 10, no. 1, p. 3007, 2019.