A Software Platform to Support Disaggregated Quantum Accelerators

Ercüment Kaya[‡†], Jorge Echavarria[‡], Muhammad Nufail Farooqi[‡], Aleksandra Swierkowska[‡†], Patrick Hopf[‡†◇], Burak Mete[‡†], Lukas Burgholzer[†], Robert Wille[†], Laura Schulz[‡], Martin Schulz[‡†],
[‡]Leibniz Supercomputing Centre of the Bavarian Academy of Sciences and Humanities, Garching, Germany
[†] Technical University of Munich (TUM), Munich, Germany
[◇] Ludwig-Maximilians-Universität München (LMU), Munich, Germany

Email: {ercuement.kaya, jorge.echavarria, muhammad.farooqi, aleksandra.swierkowska, patrick.hopf, burak.mete, laura.schulz}@lrz.de[‡]
{lukas.burgholzer, robert.wille, martin.w.j.schulz}@tum.de[†]

Abstract*—Quantum computers are making their way into High Performance Computing (HPC) centers as next-generation accelerators. Due to their physical implementation as mostly large appliances in separate racks, their number in typical data centers is significantly lower than the number of nodes offloading work to them, unlike the case with GPU accelerators. As a consequence, they form large-scale disaggregated infrastructures that pose a number of integration challenges due to their diverse implementation technologies and their need to be used as a shared resource for optimal utilization. Running hybrid High Performance Computing-Quantum Computing (HPCQC) applications in HPC environments, where the quantum portion is offloaded to the quantum processing units (QPUs), requires sophisticated resource management strategies to optimize resource utilization and performance. In this paper, we present one aspect of the Munich Quantum Software Stack (MQSS) - a Just-In-Time (JIT) compilation and execution software stack for quantum and hybrid quantum-HPC workloads - beneficial for integrating disaggregated quantum accelerators into traditional HPC workflows. MQSS is centered around a series of novel contributions aimed at optimizing the compilation process while managing the disaggregated resources ensuring efficient utilization of all available quantum resources. The key stages of our JIT compilation stack involve hardware-agnostic optimizations, quantum circuit cutting – when necessary – and a novel hardware platform selection process. Included here is description of the use of the MQSS's Quantum Device Management Interface (QDMI), a unified interface between the software stack and the quantum accelerators, which allows connection to several, potentially different, quantum resources. Our methodology demonstrates notable advancements in HPCQC integration workflows, offering researchers a powerful framework for leveraging quantum computational capabilities. Overall, our work represents a significant step towards the practical integration of disaggregated quantum devices within the realm of supercomputing, unlocking new avenues for computational exploration and discovery.

***Index Terms*—Quantum Systems, Quantum Computing, HPCQC Integration, Disaggregated Infrastructures, Hybrid Software Stacks**



Fig. 1: A view into the Quantum Integration Centre (QIC) at LRZ/Munich showing a superconducting system (left), an ion-trap system (middle) and HPC racks covering the classic compute. The result is a strongly disaggregated infrastructure combing classical HPC clusters with large-scale accelerator appliances, which nevertheless need to operate as a seemless, single system.

I. Introduction

The decreased performance pace caused by the end of Dennard scaling and the looming conclusion to Moore's law strongly contrasts the rapidly increasing computation demands for many critical applications in science and technology leveraging HPC and AI methods. While GPUs and other specialized accelerators somewhat alleviate this pressure, they do not fundamentally solve the situation at hand.

Quantum Computing could potentially offer some relieve, emerging as a promising path towards new algorithms capable of solving computational problems using lower complexity classes. However, quantum computing targets only specific problems suitable for quantum formulation, requires classical (i.e., HPC) systems for their control and I/O, and relies

This work was funded by the German Federal Ministry of Education and Research (BMBF) under the funding program *Quantum Technologies - From Basic Research to Market* under contract numbers 13N16078, 13N15689, 13N16187 and 13N16087, as well as from the Munich Quantum Valley (MQV), which is supported by the Bavarian State Government with funds from the Hightech Agenda Bayern.

979-8-3503-5554-3/24/$31.00 ©2024 IEEE
DOI 10.1109/SCW63240.2024.00205

on computationally demanding algorithms for compilation, optimization and place and route. Consequently, quantum computing works only as an accelerator technology in close connection with classic HPC, forming hybrid systems. A tight, low-latency connection is therefore imperative both to reduce round-trip latencies for iterative algorithms, like QAOA approaches, and to ensure an efficient usage during compilation and execution.

Quantum systems, however, differ in form from usual accelerators, like GPUs or FPGAs, as typically large systems in separate cabinets, some connected to cryogenic fridges, and with their need for separate racks of electronics for laser and/or microwave control[1]. Figure 1 gives a small view into such a shared HPC and QC installation. As a consequence, hybrid HPCQC systems are, by their nature, disaggregated and require a mapping of one (or ideally, few) quantum resources to a large number of compute nodes.

Operating such hybrid HPCQC systems, therefore, requires a carefully structured software infrastructure able to manage the disaggregated structure and support the efficient execution of quantum codes from within classic nodes. This requires new techniques for scheduling, dynamic and on-demand computation, dynamic backend resource selection, triggering and control of circuit execution, as well as run time measurements and processing.

The Munich Quantum Software Stack (MQSS) [1] provides such a solution, enabling the execution of quantum kernels, possibly embedded in native HPC applications, on disaggregated quantum infrastructures that are connected to large scale HPC systems. Its core infrastructure can be executed either within a cloud or cloud-like servers or directly on the HPC node that triggers the quantum kernel. It applies dynamic scheduling and device selection using properties of the quantum kernel, triggers the needed dynamic compilation using an LLVM-inspired and Quantum Intermediate Representation (QIR)-based [2] compilation framework, transfers it via a portable interface to the respective backend, and then processes and transmits back the result of the execution. This offers a flexible, extendable and efficient solution to integrate and operate HPCQC systems and notably enables the use of a range of different, disaggregated backends.

In particular, we make the following contributions:

- We describe the properties of hybrid HPCQC systems as disaggrated accelerators (Section II)
- We introduce concrete requirements for the needed software stack and introduce the MQSS (Section III-A).
- We detail the implementation of the MQSS framework and its components (Section III).
- We provide a quantitative evaluation of the MQSS and the minimal overheads it creates (Section IV).

Overall, we present a flexible solution that helps data centers support disaggregated infrastructures combing HPC and large-scale quantum systems. It provides a flexible solution, covering scheduling, compilation, optimization and execution, of hybrid HPCQC applications.

[1]It should be noted that concepts of quantum systems on PCI cards exist – typically using diamond-vacancy techniques – but those are very limited in size and functionality.

II. Quantum Computers as Disaggregated Resource

Quantum computers offer a radically different accelerator technology with the potential to change the complexity class of several problems from factorization to optimization, from quantum linear solvers to quantum machine learning. With this potential power, however, comes new requirements on users, compute centers and software developers.

A. Quantum Computing

Quantum computing builds on qubits as its basic type of information. It draws its power from the fact that qubits can represent any state of its two states $|0>$ and $|1>$ simultaneously (superposition) and that two or more qubits can be combined (entangled) in a way that their state cannot be represented by the same number of individual qubits anymore; hence, they can be manipulated together. This foundation enables the creation of new algorithms that can explore large parameter spaces concurrently and ultimately will allow the implementation of NP-hard problems in polynomial time. For more details on quantum computing, please refer to Kaye et al. [3].

Quantum computers leveraging these properties can be implemented in different ways; the most common of these modalities are superconducting qubits (SC), trapped ions (IT), and neutral (or cold) atoms (NA). Each has quite different execution properties in terms of execution speed, number of supportable qubits, physical environment, fidelity and coherence times. Which modality proves most advantageous is still an open research question and current assumption is that it will depend on the targeted quantum kernel and its properties. This then necessitates the support of many modalities within one HPC infrastructure.

In all cases, qubits are implemented using a respective medium and then manipulated by an external control system. The latter uses, in most cases, pulses either based on microwaves (for SC systems) or lasers (for IT or NA systems) directed at the qubits which manipulate their state to entangle them and to measure their current state. Each pulse sequence is represented by a gate, which are operations that manipulate qubit state. Gates are then combined into sequences, called circuits, which form the currently most common abstraction level for quantum programs.

B. Quantum Acceleration as Part of HPC

What the modalities all have in common, though, is the fact that they are only suitable for very specialized problems that are in a high complexity class for classic computing, and that can be represented by quantum algorithms so that the computational complexity can be reduced. They are suitable for neither general computation nor polynomial problems as the sheer size of classic systems ensures they will outperform

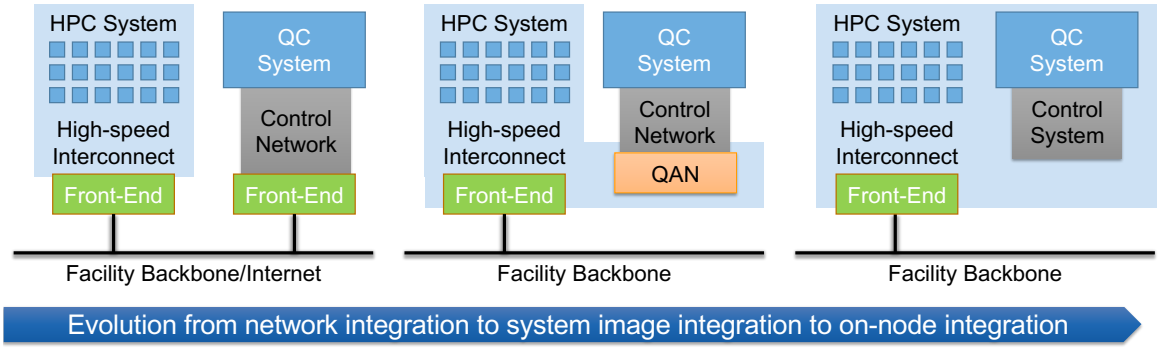


Fig. 2: Different options to combine HPC and QC resources: from loose coupling (left) with HPC and QC systems only connected via a backbone, to network-level integration (middle) by pulling the QC system into the HPC high speed network, and to a full integration into a single system (right).

any quantum solution[2]. This means they are a specialized accelerator solution. At the same time, the preparation of quantum circuits (compilation, optimization and transpilation to native gate sets) as well as the needed execution environment (in terms of I/O and control) requires strong classic systems.

Several models for integration exist [4, 5], ranging from loose coupling using remote access to tightly coupled on-premise solutions (see also Figure 2, which illustrates this spectrum). We are particularly aiming at tightly coupled solutions (right side of the figure), which enables us to treat the entire accelerated system with a single-system view and operation. This has not only the advantage of low latencies for kernel offloads, but also allows the software stack to be executed in physical and logical proximity to the quantum systems.

C. Disaggregated Infrastructure

As a consequence, the operation of quantum systems must rely on a tight connection with HPC or HPC-like classic systems that support the development environment and that act as the host for the acceleration. At the same time, however, quantum systems are typically large-scale appliances. What started as physics experiments in laboratory environments has morphed to more condensed solutions in rack form, but often still requires significant engineering features like cryogenic systems or anti-vibration suspensions. Figure 1 shows the Quantum Integration Centre (QIC) at LRZ in Munich to illustrate the issues with the physical setup.

The result is a strongly disaggregated infrastructure, even within the same room, but which needs to be operated as a single, tightly coupled system from the users' perspective. This requires new software efforts to schedule, allocate and execute the quantum kernels while maintaining efficiency for the coupled HPC components.

[2]At least for the foreseeable future.

III. The Munich Quantum Software Stack

The Munich Quantum Software Stack (MQSS) [1] is a comprehensive framework we have designed to seamlessly integrate quantum acceleration into the HPC ecosystems. It forms a bridge between the HPC system, which serves as a host, and one or more disaggregated large-scale quantum appliances. The software stack itself is highly flexible and extensible, supports any kind of quantum technology via a plug-in concept, and is intended to be deployed within the HPC systems whether as a stand-alone or as a cloud solution. With that, it offers a novel approach to harnessing the power of quantum computing systems within existing computational infrastructures.

A. Guiding Design Principles

The design of the MQSS was governed by several requirements, which were collected from vendor partners, users, quantum developers and system software experts. This then translated into the following design principles:

- The software stack is designed to offer different programming models and abstractions on top of the same software infrastructure core. It does so by decoupling the abstraction from the compilation system, rather than building end-to-end stacks per abstraction and/or system.
- Abstractions must support both existing, quantum-only models, but also HPCQC models that enable execution directly from the HPC node.
- Scheduling is critical to support disaggregated execution efficiently. In particular, it must be distinct from the HPC job scheduler, typically designed to block resources for the entire application run time. Instead, we require a multi-level resource management scheme that enables efficient sharing of the disaggregated resources among multiple processes, jobs or users.
- Quantum computing requires most of the development environment (in particular the compiler) to be executed at run time, after the quantum kernel is generated and before its execution. This requires separately available resources

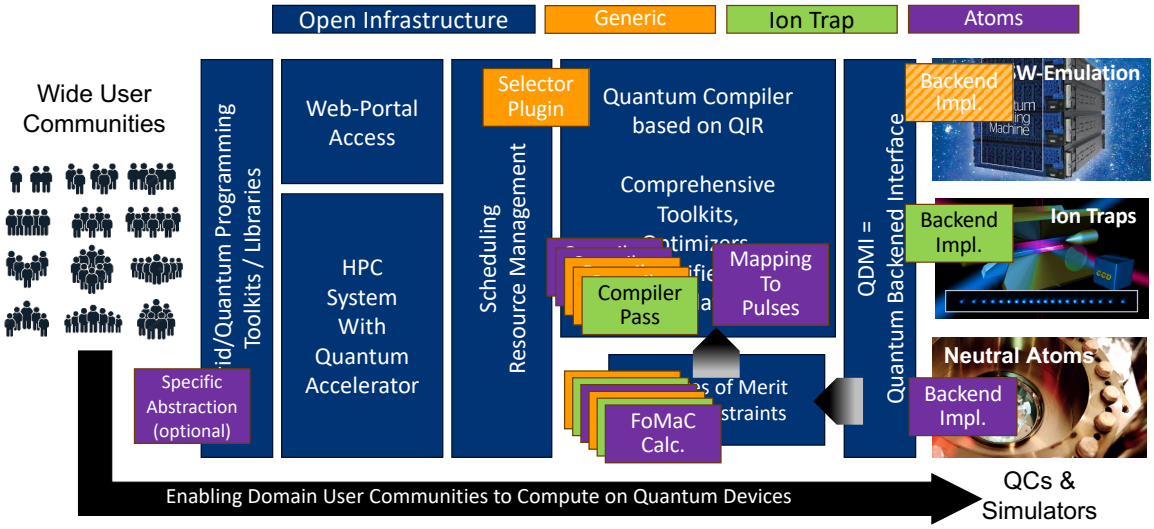


Fig. 3: A schematic overview of the Munich Quantum Software Stack (MQSS), connecting the end users in science domains (left) with one or more quantum systems (right). In between, the MQSS provides the needed workflow for specifying programs using abstraction, to provisioning and scheduling, to compilation, and finally deployment. The system is extensible via plugins, which can be platform neutral or they can support a particular modality (from Schulz et al. [1]).

that must be considered when scheduling and that must be sufficiently powerful to avoid high compilation times.
- While the number of quantum appliances will be significantly smaller than the number of HPC nodes deployed, large centers are expected to host several such systems, ideally with different modalities. The MQSS must be designed to support this and to support flexible decisions on which system to use.
- Finally, the system must be extensible, both to grow with the still nascent technology, but also to be adaptable and specializable for different modalities and vendors.

B. Architectural Overview

Following these design guidelines, we develop the MQSS [1]. We use different programming languages and libraries. Even though C++ is the primary used language, C and Python are also used. To support specific infrastructures, the LLVM and RabbitMQ libraries are used. Its main goal is to connect end users in the traditional HPC science domains to a set of quantum systems available either at, or through, the site to facilitate the execution of hybrid HPCQC applications. An overview of this concept is shown in Figure 3. It consists of an open-core infrastructure that implements the workflow along with a set of plugins that enable us to dynamically customize the infrastructure for specific modalities or systems, integrate new compiler techniques, or to add programming abstractions. Further, at the backend, it features a new interface, the Quantum Device Managament Interface (QDMI) [6], which can be used to integrate any number and type of quantum device into the infrastructure.

C. Components and Workflow

The MQSS supports the entire workflow from the end user (left in the figure) to the systems (right in the figure) and back. The user specifies their quantum kernel either directly in the form of a quantum circuit or using a higher-level abstraction. In general, the MQSS is compatible with any programming model as a frontend that can translate a quantum algorithm into the Intermediate Representations (IRs) that it supports, that is, MLIR [7] and QIR [2].

Currently, we support three programming interfaces as frontends, QPI [8], CUDA-Q [9], and a custom Qiskit Provider [10]. Combined, they demonstrate the broad spectrum of supported solutions: from ones that are available as native C libraries and, hence, are capable of in-code HPC integration, to ones that are designed for existing QC workflows and experimental studies based on Python.

The frontends transform the quantum circuit to QIR. From there, it is passed to the Quantum Resource Manager (QRM) with its distinct stages, each of them designed to tackle the NP-completeness inherent in combinatorial problems encountered in quantum acceleration. The first stage, the **Target-Agnostic Optimization Stage**, aims to streamline the submitted quantum circuit, reducing computational complexity and paving the way for subsequent processing. This stage consists of the selection of the optimization passes and running the selected passes. At the current time, the selection of the optimization passes is fixed, which means the same passes are executed on each quantum circuit. To improve this situation, MQSS will adopt a smarter *pass selector* [11]. Following this, the **Generation Stage** divides a parent circuit into child circuits, ensuring it fits into available quantum accelerators. The **Scheduling**

Algorithm 1 Simplified version of the Quantum Resource Manager (QRM) optimization stages

```
1  void qrm(Module *qirmod, QDMI_Device dev,
       ↪ int numshots)
2  {
3     QDMI_Fragment frag;
4     QDMI_Job job;
5     QDMI_status status;
6
7     QIR_pass_t *pass = NULL;
8
9     QRM_agnostic_selector(selector_a, pass);
10    pass = NULL;
11    while(pass = next_pass(pass))
12       pass->run(qirmod);
13
14    QRM_specific_selector(selector_s, pass);
15    pass = NULL;
16    while(pass = next_pass(pass))
17       pass->run(dev, qirmod);
18
19    QDMI_control_pack_qir(qirmod, &frag);
20    QDMI_control_submit(dev, frag, numshots,
         ↪ &job);
21    QDMI_control_wait(dev, job, &status);
22    //...
23 }
```

Stage comes right after the Generation Stage and assigns a target device to each generated child circuit. Currently, the scheduler assigned the first available device to the given task. An ideal scheduler needs to communicate with the available target devices and assign tasks based on the availability and characteristics of the circuit. To ensure this, the MQSS will adopt a more sophisticated scheduler. Once the scheduler has selected a target architecture for each child circuit, the **Target-Specific Optimization Stage** performs additional transformations to ensure compliance with the chosen accelerator. This process guarantees that each circuit is expressed exclusively with gates supported by the target device and properly mapped to its topology. Both the Target-Agnostic and Target-Specific Optimization Stages are illustrated in Alg. 1. Each of the stages is driven by a set of compiler, optimization, or analysis passes that are loaded into the QRM dynamically as plugins.

The result of the MQSS process is an optimized quantum circuit in the native form of the targeted accelerator, which is offloaded to the assinged backend through the QDMI. The QDMI [6] is an open interface orchestrating the hand-of with the respective backend. It consists of several interfaces: most importantly is a control interface that uses a set of queues and to submit circuits coupled with flexible measurements. These are then gathered, analyzed and at the end passed to the end user.

D. Deployment Options

The MQSS is a flexible stack that can be deployed in different forms, depending on the underlying infrastructure and the site's needs. At LRZ, the MQSS is currently deployed as a cloud-like service on a separate machine, so that the QC system is ready and available when needed. Ultimately, though, we will migrate the MQSS onto the HPC nodes themselves, so it can be activated with local access when a programmer indicates that a job will use quantum resources.

On the other hand, it is also possible to run the stack separately for single user instances, be it on the HPC system or on local systems. This facilitates testing and development and enables small infrastructures.

Which form the MQSS will take is decided during installation. At that time, the system can be configured to either establish the entire MQSS as a single block, or to decouple some components and replace their interface with a remote execution option. In this way, a user of the MQSS can customize the setup directly to the site's needs.

Algorithm 2 Bell state implementation with QPI

```
1  #include <qpi.h>
2
3  int main(){
4    QCircuit circuit;
5    int numshots = 100;
6    qCircuitBegin(&circuit);
7
8    QClassicalRegisters cr;
9    qInitClassicalRegisters(&cr, 2);
10
11   qH(0);
12   qCX(0, 1);
13   qMeasure(0, cr, 0);
14   qMeasure(1, cr, 1);
15
16   qCircuitEnd();
17
18   int isErr = qExecute(circuit, numshots);
19   if(!isErr) {
20     QuantumResult* results = qRead(circuit);
21     while(results){
22        printf("%s %d\n", results->state,
             ↪ results->count);
23        results = results->next;
24     }
25   }
26
27   qCircuitFree(circuit);
28
29   return 0
30 }
```

IV. Experimental Setup

To demonstrate and evaluate the MQSS, we use the setup at LRZ, which features several classical and quantum systems. We show that the MQSS can successfully target multiple backends within the larger disaggregated infrastructure and we evaluate the overhead induced by the infrastructure itself.

For this purpose, we install the QRM component of the MQSS in the *Compute Cloud* resources at the Leibniz Supercomputing Centre (LRZ) as a service, which can be accessed by any system, including from the HPC nodes of our testbed

Accelerator	QExa20	QLM	Q20	Q5
Circuit Creation	0.013	0.012	0.013	0.013
Agnostic Opt.	2,039.492	2,002.971	2,026.1	2,074.573
Generator	56.715	56.075	54.768	53.25
Scheduler	49.412	43.38	48.001	49.565
Execution	4,671.053	2,703.299	5,990.4	6,285.92
QRM Execution	7,532.667	5,233.634	8,217.7	8,562.327
Total Execution	7,550.254	5,251.331	8,234.8	8,580.013

TABLE I: Average creation and execution time of the sections in milliseconds.

cluster. The virtual node used for the QRM has 18 GB of memory and features four virtual CPUs.

On the quantum side, we target four quantum systems, which are available at LRZ:

- The Q-Exa system, a 20-qubit superconducting quantum device from IQM (QExa20) installed in LRZ's production environment and intended for continuous operation.
- A 38-qubit Qaptiva quantum systems emulator from Eviden (QLM).
- A second 20-qubit superconducting quantum device from IQM, installed in the Quantum Integration Centre (QIC) intended for more experimental purposes (Q20).
- A 5-qubit superconducting quantum device from IQM, also installed in the QIC (Q5).

This diversity of systems represents the targeted disaggregated infrastructure very well, in which multiple devices of different capabilities are available, but their number is significantly less than the number of nodes, jobs, or processes.

On the benchmark side, we use a very simple, synthetic quantum circuit that creates a Bell State. It is implemented using the QPI interface. The implementation is given in the Alg. 2. By using a simple code, we reduce the time spent on work directly for the code, which allows us to emphasize – for demonstration purposes – the time spent in the infrastructure itself.

V. Experimental Results

We execute the Bell state code ten times and measure the execution times of the different stages in the MQSS. For this purpose, we instrument the QRM code in a way that we can distinguish the phases *circuit creation*, *generator*, *scheduler*, *agnostic optimization*, and *execution* on the target quantum system. We measure the execution time of each individual phase, along with the overall execution time of the QRM. Furthermore, we also assess the total execution time of the Bell state application. The latter two full end to end measurements enable us, in comparison with per-phase measurements, to identify possibly hidden overheads not covered in the phases.

Table I shows the average execution times of sections, along with the overall execution time of the QRM and the application. All results are in milliseconds and round up to 3-digit precision. This first of all demonstrates that MQSS can target a wide range of resources – in the same room but also remotely – and can successfully orchestrate this distributed execution on disaggregated resources.

Looking at the quantitative data, we observe that – comparing the devices – the most significant difference is the execution time on the QPU, which is not surprising. The difference is expected since the *QExa20* is a real quantum device and the *QLM* is a quantum systems emulator. The other differences are negligible.

Besides the execution on the quantum systems, the most time-consuming section is the *agnostic optimization*. During this section, we execute 24 LLVM passes, i.e., 24 optimization and transformation steps that take the initial code to a final quantum circuit to be executed. Similar to classical compilers, it is not clear that all passes are helpful in all scenarios, which offers one avenue for improvement. To address this issue, we are working on more intelligent pass-selection algorithms. This is, however, independent of the MQSS core infrastructure, as such a selection can simply be loaded as a plugin.

The remaining phases of the QRM show only minimal overhead, demonstrating the efficiency of the chosen approach. However, it should be noted that we are currently only using a very small circuit, which reduces the actual processing times and with that emphasizes the overheads of the infrastructure. This will change, as circuits grow in width and depth, as this will lead to more computational requirements for the actual processing, while the management overhead in MQSS is expected to remain more or less constant. This stresses the need for improved processing algorithms as well as their implementation using HPC techniques, e.g., parallelization. The MQSS supports these options and hence improved processing passes can easily be deployed.

VI. Related Work

The integration of QC resources within traditional HPC ecosystems has gained significant attention in recent literature [12]. One of the early works is ScaffCC compiler [13] for the quantum programming language named Scaffold [14]. Similar to MQSS, this compiler is also LLVM-based, however, both the classical and the quantum parts are translated to QASM. Moreover, unlike MQSS, ScaffCC only supports its programming language.

Qiskit [15] is the most popular quantum toolkit. Users can send quantum circuits to the IBM Quantum cloud platform or simulate them locally. However, it only allows users to create hybrid applications using Python programming language.

Amy and Gheorghiu [16] present *staq*, a comprehensive solution offering a full-stack approach to quantum optimization, transpilation, and device mapping. Similar to MQSS, *staq* leverages the power of standard C++ and comprises a suite of tools, each designed to tackle specific tasks using state-of-the-art methodologies. However, one difference and advantage of MQSS is its tailored approach to HPCQC by relying on the community standard QIR, while *staq* opts for the more specialized and proprietary QASM (from IBM). Further, it lacks customization abilities for HPC environments. Although *staq* provides a robust toolkit for quantum circuit processing,

its reliance on Quantum Assembly Language (QASM) and lack of optimisation for HPC environments will limit its applicability in certain contexts.

Despite notable progress in HPCQC integration frameworks and the development of JIT (Just-in-Time) compilation and optimization techniques, several gaps remain in the existing literature. One prominent gap, for example, pertains to the comprehensive management of quantum resources within heterogeneous computing environments. Existing approaches often lack robust mechanisms for orchestrating quantum job submissions, optimizing resource allocation, and dynamically adapting compilation strategies to evolving device characteristics. The MQSS is desiged from the ground up to cover these issues and can be extended with plugins to improve pass performance.

VII. Summary

After decades of hard work in physics laboratories, both academia and industry are interested in quantum computers. While the developments in quantum computers are highly promising, quantum computers show improvements in certain problems, especially problems that would require exponential resources to solve for their classical counterparts.

It is highly expected that quantum computers will provide significant improvements in artificial intelligence and machine learning due to their high resource usage.

With their unique computing paradigm, they can serve as an accelerator for HPC systems. Due to their nature, many come as large-scale appliances, lending to a setup as disaggregated resources. Regardless, we must operate them as a single system with a unified software environment.

To tackle this issue, this paper presented the capabilities of the Munich Quantum Software Stack (MQSS) to support disaggregated quantum acceleration infrastructures. It is designed to connect domain end users to one or more quantum systems, covers the entire workflow from end-to-end, and is designed to be expandable by offering a comprehensive plugin infrastructure. Further, it can be deployed in different configurations from a stand-alone setup to a cloud-based service solution and ultimately on the HPC nodes themselves for lowest latency.

We demonstrated the operation of the MQSS using the infrastructure at the Leibniz Supercomputing Centre (LRZ), which features three different quantum accelerators plus a commercial grade simulator, all of which can be addressed via the MQSS. Our experiments have shown that the overhead of the infrastructure itself is small, but also that it will be essential in the future to focus on performance optimizations of the plugins deployed as part of the MQSS.

Overall, the MQSS offers a comprehensive software infrastructure that can be used to drive hybrid HPCQC systems, hiding their physical disaggregation from the users. This opens the path to efficient use of HPCQC and with that offers new paths for applications to exploit this new technology.

Acknowledgments

We kindly thank our partners in the Munich Quantum Valley (MQV), in particular the Q-DESSI and QACI projects, for many inspiring discussions and common effort on the MQSS.

References

[1] M. Schulz, L. Schulz, M. Ruefenacht, and R. Wille, "Towards the munich quantum software stack: Enabling efficient access and tool support for quantum computers," in *2023 IEEE International Conference on Quantum Computing and Engineering (QCE)*, Bellevue, WA, USA, September 2023, pp. 399–400. [Online]. Available: https://doi.org/10.1109/QCE57702.2023.10301

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

[3] P. Kaye, R. Laflamme, and M. Mosca, *An Introduction to Quantum Computing*. Oxford Academic, 2006. [Online]. Available: https://doi.org/10.1093/oso/9780198570004.002.0001

[4] A. Elsharkawy, X.-T. M. To, P. Seitz, Y. Chen, Y. Stade, M. Geiger, Q. Huang, X. Guo, M. A. Ansari, C. B. Mendl, D. Kranzlmüller, and M. Schulz, "Integration of quantum accelerators with high performance computing – a review of quantum programming tools," 2023, accepted for publication by ACM Transactions on Quantum Computing (TQC). [Online]. Available: https://arxiv.org/abs/2309.06167

[5] A. Elsharkawy, X.-T. M. To, P. Seitz, Y. Chen, Y. Stade, M. Geiger, Q. Huang, X. Guo, M. A. Ansari, M. Ruefenacht, L. Schulz, S. Karlsson, C. B. Mendl, D. Kranzlmüller, and M. Schulz, "Challenges in hpcqc integration," in *2023 IEEE International Conference on Quantum Computing and Engineering (QCE)*, vol. 02, 2023, pp. 405–406.

[6] R. Wille, L. Schmid, Y. Stade, J. Echavarria, M. Schulz, L. Schulz, and L. Burgholzer, "Qdmi – quantum device management interface: Hardware-software interface for the munich quantum software stack," in *2024 IEEE International Conference on Quantum Computing and Engineering (QCE)*, Montreal, Canada, September 2024.

[7] The CUDA-Q development team, "CUDA-Q." [Online]. Available: https://github.com/NVIDIA/cuda-quantum/blob/main/include/cudaq/Optimizer/Dialect/Quake/QuakeOps.td

[8] E. Kaya, B. Mete, L. Schulz, M. N. Farooqi, J. Echavarria, and M. Schulz, "Qpi: A programming interface for quantum computers," 2024, accepted for publication by The Third International Workshop on Integrating High-Performance and Quantum Computing at 2024 IEEE International Conference on Quantum Computing and Engineering (QCE).

[9] NVIDIA, *CUDA-Q*, 2024, also see https://nvidia.github.io/cuda-quantum/latest/index.html. [Online]. Available: https://developer.nvidia.com/cuda-q#

[10] MQSS, "MQPProvider." [Online]. Available: https://pypi.org/project/mqp-qiskit-provider/

[11] A. Swierkowska, J. Echavarria, L. Schulz, and M. Schulz, "Achieving pareto-optimality in quantum circuit compilation via a multi-objective heuristic optimization approach," 2024, accepted for publication by The Third International Workshop on Integrating High-Performance and Quantum Computing at 2024 IEEE International Conference on Quantum Computing and Engineering (QCE).

[12] A. Elsharkawy, X. M. To, P. Seitz, Y. Chen, Y. Stade, M. Geiger, Q. Huang, X. Guo, M. A. Ansari, C. B. Mendl, D. Kranzlmüller, and M. Schulz, "Integration of Quantum Accelerators with High Performance Computing - A Review of Quantum Programming Tools," *CoRR*, vol. abs/2309.06167, 2023. [Online]. Available: https://doi.org/10.48550/arXiv.2309.06167

[13] A. JavadiAbhari, S. Patil, D. Kudrow, J. Heckey, A. Lvov, F. T. Chong, and M. Martonosi, "Scaffcc: Scalable compilation and analysis of quantum programs," *Parallel Computing*, vol. 45, pp. 2–17, 2015.

[14] A. J. Abhari, A. Faruque, M. J. Dousti, L. Svec, O. Catu, A. Chakrabati, C.-F. Chiang, S. Vanderwilt, J. Black, F. Chong *et al.*, "Scaffold: Quantum programming language," *Princeton Univ NJ Dept of Computer Science*, 2012.

[15] G. Aleksandrowicz, T. Alexander, P. Barkoutsos, L. Bello, Y. Ben-Haim, D. Bucher, F. J. Cabrera-Hernández, J. Carballo-Franquis, A. Chen, C.-F. Chen *et al.*, "Qiskit: An open-source framework for quantum computing," *Accessed on: Mar*, vol. 16, p. 61, 2019.

[16] M. Amy and V. Gheorghiu, "staq – A full-stack quantum processing toolkit," *Quantum Science and Technology*, vol. 5, no. 3, p. 034016, Jun. 2020. [Online]. Available: http://dx.doi.org/10.1088/2058-9565/ab9359