Technische Universität München
TUM School of Computation, Information and Technology

Hybrid Runtime Environments for Quantum Computers and Simulators

Philipp Seitz

Vollständiger Abdruck der von der TUM School of Computation, Information and Technology der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitz: Prof. Dr. Martin Schulz

Prüfer der Dissertation:
1. Prof. Dr. Christian Mendl
2. Prof. Sven Karlsson, Ph.D.

Die Dissertation wurde am 24.01.2025 bei der Technischen Universität München eingereicht und durch die TUM School of Computation, Information and Technology am 16.06.2025 angenommen.

Abstract

Quantum computing is a rapidly evolving field offering theoretical advantages over classical computing. However, tapping into these advantages requires overcoming significant technical challenges. Available quantum computers are currently noisy, restricted in size, and susceptible to their environment. For this reason, the current quantum computing era is often called the noisy intermediate-scale quantum (NISQ) era. Nevertheless, hardware is improving rapidly, and the field is moving towards fault-tolerant quantum computing. The availability of quantum computers is steadily increasing, and the scale of quantum experiments is growing, as showcased by recent experiments by Google and IBM. High-performance computing centers are primary candidates for integrating quantum computers into existing high-performance computing environments. The software landscape for quantum computing targets operation through the cloud; a comprehensive software stack for integrating quantum devices into high-performance computing environments is still lacking.

In this work, I present my research on hybrid runtime environments for quantum computers and simulators. It targets the shortcomings of current quantum software by using established software engineering principles and applying them to quantum computing. The goal is to provide a hybrid software environment with two main characteristics: First, tight integration of components to enable efficient execution. Second, loose coupling of components allows for flexibility and extensibility, keeping up with the rapidly changing infrastructure.

I propose the *Unified Toolchain* for quantum computing to achieve this goal. In it, I present a modular software architecture suitable for high-performance environments. It targets arbitrary quantum resources that can be *heterogeneous*. Allowing heterogeneous hardware is a novel approach in quantum computing and is necessary to enable smooth integration. Further, my contribution contains proof-of-concept implementation components in the *Unified Toolchain*. I present a tensor network-based, high-performance *TTN simulator* for validation and simulation. I provide `Flow Synth`, a compilation technique for quantum circuits targeting multiple backends simultaneously. `MILQ` is an HPC-compatible scheduler utilizing quantum-specific techniques for efficient runtime scheduling. Finally, I discuss the necessary steps to transform the *Unified Toolchain* for fault-tolerant quantum computing.

Zusammenfassung

Quantum Computing ist ein sich schnell entwickelndes Forschungsfeld, das theoretische Vorteile gegenüber herkömmlichen Methoden bietet. Die Nutzung dieser Vorteile erfordert jedoch das Lösen erheblicher technischer Herausforderungen. Die derzeit verfügbaren Quantencomputer sind allerdings fehlerhaft, in ihrer Größe begrenzt und anfällig für kleinste Umwelteinflüsse. Aus diesem Grund wird die aktuelle Ära oft als ein temporärer, fehlerhafter Übergangszeitraum bezeichnet. Die Hardware entwickelt sich jedoch rasant weiter, und das Feld bewegt sich in Richtung fehlertoleranter Quantencomputer. Die Verfügbarkeit von Quantencomputern nimmt stetig zu, und das Ausmaß einzelner Quantenexperimente wächst, wie die jüngsten Experimente von IBM und Google zeigen. Hochleistungsrechenzentren sind die Hauptkandidaten für die Integration von Quantencomputern in bestehende Hochleistungsrechnungsumgebungen. Die Softwarelandschaft im Quantum Computing ist auf Cloud-Betrieb ausgerichtet und daher nicht für die Integration in solche Umgebungen ausreichend.

In dieser Arbeit präsentiere ich meine Forschung zu hybriden Laufzeitumgebungen für Quantencomputer und Simulatoren. Ziel ist es, die Schwächen der aktuellen Quantensoftware durch die Anwendung bewährter Prinzipien der Softwareentwicklung zu adressieren. Dafür stelle ich eine hybride Softwareumgebung bereit. Diese hat zwei Hauptmerkmale: Erstens eine enge Integration der Komponenten, um eine effiziente Umgebung zu ermöglichen. Zweitens eine lose Kopplung der Komponenten, die Flexibilität und Erweiterbarkeit gewährleistet, um mit den schnellen Änderungen der Infrastruktur Schritt zu halten.

Ich schlage die sogenannte *Unified Toolchain* (eine vereinheitlichte Werkzeugkette) vor, um dieses Ziel zu erreichen. Darin präsentiere ich eine modulare Softwarearchitektur, die sich für Hochleistungsumgebungen eignet. Sie zielt auf beliebige Quantenressourcen ab, die *heterogen* sein können. Das Einbeziehen heterogener Hardware ist ein neuartiger Ansatz im Quantum Computing und notwendig, um eine nahtlose Integration zu ermöglichen. Darüber hinaus umfasst mein Beitrag eine prototypische Implementierung von Komponenten in der Werkzeugkette. Ich präsentiere einen tensorbasierten, leistungsstarken *TTN-Simulator* zur Validierung und Simulation. Ich stelle `Flow Synth` vor, eine Kompilationstechnik für Quantenschaltkreise, die mehrere Quantencomputer gleichzeitig adressiert. `MILQ` ist ein HPC-kompatibler Scheduler, der quantenspezifische Techniken für eine effiziente Laufzeitplanung nutzt. Abschließend diskutiere ich die notwendigen Schritte, um die Werkzeugkette für fehlertolerante Quanteninformatik weiterzuentwickeln.

Acknowledgements

I want to express my sincere gratitude to everyone who has supported me throughout my dissertation journey.

Words cannot express my gratitude to my supervisor, Professor Christian Mendl, for his invaluable guidance, feedback, and patience throughout this journey. You have been instrumental in shaping both this dissertation and my personal growth.

I am equally grateful to Professor Sven Karlsson for his insightful feedback and constructive criticism, and to Professor Martin Schulz, for chairing the committee and mentoring me throughout my doctoral studies.

To my colleagues and friends: thank you for tolerating my quirks and antics with such good humor. Your camaraderie and shared struggles made this experience worthwhile. Special thanks go to Janette and Sandra, who tirelessly assist in administrative tasks.

Finally, I sincerely thank my parents and siblings for their unwavering support, love, and encouragement. You have been my anchor through every high and low, and I wouldn't have made it here without you.

Thank you all for being there for me.

Contents

1 Introduction

The idea of quantum computing (QC) was established in the early 1980s by Richard Feynman [1] and Yuri Manin [2]. At its core, one studies a quantum mechanical system of interest by a proxy system that one can control to mimic the original system. Technological advancement soon made it possible to build such systems, although they are still heavily restricted in size and complexity. Other physical restrictions, such as the need for low temperatures, short live times, or limited connectivity, make large-scale simulation prohibitively expensive. Due to the nature of quantum mechanics, these systems are intricate and require specialized equipment. Nonetheless, the first quantum computers were successfully formalized and built in the early 1990s [3, 4]. Outside of the physics domain, the discovery of Shor's algorithm [5] in 1997 pushed the field into the focus of computer science. Due to its application in cryptography, the algorithm showed that quantum computers theoretically can solve problems previously thought to be intractable. Based on the increased interest, the development of quantum computers has accelerated.

An immediate realization was the difference between quantum computers' theoretical and practical capabilities. Quantum computers are inherently noisy and thus prone to errors to the extent that realistic applications that grant a quantum advantage are not yet feasible. To distinguish between the two, the term NISQ was coined [6], in contrast to the more capable, *fault-tolerant* quantum devices. The newest hardware far surpasses the original definition, but the reduction of error rates has not decreased equally, and fault tolerance is not yet realizable on a practical scale. A significant milestone toward utility in realistic applications is the progression out of the physics lab into computing centers. This move allows interdisciplinary research and utilization of the hardware by a broader audience, eventually leading to the development of practical applications.

In the long term, quantum computers are expected to operate next to and in conjunction with classical computers. This goal contrasts the current mode of operation, where quantum computers operate as standalone devices. In a typical scenario, a quantum program is prepared manually and executed as an isolated, one-off experiment. Results are then analyzed separately from the execution. Computing centers are well-suited to operate quantum computers, given their experience with specialized hardware, but operate based on a different paradigm. For example, computing centers provide certain performance guarantees typically unavailable for quantum computers. Furthermore, due to the sheer complexity of quantum hardware, existing approaches are insufficient to accommodate quantum computers. The integration of hardware and software is a key challenge to overcome, and the result, a successful combination of QC and high performance computing (HPC), is summarized by the term high performance computing

- quantum computing (HPCQC). This thesis aims to contribute to the development of HPCQC and prepare for the switch to the fault-tolerant era.

Typically, large-scale integration is a long-term effort and depends on incremental improvements. In the HPCQC context, this has the benefit that software development can match the hardware and grow in complexity over time. But, this also mandates a scalable architecture based on modularity and sufficient abstraction. This work focuses on the software side of HPCQC, providing components that facilitate using quantum computers in a computing center. As typical in software development, the components are similar to existing building blocks from the HPC domain, geared toward long-term integration. The main goal of this work is to extend a key component in HPC, the runtime system for quantum programs. Incredibly challenging is the hybrid nature of HPCQC, which combines classical computation, classical simulation of quantum computation, and quantum computation.

This thesis also reflects this complexity in its scope and the accompanying software project I developed. It fits seamlessly into the existing HPC software stack, implementing familiar services. In the best case, the user interaction should not differ between classical and quantum programs. One critical aspect is the aforementioned shift of paradigm from a bespoke, one-off experiment to a long-running and robust service. In connection with this, various tools are necessary to cover potential use cases. These tools also need to advance simultaneously with the hardware, which creates a complex interplay with the runtime.

With this in mind, the thesis provides a comprehensive overview of the planned software stack. I derive this from state-of-the-art implementations, addressing some of their shortcomings. In particular, I analyze the interaction between individual components and the environment. The analysis is structured top-down, focusing on *compilation* and *scheduling*. To complement this analysis, I cover simulation and provide an outlook on error correction.

The components I provide reflect the main contributions I have developed from this work. I improve existing tensor network simulation techniques and explore their application in the context of HPCQC. Furthermore, I analyze specific programming models' viability in conjunction with their compilation approaches in this domain. I adapt existing compilation methods to HPCQC and provide general and hardware-specific optimizations. Finally, I provide a scheduling algorithm capable of handling the complex nature of quantum hardware.

1.1 Rationale

Currently, quantum computing is subject to rapid development. The hardware is reaching complexity levels beyond the scope of lab experiments. With the integration into the cloud, hardware is now available to a broader audience. However, in the long term, accessing quantum computers through the cloud inhibits their full potential, which only tight integration can realize. The cloud is not suitable for handling the complexity of

quantum hardware; this is mainly due to the available software stack. The next step is to build larger systems that combine the capabilities of classical and quantum computers.

Larger systems come with an increase in complexity. Besides the challenges of physical integration, software is still a large unknown. Driven by the need to accommodate new systems, the established software stack poses new challenges. The software stack needs to incorporate new techniques to leverage the full potential of quantum computers Hybrid algorithms, making up most quantum programs currently, require complex interaction schemes.

The interaction between classical and quantum components crucially happens during the execution of a hybrid algorithm. Runtime environments, systems that manage the execution of quantum programs, are the key to this interaction. They need to manage a mix of hybrid resources: classical, simulated quantum, and actual quantum. Complexity arises from multiple sources, such as communication, synchronization, and resource scheduling. But even the efficient simulation of quantum programs is a challenge. Finally, combining multiple systems into a functional unit is a desired final goal.

1.2 Problem Statement and Contribution

In its current state, QC is still a technology driven by early adopters. Due to the variety in hardware, software, and application areas, advances are made in different directions rapidly. A consolidation toward a standard way of operating quantum hardware and integrating it into existing systems is not yet foreseeable. The HPC domain, on the other hand, is built on existing standards, which are widely adopted and matured throughout decades of use. This conflict arises from the underlying problem of HPCQC integration. Solving this problem in its entirety is impossible. Instead, I focus on scaling systems from single standalone quantum hardware to heterogeneous systems. The central gap for such systems is in the runtime software components, which use advantages and increase performance.

This thesis aims to bridge the gap between quantum computing and high-performance computing. My work focuses on the software-side integration of multiple heterogeneous quantum systems. I will describe an abstract software stack that summarizes the common goal of HPCQC integration. Within this stack, I provide multiple components that facilitate quantum acceleration. The core novelty is investigating systems with more than one quantum backend. These backends work based on different principles and require adaptive runtime management. Each component fits seamlessly into the software stack, enabling tight integration while maintaining loose coupling.

First, I provide a comprehensive overview of the software stack. The key principle of this is the interaction and interfaces of components. Handling metadata and communicating intent is valuable for managing the system's complexity.

Then, I discuss tensor network methods. Tensor networks are a powerful tool to simulate quantum systems. Using simulators enables the study of quantum accelerators without access to hardware. Once the hardware is available, simulators can work with or as a validation tool for algorithms and software. The flexibility and simplicity of tensor

networks make them a valuable tool in the HPCQC context, significantly as they benefit from HPC systems.

Next, I focus on compilation, both online and offline. Synthesis is one of the first steps in offline compilation. I extend existing techniques intending to address multiple backends simultaneously. Thereby, I improve efficiency and flexibility. During the circuit generation phase, the interplay of optimization and runtime is crucial. Circuit cutting is an essential option for multi-backend systems.

I provide a scheduling algorithm capable of handling the characteristics of quantum computing. The paradigm shift from classical to quantum computing requires new approaches. Considering their advantages, a hybrid runtime environment can address simulators and real hardware. I formalize the scheduling problem and propose a suitable solution.

Finally, I provide an outlook on error correction. Error correction has tremendous implications for the software stack. I investigate how my work can evolve to accommodate error correction. Additionally, I contribute to design automation for topological error correction. With this, I enable the synergistic co-development of system and application software.

2 Background

On first observation, HPCQC contains two disparate paradigms: HPC and QC. In this section, *I* will first introduce the necessary concepts of HPC and QC separately. This introduction will briefly summarize the evolution of both fields, covering the most essential concepts. Due to the nature of HPCQC, it combines the two fields, where each field already has domain experts. One challenge is to find a common understanding that is mandatory for integration. *I* will therefore introduce the concept of HPCQC under the guise of *hybrid computing*.

2.1 HPC

HPC encompasses a wide range of topics. At its core, HPC combines specialized hardware, typically supercomputers, with the challenge of programming these massively parallel systems. The surrounding software and hardware infrastructure are also components of HPC. This software stack is then available to users, typically researchers from various fields. The problems of interest are typically too large to be solved on a single machine, which creates the need for advanced computation models and hardware. Hence, users resort to computation clusters, which offer a variety of operation modes, which I will discuss in the following section.

A critical aspect of HPC is the system's performance. Various metrics measure a system's performance, typically combined and weighed. The driving factor from the user's perspective is the time to solution, which is the time it takes to solve a problem. This importance is tied to the system's cost, as users pay for their use time. Maintaining a dedicated system is not feasible for a single user, which is why computing centers exist. Computing centers provide a shared infrastructure, which multiple users use.

2.1.1 Infrastructure

The infrastructure of a computing center differs from that of regular computers. Figure 2.1 shows a high-level overview of the infrastructure inside a computing center. The nodes fall into three broader categories: management, compute, and storage nodes. Management nodes orchestrate the operation inside the computing center. They provide the main point of interaction for users and administrators. The login nodes host the user environment; other nodes provide the runtime environment. Two essential components are the scheduler, responsible for distributing the workload, and the resource monitor, responsible for monitoring the system's health.

Actual computation occurs on interconnected compute and storage nodes inside a protected environment. All compute nodes fulfill computational tasks but differ based

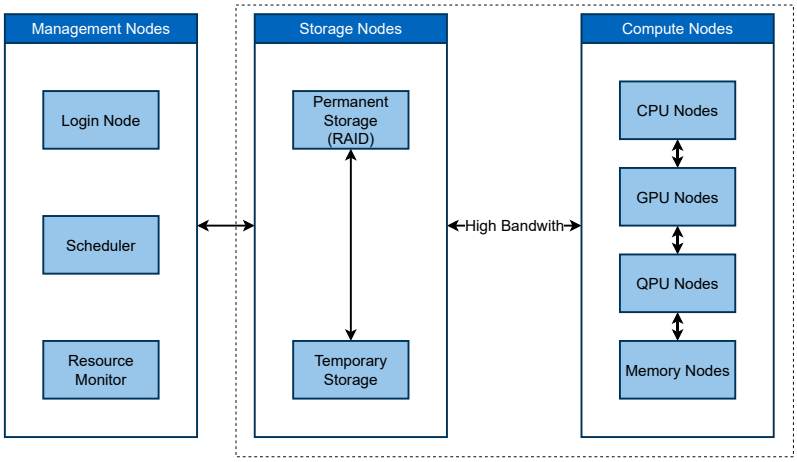


Figure 2.1: High-level overview of the infrastructure inside a computing center. It consists of three different nodes: management, compute, and storage nodes.

on their hardware. Each node has a dedicated central processing unit (CPU) and some hardware accelerators, for example, graphics processing units (GPUs) or tensor processing units (TPUs). General-purpose compute nodes (denoted CPU Nodes) are reasonable for all tasks, while specialized nodes can accelerate specific tasks. Typical examples are GPU and TPU nodes, suitable for tasks that offer parallelization potential, typically tensor operations. Memory nodes are necessary for tasks that require a large amount of local memory. In the future, quantum processing unit (QPU) nodes will similarly solve quantum tasks.

Big computations exceed the limits of a single node, so they are distributed across multiple nodes. This distribution creates new challenges like communication, synchronization, and load balancing. On a physical level, this requires a high-bandwidth, low-latency network. The topological layout of the network is crucial for the system's performance and has implications for workload distribution. To keep up with the hardware, programming models have emerged that allow users to write highly efficient programs on any hardware. These models include SIMD for vectorized operations, MUMA for memory access, and MPI for distributed memory parallelization.

Various software components are used to manage the system's complexity, referred to as software stack. A typical software stack includes a package manager, compiler, and a scheduler on the system side and HPC libraries on the user side. Even the choice of programming language is essential, as it has implications for the system's performance. C++ is the most common language, but Fortran and Python are also relevant. A standard system stack includes `SPACK` for package management, `LLVM` for compilation, and `SLURM` for scheduling. Users typically use MPI for distributed memory parallelization and OpenMP for shared memory parallelization. For specialized hardware, `CUDA` and `OpenCL` are used for programming GPUs and TPUs. In Section 3.1, I will discuss the implications of these choices for quantum programming languages.

Metric	Description
Flops	The number of floating-point operations per second.
Memory Bandwidth	The amount of data that can be transferred from memory to the processor.
Network Bandwidth	The amount of data that can be transferred between nodes.
Time to Solution	The time it takes to solve a problem.
Quality of Solution	The quality of the solution. Commonly used for approximation algorithms.

Table 2.1: Quantities of interest in HPC. The first three are used to measure the performance of the system, the last two are used to measure the performance of the program.

2.1.2 Workflow

In HPC, the typical workflow of creating and running a program differs from standard practice. The most notable difference is the separation of the development and execution environment. Like regular programs, one solves an HPC problem on a small scale on a local system. Once the program works, the user transfers it to the computing center via a login node. The user then compiles the program for the target architecture and submits it to the scheduler. In the request, the user specifies the resources required for the program, such as the number and type of nodes and the expected runtime. The program is then scheduled and executed according to the parameters, and the system notifies the user once the program terminates.

As mentioned before, performance is a critical aspect of HPC, and the user is responsible for optimizing the program. This optimization includes the choice of algorithms, programming language, data structures, and parallelization implementation. Even the choice of the compiler and the compiler flags can significantly impact the program's performance. Additionally, the number of hyperparameters, such as the number of nodes, threads, and the memory layout, increases the complexity of the problem. A critical aspect of the runtime environment is to enable these optimizations, supporting a wide variety of users and hardware.

2.2 Quantum Computing

Quantum computing is a new paradigm of computation enabled by sophisticated hardware. Initially, Richard Feynman proposed the idea of a quantum computer, which is a computer that uses controllable quantum mechanics to simulate a quantum system under investigation. The utility of such a computer is that it simulates quantum systems more efficiently than classical computers. By using quantum mechanical principles, such as superposition and entanglement, quantum computers can solve problems from a different perspective. The advantage lies in the exponential growth of the combination of states, which is impossible with classical computation.

The term "computing" is understood as the controllable manipulation of quantum states, not in the sense of classical computation. As a result, the quantum circuit model, which defines the operations of quantum computation as a sequence of quantum gates, is the most common model of a quantum computer. This concept follows the classical circuit model, where computation breaks into logic gates, components available in any computer. While computationally superior, quantum computers are not a replacement for classical computers, as they are unsuitable for all tasks. It is more accurate to think of quantum computers as accelerators used for specific tasks. This caveat is due to the nature of quantum computation, which is probabilistic and error-prone.

The susceptibility to errors is a significant challenge in quantum computing. The current state-of-the-art provides an average error rate of 10^{-4} per quantum operation, compared to 10^{-16} for classical computers. The error rate must decrease further to leverage the power of quantum computers. Subsequently, this target has been termed the *fault-tolerance threshold*, which will enable fault-tolerant quantum computation, unlocking the full potential of quantum computers. NISQ describes the quantum computation paradigm before the fault-tolerance threshold is reached. Already, Kim et al. [7] show quantum utility for the NISQ era, and many concepts translate to the fault-tolerant era. In this work, I will mainly focus on the NISQ era, as this is the current state of quantum computing. In Chapter 8, I will discuss error correction and its implications for the previously discussed concepts.

For now, it is essential to understand the basic concepts of quantum computing. In the following, I will introduce the theoretical aspects of quantum computing, heavily inspired by the book of Nielsen and Chuang [8]. I aim for a general introduction and will provide more details for the respective sections in the following chapters.

2.2.1 Concepts of Quantum Computing

In this section, I will introduce the basic concepts of quantum computing. Readers familiar with quantum mechanics can safely skip this section. A more detailed introduction is available in the book of Nielsen and Chuang [8]. After the introduction, I will also look at different hardware platforms and sketch their integration into HPC. The goal is to base the analysis on a common understanding, which has implications for the choices made in the following chapters.rs.

2.2.1.1 Quantum Mechanics

Quantum mechanics build on linear algebra, specifically vector spaces. The so-called *Hilbert space* is a finite-dimensional complex vector space, in combination with an inner product operator. *Inner products* are typically denoted for vectors $|u\rangle, |v\rangle \in \mathbb{C}^n$ as

$$\langle u|v\rangle = \sum_i u_i^* v_i \,. \tag{2.1}$$

The Hilbert space is the space is spanned by n basis states $\{|i\rangle\}$, which are orthonormal, i.e., $\langle i|j\rangle = \delta_{ij}$. A second operation, the *tensor product* $|u\rangle \otimes |v\rangle = |u\rangle |v\rangle = |u, v\rangle$, is a

generalization of the outer product. It describes the state of a composite system with size $|U| \cdot |V|$, which is the root of the exponential growth. An ensemble of quantum states can also be described by the density operator (matrix)

$$\rho \equiv \sum_i p_i \left|\psi_i\right\rangle \left\langle\psi_i\right| , \tag{2.2}$$

It is beneficial for mixed states, which are a probabilistic mixture of pure states and the description of noise. For most of the dissertation, I will stick to the vector notation for simplicity, which intuitively fits with the subsequently defined circuit model.

Definition 1 (State Space, Quantum State) *Any quantum mechanical system can be described by a state vector $|\Psi\rangle$ within an associated Hilbert space $\mathcal{H}$.*

Definition 2 (Time Evolution) *The* Schrödinger equation *describes the time evolution of a quantum state $|\Psi(t)\rangle$ as $i\hbar\frac{d}{dt}|\Psi(t)\rangle = H|\Psi(t)\rangle$. Here H is the Hamiltonian operator, which describes the system's energy, and $\hbar$ is the reduced Planck constant.*

The Hamiltonian operator, a Hermitian matrix, describes the time evolution of a quantum state. It contains a complete description of the system energies and interactions. If one can solve the Schrödinger equation, one can predict the system's state at any time. Since this is computationally infeasible for most systems, the idea is to use a controllable quantum system to simulate the desired system. The result is the implementation of Feynman's idea of a quantum computer. Due to quantum mechanics, interaction with the system also affects the state of the system. Consequently, the observation of the system during its evolution is impossible. Instead, the manipulation of the system targets strategically to maximize the probability of measuring the desired outcome.

From their quantum nature, some properties do not exist in classical computers. Foremost, the superposition principle allows a quantum state to be in multiple states simultaneously, a linear combination of the basis states of $\mathcal{H}$. Secondly, quantum computing is non-deterministic due to the probabilistic nature of quantum mechanics. The randomness can be alleviated by repeating the computation and tracking the probability of the outcome. A direct consequence of this is the concept of entanglement, a property of correlated quantum systems. A tensor product of individual states can not express entangled states. Finally, the no-cloning theorem also restricts quantum systems, which states that copying an arbitrary quantum state is impossible.

2.2.1.2 Quantum Circuit Model

Quantum mechanics admit multiple equivalent representations of the evolution of a quantum system. The most common representation is the quantum circuit model, a sequence of quantum gates operating on a quantum state. A vector in a Hilbert space represents the quantum state, and the gates are unitary matrices operating on this vector. Figure 2.2 gives an example circuit.

The base unit of a quantum circuit is the qubit, which is a two-level quantum system, each represented by a *wire* of the circuit. By convention, the state of a qubit is a

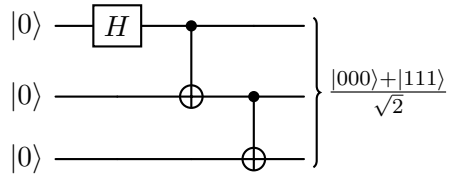


Figure 2.2: Example of a quantum circuit. The circuit prepares a three-qubit GHZ state.

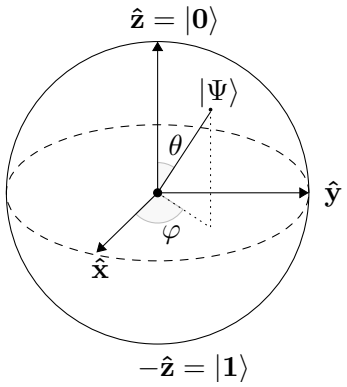


Figure 2.3: The Bloch sphere representation of a qubit state. The state $|\Psi\rangle$ is described by the angles θ and φ.

superposition of the basis states $|0\rangle = \begin{pmatrix}1\\0\end{pmatrix}$ and $|1\rangle = \begin{pmatrix}0\\1\end{pmatrix}$, which are the eigenstates of the Pauli-Z operator.

Definition 3 (Qubit) *A Qubit is a two-level quantum system, which is described by a state vector* $|\Psi\rangle = \alpha\,|0\rangle + \beta\,|1\rangle$ *with the normalization condition* $|\alpha|^2 + |\beta|^2 = 1$.

The state of a qubit can be visualized as a point on the Bloch sphere (c.f. Figure 2.3), a unit sphere in three-dimensional space.

One could use any Pauli operators Table 2.2 to define the basis states, but the Pauli-Z operator is the most common choice. A state of multiple qubits is a tensor product of the individual qubit states

$$|\Psi\rangle = |\psi_1\rangle \otimes |\psi_2\rangle \otimes \ldots \otimes |\psi_n\rangle \ . \tag{2.3}$$

From this, the state of a quantum system with n qubits is a vector in a 2^n-dimensional Hilbert space. Considering the higher dimensional extension of qubits, called qudits, the dimension of the Hilbert space is d^n, which grants only a fraction of additional computational power by adding high complexity.

A unitary operation describes the evolution of the system

$$|\Psi'\rangle = U\,|\Psi\rangle \tag{2.4}$$

Table 2.2: Basis states of the Pauli operators. The states are the +1 and −1 eigenstate of the respective operator.

Pauli Operator	**+1 Eigenstate**	**−1 Eigenstate**
$X = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$	$\|+\rangle = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 \\ 1 \end{pmatrix}$	$\|-\rangle = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 \\ -1 \end{pmatrix}$
$Y = \begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix}$	$\|\rightarrow\rangle = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 \\ i \end{pmatrix}$	$\|\leftarrow\rangle = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 \\ -i \end{pmatrix}$
$Z = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}$	$\|0\rangle = \begin{pmatrix} 1 \\ 0 \end{pmatrix}$	$\|1\rangle = \begin{pmatrix} 0 \\ 1 \end{pmatrix}$

which can be implemented by a quantum gate. In the most general form, the gate implements the system Hamiltonian $U = e^{-iHt}$ for a given time t. Hence, the computation is simply a matrix-vector multiplication but classically infeasible due to the exponential size of the matrix. This operation can be directly implemented in analog quantum computers, but in gate-based quantum computers, a sequence of quantum gates approximates the operation. Each gate is a unitary matrix that operates on a subset of qubits. As a unitary matrix, it is reversible, a fundamental property of quantum computation. One needs a computationally complete set of basic gates to closely approximate the desired operation. The most common set is the *Clifford+T* set, which consists of the Pauli operators $\{X, Y, Z, I\}$, the Hadamard gate (H), the T gate, and the CNOT (CX) gate. The CX gate takes on a unique role, as it conditionally flips a target qubit if the defined control qubit is in the $|1\rangle$ state. Using a qubit in the $|+\rangle$ state as a control qubit transforms the target qubit into a superposition, conditional on the control. The two qubits are then in a shared superposition, the basis of entanglement. Figure 2.2 is an example of a quantum circuit that entangles three qubits, which all are simultaneously either in the $|0\rangle$ or $|1\rangle$ state.

Evaluating the quantum system requires interaction, where one performs a measurement, which collapses the state into one of the basis states. Due to the probabilistic nature of quantum mechanics, the outcome of the measurement is not deterministic. The probability of measuring a state follows from the Born rule, which is the square of the amplitude of the state, e.g., $P(|\Psi\rangle = |0\rangle) = |\alpha|^2$ in Definition 3. For m potential outcomes, there exist a set $\{M_m\}$ measurement operators. The probability of measuring a state obeys the following $P(m) = \langle\Psi| M_m^\dagger M_m |\Psi\rangle$. The collapsed stated $\frac{M_m|\Psi\rangle}{\sqrt{\langle\Psi|M_m^\dagger M_m|\Psi\rangle}}$ remains after. For most purposes, it suffices to measure the state in the computational basis $M_0 = |0\rangle\langle 0|$ and $M_1 = |1\rangle\langle 1|$. Therefore, quantum computation can be interpreted as a strategical manipulation of quantum states to maximize the probability of measuring the desired outcome.

2.2.1.3 Noise and Error Correction

In isolation, quantum systems act according to their probability distribution. Unfortunately, perfect isolation is impossible, and the system interacts with its environment.

Additionally, not all operations function flawlessly. In summary, this leads to multiple noise sources, which affects the quantum computation. When formalizing this, the noise according to a quantum operation $\mathcal{E}(\rho)$, which acts on the quantum state expressed by the density operator ρ. This formulation is equivalent to the previously introduced state space representation but more suitable for mixed states. A classical analogy is the communication under a noise channel, where there is a probability of flipping a bit. For qubits, the Pauli operators sufficiently describe this noise in its most general form. In particular, it suffices to consider the bit-flip (X) and phase-flip (Z) operators and combinations thereof, which follow from

$$E = aI + bX + cY + dZ, \quad \text{and} Y = XZ\,. \tag{2.5}$$

Sources of noise stem from operations, such as gate errors, readout errors, and crosstalk. But, there are further causes due to the delicate nature of quantum systems. Errors on qubits result from a physical process called *decoherence*. Over time, the quantum state can lose its ability to maintain superposition, or the entanglement can be lost. Depending on the hardware, it is also possible for *leakage*, where the qubit is not entirely in a desired state. Also, quantum systems are subject to outside effects, such as cosmic rays interfering with computation. To describe the amount of noise a system is under, one can use the average *fidelity* $\mathcal{F}(\mathcal{E})$ of a (trace-preserving) operation $\mathcal{E}$[9]. Typically, instead of a general fidelity, gate fidelity is used, which describes the average fidelity of a unitary operation U as

$$\mathcal{F}(\mathcal{E}, U) = \int d\psi \, \langle\psi| \, U^\dagger \mathcal{E}(|\psi\rangle \, \langle\psi|) U \, |\psi\rangle \ . \tag{2.6}$$

Two approaches are common to prevent errors from influencing computation results Firstly, *error mitigation* aims to reduce the impact of errors on the computation. It uses techniques to make the computation more resilient, such as predicting and canceling potential errors or extrapolating to the zero noise limit [10]. An interesting approach is to execute part of the computation on perfect simulators, especially if these parts can still be simulated efficiently [11, 12]. Error mitigation is intended as a short-term solution, as it does not address the root cause of the errors. *Error correction* actively corrects errors during the computation or tracks them and applies corrections at the end. One introduces redundancy by encoding the quantum state in a larger Hilbert space, achieving error correction. There are multiple error correction codes with hardware-specific advantages and disadvantages. One can even customize them to fit the noise if known beforehand. Error correction works due to three mechanisms:

- **Pauli twirling:** Most errors can be transformed to Pauli errors [13,14]. Therefore, only a discrete number of errors has to be accounted for.
- **Threshold Theorem:** Given a specific error rate, a threshold exists below which error correction is possible [15].
- **Magic State Distillation:** Universal quantum computation can be achieved in error correction codes by injecting magic states [16].

Fault tolerance is the ultimate goal of error correction, but no hardware implementation has reached it yet. I discuss the implications of error correction for the software stack in Chapter 8.

2.2.1.4 Quantum Algorithms

With a 53-qubit superconducting quantum computer, Google claimed to show quantum supremacy[1] [17], which solves a problem on quantum hardware that is infeasible on classical hardware. While this claim was refuted after [18,19] soon, quantum supremacy remains sought after. The chosen algorithm, cross-entropy benchmarking, is known for its simplicity on quantum hardware compared to classical hardware, with no real application value. The science community has now shifted away from the term quantum supremacy; instead, it focuses on quantum advantage, tasks that classical hardware can not perform, and quantum utility, where quantum hardware benefits classical problems.

In the realm of quantum advantage, a few well-known algorithms are of interest. First and foremost, Shor's algorithm solves the integer factorization problem in polynomial time. The algorithm is relevant for cryptography, breaking the RSA encryption scheme. It relies on the quantum Fourier transform a quantum version of the discrete Fourier transform. Quantum signal processing [20] is a standard description unifying most quantum algorithms. The accuracy required by these algorithms is higher than the current hardware can provide, so I will not discuss them further.

Instead, I will focus on variational algorithms, which also can provide quantum utility. Close interaction loops between classical and quantum computations characterize them. The typical example is the variational quantum eigensolver (VQE) (c.f. [21] for an extensive review), which allows finding the ground state of a quantum system. In the VQE, a quantum computer prepares a parametrized state, which is measured and the results submitted to a classical computer. The classical computer then optimizes the quantum state's parameters and repeats the procedure, as pictured in Figure 2.4. In quantum chemistry, conceptually similar algorithms are used to find the ground state of a molecule, for example, the Coupled Cluster [22] method. They are of special interest for two reasons. First, they require high-performance systems on the classical side, compared to the simple optimization problems of the VQE. This demand means the quantum hardware must tightly interact with the classical system to ensure the overall performance. Second, the required quantum circuits are not known beforehand. This uncertainty means that the quantum circuits must be generated and optimized on the fly, posing challenges to the software stack. Compared to current algorithms, where the compiler fully optimizes circuits before execution, the compilation has to happen in real time. Having the entire computation halt for the compilation is infeasible, and the quality of the results still has to be ensured. These two aspects, *tight integration* and *real-time compilation*, are the main focus of my work.

[1] The term *supremacy* is considered problematic and has been replaced by *advantage*.

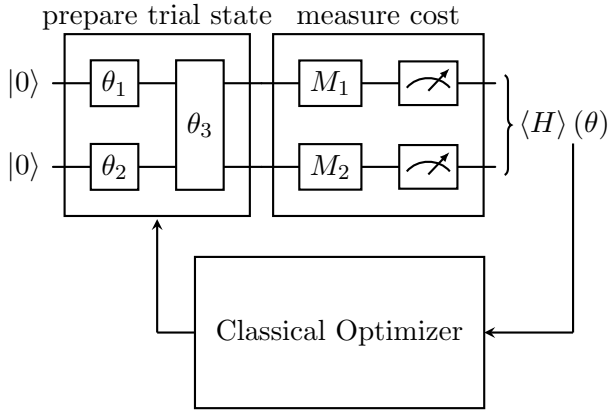


Figure 2.4: Schematic of the VQE algorithm. A ground state is prepared by a parametrized quantum circuit, which is then measured. The classical computer optimizes the parameters of the quantum circuit. The process is repeated until a convergence criterion is met.

2.2.2 Hardware

Constructing a quantum computer is a delicate matter. On one hand, it should have all the favorable properties, but it must be controllable to get the desired outcomes. Additionally, the susceptibility to errors is a significant challenge. As a result, most quantum computers can only be realized under extreme conditions, for example, at temperatures close to absolute zero. DiVincenzo's criteria [4] are a set of requirements that comprise a quantum computer that can exhibit quantum advantage.

Definition 4 (DiVincenzo's Criteria) *The criteria are:*

1. *Well-characterized qubits*
2. *Reliable state initialization of the qubits*
3. *Long coherence times, concerning the gate operation time*
4. *A universal set of quantum gates*
5. *Measurement of individual qubits*

6* *Conversion between communication and computation qubits*

7* *Transmission of quantum information between multiple sites*

Criterion 6 and 7 are not necessary generally but are mandatory for quantum communication. Besides these theoretical requirements, there are also practical requirements. The number of qubits is the defining power for the computation. Considering the noise and the resulting overhead of error correction, the number of qubits is critical. Therefore, scaling up the system is an additional challenge. Currently, several technologies fulfill

Table 2.3: Orders of magnitude of operation times in 1 ns. Superconducting qubits (SC), neutral atoms (Atoms), and trapped ions (Ions) are compared. Generally, superconducting qubits are faster than the other two technologies but have a lower coherence time. Data taken from [29–31].

Operation	SC	Atoms	Ions
1-Qubit Gate	5	$6 \cdot 10^4$	$2 \cdot 10^4$
2-Qubit Gate	$2 \cdot 10^1$	$3 \cdot 10^3$	$3 \cdot 10^3$
Measurement	$1 \cdot 10^1$	$1 \cdot 10^5$	$8 \cdot 10^4$
Preparation	$1 \cdot 10^2$	$1 \cdot 10^4$	$2 \cdot 10^2$
Coherence	$5 \cdot 10^5$	$1 \cdot 10^{10}$	$6 \cdot 10^{11}$
Setup	$>10^8$	$>10^8$	$>10^8$

at least five of the criteria. For example, superconducting qubits, trapped ions, neutral atoms, and photonics are among the most developed technologies [23–26]. Within each group, also called *modalities*, there are various design choices, for example, which ions to use. Still, hardware development is ongoing.

Each approach is unique in its own right and has its advantages and disadvantages; there is no superior choice yet. Hence, the software has to accommodate both, be abstract enough to work across most modalities, and utilize each to its full potential. The circuit model helps to achieve abstraction, and compilers optimize for hardware. The capabilities are comparable to different instruction sets for CPUs. Similarly, the hardware characteristics can entail hardware-specific differences for different problem scenarios. One of these is qubit *connectivity*, which describes between which qubits operations are possible. For example, superconducting qubits typically have grid-like, nearest-neighbor connectivity, a limitation for some algorithms. To overcome this limitation, one can build a so-called *move* operation [27], which might not be necessary for other platforms. Other platforms can physically rearrange qubits as part of a machine setup or even during computation (called *shuttling*)[28]. Developers (or suitable tooling) must consider this when designing algorithms, choosing hardware, and optimizing for it.

Standard metrics can also quantify the hardware. Most commonly, the number of qubits and gate-fidelities are meaningful metrics. Also, the shared properties describing the physical systems, such as coherence times, are reported. Compared to the duration of the quantum operations, they provide insight into the computational capabilities of the system. These can have differences in orders of magnitude, which I display in Table 2.3.

2.2.3 Integration

In its current state, quantum computing is still subject to regular innovation. For software researchers, access to quantum computing is mandatory. A standard mode of operation is to offer restricted access to quantum devices through the cloud. This model follows the example of IBM `Qiskit`, and established cloud providers advocate it. It allows hardware vendors to access their devices while still in the laboratories. From

	GPU	ASIC	QPU
Bandwidth	1.5 Tbit/s	1 Tbit/s	1 Gbit/s
Throughput 16-bit	195 Top/s	550 Top/s	10.5 kop/s
Clock speed	500 MHz	5 GHz	10 kHz

Table 2.4: Physical limitations of different hardware types. The values are estimates and can vary depending on the specific hardware. Table adapted from [34].

their perspective, this has the advantage of already capitalizing on the technology while familiarizing potential customers.

This model is a convenient way to access quantum computing from the user's perspective. However, due to the restrictions of the cloud, some of the hardware's capabilities might not be accessible. Additionally, using the cloud model results in long queueing times due to the scarcity of quantum devices. This time penalty is a significant limitation for some classes of algorithms as they require short interaction cycles between classical and quantum hardware. These hybrid algorithms are among the most promising applications for near-term quantum computing.

Scaling up the systems individually and integrating them into existing HPC systems remains a challenge. Tight integration typically enables high performance, which comes at the cost of more complexity. Furthermore, to avoid lock-in effects, tight coupling has to be avoided. One requirement is the on-premise installation of quantum hardware in computing centers. This scenario places the quantum hardware physically close to a host system, enabling near-time communication. Ultimately, having on-node accelerators would allow for real-time, highly efficient quantum-classical computations. Multiple integration scenarios are already being explored [32,33], but only on-premise installations are now feasible.

An interesting question is how to utilize the quantum hardware for HPC applications. Compared to classical operations, quantum operations are slower and more error-prone. This comparison becomes significantly more critical when considering the overhead of error correction. Assuming a logical operation time of 10 µs and 10000 error-corrected qubits, [34] concludes that the quantum hardware is only beneficial for large-scale problems requiring little input data. This conclusion follows from two facts: first, clock times are significantly slower, and second, the limits of converting classical data to quantum data. In Table 2.4, I have summarized the differences between classical and quantum hardware limitations. Accordingly, quantum hardware will only be beneficial if it reliably outperforms classical hardware in real applications.

2.3 Heterogeneous Computing

Heterogeneous computing combines different types of hardware to solve a problem. The trend started with introducing of GPUs as accelerators for HPC systems. They specialize in parallel matrix operations, common in many research subjects. With the slowing down

of Moore's law, the trend has continued with the introduction of other types of hardware. Instead of pushing general-purpose hardware to its limits, specialized hardware solves specific tasks. Potentially, QPUs can accelerate specific tasks. It is possible to imagine a system with different quantum *hierarchies* that uses long-lasting qubits for storage and fast qubits for computation. The knowledge gained about integrating GPU can be transferred to QPUs. This approach seems reasonable and is the basis of much of the research in the field, which I will discuss in Chapter 3.

The idea of heterogeneous computing is inherently hybrid. Multiple types of hardware communicate and coordinate to solve a problem. The software needs to be able to capitalize on the hardware to do so efficiently, If done correctly, it results in a system that is more than the sum of its parts. For quantum computing, this paints a clear path for integration into HPC systems. Depending on their level of integration, quantum resources can aid in the computation accordingly. Concepts from other hardware can be adapted and applied to quantum computing.

3 Related Work

Integration of quantum computing with HPC is slowly progressing. On the software side, the development has been theorized based on assumptions about the hardware. Following classical computer architectures, one of the earliest models for quantum computing was the quantum random access machine (QRAM) [35] model. Qubits remain in an addressable, memory-like register with arbitrary access. While highly desirable, current technology still cannot implement this model. Instead, there have been refinements to the model, such as the (refined) heterogeneous quantum-classical computation (HQCC) [36, 37] model. In these models, a quantum computer acts as a co-processor to a classical computer, which is still responsible for the control flow.

Analyzing these models, the role of the software stack becomes apparent. The quantum controller can be optimized toward quantum operations by relegating complex logic to the classical computer. In this chapter, I will analyze software designed for quantum computing. From an initial observation, it is clear that HPC was not the primary focus of these software stacks. In general, there are three categories of software tools:

- **Quantum Programming Languages** they focus on programming quantum algorithms. They introduce higher-level abstractions to simplify the development of quantum algorithms. Usually, a simplified hardware model is assumed.
- **Compilation Toolchains** focus on translating quantum algorithms to quantum hardware. This compilation is necessary due to the hardware and programming model discrepancy.
- **Runtime Environments** focus on executing quantum algorithms on quantum hardware. Commonly, they mix concepts from the previous two categories and add interfaces to the hardware.

I will analyze each category in the following sections based on selected examples. The goal is to identify gaps in the current software stack under the viewpoint of HPC. These will serve as guiding principles and are the foundations for the proposed *Unified Toolchain* (c.f. Chapter 4) for HPCQC.

3.1 Quantum Programming Languages

Quantum programming languages are motivated by the need to simplify the development of quantum algorithms. Manually calculating a quantum circuit is tedious and error-prone and usually does not fit the mental model of a computation. Hence, a natural step

is to introduce higher-level abstractions. The most general abstraction would be physics-based. In such an example, constructing the problem Hamiltonian of a system would be sufficient to describe the computation. Usually, this is not practical and unconventional for programmers; therefore, some concessions are necessary. As seen in Chapter 2, there are several equivalent formulations of quantum computing. At least one of these formulations is the foundation for a quantum programming language.

General computation is also possible with quantum programming languages. There are two common ways to achieve this. The first is to design a quantum programming language that includes classical logic. For example, this approach is taken by `Q#` [38].

```
1  namespace Bell {
2      open Microsoft.Quantum.Intrinsic;
3      open Microsoft.Quantum.Canon;
4
5      operation SetQubitState(desired : Result, target : Qubit) : Unit {
6          if desired != M(target) {
7              X(target);
8          }
9      }
10     ...
11     @EntryPoint()
12     operation TestBellState() : (Int, Int, Int, Int) {
13             SetQubitState(initial, q1);
14             SetQubitState(Zero, q2);
15
16             H(q1);
17             CNOT(q1, q2);
18     }
19 }
```

Listing 3.1: `Q#` example for setting a qubit state and using to create a bell state.

The operation `SetQubitState` is a quantum operation that one can call from a classical program. Quantum operations can interleave with classical logic and control flow.

The second approach is to design a quantum programming language embedded in a classical programming language. Built-in functions or an embedded domain-specific language can create quantum constructs. For example, `Quingo` [37] takes this approach. A result of the second approach is a more *kernelized* system. By this, I mean that quantum logic exists separate from classical logic. Like other accelerator models, the quantum program is a parametrized version of the quantum algorithm.

The kernelized approach has some implications for compilation, which I will discuss in the next section. Qubit allocation also requires special consideration. In the example below, the `using` keyword allocates qubits. Qubits are resources similar to memory and have to be reserved and released. This concept builds on the QRAM model, where the processor can directly access qubits as regular addresses. In practice, the connection is not always direct; a complex process occurs behind the scenes.

```
1  //kernel.qu
2  operation sum_random (arr: int[], r: bool ): (int, int) {
3      int sum = 0, i = 0, random = 0;
4      while(i < arr.length) {
5          sum += arr[i];
6          i += 1;
7      }
8      if(r){
9          using(q: qubit) {
10             init(q); H(q);
11             if(measure(q)) {random = arr[0];}
12             else {random = arr[1];}
13     }}
14     return (sum, random);
15 }
```

Listing 3.2: Quantum Summation with Randomness as a kernel in Quingo. The kernel can be invoked from a classical program. Control flow can act on quantum data and is executed on the control software.

A third alternative emphasizes the theoretical aspects of programming language design. Verification techniques based on semantical reasoning also extend to quantum programming languages (QPLs). Languages such as `QWIRE` [39] can ensure that a quantum program is correct by construction. For example, linear types can ensure that users cannot clone qubits cannot. Similarly, by defining special semantics, languages can enforce other quantum properties. In `QWIRE`, an arbitrary logic formula can be verifiably correct and implemented as a quantum circuit. With these powerful tools, higher-level abstractions, such as a *logic quantum oracle*, can be provided to the user. The reverse is also possible; given a QPL proven to faithfully represent quantum mechanics, reasoning about the program allows conclusions about the quantum system.

In practice, most of these languages have only a small user base. Therefore, the maintainability of these languages is a concern, and the reward for investing in these languages is unclear. The most popular interface to quantum computing is `Qiskit` [40], a Python library. It has more runtime features, which I will discuss later. From the perspective of HPC, user adoption is only one of the criteria. Additionally, performance and scalability are essential. The QPLs needs extensive tooling support to achieve this. Most of the QPLs are not designed with this in mind and are therefore categorically unsuitable for HPC. Other aspects of HPC are rarely covered by QPLs [41]. With the prospect of integration, the options are limited. The language is compatible with the existing infrastructure or can produce compatible output.

3.2 Compilation Toolchains

Compilation is the process of transforming a high-level program into a low-level program. The term is usually associated with code optimization and machine code translation. *Transpilation* is a unique form of compilation where the target language is similar to the

source language. It was made popular by the JavaScript community, where TypeScript is transpiled to JavaScript. Both terms are used interchangeably in quantum computing; no clear distinction exists. Additional confusion arises if the quantum program is defined using a classical compiled language such as C++. For the remainder of this work, I will use the term *compilation* to transform a quantum program to run it on a quantum computer. This definition entails complete optimization, which is especially necessary for NISQ hardware.

From the point of view of code optimization, classical and quantum optimization are similar. Some high-level techniques can even be applied to both, especially when code is mixed, for example, loop unrolling. In quantum computing, these optimizations are usually more complex and involve additional knowledge about the quantum program. Some general techniques still apply mainly to reduce the number of gates. For NISQ, the decoherence time is limited; hence, the depth of a circuit is crucial. Due to the complexity of the optimization steps, there are several established steps in the compilation process:

1. High-level optimization.
2. **Synthesis:** Decomposition of multi-qubit gates into single- and two-qubit gates.
3. **Mapping:** Assigning logic qubits in the algorithm to physical qubits of the hardware.
4. **Routing:** Ensuring gates are possible by placing them on physically interacting qubits. This step is only necessary for restricted topologies.
5. Translation to hardware-native gates.
6. Low-level optimization.
7. **Gate Scheduling:** Assigning timings for executing the gate.
8. Conversion to pulse sequences.

Error correction will eventually be part of the compilation process, but at which step is still an open question. These steps follow a stringent order to make the quantum program executable on the hardware while minimizing the resources used.

A common distinction in classical compilation is between ahead-of-time and just-in-time compilation. The main difference is when the final binary is written: offline during compile time or online during runtime. In ahead-of-time compilation, the compiler constructs the entire program into a binary executable before execution. This approach means all the hardware information must be available at compile time. Performance-wise, this is typically the fastest since the system does not spend time on optimization during runtime. Just-in-time compilation has more flexibility and allows for deeper optimizations due to information availability. This flexibility comes at the cost of runtime performance, which includes a final compilation step, where the compiler translates the bytecode to machine code.

Quantum programs are usually generated as part of a hybrid classical-quantum program. Offline compilation of the quantum part is generally not possible. The exception to this rule is if the circuit is wholly defined beforehand and the hardware fixed. A special case is parametrized circuits, where the circuit structure does not change. Parametrized gates, such as rotation gates, are only instantiated with different parameters. This trick is common in variational algorithms, where the parameters are optimized to minimize a cost function. For example, Younis et al. [42] have developed compilation techniques that can optimize despite missing the parameters. For HPC, this is especially relevant because this positively impacts performance. A preoptimized circuit artifact can be used instead of running compilation at every algorithm step. Furthermore, the communication overhead decreases, too, since only parameters must be transmitted. Even more significant performance gains are possible by end-to-end optimization. This concept optimizes the entire quantum circuit, usually including algorithmic and hardware-specific optimizations. Such optimizations work only for minor problems, but performance gains can be significant [43].

3.2.1 LLVM and Intermediate Representations

The `LLVM` project is a compiler infrastructure used in many programming languages. It was initially developed for the C programming language but has since been extended to support many others [44]. The main feature is to provide unified access to a program's abstract syntax tree representation. An intermediate representation (IR) represents the code, which a series of passes then transforms. Each pass can optimize the code in a specific way and stretch over multiple iterations. With each pass application, the code is *lowered* in abstraction, getting closer to the machine code. Later stages, which are hardware-specific, can be swapped to target different architectures.

`LLVM`-based systems are already widely used in HPC in the form of `NVCC` the `NVIDIA CUDA` compiler and other accelerator compilers. Hence, it is only natural to consider `LLVM` as a basis for quantum compilation. The extension to the quantum domain is straightforward. Optimization stages, as described before (see Section 3.2), can be implemented as `LLVM` passes. The process of lowering is also similar, where platform-specific optimizations follow generic ones. Leveraging the `LLVM` infrastructure immediately results in consistent performance and offers a familiar environment for developers. In supported languages, this also allows for mixed classical-quantum compilation, as the quantum part is also considered valid in the host language. There are now multiple frameworks that follow this paradigm, such as `cuda-q` [45], `XACC` [46], `Catalyst` [47], `Q#` [38] the `Intel Quantum SDK` [48] and others. Most of them provide additional features, including runtime environments so that I will discuss them more closely in section 3.3.

These frameworks share another commonality: based on the Quantum Intermediate Representation (QIR) [49]. It is a common intermediate representation for quantum programs based on `LLVM`. Functions represent quantum operations, which the program can call with qubits as arguments. These arguments are simple opaque pointers that are made available during allocation. I give an example in the listing below.

```
1     define void @BellPair__body(%Qubit* %qb1, %Qubit* %qb2) {
2 entry:
3   call void @__quantum__qis__h(%Qubit* %qb1)
4   call void @__quantum__qis__cnot(%Qubit* %qb1, %Qubit* %qb2)
5   ret void
6 }
```

Listing 3.3: Bell pair function defined in QIR.

It has gained some traction as it promises interoperability between different quantum frameworks and backends. Access to quantum hardware is possible through a not further specified *quantum runtime*. Primarily, QIR is a common target, a checkpoint between compilation and execution. For optimization, QIR is not ideal, as not all necessary information is immediately accessible. One can use `MLIR` [50] representations as a solution. They can encode optimization-specific information more efficiently and aid the optimization process. These so-called *dialects* can be shared among frameworks, which reduces work duplication. Other than QIR, OpenQasm3 [51] is also used as an intermediate representation. It is on a higher abstraction level, closer to the original quantum program, and eventually will become a standalone programming language. In the meantime, OpenQasm provides a serialization format for quantum programs, especially within IBM's infrastructure.

The choice of intermediate representation has some implications for the runtime. OpenQasm includes complex logic dependent on classical control flow, which, in theory, runs on quantum hardware. This execution must happen in a *near-time* context on classical hardware as a co-processor. A compiler must, therefore, be able to separate everything that can not resolve in *real-time* on the controller. For the controller, a global orchestrator is tracking time stepping [51]. The communication and data interaction between the controllers is not specified and shifted to the implementation. In the picture of QIR, there are minimal requirements toward the runtime, namely allocating and freeing qubits for use. This definition provides more of an ABI to program against; a runtime library can handle the execution. Further, vendors can specify hardware profiles. Each profile provides a set of capabilities that the hardware realizes physically. These profiles enable hardware-agnostic optimization while simultaneously providing different platforms to target.

3.2.2 tKet

Most optimization techniques are either standalone or embedded in a larger framework. End-to-end tools are uncommon, as generic approaches require high development and maintenance effort. One of the few exceptions is `tKet` [52]. It considers itself a platform-agnostic NISQ compiler. Initially, it covered only the ion trap hardware of Quantinuum, but since then, its functionality has extended to other platforms. It interfaces with common quantum frameworks and intermediate representations, such as `Qiskit` and OpenQasm. Contrary to other frameworks, it is not based on `LLVM`, but it still offers performance by using a Rust backend.

`tKet`, as a standalone tool, offers a case study for successful software development in the quantum domain. It compares similarly to the classical compilation infrastructure, where end-to-end compilation is standard. One of the main reasons for its success is its interoperability with other frameworks. It offers integration with hardware and cloud providers of quantum hardware, but also high-performance simulators. Further, it can interpret programs provided by other languages, like `Quil`, or intermediate representations, like QIR. This interoperability makes the adoption of `tKet` easier, and due to its interoperability with other frameworks, it can act as a drop-in replacement for specific stages of the compilation process. One drawback is the re-casting of the problem to its DAG-based representation, which induces a runtime overhead when converting back and forth.

A standard interface unifies the access to the hardware. Logical predicates encode hardware properties, which `tKet` can check a circuit against. Only circuits that satisfy the predicates can continue onto the hardware. One such predicate is the possibility of adding conditional logic based on quantum data. There is an advantage of writing hardware-specific circuits [43], but due to the volatility of the hardware, this is only feasible with exclusive hardware access. `tKet` offers a batched, total-job, hardware-agnostic approach. In the latter, it checks circuits at runtime against hardware predicates. An interesting feature for HPC is the ability to asynchronously access quantum backends. Typically, halting an entire system for a quantum computation is not feasible. Instead, `tKet`s backends can store information over multiple sessions, allowing complex computations to continue and react over callbacks. Direct hardware access is not part of `tKet` and, therefore, contains restricted access to the public interfaces of the hardware providers.

`tKet` adopts the concept of passes for compilation. A pass can define preconditions, which should hold before the pass by defining predicates. Passes can be composed to form a pipeline; the default consists of placement, routing, gate decomposition, and base gate translation. Most of these passes are localized; circuits are transformed to a *ZX-Calculus* description for circuit optimization. This graph-based representation emits specific optimization techniques that translate well to the quantum use case. To speed up compilation `tKet` offers partial and symbolic compilation. Partial compilation allows for the reusing of consistent parts of the circuit by only recompiling the changed parts. This shortcut restricts the extent of procedures; for example, the mapping must occur at the beginning of the compilation of a circuit. Symbolic compilation performs optimization for parametrized gates without evaluating the parameters. This technique speeds up the compilation, where the entire circuit structure is predetermined.

3.3 Runtime Environments

Quantum programming languages and compilation toolchains abstract the quantum program from the hardware. A runtime environment links the two together and ensures smooth execution. In the classical sense, a runtime environment exposes an ABI, typically through system calls, to the running program. The system calls are, loosely

speaking, primitive operations efficiently implemented by the operating system. Due to the hybrid nature of quantum programs, where the classical part is responsible for constructing or calling quantum circuits, the runtime environment is more complex. Not only does it still offer all classical functionality, but it also needs to provide access to the quantum hardware. The primary responsibility is exposing the quantum capabilities. These capabilities include allocating the qubits, executing the quantum operations, and reading the results. While these operations map nicely to existing classical concepts, their realization fundamentally differs. Preparing a qubit register is a process that covers the entire device, where control electronics individually prepare each qubit. Operations are physical manipulations of qubits, usually using laser pulses depending on the calibration and calculation of the correct waveforms.

Besides the quantum operations, the runtime environment must handle classical tasks. One can achieve this by embedding the quantum runtime into a classical system. For example, `tKet` uses a modified WebAssembly runtime accessible through their Python interface. In such a sandboxed environment, the runtime can execute almost arbitrary code inside a virtualized operating system. This indirection is a suitable approach for a cloud platform, but for HPC, any indirection is undesirable. The overhead of the virtualization layer is unacceptable. Hence, a more direct approach is necessary. Optimal is a runtime that supports hybrid capabilities, bridging the borders between classical and quantum execution. Additionally, it should support some HPC features for integration purposes.

HPC primarily focuses on performance and scalability. This notion includes many vital metrics, such as sustainability and high utilization of systems. These metrics arise from lower-level concepts, such as latency, throughput, robustness, and resilience. HPC runtime environments contain several components to achieve these performance goals, Standard components are a schedule or resource monitoring and management built on top of heterogeneous hardware optimized for performance and low latency. In this system, the quantum runtime must integrate smoothly to ensure operability. The goal is a tightly integrated system for maximal performance while maintaining loose coupling to allow for flexibility. An additional challenge is the use of error correction techniques. Even though quantum computing is probabilistic, the expected operation should still ensue most of the time. For experts, there might be an upside to choosing specific error correction options, but this should be handled automatically for most users. Besides choosing a suitable code, which can depend on hardware and target application, implementing and controlling the error correction is computationally expensive. The runtime environment must respect these conditions while competing with highly optimized classical code. Quantum capabilities do not guarantee quantum advantage if no classical systems enable their smooth operation. In the following, I will compare two approaches to this problem against the requirements of HPC.

3.3.1 Qiskit

`Qiskit` [40] is a Python SDK for quantum computing developed by IBM. Initially, IBM developed it as a high-level interface to their in-house quantum hardware. By offering

real hardware access, it has gained a large user base. It still serves as a baseline for quantum computing, and many other tools benchmark their results against it. Currently, `Qiskit` includes multiple independent components, including simulators, application-specific libraries, and a runtime environment. Additionally, it offers IBM's quantum hardware and the possibility to integrate your custom hardware. Combined with the high-level interface, this makes `Qiskit` a convenient tool for quantum research. An example VQE calculation is below in Listing 3.4.

```
# Select hardware
service = QiskitRuntimeService(channel="ibm_quantum")
backend = service.least_busy(operational=True, simulator=False)
pm = generate_preset_pass_manager(target=backendtarget,
        optimization_level=3)

# Define VQE and customize it for hardware
hamiltonian = SparsePauliOp.from_list([
    ("YZ", 0.3980), ("ZI", -0.3980), ("ZZ", -0.0113), ("XX", 0.1810)
])
ansatz = EfficientSU2(hamiltonian.num_qubits)
ansatz_isa = pm.run(ansatz)
hamiltonian_isa = hamiltonian.apply_layout(layout=ansatz_isa.layout)

# Run VQE with the estimator primitive
with Session(backend=backend) as session:
    estimator = Estimator(session=session)
    estimator.options.default_shots = 10000
    res = minimize(
        cost_func,
        x0,
        args=(ansatz_isa, hamiltonian_isa, estimator),
        method="cobyla",
    )
```

Listing 3.4: Example of a VQE calculation in `Qiskit`.

The program consists of three steps. First, it defines the hardware, sets up the VQE calculation, and then triggers the quantum calculation. From these, the first and last steps interact with the runtime environment. Terminology in `Qiskit` is slightly convoluted, as `Qiskit` follows a platform-as-a-service approach. Developers can also access most functionality through an API or a web interface.

`Qiskit` offers a comprehensive toolchain for developing quantum algorithms. After users select an algorithm or design a circuit, they can use `Qiskit` to compile them (this is called transpilation in `Qiksit`). The process is highly configurable, and the user can select an optimization level from 0 to 3. This choice governs how many optimization passes the *transpiler* applies to the circuit and how often it repeats them. Compilation techniques fall into several categories according to stages, similar to what I already discussed in Section 3.2. Each stage comprises several passes, which users can rearrange arbitrarily. This concept is similar to the LLVM approach but uses a

Python-based internal representation. It utilizes the DAG representation of the circuit. For most passes, this is sufficient; if passes require more complex structures, they implement custom representations. Based on hardware information, a pass-manager can be automatically generated or defined by manually setting stages. Using an unrestricted backend as a proxy, this could function as a target-agnostic compilation. `Qiskit` offers the concept of parametrized circuits, but pre-optimization happens only at runtime. As Python is an interpreted language, the distinction between runtime and compile time is unnecessary. In high-performance scenarios, Python can be a limiting factor. `Qiskit` migrates to Rust-based optimization for performance-critical parts, but the Python interface remains. So far, this is limited to the compilation process, but the runtime is still Python-based.

OpenQasm and OpenPulse [53] (or a combination) offer an alternative representation to circuits in `Qiskit`. OpenQasm is a lower-level representation of the quantum program but aims to be feature-complete. In it, one can define complex logic, such as measurement-based control flow, which integrates nicely with IBM's newer hardware. Circuits can also contain timing information, which the runtime can enforce. OpenPulse is an even lower representation, where pulse sequences implement circuits. This representation allows for granular control and delivers nice performance improvements [43,54]. Unfortunately, there is no direct integration with runtime services. Hence, the overhead for switching between representations affects both.

The estimator primitive in Listing 3.4 is part of the runtime environment. It initially employed a total-job approach, where the entire hybrid program runs on IBM hardware. Hardware time is allocated for the entire job, making a near-time interaction between classical and quantum parts possible. A co-located server still executes most classical code, but real-time interaction is the long-term goal. The functionality is part of `qiskit-serverless`, which allows quantum hardware vendors to use this in their setups. As part of the environment, it applies error mitigation and other techniques; users can opt-out. Error correction could eventually replace this, but some techniques remain valid, as error correction does not ensure 100percent accuracy. Such interference with the program could have undesirable side effects when optimizing for performance. Still, the runtime environment is a step toward tighter integration, but its Python nature is unsuitable for HPC. Besides the fact that Python is not a high-performance language, crucial features are missing. For example, control over memory allocation and communication between nodes is lacking. One could overcome such shortcomings by implementing a custom middleware layer, which would require significant effort. The separation of compile-time and runtime is even more complex in interpreted languages. As IBM focuses on the cloud market, it has not planned an alternative approach to these scenarios. Depending on the development of OpenQasm, it could overcome some of the impediments, but this is not clear yet.

In summary, `Qiskit` has multiple desirable features, which makes it a natural choice for quantum programming. Due to its significant community, it offers multiple state-of-the-art methods, especially for compilation. Hardware integration is also advanced, approaching real-time interaction with tightly integrated systems. The integration comes with the cost of tight coupling as adoption outside the ecosystem is tedious. For HPC,

this is a significant obstacle. I have outlined a possible solution, building a custom middleware layer, but this is not feasible for most users. A more generic, performance-centric approach is necessary, which contains established patterns and components and supports existing libraries such as Message Passing Interface (MPI). `Qiskit`s feature completeness can serve as a target for such a system, but the implementation has to be fundamentally different.

3.3.2 XACC

`XACC` [46] is a quantum programming framework that Oak Ridge National Laboratory developed. In contrast to `Qiskit`, it is focused on scalability and performance. It is written in C++ and can act as middleware between quantum programming languages and quantum hardware or as a standalone tool. The core concept behind `XACC` is the interpretation of QPUs as accelerators in the fashion of classical co-processors. A standard interface abstracts the hardware; it makes no distinction between simulated and real hardware. Programmatically, an *AcceleratorBuffer* object resolves the hardware and mediates with it to achieve the abstraction. The buffer stores measurement results after a quantum program has been submitted and successfully executed. From the end-user perspective, this is very similar to the mode of operation of a classical accelerator.

```
1  #include "xacc.hpp"
2  program = xacc::getCompiler("qasm")->compile(R"(
3  ----H-0
4  ----CNOT-0-1
5  ----Measure-0-1
6  )");
7  auto buffer = xacc::qalloc(3);
8  auto qpu = xacc::getAccelerator("ibm:ibmq_valencia");
9  qpu->execute(buffer, program);
10 std::map<std::string, int> results = buffer->getMeasurementCounts();
11 auto expValZ = buffer->getExpectationValueZ();
```

One of the significant differences to `Qiskit` is the separation of compile-time and runtime due to the compiled nature of C++. The above example defines and compiles a 2-qubit GHZ circuit during the runtime. But it is also possible to define the program as an external *kernel* and compile it ahead of time. `XACC` allows for parametrized circuits, which it also optimizes ahead of time. A second compilation step can occur during the runtime of an `XACC` program. Initially, this supported only a custom intermediate representation called XASM, but now QIR is also compatible. For compilation, the AST and DAG are available for use in services that extend the compilation pipeline with novel capabilities. In the above listing, the call `xacc::getCompiler("qasm")` returns a service that can compile an OpenQasm program. Multiple compilers are available through a service registry. Similarly, optimizers and accelerators are encapsulated in services, allowing to easily switch between them. This encapsulation requires tools to provide a C++ runtime library, but integration is straightforward.

`XACC` can be considered a middleware layer between algorithm description and hardware execution. Although it offers some abstractions, mainly for variational algorithms, the description of the problem is intended to reside elsewhere, e.g. QCOR [36]. `XACC` governs the execution of programs by handling circuit compilation and interaction with quantum hardware. The runtime allocates qubits and classical memory buffers accessible from the accelerator. It can manipulate parameters and results freely through these classical buffers. To make the interaction seamless, the integration of QPUs is an integral part of `XACC` Two mechanisms for this are possible: providing full functionality from a hardware service or extending existing accelerators. The prior is straightforward: the available instructions must match the compiler output. A decorator pattern is available for the latter, which developers can apply to multiple accelerators. Enabling real-time measurements on existing hardware is one of the possible extensions.

A central part of `XACC` is high performance. Scaling QPU access to multiple hardware nodes is possible through the decorator pattern. Interoperability with MPI is an optional feature by which the execution of quantum programs distributes over multiple nodes. Depending on the configuration, Different ranks can access the same hardware or different hardware. Some advanced features are possible by combining these patterns. For example, one can mix quantum hardware with simulators. During my experiments, I was able to build a mixed-target quantum program. I enabled writing wrappers around two hardware simulators and accessing them simultaneously. The communication happened classically, and the quantum programs were carried out in parallel. This prototype served as proof of concept and provided only severely limited functionality. Both hardware backends block simultaneous access from different nodes, which corresponds to just two copies of the same program. Performance was also sub-par, which I did not investigate further.

In general, the approach of `XACC` is more suitable for HPC than `Qiskit`. It provides a clear layered architecture and targets deployment in a local environment. Its service-based architecture naturally extends the notion of accelerators to the quantum domain. A significant issue is the compatibility with different versions and the lack of consistent documentation. These caveats make adopting `XACC` difficult and require a significant learning investment. Error correction is also not part of the runtime, and the service architecture can be limiting in achieving unusual setups.

3.4 Gaps in HPC

In quantum computing, a variety of software tools are available. Every aspect is covered, from specialized quantum programming languages to general-purpose quantum compilers to runtime environments. Inherently, many of these tools are unsuitable for HPC, but with some modifications, they could be integrated into an existing framework, although with some limitations. The introduction of QIR is an essential step toward interoperability, but the available options are still limited. Compiler infrastructure already exists but must be compatible with the software stack.

For runtime environments, I have highlighted two approaches. `Qiskit` promises high usability and tight but limited hardware integration. For this, it sacrifices performance and limits the user to a specific ecosystem. `XACC`, on the other hand, addresses these issues but is less user-friendly. The service approach allows looser coupling to components, but complexity can be permissively high. An optimal solution would combine the best of both worlds.

These tools do not sufficiently focus on some issues. The integration with existing HPC infrastructure is largely lacking. Resource management and data management are not covered. For heterogeneous hardware, scheduling is necessary to utilize hardware according to its characteristics. Mixed use of QPUs and simulators is also not fully explored. Data availability and offloading, an essential aspect of hybrid algorithms, is only covered in a single-node, single-user scenario. Furthermore, standard metrics like reliability and robustness are barely addressed.

Finally, error mitigation and correction and their interaction with the runtime are mainly unknown. On one side, the resulting quality of a solution should be as high as possible, but on the other side, the user should be in control of the process. For this, best practices still need to be established. Once the hardware scales to the fault-tolerant regime, computation accompanying error correction procedures will take much of the runtime. For this, the runtime has to be scalable and flexible enough to accommodate various mechanisms.

4 Unified Toolchain

As I have shown, integrating quantum computing into HPC is complex. Running quantum programs consists of multiple steps; each is a challenge. A typical software development pattern is splitting complex tasks into smaller, more manageable parts. Each problem can be solved individually, combining the solutions to form a complete system. This combination adds some overhead for the integration, allowing for more flexibility and easier maintenance. At its core, this is the idea behind the *Unified Toolchain*. I thoroughly investigated this idea in my previous work, Seitz et al. [55]. This chapter is an extension of this work.

The *Unified Toolchain* provides the underlying architecture for solving the problem of integrating HPCQC. It defines the interactions of components and the intended flow of computation. These definitions are suitable for building reliable and scalable components for arbitrary hardware. The separation into offline and online components is fundamental. The toolchain also clearly describes all responsibilities of runtime components, providing an overall scope for each. As soon as standards emerge, they can be incorporated, providing a robust software stack.

Figure 4.1 shows an overview of the toolchain. It consists of multiple layers, each representing a step in running a quantum program. Each layer contains multiple components that connect through well-defined interfaces. This overview provides a generalized view that highlights necessary components and their interactions. A realization of this can look different, for example, the Munich Quantum Software Stack [56] `MQSS`, but the general concepts are the same. The layers belong into two categories based on their execution time: offline and online. Offline refers to steps executed before a program runs, while online refers to steps executed during a program's execution. This separation is a crucial aspect of the toolchain; existing systems typically do not strictly enforce it, which is not well suited for the HPCQC domain.

To show the problem of the current systems, first, let us consider two scenarios. In a single-user environment, the user prepares the quantum program to run it on the quantum computer. Typically, the user has limited access to quantum resources and is mainly interested in the program's results. The toolchain is limited to the built-in functionality provided by the hardware vendor. For example, using IBMQ as a reference, the user can write a quantum program in `Qiskit` [40], compile it to the IBMQ backend, and run it. The tools are limited to the `Qiskit` ecosystem, and the backend has to be specified early on. Theoretically, the pass manager allows for some customization, but they will choose one of the three pre-defined optimization levels in practice. Replacing any of the components takes significant effort, and the user has to provide bindings to the Python front end. Most of the compilation will be run on the user side, limiting the available resources. Once the program is submitted, the user cannot control the execution. The

user must trust the hardware vendor to provide the best possible results. In the worst case, proprietary software comes into play, interfering with the user's intention. The hardware vendor makes any decision on the hardware side, and the user has to wait for the results. There is also no opportunity to optimize the program, as the user lacks the necessary information. Additionally, any profiling is limited to the algorithm's results and goes no further than the local program.

In contrast, a HPC environment has different requirements. Instead of a single user, multiple users share the resources. Each might have a unique application that offers multiple optimization paths. Various tools must be available to utilize the restricted hardware resources efficiently. The toolchain has to be flexible enough to support different quantum programming languages, compilers, and hardware backends. An essential aspect is the uncertainty of the hardware. The hardware might be unavailable for extended periods, and *optimality* might change based on this information. Some users might be happy to change hardware, while others might want to stick to a specific one. In any case, the toolchain needs to be able to adapt to these changes. Further, the access to HPC resources offers more potential for computationally intensive optimization. Other requirements arise from including further classical aspects, such as fairness. These requirements translate to a need for a more modular and extensible design for the toolchain. The hybrid nature of the use cases imposes additional classical constraints on the toolchain.

My design of the *Unified Toolchain* facilitates the integration of quantum computing into HPC environments. It combines classical software engineering principles with quantum-specific requirements. In the following, I will discuss the components of the toolchain and how they interact. Some of these components I will provide as prototypes to demonstrate the feasibility of the design in later chapters. A priority is the interfaces between the components and the overall flow of the application through the toolchain. I will conclude with remarks on the architecture and implementation details.

4.1 Overview

Software development in quantum computing follows the same principles as classical software development. There are two main areas of research: developing and improving quantum algorithms and developing tools to run these algorithms on quantum hardware. Due to the limitations of the hardware, tool development is focused on squeezing the most out of the available resources. As a result, and as a side effect of the unique hardware properties, this resulted in a fragmented ecosystem of multiple tools, each solving a specific problem. Hence, the term toolchain describes a collection of tools used to optimize a quantum program, working in a defined sequence. A significant challenge is either integrating these tools into existing systems or overhauling the existing tools to fit the requirements of a new system. I design the *Unified Toolchain* to address these challenges.

Modularity and extensibility are key design aspects needed to achieve this goal. It should be possible to replace any component based on the user's needs without affecting

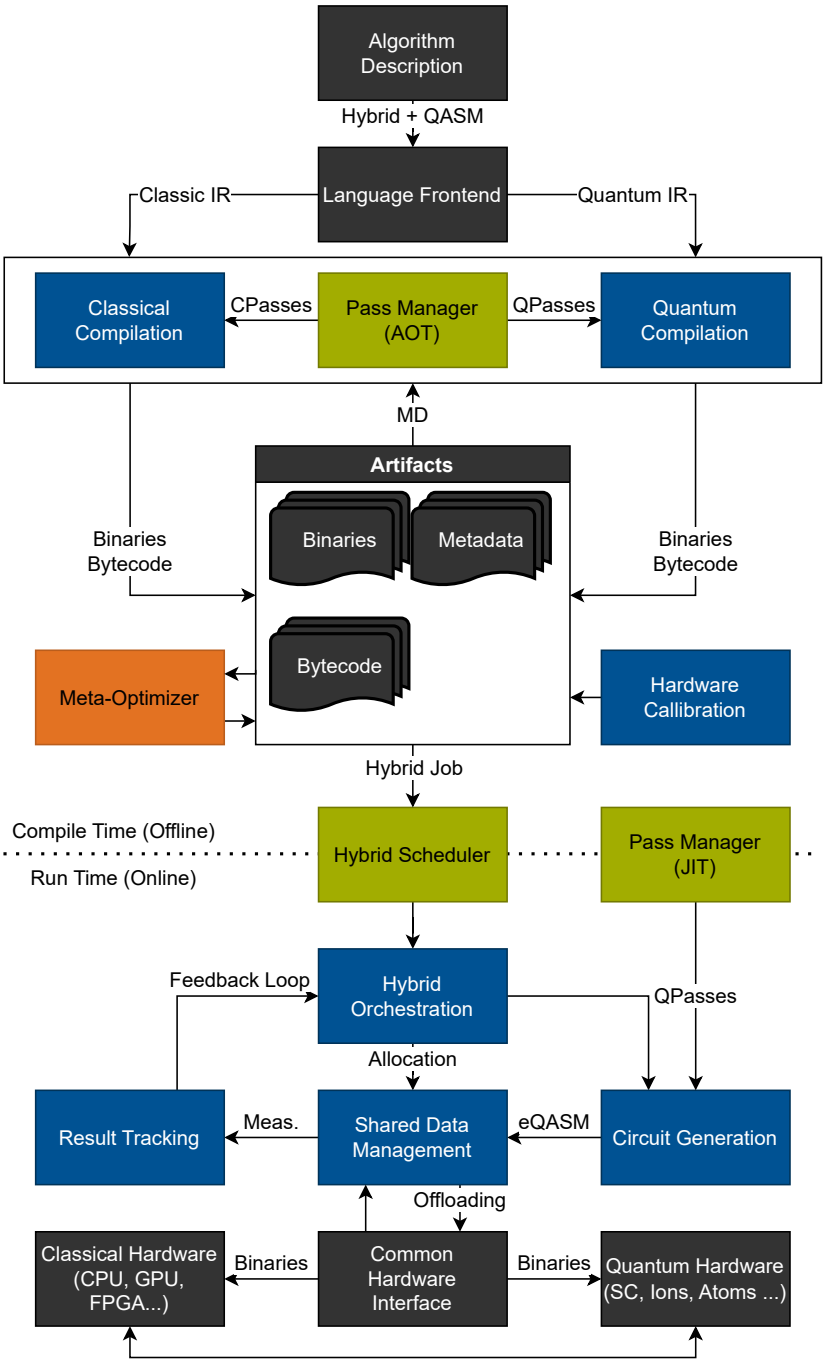


Figure 4.1: The *Unified Toolchain*intended for hybrid classical-quantum workflows. My work is centered around a generic model, which incorporate classical HPC workflows and mirrors them to the quantum domain. The toolchain is designed to be extensible and modular, which are common practice in HPC. Similar approaches are currently emerging in performance focused quantum application areas, e.g. the `MQSS`.

the rest of the system. In extreme cases, facilitating unique functionality from custom hardware should be possible by replacing the corresponding component in the toolchain without expert knowledge of the rest of the system. Such an agnostic approach has multiple advantages. First, it allows for encapsulated functionality, which developers can build against black-box implementations of the other components. This technique avoids the overhead of reimplementing multiple parts of the toolchain, potentially numerous times by different users. For example, almost every existing system (e.g. [40, 45, 47]) provides a similar, slightly different way to represent quantum circuits. Additionally, most systems provide conversion capabilities between these representations. Second, it allows for faster innovation cycles, allowing the adoption of new optimization techniques earlier. The resulting system is more flexible and can adapt to new hardware or software requirements more efficiently.

The toolchain targets deployment in a HPC environment. It must respect some standard practices to ensure the toolchain's proper functioning. One noticeable practice is the separation of offline and online components. Offline components function before the program runs, while online components operate during the program's execution. This typical pattern in classical computing allows for more efficient resource utilization. Typically, these two phases are compilation, which happens at compile time, and execution, which occurs at runtime. In quantum computing, the distinction is not as clear, as the quantum circuits are usually only fully known at runtime. This uncertainty is due to the nature of quantum programs, often generated based on ad hoc computed information. Subsequently, a second compilation round is necessary to generate the actual quantum circuits. Such compilation is often computationally intensive; therefore, it is beneficial to separate it from the actual execution as much as possible.

4.2 Components

On the highest level, tools can occupy different roles in the toolchain. Figure 4.1 shows these meta components and their relationships. They are defined by their interaction with the other components, their interfaces, and how they enable the execution of quantum programs. Depending on its functionality, a tool can be part of multiple components, but developers should respect the separation of concerns. Additionally, a distinct characteristic is the hybrid nature of the toolchain. This fact significantly impacts the functionality of some of the components, which will become apparent later. In this section, I focus on defining the components and their interfaces. The actual implementation details I discuss in later chapters.

4.2.1 Algorithm Description

The first step in running a quantum program is to describe the algorithm. This description usually relies on a quantum programming language. As already discussed in Section 3.1, this QPLs significantly impacts the complexity of other components. Python is the most popular language for quantum programming, as it is easy to use and has a large existing ecosystem. `Qiskit` [40] is a widely used Python library that provides

a high-level interface to quantum hardware and is frequent in the quantum computing community. In it, users describe quantum algorithms by their respective circuits, optimized locally before being sent to the cloud hardware. The expressive power of quantum circuits is sufficient but exhibits high complexity akin to assembly programming. Most users are not interested in the low-level details of the quantum circuits but rather in the high-level algorithm. Other languages, like `Q#` [38] or `Quipper` [57], provide a more abstract view of quantum algorithms. For example, conditional statements based on quantum data are possible in `Q#`.

Whatever the language, the algorithm description has to consider the interplay of classical and quantum parts. The control flow is purely classical for current hardware, while the data flow is quantum and classical. This distinction implies careful consideration of the interfaces between the classical and quantum parts. Additionally, some classical instructions should be executed on quantum hardware to reduce the communication overhead. For example, repeat until success [58] patterns do not require sophisticated logic but would impose a significant overhead if a system were to process the data on the classical side. Based on accelerator programming, a natural solution is to separate quantum logic in the algorithm definition. So-called quantum kernels, self-contained quantum programs, arise from this separation.

Programmatically separating a program's quantum and classical sections is a non-trivial task. Indicating quantum sections with special syntax is a straightforward way to let users make this decision. Programming languages can achieve this using a domain-specific language or by extending an existing language; the latter is more flexible but requires more effort. Additionally, the kernels are known at compile time, which enables optimization. The *language frontend* parses the quantum program and extracts the quantum kernels. It acts as the interface between the algorithm description and the rest of the toolchain. The program is transformed from a hybrid program to an intermediate representation and then passed to the next component. In this representation, the quantum and classical parts exist strictly separated.

4.2.2 Optimization

Optimization of hybrid quantum programs is a complex task, taking into account interleaved quantum and classical parts. Classical parts may generate quantum programs during runtime, which should be optimal given some optimality criterion. For simplicity, I assume a two-step optimization process for the quantum part; additionally, I require that the classical part be optimized separately by an existing compiler. During compile time, limited information is available to optimize the quantum kernels. Later, generated quantum programs are optimized at runtime based on the actual hardware and the newly available information. Besides the asymmetric information accessibility, there are more discriminating factors. In HPC, compilation should happen as early as possible to avoid runtime overhead. Assuming a hybrid HPCQC program, the worst-case scenario would be to halt an entire HPC-cluster because it depends on results from a quantum computation. Given the computational complexity of some optimization techniques (c.f.

Chapter 6), this is a valid obstacle. In most cases, this could make the quantum program permissibly slower than a classical program.

Optimization is usually one goal of compilation. Due to the sheer complexity of compilation, a common approach is to conceptually split the optimization into multiple phases. Independently, these phases run in a sequence, each responsible for a specific task. These phases are called passes in `LLVM` [44], a widespread compiler infrastructure. Quantum computing has adopted this approach of sequentially *lowering* as it fits the hardware model. On one hand, optimization is necessary to make the quantum program run efficiently. On the other hand, the circuit model does not typically reflect the hardware, which makes specific compilation steps necessary. The optimization consists of two parts based on separating offline and online components. The former is executed during compile time, while the latter is executed during runtime. In a HPC environment, this admits the possibility to execute parts of the optimization agnostic of the hardware. As a side effect, this also allows for more extensive optimization, as the available resources are not as constrained as during runtime.

The first part of the optimization is global optimization, which considers the complete problem. This extent contrasts localized, so-called *peephole* optimizations, which only consider a small part of the problem. Typically, one represents a quantum algorithm in an abstract form, not in a circuit model. For example, researchers might express the algorithm with a Hamiltonian and then use tools to transform it into a circuit. This initial transformation can already be optimized, as the Hamiltonian might contain redundant information, such as symmetries. While this is specific to the algorithm, some optimizations are generally applicable or at least relevant to a broad range of algorithms. Once the circuit is generated, the actual compilation process begins.

Like the transformation from assembly to machine code, a compiler must rewrite the quantum circuit into a hardware executable form. For most hardware, the available operations are not the default operations in the circuit model. For example, the CX gate, ubiquitous in quantum circuits, is not directly available on superconducting qubits. Instead, a sequence of single and two-quit gates implements the CX gate in hardware. This sequence uses the so-called *native gate-set,* which differs between hardware. These sets also rarely include multi-qubit gates beyond two qubits. Therefore, a compiler must decompose any multi-qubit gate into a sequence of one- and two-qubit gates. Less common are connectivity restrictions, which limit the qubits that can interact. In such hardware, mostly superconducting qubits, the qubits are arranged in a grid, and only neighboring qubits can interact. Before executing the actual gate, the compiler must insert qubit swaps to move the qubits into the correct position. This process is called *routing" and is usually closely interconnected with "mapping.* Mapping is the process of assigning the logical qubits to the physical qubits. The main goal is to minimize the noise in the system. Each connection can have a different error rate, and the compiler should consider this. Each swap is also subject to noise, which plays a meaningful role. The compiler needs detailed information about the hardware to execute the steps above. The information becomes only available at runtime, which I will discuss next.

Intermediate representations are the interface between the algorithm description provided by the language frontend. Some passes might require different representations,

but the general idea is to have a standard representation that all passes can use. Such a representation improves the modularity of the toolchain, as developers can independently provide their passes. From the pool of available passes, the *Pass Manager* can choose the most suitable passes for the given problem. The final output is a pre-compiled quantum program, which then passes on to the next component. In the cases of fully specified quantum circuits and classical programs, the output is binary files; Otherwise, bytecode, which requires further processing at runtime, is emitted. The compilation could reveal some internal structure or make assumptions to successfully transform the problem. Such information can be complementary to enable further optimization at runtime or inform the hardware decisions. Hence, it is made available to the runtime components, including other information about the compilation process, such as the executed passes. This information forms the metadata in Figure 4.1. Components shaded in blue generate this metadata, while components shaded in green consume it.

4.2.3 Runtime

At runtime, the entire program starts execution. I will mainly focus on the quantum part, assuming that a classical runtime with some quantum extensions executes the classical part. The runtime starts with the program's classical part and continues until it invokes a quantum kernel. Once the kernel is fully specified, the runtime can execute it on any hardware. The component orchestrating the execution is the *runtime environment*. It consists of multiple subcomponents, each responsible for a specific task. The distinction between compile time and runtime also exists in classical programs but is less pronounced. The main difference is the necessity of generating a runnable quantum program. Just-in-time compilation is comparable, but the extent is far more restrictive for quantum programs. Additionally, the runtime environment has to consider the hardware-specific information that will only be available at runtime.

The interface to the runtime environment is a job description. This description includes a mix of quantum and classical programs compiled to different degrees. As corresponding artifacts, the job description contains resource qualifiers for binaries, bytecode, and metadata. The user triggers the program submission to the system, after which the scheduler takes over.

4.2.3.1 Scheduling

Scheduling is the process of assigning resources to jobs. Concretely, a user can request specific properties for a job, such as the system memory or the number of qubits. From the system's perspective, the scheduler has to consider the available resources and the current load. With that knowledge, the goal is to decide which hardware to assign to the job and which time slot. During this process, the scheduler has to respect constraints, and the assignment should follow a desired optimization criterion. I will discuss the scheduling process in more detail in Chapter 7. A distinctive feature of the HPCQC environment is the interplay between the quantum and classical parts. Especially in variational algorithms, tight interaction loops are necessary. Similar scenarios in classical

settings are collected under the term *gang scheduling*, although the physical properties of quantum hardware impose additional challenges.

Besides the scheduling, the runtime environment executes and controls the quantum program. A clear separation between these tasks is not always present; I consider the *orchestration* a separate component. It takes over some of the responsibilities typically found in the scheduler. Data management, especially in distributed systems, is a critical orchestration aspect. The orchestration has to ensure that the data can be available where needed. For now, only classical data is relevant, but operating on quantum data will become a likely scenario in the future. All components handle communication independently, akin to MPI. The orchestration is partly responsible for the system's health, as it has to monitor the execution and react to failures.

As part of the runtime environment, the scheduler interfaces with the system through the job description. The scheduler provides additional information about reserved memory, assigned qubits, and timing information. During the execution, the orchestration uses this information to manage the program flow. By dispatching the quantum programs to the hardware and triggering their execution, the runtime environment acts with the hardware. However, an abstraction layer hides the hardware, which is necessary to support multiple (heterogeneous) hardware backends.

4.2.3.2 Generation

A generation process is necessary before executing the quantum program on the hardware This process is very similar to the compilation process, with the main difference being the available information. The scheduler has fixed the target hardware, which allows for specific optimizations that were not possible previously. Additionally, the system generates runtime parameters and substitutes them into the program. As a restriction, the generation should now be as fast as possible, as the classical program potentially halts until quantum results become available. But, instead of idling, the resources can also aid optimization, which can benefit techniques previously permissible. In theory, the generation process can reuse the previous compilation passes; I will give more detail in Chapter 6

The orchestration triggers the generation process, which provides the necessary information. It can also consume the metadata generated during compilation to judge which passes might no longer be applicable. Kernels follow a continual lowering to the hardware-specific representation. The output is a binary file in a standard format and associated data registers.

4.2.3.3 Error Mitigation

Error mitigation is another aspect of the runtime environment, especially for NISQ devices. In parts, the optimization step already considers the error rates of the hardware. Some unavoidable errors can still be mitigated during runtime. One technique is to utilize noise-free simulation techniques in conjunction with noisy execution. I will cover similar techniques in Chapter 6.

The core concept of error mitigation is the tradeoff between fidelity and runtime. Ensuring a high fidelity can be computationally expensive or consume more resources. This fidelity is only necessary if the algorithm requires results of a certain quality and the hardware cannot provide them. Practically speaking, one can realize this as a more elaborate generation pass. However, some execution parameters need readjustments to ensure the desired fidelity. In extreme cases, this might lead to changes in the algorithm itself that would be more beneficial to the hardware.

4.2.4 Error Correction

The paradigm shift toward fault-tolerant quantum computing is a significant challenge but inevitable. Error correction will be necessary, and the runtime environment has to facilitate it. In Chapter 8, I will discuss error correction and its implications on the runtime environment. Researchers are currently debating at which level error correction should be integrated. A likely position is after the generation, as the error correction is hardware-specific. Alternatively, one can apply error correction on the algorithm level tied to the optimization process or at the latest hardware level. Directly integrating error correction into the hardware is possible, but it is unclear if this will be feasible.

In the following, I will assume error correction occurs during the circuit generation. This addition will require restructuring the passes, as other intermediate representations are more suitable for reasoning in the logical space. A likely candidate is the ZX calculus, which is compatible with most error correction codes. One can directly generate an executable from the logical representation without returning to a circuit representation. The tailoring of the error correction has its own choices, which the system can influence based on the hardware. For example, it can include noise models in the error correction or use a more efficient error correction code. These choices also affect the classical postprocessing, which the runtime environment has to consider and communicate to other components.

Conceptually, the interfaces stay the same with and without error correction. The significant difference is the additional overhead, which has to be respected. Therefore, the toolchain must be flexible enough to support error correction at different levels to avoid unnecessary development efforts. Manipulating the components to include error-correcting measures is a significant challenge for software design.

4.2.5 Hardware and Simulators

On the lowest level, the hardware integrates physically into the system. The hardware can be either quantum hardware, simulated quantum hardware, or any variety of accelerators from the classical side. In Chapter 5, I will show how to efficiently simulate specific use cases and utilize them for the toolchain. Simulations can be either used as a drop-in replacement or complementary to the hardware. This choice also has implications for validating the software development process against multiple backends.

The interface has to allow for simple hardware replacement to make this possible. This genericness is possible through abstracting the hardware behind a standard interface.

A general purpose instruction set architecture (ISA) addresses this hardware interface, which further transforms into the hardware-specific instructions on the chip. Part of the ISA defines input and output data registers. Through these registers, the hardware can communicate with the software. Additionally, a special memory section for parametrized circuits should exist, which is accessed more frequently.

4.3 Architecture

The *Unified Toolchain* conceptually pursues a modular design. Ultimately, an implementation must reflect and facilitate this design. The HPC environment is an additional driver for architectural decisions. Tight integration between all components is necessary to achieve the desired performance But, to allow for flexibility, the components must also be loosely coupled. Such a design is not trivial; it requires a careful balance between the two.

The interfaces defined by each component are a key ingredient in the architecture. Similar to the trend toward unified application programming interfaces (APIs) in web development, a *component interface* is defined. The in- and output data structures are the expected behavior and the produced artifacts. Initially, they are defined empirically, but usage will refine them over time. Once the community adopts them, they can be considered a standard. The classical domain already provides some widely established examples in terms of standards. For example, one can leverage the LLVM infrastructure for the compilation process. The QIR [49] and its related dialects are a good starting point for the quantum domain. MPI is a standard for distributed computing suitable for orchestration. Due to its history, it is well-suited for HPCQC applications, but developing a proposal to integrate quantum is tedious.

A common problem is also the interoperability between the components. Most components are developed as standalone tools, barely considering integration. A widespread language like C++, already woven into the HPC environment, has some advantages. Many modern languages provide a foreign function interface to C++, which could provide the interface with the components. Especially in the LLVM ecosystem, it is a common practice to target a C++-compatible execution environment. A loose coupling is also possible by wrapping components in dynamically linked libraries. With this construct, one can replace single components without affecting the rest of the system at the discretion of the runtime environment.

The architecture follows the concepts described here. A primary driver program is responsible for each of the phases. Offline components are loaded within the compiler; online components are packed within the runtime environment. The user controls the overall configuration with compilation and execution flags. The drivers dynamically access the required libraries based on live execution information and the metadata, respecting the user-defined parameters.

4.4 Contribution

Software development in quantum computing is mostly piecework, as pointed out by Elsharkawy et. al. [41]. Individual tools, or pieces, are chained together to form a complete system. Currently, each tool is developed in isolation, with little consideration for integration. Integration must be done from scratch when building a new software stack. Some part of this effort is no longer necessary with a unified toolchain. The idea itself is not groundbreaking but necessary nonetheless.

In the upcoming chapters, I will showcase how to design tools substituting one of the components in the toolchain. I will show how to integrate these tools into the toolchain. This procedure focuses on defining the interfaces and produced and consumed artifacts. The focus of these implementations is the suitability for the HPCQC environment while adhering to the principles of the *Unified Toolchain*. As a result, the provided tools will provide the basic functionality of a hybrid runtime environment.

5 Simulation

At the core of quantum advantage is that quantum computers can solve specific problems significantly faster than classical computers. This statement implies that quantum computers can not be simulated efficiently on classical hardware. Although this is true, there are still many cases where classical simulation methods can guide the development of quantum algorithms and hardware. In this chapter, I will explore and use tensor networks to simulate quantum systems and circuits, validate hardware through simulation, and propose offloading some tasks to classical hardware in hybrid scenarios.

Besides tensor networks, other methods to simulate quantum systems, such as decision diagrams [59, 60], are not covered in this chapter. Still, the concepts I present here also apply to these methods. I focus on tensor networks due to their versatility and their rich theoretical background from research in the quantum computing community. Further, these simulations are also suitable for the HPC domain, as they offer parallelization potential and run on high-performance clusters. Administrators can repurpose the existing hardware for such simulations, which can benefit research institutions and companies with access to HPC hardware but lacking quantum acceleration. As long as quantum computers are not widely available, these simulators are a valid alternative to test and develop quantum algorithms.

For HPCQC, simulators present a meaningful case study providing how to integrate quantum computation into existing systems. Beyond that, simulation enables hybrid workflows where parts of a problem run on quantum hardware, but suitable parts accelerate via efficient simulators. In prior work, I defined a class of problems that can be solved efficiently and implemented in a custom circuit simulator [61]. Here, I extend this work by showing how to fit this into HPC and use it for the aforementioned hybrid systems.

A common practice in software development is writing tests for the code to ensure it works as expected. In recent years, the concept of *digital twins* has taken this concept to the next level, where engineers create a digital model of a physical system to simulate and test it before starting the tedious construction process. Most simulation methods in quantum computing are an alternative digital twin, where the quantum system is simulated on classical hardware before deployment on a quantum computer. This two-phase approach provides some semblance of validation before running the quantum algorithm on the hardware. Most simulation techniques function based on one of two underlying mechanisms. One can exploit inherent properties, e.g., symmetries, to reduce the computational complexity of specific models. Or loosening the goal of *perfect* simulation in favor of approximations is good enough for the intended use case. These approximations can match reality more closely, as their variations act similarly to the noise of quantum systems. Though the noise is not necessarily the same as the noise in real quantum

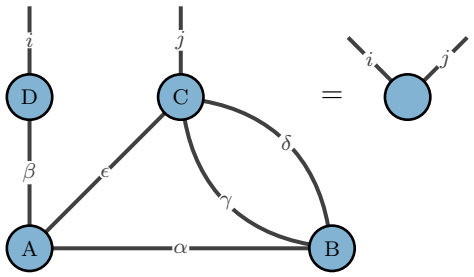


Figure 5.1: A simple tensor network diagram. It represents the following operation $\sum_{\alpha,\beta,\gamma,\delta,\epsilon} A^{\beta\zeta}_{\alpha} B^{\delta\epsilon}_{\alpha} C^{i}_{\delta\epsilon\zeta} D^{j}_{\beta}$.

hardware, the quality of the solution of simulated hardware closely compares to actual hardware. Accurately simulating hardware is an additional challenge, as the noise models can be very complex. In this chapter, I will discuss both the simulation of quantum systems and the simulation of quantum hardware.

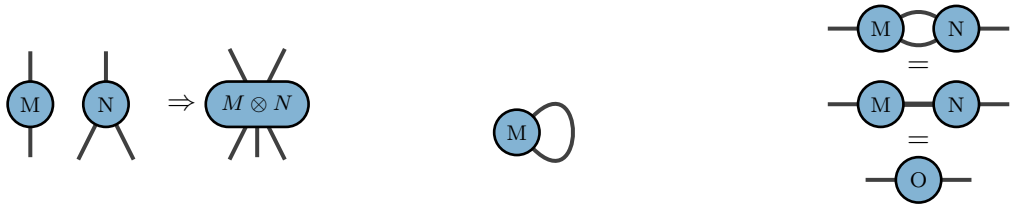
5.1 Tensor Networks

Tensor Network methods are widely successful in simulating quantum systems. Before examining the techniques in tensor networks, I want to summarize their notation. In their most simplified form, tensors are d-dimensional matrices; a vector is a 1d-tensor, a matrix a 2d-tensor, and so forth. They represent a relation between other tensors and contain complex information about their interactions. A common way to abstract instances from types is to represent tensors in a *tensor network diagram*. This graphical formalism, popularized by Penrose [62], represents typical tensor operations.

Objects, typically circles, represent a tensor, and their dimension indices by lines, so-called *legs*. Multiplying two tensors is indicated by connecting the according legs; this is then referred to as *bond dimension* while open legs are called *physical*. The shape of tensors can encode additional information, e.g., using squares for unitary tensors. Similarly, the position of the legs represents information in some contexts, mainly following the convention of putting raised indices on top. Overall, a tensor network diagram describes a computation; see fig. 5.1 as an example. All internal legs are contracted in the final tensor, resulting in a tensor with the remaining open legs.

The advantage of tensor networks is in their expressivity and flexibility. A tensor network can easily express quantum systems; the goal is to find representations with advantageous properties. The most common example is the matrix product state (MPS), which compactly represents any quantum system obeying the so-called *area law*. Before analyzing more advanced systems, I will investigate the MPS as it is the most fundamental representation.

Next to application-specific upsides, tensor networks have general advantages, making them appealing for HPC. The most important one is the ability to parallelize the com-



(a) The tensor product of two tensors.

(b) Trace over an index.

(c) Contraction of two indices.

Figure 5.2: Diagramatic representations of common tensor operations.

putation and its suitability for current hardware. *Contractions* among tensor indices is fundamentally a matrix multiplication. This operation is highly optimized on current hardware and can significantly speed up computations. As a result, tensor networks are also helpful outside of quantum computing. Nonetheless, I will focus on the simulation of quantum systems. In general, quantum systems can be simulated by their time evolution. One could first translate the time evolution operator into a quantum circuit or directly to a tensor network. Similarly, tensor network operations can express the simulation of quantum circuits. This expressiveness opens a wide range of use case scenarios. Mainly, hybrid tensor networks [63] are especially interesting, as they combine the advantages of both classical and quantum computing. In the final section of this chapter, I will discuss such hybrid systems and their potential applications in HPCQC.

5.1.1 Tensor Operations

Researchers use tensor network diagrams to reason about quantum systems. The notation has no formal standard, but the community agrees on established common symbols. Different shapes usually encode structural information about the tensor. For example, a square represents a unitary tensor, while round shapes are general tensors. A diamond represents a diagonal tensor, and a triangle is a tensor with a specific symmetry. Tensor network diagrams are mathematical formulations built from basic operations; see fig. 5.2 for the most common operations.

Computationally, the most critical operation is the contraction of the tensors (Figure 5.2c). The contraction order is crucial for the efficiency of the computation, but Unfortunately, finding the optimal order is an NP-hard problem [64]. Multiple approaches are possible to overcome this issue. One can use heuristics based on the tensor network structure to find a contraction order; usually, this disregards local effects such as data layout and computational aspects such as cache friendliness. Hence, a two-step approach is often used, first finding a global contraction order and then optimizing it for the specific hardware. A second approach is not to contract a tensor network but instead to use its structural information. For example, isometries can immediately collapse to the identity as shown in Figure 5.3.

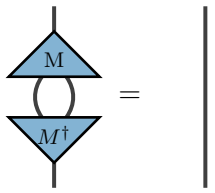


Figure 5.3: Orthonormalization property $MM^\dagger = I$ of tensors, adapted from [61].

5.1.2 Tensor Decomposition

As seen from figure Figure 5.3, the orthonormalization property of tensors plays a central role in tensor networks. Coincidentally, this property naturally arises in the decomposition of tensors. I will discuss two standard tensor decompositions, the singular value decomposition (SVD) and the QR (respectively LQ) decomposition. Conceptually, both decompositions are similar, as they decompose a tensor into a product of simpler tensors. Computationally, they have similar scaling for square matrices of size n, namely $\mathcal{O}(n^3)$, but can perform differently based on hardware.

Definition 5 (singular value decomposition (SVD), QR Decomposition) *The singular value decomposition of a matrix M is given by $M = U\Sigma V^\dagger$, where U and V are unitary matrices and Σ is a diagonal matrix. The QR decomposition of a matrix M is given by $M = QR$, where Q is a unitary matrix and R is an upper triangular matrix.*

Especially the SVD has exciting properties, which makes it worthwhile in tensor networks. The matrix Σ contains the singular values of the matrix M, which coincides with its eigenvalues for a Hermitian matrix. It can be expressed as $\Sigma = \sum_k \sigma_k \left|k\right\rangle_B \left\langle k\right|_A$ in terms of the singular values σ_k and the basis vectors $\left|k\right\rangle$. Additionally, the singular values are ordered in descending order $\sigma_1 \geq \sigma_2 \geq \cdots \geq \sigma_{\min(\dim A, \dim B)}$, which allows for truncation. Discarding the smallest $\operatorname{rank}(M) - r$ singular values provides an optimal low-rank approximation of the matrix M [65].

$$M' = U\Sigma' V^\dagger \tag{5.1}$$

$$\left\|M - M'\right\| = \min_{\operatorname{rank}(M') \leq r} \left\|M - \hat{M}\right\| . \tag{5.2}$$

Figure 5.4 represents the SVD as a tensor network operation . For higher order tensors, SVD can be extended to a *Tucker decomposition*. Computationally, it is often cheaper to reinterpret the tensor as a matrix and then apply the SVD to it. A critical component is reshuffling the tensor legs, which software tools achieve by reindexing the underlying data. The dependence on memory structure opens the room for optimizations, and due to the relevance of SVD, hardware tuning provides considerable benefits. Popular linear algebra libraries, such as `BLAS` or `cuBLAS`, are highly optimized for such operations and serve as a backbone for many tools.

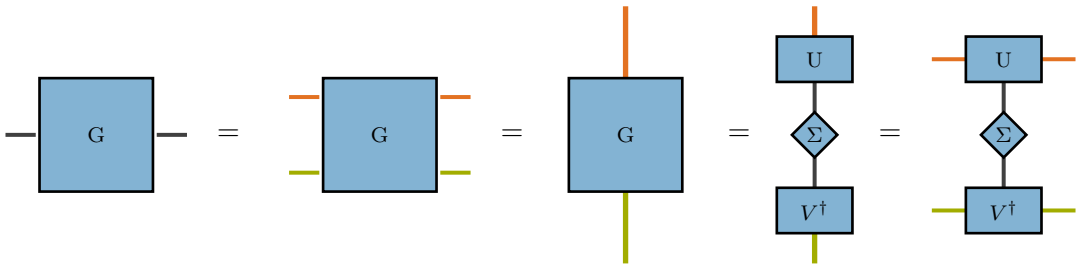


Figure 5.4: SVD of a two-qubit quantum gate. First, the 4×4 matrix is reinterpreted as $2\times2\times2\times2$ tensor. Tensor legs are then sorted and regrouped, which reinterprets the tensor as a matrix. The SVD is then applied to the matrix and the legs ungrouped and restored to the original order.

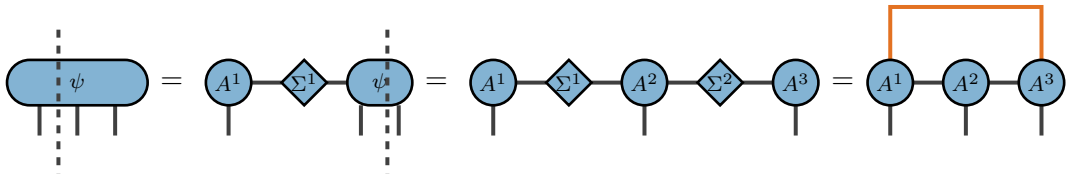


Figure 5.5: Diagrammatic presentation of MPS and its procedure, adapted from [65]. The trace operation is shown in orange, which can be inserted by an identity tensor. This is a common modification to make all tensors to have the same number of legs.

5.2 Matrix Product States

MPS are a specialized class of tensor networks representing quantum systems. Due to their simplicity, they are widely studied and used in quantum computing. A MPS represents a one-dimensional quantum system [66], where each site is represented by a 3-tensor, except for closed boundary conditions, where the first and last sites are 2-tensors. For a system with n sites and periodic boundary conditions, it can be written as

$$|\psi[A^1, A^2 \dots A^n]\rangle = \sum_{s_1,\dots,s_n} \mathrm{Tr}[A^1_{s_1} A^2_{s_2} \dots A^n_{s_n}] \, |s_1, s_2, \dots, s_n\rangle \ . \tag{5.3}$$

Figure 5.5 shows the diagrammatic representation of an MPS and its construction procedure, which I will explain next.

For creating an MPS, one can use the SVD to repeatedly decompose the system into a left and right part. Each physical dimension is singled out and split off. For a system with n sites, this results in $n-1$ SVD decompositions. One contracts the singular values into the tensors to reach the standard form of an MPS. For further simulation, some degrees of freedom (gauge freedom) offer optimization potential in some cases.

An essential property of MPS is the so-called *area law*, which holds for many physical systems. Suppose, in Equation (5.3), the number of non-zero singular values is bounded by the bond dimension χ. The entanglement entropy of the system is bounded by $S \leq \log(D)$, for a constant D such that the system uses at most $\mathcal{O}(dnD^2)$ coefficients; d is the size of the physical dimension. Hence, when only retaining all non-zero singular values, the MPS can still express a system of size d^n exactly, with a reduced size [67].

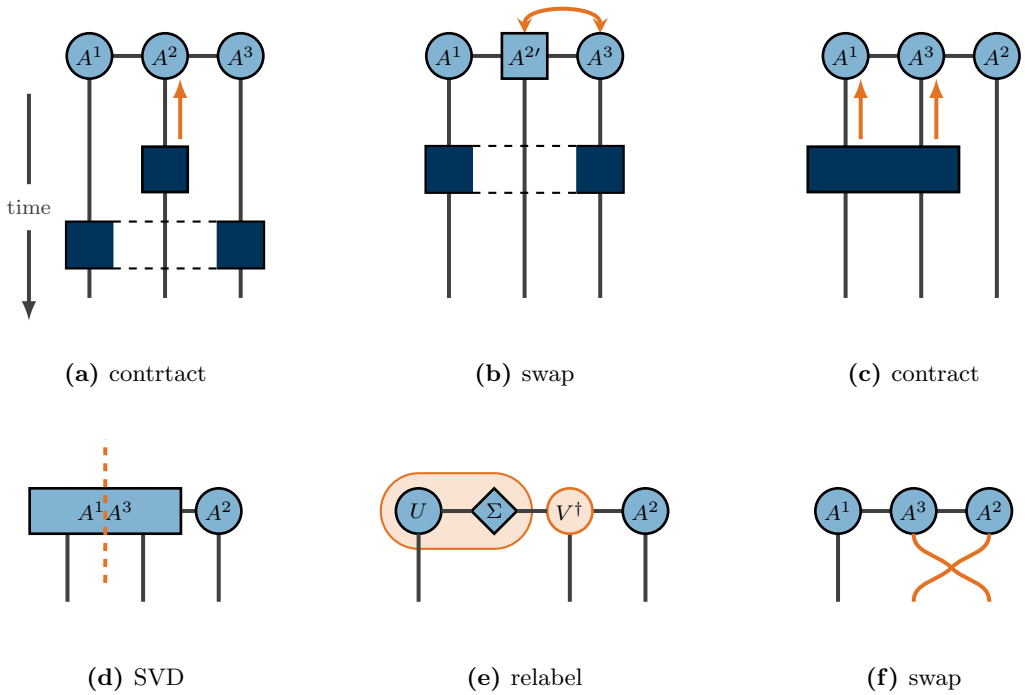
5.2.1 Simulation of Quantum Circuits

MPS can be used to simulate quantum circuits. The simulation can even be done efficiently in cases where the *area law* holds. Simulating quantum circuits requires polynomial time and space in these scenarios. Most techniques in tensor networks rely on two fundamentals: first, finding a compact representation of the system, such as the MPS, and then simulating the system step by step, maintaining the compact representation.

One can reinterpret the circuit as a tensor network for naively simulating quantum circuits with MPS. Figure 5.6 shows all steps of the simulation procedure. It first applies the gates to the system by contracting the according tensors. The application is fully parallel as long as the order of operation is respected. To ensure the MPS structure, after each gate, a SVD decomposition rebuilds the system to its original form. I use such a simulation as a baseline.

The simplicity and versatility of the MPS approach makes it an appealing choice. Contained in this simplicity are also the critical challenges of tensor network simulation. The first challenge is the contraction order, which, for MPS, can be optimized by established algorithms such as using the corner transfer matrix [69]. While I have not shown the importance of large-scale contractions, even micro-optimizations have an enormous impact due to frequent contraction usage. In most cases, simulators should avoid contractions because they are computationally expensive and impose a high memory overhead. The second challenge is the distribution of entanglement throughout the system. The area law materializes in a circuit simulation recognizable by observing swap operations. If frequent long-range operations are required, the number of swaps increases linearly with the system size. In most cases, multiple of these operations will cancel out [68], but in the worst case, many swaps remain. The problem is further exacerbated by the exponential growth of the bond dimension, making the swaps infeasible for unconstrained systems. Upfront, solving the routing and placement problem (c.f. Chapter 6) addresses this issue, which is NP-hard and, therefore, costly for linear systems. Since the area law only holds for most one-dimensional systems, other structures such as *PEPS* address the study of higher-dimensional systems.

The final component is the decomposition of the tensors. Both the SVD and the QR decomposition scale with $\mathcal{O}(n^3)$, such that simulators should avoid large matrices. However, they can also exploit the approximation properties of these decompositions to reduce the bond dimension. Zhou et al. [70] show that allowing an error budget in the simulation and mimicking the error in the physical system can reduce the bond dimension. Such a simulation runs as described above until it reaches a specific bond dimension χ. After that point, it approximates any gate increasing the bond dimension beyond



(a) contrtact **(b)** swap **(c)** contract

(d) SVD **(e)** relabel **(f)** swap

Figure 5.6: The procedure of simulating a quantum circuit with a MPS. Independent operations can be executed in parallel, for simplicity I show the operations in sequence. **(a)** shows the contraction of a single qubit gate. It can be immediately contracted with the previous tensor, a renormalization is not necessary in this case. **(b)** In order to apply a two qubit gate, the affected tensors need to be incident. Throughout the network tensors are swapped to guarantee this condition [68]. **(c)** Then the 2-qubit gate (recast as 4-tensor, see fig. 5.4) is contracted. This can be generalized to n-qubit gates. **(d)** The resulting tensor needs to be renormalized by an SVD decomposition. **(e)** The tensor resulting from the decomposition is then relabeled to maintain the MPS structure. **(f)** The procedure is repeated for each gate in the circuit. Swaps can be tracked and results permuted accordingly to safe on computation.

the limit. The decomposition in step Figure 5.6d is modified to truncate the singular values. In the case of 1D systems, this approach works well, but further techniques are required for higher dimensional systems such as the supremacy circuits [17]. Specifically, Zhou et al. [70] simulate the mentioned circuits by grouping the MPS into sub-regions. They create one larger tensor in each subregion, which combines many physical legs. Simulating these regions is relatively cheap, outweighing the cost of the more expensive operations between these operations. This approach is similar to the idea of *Tree Tensor Networks*, which I will discuss in the next section. Before that, I will discuss MPS-based algorithms for time evolution, where I further explore the tradeoffs of SVD and the QR decomposition.

5.2.2 Time Evolution

Many methods approximate a quantum system's time evolution $U(t) = e^{-tH}$. The two most common are the *Density Matrix Renormalization Group* (DMRG) and the time-evolving block decimation (TEBD) [71]. Both methods strategically apply truncated SVDs to control the approximation error. I will focus on TEBD, which applies the time evolution based on an even-odd system decomposition. The so called Suzuki-Trotter decomposition $e^{t(A+B)} = e^{tA}e^{tB} + \mathcal{O}(t^2)$ fits well with the MPS structure. The procedure is similar to the step in Figure 5.6(c-e), with a truncated SVD. During each time-step t of the simulation, a brick-wall pattern of two-qubit gates governs the system's evolution.

The QR decomposition is usually not used in the scenarios since it is associated with the identical computational cost without the approximation properties. Unfried et al. [72] show that the QR decomposition can use resources more efficiently. TEBD offers high parallelism, which specific hardware such as GPUs can exploit. Other decomposition methods do not offer the same level of performance gain[1]. Two factors usually limit performance. The implementation of the decomposition method has to be compatible with the hardware; SVD and QR decomposition are well optimized for most hardware. Additionally, most decompositions are cheaply computable based on one of these two decompositions. As a result, alternatives are usually not considered. QR-based methods offer a potential for improvement, but advanced schemes are required to exploit this potential.

5.3 Tree Tensor Networks

Tree tensor networks (TTNs) are a generalization of MPSs to a tree structure. A MPS represents a maximal unbalanced tree, limiting the system's expressiveness. TTNs also does not fully capture all entanglement in higher dimensional systems but can express more complex systems. For TTNs, as for MPS, a canonical form exists. Each node fulfills a normalization condition in this form, simplifying specific tasks. I will use this critical property in the circuit simulation in Section 5.3.1. As a result, for specific use

[1] I supervised a study on *Decomposition Methods for Tensor Networks* in student work, unfortunately with inconclusive results.

cases, TTNs can still capture systems that are out of reach for MPSs. For example, some chemical problems can efficiently be simulated by TTNs [71, 73, 74]. I will highlight a class of such problems in Section 5.3.3 along with a complexity analysis.

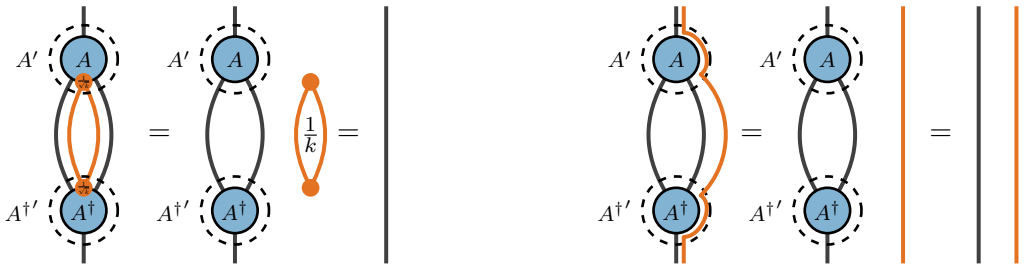
A second advantage of TTNs is the ability to encapsulate localized entanglement. This property is interesting from the high-performance perspective since this opens up the possibility of parallelizing the simulation. The tree structure straightforwardly maps to HPC cluster, where each node simulates a subtree. Further, tree structures have been studied extensively in computer science. Circuit simulation can, therefore, benefit from other well-known algorithms. My focus is still on circuit simulation due to its relevance to the *Unified Toolchain*. TTNs are highly versatile due to structural flexibility, which simulators can exploit for hybrid systems. I will briefly discuss time evolution with TTNs, for which some principles transfer from the circuit case. As a caveat, circuit simulation is very general; for time evolution, the properties of TTNs need to be utilized problem-specific.

Trees do not have an inherent structure; the only condition is that the tree is acyclic. Typically, one associates a tree with a single root node, classified by not having a parent node. Leaves, on the other hand, are nodes without children. I will use these properties during the simulation and structure search, but they are not integral. The concept of my simulator functions based on two fundamental ideas. First, the tree structure should match the entanglement structure of the circuit or system. This match means that subtrees contain the growth of the bond dimension to where computation is cheap. The simulator does not immediately compute long-range interactions between subtrees; they are stored symbolically in intermediate nodes until calculation becomes necessary. Operations are executed solely on leaf nodes to discern short and long-range interactions. Therefore, only the leaves are associated with physical indices. For TTNs, every node could support physical indices to keep the algorithm simple. Instead, I attach a leaf to every location where the simulation requires a physical operation. This restriction increases the computational cost, but only marginally, which warrants the choice.

5.3.1 Circuit Simulation

First, I explain the simulation procedure; Figure 5.9 shows a single and a two-qubit gate computation. The idea follows the general procedure below. After decomposition, single qubit gates and the remainders of multi-qubit gates can be contracted into their according to leaf node. An additional wire creates a loop for multi-qubit gates, damaging the tree structure. Therefore, I find the path between the two affected leaves, which is unique in the tree, and *route* the connection through that path. This rerouting increases the bond dimension of each of the bonds on the path multiplicative by the bond dimension of the applied gate, which is, at most, four for two-qubit gates.

Initially, the tree is in its canonical form by construction; see Section 5.3.2. After the *routing* step (Figure 5.9d), the leaves no longer fulfill the normalization condition, as in Figure 5.3. For the nodes on the path, though, it still holds, as shown by Figure 5.7. Hence, a renormalization sweep along the path is necessary. It can be delayed until some gates have accumulated if the ordering of gates allows it. For the renormalization



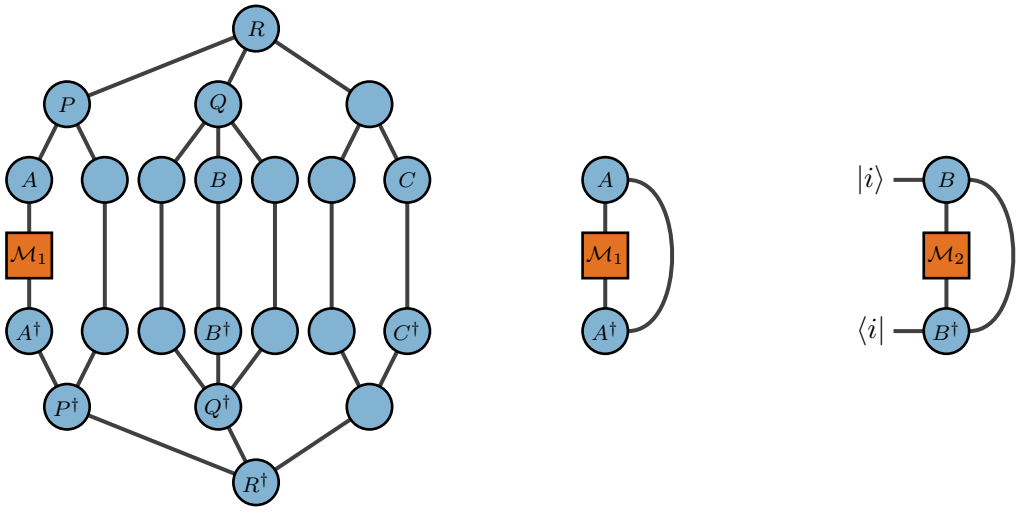
(a) Normalization of a terminating node. **(b)** Normalization of a intermediate node.

Figure 5.7: Normalization condition for nodes in a TTN.

(Figures 5.9e and 5.9f), the leaves are split using a SVD. I redefine $A' := U$, making it an isometry and contracting the remainder upward, in the example $\Sigma_A V_A^\dagger R = U_R \Sigma_R V_R^\dagger$. Here, the procedure continues by defining $R' = U_R$ and iteratively moving along the path to B. It is convenient to consider this as sweeping upward from A and B simultaneously and collecting the remainders on their common ancestor. The remainder can be stored along their respective node without an explicit contraction as long as other gates are applied. This trick saves a majority of the computational resources, which increase in the *up* direction of the tree.

During the whole procedure, the tree describes the quantum system in its current state. If it is necessary to acquire all probability amplitudes, the contraction of the tree is required. For this, optimizing the order of contraction is necessary. Through experiments, I analyzed specific predefined orderings. In general, contracting bottom-up provides the best results. Especially for balanced trees, this is competitive with state-of-the-art contraction techniques. However, hyper-optimized contraction [75] generally outperforms my procedure for unbalanced trees and cases where nodes exhibit specific memory properties.

In most cases, not all probabilities are necessary, and for simulating them, the cost of the final contraction dominates the overall simulation cost (c.f. Section 5.3.4). Measuring n-site operators or sampling from the state is more efficient as this does not require full contraction. A procedure for sampling repeatedly measures single site operators in sequence, as shown in Figure 5.8. First, a random site is measured by shifting the center of normalization toward the corresponding leaf. As a result, all other nodes are isometries, leaving only identities after contraction with their conjugate. Once the first measurement result is obtained, the site is fixed to its measurement result $|i_n\rangle$, decreasing its size. It will remain an open leg with dimension 1 for the rest of the sampling procedure, which continues with a subsequent site. The procedure builds a sequence of conditional probabilities from which one can reconstruct a complete sample.



(a) Sampling a single-site measurement operator.

(b) With the center of orthogonality at A all other nodes fulfill $NN^\dagger = I$

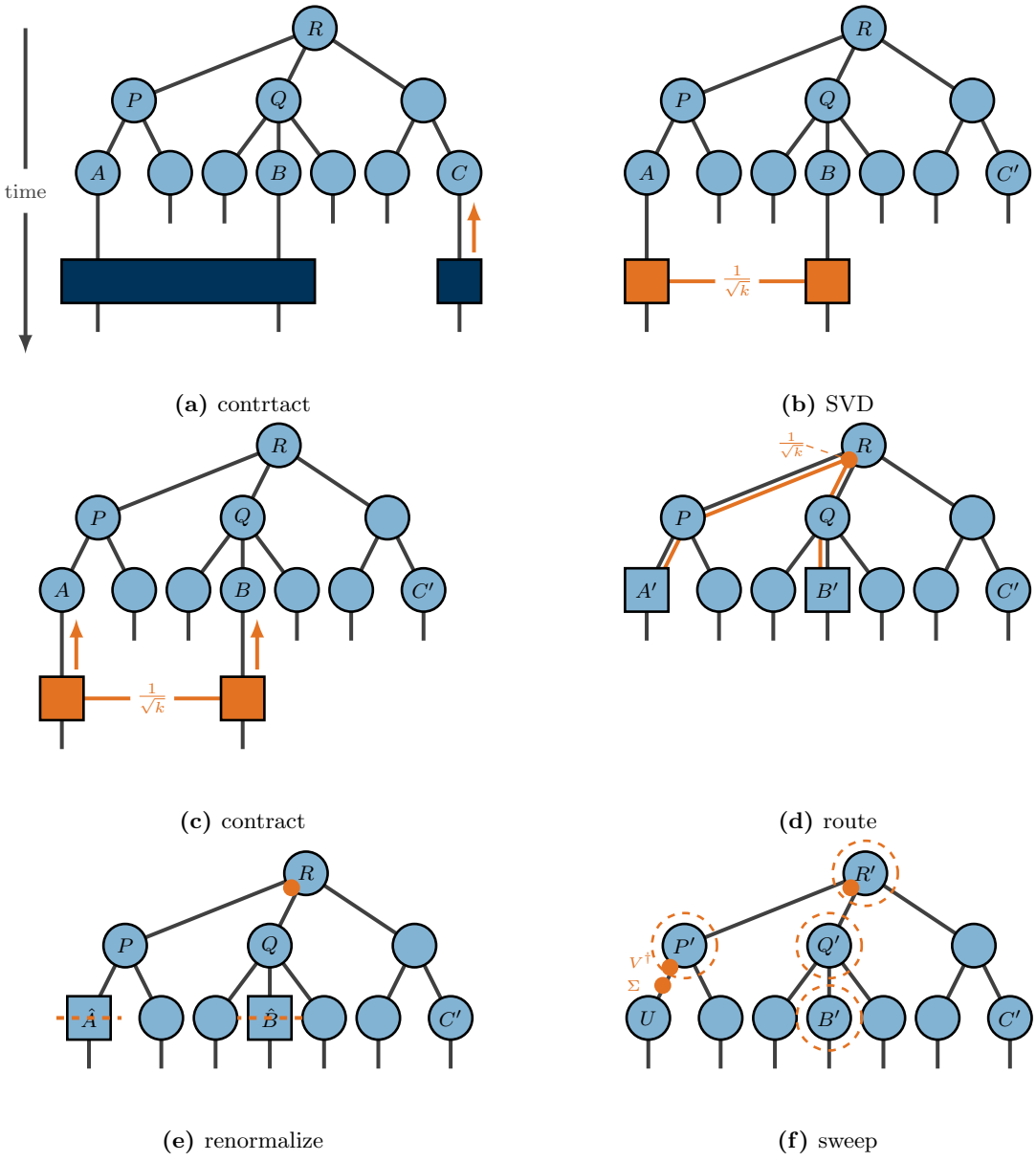
(c) Sampling multi-site observables based on conditional observables.

Figure 5.8: Sampling from repeated single site measurement in a TTN. The joint probability distribution can be constructed from conditionals $p(x_1, x_2) = p(x_1|x_2)p(x_2)$. The observables $\mathcal{M}_1$ and $\mathcal{M}_2$ can be acquired by tracing out the correct subspaces.

5.3.2 Tree Structure Search

The circuit simulation procedure clearly shows that the tree structure is vital for efficient simulation. For every multi-qubit gate, a potentially expensive renormalization sweep becomes necessary. To clarify this point, let $D_{\max}$ be the maximum bond dimension a computer can simulate. For example, assume a balanced and rooted binary tree of depth l. Any gate affecting two sites in a subtree of height $l_{\text{sim}} := \log_2(D_{\max})$ can be efficiently simulated. The number of gates spanning such subtrees is limited. A thorough analysis will follow in Section 5.3.3.

Optimizing the tree structure is a challenging task. The potential search space is permissively big; hence, most searches are restricted and should be problem-specific [74]. For example, Szalay et al. [76] variationally improve the tree structure for a given Hamiltonian. Other common restrictions include Cayley Trees [77] or binary trees [74]. I do not impose any structural restriction on the tree search but provide a heuristic instead of a truly optimal solution. Algorithm 1 shows the general algorithm. I construct the tree bottom up, similar to an agglomerative clustering. This clustering uses a similarity function, a cluster group of similar nodes under a common parent node until reaching a specific similarity limit (threshold). After that, new nodes merge beneath an increasing number of layers. A fundamental tradeoff in this procedure is the depth in contrast to the number of leaves under one node. Fewer leaves per node imply longer paths; therefore, gate applications must normalize more nodes. More leaves per node impose higher



(a) contrtact **(b)** SVD

(c) contract **(d)** route

(e) renormalize **(f)** sweep

Figure 5.9: The general gate simulation procedure. a) Single-qubit gates can be immediately contracted; b) Multi-qubit gates are split using SVD and then c) contracted. d) The additional bond is wired on the unique path between the two leafs, the normalization factor stored at their common ancestor. e) Renormalization is triggered on the leaves by splitting them with SVDs; f) The non-normal parts are swept upward.

memory requirements, increasing the computational cost of single operations. During initial experimentation, I found empirical evidence for having the larger nodes in the deeper layers while maintaining somewhat balanced trees.

The similarity function is the basis for the clustering algorithm. Due to the modular nature of my approach, users can swap the similarity function quickly and, therefore, adapt the algorithm to their use case. For circuit simulation, predicting the entanglement between two sites is straightforward. Each two-qubit gate increases the bon dimension up to a factor of four. Hence, counting and weighing the gates already provide a similarity proxy. As a final similarity function, I use

$$s_{qc}(q_i, q_j) = \sum_{d=1}^{4} d \cdot |G_d(q_i) \cap G_d(q_j)| + \frac{1}{|G_{1\ldots4}(q_i)| + |G_{1\ldots4}(q_j)|} \tag{5.4}$$

for two qubit sites q_i and q_j; G_d represents the set of gates of bond dimension d applied to the respective qubit. The sum is the primary indicator of the entanglement; as a tiebreaker, qubits with higher entanglement should be placed further down the tree. Alternatives to this approach are straightforward; any proxy to the entanglement is sufficient. For example, one could use the coordinate number [76] to calculate similarity in chemical applications. Ferrari et al. [78] use a similar approach based on weighted binary trees for quantum many-body systems. Hence, it is possible to generalize this approach beyond circuit simulation. Supporting multi-qubit gates or allowing qudit-style physical indices are feasible future improvements. Lastly, I want to address the creation of a tree state. Circuit simulation usually starts from the computational basis state $|0\rangle^{\otimes n}$, which can be generated by connecting physical indices with identity nodes to form the tree structure. The procedure is more complicated for arbitrary states but still possible [79]. In short, one can first form a MPS and then continually rebalance the tree until reaching the designated structure.

5.3.3 Computational Analysis

Before presenting the results, I will analyze the proposed procedure as a circuit simulator. I will compare to the MPS and showcase a class of problems infeasible in this regime. For simplicity, I restrict the analysis to *m-ary* trees, where each node has exactly m children. Recall $D_{\max}$, the upper limit for simulatable bond dimension. In total the tree contains m_{root}^{ℓ} qubits. The decomposition for the highest nodes on level ℓ dominates the cost, below ℓ any node has a bounded dimension of

$$D_\ell \leq 2^{m^{\ell-1}} \,. \tag{5.5}$$

Therefore it exists an $\ell_{\text{cluster}} := max_\ell\{\ell : 2^{m^{\ell-1}} \leq D_{\max}\}$ below which simulation is possible efficiently. The edges above this level are restricted in the amount gates they support. These are exactly the gates G_e spanning two subtrees of hight ℓ_{cluster} along edge e for which

$$\prod_{g \in G_e} k_g \leq D_{\max} \,, \tag{5.6}$$

Algorithm 1: Tree structure search algorithm, adapted from [61]

```
Function find_tree_structure(qc, c):
    Data: qc = Quantum Circuit, c = Number of clusters
    Result: Tree structure
    similarity ← similarity_matrix(qc.gates)                    // s_qc
    cluster_labels ← cluster(similarity, c)
                        // any clustering, e.g., SpectralClustering
    cluster_roots ← ∅
    foreach label in cluster_labels do
        cluster_roots ← cluster_roots ∪ create_subtree(qc(label))
    end
    /* Node(children) creates a new subtree and builds the tree
       bottom up.                                                */
    return Node(cluster_roots)
Function create_subtree(qubits):
    Data: qubits = Qubits
    Result: Cluster
    seen ← ∅
    children ← ∅
    pairs ← sort(Pair(qubits), most similar)
    sim ← similarity(pairs[0][0], pairs[0][1])
    // similarity is based on Eq. (5.4)
    foreach q_0, q_1 in pairs do
        if q_0 ∉ seen then
            seen ← seen ∪ {q_0}
            children ← children ∪ {q_0}
        if q_1 ∉ seen then
            seen ← seen ∪ {q_1}
            children ← children ∪ {q_1}
        if sim > similarity(q_0, q_1) then
            children ← {Node(children)}
            sim ← similarity(q_0, q_1)
    end
    return Node(children)
```

provides the upper limit, k_g, denoting the bond dimension added by the gate. Solving the above equation for the number of gates crossing an edge can be used to calculate the number of gates $\sharp A$ crossing a node A

$$|G_e| \leq \log_4 D_{\max} = \frac{1}{2}\log_2(D_{\max}) \tag{5.7}$$

$$\sharp A \leq \frac{1}{4}\log_2(D_{\max}^{m+1}) = \frac{m+1}{2}\log_2 D_{\max}\,. \tag{5.8}$$

This limit assumes the maximal cost for each gate and disregards any cases where gates can cancel or undo previous operations.

Computational Cost

The number of complex entries over all nodes dictates the space requirements of a TTN. For a perfect m-ary tree, the depth $\ell \leq \ell_{\text{cluster}}$ of a node limits the number of entries to $D_\ell^m D_{\ell+1}$ as it has m children and one parent nodes. Assuming a favorable case, i.e. for all nodes $\ell \geq \ell_{\text{cluster}}$ it holds that $D_\ell \leq D_{\max}$. Using Equation (5.5) yields $D_\ell^m D_{\ell+1} \leq 4^{m^\ell}$ for the remaining nodes. The total number of complex entries $\sharp T$ of tree T of hight ℓ_{root} is therefore bounded by

$$\begin{aligned}
\sharp T &\leq m^{\ell_{\text{root}}} 2D_1 + \sum_{\ell=1}^{\ell_{\text{root}}-1} m^{\ell_{\text{root}}-\ell} D_\ell^m D_{\ell+1} + D_{\ell_{\text{root}}}^m \\
&\leq 4m^{\ell_{\text{root}}} + \sum_{\ell=1}^{\ell_{\text{cluster}}-1} m^{\ell_{\text{root}}-\ell} 4^{m^\ell} + \sum_{\ell=\ell_{\text{cluster}}}^{\ell_{\text{root}}-1} m^{\ell_{\text{root}}-\ell} D_{\max}^{m+1} + D_{\max}^m \\
&\leq \sharp(nodes) D_{\max}^{m+1} \leq \sum_{\ell=0}^{\ell_{\text{root}}} m^{\ell_{\text{root}}-\ell} D_{\max}^{m+1} = \frac{m^{\ell_{\text{root}}=1}-1}{m-1} D_{\max}^{m+1} = \frac{mN}{m-1} D_{\max}^{m+1}
\end{aligned}$$

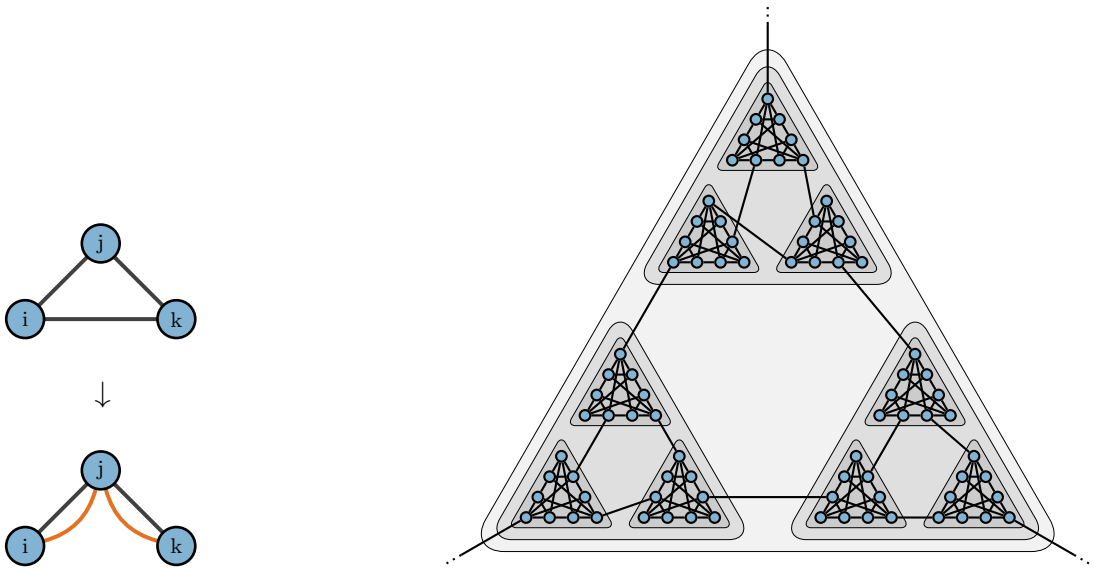
for a system with N qubits. Consequently, for fixed m and $D_{\max}$ the system size scales $\sim \mathcal{O}(N)$. [61]

The normalization procedure bounds the computational cost and the necessary SVD. For a matrix $M^{p\times q}$ the SVD scales with $\mathcal{O}(\min(p^2q, pq^2))$. A gate connecting two sites along the maximal path in the tree visits $2\ell_{\text{root}}$ nodes. Thus, the cost of a single gate scales as

$$\sharp(FLOPS) \leq \mathcal{O}(\ell_{\text{root}} D_{\max}^{m+2}) = \mathcal{O}(\log_m(N) D_{\max}^{m+2})\,. \tag{5.9}$$

Application Scenario

For systems not submitting to the area law, the bond dimension in a MPS will eventually diverge for $N \to \infty$. Any gate connecting site i and k will also affect all $i < j < k$. Some of these cases can still simulated with a TTN though. In these cases, Equation (5.7) holds recursively for all subtrees above ℓ_{cluster}, whereas at least one connection would include a case $I < j < k$ in a MPS. Figure 5.10 gives an example of such a scenario. Each triangle represents a cluster connected by a limited amount of two-qubit gates,



(a) Pairwise connections in the MPS

(b) $m = 3$, $D_{\max} = 64$

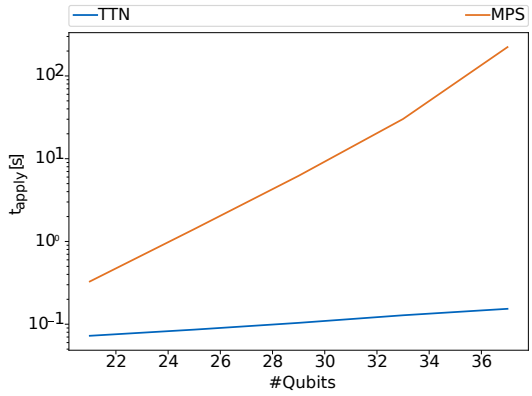
Figure 5.10: Example scenario for which a TTN simulation is possible, while the bond dimension for a MPS diverges. For each triangle (cluster), only three gates cross their boundary, therefore a TTN can still handle them. A MPS-based simulation would result in one node playing the role of j through which an additional bond is routed, exceeding the limit of $D_{\max}$. Figure adapted from [61].

as indicated by the lines between them. Arbitrary connectivity is possible inside the clusters without infringing $D_{\max}$. Simulating the same problem with a MPS will always yield a node j through which an additional bond is routed (shown in Figure 5.10a).

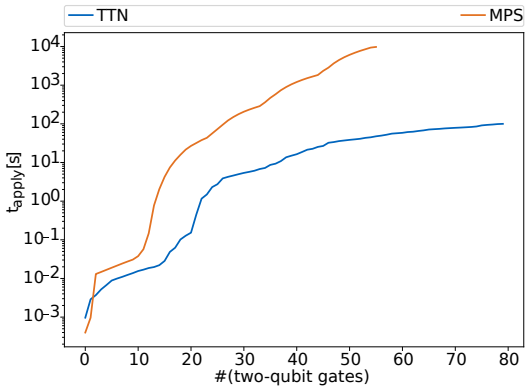
5.3.4 Simulation Results

As a feasibility study, I simulate different circuits on AMD Ryzen 7 3700 with 32GB of RAM (and 256GB SWAP), taken from the previous work [61]. Both decomposition methods, QR and SVD, produce similar results; therefore, the results focus on the SVD. I compare against a MPS simulator as a baseline on two circuit categories with varying sizes; I provide the circuits in Appendix A. A scaled-down version of the cross-entropy circuit from Arute et al. [17] referred to as *lattice* circuit illustrates the worst-case behavior. These circuits produce a highly entangled state across all $n \times n$ sites in two dimensions, beyond the expressiveness of TTNs. The *tree-like* circuits represent the best-case scenario, as they fit nicely to the tree structure; this means the condition on $D_{\max}$ according to Equation (5.5) is always satisfied. Such a circuit could result from a variational ansatz, e.g., in VQE, or from encoding a quantum chemistry Hamiltonian to a circuit.

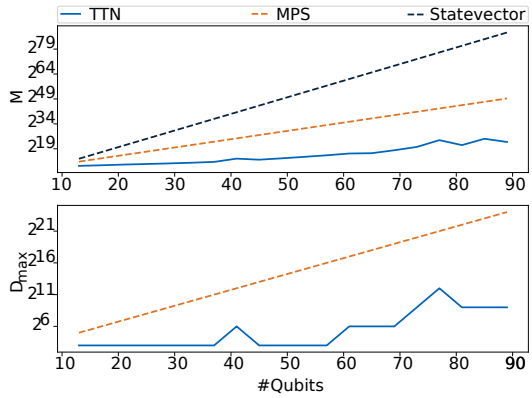
During the initial tree structure search, I imposed no restrictions. One emerging insight is the advantage of larger clusters corresponding to shallow subtrees. The exper-



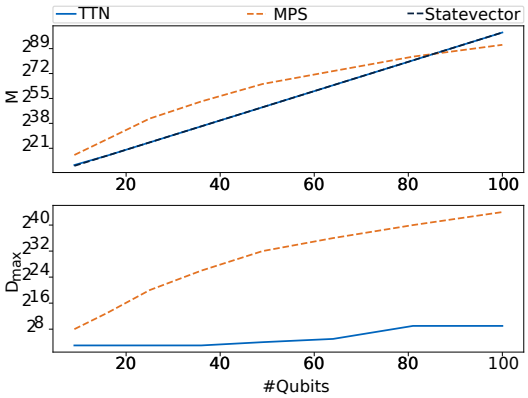
(a) Wall-clock time of applying gates and re-orthonormalization, comparing a MPS with a TTN representation of the quantum state for the *tree-like* circuit.



(b) Accumulated wall-clock time of applying gates and re-orthonormalization, on the *lattice* circuit (for 25 qubits). For gate 57 the MPS fails to normalize.



(c) Results of the *tree-like* circuit *dry-runs* compared to the MPS. The non-monotonicity is caused by the heuristics in the tree creation. In some cases, the algorithm fails to find a good clustering, namely for *41* and *77* qubits.



(d) Results of the *lattice* circuit *dry-runs* compared to the MPS.

Figure 5.11: Simulation results for *tree-like* and *lattice* circuits.

iments use the best structure found during the search, biased toward shallowness. I use predefined structures shown in Appendix A.

As a target metric, I use wall-clock timing, which is the overall time it takes to simulate a circuit. The initial trial of a 37-qubit *tree-like* circuit is depicted in Figure 5.11a, showing the duration of applying gates during the simulation. It confirms the hypothesis in Section 5.3.3, showing a clear advantage of the TTN over the MPS. This result is expected behavior, as the structure requires unfavorable gates for the MPS, forcing it to swap opposite ends toward each other. Surprisingly, on a 5×5 *lattice* circuit, the TTN still outperforms the MPS, although less extreme. A likely reason for this is the logarithmic path length inside a tree. Due to the shorter paths, fewer decompositions are necessary during the normalization. A side effect of this is that the underlying *BLAS* library fails to converge during the SVD (QR) subroutine.

Beyond feasible simulations, I provide a scaling estimate using *dry-runs*. Instead of contracting and normalizing tensors, I track the maximal internal bond dimension $D_{\max} = \max_{T \in \mathrm{TN}} (\max_i \dim_i(T))$ and a proxy to the memory overhead $M = \sum_{T \in \mathrm{TN}} \prod_i \dim_i(T)$; $\mathrm{TN} = \{\mathrm{MPS}, \mathrm{TTN}\}$. I show the results for up to 100 qubits in Figure 5.11c for the *tree-like* circuits and Figure 5.11d for the *lattice* circuits. The trend evident in the previous simulation extends to this study. The performance gain is significant when suitable tree structures are available (as for the *tree-like* circuits). This result is significant because these experiments do not use hand-tuned tree structures. The results on the *lattice* circuits are less extreme but still favorable for the TTN in most cases.

Using the *dry-runs*, I reason that TTNs is a valuable tool in HPCQC. First, *tree-like* circuits can still be challenging for quantum hardware, making them an ideal candidate for validation. Second, using the inherent structure of quantum circuits is a reasonable approach. Compilers can use this knowledge to favor such emergent structures and scheduling to map existing structures to optimal hardware. Both of these ideas, I will explore in the upcoming chapters.

5.4 Hardware Simulation

So far, I have only discussed how to simulate circuits, but not the utility the simulation provides There are two situations for which this is suitable. The obvious use case is as an additional accelerator, working in tandem with quantum hardware. Before I study these cases, I want to highlight the role of simulators in validating the toolchain. Maintaining quantum hardware is tedious and expensive. The incremental software development process, in which engineers evaluate their progress regularly, contradicts this restriction. Keeping up quantum hardware only to evaluate these increments can be wasteful. A viable alternative is to simulate feasible problems and extrapolate to larger sizes.

For this, I have to extend the simulator in two ways. First, I will explain how to transform this into a HPC compatible system. From this, I can draw conclusions and requirements for the `Unified Toolchain`. The software I develop throughout the remaining chapters can also be validated using this simulator. Although, some problems require an artificial increase in complexity to emulate real hardware.

5.4.1 High Performance Simulator

The apparent change is to switch to a high-performance language. Additionally, some subtle changes are possible. Micro optimizations, e.g., memory layout, offer multiple venues, but this is not my focus. Instead, I will consider the two primary performance criteria, multiprocessing and multi-node computing.

Multiprocessing is straightforward in the gate application procedure. The two necessary operations, contraction and decomposition, can be considered the base units. Each process should handle at least one of these units. By default, cores can handle gates affecting two sites independently; hence, two processes are necessary per gate. Gates are also data-parallel unless they affect common sites. Therefore, the simulator can check each gate to see if a core processes an interfering gate. If not, it can dispatch it to an independent process. If a site is affected by multiple consecutive gates, additional options are possible. The simplest one is to treat two (n) two-qubit gates as one three ($2n-1$) qubit gate. In the case of $2n-1=N$, the gate affects the entire tree, which the simulator must normalize, although such gates indicate scenarios where the efficient simulation regime is impossible. The normalization is fully parallel from a HPC perspective. I employ the parallelization scheme depicted in Figure 5.12, which I explored in student work with inconclusive results.

The scheme works by balancing the load between cores and distributing subtrees over multiple compute nodes. It considers only the case where $D_{\max}$ is guaranteed as the upper bond dimension. This assumption means the bulk of the computation occurs in subtrees of height ℓ_{cluster}. Each subtree of height $\ell_{\text{cluster}}-1$ is assigned one core, depicted in orange in Figure 5.12. Above that (in green), additional cores are reserved for nodes at ℓ_{cluster}. Due to the node size, the computations on this level are the most time-consuming while still frequent. Higher level nodes are of similar size but to adhere to the $D_{\max}$ limit, occur infrequently, such that a single core can handle them (blue). One additional node can be reserved for communication between nodes (ivory), which also can be handled by the prior node.

I also use the HPC simulator to gain insight into the requirements of the toolchain. The hardware interface is the first observation. Similar to QC, the simulator is gate-based, and the results are sampling-based when not explicitly contracting to the state vector. Hence, I can define hardware's in- and output interfaces; a sequence of gates is universal, and the desired accuracy and tractability ($D_{\max}$) are simulator-specific configurations. Other capabilities, such as finding the entire state vector, allowing higher-level systems (qudits), or multi-qudit gates, can be exposed to the system to optimize the gate sequence. A reasonable approach is to split functionality into two: a standard set of capabilities every system should have and unique, optional features. Device characteristics must be considered during compilation and scheduling to ensure the system performs at its best. For example, a compiler aware of the subtrees mapped to a compute node should try to contain most interactions within these regions. The *mapping* phase during compilation for quantum hardware can include this as an additional restriction. The tractability of a simulation is attractive to the scheduler as it can determine if a (sub) problem is suitable for simulation. Given a potential hardware, the scheduler

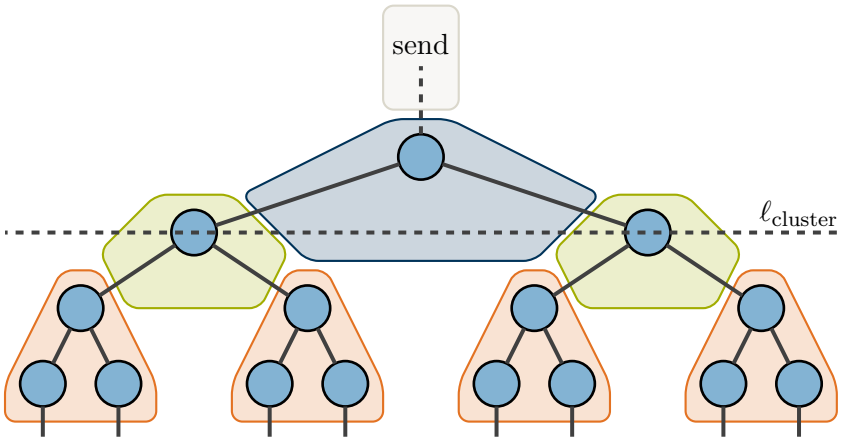


Figure 5.12: Mapping a binary tree with $D_{\max} = 8$ to an equal number of processes. A subtree of height $\ell_{\text{cluster}} - 1$ is mapped to a physical core, shown in orange. Nodes at level ℓ_{cluster} are individually assigned to cores. Above a single core manages any upper level operations. The final core is used to synchronize operations across multiple nodes.

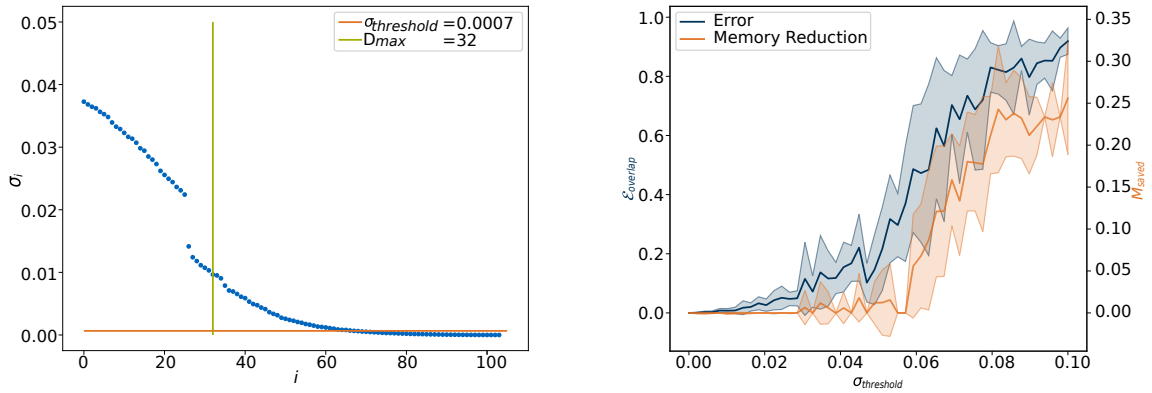
can calculate the expected characteristics of a circuit beforehand. For this, a concise query mechanism must be established, presenting the *figures of merit and constraints*. Combined, an abstract hardware interface is possible. It enables hardware-agnostic and hardware-aware optimization, as well as scheduling.

Tight integration is necessary for hybrid applications (see Section 5.5). Next to the hardware requirements, this also means that quantum hardware should behave similarly to the default. Infrastructure for monitoring is of utmost importance due to the fragility and susceptibility of noise of quantum hardware. The classical load has to be carefully tuned to match the quantum system. If the system is idle during simulation, it could bear parts of a computation via a simulator. Load balancing can ensure high utilization and increased performance. One of the milestones for this is to have QPUs accessible similar to GPUs while keeping the hardware-specific advantages.

5.4.2 Error Models

Simulating erroneous hardware is a challenging task. The first idea is to use the properties of the SVD to allow an error budget during simulation [70]. This argument works in the MPS simulation due to tying one gate to one decomposition. Extrapolating the errors of one gate (*gate fidelity*) to an overall simulation fidelity is sound. In contrast, multiple SVDs are necessary during one normalization sweep. The same argument, therefore, does not extend toward TTN.

Nonetheless, I still explore the possibility of truncating singular values during simulation. The idea is to fix $D_{\max}$ during the simulation, impacting all levels $\ell > \ell_{\text{cluster}}$. One can identify all nodes above this level before the simulation. For these nodes, singu-



(a) Example SVD truncation.

(b) Error to memory tradeoff on randomized 16 qubit circuits.

Figure 5.13: Results of a truncated simulation. The relative saved memory is $\propto M_{\text{saved}} = 1 - \frac{M_{\text{truncated}}}{\text{perfect}}$. The error to memory tradeoff is not justified.

lar values $\sigma_1, ...\sigma_{D_{\max}}, \sigma_{D_{\max}+1}, ..., \sigma_n$ are truncated below $\sigma_{D_{\max}}$. This truncation limits the growth of the bond dimension and also saves computing overhead to $\mathcal{O}((4D_{\max})^3)$ per node. Unfortunately, this approximation scheme is not sufficient for higher truncation values. The results of my initial investigation are displayed in Figure 5.13 An accumulated cutoff of 0.1 results in a simulation error of 25 percent in state overlap $\mathcal{E} = 1 - |\langle\Psi_{\text{exact}}|\Psi_{\text{approx}}\rangle|^2$. Enforcing $D_{\max} := 32$ would require significantly more truncation, making this truncation scheme unreasonable. As every iteration sweep requires multiple SVDs, the truncation error accumulates too quickly. This result does not imply the absence of a good scheme for TTNs; potentially, more advanced techniques can be successful.

A general problem of the truncation approach is the limited correlation of simulation results with real hardware. The overall amount of error might be the same, but the sources could be completely different. Defining an accurate noise model of a single chip cannot be compared with a general error value. For example, such a simple model does not model correlated errors or physical effects like decoherence.

5.5 Hybrid Systems

Due to the limited size of current quantum hardware, many algorithms solve problems in a hybrid fashion. Recently, Schuhmacher et al. [63] proposed hybrid tree tensor networks falling into the above category. For HPCQC, hybrid TTNs offers a natural use case. In these hybrid networks, nodes exceeding $D_{\max}$ could be stored as a quantum state. A scheme could simulate short-range interactions on classical hardware and store long-range interactions in the quantum hardware. This distribution allows scaling beyond the limits of both individual systems. Although the hardware is not yet sufficient to support such a scenario, researchers can use it to define the future requirements of the

software stack. In such heterogeneous systems, communication schemes become incredibly challenging. Additionally, the quantum hardware has to be scheduled simultaneously with classical resources.

A secondary challenge is identifying program parts that lend themselves to classical simulation. Similarly, I showed that specific circuits can be efficiently simulated by TTNs, and similar results also exist for other simulation techniques [80]. Circuit cutting [81], which I will be discussing in the context of compilation and scheduling, allows a distribution among HPC and quantum resources. Loading off parts of a computation without exponential overhead allows for higher utilization. As a side effect, in specific circumstances, error mitigation can be achieved by such hybrid approaches [11].

To conclude, the benefit of simulation is apparent. First, it allows system software validation without access to quantum hardware. Interfaces to the software stack can be defined based on this. The second upside is the use of hybrid systems. Utilizing available resources is one of the primary goals of HPCQC, which profits from high-performance simulators.

6 Multi-target Compilation

Quantum circuit compilation transforms a high-level quantum algorithm into a sequence of quantum operations available on a quantum computer. As outlined in Chapter 3, most existing quantum platforms include compilation techniques as part of their offering. The typical framing of the compilation process is simple. Given a quantum algorithm in circuit form, optimize the circuit such that it optimally fulfills all of its restrictions on a given target hardware. Optimal can mean different things, but most commonly, it means minimizing the number of gates or noise in the circuit. The optimization is necessary because quantum hardware is still limited in its capabilities, especially in the coherence time of qubits.

The systems I describe above have two underlying assumptions. First, the quantum circuit is available in advance, and second, the user fixes the target hardware to a specific device. From the *Unified Toolchain* perspective, the compilation process for hybrid systems works differently. When users formulate a hybrid algorithm, they only know parts of the quantum circuit if the rest depends on a complex classical problem. For example, the structure of a VQE could depend on an NP-Hard problem that HPC system computes classically. Concretely, it could run a low-fidelity molecular-dynamics simulation computing the starting positions of particles, which the simulation then refines using a dependant VQE. Similarly, it could rely on a partial kernel, which depends on some unknown success criterion. If multiple quantum computers are available, optimizations must consider all options. Due to the computational cost of some optimization techniques, this is not always feasible. Therefore, the *Unified Toolchain* follows a two-step approach. The quantum circuit is optimized for a generic quantum computer during compile time. Targeting a generic quantum computer restricts the set of possible optimizations. However, time restrictions do not apply here, enabling more complex optimization techniques. The scheduler will assign a specific quantum computer to the circuit at runtime. While waiting in the queue, the circuit can be generated and optimized for the selected backend. The optimization incorporates hardware restrictions such as limited connectivity or available gate operations. This optimization could happen when a complex HPC task runs simultaneously. Waiting for the compiler would be extremely expensive and should be avoided.

I will focus on two parts of the compilation process in this chapter. First, I will explain how to adapt the compilation process to the *Unified Toolchain*. The main distinction is the separation of offline and online compilation. During the latter, the system generates quantum circuits on the fly. I will also introduce the concept of *Circuit Cutting* [81], a technique to split a quantum circuit into multiple parts. This technique will be necessary for the scheduler and enable the execution of multiple parallel circuits. Second, I provide additional tools for the compilation process. These tools are designed explicitly

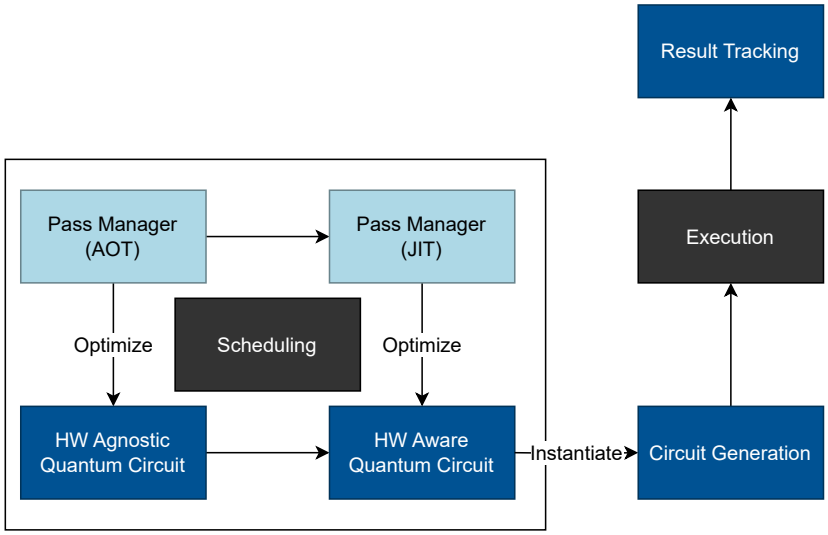


Figure 6.1: The two-stage compilation process in the *Unified Toolchain*.

for the two-step approach. I introduce a machine learning-based approach to shifting the optimization cost to the training phase. Therefore, the optimization is reduced to an inference task of the trained model, saving time during the runtime phase. I also consider scenarios in which *heterogeneous* quantum computers are available, enabling simultaneous compilation toward multiple backends. The second part follows my unpublished work.

6.1 Compilation in HPCQC

As outlined in Chapter 4, the compilation process in the *Unified Toolchain* is split into two parts. Figure 6.1 shows a simplified two-stage compilation process which occurs in the *Unified Toolchain*. This split stands in contrast to the traditional compilation process. Still, some of the practices apply to quantum computing. First, consider the split between offline and online compilation. I adhere to the compilation steps listed in Section 3.2. For the third step, a compiler already requires device information. This dependency is problematic, as much of this process should happen at compile time to prevent stalling at runtime. The concept of just-in-time compilation works under similar constraints. Therefore, I will draw inspiration from the JIT compilation process and adapt it to the quantum computing domain.

6.1.1 Offline Compilation

Offline quantum compilation corresponds to the traditional compilation process. The significant difference is the techniques used to optimize quantum algorithms. Standard algorithms do not formulate the problem in terms of quantum circuits. Instead,

a problem Hamiltonian describes the desired systems. This Hamiltonian is typically transformed to match a qubit description and then decomposed into a unitary matrix. Various techniques for this process exist and typically are not considered part of the compilation process. In Section 6.2.2, we will investigate how practical it is to include this step in a compiler. From the unitary matrix, one can then synthesize a quantum circuit.

Most existing quantum compilers, such as `tKet` or `Qiskit`, use predefined optimization techniques. Users choose a target hardware and an optimization level, and the compiler will respect the hardware limitations to provide an optimized circuit. A unitary matrix represents a single gate acting on all system qubits. The first step during compilation, *synthesis*, is to decompose the unitary matrix into a sequence of gates. Afterward, the circuit is optimized and transpiled to match the hardware restrictions. During compile time, the compiler has an arbitrary amount of time to optimize the circuit; hence, expensive techniques like brute force search or multiple iterations are feasible. Still, it is worth considering the overall runtime of each compilation step. Compared to the runtime of quantum systems, spending more time to compile and optimize a circuit is not reasonable. Therefore, the offline compilation process should be as efficient as possible. A second benefit arises when reusing techniques during runtime, as they must be as fast as possible.

There are two possible ways to tackle this challenge. First, one can make the toolchain as efficient as possible from a software standpoint. Leveraging `LLVM` and the associated practices, e.g., `MLIR`, is a reasonable approach. It also is finding broader adoption in tools like `cuda-q` [45] and `Catalyst` [47]. The concept behind this is straightforward and aligns well with the goals of the *Unified Toolchain*. A compiler extracts the quantum-related parts from the original program and translates them into an intermediate representation. This representation is typically chosen based on the optimization techniques one wants to apply. Optimizations happen in a predefined sequence; the compiler can choose which to apply. The steps are referred to as *passes* and represent a single optimization step, e.g., gate merging or gate cancellation. When traversing the pass sequence, conceptually, each step moves the circuit closer to the desired form; this process is called *lowering*. On the lowest level, the circuit is represented in a format close to the hardware level, and the runtime can then generate the circuit dynamically. The QIR [49] is an emerging standard for this purpose.

QIR defines a minimal set of operations that a quantum controller can implement through an instruction set [82]. Therefore, it is an abstraction layer which can server as a common hardware interface. In the context of the *Unified Toolchain*, compilers can target this interface to interact with a generic runtime environment. Since the optimization consists of passes, reusing parts of the compilation process is straightforward. The offline and online compilations have access to the same set of passes. Hence, the compilers can reuse existing techniques and share state information. This capability can be beneficial for optimization over multiple runs, so the *meta optimizer* can fine-tune the optimization process. Tight integration is possible using task and QPU specific passes; the coupling is loose, as a compilation pass can encapsulate arbitrary techniques. Due to the `LLVM`

infrastructure, it is possible to completely swap out QIR, as long as suitable lowering passes to other representations exist.

Parametrized Compilation [42] is a technique reducing the number of necessary compilations. In essence, the compiler sets placeholder values for parameters and compiles a circuit without using the actual values. While operating, the runtime will substitute the placeholders with the current values. This substitution is beneficial for hybrid algorithms, where the circuit structure never changes, but the parameters do, for example, in the VQE algorithm. The compiler can optimize upfront toward using this technique; only micro-optimizations are necessary at runtime. Similar approaches are promising for the *Unified Toolchain* and are necessary to reduce the runtime overhead of the compilation process. In section 6.1.3, I will revisit this concept in the context of *Circuit Cutting*.

6.1.2 Online Compilation

The online compilation focuses on generating the quantum circuit at runtime. Similar to JIT compilation, the goal is to use as many precomputed parts as possible to reduce the runtime overhead. This step is necessary due to the hybrid nature of the *Unified Toolchain*, where the runtime instantiates the quantum circuit online. The online compilation process consists of two steps. As the hardware information becomes available, the compiler can generate a circuit optimized for the specific hardware. The second step is the generation of the circuit; contained in this term are multiple related techniques. In kernelized approaches, the circuit generator combines all the optimized parts to form the final circuit. When using *Circuit Cutting*, the generator splits circuits into multiple independent parts, which are then treated separately. Lastly, a constructor instantiates preassembled parametrized circuits with the current parameters. A final optimization step is necessary to ensure the circuit still fulfills the hardware restrictions.

Compared to hardware-agnostic optimizations, hardware-specific optimization yields better results by including native operations. The first example is *mapping*, which selects the best physical qubits to use for an operation. To efficiently map qubits, the compiler needs accurate information about the current state of the hardware. In the *Unified Toolchain*, this information is available from two sources, channeled through the *runtime orchestration*. Live calibration data can influence mapping, depending on its accuracy and age. Newer data typically has higher confidence, but problems might still be undetected. The second source comes from the *meta optimization*, which tracks performance and provides monitoring capabilities. Over multiple executions, the meta optimizer can build an accurate noise model by tracking metrics. During scheduled downtimes, specific tomography tasks [83] can increase the accuracy of the noise model.

Optimization becomes necessary due to the noise in the current hardware. Therefore, the choice of hardware is crucial for the algorithm's success. Choosing the proper hardware is the scheduler's task (see Chapter 7), but the compiler deals with the consequences. The compiler must be able to deal with hardware-specific operations; therefore, the hardware must supply the necessary information. In the `MQSS`, the *Figure of Merit and Constraints* describes the hardware capabilities. Especially in trapped ion and

neutral atom systems, leveraging the hardware potential can lead to significant improvements. In contrast to superconducting qubits, these systems can use operations like shuttling qubits or multi-qubit operations. Before the computation starts, Shuttling can set up an optimal layout matching the circuit requirements. Stade et al. [84] showed that this can significantly improve the routing overhead. Other techniques, like mid-circuit measurements [85], can enable qubit reuse techniques to simplify computations [86].

6.1.3 Circuit Cutting

A common problem in NISQ devices is the limited amount of available qubits. Combined with the limitations of the cloud infrastructure, namely blocking devices during use, this leads to underutilization of the hardware. Large circuits compete for the same resources and cannot run at all, depending on the size. In classical computing, this problem vanished when parallelizing the computation. Distributed quantum computing is a possible scenario, but infeasible in the short term. An alternative approach is to split the circuit into smaller parts, which can operate separately. This approach is called *circuit cutting* [81] and is part of the *circuit generation* in the *Unified Toolchain*. Its main advantage is running circuits that would otherwise not fit on the hardware.

Informally, circuit cutting partitions a quantum circuit by trading sample overhead for circuit size. There exist two variants of circuit cutting:

- *Wire cutting* [87–92]: The circuit is split along the qubit lines.
- *Gate cutting* [81, 93–99]: The circuit is split along the gate lines.

Wire cutting conceptually saves the computation by sampling specific operators and using them later to reconstruct a new starting state. In practice, this is not feasible for many quantum backends and is limited due to the execution dependencies. *Gate cutting* is more practical; One can choose the size of the subcircuits to fit the hardware, and the base operations are simple one-qubit gates. As I show later, the execution order can be independent based on the cutting technique, making this a valuable tool for scheduling. Using circuit cutting as a generator enables two essential features. First, multiple circuits can execute in parallel, and second, the scheduler can choose the hardware for each part independently. The second point is practical when leveraging the device-specific capabilities according to the subcircuits.

Circuit cutting formally describes a computation $\mathcal{W}$ (gates or wires) by its quantum channel $\mathcal{W}$ and its decomposition into L local operations $\mathcal{F}_i$ according to

$$\mathcal{W} = \sum_{i=1}^{L} a_i \mathcal{F}_i \,. \tag{6.1}$$

For *gate cutting*, one decomposes a multi-qubit gate into local operations, including measurements. Three conditions are necessary to reduce the circuit size, which *gate cutting* can meet under the right circumstances. First, quantum computers must be able to implement local operations. Hence, only gates, measurements, and resets are allowed.

Second, the resulting circuit must be fully independent; such partitioning is possible through multiple cuts. To allow for arbitrary execution order, the sub-circuits must also faithfully represent the original circuit by only applying pre- and post-processing. Quantum or classical communication between the sub-circuits during execution prevents this.

The reconstruction of the original probability distribution is possible through a technique called quasi-probability sampling [100]. In the context of circuit cutting, the sampling of the channels $\mathcal{F}_i$ is weighted proportional to the coefficients $|a_i|$; this is possible a priori. The sign of a_i also affects the sign of the expectation values of measured observables. Circuit cutting is a tool for trading circuit size for the sampling overhead. The overhead κ^2 results from the reconstruction, which depends on the number L of sub-circuits and the number of samples κ. It depends on the decomposition coefficients

$$\kappa = \sum_{i=1}^{L} |a_i| \,. \tag{6.2}$$

Both values, L and κ, depend on the number and type of gates cut in the original circuit. For example, cutting a single CX gate into two single qubit gates results in $L = 6$ and $\kappa = 3$. Cutting multiple gates then scales exponentially with these values unless parallel cutting [91] is possible.

A compiler can optimize for circuit cutting in various ways. Reducing the number of gates to cut is a (bi) partitioning problem. During the mapping phase, the compiler can consider potential cuts, placing gates to minimize the number of cuts. Bandic et al. [101] apply a similar idea to the case of distributed quantum computing. Similarly, I adapt their QUBO formulation to account for the cutting overhead[1]. The second optimization venue is to reduce κ by optimizing toward other gates/ Cutting Z-rotation gates is cheaper than cutting CX gates [97]. Still, most synthesis techniques target the CX gate, as it's well-understood and optimized. Changing the compiler to use different gates can reduce the overhead of cutting[2]. The third optimization is related to reusing compilation for similar circuits. Each partition consists of L similar fragments that only differ in the local operations replacing the cut gates. Hence, most optimization stays the same; only the local operations undergo optimization twice. Therefore, the compiler can reuse the optimization results from the first partition for the subsequent ones. This concept fits well with the *Unified Toolchain* but requires the generator to always cut a circuit in the identical location. Combining all these techniques in one compiler already improves the overall utilization of the quantum hardware. However, the conventional compilation targets still limit this; therefore, I introduce a novel approach in Section 6.2.

6.1.4 Meta Optimization

Both offline and online compilation work based on selecting optimization techniques in the form of passes. Current systems, like `Qiskit`, either define a strict order, or the

[1] Implementing cutting aware mapping is part of supervised student work.
[2] Compiling for circuit cutting efficient gates is the goal of another student works.

user has to choose it. Defining this is only possible by imposing the notion of lowering the selection. In the *Unified Toolchain*, the compiler should choose the order of passes based on the system's current state. Choosing an optimal order of passes is only possible with a few passes. The crucial step is choosing the direction of lowering; if not enforcing distinct levels, the optimization might loop indefinitely between two passes.

In the *Unified Toolchain*, the *meta optimizer* is responsible for providing additional information to the compiler. For example, it can track the benefits of a specific pass given a problem algorithm. Quantifying the exact benefits is difficult, but one can compare a cost-to-improvement ratio. Such information can be transferable in the case of circuits belonging to the same algorithm class. To implement meta-optimization, one must define the action space and environment of the optimizer. This task means the passes should follow a strict interface, including performance on given metrics, runtime, and memory consumption. Defining clear stages with a defined input and output allows for reducing the size of the search space. Next to the problem-specific information, the meta-optimizer can also track the hardware-specific information. It can tag passes, e.g., denoting the hardware a pass best optimized for or provide an overall score given target hardware. The *meta optimizer* can construct a suggested pipeline of passes for the compiler using all this information.

Reducing the complexity of the described problem is also possible through other means. Prioritizing the system's runtime, especially during online compilation, one can weigh quantum optimization versus wall time. This choice accepts a suboptimal solution by not applying optimization passes that have little gain. Mandatory passes to ensure the circuit is executable on the hardware can not be left out. The second approach simplifies the number of passes by applying larger-scale transformations. For example, it might be unnecessary to consider mapping and routing separately but combine them into a single pass instead. One-step approaches are preferable if they do not come with an additional cost overhead. Directly transforming a high-level algorithm into an optimized, executable circuit can outperform multi-pass approaches. This one-step approach comes with a loss of flexibility and the downside of not tracking the individual benefits of each pass. In HPCQC, the exceptional circumstances pose further challenges, which I will discuss in the next section.

6.2 HPCQC Compilation Tasks

In the HPCQC context, compilation tasks deviate from the standard quantum compilation tasks. The main reason is the unique infrastructure and the relation to HPC hardware. As shown above, conventional tools do not address two challenges. First, the compilation process must be heavy during compile time but fast during runtime to not stall the runtime. Second, the target hardware is heterogeneous, and the compiler must be able to account for changes in the hardware. Accounting for both of these challenges, I propose `Flow Synth`, a tool tackling some of the most common compilation tasks. I build on the existing work by Fürutter et al. [102] and extend it to the *Unified Toolchain*. The tool works based on generative models (see [103]); leveraging machine learning al-

lows me to shift the optimization cost to the training phase. I make it suitable for the *Unified Toolchain* by targeting three main tasks:

1. *Synthesis*: Transform a unitary matrix into an optimized quantum circuit in a single step. My synthesis technique simultaneously targets multiple backends for different circuit parts.

2. *Hamiltonian Simulation*: Generate a quantum circuit for a given Hamiltonian. This task builds on the synthesis task, taking the unitary synthesis as an intermittent step.

3. *ZX-Synthesis*: Leverage ZX-diagrams to optimize computations instead of circuits. ZX-calculus [104] offers an alternative approach to quantum computation, more suitable for error-corrected computation.

I will discuss only the synthesis task; the remaining tasks are part of my ongoing work.

6.2.1 Diffusion and Rectified Flow

Generative machine learning models have gained popularity in recent years. For example, Stable Diffusion [105] allows users to generate images from a text prompt. The concept works based on statistical modeling; One assumes all the interesting images stem from one common probability distribution Π, which one then tries to approximate. The general problem is transforming one probability distribution into another following a discrete number of steps. For stable diffusion, the initial distribution $\mathbb{P}_0^\theta$ is fixed to a well-known distribution, e.g., a Gaussian distribution. Transforming $\mathbb{P}_0^\theta$ to Π is possible by following a latent trajectory Z. The key insight is to model this transformation as a discrete, stochastic Markov process. Hence, $Z = \{Z_t : t \in [0, T]\}$ models $\mathbb{P}_0^\theta$ at $t = 0$ and Π at $t = T$. Z is a diffusion process, described by the stochastic differential equation

$$dZ_t = s_t^\theta(Z_t)dt + \sigma_t(Z_t)dW_t, \quad \forall t \in [0, T], Z_0 \sim \mathbb{P}_0^\theta \tag{6.3}$$

Liu et al. [106] thoroughly discuss the properties of the diffusion process and provide a detailed mathematical derivation. Therefore, the learning process represents an iterative process in which the model adds tiny amounts of noise to learn $p(X_{t+1}|X_t)$. The model needs to learn n such conditional probabilities; crucially, it starts from the target Π. Adding noise according to a given schedule allows the model to converge to the known distribution $\mathbb{P}_0^\theta$. Finding the inverse of the learned functions is trivial; therefore, the model reverses the process during inference. Naturally, one wants to control the output of the model. The current sample can be enriched by supplementing additional information. In literature, the concept is called *conditioning*. Such a *prompt* is typically a textual description of the desired output.

Rectified flow [107] and *latent diffusion* [105] are two extensions of the diffusion process. I utilize both techniques, as they promise faster inference times without compromising the results. In essence, rectified is a combination of a specific noise schedule and reformulation of transport maps $s_t^\theta(Z_t)d$. The goal is to linearize the trajectory of the

diffusion process, which, in theory, requires fewer conditionals. The linearization realizes an interpolation of the different step functions

$$\min_{s_t^\theta} \int_0^1 \mathbb{E}[\|(Z_{t+1} - Z_t) - s_t^\theta(Z_t)\|^2]dt\,. \tag{6.4}$$

Latent Diffusion is a further extension that uses a latent data representation. One uses compressed data instead of learning a potentially high-dimensional function. This technique is common in machine learning tasks, such as the variational autoencoder (VAE) [108]. For the training process, the model has to learn the latent representation of the data. For practical purposes, this can occur detached from the diffusion process. In `Flow Synth`, I formulate the task of circuit compilation as generative tasks. Given a conditioning unitary matrix, the goal is to generate an optimized quantum circuit in a few steps. I provide the exact details of the pipeline in the following section.

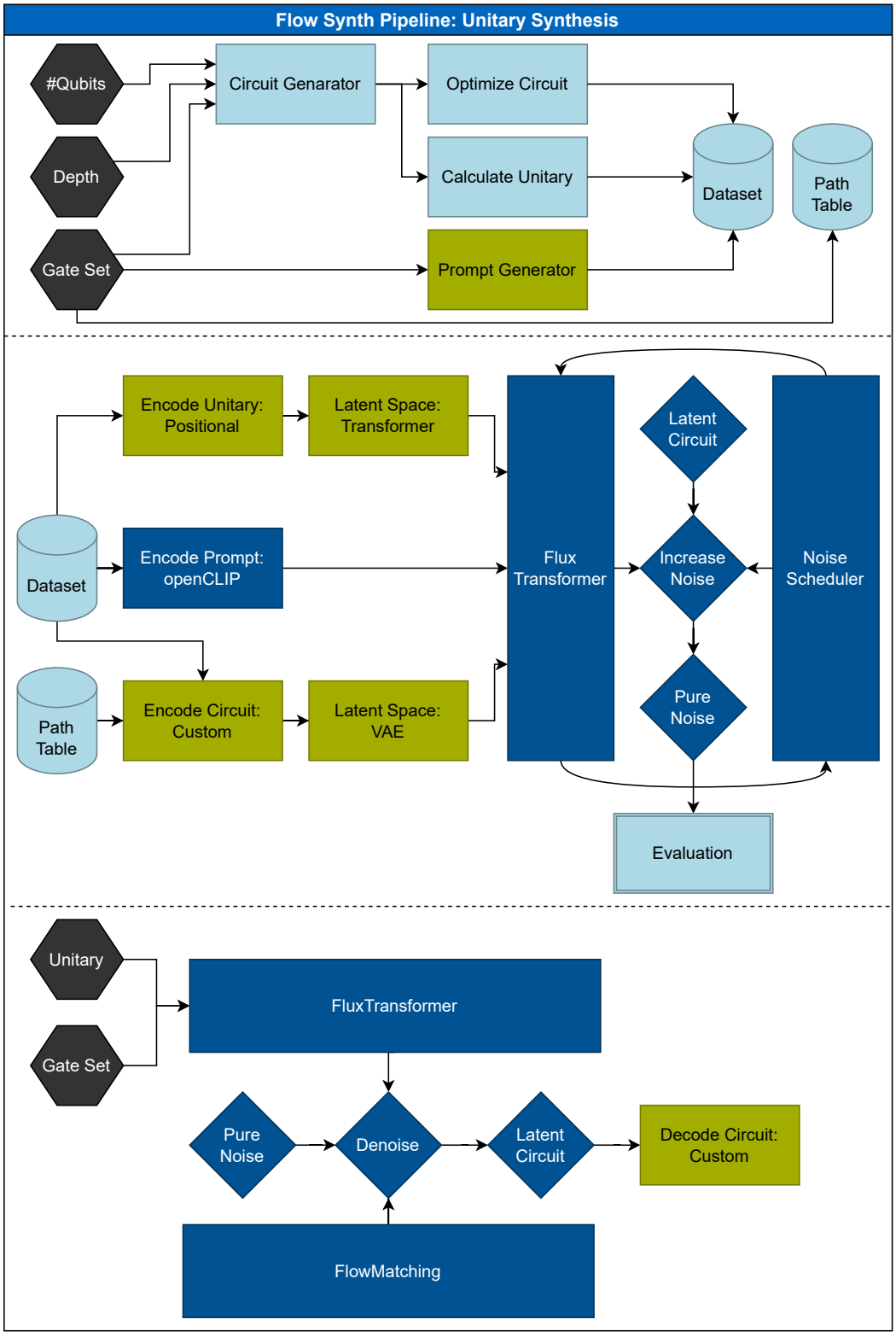
6.2.2 Flow Synth Pipeline

I summarize the `Flow Synth` pipeline in Figure 6.2. It follows three common steps of machine learning: data generation, training, and inference. The inference will become part of the online compilation process, which I will discuss after introducing the general technique. Figure 6.2 shows the approach for the synthesis task, but the other functions follow a similar structure. I will discuss the technical details of the pipeline in the following sections, using the example of synthesis, and only mention differences when necessary. The quality of the training data is crucial for the model's success. Hence, I include a discussion on the data generation process and consider it part of the pipeline.

Overall, the diffusion process is straightforward; I describe the main components in the following sections. First, I generated high-quality data for the HPCQC tasks. It serves as the ground truth for the forward diffusion process, in which the model learns the noisy transport maps. The pipeline encodes a data point and adds descriptive information (the prompt). Iteratively, it adds tiny amounts of noise to the encoded representation until reaching a completely noisy sample. Inference is the reverse process. Pure noise of the desired shape transforms using the prompt as guidance and the model applying the inverse of the learned functions. This process is very generic and transfers to any task, represented in the required tensor format similar to images. For compilation, the lever is in the correct prompting and the quality of the training data. The model will generate similar circuits that follow similar patterns by learning about optimized circuits. My implementation follows the work of Dao et. al [109].

Data Generation

The data generation process is simple. In Appendix B, I show examples of the generated data and the code snippets to generate them It produces a set of quantum circuits (ZX-diagrams) and their corresponding unitaries (Hamiltonians, Tableaus) and prompts. The quantum circuits are the generation target and should be optimal. I propose a straightforward approach to generating the data. Using `MQT Bench` [110], I generate



76 **Figure 6.2:** The `Flow Synth` pipeline. Image taken from my unpublished work.

Table 6.1: Unitary synthesis datasets and their configuration parameters.

Dataset	Max Qubits	Max Depth	Samples
random	8	256	$4 \cdot 10^4$
algorithm	8	256	$4 \cdot 10^4$
heterogeneous	4+4	256	$4 \cdot 10^4$

random circuits and optimize them using the built-in `Qiskit` tools. For the optimization, I select a target gate set, which is also embedded in a prompt of the form

```
"Transpile the unitary for the following gate set: ... ."
```

I calculate the unitary matrix from the circuit by *simulating* it on a classical computer. For the unitary use case, I also generate two additional data sets. One for circuits based on *algorithms* taken from `MQT Bench` and one for circuits suitable for *heterogeneous* hardware. I construct the latter using circuits from the other two data sets and joining them by inserting CX between partitions.

Table 6.1 summarizes the resulting data sets. It is possible to have multiple circuits corresponding to the same unitary matrix. The data is

Encoding

The circuit encoding process is one of the novelties I introduce. It combines the two approaches proposed by Fürutter et al. [102] and Tan et al. [111]. The primary encoding generates a three-dimensional tensor from the circuit of the form $N \times D \times G = (1 + 2 + t)$. N represents the qubit index of a gate, D the depth of the gate, and G the gate representation One can think of this tensor as a matrix where each element is a quantum gate encoded as a vector. The first two dimensions are trivial and can directly arise from traversing the circuit. The third dimension is an embedded gate representation following the path table approach of Tan et al. [111]. It contains information about the gate type by setting a discrete value. Since I train on optimized circuits, only considering one- and two-qubit gates suffices, the encoding could accommodate multi-qubit gates. For example, 0 presents the identity, 1 presents CX, and so on. Negative values of the gate type indicate target qubits, e.g., -1 represents the target of an CX gate. The parameters of gates span the range $[-\pi, \pi]$, and I store them directly in the vector after normalization. The path table entries of the gate produce the remaining elements of the vector. I realize this by traversing the DAG of the circuit, taking l steps to parents, children, or siblings; I repeat this k times. Based on the type of step and gate, I generate a path, which I then look up in the table. To keep the table small, I do not consider the qubit index during path generation since this information already exists in the first dimensions. The $G - 3$ elements, therefore, represent the gate's local environment. Since latent diffusion operates on latent representations, I take this tensor as input for an VAE. A VAE compresses a model by creating a low dimensional bottleneck, which

is then restored and compared to the original input. The latent representation is the output of the encoder and provides the input for the diffusion process.

The matrices follow a simplified approach. First, I transform it by $U \to (\Re(u), \Im(u) \forall u \in U)$. I then add positional encoding [112] to the matrix, a common technique in transformer models. Like the circuits, I train a transformer based encoder to generate the latent representation of the matrix. To encode the text prompt, I use a pretrained `openCLIP` [113] encoder. The latent diffusion is an augmentation of an image pipeline; to further compress the data, it contains a patch embedding for unitaries and circuits. Additionally, many state-of-the-art techniques from vision models such as [114] are part of the pipeline.

Inference

The inference process reverses the training process. I wrap it in a command line interface as a standalone tool; the integration into the MQSS is planned for a future release. As an input, the tool takes the target unitary and a prompt containing the desired gate set. From the unitary, the tool first generates pure noise of the correct size and repeatedly applies the inverse transformations. Due to the flow matching technique, the required steps are relatively small, and the process is fast. The output is the latent representation of the circuit, which the tool decodes to the corresponding circuit. The VAE decoder generates a circuit representation for which I initialize an empty circuit. I iterate over the resulting tensor, adding gates based on the depth, qubits, and gate type. Compared to the encoding, this is cheap, as it is unnecessary to reverse the path table construction. All the necessary information is already present in the other tensor entries.

6.2.3 Preliminary Results

For the preliminary results I present here, I concentrate on two categories. Typically, one would measure the quality of the circuits based on their distribution using the Fréchet Inception Distance [115]. However, I focus on the criteria more relevant to the HPCQC environment. I measure the average inference (compilation) time and the generalization capabilities of the model. For the latter, I evaluate the model trained on the random dataset of the algorithm datasets. To evaluate the optimality, I compare the unitary matrix of the generated circuit to the target unitary. For this, I use the Frobenius norm of the difference matrix:

$$\|U_{\text{target}} - U_{\text{generated}}\|_F = \sqrt{\sum_{i,j} |U_{\text{target}}(i,j) - U_{\text{generated}}(i,j)|^2}\,. \tag{6.5}$$

Table 6.2 shows the preliminary results of the `Flow Synth` pipeline. I train the circuit encoder for 20 epochs; the diffusion pipeline I train for 1500 epochs. Compared to image diffusion models, this model is very small due to the limited resource availability. One advantage of the pipeline is its scalability. The model can be trained on larger data sets for longer without much effort. Although the current pipeline is not agnostic of the input size, one can adapt this using masking [102].

Table 6.2: Preliminary results of the `Flow Synth` pipeline compared to the `Qiskit` optimization. For evaluation, 300 random unitary matrices are generated and optimized.

Model	Inference Time	Frobenius Norm
`Flow Synth`	65.99s	$1.6 \cdot 10^{-5}$
`Qiskit`	81.02s	$1.2 \cdot 10^{-8}$

My preliminary results show the expected tradeoff between inference time and the approximation quality. The results I show serve as a proof of concept; I generated them from a model checkpoint generated at epoch 50. The utility of this class of models is apparent, as the cost of evaluation dominates the overall cost, which scales better with the input size compared to existing methods. Typically, the input size of a training circuit is dominated by the length of the path table and the circuit depth, which the user can control. Scaling the qubit dimension is not as challenging, especially considering diffusion models work well even for high-resolution images with a similar order of magnitude. Therefore, the biggest constraint is the data availability. `Flow Synth` already offers some integration with HPC components due to the built-in functionality of underlying libraries. Still, I achieved the results without fine-tuning the model or the training process. Leveraging customized parameters and improving the model architecture, I expect to achieve better results in the future.

The `Flow Synth` pipeline is a promising tool for the *Unified Toolchain*. Still, there remain many possible improvements. So far, I only compare the results by a proxy metric, the Frobenius norm. Evaluating the circuits on the actual hardware will be necessary to get a true sense of the performance. I have not yet integrated the model into the `MQSS`, and the inference time is too high. And aside from the gate set, I do not impose any hardware restrictions on the model. The model must support this, enabling end-to-end functionality. One possible way to incorporate hardware constraints is to include them in the conditioning. For example, one can treat available backends as classifiers, similar to image classification. Alternatively, if a mapping to a tensor is possible, the model can support further inputs.

6.3 Hybrid Compilation

So far, I have focused on reducing the compilation time of quantum circuits. Compilation time is one of the significant bottlenecks in the current software environment. Still, possible improvements are targeting different points of the compilation process. In between the two parts of compilation, the scheduler handles the circuit. I will cover the scheduling aspect of the runtime in the next chapter; the compiler still benefits from tight integration with the scheduler. First, the compiler can run during the circuit's queueing time. The compiler can decide which passes to choose by querying the queue times. Vice versa, the compilation's effects can then inform the scheduler. I consider compilation time part of the setup time of a QPU, a crucial metric for the scheduler.

Further, the compiler can also take advantage of caching strategies. For example, it can exploit data locality when reusing precompiled circuits, reducing overall runtime.

Compilation is still a volatile topic, and new techniques emerge frequently. To leverage the advances, QIR offers a standardized interface for developers. In the *Unified Toolchain*, the resulting pass system has a twofold benefit from separating offline and online compilation. Using QIR allows tight integration, but this does not solve all problems. Novel techniques are necessary to enable efficient compilation for HPCQC environments. I include two new techniques in the compilation process to deal with the challenges arising from the split compilation process. For the short term, using *circuit cutting* enables users to run more significant problems not feasible on current hardware. I investigate the potential of *circuit cutting* and introduce compilation techniques to optimize for it. The concepts are independent of the underlying cutting implementation and applicable to distributed quantum computing. A unique challenge arises from the heterogeneity of the hardware. I introduce `Flow Synth`, a machine learning-based approach to quantum compilation, to tackle this. It provides two main benefits: first, it shifts the optimization cost to the (offline) training phase, and second, it enables the compilation of multiple backends. In conjunction with the proposed *meta optimizer*, this approach offers a promising solution for the *Unified Toolchain*. Thereby, I provide a comprehensive component of the runtime environment for the hybrid HPCQC environments.

7 Scheduling

In HPC, scheduling is at the intersection between compile time and runtime of a program. Once the user has optimized, they will submit the program for execution. Classical environments shape the expectations toward such a system. Limited resources are shared among many users without compromising the performance of individual programs and the entire system. A notion of *fairness* is expected, such that no single user can starve the rest; I will define this in Section 7.2. In terms of execution, the scheduler provides the necessary resources to the program. Under the assumption that multiple hardware resources are requested, a scheduler will allocate program execution across the available resources. The distribution also has implications for communication and data locality during execution. It might be necessary for the scheduler to reorder the execution of programs to minimize wait times, but scaling should remain unaffected.

When adding more hardware components, scheduling increases in complexity. Deciding which parts of the program can run on specific hardware is left for the user to determine and is not considered a responsibility of the scheduler. The choice can become a bottleneck, especially with limited quantum resources, as users typically lack the bigger picture. High utilization is crucial when dealing with resource limits, but in quantum computing, hardware exclusivity prevents users from sharing resources. The notion of parallelism is also different in quantum computing. While the simultaneous exploration of the solution space is a crucial feature, the actual execution of the program is sequential. For example, allowing users to run multiple instances of the same program in parallel improves the time-to-solution; other parallelization techniques could also benefit the user.

In HPCQC, scheduling is an underrepresented field caused mainly by current systems employed in standalone systems. From an integration standpoint, scheduling is an integral part of the runtime environment, enabling the efficient targeting of hybrid systems. My contribution builds on my previous works on scheduling [116, 117]; to my knowledge, it is the first approach to target heterogeneous quantum systems and combine scheduling decisions with circuit cutting. As a prototype, the scheduler allows for exploring the requirements and limitations in multi-user environments, focusing on hybrid quantum algorithms. Still, it attempts to be tightly integrated to enable high utilization in near-term use cases.

7.1 Challenges

Simply integrating quantum resources into classical schedulers is not feasible. The technological requirements vary greatly, and the different platform time scales (examples given in Table 2.3) can not be ignored. As an example, a circuit with ten sequential

two-qubit gates would take approximately 200 ns on a superconducting platform but over 3 µs on an ion trap. This example does not include any other operations, but considering that superconducting platforms typically have restricted topologies, the time scales can differ even more. Further, *fairness* is built on an account of the runtime of a program. If a program is stretching the limits, it will be halted and resumed at a later point. The process of *preemption* takes the entire state of a program and stores it for later resumption. Preemption includes the stack and the heap, for which storage is cheap for most classes of classical programs. For quantum programs, *preemption* is restricted by two factors. First, the state of a quantum program is not easily serializable. Serialization would require long, coherent quantum memory or the ability to store the state in a classical memory. The former is still impractical, and the latter approach would require a lot of classical memory. Replaying the entire computation would also not be feasible, as this would require the same amount of time as the original computation. Second, compared to the time scales of operation, the configuration of the quantum system is typically slower. Recompilation of programs might be necessary, but the programs must be guaranteed to run on the same device to do this in advance. Some device-specific processes, for example, generating the correct device layout, would still add unavoidable overhead. Differences in device calibration between runs could still lead to unexpected results, negatively impacting computations, especially in the case of variational algorithms. Estimating quantum runtime is also hard [118] and unreliable enough to be used for scheduling. For example, repeat-until-success circuits can take an arbitrary amount of time, and even for a high probability of success, they are still nondeterministic.

Extending optimization to include scheduling is also appropriate. Quantum hardware has unique and exploitable properties, exemplarily by the variety of platforms, Tied to these properties are quantum-specific metrics, most common fidelity $\mathcal{F}$, which a scheduler must consider to ensure a successful computation. Minimizing the effects of noise is usually done when mapping the logical qubits to the physical qubits for a single circuit. If the scheduler maps multiple circuits to the same device without considering that this decreases average gate fidelity, results might differ from the expected isolated outcome. Some side-effects of the noise have to be taken into account as well. Quantum systems require downtime for calibration and maintenance. As a result, hardware properties change over time, typically degrading until a reset is performed. Ravi et al. [119] show that running a program on different calibrations can lead to diverging results. Further, running multiple programs on the same device can have adversarial effects. For example, running repeated CX gates affects gate-fidelities negatively on neighboring qubits [120]. Even some privacy concerns can arise from this, as it is possible to extract information about the state of a qubit by running a program on a neighboring qubit [121].

Besides making the scheduler aware of this, one can use hardware functionality to mitigate these effects. One of the beneficial properties of trapped ions and neutral atoms is the ability to reconfigure the device. In trapped ions, interactions are performed in specific areas of the chip [28]. Reserving these areas for specific programs can mitigate the effects mentioned above. Similarly, neutral atoms can be moved around in 3d space [122], allowing for a physical separation of programs. Other platforms could be

decoupled internally on a hardware level. The toolchain and especially the scheduler must be sufficiently capable of using these features. One conclusion from this is that the interaction patterns for the scheduler are different than in classical systems. I will detail these interactions below. Further, this increases the device configuration times. An optimal hardware setup must be computed by the compiler and physically implemented by the device. Such schemes align with the established problem of higher setup times for quantum devices than their operation times.

Taking advantage of these features requires a more sophisticated scheduler. In the toolchain, the scheduler is at the intersection of the compilation and the execution of the program. Its goal is to schedule resources as early as possible to allow for a more efficient compilation. However, this requires fully specifying programs in advance. Classical schedulers like `SLURM` [123] can already deal with heterogeneous resources, but only ahead of time. For quantum algorithms, two additional challenges can arise from this. Firstly, there are timing critical parts of the program due to the coherence times of the qubits. This limit requires that the classical thread, responsible for communication, is not suspended during the interaction. Further, some complex interaction schemes arise from the compilation process. For example, optimization needs hardware information to be effective. During optimization, the characteristics of the circuit might change, making the hardware information outdated. A second round could be started from here, potentially triggering a continuous feedback loop. The concept is visualized in Figure 7.1. Existing techniques can address some of the problems. Gang scheduling [124], for example, can be used to ensure that all parts of a program execute at the same time. It is advantageous for scheduling codependent tasks, but extending these techniques to quantum computing is not trivial. Novel approaches are required to resolve the issues of the feedback loop. In this chapter, I systematically develop a suitable solution. For this, I will establish a scheduling foundation to incrementally improve and derive a solution. Finally, the integration into the toolchain is the last step, which I will also cover in this chapter.

7.2 Premises

Before improving the scheduling, I need to establish a baseline. For that, I will first recap the current state of scheduling in quantum computing and then derive a baseline approach to scheduling. To faithfully capture the nature of the problem, I make some reasonable assumptions about the system. First, I assume a scenario in which one HPC system integrates with *multiple and unique* quantum devices. Each device has a black-box interface to which the runtime can submit individual quantum circuits. These devices have some limitations that a compiler must resolve before a circuit can run on them. A common characteristic is that these devices execute circuits sequentially over multiple *shots*; I do not consider analog devices. The quantum circuits submitted by the users are not known in advance, meaning the compiler constructs them only at classical runtime, and therefore, the devices cannot be pre-determined. Hence, their compilation also runs after the runtime establishes an initial hardware schedule. Additionally,

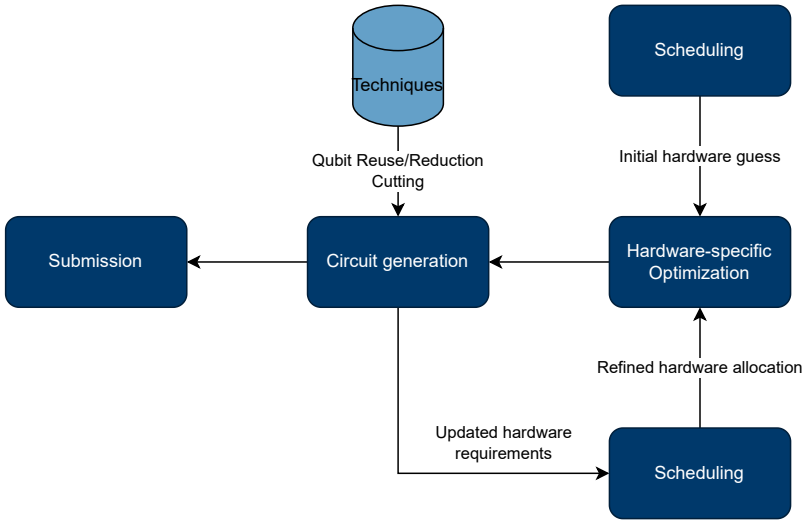


Figure 7.1: Feedback loop between the scheduler and circuit generation components.

the resources are *shared* among multiple users; The system should maximize resource utilization.

To analyze the current state of scheduling in quantum computing, I will compare the approaches of `Qiskit` [40] and `XACC` [46], which I introduced in Chapter 2. I will evaluate them against the premises established above. To highlight their shortcomings, I will also compare this to standard approaches in HPC. The goal is to identify the differences and derive a baseline approach.

7.2.1 Quantum Scheduling

`Qiskit` offers a cloud-centric approach to scheduling. For this, they offer two different modi operandi. The first is a simple priority queue, with some fairness mechanisms depending on the devices. A user can specify a device before submission, and the optimization pipeline runs in advance on the user side. Once the circuit is submitted, it is added to the queue and executed once the device is available. During the execution, the circuit occupies the whole device, independent of the requested resources, for the duration of a circuit times the number of shots. Until the results are available, the user can do other tasks or wait for them. This limitation leads to high queueing times, as only one user can execute a single circuit, making this a bottleneck. First, on-the-fly reconfiguration of the circuit is impossible, except for parametrized circuits. Regular updates are necessary for variational algorithms, so the queueing time quickly increases. In the worst-case scenario, between runs, a recalibration of the device is necessary, invalidating the previous results.

Some tools introduce a second approach as a solution to this problem. Here, the user submits a program instead of a circuit. The user specifies a target device beforehand,

for which the provider reserves an exclusive timeslot for the user. Variational algorithms profit from this as the interaction loops get shorter. Eventually, this approach could include real-time computation, but such features have yet to be available. Some of the shortcomings of the previous approach still apply. Exclusive access is a limiting factor for utilization. Exclusivity is suitable for IBM since they charge for access, not utilization. Putting the responsibility of choosing a suitable device is then a cost decision and not only an optimization for the user. Hence, applying such scheduling approaches only partially applies to HPC.

`XACC` operates based on an on-premise approach. While it is possible to have remote quantum devices, they suffer the same downfalls as the cloud-centric approach. Built-in, there is no direct scheduling mechanism. This lack of a scheduler means the user has exclusive access to the hardware, or the scheduling is part of the black box interface. In `XACC`, it is possible to define a scheduler service or build one as a decorator to extend devices using the plugin system. With the latter, the problem is only moved to the user space; hence, the problem is not solved. Providing scheduling as part of a service is a better approach and can solve some problems that other tools still need to address. For example, running this service as a shared library would allow multiple users to submit circuits to the same device. The problems arise when trying to resolve the interaction scheme between the scheduler and other components. A back-and-forth between the scheduler and the compiler is not intended in `XACC` and would require multiple translations between the components. Also, making a scheduler mandatory and preventing direct access would depend on overhauling some core components of `XACC`.

For the scenario described above, neither approach is suitable. The cloud-centric approach is not feasible due to the shared resources, and the on-premise dedicated approach needs to be more flexible to address the problem. In the context of the toolchain, it is clear that other approaches are necessary. The scheduler has to consider the uncertainty of the quantum program by design. Further, it also needs to apply the principles of HPC scheduling to ensure high utilization of the resources.

7.2.2 HPC Scheduling

In HPC, scheduling has longstanding applications like `SLURM` [123] or `PBS` [125] which are well established. Despite their functional maturity, their scheduling algorithms are reasonably straightforward. For example, first-come, first-serve is a common approach, sometimes prioritizing larger jobs to utilize the resources better [126]. Simple approaches are suitable due to the complexity and size of the systems they manage, making more sophisticated algorithms infeasible. While implementing quantum resource scheduling in these systems is a long-term goal, adopting some of the principles is more feasible for the time being. An important notion is the concept of the runtime of a job and the resources it requires. Usually, a single metric called virtual runtime combines these aspects, including values such as priority, niceness, and other factors. Accurate estimates enable schedulers to make informed decisions; acquiring these estimates is a critical challenge in quantum computing. Hierarchical scheduling can be employed to deal with the

increasing number of resources, dividing the resources into smaller parts scheduled individually. A similar approach could work for a smaller number of quantum devices, each assigned to a different hierarchy member. For complex systems, gang scheduling [127] ensures that all program parts execute simultaneously. As long as decoherence is still a limiting factor, this is optional. Single programs will be short enough not to require this; however, preparing quantum resources, e.g., encoding classical data, is still mandatory. The schedule must, therefore, be precomputed to accommodate this. Other scheduling concepts, like backfilling and outages, directly apply to quantum computing. One can easily extend them to quantum resources, albeit with some modifications. Similarly, adding a noise model to the scheduler considers noisy resources and uses the knowledge to estimate the fidelity of the results. This estimate can then be integrated into the scheduling decision to ensure the results are within the desired range.

The difficulties in integrating quantum resources into classical schedulers are manifold. For example, `SLURM` offers a system for heterogeneous hardware to be integrated and considered in the scheduling. But this is built on the assumption that the problem is known beforehand. And even if the problems are predetermined, this would not consider the quantum-specific circumstances. This situation might change once quantum resources become stable over extended time frames, but this is not feasible for now. In a worst-case scenario, the scheduler could preempt a quantum program close to completion. Then, a second program would undergo compilation and hardware trigger reconfiguration, and the program would start to execute. A second reconfiguration would be necessary to resume the original program for its final iterations. Multiple reconfigurations are infeasible compared to the cost of switching to a classical context. A scheduler for quantum resources has to be aware of such inhibitions. Considering them requires close interaction with the circuit generation components and tight hardware integration. One such component is responsible for generating distributed circuits from a single program. In the next section, I provide the technical background for this.

7.2.3 Parallelization

One crucial aspect of HPC and the ability to utilize resources is parallelization. The ability to produce flying qubits is part of the original DiVincenzo criteria (c.f. Definition 4). Quantum operations can interact over long distances due to the entanglement of qubits. These qubits can be part of different devices so that algorithms can communicate over multiple devices. Using them would require that before the operation, entanglement resources are created via the teleportation protocol [128]. In an algorithm, computation qubits must be converted to or entangled with the flying qubits to realize the communication. The technique to achieve this is called *gate teleportation* [129], the respective circuit is shown in Figure 7.2.

The problem with gate teleportation is twofold. First, a resource manager must create entanglement resources in Bell pairs and keep them stable throughout the computation. Second, the teleportation protocol requires measurements based on classical information. Classical interaction requires modifying the circuit on the fly and communicating the condition during the coherence time of the qubits.T the term *local operations and*

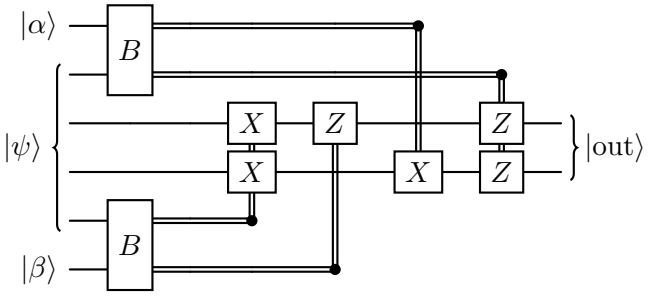


Figure 7.2: Gate teleportation circuit, adapted from [129].B is a Bell measurement.

classical communication (LOCC) summarizes these requisites, which is infeasible for current hardware. The desired alternative, *local operations* (LO), requires different schemes to thoroughly detangle communication and computation. *Circuit cutting* [81] presents a solution to this problem, which I have introduced as part of a compilation in Section 6.1.3. Initially, it still required two-way classical communication [93, 94], which was first relaxed to one-way communication [95] (order of execution matters) and then to local operations [96]. To summarize the concept, a tool cuts a circuit into smaller parts, which can run in parallel if space is available but also sequential. Reducing the size of circuits is traded for an increase in the number of circuits and an exponential number of required samples. This trade-off is only necessary if a circuit does not fit on hardware, so I will prioritize the value of running a circuit to not running it at all. The techniques in my proposal work based on circuit cutting, but once distribution arises, one can lift them to support other means of parallelization.

7.2.4 Baseline

The current approaches are not suitable for the scenario described above. They do not consider quantum computers a shared resource and do not view scheduling as part of the toolchain. Further, the algorithms are not capable of utilizing hardware parallelism. For a better comparison, I will describe a baseline approach requiring minimal effort. I establish it on the following assumptions:

1. Circuits are limited to the size of the maximum available quantum device. This restriction means that circuit cutting has already taken place.

2. If reasonable, multiple circuits can be combined to run simultaneously on the same device. I assume the simultaneous execution does not amplify noise.

3. There will be only one scheduling phase. After the scheduling decision, the circuit is fed once through a device-specific compilation time.

4. Devices offer discrete time slots in which they can execute a circuit. Assuming that each shot of a circuit takes the same amount of time, one can interpret the time slots as a minimal common denominator of the shots of all circuits.

Algorithm 2: Scheduling with first-fit decreasing bin packing, following [130].

```
Input  : J := set of jobs, B := list of bins
Output: Bins filled with jobs
J' ← sorted(J, qubit_count descending);
open ← B;
closed ← ∅;
foreach job ∈ J' do
    bin ← FindFirstFitting(job, open);
    if bin is none then
        new ← B;
        bin ← FindFirstFitting(job, new);
        extend open with new;
    end
    add job to bin;
    if bin is full then
        remove bin from open;
        add bin to closed;
    end
end
add all open to closed;
return closed;
```

From these assumptions, one can see that this approach can only lift the restrictions of the shared resources. The notion justifies that one would instead run a circuit, potentially with a lower fidelity, than not run it at all. An approach under these restrictions is bin packing and equipping each device with a FIFO queue. The algorithm in Algorithm 2 shows an implementation of this approach. The scheduler sorts the closed bins by their device and, according to their index, enqueues the bins' contents in the device queues. Bin packing has already improved compared to existing approaches in that it includes circuit parallelization.

7.3 Linear Programming

The first step to improve the scheduling is to formalize the problem. In literature, one typically programs the definition of the scheduling problem. This programming approach is commonly used in traditional scheduling problems [131–134] and extends to quantum computing. The classical formulation assigns open tasks to the available machines. In quantum computing, tasks correspond to circuits and machines to devices; I use the terms interchangeably. For example, [135] uses a partitioning-based optimization to solve the scheduling problem for quantum computers. An objective function, modeling the problem, has to be defined. So far, related work focused on minimizing the expected noise. For single QPU systems, this is analog to the mapping problem, but for multiple

Table 7.1: MILP input parameters.

Variable	Definition
J	Set of jobs (circuits)
M	Set of machines (QPUs)
p_{im}	Processing time of job i on machine m
s_{ijm}	Setup time of job j after job i on machine m
q_i	Required resources (qubits) of job i
Q_m	Available resources (qubits) of machine m

QPU systems, this is more complex. A QUBO formulation has been proposed for the distributed case [101], based on the assumption that gate teleportation has considerable noise and time overhead. As discussed in Section 6.1.3, one can similarly reformulate the scenario of unrelated backends. Recently, Bhoumik et al. [136] proposed an ILP formulation for the scheduling problem focusing on individual qubits on a single device, offering fine-grained control. My approach can be extended by such possibilities in a multi-leveling scheduling algorithm:

1. Assign circuits to devices, including interconnected setups through my work [116].
2. Assign circuits between distributed systems, e.g., through [101].
3. Assign circuits to qubits on a single device through [136].

7.3.1 Input

I will follow the approach of Al-harkan and Qamhan [133] and formalize the scheduling problem as a mixed integer linear program. Table 7.1 contains a brief overview of the notation I use. This formulation allows the encoding of processing and setup times inaccurate values. The processing times p_{im} expresses the time a circuit i takes on device m. A requirement is that the runtime environment can estimate the processing times in advance for all machines. This condition sets up the challenge of generating a runnable circuit for each device; for simplicity, I assume online compilation can satisfy this. The variance of quantum hardware is low enough to allow for this assumption. If this cannot be guaranteed, one can extrapolate to a set of shots. For example, one can normalize the processing times to the shortest job in terms of shots. A single execution would then be defined as a multiple of the base amount, giving the more robust estimates for execution. This redefinition converts p_{im} to p'_{im}, which are integer values and can replace the original values.

The second real value to encode is the setup times. For each combination of jobs i, j and a machine m, the setup time $s_{ijm} multiple factors govern$. First, the hardware reconfiguration contributes to the setup time. Setup can be a significant factor, especially in the platforms that allow for physical tweaking of the device. For example, reconfiguring the layout of an atom trap can take up to 50 ms [122]. All hardware platforms have

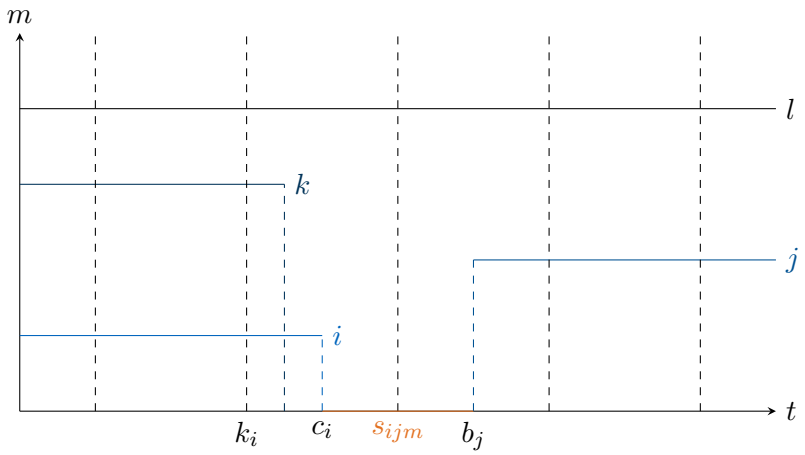


Figure 7.3: Example of the successor relationship as defined in Equation (7.1). The job i is the latest job to terminate at c_I before job j starts at b_j. The setup time s_{ijm} is the time between the end of i and the start of j. Jobs k and l do not terminate in between and therefore are not considered in the relationship.

these specific setup times, making these setup values necessary. Second, the compilation time of the circuit plays an important role. In the simplest case, switching between parametrized circuits, compilation is negligible. Similarly, the recompilation for circuits created from circuit cutting can be assumed to be small when the number of cut gates is low. The hardware characteristics also influence compilation. All-to-all connected devices can skip mapping and routing, while for restricted devices, this is mandatory to ensure the conditions of the circuits. As a result, setup times can vary significantly between devices and between jobs on the same device.

With the goal of high utilization comes an additional challenge. Multiple circuits can run on the same device. Not considering noise, this opens a scenario where one circuit runs while multiple other circuits use the rest of the according device. In common scheduling problems, a machine usually is restricted to one job at a time. To lift this restriction, one can discretize time. The binary variable z_{imt} is 1 if job i is run on machine m at time t. For each machine, the restriction is then $\sum_i z_{imt} \leq 1$ valid for all times. Instead, one could consider every qubit individually, which would increase the problem's size significantly. To solve this issue (c.f. [116]), I propose a successor relationship encoded in the binary decision variable y_{ijm}.

$$y_{ijm} = 1 \iff c_i < c_j \wedge \nexists k \in J : c_i < c_k < b_j \wedge \gamma_{ijm} \wedge \gamma_{ikm} \tag{7.1}$$

I use the helper variable γ_{ijm} to encode two jobs i, j running on the same device. The key to establishing a successor is to compare the completion times of other jobs on the same machine. According to the definition, job j succeeds job i if the completion time of job i is smaller than the start time of job j. Additionally, I require that no other job k finishes between the end of i and the start of j. Figure 7.3 contains an example.

Table 7.2: Real decision variable, defining the position of a job in the schedule.

Variable	Definition
c_j	Completion time of job j
$c_{\max}$	Makespan
b_j	Start time of job j

Table 7.3: MILP Helper variables.

Variable	Definition
α_{ij}	1 if job i completes before job j starts
β_{ij}	1 if job i completes before job j completes
γ_{ijm}	1 if jobs i, j are scheduled on machine m
δ_{ijkm}	1 if job k completes after i completes but before j starts on m

7.3.2 Formulation

The utilization of quantum systems has been rarely considered. Optimizing the timing of circuit submission has been discussed for a single IBM cloud device in [119] and already highlights the need for such an approach. A simple metric for this is the time to solution. In scheduling it is common to use the notion of makespan, which is the duration from the start to completion of a job, to optimize the utilization of resources. The selection of the objective function (OBJ) is therefore straightforward, given the real decision variables listed in Table 7.2. The completion time of the last job to complete in the schedule should be minimized ((C1)) Hence, the objective function is formulated as:

$$\min(c_{\max}) \tag{OBJ}$$

Subject to:

$$c_j \leq c_{\max} \quad \forall j \in J \tag{C1}$$

$$c_0 = 0\,. \tag{C2}$$

Which defines a dummy job 0 that starts at the completion time 0 (C2).

The quantity of interest is the schedule and not the actual completion times. It can be reconstructed from the binary decision variables $x_{im} = 1 \iff$ job i is scheduled on machine m and the successor relationship. Succession is defined as $y_{ijm} = 1 \iff$ job i is the latest job to complete before job j on machine m according to (7.1). Additionally, I treat qubits as a limited resource, which is refreshed at each time step. To ensure this I use a binary decision variable $z_{imt} = 1 \iff$ job i is scheduled on machine m at time t. According to the Graham classification scheme [137], the problem is therefore an instance of $\alpha|\beta|\gamma$. I formulate it explicitly(c.f. [116]), using the helper variables Table 7.3, which can be simplified to use fewer variables in the actual implementation.

The problem is subject to the following constraints:

$$\sum_{m \in M} x_{jm} = 1 \quad \forall j \in J \tag{C3}$$

$$\sum_{m \in M} z_{jmt} \leq 1 \quad \forall j \in J, \forall t \in T \tag{C4}$$

$$c_j \geq b_j + \sum_{m \in M} p_{jm} \cdot x_{jm} + \sum_{i \in J \cup \{0\}} \sum_{m \in M} s_{ijm} \cdot y_{ijm}$$
$$\forall j \in J \tag{C5}$$

$$b_j \geq c_i + \mathbb{M} \cdot \left(\sum_{m \in M} y_{ijm} - 1 \right)$$
$$\forall j \in J, \forall i \in J \cup \{0\} \tag{C6}$$

$$\sum_{m \in M} \sum_{t \in T} z_{jmt} = c_j - b_j + 1 \quad \forall j \in J \tag{C7}$$

$$\sum_{t \in T} z_{jmt} \leq \mathbb{M} \cdot x_{jm} \quad \forall j \in J, \forall m \in M \tag{C8}$$

$$c_j \geq t \cdot \sum_{m \in M} z_{jmt} \quad \forall j \in J, \forall t \in T \tag{C9}$$

$$s_j \leq t \cdot \sum_{m \in M} z_{jmt} + \mathbb{M} \cdot \left(1 - \sum_{m \in M} z_{jmt} \right)$$
$$\forall j \in J, \forall t \in T \tag{C10}$$

$$\sum_{j \in J} q_j \cdot z_{jmt} \leq Q_m \quad \forall m \in M, \forall t \in T \tag{C11}$$

(C3) and (C4) enforce that each job is scheduled on exactly one machine and at most one machine at any time. The completion time of a job is at least its processing time, and setup time according to its predecessor after it has started ((C5)). (C6) ensures that the predecessor of a job is done before starting it, including the dummy job at time 0. Entirely processing a job is fixed by (C7); (C8) align x_{im} and z_{ijm} such that they use the same machine. (C9) and (C10) ensure the execution takes place between start end completion. (C11) constrain the resource usage, namely that the qubit count of all jobs running on a machine at a time is below the available qubit count of the machine. The previous constraints could easily be adapted to time-dependent resource availability to model erroneous qubits or partial recalibration procedures.

The successor relationship is ensured by the constraint (C20) which requires some additional setup:

$$1 \leq \sum_{i \in J \cup \{0\}} \sum_{m \in M} y_{ijm} \quad \forall j \in J \tag{C12}$$

$$\mathbb{M} \cdot x_{jm} \geq \sum_{i \in J \cup \{0\}} y_{ijm} \quad \forall j \in J, \forall m \in M \tag{C13}$$

$$\mathbb{M} \cdot x_{jm} \geq \sum_{i \in J \cup \{0\}} y_{jim} \quad \forall j \in J, \forall m \in M \tag{C14}$$

$$z_{jm0} = y_{0jm} \quad \forall j \in J \forall k \in M \tag{C15}$$
$$\mathbb{M} \cdot \alpha_{ij} \geq b_j - c_i \quad \forall i, j \in J, i \neq j \tag{C16}$$
$$\mathbb{M} \cdot \beta_{ij} \geq c_j - c_i \quad \forall i, j \in J, i \neq j \tag{C17}$$
$$\gamma_{ijm} \geq x_{im} + x_{jm} - 1 \quad \forall i, j \in J, i \neq j, \forall m \in M \tag{C18}$$
$$\delta_{ijkm} \geq \alpha_{kj} + \beta_{ij} + \gamma_{ijm} + \gamma_{ikm} - 3$$
$$\forall i, j, k \in J, i \neq j, \forall m \in M \tag{C19}$$
$$y_{ijm} \geq \alpha_{ij} + \left(1 - \sum_{k \in J} \delta_{ijkm}\right) + \gamma_{ijm} - 2$$
$$\forall i, j \in J, \forall m \in M \tag{C20}$$

Each regular job has one predecessor, which can be the dummy job 0, according to (C12); if a job starts at time 0, it has to use the dummy as predecessor (C15). A job and its predecessor are scheduled on the same machine, due to (C13) and (C14). The helper variables (Table 7.3) are used to ensure that the successor relationship is maintained and enforced by (C16)–(C19).

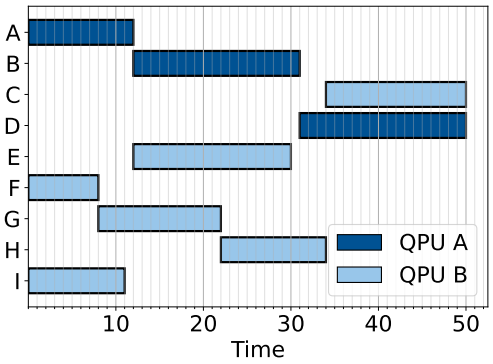
A clear drawback to this design is the complexity of the problem. For realistic numbers of jobs and machines, the number of variables and constraints can quickly become infeasible. A simple assignment for the processing times in Figure 7.4b already results in a problem with 3188 variables and 3272 constraints. To simplify this one can omit sequence-specific setup times, and instead use only machine-specific setup times. This can be changed by setting $s_{jm} := \max_i s_{ijm}$, and update (C5) to

$$c_j \geq b_j + \sum_{m \in M} p_{jm} \cdot x_{jm} + \sum_{m \in M} s_{jm} \cdot y_{ijm} \quad \forall j \in J\,. \tag{7.2}$$

In the example this would reduce the number of constraints to 1208 and the number of variables to 1355. In general this is a reduction from $\mathcal{O}(|J|^2)$ to $\mathcal{O}(|J|)$, which is a significant improvement. I will discuss the results later, but it is clear that this approach is not scalable to suffice the HPC requirements.

7.4 Heuristics

The two alternatives to the linear programming approach are higher-order constraints or heuristics. I will focus on the latter since the former does not guarantee better performance. Using metaheuristics or machine learning, one can achieve approximate results. Multiple metaheuristic techniques exist in scheduling literature, including genetic algorithms, simulated annealing, and tabu search [138]. Combinations of those tools are possible, usually done for performance, result quality, and scalability tradeoffs. The key to these approaches is to encode domain-specific knowledge into the algorithm. For machine learning, the algorithm can learn the patterns in the data and generalize them to new instances. Li et al. [139] use reinforcement learning to optimize the scheduling of quantum circuits. The biggest drawback is the ample solution space, which requires



(a) Example schedule where most calculation is executed on a single QPU.

Job	QPU 1	QPU 2
0	0	0
A	12	11
B	12	10
C	9	12
D	10	10
E	10	8
F	7	8
G	6	9
H	6	8
I	9	8

(b) Machine dependent processing times ($p_{i,k}$).

Algorithm 3: Scatter search for scheduling.

Input : $J :=$ set of jobs, $M :=$ list of machines, $N :=$ number of iterations
Output: Approximate Schedule
population ← `InitializePopulation(`*J, M*`)`;
bestSolution ← `SelectTopN(`*population, 1*`)`;
for 1..*N* **do**
  newSolutions ← `GenerateNewSolutions(`*population*`)`;
  improvedSolutions ← `ImproveSolutions(`*population*`)`;
  population ← *population* ∪ *newSolutions* ∪ *improvedSolutions*;
  eliteSolutions ← `SelectTopN(`*population*`)`;
  diverseSolutions ← `SelectMostDiverse(`*population, topN*`)`;
  population ← *eliteSolutions* ∪ *diverseSolutions*;
  currentBestSolution ← `SelectTopN(`*population, 1*`)`;
  if *currentBestSolution* < *bestSolution* **then**
    bestSolution ← *currentBestSolution*;
  end
end
return *bestSolution*;

much data for the initial training procedure. A suitable restriction is considering a batch system, where the number of jobs is limited; I discuss this in Section 7.5. Both approaches benefit from this restriction, making the problem more tractable.

7.4.1 Scatter Search

Multiple metaheuristics are suitable for scheduling problems [131, 132]. I concentrate on *Scatter Search*, which combines genetic algorithms and tabu search concepts. The algorithm is shown in Algorithm 3.

Scatter search is a population-based algorithm that starts with an initial population of solutions. These solutions are then improved by generating new solutions and enhancing the existing ones. To ensure diversity, the best solutions are mixed with solutions that differ according to a distance metric. The best solutions are then selected, and the process is repeated. One of the advantages of scatter search is that one can efficiently parallelize IT. The loops are data-independent and, hence, embarrassingly parallel. Factually, using parallelism increases the number of solutions generated and improves the quality of the results. Ultimately, the workers must communicate the best individual solutions to select the best overall. Alternatively, one could produce a large set of initial populations and create a map-reduce-like approach.

7.4.1.1 Initialization

The initialization of the population is essential for the algorithm's performance. Close to optimal initial solutions improve convergence. Additionally, the diversity of the initial population forms the basis of the search in the solution space. If the solution space has multiple local optima, the algorithm can converge at least provide the best local optima. A greedy algorithm can initialize solutions to provide performance guarantees, for example, the first-fit decreasing bin packing algorithm from Algorithm 2. Further, one can use any other scheduling algorithm to generate a set of initial solutions.

The initialization phase can tackle the interplay between cutting and scheduling (c.f. Figure 7.1). The initial solution consists of possible cut choices. These cut choices generate new circuits, which are then scheduled, for example, by bin-packing or randomly assigning them to machines. The two approaches for cutting I present here implement the idea of minimizing cutting overhead. If possible, one should avoid cutting, but rather than depending on only one device (typically the largest), one can still benefit from scheduling to smaller devices. If circuits are larger than the biggest device, cutting becomes necessary.

The first idea is to minimize the number of cuts. By searching for a partitioning of the circuit, one can guarantee that the number of cuts is minimized. Unfortunately, this scales poorly with the number of qubits and desired partitions. Approximate solutions are sufficient for the use case; if the procedure is nondeterministic, it can generate multiple initial solutions. Cutting continues until each subcircuit fits on a device. One can enforce additional cuts if most devices are significantly smaller than the largest. As a second approach, the qubit count of the devices governs the cut choices. I select the device randomly from a uniform distribution; weighting by size can be beneficial. After generating the partitions, the scheduler requests the cuts from the circuit generation component.

7.4.1.2 Generation

The central part of the scatter search algorithm is the generation of new solutions. It consists of two procedures: Diversification and intensification. *Diversification* is the generation of new solutions that are different from the current ones. I achieve this by

swapping two jobs on the same machine or a different machine. With this approach, the algorithm explores local solutions by swapping jobs on the same machine and global solutions by swapping jobs on different machines. A dummy job is present on any machine during all time steps to move a job to a new position in the schedule. A parameter controls the number of swaps, which, in combination with the population size, determines the diversity of the solutions.

Intensification is the improvement of the current solutions. The goal of this phase is to improve schedules by using domain knowledge. For this, the algorithm tries to reduce the schedule of the machine with the highest completion time. A timeslot is selected, and all active jobs are redistributed to other machines. This rescheduling is done randomly and can be biased toward machines with smaller completion times. One could also use the setup and processing times information to optimize this step further. Calculating the setup times is expensive and can only be done at specific steps to save time. Evaluating the processing times beforehand and cheaply looking it up during the optimization also improves performance. These times are so similar in praxis that the optimization does not yield significant improvements. Additionally, more cuts are possible during this phase, but this would require a recompilation of the circuits. Therefore, I will not consider this in the first iteration of the algorithm, but it could be a promising extension.

One important consideration is to allow invalid solutions instead. In such solutions, a job could exceed the available resources of a machine. In theory, this relies on the possibility of moving jobs to other machines at a later point to fix the constraint violation. In practice, the majority of randomly generated solutions are invalid. During intensification, a second step could fix violations in promising candidates. I chose to turn off this feature, as it yielded little results in initial tests.

7.4.1.3 Evaluation

At each round of the algorithm, the best solutions persist based on multiple metrics calculated for each. A forward pass through the schedule is necessary to determine the successor relationship. The initial metric is the makespan c_max, which the algorithm calculates for all machines in the schedule in parallel. Other metrics make the algorithm suitable for HPC. For example, I will provide a metric that considers priorities and device selection and a metric including expected noise in Section 7.5. The procedure is extensible to any other metric related to the schedule without outside information. Calculating the makespan could be sped up by dynamic programming, but the number of jobs is usually small enough to make this unnecessary.

The evaluated solutions then form the starting population for the next iteration of the algorithm. It retains the top n solutions and adds diversity by selecting maximally different solutions. This selection uses a distance metric between two schedules S_1, S_2, inspired by the Hamming distance as follows:

$$d(S_1, S_2) = \sum_{m \in M} \sum_{I_m} d(i_{S_1}, i_{S_2}) \,,$$

$$d(i,j) = \begin{cases} |b_i - b_j| & j \in I_m \\ \max(\|I_m\|_\ell, \|J_m\|_\ell) & \text{otherwise} \end{cases}.$$

The metric compares all jobs $I_m = \{\text{circuit } i : i \text{ is assigned to machine } m\}$ of the schedules in the population. If the jobs are assigned to the same machine in both schedules, the distance is the difference in start times. If the jobs are not assigned to the same machine, the distance is the maximum of the makespans of the two relevant machines. I calculate the machine makespans $\|I_m\|_\ell = \max_{i \in I_m} c_i$ once for this.

7.4.2 Reinforcement Learning

A second approximation approach is to use reinforcement learning. In contrast to the scatter search, reinforcement learning is a model-free approach. The algorithm learns the optimal policy by interacting with the environment. For this, the scheduling problem must be encoded so the algorithm can interact with it. I can reuse multiple concepts from the scatter search, but the interaction with the cutting component behaves differently. Instead of initializing multiple solutions, the algorithm starts from the baseline schedule.

I use the *Proximal Policy Optimization* [140] algorithm for this approach. PPO is a policy gradient method known for its stability, good performance, and simplicity. A learning agent interacts with an environment to maximize the expected reward. The scheduling problem is, therefore, encoded in the environment. This environment is similar to the scatter search; multiple devices are available with different capacities and queues. Additionally, the access has access to each device's makespan of the current schedule. As actions, the agent can manipulate the schedule in four ways:

- *Partition* a circuit i by requesting a cut of size n. The circuit generation component then estimates the cut. Resulting subcircuits join the schedule at the end of the queue.
- *Move* a circuit i at t_i on machine m to a different machine m' at time $t_{i'}$. Moving can invalidate the schedule, but the agent can fix this with a later action.
- *Swap* two jobs on any machine and any timestep.
- *Terminate* the process and evaluate the schedule immediately.

Once the agent has acted, the environment updates by recalculating the schedule's makespan. This update is similar to the evaluation step in the scatter search, the only difference being that only one solution remains at a time.

One main advantage of reinforcement learning is optimizing multiple objectives simultaneously. The agent can optimize the makespan, the noise, or a combination thereof, depending on the incentive provided by a reward function. Since this is a minimization problem, the reward function is negative. I define the reward function as a weighted sum of the makespan and the noise:

$$R = -(\mu \cdot \mathbf{P}_{\max} + \nu \cdot \mathcal{F}). \tag{7.3}$$

$\mathbf{P}_{\max}$ is a custom cost function similar to the makespan (explained in Section 7.5.1) weighted by μ. The second term is the noises in the schedule, weighted by ν. To facilitate learning, the reward of a period is biased toward shorter periods. For each action that is not an improvement, the reward decreases by a small fraction. When deciding to terminate, a high negative reward penalizes the agent if the schedule is infeasible.

The final problem is the size of the action space. Proximal Policy Optimization is typically not used for variable action spaces. Allowing the action space to change would be necessary because the circuit cutting produces more circuits and, therefore, more actions. A standard solution for this would be to either find an encoding of the environment using a fixed-sized projection or using action masking. For technical reasons, I implemented a simpler scheme. I prepopulate the devices with empty circuits to fix the action space. Cutting then replaces some of the empty circuits with the newly generated circuits.

Reinforcement learning is a powerful tool that requires a lot of data to train. It offers an alternative approach to tackling the cutting and scheduling feedback loop. Due to the size of the action space, it can quickly reach its limits. The potential for scheduling is clear, but the implementation needs further refinement.

7.5 Technical Considerations

The above approaches can form one component in the toolchain. `MILQ` [116] and `SCIM MILQ` [117] are two prototypes that implement scheduling techniques. Appropriate integration is necessary to fit the HPC systems requirements. This section shows how I incorporate the scheduling into the toolchain. I will also discuss the available features and the interaction with other components. Figure Figure 7.5 shows the entire structure of the scheduler.

I will consider an example of two QPUs, which I also used for Figure 7.4a to describe the functionality. While the exact functionality of the devices is not relevant, I assume that the properties of the devices are available through a query. The respective interface I define in Section 7.5.2. Based on the assumption that the devices have different properties, the scheduler works best; if the devices are similar, the scheduler still works, but alternative options might be preferable. Additionally, as before, I assume that compilation consists of two steps. First before the scheduling without explicit knowledge of the devices and second after the scheduling with device information. After the initial compilation, `MILQ` handles the circuit.

Before scheduling a circuit, two mechanisms can trigger. Backfilling allows bypassing the scheduler completely. If a device has a free slot fitting the circuit characteristics, the circuit executes there immediately. The circuit submitter can limit the devices that allow this, restricting backfilling; hence, queueing times increase. Typically, this works well for jobs with low resource requirements. Algorithms that require continuous interaction with classical hardware can also interact differently with quantum devices. For example, variational algorithms like VQE should not queue again for every set of parameters. Instead, the scheduler can reserve a device for a fixed time; the device is unavailable for

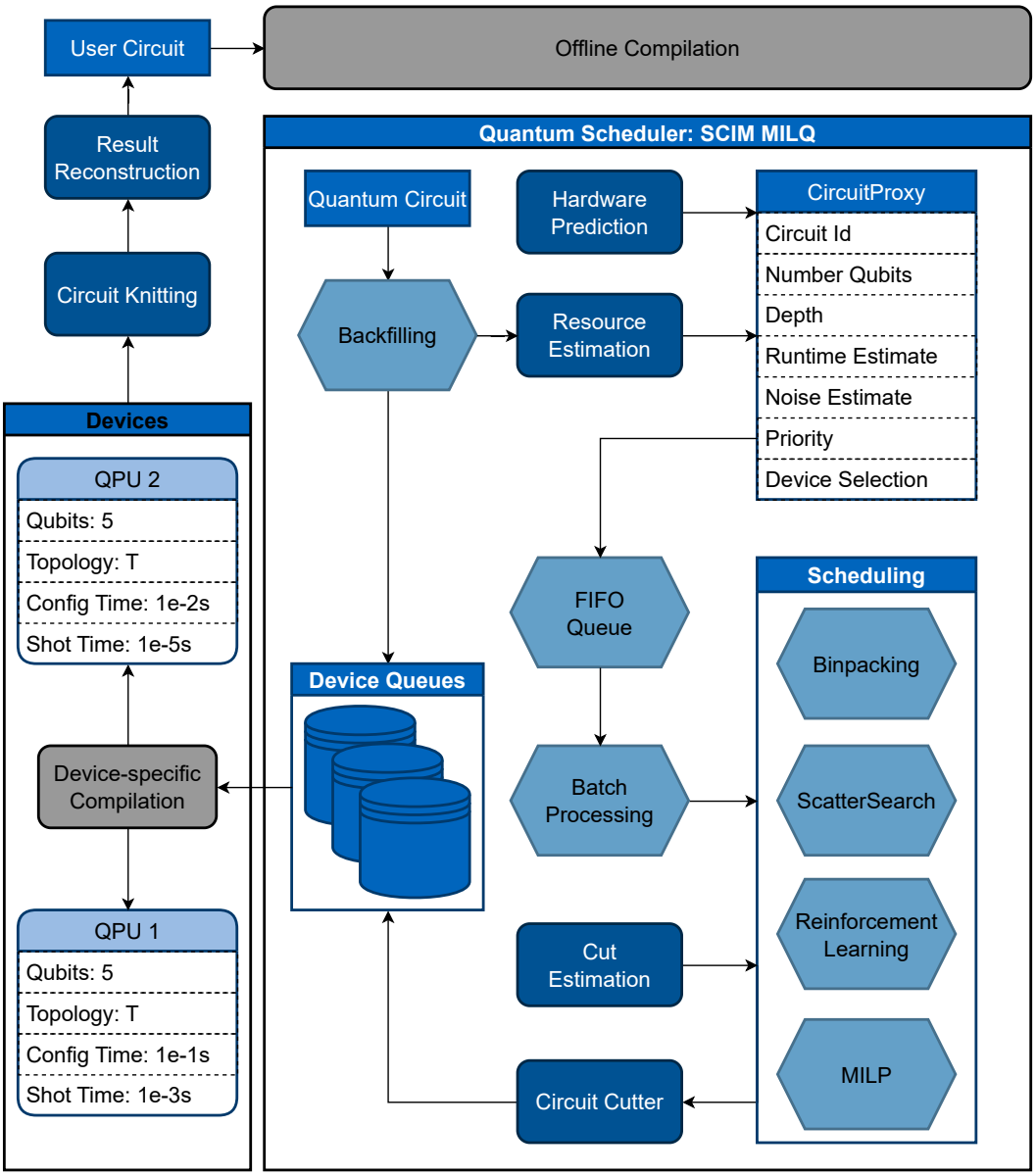


Figure 7.5: Structural overview of the implemented scheduler. It includes multiple features, such as cutting, resource estimation, and reconstruction. Devices are not part of the design but are shown for completeness. Figure adapted from [117].

other jobs, and its queue is suspended. Such possibilities result in administrative choices configurable by the platform provider.

Other circuits pass through a more elaborate process. First, `MILQ` converts the circuit to a proxy object. Proxies contain all the information necessary for the scheduler to make a decision. The advantages of this are twofold. The backend interfaces must be queried only once (one can cache results), and `MILQ` must transfer only partial circuit data to the scheduling components. Besides circuit-specific information, such as qubit count and depth, additional estimates, e.g., for noise, complete the information. Estimations exist for runtimes, setup times, resource requirements, noise, and hardware choices. I cover the exact concepts in Section 7.5.1.

After the conversion, the proxies stay in a queue and are transferred to the scheduling algorithms in batches. `MILQ` contains the four scheduling approaches discussed in this chapter. Some of the algorithms could decide to cut proxies. These cuts are also estimated by looking up the original circuit. For the resulting proxies, computationally cheaper estimates, based on their parent's values, are used. Once the scheduling phase finishes, a reconstruction of the schedule is necessary. `MILQ` then converts the proxies in the schedule back to circuits associated with devices and queue positions. Circuit generation is responsible for cutting the circuits according to the schedule. It is also possible to run circuits simultaneously, which requires a combination into a single circuit for some devices. Circuit generation happens while the circuits wait in the device queue as part of the device-specific compilation.

7.5.1 Proxy Estimation

A problematic part of scheduling is knowing circuit properties in advance. Calculating the exact values is expensive, especially when repeated for subcircuits, but can be avoided. To circumvent this issue, I propose a two-part procedure. Expensive estimations are done only once at the beginning of the scheduling. `MILQ` caches these estimates in proxy objects and reuses them for scheduling. An overview of the parameters and their description I summarize in Table 7.4. During the scheduling, cheaper estimates are used based on the depth ratio of the subcircuits. For this, a reference to the original circuit information is necessary:

$$d_{\text{ratio}} = \frac{d_i}{d_{Id_i}}\,. \tag{7.4}$$

This procedure becomes necessary after cutting. The scheduler needs this information to schedule the new subcircuits. Repeatedly requesting these values through an interface can quickly become a bottleneck; hence, I choose this simplification.

7.5.1.1 Runtime Estimation

Estimating the runtime of a task is at the core of scheduling algorithms. The more accurate the estimation, the better the scheduler can make decisions. Both processing and setup time are of interest in the described setting. Unfortunately, hardware providers

Table 7.4: Parameters of interest for a proxy.

Parameter	Description
	Circuit Parameters
Id_i	UUID of the original circuit job i belongs to
q_i	Number of qubits of job i
d_i	Circuit depth of job i
	At-Runtime System Queries
$\mathcal{T}_i$	Device prediction
$p(i,m)$	(Estimated) Processing time of job i on m
$s(i,j,m)$	(Estimated) Set-up time between job i and j on m
$f(i,m)$	(Estimated) Noise of job i on m

usually do not disclose this information; one must confirm experimentally. For the processing times, the critical path of the circuit combined with the average gate length is a good indicator. The problem is that the circuit must be optimal for the hardware to calculate the critical path accurately. It can become expensive to repeat compilation for each hardware-circuit combination. Alternatively, the Azure resource estimator [118] provides a rough runtime estimate. These estimates provide a consistent level of accuracy; hence, they are suitable as the basis for comparison. Additionally, resource estimates for fault-tolerant operations, e.g., T-States, are already included. These can contribute to setup time and their estimation.

The majority of the setup time is still the compilation. Especially optimization passes can take significant time. The most considerable accelerations result from caching compilation and skipping passes, which I showed in Chapter 6. I analyzed the case for parametrized circuits and applied it to circuit cutting. After cutting, the resulting circuits differ only in a few gates, whereas the rest of the optimization is still valid. I model this by adding a step function, which only applies to circuits that undergo a complete compilation. The second factor of setup times is hardware-dependent. As mentioned, physically setting up the hardware takes a fixed amount of time. In the examples of atom trap layout [122], this can depend on the circuit, but generally, it is on the same magnitude..

During scheduling, `MILQ` only requests expensive calculations for the initial circuits. All successive circuits resulting from cutting are estimated based on the depth ratio d_{ratio}. This ratio is stored in the proxy objects, significantly reducing computational overhead.

7.5.1.2 Noise Estimation

The noise estimation functions on a similar principle to the runtime estimation. At the start, `MILQ` estimates the circuit's noise with high accuracy and succinctly reused for ratio-based, cheap estimates. For the initial estimation, the device's noise model is used.

I use Mapomatic [141], which uses the coupling graph of the device to estimate the noise. The coupling graph is available through the device interfaces and matched with noise data from calibration intervals. A graph isomorphism between the circuit connectivity and the device is the basis for the noise calculation. By default, Mapomatic evaluates all possible isomorphisms, which can be computationally expensive. To limit runtime, I restrict the number of isomorphisms to a fixed number.

A drawback of this approach is that it limits circuits to the device's size. If the circuit is larger, the noise estimation is not possible. I propose to solve this with additional estimations, which can give cut estimates according to the device size. The average noise can be sampled and extrapolated from the resulting subcircuits to the complete device size. Finally, to get a single value for the noise over multiple devices, I use

$$f(j) = \max_{m \in M} f(j, m) \cdot d_{\text{ration}} \,. \tag{7.5}$$

7.5.1.3 Cutting Estimation

The cutting estimation consists of sampling overhead and the increased number of circuits. Since the overhead is exponential in the number of cut gates, the number of cuts should be minimal. Optimally partitioning the circuit is necessary for this. In my work, I implement a brute-force search solution, which is usually infeasible for complex circuits. Using the compilation techniques I developed in Chapter 6, I decrease the number of cuts, which means my brute force approach, therefore, remains valid for the short term. In the long term, a more sophisticated approach is necessary. The cost is an upper bound to the actual overhead, as I calculate it based on CX gates, which are more expensive than the other gates.

The cost for the additional circuits is just due to the increased processing time. As already mentioned, most of the additional costs are due to compilation. A unique compilation step is necessary, which treats the subcircuits differently. In such circuits, every cut gate is replaced by the quantum channel $\mathcal{F}_i$, which is then substituted. Instead of facing exponential numbers of additional compilations, only one for each partition is necessary. This result highlights the importance of the two-step compilation process and the techniques I developed for circuit cutting.

7.5.2 HPC Interfaces

The scheduler already includes techniques for HPC systems like backfilling and batching. But a full integration with the software stack is necessary, to ensure performance. For testing, I implemented a prototype in Python as a reference. The final implementation is written in C++ and uses high-performance libraries. It also has to respect the interfaces of other components as defined in the *Unified Toolchain* (c.f. Chapter 4).

The first goal is to provide a simple interface for the scheduler, that can be easily replaced by other implementations. Therefore it is implemented as a dynamically loaded runtime library, which exposes a single function.

```
int scheduler(Device2SubmitterType device2Submitter,
    std::vector<QuantumTask> *tasks)
{
    std::vector<QDMI_Device> devices
        = FOMAC_available_devices(false /*verbose*/);
    // Implementation
    childTask.parentId = id;
    childTask.mScheduledQpu = device;
    return 0;
}
```

Listing 7.1: Interface of the scheduler.

However, full integration with the software stack is necessary to ensure performance. For testing, I implemented a prototype in Python as a reference. The final implementation is written in C++ and uses high-performance libraries. It also has to respect the interfaces of other components as defined in the *Unified Toolchain* (c.f. Chapter 4).

The first goal is to provide a simple interface for the scheduler that other implementations can easily replace. Therefore, I implemented it as a dynamically loaded runtime library that exposes a single function.

It consumes a list of quantum tasks and sets the scheduled device for each task. Communication with other components occurs through querying a standard interface. These interfaces are similarly defined and implemented as runtime libraries. For example, the figure of merits and contsraints (FOMAC) interface allows to query devices; a device reserved for a tightly interacting job will not appear. Devices also provide regular interfaces for direct access; FOMAC resolves more complex queries if necessary.

The scheduler sits between the circuit generation and the device-specific compilation in the toolchain. I thoroughly explained the complexity of the cutting and the scheduling interaction. To faithfully reconstruct a circuit's results, the `parentId` property must match the results. Cutting instructions generated by the scheduler are resolved by the generation component. After scheduling, the circuits continue to the device-specific processes. While in the device queue, final optimizations still occur. The scheduler stores the partition information to reduce the compilation overhead (as discussed in Section 7.5.1.3). Combined with the `parentId`, the compiler can infer which circuits belong together.

My scheduling technique also works in combination with other schedulers. Scatter search uses OpenMP [142] for on-node parallelization and can quickly improve existing schedules. Before setting a final schedule, a reduced loop of scatter search can quickly check for local optimizations. For example, supervised learning approaches profit from this as the improved results are valuable for re-training. The proxy objects' information is available through the device interfaces and reusable by other components. I plan a similar solution for resource estimation, but it is unclear if it will be implemented separately or as a global service for each device. The scheduler can easily interact with the interface as long as it is uniform.

A second performance-relevant feature is caching. Saving computational expenses wherever possible is a standard performance optimization. Reusing the compilation for cut circuits is one example. A similar scheme applies to variational algorithms and parametrized circuits. Some of the information in the proxies is cacheable; noise and runtime estimates should not vary significantly between runs. Also, the scheduler can reuse prior hardware selection to seed the next run if the non-exclusive device selection is enabled. Automatic data parallelism is a feature that can speed up variational algorithms: Running multiple instances of the same circuit with different parameters can speed up optimization. An open question is how different hardware characteristics and even calibrations can affect the algorithm[1].

The generated metadata is valuable information for other components during the scheduling process. For example, the device selection can influence the compilation pass selection of future circuits similar to the scheduled ones. Execution data of circuits can (in)validate the estimated values, and estimations improve when incorporating prior data. The scheduler will play an even more important role in highly distributed systems. A scheduler must assign data to the correct devices and consider complex interactions between components. This complexity tends toward the direction of gang scheduling, where in QC, a scheduler must consider state preparation, entanglement distribution, and qubit conversion. Some techniques might need refinement in light of error correction, but the general approach is still valid. The ultimate goal is full integration into existing HPC systems, where QPUs are another resource available to the scheduler. Even automatic decisions to delegate computation to quantum hardware seem reasonable.

7.6 Preliminary Results

I test the prototype implementation of the scheduler on a set of artificial benchmarks. A setup, as I describe in Figure 7.4a, does not yet exist in real hardware. The current hardware only includes a single quantum device; hence, I resort to simulators. A benefit of this approach is to evaluate hardware configurations and test their suitability. This assessment shows the approach's feasibility and provides a baseline for future implementations. The integration into the toolchain coincides with the hardware upgrade, in which a heterogeneous setup is planned. The high-performance solution will be available and re-evaluated in a real-world scenario.

7.6.1 Target Metrics

I evaluate the scheduler on multiple target metrics. The primary metric is the makespan, which translates to a wall clock time. For this, let $\mathbf{M} = \max_{j \in J} c_j$ be the makespan, the maximum completion time of all jobs, which is measured in seconds. Makespan is the most straightforward metric, and I use it for all benchmarks. I also evaluate a custom metric consisting of multiple factors for the heuristic approaches.

$$\mathbf{P_m} = \ell(m) + \max_i(c_i \cdot \rho_i \alpha + \delta_{i,\tau_i} \cdot \sigma_i \beta) \tag{7.6}$$

[1] A supervised student work showed that under given circumstances, no adaption is necessary.

$$\mathbf{P}_{\max} = \max_{m \in M} \mathbf{P_m} \tag{7.7}$$

The completion time of a job scales with the priority ρ_i and a configuration parameter α. Before taking the maximal value, I evaluate whether a circuit runs on the device specified by the user with the delta function δ_{i,τ_i}. The result is weighted with the user-defined strictness for hardware decisions σ_i and a configuration parameter β. Finally, the queue length $\ell(m)$ is added to the completion time. For all the machines, I then compare the values of $\mathbf{P}_{\max}$ As a final metric, I also evaluate the noise of the schedule. Let $f(i, m)$ be the noise of a job, for example, as calculated by the circuit fidelity $\mathcal{F}(\mathcal{E}, j)$ (c.f. Equation (2.6)). Then, the total noise of the schedule is $\mathbf{F} = \sum_{m \in M} \sum_{i \in J} f(i, m)$.

7.6.2 System Configuration

I use the MMQT `Bench` [110] benchmark suite to generate circuits. I use randomized circuits with an optimization level of 0.

MILP

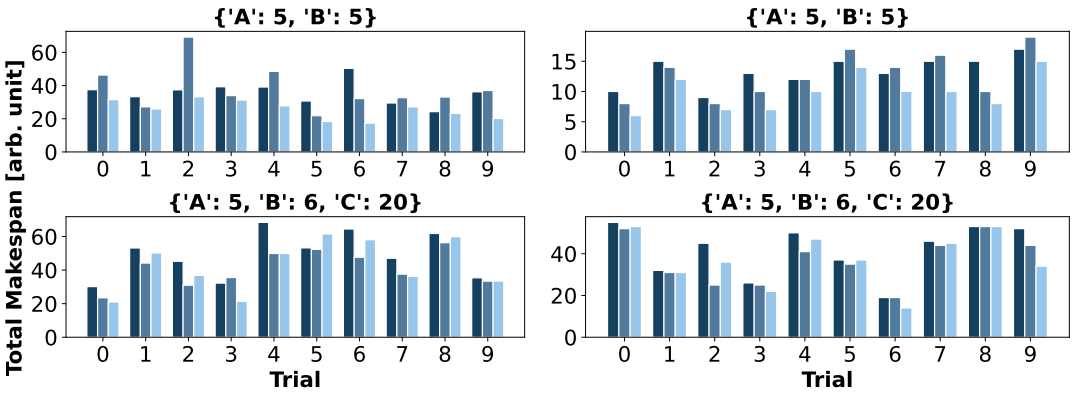
For the benchmark of the MILP approaches, I use the *simplified* formulation without sequence-specific setup times and *extended* formulation. I simulate ten batches of circuits with seven circuits each. The number of qubits is limited to the size of the largest device, which simulates the situation after circuit cutting. Processing times and setup times are artificial. Additionally, I compare the results for real-valued and integer-valued processing and setup times. I examine two scenarios, one with two devices of size five and one with three devices of sizes (5,6,10). The results can be seen in Figure 7.6, only the makespan is evaluated. [116]

Heuristics

For the heuristic approaches, I turn off backfilling and exclusive access. The system configuration is listed in Table 7.5. The generated circuits are limited in size to the sum of all available qubits. I evaluate a scenario with two devices of size (5,7) with identical setup times and one with three devices of size (5,5,7). If a circuit exceeds the size of the largest device, the scheduler cuts it faithfully into smaller circuits. The user-specified parameters ρ_i, $sigma_i$, and τ_i, and, the device queues $\ell(m)$, are generated randomly. The heuristic optimization loop runs for 128 iterations on 16 cores. During diversification, the number of swaps is set to at least 5 and weighted with a random factor governed by the number of available jobs. I use the Proximal Policy Optimization algorithm provided by *Stable Baselines* for the reinforcement learning approach. It trains on 5000 iterations before the evaluation. For this, I use a reward function, a linear combination of the makespan and the noise

$$R = -(\mu \cdot \mathbf{P}_{\max} + \nu \cdot \mathbf{F}) \,. \tag{7.8}$$

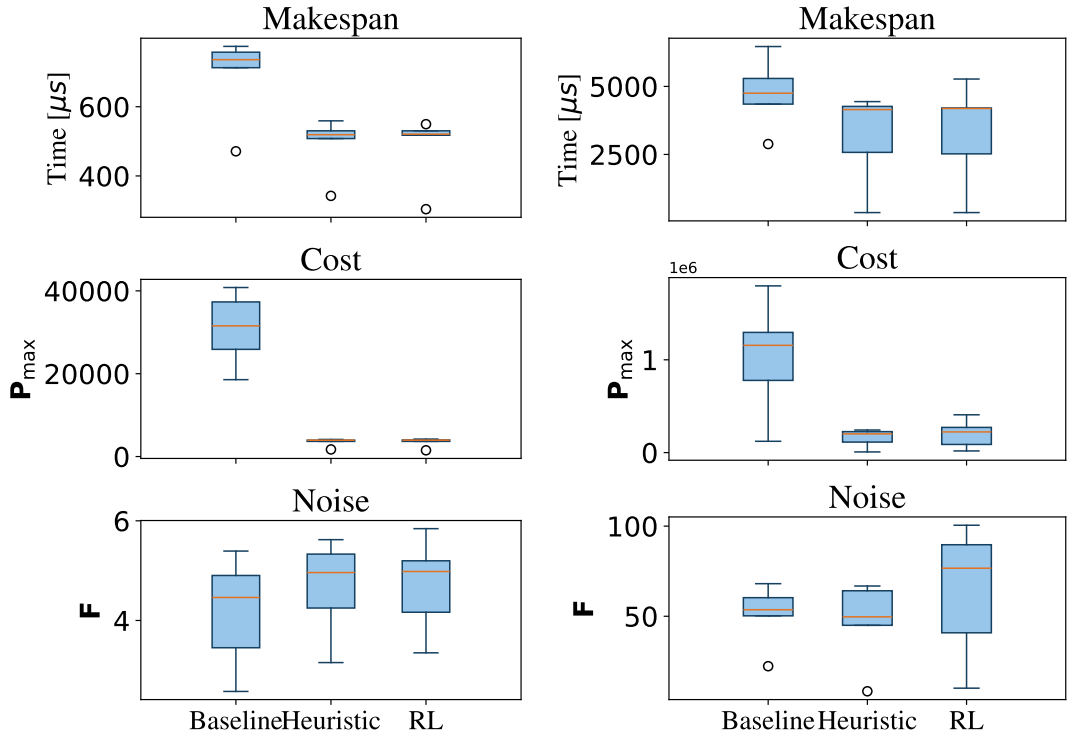
Figure 7.7 shows the results.



(a) Results for real-valued s_{jm} and p_{jm}

(b) Results for integer-valued s_{jm} and p_{jm}

Figure 7.6: Makespan results for the three configurations *baseline* (darkblue), *simple* (blue) and *extended* (lightblue) in two different settings according to [116].



(a) Results for setting with machines of size 5 and 7.

(b) Results for setting with machines of size 5,7 and 5.

Figure 7.7: Results of the target metrics for the heuristic and reinforcement learning approach compared to the baseline. Settings are defined by the number of available devices according to [117].

Table 7.5: Default scheduling configuration used for the experiments.

Parameter	Value
Batch size	5
α	1
β	1
μ	0.5
ν	5

7.6.3 Discussion

Overall, the improvements in scheduling are apparent. Especially considering that the baseline algorithm is already an improvement over existing approaches. In the makespan optimization, my approach performs robustly over all scenarios. It accounts for the number of devices, differing processing and setup times, and circuit variations.

In the initial setting of Figure 7.6, I compare the baseline against the exact scheduling approaches. The *extended* model shows an average improvement of 25 % in the setting with two devices and integer timing properties. Using real-value configurations shows a marginal decrease in performance, but the more considerable variance is due to the device size. In such settings, makespan is still reduced but only by 11 % on average. A likely reason for this is that one of the devices is significantly larger than the rest. The baseline profits from this imbalance since it can schedule most jobs to the large device. Indirectly, this also confirms the heuristic not to cut circuits if possible. A significant drawback of the *extended* model is its considerable performance cost. Measured in wall clock time, it is six magnitudes slower than the baseline, even for reasonably sized problems. The *simple* model is faster by two orders of magnitude, but the results are worse. Device size differences, in particular, potentially lead to gross overestimates of setup time, resulting in cases where the baseline outperforms the *exact* model. A second conclusion is the viability of the baseline algorithm. It performs surprisingly well in terms of simplicity and the ratio of makespan compared to runtime. This result aligns with classical scheduling, where simple approaches often trump complex systems.

The lack of performance of the exact models is the motivation to provide alternative solutions. The heuristics are, at worst, one order of magnitude slower than the baseline while including cutting decisions. Most of the runtime increase results from the cutting itself, not the scheduling. Makespan results in Figure 7.7 show a reduction of makespan, which is similar to the original models, up to 25 %. Due to my construction of the algorithm, the scatter search cannot be worse than the baseline. The reinforcement learning agent provides only minor improvements, hinting that more training might be necessary. In favorable settings, the agent can improve the noise by 10 %, but this depends strongly on the quality of the initial solution.

The most significant improvement is made in the custom cost function $\mathbf{P}_{\max}$. On average, scatter search lowers the scores by 80 %. This result is realistic since the baseline algorithm does not consider this score. My implementation clearly shows how to cheaply

improve custom losses with only a minor cost overhead. Also, these results show how the scheduler can resolve the interaction with circuit generation. By requesting cuts without executing them, it can find quality solutions. The optimization of the cuts itself is also better suited to the generation components. Further, my results show the importance of accurate estimations. Currently, they are provided at low resolution during most of the scheduling process. With better estimates, the scheduling should improve as well.

What remains open is the performance with multiple users and devices. Making the scheduling aware of algorithms and optimizing for use cases are further improvements. Noise is a big concern; transitioning to the error-corrected paradigm might solve this eventually. On the technical side, gang-scheduling approaches and embedding in an established scheduling solution are the following essential steps.

8 Error Correction

So far, I have investigated techniques in scheduling and compilation which fit the current hardware limitations. However, fault-tolerant quantum computing will inevitably pose new challenges. In this chapter, I want to analyze the impact of error correction on the software stack. Compared to the previous chapters, this will be more speculative; implementing large-scale, error-corrected operations is a tremendous challenge. To reason about fault-tolerant computation, providing sensible experiments is necessary. Hence, I will describe my work on design automatization for error-corrected experiments.

One distinguishes two qubits for error correction: *logical* and *physical*. Quantum algorithms typically consider *logical* qubits; generally, one can assume that errors are absent. *Physical* qubits constitute *logical* qubits since they are qubits realized by the hardware through physical systems. In NISQ, there exists a one-to-one correspondence between *logical* and *physical* qubits. Any error directly impacts the computation, which can render the computation useless depending on the error rate. As in classical coding theory, multiple *physical* qubits can encode *logical* qubits. Depending on the selected code, this is an n-to-k mapping, typically $n \ggg k$. A code is usually described by the number of *physical* qubits n, the number of *logical* qubits k, and the distance d as a triple $[n, k, d]$ (see Nielsen and Chuang [8] Section 10.4). The distance d describes the Hamming distance between two codewords, the minimum number of errors occurring to transform one codeword into another. In other words, the distance describes the length of an error chain that the code can no longer correct.

The versatility of error correction codes is vast, and many codes exist. Depending on hardware limitations, researchers commonly employ two approaches to achieve fault tolerance. Given enough (potentially low-quality) qubits n, even high overhead codes are suitable to realize fault tolerance. High qubit numbers are typical for hardware where scaling system size is more feasible than improving qubit quality, most common in superconducting qubits. Codes in this group are typically less complex, such as the Steane code [143], color codes [144], or surface codes [145]. The second approach uses a few high-quality qubits in combination with a complex set of unique operations not available in other hardware. Resulting codes are typically bespoke, such as the Tessaract code [146] for trapped ions. Both options have a common challenge, namely the complexity of the operations. Simply maintaining the state of a logical qubit requires a large number of physical qubits and operations. An example of the required operations of a $[9, 1, 3]$ surface code is shown in Figure 8.1; I will discuss the operations in more detail in the following section. Manually scaling up such a system becomes tedious and error-prone. As a result, prototyping new codes and realizing new experiments are significant challenges. Especially considering that even apparently simple operations are not trivial to implement. Simply maintaining the state of a logical qubit requires a large

number of physical qubits and operations. A slice of a so-called memory experiment is depicted in Figure 8.2.

With my work[1], I aim to provide a tool to automate the design of experiments for error correction. My efforts are part of an open-source software project called `tqec`, which aims to accelerate the development of error correction codes. The goal is to be able to define arbitrary topological computations, optimize them, generate circuits associated with different code distances and plaquettes, and, finally, simulate the results. Achieving this is beyond the scope of an individual, and `tqec` is developed by multiple contributors. In the context of this thesis, I will focus on topological error correction to express its challenges. The impact of error correction on the toolchain concludes this chapter, where I will discuss the fundamental pieces toward a fault-tolerant software stack.

8.1 The Surface Code

In topological error correction, qubits conceptually exist on a toric surface [147]. Using specific operations, errors on these qubits form loops on the surface, which measurements can detect. The *surface code* is a specific realization on a 2D lattice, removing the need for special topological features [145]. Fowler et al. [145] show that the threshold error rate is around 0.75%per operation for such lattices. Combined with the fact that most quantum hardware exposes at least nearest neighbor connectivity, the surface code is a popular choice for error correction. It exhibits excellent properties, such as error locality and easy decoding, which are additional benefits.

The surface code operates based on the concept of stabilizers. A stabilizer group $\mathcal{G}$ is a (sub)group of operators that leave a state invariant for all elements $x \in X$. Formally

$$\mathcal{G}_x = \{g \in \mathcal{G} | g \cdot x = x\}\,. \tag{8.1}$$

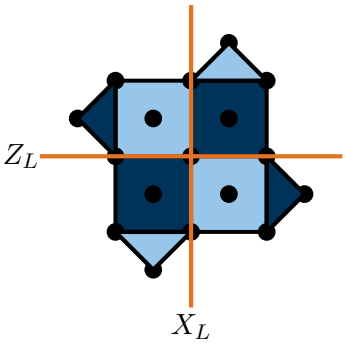
For most quantum error correction codes, the Pauli operators X and Z and their signed tensor products form the stabilizers. For example $Z \otimes Z = ZZ$ is a stabilizer for the state $|00\rangle$, but also $\frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)$. The essential properties of stabilizers are the commutation relation with each other and the resulting change in the sign of the eigenstate. An anticommuting error $\mathcal{E}$ of a stabilizer A, will flip the sign of the stabilized state

$$A\,|\Psi\rangle = |\Psi\rangle \xrightarrow{\mathcal{E}} \mathcal{E}\,|\Psi\rangle = \mathcal{E}A\,|\Psi\rangle = -A\mathcal{E}\,|\Psi\rangle\ . \tag{8.2}$$

Such sign changes can be detected by measuring the stabilizers and recorded as a so-called *syndrome*. Combining multiple syndromes allows one to reason about the error that occurred and, therefore, keep track of the most likely error. Decoding means finding the most likely error given the syndromes, which I discuss in Section 8.1.2.

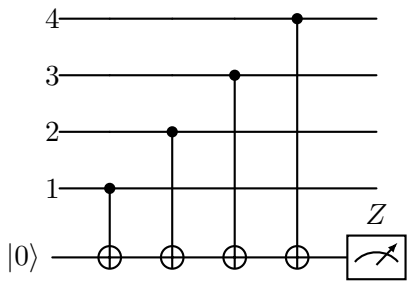
The surface code is a specific realization of a stabilizer code, focusing on weight-4 stabilizers; on boundaries, they are truncated to weight two but work virtually the same. They take the form of $I^{\otimes n}KKKKI^{\otimes m}, \quad K \in \{X, Z\}$, which are visualized as

[1] At the time of writing, the work has not been uploaded to a preprint server. Additionally, an accompanying work describing the theoretical background is being prepared.

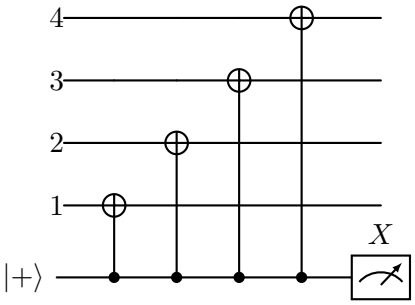


(a) Diagramatic representation of the Surface code. Dark blue patches (called plaquettes) are the X stabilizers, light blue plaquettes are the Z stabilizers. X_L and Z_L are the logical operators associated with the code.

(b) Order of operations on the respective plaquettes. Plaquettes on the boundary follow the same order just with less qubits. The qubit in the middle of a plaquette is called *measure qubit*, the others are *data qubits*.



(c) Phyical realisation of one round of error correction on a single Z plaquette.



(d) Phyical realisation of one round of error correction on a single X plaquette.

1	5	6	2
7	9	10	8
8	10	9	7
3	6	5	4

(e) Template structure of the surface code as used in `tqec`. A template consists 1×1, $2 \times k$, $k \times 2$ and $k \times k$ rectangles which can be populated by plaquettes according to the pattern indicated by numbers.

Figure 8.1: Example of a so called memory experiment in the $[9, 1, 3]$ surface code. The code is defined on a 2D lattice, where each qubit is connected to four neighbors. The logical qubit is encoded in the lattice, and errors are detected by the stabilizers.

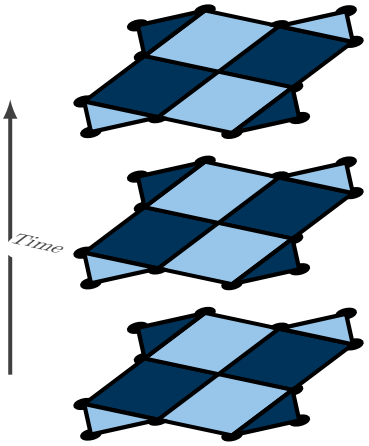


Figure 8.2: Continues round of syndrome measurement and correction in the $[17, 1, 3]$ surface code. The measurements have to be repeated $k = 3$ times to correct on error on the time axis, too. The constant k is often referred to as number of cycles or rounds.

squares on the lattice for example in Figure 8.1a. This visual representation is called a *plaquette* and allows for more straightforward reasoning about the stabilizers. Each plaquette is associated with a X or Z stabilizer, measured in a round of error correction. The circuits for the stabilizer measurement are shown in Figure 8.1c and Figure 8.1d, respectively. An essential property of the plaquettes is the order of operations (shown in Figure 8.1b), which is necessary to prevent the propagation of errors. The locality of the stabilizers makes the surface code a low-density parity check (LDPC) code, which has favorable characteristics [148]. As stabilizer codes function based on Pauli operators, simulating them can be done classical, although only without implementing computation. Additional resources are necessary for computation to yield a quantum advantage and enable the full computational spectrum. The related operations and their realizations I will explain in the following section.

The main reason for the order of operations is to prevent errors from spreading throughout the circuit. As explained in Section 2.2.1.3, the relevant errors can be expressed as the combination of X and Z errors. Each stabilizer commutes with one type of error and anti-commutes with the other. Hence, all combinations of such errors will be detectable. The order of operations is crucial so errors can not spread through the circuit. For example, in Figure 8.1b, merging the two plaquettes at *any* edge would result in an explicit ordering of operations. Both measurements of one plaquette will be completed before the other on their respective data qubits. Problems can arise when errors occur on the measurement qubits; such errors are called *hook errors*. Since both measurement circuits are typically implemented with CX gates, the propagation of X and Z operators through the gate has to be carefully considered. The following rules specify the propagation:

$$X \otimes I \xrightarrow{CX} X \otimes X \tag{8.3}$$

$$Z \otimes I \xrightarrow{CX} Z \otimes I \tag{8.4}$$

$$I \otimes X \xrightarrow{CX} I \otimes X \tag{8.5}$$

$$I \otimes Z \xrightarrow{CX} Z \otimes Z \tag{8.6}$$

Colloquially, the gate copies an X operator from the control to the target qubit, while the Z operator moves from the target to the control qubit. Due to the construction of the plaquettes, a problem could occur when errors in its measurement qubits spread to neighboring plaquettes. If the error does not spread orthogonal to the respective logical operator, it will affect two plaquettes that are not adjacent. Such an error would be detected but decoded incorrectly since more likely explanations exist. The wrong orientation of plaquettes could, therefore, effectively halve the distance of the code; hence, aligning the plaquettes is crucial for the code's performance. In the current example, the order of operations is correct, but generally, this needs to be checked for each code realization.

So far, I have discussed the surface code based on the specific stabilizer implementation called the rotated surface code [149]). However, the surface code is a family of codes that can be realized with different plaquettes. For example, the so-called $XZZX$-surface code [150] uses only one type of plaquette, which has advantages for biased noise. Experiments with different implementations of the surface code are currently handcrafted, from building the circuits to analyzing the results. When scaling to higher distances, the complexity of resulting circuits proliferates, and running extensive experiments on hardware becomes increasingly expensive. The goal of the `tqec` project is to automate this process, allowing for a more systematic approach to error correction. The core concept revolves around the independence of computation operations from code realization. Templates (shown in Figure 8.1e) represent a computation that instantiates circuits with different code distances and plaquettes. As a user, this significantly simplifies the experiment design. The computation is defined once, and the tool automatically generates different stabilizer circuits and code distances. `tqec` provides the necessary tooling interfacing with stabilizer simulators and programs to define the computation.

8.1.1 Operations

Using the stabilizers, one can ensure that the logical qubit encoded is in the simultaneous eigenstate of all stabilizers. Conveniently, stabilizers also describe the current state, i.e., $|0\rangle$ is the $+1$ eigenstate of the Z stabilizer. The equivalent for the logical qubit $|0_L\rangle$ has to be the $+1$ eigenstate of *all* Z stabilizers, for example, when all data qubits are in the state $|0\rangle$. Logical operations, therefore, must change the eigenstate accordingly, giving rise to the logical operators X_L and Z_L. In the surface code, there exists some freedom in the choice of logical operators due to the combination of possible stabilizers. The operators canonically act on the lattice's *midline* as depicted in figure Figure 8.1a, but freedom of choice exists. Both form a chain of X and Z operators, which can chain the logical qubit state by applying the respective operations to the data qubits. Such operations, which can implemented by applying the equivalent physical operations, are called *transversal*

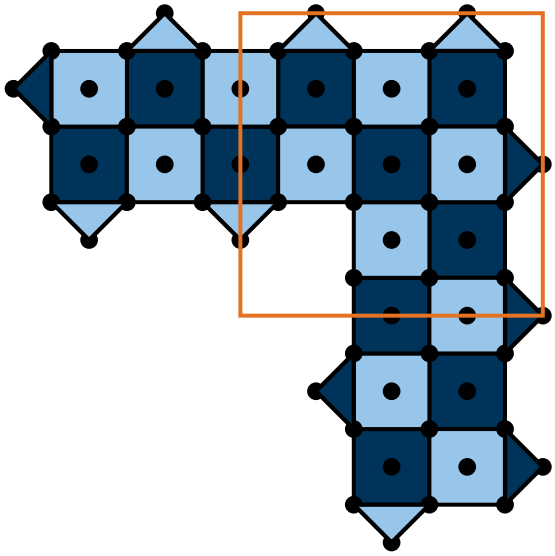
operations. Unfortunately, Knill [151] shows that every error correction code has at least one gate it cannot implement in such a fashion. For the surface code on a 2d lattice, among these gates are the T and CX gates, which are necessary for universal quantum computation. Realizing logical operations corresponding to these gates is still possible but using potentially non-deterministic procedures. The CX gate can be implemented by *lattice surgery* [152]; the T gate can be implemented by *magic state injection* [16].

Lattice surgery is a technique to implement a CX gate between two logical qubits encoded in the surface code. Hardware controlling qubits in 3d space (e.g., neutral atoms) could implement the CX gate by moving the logical qubits close and applying the gate transversally. If this is impossible, two surface code patches, representing the qubits, can be merged and split in a specific way to implement the gate instead. Implementing a structure, following the flow rules (8.3) of the CX gate, yields the correct procedure. For this procedure, tracking multiple logical operators and their byproducts is necessary since errors can still occur during the operation. One can infer the current state from the stabilizer measurements and update the state accordingly. Horsman et al. [152] provide a complete derivation. From a software perspective, the operation represents an L-shaped template (Figure 8.3a), which temporarily consists of three logical qubits. The according surface is sketched in Figure 8.3b, where the Z axis represents time. Each cube represents a distance k surface code repeated for k rounds. The rectangular structure is a visual aid representing the merging of logical qubits; the start and termination of blocks correspond to the initialization and final measurement of the logical qubits.

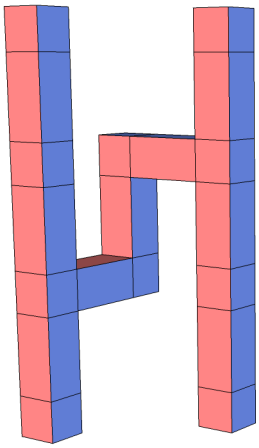
Magic state injection [16] is a way to extend clifford circuits to universal quantum computation. Coincidentally, the stabilizer circuits of the surface code are Clifford circuits. The process uses a form of gate teleportation (see Figure 7.2) to apply a gate from a specifically prepared state into the computation. Preparing such a magic state becomes the new challenge, typically using *state distillation*. The process consists of distilling multiple low-fidelity copies of the state into a single high-fidelity copy. The original proposal is based on a 15-to-1 distillation process, built on an error correction code with transversal T gates. Distillation is inherently probabilistic, as the distillation process can fail. Failure is costly regarding time and resources, so the distillation process can quickly become a computation bottleneck. While the magic states are prepared, the related computation has to be error-free until the states are ready. Reducing the overhead of the distillation process is an active field of research; Gidney et al. [153] propose a novel method based on the color code. The process consists of three stages: firstly, injecting a T state in a low distance surface code, then gradually increasing the distance of the *fault*, and finally expanding into a high distance code. Next to the challenge of implementing these procedures correctly, mixing codes leads to deformities in templates as depicted in Figure 8.4, which require careful handling.

8.1.2 Decoding

Quantum error correction will rely on HPC for the short term. This tie results from the decoding process; measurements must be evaluated, and the decoder must infer the most likely error. The higher the error rate, the more capable the decoder has to be. A



(a) 2d template reqired for lattice surgery.



(b) 3d Modle representing the CX computation.

Figure 8.3: Sketch of the lattice surgery operation template. The initialization and measurement of the corner piece highlighted in orange is important to the operation. Additionally, the change of the logical operators has to be tracked to ensure the correct operation. To fully realize the computation a temporal component has to be added, which is not shown here.

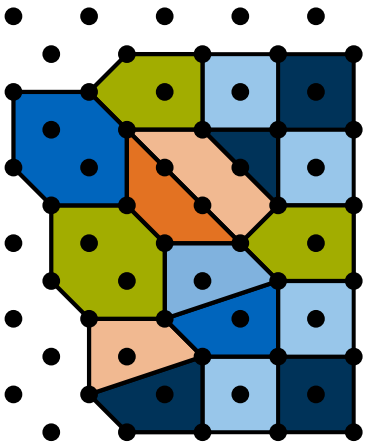


Figure 8.4: Error correction template for the surface code deformed by injecting a state distilled in the color code. Figure adapted from [153].

recurring problem is the speed of the decoder, which typically needs to catch up with the clock rate of the hardware. As a result, a backlog of errors can accumulate, leading to a computation breakdown. Achieving real-time decoding plus correction is, therefore, not realistic yet.

Code construction influences the decoding process. Measuring the stabilizers results in a syndrome, indicating the change of a sign or parity. Combining multiple consecutive syndromes, for which you can determine the expected parity in the absence of errors, is called a *detector*. For example, measuring $|0\rangle$ with a Z stabilizer should result in a $+1$. Tracking all measurements over time yields a (hyper) graph, where detectors form edges between the syndromes. A decoder then operates on the graph, together with information about the expected errors, to infer the most likely error. Decoders must trade off accuracy for speed. For example, the minimum weight perfect matching decoder [154] is fast but less accurate. When designing a quantum code, an off-the-shelf decoder must be adapted to the specific code. Identifying which detectors exist and which form the take is essential for customizing and performance tuning the decoder. Finding decoders is, unfortunately, non-trivial and not immediately apparent from the code construction. The costly way of doing this is through straightforward simulation. Detectors are always related to well-known states, which are either deterministic initializations or, after measurement, collapse in the quantum domain. Starting from these operations, one can track all possible paths an error could take and find its termination criterion. This method becomes computationally expensive for complex codes, so intelligent automation is necessary.

```
block_graph = BlockGraph.from_dae_file("logical_cnot.dae")
observables, _ = block_graph.get_abstract_observables()
compiled_computation = compile_block_graph(block_graph,
    observables=[observables[1]])
circuit = compiled_computation.generate_stim_circuit(
    k=2, # The distance=2k-1  of the code can be changed.
    noise_model=NoiseModel.uniform_depolarizing(0.001),
)
circuit_with_detectors = annotate_detectors_automatically(circuit)
```

Listing 8.1: Basic usages of the `tqec` tool.

8.2 The tqec tool

Through the previous sections, I have outlined the complexity of the surface code, which is deemed one of the simplest but still promising error correction codes. Implementing an error-corrected computation includes optimizing the code, physically realizing the operations, scaling them up, and decoding the result. It is a complex and error-prone process that requires a systematic approach. In this process, tools can automate multiple tasks; implementing them once on a small and still verifiable scale can be reused in more

extensive experiments. The `tqec` project [2] aims to provide the tooling to automate the error correction process. It significantly speeds up the discovery process of error correction experiments. The current features only cover the rotated surface code, but the community continuously adds new features. An example of the primary usage of the tool is shown in Listing 8.1. Also, the target simulator for this is `stim` [155], a high-performance stabilizer simulator.

Building a Computation

A computation can either be built as a *block graph* or from a ZX diagram. A block graph (see Figure 8.3b for an example) represents a computation as a 3d structure, where one axis represents time, canonically, the Z-axis. The choice of the time axis is, in theory, arbitrary, as long as the in and output are correctly identified, as any rotation of the structure does not change the computation. Each block (corresponding to ZX nodes) corresponds to a specific template, such as Figure 8.1e, the default surface code logical qubit. The numbers in the template correspond to plaquettes, representing circuits that the user can assign. Templates can then be scaled up to an arbitrary distance automatically. In and outputs are any open surface representing initializations and measurements of the physical qubits. Pipes connect multiple blocks to realize operations, which are replacement rules for the boundaries of the templates. For example, the lattice surgery operation connects three templates (blocks) joined by two small code patches (pipes).

Identify Observables

Observables are the values of interest in the computation. The observables can be inferred from the block structure according to the propagation rules. Each combination of output and stabilizer defines one observable. `tqec` identifies observables by tracking a correlation surface of the according operator through the computation. A correlation surface can be considered a plane *always* spanning the inside of the block between two sides of the same colors. More accurately, the correlation surface is a set of all measurement parities you must consider when connecting an input to an output of a computation. Therefore, the computation in Figure 8.3b has four observables. `tqec` detects observables using a ZX diagram of the computation; at the moment, `tqec` supports only acyclic diagrams.

Circuit Compilation

Once the user specifies which observables are interesting, they can compile the computation to a circuit. Compilation includes the generation of the stabilizer circuits with the specified distance. A template is instantiated with the given distance, tiled with plaquettes, and repeated over the time axis according to the distance. The instantiation assumes an underlying 2d lattice of physical qubits with nearest-neighbor interactions

[2] `https://github.com/tqec/tqec`

for this. The blocks are positioned over the lattice, and the pipes replace the boundary plaquettes circuits. The noise model can be specified and applied to the physical qubits for simulation purposes.

Detector Annotation

The final step is the annotation of detectors. Detectors are necessary for `stim` to automatically track and simulate decoder performance. Only after the entire circuit is known can the detectors be inferred; Currently, only simple cases are supported, but the goal is to fully automate this process. After the annotation, `stim` can simulate the circuit with different noise models, and users evaluate the performance of the error correction experiment.

8.3 Impact on the Software Stack

Error correction capabilities are not fundamentally changing the software stack. Still, the introduced complexity will require changes in almost all components. All the more important is the loose coupling of the components, allowing room for the integration of error correction. As a first observation, transitioning to the fault-tolerant regime will not make the current software stack obsolete. The components are still necessary but will need extended capabilities based on existing functionality. For example, "error-corrected" does not mean free of any noise. Error mitigation techniques will still be necessary as the computation remains non-deterministic and can be affected by noise. Treating decoding errors as a noise source is a possible approach where novel mitigation techniques require further investigation. I will analyze the hypothetical *error corrected unified toolchain*, focusing on the three components: simulation, compilation, and scheduling. Naturally, I follow the assumption of multiple backends of heterogeneous hardware, potentially admitting different error correction codes.

The separation into *online* and *offline* parts of the software stack is still valid. The *offline* part of the software stack will be affected mainly by the introduced variety of error correction techniques. Finding truly optimal implementations will become nearly impossible but less necessary if the results are sufficient. One question arises: How can the utility of the error correction code be measured, and who is responsible for the decision? In this case, the utility is a tradeoff between the error correction overhead in terms of qubits, decoding, and operations against the error rate and, therefore, the quality of the results. In most cases, the choice should be transparent to the user unless specifically requested. Only expert users can handle the complexity of error correction codes, similar to how only a minority of HPC users can write assembly code. Users must be able to define conditional logic to run in real-time to enable fine-grained control. The desired algorithm may indicate an optimal error correction code, but in most cases, users will be OK with any code that works. The *meta-optimizer* can hide the remainder of the complexity. Optimizing the compilation process and fine-tuning hardware calibrations is a perpetual optimization process. Reinforcement learning agents are a likely solution

for taming the compilation process, as they can train based on the results of prior experiments.

Although optimization targets might differ, the *online* components are affected similarly. Instead of targeting hardware native operations, the optimization process will target the operations arising from the error correction code. This change is insignificant, as compilers can treat the newly available operations as an altered basis set. But compared to NISQ devices, tracking where information is encoded is essential. Keeping track of the information is akin to classical problems where allocating and releasing resources at the correct time is necessary. The components affected the most by introducing error correction are *circuit generation* and result tracking. So far, the `tqec` tool functions as a standalone, ab initio tool, generating static circuit Given a problem and the target accuracy, a similar tool could select an appropriate code with a sufficiently high distance. Then, it automatically generates the circuits and translates the operations to match the error correction code. *Result tracking* will be mainly replaced by decoding and tracking byproducts and signs of operations. This information is required to correct the computation whenever real-time correction is impossible. The interfaces between hardware and software must also keep up with the new requirements. Real-time operations require the lowest latency possible, which might demand redesigning the hardware interface. As long as generic error correction codes are competitive, interfaces can stay minimal but might require new exchange formats. For example, offloading data encoded in a state might need exact timing information when interrupting memory operations.

8.3.1 Simulation

Due to the absence of errors, simulation is unaffected by error correction. However, the use of simulation might change in the context. As seen in the example of `tqec`, the validation aspect will likely be more prominent. Simulators can validate error correction codes before they run on the hardware. Especially stabilizer circuits have a high potential for simulation. Without complex operations (such as T-gates), the simulation can run efficiently on classical hardware. For example, `stim` [155] is a highly optimized simulator for stabilizer circuits, which can simulate up to 10^4 qubits. In `tqec`, we leverage this fact to plot the logical error rate against the code distance to check the performance of the code.

A second use of HPC simulations is to manage the hardware load. Problems that can be simulated efficiently should be if doing so saves resources or achieves higher utilization. Two facets of simulating systems are relevant for this, especially in the context of hybrid simulations[156]. Knowing precisely how good an approximation is, users can tailor the simulation of a system to a fixed error budget. Critical parts can, therefore, be simulated (even exactly) on classical hardware to boost the overall algorithm. Usually, these parts are precisely the ones that cannot be efficiently simulated. However, an inverted scheme is possible, where the critical part is simulated with a high error resilience due to a larger code distance, enabled by dispatching trivial parts of the algorithm to classical hardware. Realizing this in real-time requires high bandwidth and low latency communication between classical and quantum hardware.

Unfortunately, a likely bottleneck is the decoding process, which will supersede simulation in terms of classical computing requirements. Accurate real-time decoding is computationally intensive but necessary. On-device decoding is possible but limited by controller resource restrictions. Hence, decoding will likely be a hybrid process, where most decoding resides on classical hardware. For example, a two-stage approach is possible. An ensemble of lightweight decoders runs on the controller to correct errors where confidence is high. The controller then transmits the proposed corrections and the raw syndrome data to the classical hardware for more accurate decoding. Likely, an accurate tensor network decoder[157] is running on the connected HPC system. The decoder or *meta optimizer* can draw multiple conclusions from the resulting data. First, they could extract insights into the low-level detectors using them to fine-tune parameters during the runtime. Second, if it is too late to correct the computation, this can be addressed by informing the algorithm or re-running the computation. In the worst case scenario, a complete failure, the information can be raised to the user, alerting them of the issue. The *Unified Toolchain* uses this information to optimize further experiments by identifying causes of failure or discouraging setups that are prone to failure.

8.3.2 Compilation

The compilation process needs to undergo significant changes to accommodate error correction. Mainly due to the change of optimization target, for NISQ applications, it is necessary to optimize toward hardware capabilities. For fault-tolerant quantum computing, such optimizations could be detrimental; instead, the focus must lay on efficient computation. The changes need to be reflected by the toolchain; an updated optimization flow is depicted in Figure 8.5. An additional component, ZX-calculus[158], is likely necessary to minimize the cost of computations. The optimization targets change between offline and online stages.

The offline stage optimizes the algorithm and its implementation as a *logical* circuit. This optimization might include selecting the error correction code, code distance, and choice of a decoder, although only the selection is upfront. For example, knowing the noise model, it is possible to tailor the error correction code to hardware[159] and, consequently, the circuit. After providing an optimized logical circuit, the first conversion occurs, reformulating the circuit into an alternative representation. In the case of surface codes, the ZX-calculus is a suitable representation, as it can express the stabilizer formulation. The ZX diagram will undergo a series of optimizations, which are necessary to minimize the cost of the computation. Compared to circuits, optimizations in the ZX-calculus follow a strict set of rules[158]; overall, the concepts stay similar to *JIT* compilation. The computation is then converted to a *physical* circuit, and the error correction code is applied. Conversion involves transforming the logical qubits into physical qubits and adding the necessary detector measurements at the given code distance. Prior, *mapping* and *routing* would happen at this stage, but both optimizations take on a different form. Mapping comes down to choosing between data and measurement qubits, which is less of an optimization and instead bound to hardware capabilities. Routing becomes more complex as it affects all physical qubits encoding the logical

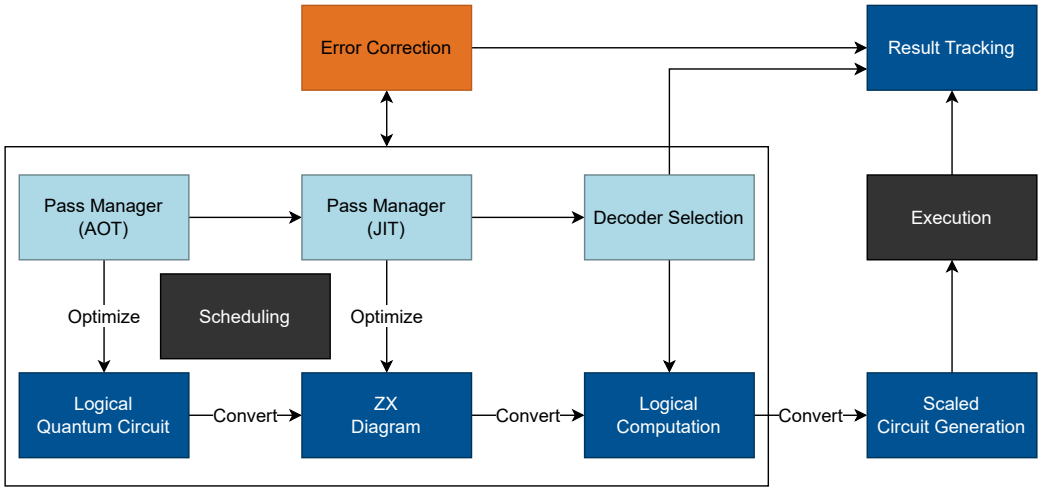


Figure 8.5: An Updated compilation flow (c.f. Figure 6.1) for error corrected quantum computing. The necessary optimizations change between stages and multiple transformations are necessary. ZX-calculus is an important tool to minimize the cost of computations.

qubit. Tracking where information is encoded and moving it to the correct location will be crucial. A rising paradigm is the introduction of specific zones[160] inspired by the capabilities of trapped ions and neutral atoms. For example, two logical qubits can be moved to a compute zone, where they interact, and then moved to a memory zone for error correction. Controllers could execute non-transversal operations in specific gate zones, where state distillation processes prepare necessary resources.

A point arising from this is the choice of gate set to optimize for. The cost of non-transversal operations is typically significantly higher than regular or virtual operations. Hence, optimization should reduce the number of non-transversal operations. For surface codes, T-gates are not transversal; this aligns with the goal of T-gate optimization, where the number of T-gates is minimized. On the one hand, this is beneficial as it aligns with current research [161]. An alternative approach is to select an error correction code that transversally realizes gates that are classically hard to execute. This choice would then synergize with the simulation capabilities described earlier. The type of gate required for an algorithm might still be a deciding factor. For example, Shor's algorithm requires many T-gates, which regular optimization cannot reduce. During compilation, the combination of algorithm, error correction code, hardware capabilities, and optimization target delivers multiple venues for optimization. Such a complex optimization requires the versatility of tightly interacting tools. The *Unified Toolchain* is the foundation for this, but additional research is necessary to unite the different components.

8.3.3 Scheduling

Scheduling, if affected by error correction, can be done in two ways. First, the scheduling problem gets more complex due to the implications of the error correction code. The schedule must consider the resource overhead of each code, its implications for the algorithm, and compatibility with hardware. For example, I will discuss scheduling Schor's on two hardware, one superconducting with a square lattice connectivity and an all-to-all connected trapped ion system. Let us assume that the superconducting system can only realize a surface code with 10 logical qubits. In contrast, the trapped ion system realizes 5 logical qubits with a Triorthogonal code [162], which realizes a transversal T-gate[163]. The scheduler has to consider the following aspects: It should maximize resource utilization; resource overhead for the surface code is significantly higher than for the Triorthogonal code. An initial assumption might lead to scheduling the ion trap system, as another user might want to utilize the superconducting system. However, Shor's algorithm depends on the number of qubits, and the accuracy of running on the ion trap system might be insufficient. Additionally, the transversal T-gate might be significantly faster than the surface code realization. Error correction can introduce multiple layers of conflicting information, which need to be resolved by the scheduler. Additionally, the considerations present in the current scheduler form are still valid. A balanced approach is necessary for all these aspects, and the scheduler must be aware of the hardware capabilities. Available combinations of error correction codes and hardware and their implication on target metrics must also be included. The scheduler can then make an informed decision, which will likely be a tradeoff between resource utilization and computational efficiency.

The second aspect is the hardware capabilities that are associated with error correction. A scheduler must manage many additional quantum resources enabled by longer coherence times. Due to these resources, quantum systems resemble classical HPC systems, where schedulers manage memory and compute resources. Novel resources include magic states generated by distillation, entanglement for communication in the form of Bell pairs, and quantum *memory*. The scheduler needs to track which resources the computation requires at each time point and how to prepare them accordingly. Tracking needs to happen in close cooperation with the *program orchestration* component, which should keep track of the state and flow of the computation. A likely approach is introducing a *hierarchical scheduler* [164], which assigns priorities to different resources. The scheduler I introduced in Chapter 7 could be responsible for machine assignment, while additional components handle the resource allocation. Advanced techniques, already established in HPC, can be introduced to quantum resources from here. For example, an analog to *data offloading*, state preparation could be responsible for encoding classical into quantum data at the correct time. Triggering such procedures should fall to the orchestrator, who the scheduler can then inform. Similarly, the system can handle different zones in quantum hardware. In contrast to classical HPC, such operations translate to physical realizations, such as actively moving atoms inside a quantum chip. A new management level becomes necessary from such capabilities, which must be integrated into the scheduler.

Unifying their management is another requirement resulting from quantum resources becoming more similar to classical resources. A completely hybrid scheduler, consisting of classical and quantum components, is likely necessary. Only tightly integrated components can enable efficient hybrid computations. The natural avenue for this is to extend existing software to include quantum resources. While not part of this work, integration into `SLURM` is a long-term goal for my runtime components. Some preparation has been made for this, but as error correction is not yet a reality for large-scale computation, this is in a prototypical state.

9 Conclusion and Outlook

Bridging the gap between quantum computing and HPC is the formulated goal of this work. Novel approaches are necessary, adapting established techniques from HPC to the unique challenges of quantum computing to link quantum computing and HPC With the presented work, I formulate a sound foundation from a software engineering perspective. I focus on the development of runtime components for near-term hybrid quantum-classical systems. Specifically, I target systems with multiple hardware architectures combined to enable efficient quantum-accelerated computations. Such systems are characterized by heterogenous backends, which require regular interaction between classical and quantum components. To achieve this goal, I develop components for hybrid runtime environments capable of taking advantage of quantum computers and classical simulators. I propose and adhere to three guiding principles for the development of these components, which extend beyond my efforts to the future development of similar systems. These principles are: *tight integration* but simultaneously *loose coupling* of quantum and classical components, *abstraction* of the underlying hardware, and high *utilization* of the system through HPC principles.

Hybrid simulations are promising candidates for providing near-term quantum utility. Such applications simulate quantum systems on a classical computer while offloading certain parts of the computation to a quantum computer; close interaction is necessary. Targeting such applications[1] provides ground to evaluate the developed runtime components. Analyzing the current software landscape leads to two main observations. Firstly, the software tools are fragmented, tied to specific hardware vendors, with different grades of maturity and completeness, and usually not designed to work together. Secondly, the software tools are designed for the cloud, again tied to a single hardware. Users submit jobs in total-job mode, specifying the execution environment in advance and no longer controlling it during runtime. In contrast, users can configure innumerable parameters on HPC systems to achieve the highest performance possible.

The first effort of this work is to identify the gaps in HPCQC integration. Most existing software tools are inadequate for this task purely due to their choice of programming language. Few tools exist that specifically target HPC environments, such as `XACC` [46] or `cuda-q` [45]. However, these tools lack prominent features compared to cloud-based quantum software libraries and must be more far-reaching to cover the demands of HPC environments. Unification processes are necessary when recognizing the scattered landscape of existing tools and techniques. The QIR is a promising candidate to work as an intermediate target language. Considering this, my first contribution is the proposal of the *Unified Toolchain*. The *Unified Toolchain* is a software stack that aims to provide

[1] I am currently contributing to an early prototype of such an application, see [165].

a standard definition of components and their interfaces and interactions. Parts of this have been realized in the `MQSS` [56], allowing seamless integration of new techniques and tools. Additionally, I propose a data-driven way to handle the complexity of optimizing performance for hybrid quantum-classical systems. I will focus on the produced metadata and outline ways to utilize it. The remainder of my work focuses on implementing prototypes for the *Unified Toolchain* and evaluating them toward the guiding principles. A significant hurdle to successful integration is targeting multiple backends, which cloud-based approaches currently do not offer. Hence, my work specifically targets components capable of interacting with multiple *heterogenous* backends. Concretely, I focus on compiling single circuits for distribution over multiple backends and scheduling to automatically distribute the resulting workload. For validation, I use a high-performance simulator that could also be used as a target to offload parts of the computation.

The first component I cover in this work is a high-performance simulator based on tensor network techniques. Tensor networks are well-suited for HPCQC; they offer rich theory from quantum physics and ample opportunities for parallelization and optimization. From the perspective of the *Unified Toolchain*, the simulator has two main functions. Firstly, it serves as a reference when exploring interfaces for quantum backends. Other components can be validated using the simulator. A key learning from this effort is the necessity to abstract the quantum hardware with one standard interface. The resulting standard must entail the specific properties of the hardware, such as error budgets or, in the case of simulators, advantageous entanglement patterns. Different simulators can provide advantages depending on the formulated problem. I show a specific use case for Tree Tensor Networks, which are well-suited for quantum circuits with particular properties. A MPS-based simulator might be more advantageous when approximating a computation, potentially due to a higher error budget. Implementing my approach for using HPC systems also opens the possibility of hybrid simulations. In a hybrid paradigm, the most expensive parts of the computation are offloaded to a quantum computer, while the remainder can still be handled classically. Such systems provide immediate utility while similar to the future scenarios described above.

Compilation is a heavily researched topic in quantum computing, trying to overcome the limitations of NISQ hardware. The existing tools focus on single backends, which do not change during execution. In contrast, the *Unified Toolchain* must support multiple backends and recognize a two-stage compilation approach. I highlight the impact on the runtime of the inefficient second stage and the necessity of transferring the computational load to the first stage. With `Flow Synth`, I provide a prototype implementation leveraging state-of-the-art machine learning techniques to speed up the compilation process. I implement multiple variations specifically targeting HPCQC use cases, for example, the ability to compile circuits for multiple backends simultaneously. The tool follows a generic pipeline, which also extends to other techniques. Generic approaches such as QIR, an emerging standard, allow reusing compilation passes. In combination with other quantum-specific approaches, compilation performance improves significantly. I show how to implement them and their interaction with other tools in the *Unified Toolchain*. With that, I contribute a necessary component to realizing hybrid quantum-classical systems.

Finally, I provide `MILQ`, a prototypical scheduler for heterogenous quantum backends, which is also suitable for HPC environments. It can deal with the unique challenges of quantum computing. To summarize, current quantum hardware operates on premises different from classical hardware. Preempting a computation is not feasible, and context switching is expensive due to high set-up costs. Set-up costs include platform-specific features such as trap layout and compilation times. For the NISQ era, hardware noise is an additional side-effect that schedulers must consider during the scheduling process. Handling noise requires the optimization of multiple objectives during scheduling. I formalize the scheduling problem and provide an exact solution using a linear programming approach and two heuristic solutions based on scatter search and reinforcement learning. Using a technique called *circuit cutting*, I propose a multithreading approach to quantum computing. The scheduler can automatically cut and distribute circuits over multiple backends for multiple users. It uses familiar techniques from HPC scheduling, such as backfilling, to maximize the utilization of the system. I evaluate the prototype in different settings, identifying necessary interactions with backends and other components. The need for noise and resource estimation is especially apparent in the scheduling process.

Beyond discussing the implementation of the *Unified Toolchain*, I also provide an outlook toward future developments. Error correction is an inevitable step in the development of quantum computers. Therefore, I analyze the toolchain to determine the requirements for error correction. For this, I first investigate one of the most promising error correction codes, the surface code. Supporting software is highly underdeveloped for automating and scaling codes to larger systems. To accelerate the analysis and development of error correction codes, I contribute to the development of automatization tooling, namely `tqec`. The tool allows it to quickly build error correction circuits for computations on the surface code. Running the resulting experiments, the gaps in the toolchain become apparent. I highlight the most prominent obstacle: the increased choices and complexity. For the toolchain, this translates to vertical changes to accommodate the new requirements. My proposed scheme of loosely coupled components supports the necessary changes. The interplay of error correction codes, decoders, and hardware capabilities must be evaluated for user-specific applications. I outline the necessary steps to introduce the changes into the component prototypes I developed. But it is also clear that a significant development effort is necessary to fully support error correction codes in the *Unified Toolchain*.

In summary, the *Unified Toolchain* is a promising approach for strengthening the integration between quantum computing and HPC. My work provides the foundation for a tightly integrated but loosely coupled system. It forms the basis for developing future hybrid runtime environments, especially targeting heterogeneous quantum backends. For large-scale hybrid simulations, simulators can serve as specialized backends; utilization of simulators is still largely unexplored. My work is still in the pioneering phase and only exploratory. However, my findings can be used as a guideline for future development. With the `MQSS`, production-ready components are already in development, realizing concepts of the *Unified Toolchain*. The interfaces I propose are theoretical but enable a suitable exchange between components. My scheduler `MILQ`, for example, already uses

device-independent interfaces and figures of merit and constraints. The essential next step is a maturing phase, in which the components are rolled out and tested under real-world conditions. For this, novel algorithms utilizing the unique properties of emerging quantum hardware are necessary.

Bibliography

[1] R. P. Feynman. Simulating physics with computers. *International Journal of Theoretical Physics*, 21(6):467–488, June 1982. URL: https://doi.org/10.1007/BF02650179, doi:10.1007/BF02650179.

[2] Y. Manin. Computable and uncomputable. *Sovetskoye Radio, Moscow*, 128:28, 1980.

[3] C. Monroe, D. M. Meekhof, B. E. King, W. M. Itano, and D. J. Wineland. Demonstration of a fundamental quantum logic gate. *Phys. Rev. Lett.*, 75:4714–4717, Dec 1995. URL: https://link.aps.org/doi/10.1103/PhysRevLett.75.4714, doi:10.1103/PhysRevLett.75.4714.

[4] D. P. DiVincenzo. The Physical Implementation of Quantum Computation. *Fortschritte der Physik*, 48(9-11):771–783, 2000. _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/1521-3978%28200009%2948%3A9/11%3C771%3A%3AAID-PROP771%3E3.0.CO%3B2-E. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/1521-3978%28200009%2948%3A9/11%3C771%3A%3AAID-PROP771%3E3.0.CO%3B2-E, doi:10.1002/1521-3978(200009)48:9/11<771::AID-PROP771>3.0.CO;2-E.

[5] P. W. Shor. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM Journal on Computing*, 26(5):1484–1509, 1997. URL: https://doi.org/10.1137/S0097539795293172, arXiv:https://doi.org/10.1137/S0097539795293172, doi:10.1137/S0097539795293172.

[6] J. Preskill. Quantum Computing in the NISQ era and beyond. *Quantum*, 2:79, Aug 2018. URL: https://doi.org/10.22331/q-2018-08-06-79, doi:10.22331/q-2018-08-06-79.

[7] Y. Kim, A. Eddins, S. Anand, K. X. Wei, E. van den Berg, S. Rosenblatt, H. Nayfeh, Y. Wu, M. Zaletel, K. Temme, and A. Kandala. Evidence for the utility of quantum computing before fault tolerance. *Nature*, 618(7965):500–505, Jun 2023. URL: https://doi.org/10.1038/s41586-023-06096-3, doi:10.1038/s41586-023-06096-3.

[8] M. A. Nielsen and I. L. Chuang. *Quantum Computation and Quantum Information: 10th Anniversary Edition*. Cambridge University Press, Cambridge, 1 edition, 2012. doi:10.1017/CBO9780511976667.

[9] M. A. Nielsen. A simple formula for the average gate fidelity of a quantum dynamical operation. *Physics Letters A*, 303(4):249–252, 2002.

[10] K. Temme, S. Bravyi, and J. M. Gambetta. Error mitigation for short-depth quantum circuits. *Phys. Rev. Lett.*, 119:180509, Nov 2017. URL: `https://link.aps.org/doi/10.1103/PhysRevLett.119.180509`, `doi:10.1103/PhysRevLett.119.180509`.

[11] J. Liu, A. Gonzales, and Z. H. Saleem. Classical simulators as quantum error mitigators via circuit cutting, 2022. unpublished. URL: `https://arxiv.org/abs/2212.07335`, `arXiv:2212.07335`, `doi:10.48550/arXiv.2212.07335`.

[12] K. N. Smith, M. A. Perlin, P. Gokhale, P. Frederick, D. Owusu-Antwi, R. Rines, V. Omole, and F. Chong. Clifford-based circuit cutting for quantum simulation. In *Proceedings of the 50th Annual International Symposium on Computer Architecture*, ISCA '23, New York, NY, USA, 2023. Association for Computing Machinery. URL: `https://doi.org/10.1145/3579371.3589352`, `doi:10.1145/3579371.3589352`.

[13] J. J. Wallman and J. Emerson. Noise tailoring for scalable quantum computation via randomized compiling. *Phys. Rev. A*, 94:052325, Nov 2016. URL: `https://link.aps.org/doi/10.1103/PhysRevA.94.052325`, `doi:10.1103/PhysRevA.94.052325`.

[14] B. Pokharel, N. Anand, B. Fortman, and D. A. Lidar. Demonstration of fidelity improvement using dynamical decoupling with superconducting qubits. *Phys. Rev. Lett.*, 121:220502, Nov 2018. URL: `https://link.aps.org/doi/10.1103/PhysRevLett.121.220502`, `doi:10.1103/PhysRevLett.121.220502`.

[15] E. Knill, R. Laflamme, and W. H. Zurek. Resilient quantum computation. *Science*, 279(5349):342–345, 1998. URL: `https://www.science.org/doi/abs/10.1126/science.279.5349.342`, `arXiv:https://www.science.org/doi/pdf/10.1126/science.279.5349.342`, `doi:10.1126/science.279.5349.342`.

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

[17] F. Arute, K. Arya, R. Babbush, D. Bacon, J. C. Bardin, R. Barends, R. Biswas, S. Boixo, F. G. S. L. Brandao, D. A. Buell, B. Burkett, Y. Chen, Z. Chen, B. Chiaro, R. Collins, W. Courtney, A. Dunsworth, E. Farhi, B. Foxen, A. Fowler, C. Gidney, M. Giustina, R. Graff, K. Guerin, S. Habegger, M. P. Harrigan, M. J. Hartmann, A. Ho, M. Hoffmann, T. Huang, T. S. Humble, S. V. Isakov, E. Jeffrey, Z. Jiang, D. Kafri, K. Kechedzhi, J. Kelly, P. V. Klimov, S. Knysh, A. Korotkov, F. Kostritsa, D. Landhuis, M. Lindmark, E. Lucero, D. Lyakh, S. Mandrà, J. R. McClean, M. McEwen, A. Megrant, X. Mi, K. Michielsen,

M. Mohseni, J. Mutus, O. Naaman, M. Neeley, C. Neill, M. Y. Niu, E. Ostby, A. Petukhov, J. C. Platt, C. Quintana, E. G. Rieffel, P. Roushan, N. C. Rubin, D. Sank, K. J. Satzinger, V. Smelyanskiy, K. J. Sung, M. D. Trevithick, A. Vainsencher, B. Villalonga, T. White, Z. J. Yao, P. Yeh, A. Zalcman, H. Neven, and J. M. Martinis. Quantum supremacy using a programmable superconducting processor. *Nature*, 574(7779):505–510, Oct 2019. URL: https://doi.org/10.1038/s41586-019-1666-5, doi:10.1038/s41586-019-1666-5.

[18] C. Huang, F. Zhang, M. Newman, J. Cai, X. Gao, Z. Tian, J. Wu, H. Xu, H. Yu, B. Yuan, et al. Classical simulation of quantum supremacy circuits. *arXiv preprint arXiv:2005.06787*, 2020.

[19] F. Pan and P. Zhang. Simulation of quantum circuits using the big-batch tensor network method. *Phys. Rev. Lett.*, 128:030501, Jan 2022. URL: https://link.aps.org/doi/10.1103/PhysRevLett.128.030501, doi:10.1103/PhysRevLett.128.030501.

[20] J. M. Martyn, Z. M. Rossi, A. K. Tan, and I. L. Chuang. Grand unification of quantum algorithms. *PRX quantum*, 2(4):040203, 2021.

[21] 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*, 986:1–128, 2022.

[22] A. Anand, P. Schleich, S. Alperin-Lea, P. W. Jensen, S. Sim, M. Díaz-Tinoco, J. S. Kottmann, M. Degroote, A. F. Izmaylov, and A. Aspuru-Guzik. A quantum computing view on unitary coupled cluster theory. *Chemical Society Reviews*, 51(5):1659–1684, 2022.

[23] M. Kjaergaard, M. E. Schwartz, J. Braumüller, P. Krantz, J. I.-J. Wang, S. Gustavsson, and W. D. Oliver. Superconducting qubits: Current state of play. *Annual Review of Condensed Matter Physics*, 11(1):369–395, 2020. URL: https://doi.org/10.1146/annurev-conmatphys-031119-050605, arXiv:https://doi.org/10.1146/annurev-conmatphys-031119-050605, doi:10.1146/annurev-conmatphys-031119-050605.

[24] C. D. Bruzewicz, J. Chiaverini, R. McConnell, and J. M. Sage. Trapped-ion quantum computing: Progress and challenges. *Applied Physics Reviews*, 6(2):021314, 05 2019. URL: https://doi.org/10.1063/1.5088164, arXiv:https://pubs.aip.org/aip/apr/article-pdf/doi/10.1063/1.5088164/14577412/021314_1_online.pdf, doi:10.1063/1.5088164.

[25] X. Wu, X. Liang, Y. Tian, F. Yang, C. Chen, Y.-C. Liu, M. K. Tey, and L. You. A concise review of rydberg atom based quantum computation and quantum simulation*. *Chinese Physics B*, 30(2):020305, feb 2021. URL: https://dx.doi.org/10.1088/1674-1056/abd76f, doi:10.1088/1674-1056/abd76f.

[26] S. Slussarenko and G. J. Pryde. Photonic quantum information processing: A concise review. *Applied Physics Reviews*, 6(4), 2019.

[27] F. Marxer, A. Vepsäläinen, S. W. Jolin, J. Tuorila, A. Landra, C. Ockeloen-Korppi, W. Liu, O. Ahonen, A. Auer, L. Belzane, V. Bergholm, C. F. Chan, K. W. Chan, T. Hiltunen, J. Hotari, E. Hyyppä, J. Ikonen, D. Janzso, M. Koistinen, J. Kotilahti, T. Li, J. Luus, M. Papic, M. Partanen, J. Räbinä, J. Rosti, M. Savytskyi, M. Seppälä, V. Sevriuk, E. Takala, B. Tarasinski, M. J. Thapa, F. Tosto, N. Vorobeva, L. Yu, K. Y. Tan, J. Hassel, M. Möttönen, and J. Heinsoo. Long-distance transmon coupler with cz-gate fidelity above 99.8%. *PRX Quantum*, 4:010314, Feb 2023. URL: `https://link.aps.org/doi/10.1103/PRXQuantum.4.010314`, `doi:10.1103/PRXQuantum.4.010314`.

[28] V. Kaushal, B. Lekitsch, A. Stahl, J. Hilder, D. Pijn, C. Schmiegelow, A. Bermudez, M. Müller, F. Schmidt-Kaler, and U. Poschinger. Shuttling-based trapped-ion quantum information processing. *AVS Quantum Science*, 2(1):014101, 03 2020. URL: `https://doi.org/10.1116/1.5126186`, `arXiv:https://pubs.aip.org/avs/aqs/article-pdf/doi/10.1116/1.5126186/19738817/014101\_1\_online.pdf`, `doi:10.1116/1.5126186`.

[29] A. Sen and K. Rezai. Comparing qubit platforms in the race to feasible quantum computing. *Journal of Student Research*, 10(4), 2021.

[30] M. Suchara, A. Faruque, C.-Y. Lai, G. Paz, F. T. Chong, and J. Kubiatowicz. Comparing the overhead of topological and concatenated quantum error correction. *arXiv preprint arXiv:1312.2316*, 2013.

[31] F. Voichick, L. Lampropoulos, and R. Rand. Cognac: Circuit optimization via gradients and noise-aware compilation, 2024. `arXiv:2311.02769`.

[32] T. S. Humble, A. McCaskey, D. I. Lyakh, M. Gowrishankar, A. Frisch, and T. Monz. Quantum computers for high-performance computing. *IEEE Micro*, 41(5):15–23, sep 2021. URL: `https://doi.org/10.1109/MM.2021.3099140`, `doi:10.1109/MM.2021.3099140`.

[33] M. Ruefenacht, B. G. Taketani, P. Lähteenmäki, V. Bergholm, D. Kranzlmüller, L. Schulz, and M. Schulz. Bringing quantum acceleration to supercomputers, 2022. URL: `https://www.quantum.lrz.de/fileadmin/QIC/Downloads/IQM_HPC-QC-Integration-Whitepaper.pdf`.

[34] T. Hoefler, T. Häner, and M. Troyer. Disentangling hype from practicality: On realistically achieving quantum advantage. *Commun. ACM*, 66(5):82–87, apr 2023. URL: `https://doi.org/10.1145/3571725`, `doi:10.1145/3571725`.

[35] E. Knill. Conventions for quantum pseudocode. Technical Report LA-UR-96-2724, Los Alamos National Lab., 6 1996. URL: `https://www.osti.gov/biblio/366453`, `doi:10.2172/366453`.

[36] T. M. Mintz, A. J. Mccaskey, E. F. Dumitrescu, S. V. Moore, S. Powers, and P. Lougovski. Qcor: A language extension specification for the heterogeneous quantum-classical model of computation. *ACM Journal on Emerging Technologies in Computing Systems (JETC)*, 16(2):1–17, 2020. doi:10.1145/3380964.

[37] X. Fu, J. Yu, X. Su, H. Jiang, H. Wu, F. Cheng, X. Deng, J. Zhang, L. Jin, Y. Yang, L. Xu, C. Hu, A. Huang, G. Huang, X. Qiang, M. Deng, P. Xu, W. Xu, W. Liu, Y. Zhang, Y. Deng, J. Wu, and Y. Feng. Quingo: A programming framework for heterogeneous quantum-classical computing with nisq features. *ACM Transactions on Quantum Computing*, 2(4), dec 2021. URL: https://doi.org/10.1145/3483528, doi:10.1145/3483528.

[38] Microsoft. *Q# Language Specification*, 2020. URL: https://github.com/microsoft/qsharp-language/tree/main/Specifications/Language#q-language.

[39] J. Paykin, R. Rand, and S. Zdancewic. QWIRE: a core language for quantum circuits. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 846–858, Paris France, January 2017. ACM. URL: https://dl.acm.org/doi/10.1145/3009837.3009894, doi:10.1145/3009837.3009894.

[40] A. Javadi-Abhari, M. Treinish, K. Krsulich, C. J. Wood, J. Lishman, J. Gacon, S. Martiel, P. D. Nation, L. S. Bishop, A. W. Cross, B. R. Johnson, and J. M. Gambetta. Quantum computing with Qiskit, 2024. unpublished. arXiv:2405.08810, doi:10.48550/arXiv.2405.08810.

[41] 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. *ACM Transactions on Quantum Computing*, June 2025. Just Accepted. URL: https://doi.org/10.1145/3743149, doi:10.1145/3743149.

[42] E. Younis and C. Iancu. Quantum circuit optimization and transpilation via parameterized circuit instantiation. In *2022 IEEE International Conference on Quantum Computing and Engineering (QCE)*, pages 465–475. IEEE, 2022.

[43] C. Campbell, F. T. Chong, D. Dahl, P. Frederick, P. Goiporia, P. Gokhale, B. Hall, S. Issa, E. Jones, S. Lee, A. Litteken, V. Omole, D. Owusu-Antwi, M. A. Perlin, R. Rines, K. N. Smith, N. Goss, A. Hashim, R. Naik, E. Younis, D. Lobser, C. G. Yale, B. Huang, and J. Liu. Superstaq: Deep optimization of quantum programs. In *2023 IEEE International Conference on Quantum Computing and Engineering (QCE)*, volume 01, pages 1020–1032, 2023. doi:10.1109/QCE57702.2023.00116.

[44] C. Lattner and V. Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code*

Generation and Optimization: Feedback-Directed and Runtime Optimization, CGO '04, page 75, USA, 2004. IEEE Computer Society.

[45] T. C. Q. development team. Cuda quantum, July 2023. URL: https://doi.org/10.5281/zenodo.8110867, doi:10.5281/zenodo.8110867.

[46] A. J. McCaskey, D. I. Lyakh, E. F. Dumitrescu, S. S. Powers, and T. S. Humble. Xacc: a system-level software infrastructure for heterogeneous quantum–classical computing. *Quantum Science and Technology*, 5(2):024002, 2020.

[47] J. Izaac. Pennylane blog - introducing catalyst: Quantum just-in-time compilation, March 2023. URL: https://pennylane.ai/blog/2023/03/introducing-catalyst-quantum-just-in-time-compilation/.

[48] P. Khalate, X.-C. Wu, S. Premaratne, J. Hogaboam, A. Holmes, A. Schmitz, G. G. Guerreschi, X. Zou, and A. Y. Matsuura. An llvm-based c++ compiler toolchain for variational hybrid quantum-classical algorithms and quantum accelerators. *arXiv preprint arXiv:2202.11142*, 2022.

[49] QIR Alliance. *QIR Specification*, 2021. Also see https://qir-alliance.org. URL: https://github.com/qir-alliance/qir-spec.

[50] C. Lattner, M. Amini, U. Bondhugula, A. Cohen, A. Davis, J. Pienaar, R. Riddle, T. Shpeisman, N. Vasilache, and O. Zinenko. MLIR: Scaling Compiler Infrastructure for Domain Specific Computation. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 2–14, Seoul, Korea (South), February 2021. IEEE. URL: https://ieeexplore.ieee.org/document/9370308/, doi:10.1109/CGO51591.2021.9370308.

[51] A. Cross, A. Javadi-Abhari, T. Alexander, N. De Beaudrap, L. S. Bishop, S. Heidel, C. A. Ryan, P. Sivarajah, J. Smolin, J. M. Gambetta, and B. R. Johnson. Openqasm 3: A broader and deeper quantum assembly language. *ACM Transactions on Quantum Computing*, 3(3):1–50, September 2022. URL: http://dx.doi.org/10.1145/3505636, doi:10.1145/3505636.

[52] S. Sivarajah, S. Dilkes, A. Cowtan, W. Simmons, A. Edgington, and R. Duncan. TKET: A Retargetable Compiler for NISQ devices. *Quantum Science and Technology*, 6, November 2020. doi:10.1088/2058-9565/ab8e92.

[53] A. Cross, A. Javadi-Abhari, T. Alexander, N. De Beaudrap, L. S. Bishop, S. Heidel, C. A. Ryan, P. Sivarajah, J. Smolin, J. M. Gambetta, and B. R. Johnson. OpenQASM 3: A Broader and Deeper Quantum Assembly Language. *ACM Transactions on Quantum Computing*, 3(3):1–50, September 2022. URL: https://dl.acm.org/doi/10.1145/3505636, doi:10.1145/3505636.

[54] X. Fei, L. T. Brady, J. Larson, S. Leyffer, and S. Shen. Binary Control Pulse Optimization for Quantum Systems. *Quantum*, 7:892, January 2023. URL: https://doi.org/10.22331/q-2023-01-04-892, doi:10.22331/q-2023-01-04-892.

[55] P. Seitz, A. Elsharkawy, X.-T. M. To, and M. Schulz. Toward a unified hybrid hpcqc toolchain. In *2023 IEEE International Conference on Quantum Computing and Engineering (QCE)*, volume 02, pages 96–102, 2023. doi:10.1109/QCE57702.2023.10191.

[56] 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)*, volume 02, pages 399–400, 2023. doi:10.1109/QCE57702.2023.10301.

[57] A. S. Green, P. L. Lumsdaine, N. J. Ross, P. Selinger, and B. Valiron. Quipper: a scalable quantum programming language. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, pages 333–342, New York, NY, USA, June 2013. Association for Computing Machinery. URL: https://doi.org/10.1145/2491956.2462177, doi:10.1145/2491956.2462177.

[58] Y. L. Lim, A. Beige, and L. C. Kwek. Repeat-until-success linear optics distributed quantum computing. *Physical review letters*, 95(3):030505, 2005.

[59] T. Grurl, J. Fuß, and R. Wille. Noise-aware quantum circuit simulation with decision diagrams. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 42(3):860–873, 2023. doi:10.1109/TCAD.2022.3182628.

[60] S. Li, Y. Kimura, H. Sato, and M. Fujita. Parallelizing quantum simulation with decision diagrams. *IEEE Transactions on Quantum Engineering*, 5:1–12, 2024. doi:10.1109/TQE.2024.3364546.

[61] P. Seitz, I. Medina, E. Cruz, Q. Huang, and C. B. Mendl. Simulating quantum circuits using tree tensor networks. *Quantum*, 7:964, March 2023. URL: https://doi.org/10.22331/q-2023-03-30-964, doi:10.22331/q-2023-03-30-964.

[62] R. Penrose. Applications of negative dimensional tensors. *Combinatorial mathematics and its applications*, 1:221–244, 1971.

[63] J. Schuhmacher, M. Ballarin, A. Baiardi, G. Magnifico, F. Tacchino, S. Montangero, and I. Tavernelli. Hybrid tree tensor networks for quantum simulation, 2024. URL: https://arxiv.org/abs/2404.05784, arXiv:2404.05784.

[64] L. Chi-Chung, P. Sadayappan, and R. Wenger. On optimizing a class of multi-dimensional loops with reduction for parallel execution. *Parallel Processing Letters*, 7(02):157–168, 1997.

[65] J. Biamonte and V. Bergholm. Tensor networks in a nutshell, 2017. unpublished. URL: https://arxiv.org/abs/1708.00006, arXiv:1708.00006, doi:10.48550/arXiv.1708.00006.

[66] J. C. Bridgeman and C. T. Chubb. Hand-waving and interpretive dance: an introductory course on tensor networks. *Journal of physics A: Mathematical and theoretical*, 50(22):223001, 2017.

[67] J. Eisert, M. Cramer, and M. B. Plenio. Area laws for the entanglement entropy - a review. *Rev. Mod. Phys.*, 82:277–306, 2010. `arXiv:0808.3773`, `doi:10.1103/RevModPhys.82.277`.

[68] E. M. Stoudenmire and S. R. White. Minimally entangled typical thermal state algorithms. *New Journal of Physics*, 12(5):055026, may 2010. URL: `https://dx.doi.org/10.1088/1367-2630/12/5/055026`, `doi:10.1088/1367-2630/12/5/055026`.

[69] R. Orús and G. Vidal. Simulation of two-dimensional quantum systems on an infinite lattice revisited: Corner transfer matrix for tensor contraction. *Physical Review B—Condensed Matter and Materials Physics*, 80(9):094403, 2009.

[70] Y. Zhou, E. M. Stoudenmire, and X. Waintal. What limits the simulation of quantum computers? *Phys. Rev. X*, 10:041038, Nov 2020. URL: `https://link.aps.org/doi/10.1103/PhysRevX.10.041038`, `doi:10.1103/PhysRevX.10.041038`.

[71] G. Vidal. Efficient classical simulation of slightly entangled quantum computations. *Phys. Rev. Lett.*, 91:147902, 2003. `doi:10.1103/PhysRevLett.91.147902`.

[72] J. Unfried, J. Hauschild, and F. Pollmann. Fast time evolution of matrix product states using the qr decomposition. *Phys. Rev. B*, 107:155133, Apr 2023. URL: `https://link.aps.org/doi/10.1103/PhysRevB.107.155133`, `doi:10.1103/PhysRevB.107.155133`.

[73] Y.-Y. Shi, L.-M. Duan, and G. Vidal. Classical simulation of quantum many-body systems with a tree tensor network. *Phys. Rev. A*, 74:022320, 2006. `doi:10.1103/PhysRevA.74.022320`.

[74] L. Tagliacozzo, G. Evenbly, and G. Vidal. Simulation of two-dimensional quantum systems using a tree tensor network that exploits the entropic area law. *Phys. Rev. B*, 80:235127, 2009. `doi:10.1103/PhysRevB.80.235127`.

[75] J. Gray and S. Kourtis. Hyper-optimized tensor network contraction. *Quantum*, 5:410, 2021. `doi:10.22331/q-2021-03-15-410`.

[76] S. Szalay, M. Pfeffer, V. Murg, G. Barcza, F. Verstraete, R. Schneider, and O. Legeza. Tensor product methods and entanglement optimization for ab initio quantum chemistry. *Int. J. Quantum Chem.*, 115:1342–1391, 2015. `doi:10.1002/qua.24898`.

[77] V. Murg, F. Verstraete, Ö. Legeza, and R. M. Noack. Simulating strongly correlated quantum systems with tree tensor networks. *Physical Review B*, 82(20), 11 2010. URL: `http://dx.doi.org/10.1103/PhysRevB.82.205105`, `doi:10.1103/physrevb.82.205105`.

[78] G. Ferrari, G. Magnifico, and S. Montangero. Adaptive-weighted tree tensor networks for disordered quantum many-body systems. *Phys. Rev. B*, 105:214201, 2022. doi:10.1103/PhysRevB.105.214201.

[79] A. S. Jermyn. Efficient tree decomposition of high-rank tensors. *Journal of Computational Physics*, 377:142–154, 2019. URL: https://www.sciencedirect.com/science/article/pii/S0021999118306934, doi:https://doi.org/10.1016/j.jcp.2018.10.026.

[80] Y. Pang, T. Hao, A. Dugad, Y. Zhou, and E. Solomonik. Efficient 2d tensor network simulation of quantum systems. In *SC20: International conference for high performance computing, networking, storage and analysis*, pages 1–14. IEEE, 2020.

[81] H. F. Hofmann. How to simulate a universal quantum computer using negative probabilities. *J. Phys. A: Math. Theor.*, 42:275304, 2009. URL: https://dx.doi.org/10.1088/1751-811/42/27/275304, doi:10.1088/1751-8113/42/27/275304.

[82] A. Elsharkawy, X. Guo, and M. Schulz. Integration of quantum accelerators into hpc: Toward a unified quantum platform. In *2024 IEEE International Conference on Quantum Computing and Engineering (QCE)*, volume 01, pages 774–783, 2024. doi:10.1109/QCE60285.2024.00097.

[83] E. Nielsen, J. K. Gamble, K. Rudinger, T. Scholten, K. Young, and R. Blume-Kohout. Gate set tomography. *Quantum*, 5:557, 2021.

[84] Y. Stade, L. Schmid, L. Burgholzer, and R. Wille. An abstract model and efficient routing for logical entangling gates on zoned neutral atom architectures. In *2024 IEEE International Conference on Quantum Computing and Engineering (QCE)*, volume 01, pages 784–795, 2024. doi:10.1109/QCE60285.2024.00098.

[85] E. Deist, Y.-H. Lu, J. Ho, M. K. Pasha, J. Zeiher, Z. Yan, and D. M. Stamper-Kurn. Mid-circuit cavity measurement in a neutral atom array. *Phys. Rev. Lett.*, 129:203602, Nov 2022. URL: https://link.aps.org/doi/10.1103/PhysRevLett.129.203602, doi:10.1103/PhysRevLett.129.203602.

[86] M. DeCross, E. Chertkov, M. Kohagen, and M. Foss-Feig. Qubit-reuse compilation with mid-circuit measurement and reset. *Phys. Rev. X*, 13:041057, Dec 2023. URL: https://link.aps.org/doi/10.1103/PhysRevX.13.041057, doi:10.1103/PhysRevX.13.041057.

[87] T. Peng, A. W. Harrow, M. Ozols, and X. Wu. Simulating large quantum circuits on a small quantum computer. *Phys. Rev. Lett.*, 125:150504, 2020. URL: https://link.aps.org/doi/10.1103/PhysRevLett.125.150504, doi:10.1103/PhysRevLett.125.150504.

[88] A. Lowe, M. Medvidović, A. Hayes, L. J. O'Riordan, T. R. Bromley, J. M. Arrazola, and N. Killoran. Fast quantum circuit cutting with randomized measurements. *Quantum*, 7:934, 2023. URL: https://quantum-journal.org/papers/q-2023-03-02-934/, doi:10.22331/q-2023-03-02-934.

[89] G. Uchehara, T. M. Aamodt, and O. D. Matteo. Rotation-inspired circuit cut optimization. In *2022 IEEE/ACM Third International Workshop on Quantum Computing Software (QCS)*, page 50, Los Alamitos, CA, USA, 2022. IEEE Computer Society. URL: https://doi.ieeecomputersociety.org/10.1109/QCS56647.2022.00011, doi:10.1109/QCS56647.2022.00011.

[90] P. Pednault. An alternative approach to optimal wire cutting without ancilla qubits, 2023. unpublished. URL: https://doi.org/10.48550/arXiv.2303.08287, doi:10.48550/arXiv.2303.08287.

[91] H. Harada, K. Wada, and N. Yamamoto. Optimal parallel wire cutting without ancilla qubits, 2023. unpublished. URL: https://doi.org/10.48550/arXiv.2303.07340, doi:10.48550/arXiv.2303.07340.

[92] L. Brenner, C. Piveteau, and D. Sutter. Optimal wire cutting with classical communication, 2023. unpublished. URL: https://doi.org/10.48550/arXiv.2302.03366, doi:10.48550/arXiv.2302.03366.

[93] K. Mitarai and K. Fujii. Constructing a virtual two-qubit gate by sampling single-qubit operations. *New J. Phys.*, 23:023021, 2021. URL: https://doi.org/10.1088/1367-2630/abd7bc, doi:10.1088/1367-2630/abd7bc.

[94] K. Mitarai and K. Fujii. Overhead for simulating a non-local channel with local channels by quasiprobability sampling. *Quantum*, 5:388, 2021. URL: https://quantum-journal.org/papers/q-2021-01-28-388/, doi:https://doi.org/10.22331/q-2021-01-28-388.

[95] C. Piveteau and D. Sutter. Circuit knitting with classical communication. *IEEE Trans. Inf. Theory*, page 1, 2023. URL: https://doi.org/10.1109/TIT.2023.3310797, doi:10.1109/TIT.2023.3310797.

[96] C. Ufrecht, M. Periyasamy, S. Rietsch, D. D. Scherer, A. Plinge, and C. Mutschler. Cutting multi-control quantum gates with ZX calculus. *Quantum*, 7:1147, 2023. URL: zsp, doi:10.22331/q-2023-10-23-1147.

[97] C. Ufrecht, L. S. Herzog, D. D. Scherer, M. Periyasamy, S. Rietsch, A. Plinge, and C. Mutschler. Optimal joint cutting of two-qubit rotation gates. *Phys. Rev. A*, 109:052440, May 2024. URL: https://link.aps.org/doi/10.1103/PhysRevA.109.052440, doi:10.1103/PhysRevA.109.052440.

[98] L. Schmitt, C. Piveteau, and D. Sutter. Cutting circuits with multiple two-qubit unitaries, 2024. unpublished. URL: https://doi.org/10.48550/arXiv.2312.11638, doi:10.48550/arXiv.2312.11638.

[99] A. W. Harrow and A. Lowe. Optimal quantum circuit cuts with application to clustered hamiltonian simulation, 2024. unpublished. URL: https://doi.org/10.48550/arXiv.2403.01018, doi:10.48550/arXiv.2403.01018.

[100] H. Pashayan, J. J. Wallman, and S. D. Bartlett. Estimating outcome probabilities of quantum circuits using quasiprobabilities. *Phys. Rev. Lett.*, 115:070501, 2015. URL: https://link.aps.org/doi/10.1103/PhysRevLett.115.070501, doi:https://doi.org/10.1103/PhysRevLett.115.070501.

[101] M. Bandic, L. Prielinger, J. Nüßlein, A. Ovide, S. Rodrigo, S. Abadal, H. van Someren, G. Vardoyan, E. Alarcon, C. G. Almudever, and S. Feld. Mapping quantum circuits to modular architectures with qubo. In *2023 IEEE International Conference on Quantum Computing and Engineering (QCE)*, volume 01, pages 790–801, 2023. URL: https://doi.org/10.48550/arXiv.2305.06687, doi:10.1109/QCE57702.2023.00094.

[102] F. Fürrutter, G. Muñoz-Gil, and H. J. Briegel. Quantum circuit synthesis with diffusion models. *Nature Machine Intelligence*, 6(5):515–524, May 2024. URL: https://doi.org/10.1038/s42256-024-00831-9, doi:10.1038/s42256-024-00831-9.

[103] A. Oussidi and A. Elhassouny. Deep generative models: Survey. In *2018 International Conference on Intelligent Systems and Computer Vision (ISCV)*, pages 1–8, 2018. doi:10.1109/ISACV.2018.8354080.

[104] B. Coecke and R. Duncan. Interacting quantum observables: categorical algebra and diagrammatics. *New Journal of Physics*, 13(4):043016, apr 2011. URL: https://dx.doi.org/10.1088/1367-2630/13/4/043016, doi:10.1088/1367-2630/13/4/043016.

[105] R. Rombach, A. Blattmann, D. Lorenz, P. Esser, and B. Ommer. High-resolution image synthesis with latent diffusion models. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 10684–10695, June 2022.

[106] X. Liu, L. Wu, M. Ye, and Q. Liu. Let us build bridges: Understanding and extending diffusion generative models, 2022. unpublished. URL: https://arxiv.org/abs/2208.14699, arXiv:2208.14699, doi:10.48550/arXiv.2208.14699.

[107] X. Liu, C. Gong, and Q. Liu. Flow straight and fast: Learning to generate and transfer data with rectified flow, 2022. unpublished. URL: https://arxiv.org/abs/2209.03003, arXiv:2209.03003, doi:10.48550/arXiv.2209.03003.

[108] D. P. Kingma and M. Welling. Auto-encoding variational bayes. In Y. Bengio and Y. LeCun, editors, *2nd International Conference on Learning Representations, ICLR 2014, Banff, AB, Canada, April 14-16, 2014, Conference Track Proceedings*, 2014. URL: http://arxiv.org/abs/1312.6114.

[109] Q. Dao, H. Phung, B. Nguyen, and A. Tran. Flow matching in latent space. *arXiv preprint arXiv:2307.08698*, 2023.

[110] N. Quetschlich, L. Burgholzer, and R. Wille. MQT Bench: Benchmarking Software and Design Automation Tools for Quantum Computing. *Quantum*, 7:1062, July 2023. URL: https://doi.org/10.22331/q-2023-07-20-1062, doi:10.22331/q-2023-07-20-1062.

[111] S. Tan, C. Lang, L. Xiang, S. Wang, X. Jia, Z. Tan, T. Li, J. Yin, Y. Shang, A. Python, L. Lu, and J. Yin. Quct: A framework for analyzing quantum circuit by extracting contextual and topological features. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '23, pages 494–508, New York, NY, USA, 2023. Association for Computing Machinery. URL: https://doi.org/10.1145/3613424.3614274, doi:10.1145/3613424.3614274.

[112] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. u. Kaiser, and I. Polosukhin. Attention is all you need. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc., 2017. URL: https://proceedings.neurips.cc/paper_files/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf.

[113] G. Ilharco, M. Wortsman, R. Wightman, C. Gordon, N. Carlini, R. Taori, A. Dave, V. Shankar, H. Namkoong, J. Miller, H. Hajishirzi, A. Farhadi, and L. Schmidt. Openclip, July 2021. If you use this software, please cite it as below. URL: https://doi.org/10.5281/zenodo.5143773, doi:10.5281/zenodo.5143773.

[114] R. Wightman. Pytorch image models. https://github.com/rwightman/pytorch-image-models, 2019. doi:10.5281/zenodo.4414861.

[115] M. Heusel, H. Ramsauer, T. Unterthiner, B. Nessler, and S. Hochreiter. Gans trained by a two time-scale update rule converge to a local nash equilibrium. *Advances in neural information processing systems*, 30, 2017.

[116] P. Seitz, M. Geiger, and C. B. Mendl. Multithreaded parallelism for heterogeneous clusters of qpus. In *ISC High Performance 2024 Research Paper Proceedings (39th International Conference)*, pages 1–8. Prometeus GmbH, 2024.

[117] P. Seitz, M. Geiger, C. Ufrecht, A. Plinge, C. Mutschler, D. D. Scherer, and C. B. Mendl. Scim milq: An hpc quantum scheduler. In *2024 IEEE International Conference on Quantum Computing and Engineering (QCE)*, volume 02, pages 292–298, 2024. doi:10.1109/QCE60285.2024.10294.

[118] W. van Dam, M. Mykhailova, and M. Soeken. Using azure quantum resource estimator for assessing performance of fault tolerant quantum computation. In *Proceedings of the SC '23 Workshops of The International Conference on High*

Performance Computing, Network, Storage, and Analysis, SC-W '23, pages 1414–1419, New York, NY, USA, 2023. Association for Computing Machinery. URL: https://doi.org/10.1145/3624062.3624211, doi:10.1145/3624062.3624211.

[119] G. S. Ravi, K. N. Smith, P. Murali, and F. T. Chong. Adaptive job and resource management for the growing quantum cloud. In *2021 IEEE International Conference on Quantum Computing and Engineering (QCE)*, pages 301–312, 2021. doi:10.1109/QCE52317.2021.00047.

[120] B. Harper, B. Tonekaboni, B. Goldozian, M. Sevior, and M. Usman. Crosstalk attacks and defence in a shared quantum computing environment. *arXiv preprint arXiv:2402.02753*, 2024.

[121] A. A. Saki, M. Alam, K. Phalak, A. Suresh, R. O. Topaloglu, and S. Ghosh. A survey and tutorial on security and resilience of quantum computing. In *2021 IEEE European Test Symposium (ETS)*, pages 1–10, 2021. doi:10.1109/ETS50041.2021.9465397.

[122] D. Barredo, V. Lienhard, S. de Léséleuc, T. Lahaye, and A. Browaeys. Synthetic three-dimensional atomic structures assembled atom by atom. *Nature*, 561(7721):79–82, Sep 2018. URL: https://doi.org/10.1038/s41586-018-0450-2, doi:10.1038/s41586-018-0450-2.

[123] A. B. Yoo, M. A. Jette, and M. Grondona. Slurm: Simple linux utility for resource management. In D. Feitelson, L. Rudolph, and U. Schwiegelshohn, editors, *Job Scheduling Strategies for Parallel Processing*, pages 44–60, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg. doi:10.1007/10968987_3.

[124] J. K. Ousterhout et al. Scheduling techniques for concurrent systems. In *ICDCS*, volume 82, pages 22–30, 1982.

[125] B. Nitzberg, J. M. Schopf, and J. P. Jones. Pbs pro: Grid computing and scheduling attributes. In *Grid resource management: state of the art and future trends*, pages 183–190. Springer, 2004. URL: https://doi.org/10.1007/978-1-4615-0509-9_13, doi:10.1007/978-1-4615-0509-9_13.

[126] Y. Fan. Job scheduling in high performance computing, 2021. URL: https://arxiv.org/abs/2109.09269, arXiv:2109.09269.

[127] D. G. Feitelson and L. Rudolph. Gang scheduling performance benefits for fine-grain synchronization. *Journal of Parallel and Distributed Computing*, 16(4):306–318, 1992. URL: https://www.sciencedirect.com/science/article/pii/074373159290014E, doi:https://doi.org/10.1016/0743-7315(92)90014-E.

[128] C. H. Bennett, G. Brassard, C. Crépeau, R. Jozsa, A. Peres, and W. K. Wootters. Teleporting an unknown quantum state via dual classical and einstein-podolsky-rosen channels. *Phys. Rev. Lett.*, 70:1895–1899, Mar 1993. URL: https://link.

aps.org/doi/10.1103/PhysRevLett.70.1895, doi:10.1103/PhysRevLett.70.1895.

[129] D. Gottesman and I. L. Chuang. Quantum teleportation is a universal computational primitive. *arXiv preprint quant-ph/9908010*, 1999.

[130] B. S. Baker. A new proof for the first-fit decreasing bin-packing algorithm. *Journal of Algorithms*, 6(1):49–70, 1985. URL: https://www.sciencedirect.com/science/article/pii/0196677485900185, doi:https://doi.org/10.1016/0196-6774(85)90018-5.

[131] L. Fanjul-Peyro and R. Ruiz. Size-reduction heuristics for the unrelated parallel machines scheduling problem. *Comput. Oper. Res.*, 38(1):301–309, jan 2011. URL: https://doi.org/10.1016/j.cor.2010.05.005, doi:10.1016/j.cor.2010.05.005.

[132] F. J. Rodriguez, M. Lozano, C. Blum, and C. García-Martínez. An iterated greedy algorithm for the large-scale unrelated parallel machines scheduling problem. *Comput. Oper. Res.*, 40(7):1829–1841, jul 2013. URL: https://doi.org/10.1016/j.cor.2013.01.018, doi:10.1016/j.cor.2013.01.018.

[133] I. M. Al-harkan and A. A. Qamhan. Optimize unrelated parallel machines scheduling problems with multiple limited additional resources, sequence-dependent setup times and release date constraints. *IEEE Access*, 7:171533–171547, 2019. URL: https://doi.org/10.1109/ACCESS.2019.2955975, doi:10.1109/ACCESS.2019.2955975.

[134] J. K. Lenstra, D. B. Shmoys, and É. Tardos. Approximation algorithms for scheduling unrelated parallel machines. *Mathematical Programming*, 46(1):259–271, Jan 1990. URL: https://doi.org/10.1007/BF01585745, doi:10.1007/BF01585745.

[135] S. Kan, Z. Du, M. Palma, S. A. Stein, C. Liu, W. Wei, J. Chen, A. Li, and Y. Mao. Scalable circuit cutting and scheduling in a resource-constrained and distributed quantum system. In *2024 IEEE International Conference on Quantum Computing and Engineering (QCE)*, volume 01, pages 1077–1088, 2024. doi:10.1109/QCE60285.2024.00127.

[136] D. Bhoumik, R. Majumdar, and S. Sur-Kolay. Resource-aware scheduling of multiple quantum circuits on a hardware device. 2024. unpublished. URL: https://arxiv.org/abs/2407.08930, doi:10.48550/arXiv.2407.08930.

[137] R. Graham, E. Lawler, J. Lenstra, and A. Kan. Optimization and approximation in deterministic sequencing and scheduling: a survey. In P. Hammer, E. Johnson, and B. Korte, editors, *Discrete Optimization II*, volume 5 of *Annals of Discrete Mathematics*, pages 287–326. Elsevier, 1979. URL: https://www.sciencedirect.com/science/article/pii/S016750600870356X, doi:https://doi.org/10.1016/S0167-5060(08)70356-X.

[138] L. Rosenbauer., A. Stein., H. Stegherr., and J. Hähner. Metaheuristics for the minimum set cover problem: A comparison. In *Proceedings of the 12th International Joint Conference on Computational Intelligence (IJCCI 2020) - ECTA*, pages 123–130. INSTICC, SciTePress, 2020. URL: https://doi.org/10.5220/0010019901230130, doi:10.5220/0010019901230130.

[139] L. Li, P. Anand, K. He, and D. Englund. Dynamic inhomogeneous quantum resource scheduling with reinforcement learning. 2024. unpublished. URL: https://arxiv.org/abs/2405.16380, doi:10.48550/arXiv.2405.16380.

[140] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov. Proximal policy optimization algorithms, 2017. unpublished. URL: https://doi.org/10.48550/arXiv.1707.06347, arXiv:1707.06347, doi:10.48550/arXiv.1707.06347.

[141] P. D. Nation and M. Treinish. Suppressing quantum circuit errors due to system variability. *PRX Quantum*, 4:010327, Mar 2023. URL: https://link.aps.org/doi/10.1103/PRXQuantum.4.010327, doi:10.1103/PRXQuantum.4.010327.

[142] L. Dagum and R. Menon. Openmp: An industry-standard api for shared-memory programming. *IEEE Comput. Sci. Eng.*, 5(1):46–55, jan 1998. URL: https://doi.org/10.1109/99.660313, doi:10.1109/99.660313.

[143] A. Steane. Multiple-particle interference and quantum error correction. *Proceedings of the Royal Society of London. Series A: Mathematical, Physical and Engineering Sciences*, 452(1954):2551–2577, 1996. URL: https://royalsocietypublishing.org/doi/abs/10.1098/rspa.1996.0136, arXiv:https://royalsocietypublishing.org/doi/pdf/10.1098/rspa.1996.0136, doi:10.1098/rspa.1996.0136.

[144] H. Bombin and M. A. Martin-Delgado. Topological quantum distillation. *Phys. Rev. Lett.*, 97:180501, Oct 2006. URL: https://link.aps.org/doi/10.1103/PhysRevLett.97.180501, doi:10.1103/PhysRevLett.97.180501.

[145] A. G. Fowler, M. Mariantoni, J. M. Martinis, and A. N. Cleland. Surface codes: Towards practical large-scale quantum computation. *Physical Review A—Atomic, Molecular, and Optical Physics*, 86(3):032324, 2012.

[146] N. Delfosse and B. W. Reichardt. Short shor-style syndrome sequences, 2020. unpublished. URL: https://arxiv.org/abs/2008.05051, arXiv:2008.05051, doi:10.48550/arXiv.2008.05051.

[147] A. Y. Kitaev. Quantum computations: algorithms and error correction. *Russian Mathematical Surveys*, 52(6):1191, 1997.

[148] A. A. Kovalev and L. P. Pryadko. Improved quantum hypergraph-product ldpc codes. In *2012 IEEE International Symposium on Information Theory Proceedings*, pages 348–352, 2012. doi:10.1109/ISIT.2012.6284206.

[149] H. Bombin and M. A. Martin-Delgado. Optimal resources for topological two-dimensional stabilizer codes: Comparative study. *Phys. Rev. A*, 76:012305, Jul 2007. URL: https://link.aps.org/doi/10.1103/PhysRevA.76.012305, doi:10.1103/PhysRevA.76.012305.

[150] J. P. Bonilla Ataides, D. K. Tuckett, S. D. Bartlett, S. T. Flammia, and B. J. Brown. The xzzx surface code. *Nature Communications*, 12(1):2172, Apr 2021. URL: https://doi.org/10.1038/s41467-021-22274-1, doi:10.1038/s41467-021-22274-1.

[151] E. Knill, R. Laflamme, and L. Viola. Theory of quantum error correction for general noise. *Phys. Rev. Lett.*, 84:2525–2528, Mar 2000. URL: https://link.aps.org/doi/10.1103/PhysRevLett.84.2525, doi:10.1103/PhysRevLett.84.2525.

[152] D. Horsman, A. G. Fowler, S. Devitt, and R. Van Meter. Surface code quantum computing by lattice surgery. *New Journal of Physics*, 14(12):123011, 2012.

[153] C. Gidney, N. Shutty, and C. Jones. Magic state cultivation: growing t states as cheap as cnot gates. *arXiv preprint arXiv:2409.17595*, 2024. unpublished. URL: https://arxiv.org/abs/2409.17595, doi:10.48550/arXiv.2409.17595.

[154] L. Skoric, D. E. Browne, K. M. Barnes, N. I. Gillespie, and E. T. Campbell. Parallel window decoding enables scalable fault tolerant quantum computation. *Nature Communications*, 14(1):7040, Nov 2023. URL: https://doi.org/10.1038/s41467-023-42482-1, doi:10.1038/s41467-023-42482-1.

[155] C. Gidney. Stim: a fast stabilizer circuit simulator. *Quantum*, 5:497, July 2021. URL: https://doi.org/10.22331/q-2021-07-06-497, doi:10.22331/q-2021-07-06-497.

[156] N. Tornow, C. B. Mendl, and P. Bhatotia. Quantum-classical computing via tensor networks, 2024. unpublished. URL: https://arxiv.org/abs/2410.15080, arXiv:2410.15080, doi:10.48550/arXiv.2410.15080.

[157] A. J. Ferris and D. Poulin. Tensor networks and quantum error correction. *Phys. Rev. Lett.*, 113:030501, Jul 2014. URL: https://link.aps.org/doi/10.1103/PhysRevLett.113.030501, doi:10.1103/PhysRevLett.113.030501.

[158] J. van de Wetering. Zx-calculus for the working quantum computer scientist, 2020. unpublished. URL: https://arxiv.org/abs/2012.13966, arXiv:2012.13966, doi:10.48550/arXiv.2012.13966.

[159] A. M. Stephens, W. J. Munro, and K. Nemoto. High-threshold topological quantum error correction against biased noise. *Phys. Rev. A*, 88:060301, Dec 2013. URL: https://link.aps.org/doi/10.1103/PhysRevA.88.060301, doi:10.1103/PhysRevA.88.060301.

[160] D. Bluvstein, S. J. Evered, A. A. Geim, S. H. Li, H. Zhou, T. Manovitz, S. Ebadi, M. Cain, M. Kalinowski, D. Hangleiter, J. P. Bonilla Ataides, N. Maskara, I. Cong, X. Gao, P. Sales Rodriguez, T. Karolyshyn, G. Semeghini, M. J. Gullans, M. Greiner, V. Vuletić, and M. D. Lukin. Logical quantum processor based on reconfigurable atom arrays. *Nature*, 626(7997):58–65, Feb 2024. URL: https://doi.org/10.1038/s41586-023-06927-3, doi:10.1038/s41586-023-06927-3.

[161] M. Amy, D. Maslov, and M. Mosca. Polynomial-time t-depth optimization of clifford+t circuits via matroid partitioning. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 33(10):1476–1489, 2014. doi:10.1109/TCAD.2014.2341953.

[162] S. Bravyi and J. Haah. Magic-state distillation with low overhead. *Phys. Rev. A*, 86:052329, Nov 2012. URL: https://link.aps.org/doi/10.1103/PhysRevA.86.052329, doi:10.1103/PhysRevA.86.052329.

[163] N. Rengaswamy, R. Calderbank, M. Newman, and H. D. Pfister. On optimality of css codes for transversal t. *IEEE Journal on Selected Areas in Information Theory*, 1(2):499–514, 2020. doi:10.1109/JSAIT.2020.3012914.

[164] G. Lipari, P. Gai, M. Trimarchi, G. Guidi, and P. Ancilotti. A hierarchical framework for component-based real-time systems. In I. Crnkovic, J. A. Stafford, H. W. Schmidt, and K. Wallnau, editors, *Component-Based Software Engineering*, pages 209–216, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[165] T. M. Bickley, A. Mingare, T. Weaving, M. W. de la Bastida, S. Wan, M. Nibbi, P. Seitz, A. Ralli, P. J. Love, M. Chung, M. H. Vera, L. Schulz, and P. V. Coveney. Extending quantum computing through subspace, embedding and classical molecular dynamics techniques, 2025. unpublished. URL: https://arxiv.org/abs/2505.16796, arXiv:2505.16796, doi:10.48550/arXiv.2505.16796.

List of Figures

List of Tables

Acronyms

API	application programming interface.
CPU	central processing unit.
FOMAC	figure of merits and contsraints.
GPU	graphics processing unit.
HPC	high performance computing.
HPCQC	high performance computing - quantum computing.
HQCC	heterogeneous quantum-classical computation.
IR	intermediate representation.
ISA	instruction set architecture.
MPI	Message Passing Interface.
MPS	matrix product state.
MQSS	Munich Quantum Software Stack.
NISQ	noisy intermediate-scale quantum.
QC	quantum computing.
QIR	Quantum Intermediate Representation.
QPL	quantum programming language.
QPU	quantum processing unit.
QRAM	quantum random access machine.
SVD	singular value decomposition.
TEBD	time-evolving block decimation.
TPU	tensor processing unit.
TTN	tree tensor network.
VAE	variational autoencoder.
VQE	variational quantum eigensolver.

A Simulation Circuits

The circuits depicted here are taken from my work Seitz et al. [61].

The *lattice* circuit is depth 8 random benchmarking circuit with random one and two-qubit gates. Two qubit gates are always applied to nearest neighbors in a 2d lattice, and drawn from a random set. The exact specifications are given in [17]; The mapping to the tree is depicted in Figure A.1 For scaling this circuit I choose a square layout of $n \times n, n \in \{2, ..., 10\}$. The structure remains roughly the same, partitioning the square in its four corners; edges are always on the lower levels as they have less direct neighbors.

The *tree-like* structure is given in Figure A.2. It consists of a four qubit base pattern e.g 0–3 , where one qubit acts as connection to the final qubit. Scaling starts with five qubits, and always adding a group of four. This pattern can evidently mapped to a tree, where most interactions are local. The ultimate qubit subjects to the $D_{\max}$ restriction, but violates the MPS area law.

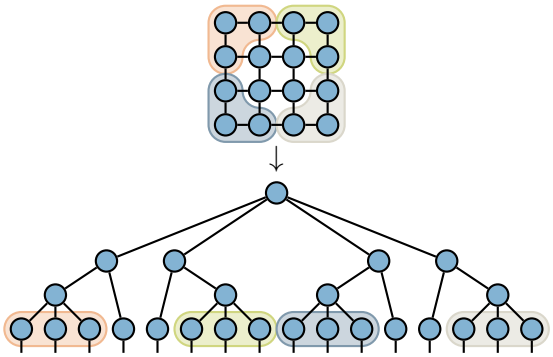


Figure A.1: 4×4-qubit lattice with a nearest neighbor gate pattern, and corresponding tree architecture for representing the output quantum state.

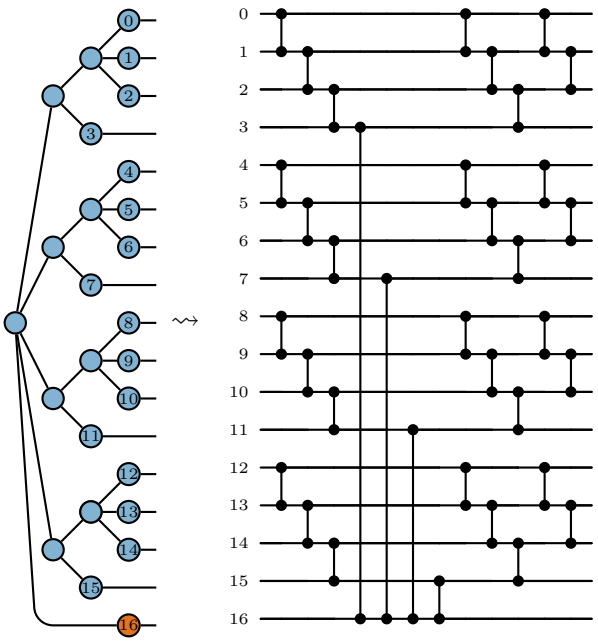


Figure A.2: Well-structured circuit pattern resulting from a clusterable tree. The highlighted subtree contains all connections crossing boundaries.

B Data Sets

Listing B.1 shows the code used to generate the data sets for the random dataset. Given a number of samples, it places a random number of gates in a circuit up to a maximum depth, similarly to the `MQT Bench` [110] functionality. Using the `Qiskit` [40] transpile on optimization level 3, it generates an optimized circuit for a provided gate set. If none is provided, the program randomly chooses from the list of default gate sets. For the resulting circuit I calculate its unitary matrix using the `Qiskit` simulator. In postprocessing all circuits with a depth above the threshold of 256 are discarded. The code generates random circuits for the random synthesis task, circuits from the algorithmic synthesis task, and circuits for the heterogeneous synthesis task. The code generates the circuits using the `MQT Bench` library and the `Qiskit` library [40]. The code generates the circuits using the following parameters: Generating algorithm circuits works the same; additionally the algorithm parameter configures `MQT Bench`. Figure B.1 and Figure B.2 show example circuits with 5 qubits. The hybrid synthesis tasks combines two such circuits by placing random CX gates between them. Figure B.3 shows an example circuit for the heterogeneous synthesis task generated from two random circuits of size 5 joined by two CX gates.

```python
def generate_random_circuit_dataset(
    n_samples: int, # number of samples to generate
    limits: Limits, # A Limits object with the max/mins of size and depth
    gate_sets: list[list[str]] | None = None, # Custom gate set list.
) -> list[CircuitData]:
    if gate_sets is None:
        gate_sets = GATE_SETS # Default gate sets
    samples_per_gate_set = n_samples // len(gate_sets)
    circuits = []
    for basis_gates in gate_sets:
        for _ in range(samples_per_gate_set):
            num_qubits = np.random.randint(limits.min_size, limits.max_size + 1)
            depth = np.random.randint(limits.min_depth, limits.max_depth)
            # MQT Bench wrapper
            circuit = generate_random_circuit(num_qubits, depth, basis_gates)
            # Qiskit unitary simulation
            unitary = unitary_from_circuit(circuit)
            circuits.append(
                CircuitData(circuit, circuit.depth(), unitary, [basis_gates])
            )
    return circuits
```

Listing B.1: Dataset generation code for the random dataset.

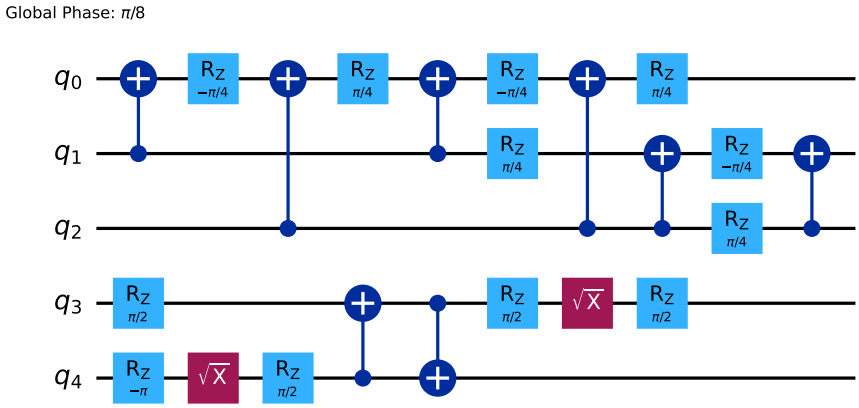


Figure B.1: Example circuit for the random synthesis task.

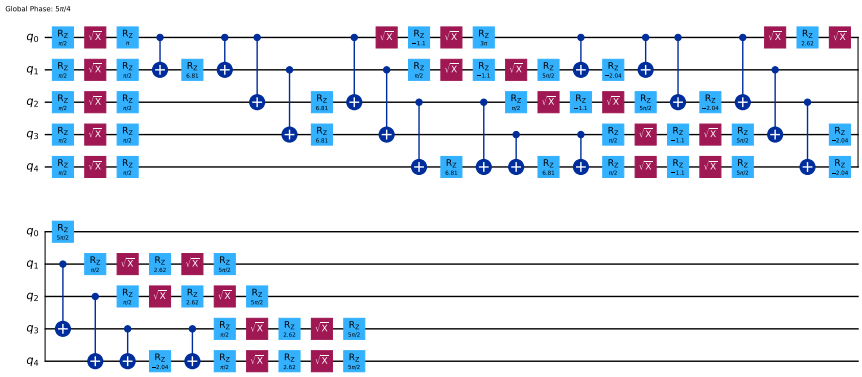


Figure B.2: Example circuit for the algorithm synthesis task.

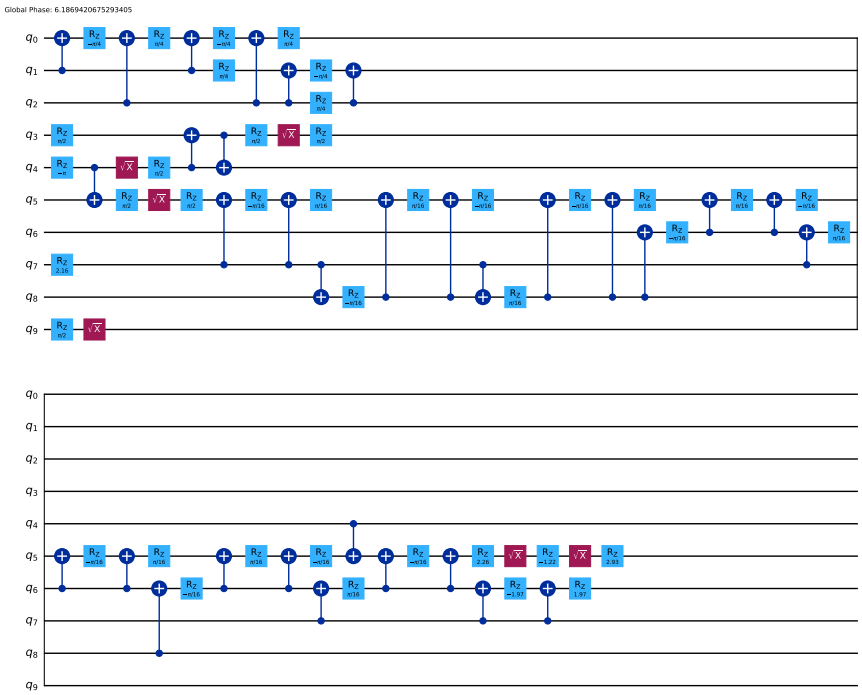


Figure B.3: Example circuit for the heterogeneous synthesis task.