SimuQ: A Framework for Programming Quantum Hamiltonian Simulation with Analog Compilation

YUXIANG PENG, University of Maryland, United States
JACOB YOUNG, University of Maryland, United States
PENGYU LIU*, Carnegie Mellon University, United States
XIAODI WU, University of Maryland, United States

Quantum Hamiltonian simulation, which simulates the evolution of quantum systems and probes quantum phenomena, is one of the most promising applications of quantum computing. Recent experimental results suggest that Hamiltonian-oriented analog quantum simulation would be advantageous over circuit-oriented digital quantum simulation in the Noisy Intermediate-Scale Quantum (NISQ) machine era. However, programming analog quantum simulators is much more challenging due to the lack of a unified interface between hardware and software. In this paper, we design and implement SimuQ, the first framework for quantum Hamiltonian simulation that supports Hamiltonian programming and pulse-level compilation to heterogeneous analog quantum simulators. Specifically, in SimuQ, front-end users specify the target quantum system with Hamiltonian Modeling Language, and the Hamiltonian-level programmability of analog quantum simulators is specified through a new abstraction called the abstract analog instruction set (AAIS) and programmed in AAIS Specification Language by hardware providers. Through a solver-based compilation, SimuQ generates executable pulse schedules for real devices to simulate the evolution of desired quantum systems, which is demonstrated on superconducting (IBM), neutral-atom (QuEra), and trapped-ion (IonQ) quantum devices. Moreover, we demonstrate the advantages of exposing the Hamiltonian-level programmability of devices with native operations or interaction-based gates and establish a small benchmark of quantum simulation to evaluate SimuQ's compiler with the above analog quantum simulators.

CCS Concepts: • **Computer systems organization** → **Quantum computing**; • **Hardware** → *Emerging languages and compilers*; • **Software and its engineering** → Domain specific languages.

Additional Key Words and Phrases: quantum simulation, analog quantum computing, pulse-level programming

ACM Reference Format:
Yuxiang Peng, Jacob Young, Pengyu Liu, and Xiaodi Wu. 2024. SimuQ: A Framework for Programming Quantum Hamiltonian Simulation with Analog Compilation. *Proc. ACM Program. Lang.* 8, POPL, Article 81 (January 2024), 31 pages. https://doi.org/10.1145/3632923

1 INTRODUCTION

1.1 Background and Motivation

Developing appropriate abstraction is a critical step in designing programming languages that help bridge the domain users and the potentially complicated computing devices. Abstraction is

*Pengyu Liu started participating in this work when he was an undergraduate at Tsinghua University, Beijing, China.

Authors' addresses: Yuxiang Peng, University of Maryland, College Park, Maryland, United States, ypeng15@umd.edu; Jacob Young, University of Maryland, College Park, Maryland, United States, jyoung25@umd.edu; Pengyu Liu, Carnegie Mellon University, Pittsburgh, Pennsylvania, United States, pengyul@andrew.cmu.edu; Xiaodi Wu, University of Maryland, College Park, Maryland, United States, xwu@cs.umd.edu.

ACM 2475-1421/2024/1-ART81
https://doi.org/10.1145/3632923

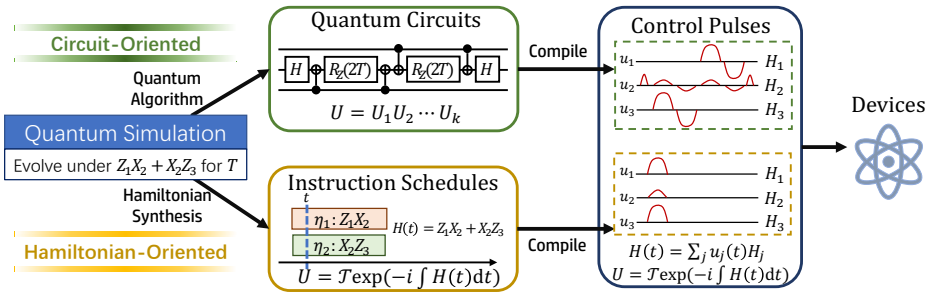


Fig. 1. The circuit-oriented and Hamiltonian-oriented schemes for compiling quantum Hamiltonian simulation on quantum devices. Here $\mathcal{T}\exp(-i\int H(t)\mathrm{d}t)$ is a solution to a Schrödinger equation governed by $H(t)$.

a fundamental factor in the productivity of the underlying programming language. Prominent early examples of such include, e.g., FORTRAN [Backus 1978] and SIMULA [Nygaard and Dahl 1978], both of which provide high-level abstractions for modeling desirable operations for domain applications and have been proven enormous successes in history.

Conventionally, abstractions for quantum computing adopt (qubit-level) quantum circuits to describe procedures, a mathematically simple approach that works well as a mental tool for the theoretical study of quantum information and algorithms [Childs 2017; Nielsen and Chuang 2002]. As a result, many quantum programming languages [Abhari et al. 2012; Green et al. 2013; Hietala et al. 2021; Paykin et al. 2017] have adopted quantum circuits as the only abstraction. Many quantum applications are implemented using these programming languages to generate quantum circuits, although only a few can be demonstrated on existing quantum devices.

Quantum Hamiltonian simulation (also called quantum simulation[1]) is arguably one of the most promising quantum applications. The evolution of a quantum system, starting from a quantum state represented by a high-dimensional complex vector $|\psi(0)\rangle$, obeys the *Schrödinger* equation:

$$\frac{\mathrm{d}}{\mathrm{d}t}|\psi(t)\rangle = -iH(t)\,|\psi(t)\rangle\,, \tag{1.1.1}$$

where $H(t)$ is generally a time-dependent Hermitian matrix, also known as the *Hamiltonian* governing the system. Probing quantum phenomena from solutions of the *Schrödinger* equation is a promising approach to tackle many open problems in various domains, including quantum chemistry, high-energy physics, and condensed matter physics [Cao et al. 2019; Hofstetter and Qin 2018; Nachman et al. 2021]. However, for an n qubit system, the dimension of both $H(t)$ and $|\psi(t)\rangle$ could be 2^n, which makes its classical simulation exponentially difficult in general. Though mature software developments for classical simulation of quantum systems using methods like quantum Monte Carlo [Foulkes et al. 2001] and density-matrix renormalization groups [Schollwöck 2005, 2011] succeed for restricted cases, many intermediate-size (~100 sites) quantum systems of significance are still out of reach for classical computers.

To address this issue, in his famous 1981 lecture, Feynman [1982] suggested employing a precisely controlled quantum system to simulate a target quantum system to avoid exponential complexity. Modern quantum technologies foster a variety of platforms to advance the realization of Feynman's proposal, for example, photonic systems [O'brien et al. 2009], superconducting circuits [Wendin 2017], semiconductor nanocrystals [Kloeffel and Loss 2013], neutral atom arrays [Saffman 2016],

[1]In certain contexts, *quantum simulation* and *quantum simulators* refer to the classical simulation of quantum circuits and the corresponding classical software tools, respectively. Yet throughout this paper, quantum simulation represents the task of simulating a quantum Hamiltonian system, and quantum simulators represent controllable quantum devices that are capable of simulating other quantum systems.

and trapped-ion arrays [Bruzewicz et al. 2019]. Most of them are described by a Hamiltonian with continuous-time parameters characterizing the signals sent through controllable physics instruments like microwaves or magnetic fields. They are called *analog quantum simulators*. Only a few devices support a specific set of system evolutions with sophisticated pulse engineering, abstracted as a set of universal quantum gates [Kitaev 1997], hence called *digital quantum computers*. They include IBM's superconducting devices [Cross 2018] and IonQ's trapped-ion devices [Debnath et al. 2016]. However, inherent noises on near-term digital quantum computers induce detrimental errors causing short coherence time (i.e., quantum states do not deteriorate to classical states within it) and preventing demonstrating large quantum applications with provable speedup. The solution through fault-tolerant quantum computing [Gottesman 2010] requires significantly lower gate implementation errors and better device connectivity, impractical in the NISQ era [Preskill 2018].

In the past decades, efficient quantum algorithms for quantum Hamiltonian simulation have been proposed [Childs 2010; Childs and Wiebe 2012; Lauvergnat et al. 2007; Lloyd 1996; Low and Chuang 2017]. Implementing them follows a circuit-oriented scheme, where the quantum algorithms are designed and programmed as quantum circuits consisting of quantum gates abstracting the evolution of a fraction of sites in the quantum system. Then a quantum circuit compiler rewrites the circuits using a small set of quantum gates and translates each gate to pulses for specific devices. However, both programming and deploying algorithms in this scheme are highly non-trivial. In a seminal project, Childs et al. [2018] spent nearly two years programming a few major quantum simulation algorithms in Quipper [Green et al. 2013] due to tedious implementation details at the circuit level. Meanwhile, [Childs et al. 2018] shows that implementing algorithms via quantum circuits, even for a simple quantum system of medium sizes (around 100 qubits), requires an astronomical number of gates (around 10^{10} before fault-tolerant encoding). Such circuits are far out of the reach of near-term quantum devices, which at most support a few thousand physical gates. The redundancies in programming and deploying algorithms via quantum circuit abstraction impede wide-range domain applications of quantum simulation. Developing better abstractions for quantum Hamiltonian simulation is highly desirable for productivity.

Motivated by the experimental success of simulation by designing and building specific precisely controlled quantum systems mimicking the Hamiltonian of target quantum systems [Ebadi et al. 2021; Gorshkov et al. 2010; Yang et al. 2020; Zohar et al. 2015], programming analog quantum simulators in a Hamiltonian-oriented scheme is a promising approach to quantum applications before fault-tolerant digital quantum computers are manufactured. Instead of programming quantum circuits implementing quantum simulation algorithms, Hamiltonian-oriented schemes directly program Hamiltonians of analog quantum simulators to synthesize an evolution equivalent to the desired quantum system evolution. Analog quantum simulators have native support for generating Hamiltonians, resulting in a succinct translation process to construct pulse schedules. Via Hamiltonian programming, complicated interactions that demand sophisticated quantum algorithms and large quantum circuits to simulate can be natively constructed and simulated on analog quantum simulators. We compare both schemes for quantum simulation in Figure 1 with further details.

By breaking the quantum circuit abstraction and exposing the Hamiltonian-level programmability of modern quantum devices, resource-efficient protocols can deliver reliable solutions to quantum applications [Shi et al. 2020] on NISQ devices, including various devices that do not support universal quantum gates, like QuEra's neutral atom devices.

For example, using the Hamiltonian-level programmability of IBM devices, an evolution governed by $H(t) = Z_1X_2 + X_2Z_3$ for time $T = 1$ (formal definitions in Section 2.1) can be simulated by a pulse schedule with two cross-resonance pulses [Malekakhlagh et al. 2020], as illustrated in Figure 1. Both are 280 nanoseconds long and approximately generate Hamiltonians Z_1X_2 and X_2Z_3, respectively.

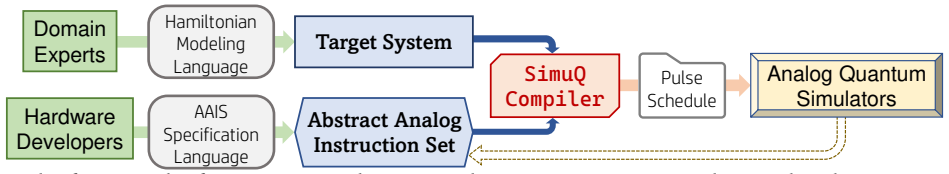


Fig. 2. The framework of SimuQ. Here abstract analog instruction sets are designed and programmed by hardware developers based on the capability of their analog quantum simulators.

Simultaneous execution of these pulses builds $H(t)$ on the IBM device, and the eventual pulse schedule is 280 nanosecond long. As a comparison, the circuit-oriented scheme uses a circuit sequentially applying 4 CNOT gates (each requiring 264 nanoseconds to implement) with several single qubit gates to simulate $H(t)$. It generates a pulse schedule of length 1660 nanoseconds, around 6 times longer. More details of this example are in Section 5.2. Shortening pulse schedule duration is especially desirable because of IBM devices' short coherence time (around 100 microseconds).

Although Hamiltonian-oriented approaches for quantum simulation are beneficial, there is a lack of formal abstractions and supplementary software stacks. Prior works of analog quantum simulation following Hamiltonian-oriented schemes [Ebadi et al. 2021; Yang et al. 2020] manually construct device-specific configurations, which are tedious, error-prone, and demanding for hardware knowledge, hence not suitable for large-scale experiments.

We propose SimuQ with the first end-to-end automatic framework for quantum simulation on general analog quantum simulators, illustrated in Figure 2. As a result, domain experts can focus on describing the desired quantum simulation problems and leave their implementation and deployment to the automation of SimuQ. Our framework lays the foundation for large-scale applications of analog quantum simulators, paving the path for a wide range of novel and practical solutions to domain problems via quantum Hamiltonian simulation for common users.

1.2 Challenges

We identify three main technical challenges in building a framework to compile quantum simulation problems on analog quantum simulators: modeling the target quantum system, characterizing analog quantum simulators, and automatic compilation.

Modeling of quantum Hamiltonian simulation. The first challenge is the lack of a scalable and user-friendly modeling language for quantum simulation. Prior programming languages to model Hamiltonian systems are designed specifically for numerical classical simulations. They treat the sites in the quantum system as a 1-dimensional array for the convenience of constructing matrix-based mathematical objects. One of the most popular languages, QuTiP [Johansson et al. 2012], employs matrices of exponential sizes to represent the quantum system, resulting in poor scalability. Another inconvenience is caused by the mandatory 1-D array labeling of the sites, like in OpenFermion [McClean et al. 2020], Pauli IR [Li et al. 2022], and Qiskit Operator Flow [Aleksandrowicz et al. 2019]. Many quantum systems of interest have complicated site arrangement structures, for example, a 3-dimensional lattice, forcing users to construct the encoding of sites in their system manually.

Abstraction and programming of analog quantum simulators. The modeling of analog quantum simulators is much more challenging. Unlike the circuit model where the fundamental primitives are a finite number of one or two-qubit quantum gates, analog quantum simulators are usually described by continuous-time Hamiltonians on the devices with almost infinite degrees of freedom, which differ significantly among platforms [QuEra 2022; Semeghini et al. 2021; Silvério et al. 2022]. Moreover, complicated pulse engineering using different technologies generates various

Hamiltonians with specific hardware restrictions even for one device. Hence a unifying and portable abstraction is in urgent demand to capture the programmability of analog quantum simulators.

Compilation of quantum simulations on analog quantum simulators. The third challenge is the lack of an automatic compilation procedure. In the circuit-oriented scheme, the primitive gates are small-dimensional matrices. Large quantum evolution could be compiled into these gates with analytical formula (e.g., the Solovay-Kitaev theorem [Kitaev 1997]). In the Hamiltonian-oriented scheme, the goal of compilation is to synthesize pulse schedules for analog quantum simulators where the Hamiltonian governing the device evolution approximately composes the target Hamiltonian $H(t)$. This compilation process needs efficient streamlines for general analog quantum simulators of medium sizes (around 100 sites) and considers realistic hardware constraints.

1.3 Contributions

To the best of our knowledge, SimuQ is the first framework for programming and compiling quantum Hamiltonian simulations on heterogeneous analog quantum simulators. The framework tackles the above three challenges with three major components correspondingly: a new programming language for descriptions of quantum systems, a new abstraction and a corresponding programming language for characterizing the programmability of analog quantum simulators, and a compiler with several novel intermediate representations and compiler passes to deploy and execute solutions to the simulation problems on analog quantum simulators.

Hamiltonian Modeling Language. Without a strong design need for numerical calculations, we propose Hamiltonian Modeling Language (HML), which employs a symbolic representation treating sites as first-class objects and depicts Hamiltonians as algebraic expressions constructed via operators on the sites. This leads to a succinct description that remains rich enough to express many interesting quantum many-body systems. Users can focus on describing complicated quantum systems without tediously handcrafting encoding, reducing the cost of experimenting with new algorithm design ideas. Many quantum systems are programmed in HML with a few lines of code, as illustrated in Section 5.4. Beyond this, developing novel Hamiltonian-oriented quantum algorithms [Leng et al. 2023] can benefit from HML because of the user-friendly description of the algorithms.

Abstract analog instruction sets and AAIS Specification Language. Inspired by the underlying control of these Hamiltonians, we propose a new abstraction called *Abstract analog instruction set* (AAIS) to describe the functionality of heterogeneous analog devices. Precisely, we abstract different patterns of engineered pulses as parameterized *analog instructions*. We expose pieces of Hamiltonian in the AAIS, which are generated by analog pulses on fractions of the system and abstracted as *instruction Hamiltonians* induced by instruction executions. The Hamiltonian governing the evolution of the device at time t is then the summation of instruction Hamiltonians of the instruction executions covering time t.

AAIS exposes the Hamiltonian-level programmability of analog quantum simulators that lies beyond circuit-level abstractions. This feature enables the programming of non-circuit-based controllable quantum devices and further exploits the capability of devices supporting quantum gates within the current hardware limits. Via Hamiltonian-level control, evolution can be simulated by a much shorter pulse duration and become more robust against device noises.

AAIS provides a new formal computational model of quantum devices and unifies the functionality descriptions for different devices with different technologies, simplifying the transfer of quantum simulation solutions to quantum devices of multiple platforms. These descriptions also inform theorists on what Hamiltonian-oriented quantum algorithms are realizable on near-term devices.

We propose several AAISs for QuEra, IonQ, and IBM devices. In general, AAISs should be designed by the hardware providers to expose the Hamiltonian-level control of their devices. We propose and implement AAIS Specification Language (AAIS-SL), a domain-specific language for hardware providers to depict the device programmability. We showcase how to design and program AAISs in AAIS-SL for the mentioned devices in Section 3.2.3.

SimuQ compiler. We propose the first compilation scheme for quantum simulation on analog quantum simulators with several new intermediate representations. We handle the synthesis of instruction executions as symbolic pattern matching inspired by the seminal work in classical analog compilation [Achour and Rinard 2020; Achour et al. 2016]. The instruction executions are then translated to executable pulses by resolving conflicts and reconstructing pulses using device-dependent programming languages and pulse engineering.

To the best of our knowledge, there is no existing compilation framework for heterogeneous analog quantum simulators. Our compiler provides a feasibility demonstration of automatically compiling quantum Hamiltonian simulation on general analog quantum simulators. Although it might not be the ultimate solution, we believe our framework provides a natural and intuitive approach to the modeling and processing of necessary information in constructing executable pulses from simulation problems. The competence of our compiler is demonstrated in Section 5.4 by showing that it can efficiently and reliably generate executable pulses for various domain applications. Pulses generated by SimuQ are executed on real devices and produce reasonable results, which has rarely been demonstrated in previous compiler works for quantum computing due to the abundance of circuit-oriented descriptions. Users can easily transport their quantum simulation experiments among different platforms and devices with our portable design of the compilation framework. It also enables the possibility of benchmarking various quantum devices on significant domain problems solvable via quantum simulation.

In summary, our contributions include:

- We design and implement Hamiltonian Modeling Language in Section 3.1, a succinct DSL for describing quantum Hamiltonian simulation.
  - Programs in HML are short for many important quantum systems, as shown in Section 5.4.
- We design Abstract Analog Instruction Set as a novel abstraction of Hamiltonian-level programmability of analog quantum simulators, as illustrated in Section 3.2. We also implement AAIS Specification Language for hardware providers to design and program AAISs.
  - AAISs enable the programming of non-circuit-based quantum devices.
  - Hamiltonian-level programming shortens pulse schedule duration and thus is more robust to device decoherence errors, with case studies detailed in Section 5.2 and Section 5.3.
- We propose and implement a compiler for quantum simulation on analog quantum simulators in Section 4 with new intermediate representations and compilation passes.
  - SimuQ compiler enables portability among different platforms of analog quantum simulators, and generated pulses are executed on real devices, as demonstrated in Section 5.1.
  - It efficiently compiles many significant quantum systems, as shown in Section 5.4.

Related Works. There are a few Hamiltonian-level programming interfaces for analog quantum simulators, such as IBM Qiskit Pulse [Cross et al. 2022], QuEra Bloqade [QuEra 2022], and Pasqal Pulser [Silvério et al. 2022] developed by hardware service providers. These interfaces are designed to represent the specific underlying quantum hardware rather than to provide a unified interface for all analog quantum simulators like AAIS. Computational quantum physics packages like QuTiP [Johansson et al. 2012] support modeling and numerical calculation of quantum simulation without any compilation to quantum devices. Software tools for quantum Hamiltonian simulation are discussed

extensively for circuit models [Bassman et al. 2022; Li et al. 2022; Powers et al. 2021; Schmitz et al. 2021; Van Den Berg and Temme 2020], while the expressiveness of the circuit abstraction limits their exploitation of analog quantum simulators. SimuQ's solver-based compilation is inspired by the seminal work in classical analog compilation [Achour and Rinard 2020; Achour et al. 2016]. However, the specific abstraction and compilation technique therein is less relevant as the nature of analog quantum devices is very different from classical ones.

2 RUNNING EXAMPLE

We present a realistic but simple example to motivate our framework and showcase the methodology of our approach. Many experiments simulating the Ising model on Rydberg atom arrays are conducted in the literature to probe quantum phenomena barely tractable numerically [Labuhn et al. 2016; Schauss 2018]. We will introduce the mathematical description of these experiments and demonstrate how to automate the process with SimuQ's DSLs, new abstractions, and compiler.

2.1 Quantum Preliminaries

Quantum systems consist of *sites* representing physics objects like atoms, mathematically described by qubits. A *qubit* (or *quantum bit*) is the analogue of a classical bit in quantum computation. It is a two-level quantum-mechanical system described by the Hilbert space $\mathbb{C}^2$. The classical bits "0" and "1" are represented by the qubit states $|0\rangle = \begin{bmatrix}1\\0\end{bmatrix}$ and $|1\rangle = \begin{bmatrix}0\\1\end{bmatrix}$, and linear combinations of $|0\rangle$ and $|1\rangle$ are also valid states, forming a *superpostition* of quantum states. An n-qubit state is a unit vector in the Kronecker tensor product $\otimes$ of n single-qubit Hilbert spaces, i.e., $\mathcal{H} = \otimes_{i=1}^{n}\mathbb{C}^2 \cong \mathbb{C}^{2^n}$, whose dimension is exponential in n. For an n by m matrix A and a p by q matrix B, their Kronecker product is an np by mq matrix where $(A\otimes B)_{pr+u,qs+v} = A_{r,s}B_{u,v}$. The *complex conjugate transpose* of $|\psi\rangle$ is denoted as $\langle\psi| = |\psi\rangle^\dagger$ ($\dagger$ is the Hermitian conjugate). Therefore, the *inner product* of ϕ and ψ could be written as $\langle\phi|\psi\rangle$. We let $\mathrm{Tr}\{M\}$ denote the matrix trace of M.

The time evolution of quantum states is specified by a Hermitian matrix function $H(t)$ over the corresponding Hilbert space, known as the *time-dependent Hamiltonian* of the quantum system. Typical single-site Hamiltonians include the famous *Pauli matrices*:

$$I = \begin{bmatrix}1&0\\0&1\end{bmatrix},\ X = \begin{bmatrix}0&1\\1&0\end{bmatrix},\ Y = \begin{bmatrix}0&-i\\i&0\end{bmatrix},\ Z = \begin{bmatrix}1&0\\0&-1\end{bmatrix}. \tag{2.1.1}$$

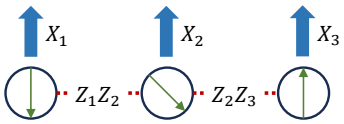
By convention, we write X_j for a multi-site Hamiltonian to indicate $I\otimes\cdots\otimes I\otimes X\otimes I\otimes\cdots\otimes I$, where the j-th operand is X. Similarly, we write Y_j and Z_j. These notations represent operations on the j-th subsystem. A product Hamiltonian P is a tensor product of Pauli matrices, for example, $X\otimes I\otimes Y$, also written as X_1Y_3. A multi-site Hamiltonian can be written as a linear combination of product Hamiltonians, e.g., $H = X_1X_2 + 2Z_2Z_3$. When the product Hamiltonians' coefficients are time functions, they are called time-dependent Hamiltonians, e.g., $H(t) = \cos(t)X$. The product Hamiltonians form a complete basis of n-site Hamiltonians by formula

$$H(t) = \sum_{P\in\{I,X,Y,Z\}^{\otimes n}} \frac{\mathrm{Tr}\{H(t)P\}}{2^n}P. \tag{2.1.2}$$

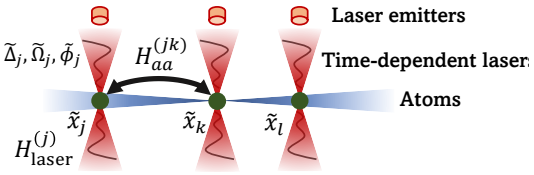
The time evolution of a quantum system under a time-dependent Hamiltonian $H(t)$ obeys the *Schrödinger equation* (1.1.1). Its solution is effectively a unitary matrix function $U(t)$ satisfying

$$\frac{\mathrm{d}}{\mathrm{d}t}U(t) = -iH(t)U(t). \tag{2.1.3}$$

If the system evolves from time 0 with initial state $|\psi(0)\rangle$, the state at time t is $|\psi(t)\rangle = U(t)\,|\psi(0)\rangle$.



(a) An illustration of the Ising model H_{Ising}. Here circles represent qubits, blue arrows, and red dots represent components of H_{Ising}.



(b) An illustration of an ideal Rydberg device.

Fig. 3. Illustrations of the target quantum system and the analog quantum simulator in our running example.

We provide basic physics intuitions of Hamiltonian operations. Hermitian operators correspond to physics effects like the influences of magnetic fields. Scalar multiplication (e.g., $2 \cdot X$) changes the effect strength. Additions of operators (e.g., $X_1 + X_2$) represent simultaneous physics effects, e.g., the superposition of forces. Multiplications of operators (e.g., X_1X_2) represent the interactions across different sites, e.g., the hopping of atoms between different sites.

A *quantum measurement* extracts classical information from quantum systems. When measuring state $|\phi\rangle$, with probability $|\langle s|\phi\rangle|^2$, we obtain a classical bit-string s and the quantum state $|\phi\rangle$ collapses to a classical state $|s\rangle = |s_1\rangle \otimes \ldots \otimes |s_n\rangle$.

2.2 Quantum Simulation of Ising Model

To understand the dynamics and properties of quantum systems, physicists have endless needs to simulate quantum systems. For example, an Ising model is mathematically expressed as

$$H = \sum\nolimits_{1 \le j < k \le n} J_{jk} Z_j Z_k + \sum\nolimits_{j=1}^{n} h_j X_j \tag{2.2.1}$$

where $J_{jk}, h_j \in \mathbb{R}$. This is a significant statistical mechanical model in the study of phase transitions of magnetic systems [Chakrabarti et al. 2008], with a simple example in Figure 3a. In physics, a qubit of an Ising model represents the magnetic dipole moment of an atomic spin. Z_jZ_k represents the interaction between spins j and k, and J_{jk} represents the tendency of align direction agreement between them. X_j represents the effect of an external magnetic field interacting with the spins, and h_j represents its strength. The evolution of a quantum system under Ising models with different parameter regimes of J_{jk} and h_j may characterize the magnetism of materials. However, its simulation generally requires exponential computations for classical computers because of the exponential dimension of the Hilbert space. Instead, we consider its simulation with analog quantum simulators. Nowadays, many controllable quantum systems may be utilized for quantum simulation, and one of the most promising platforms is Rydberg atom arrays [Saffman 2016], where neutral atoms are cooled and precisely controlled by laser beams.

In this section, we focus on the Ising model simulation using Rydberg atom devices, whose large-scale experimental demonstrations are repeated in many laboratories [Bernien et al. 2017; Ebadi et al. 2021; Labuhn et al. 2016; Schauss 2018]. In these demonstrations, experimentalists configure their quantum systems in a task-specific manner. The following illustration showcases these procedures, which are mostly done by manual parameter tuning. This procedure is analogous to the early-day development of classical computers before automated compilers appeared.

We consider a 3-qubit system of the Ising model evolving for time T under

$$H_{\text{Ising}} = Z_1Z_2 + Z_2Z_3 + X_1 + X_2 + X_3, \tag{2.2.2}$$

illustrated in Figure 3a. We want to reproduce the evolution under H_{Ising} on an *ideal Rydberg device*, a simplified Rydberg atom array, illustrated in Figure 3b. Mathematically, the device evolution is governed by $H_{\text{Rydberg}}(\tilde{x}, \tilde{\Delta}, \tilde{\Omega}, \tilde{\phi}, t)$ where $\tilde{x}, \tilde{\Delta}, \tilde{\Omega}, \tilde{\phi}$ are configurable parameters whose details are introduced later, and t is the time variable whose unit is microseconds. The goal of quantum

simulation is to reproduce the evolution of H_{Ising} on the device by finding a configuration for $\tilde{x}, \tilde{\Delta}, \tilde{\Omega}, \tilde{\phi}$ satisfying $H_{\text{Rydberg}}(\tilde{x}, \tilde{\Delta}, \tilde{\Omega}, \tilde{\phi}, t) = H_{\text{Ising}}$, assuming the device evolution time is Tms.

An ideal Rydberg device contains m atoms (viewed as qubits) and m laser beams addressing each atom. The positions of the atoms can be configured arbitrarily on a 1-D line. We denote their coordinates as vector $\tilde{x}$ (unit: μm) and assume they will not move. A Van der Waals force acts between each pair of atoms, whose effect is described by a time-independent Hamiltonian

$$H_{aa}^{(jk)}(\tilde{x}, t) = \frac{C_6}{|\tilde{x}_j - \tilde{x}_k|^6} \hat{n}_j \hat{n}_k. \tag{2.2.3}$$

Here $C_6 \approx 5.42 \times 10^6 \text{MHz} \cdot \mu\text{m}^6$ is a real physics constant and $\hat{n}_j = (I - Z_j)/2$ is the number operator of qubit j. For each atom, there is a local laser beam addressing it. It contains three configurable real-function parameters $\tilde{\Delta}_j(t)$, $\tilde{\Omega}_j(t)$, and $\tilde{\phi}_j(t)$ (unit: MHz) representing the detuning, amplitude, and phase of laser, which can be configured freely over time. It generates an effect described by

$$H_{\text{laser}}^{(j)}(\tilde{\Delta}_j, \tilde{\Omega}_j, \tilde{\phi}_j, t) = -\tilde{\Delta}_j(t)\hat{n}_j + \frac{\tilde{\Omega}_j(t)}{2}\left(\cos\left(\tilde{\phi}_j(t)\right)X_j - \sin\left(\tilde{\phi}_j(t)\right)Y_j\right). \tag{2.2.4}$$

Then the collective Hamiltonian governing the evolution is the summation of effects of Van der Waals forces $H_{aa}^{(jk)}$ and lasers $H_{\text{laser}}^{(j)}$,

$$H_{\text{Rydberg}}(\tilde{x}, \tilde{\Delta}, \tilde{\Omega}, \tilde{\phi}, t) = \sum\nolimits_{1 \le j < k \le m} H_{aa}^{(jk)}(\tilde{x}, t) + \sum\nolimits_{j=1}^{m} H_{\text{laser}}^{(j)}(\tilde{\Delta}_j, \tilde{\Omega}_j, \tilde{\phi}_j, t). \tag{2.2.5}$$

A manual way to find a device configuration is to match the coefficients of product Hamiltonians in H_{Ising} by configuring the parameters. Note that Z_jZ_{j+1} of H_{Ising} is a 2-qubit interaction which only comes from $H_{aa}^{(jk)}$. We configure $\tilde{x}_j$ accordingly by setting $\tilde{x}_j = (j-1)\times 10.52$ so that $H_{aa}^{(j(j+1))}(\tilde{x}, t) = Z_jZ_{j+1} - Z_j - Z_{j+1} + I$. Note by setting $\tilde{x}_j$, the system has unwanted $H_{aa}^{(13)}(\tilde{x}, t) = 0.016 \cdot (Z_1Z_3 - Z_1 - Z_3 + I)$. We then configure the local laser beams to create the X_j terms in H_{Ising} and compensate the unwanted Z_j terms in $H_{aa}^{(jk)}$ by setting $\tilde{\Delta}_1(t) \equiv \tilde{\Delta}_3(t) \equiv 2.032$, $\tilde{\Delta}_2(t) \equiv 4$, $\tilde{\Omega}_j(t) \equiv 2$ and $\tilde{\phi}_j \equiv 0$. We can confirm our synthesis by checking $H_{\text{Ising}} - H_{\text{Rydberg}}(\tilde{x}, \tilde{\Delta}, \tilde{\Omega}, \tilde{\phi}, t) = -0.016Z_1Z_3 + 2.016I$. Since $2.016I$ has no measurable effects on the evolved state by quantum information analysis, The error term is $-0.016Z_1Z_3$, which is small compared to H_{Ising}.

2.3 Automated Compilation by SimuQ

SimuQ provides automation to the above procedure for analog quantum simulators by establishing a workflow via new abstractions, intermediate representations, and compilation passes designed explicitly for analog compilation of quantum simulation. We illustrate how to program and compile H_{Ising} on the ideal Rydberg device in SimuQ, with a glimpse of our new abstractions.

Programming an Ising evolution. Firstly, we program H_{Ising} in HML with an implementation in Python as in Figure 4a. The first step is to declare a quantum system `Ising` (Line 1) and three sites (qubit) belonging to it (Line 2-3). By storing the sites in a list, we refer to the j-th site of the system by `q[j]`. Then we construct H_{Ising}'s terms one by one and store them in `h` (Line 4-8). Here we program the terms as an expression containing operators on the sites, e.g., X_j as `q[j].X` and Z_jZ_{j+1} as `q[j].Z*q[j+1].Z`. We then let the system evolve under `h` for time `T` (Line 9).

Characterizing ideal Rydberg devices. We propose a Rydberg AAIS to characterize the programmability of ideal Rydberg devices. An implementation is in Figure 4b.

The program starts with declaring the quantum device (Line 1) and its sites (Line 2). We can construct the number operators $\hat{n}_j$ and store them in `n` (Line 3).

We propose *analog instructions* to characterize the effects produced and configured on the device over time. An instruction execution generates an instruction Hamiltonian that adds up to the total Hamiltonian governing the system. In the Rydberg AAIS, we design instructions η_j to model the effects of the laser beam pulse signals (Line 5). Executing η_j generates an instruction Hamiltonian $\{\!|\eta_j|\!\}(\Delta_j, \Omega_j, \phi_j) = \Delta_j \hat{n}_j + \Omega_j/2(\cos(\phi_j)X_j - \sin(\phi_j)Y_j)$, where Δ_j, Ω_j, and ϕ_j are *local variables* belonging to η_j (Line 6-9). When executing η_j, one may specify a valuation $\vec{b}$ and execution starting and ending time τ_s, τ_e to induce a Hamiltonian $\{\!|\eta_j|\!\}(\vec{b})$ on the device in time interval $[\tau_s, \tau_e)$.

Instructions can be executed simultaneously to create an evolution under their collective effects, mathematically expressed as a summation of instruction Hamiltonians. For example, we can simultaneously switch on the laser beams addressing atoms 1 and 2 with configuration $\vec{b}_1$ and $\vec{b}_2$ and switch off others, generating a Hamiltonian $\{\!|\eta_1|\!\}(\vec{b}_1) + \{\!|\eta_2|\!\}(\vec{b}_2)$.

Besides the instructions executed over time, analog quantum simulators may also have inherent effects, like the atom-atom interactions in the Rydberg atom devices. We declare a *system Hamiltonian* with *global variables* for them. Let x_j be the global variables representing the position of the atoms (Line 10). Then collective Van der Waals force $\sum_{1\le j<k\le m} H_{aa}^{(jk)}(x,t) = \sum_{1\le j<k\le m} C_6/|x_j - x_k|^6 \hat{n}_j \hat{n}_k$ is characterized as the system Hamiltonian $H_{\text{sys}}(\vec{x})$ (Line 11-15).

Overall, the Hamiltonian governing an ideal Rydberg device at time t is

$$H_{IRD}(t) = H_{\text{sys}}(\vec{x}) + \sum\nolimits_{(\eta_j, \vec{b}_j)\in C_t} \{\!|\eta_j|\!\}(\vec{b}_j), \tag{2.3.1}$$

where C_t contains the active instruction executions at time t and their variable valuations.

Synethsizing H_{Ising} on ideal Rydberg devices. The SimuQ compiler automatically synthesizes a target Hamiltonian with an AAIS and generates executable pulses for devices. We go through the compilation steps on a 3-atom ideal Rydberg device, creating a configuration satisfying $H_{IRD}(t) = H_{\text{Ising}}$.

The first step is to find a site layout between the Hilbert space of H_{Ising} and the Hilbert space of ideal Rydberg devices. A trivial layout that maps the j-th site of H_{Ising} to the j-th atom of the ideal Rydberg device suffices since the atoms are homogeneous.

For simplicity, here we assume the on-device evolution time is the target evolution time T. We then synthesize H_{Ising} by matching the coefficients of its product Hamiltonians. For a product Hamiltonian P, let $H[P]$ be the coefficient of P in Hamiltonian H and $\{\!|\eta_j|\!\}[P]$ be the coefficient function of P in $\{\!|\eta_j|\!\}$. We take product Hamiltonian Z_1 as an example, whose coefficient is $H_{\text{Ising}}[Z_1] = 0$. Z_1 has

```
1  Ising = QSystem()
2  q = [Qubit(Ising)
3          for i in range(N)]
4  h = 0
5  for i in range(N) :
6      h += q[i].X
7  for i in range(N - 1) :
8      h += q[i].Z * q[i+1].Z
9  Ising.add_evolution(h, T)
```

(a) An evolution governed by H_{Ising} (2.2.2) programmed in HML. Here $N = 3$ is the number of sites. $T = 1$ is the evolution time.

```
1  Rydberg = QMachine()
2  q = [qubit(Rydberg) for i in range(N)]
3  n = [(q[i].I - q[i].Z) / 2 for i in range(N)]
4  for i in range(N) :
5      η = Rydberg.add_instruction()
6      add_lVar = η.add_local_variable
7      Δ, Ω, ϕ = add_lVar(), add_lVar(), add_lVar()
8      X_Y = cos(ϕ) * q[i].X - sin(ϕ) * q[i].Y
9      η.set_ham(-Δ *n[i] + Ω / 2 * X_Y)
10 x = [Rydberg.add_global_variable() for i in range(N)]
11 h = 0
12 for i in range(N) :
13     for j in range(i) :
14         h += (C / (x[i] - x[j])**6) * n[i] * n[j]
15 Rydberg.set_sys_ham(h)
```

(b) The Rydberg AAIS programmed in AAIS-SL. $N = 3$ is the number of sites. C is the Rydberg interaction constant.

Fig. 4. Examples of HML and AAIS specification language implemented in Python.

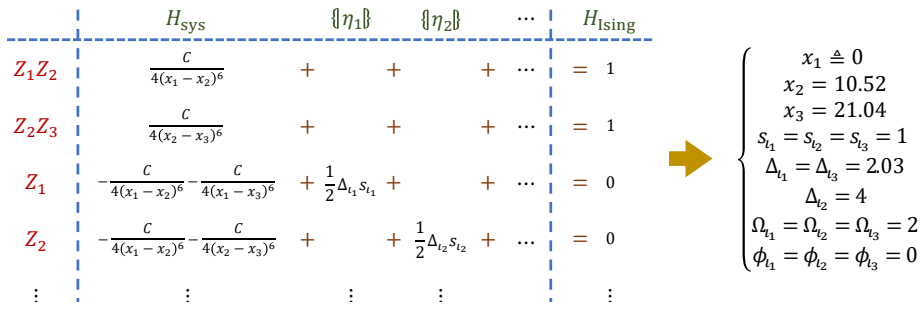


Fig. 5. On the left is the equation system to synthesize H_{Ising} using the AAIS for the ideal Rydberg device. On the right is an approximate solution to it. It can be further interpreted as a pulse schedule in Bloqade.

non-zero coefficient functions in $\{\!|\eta_1|\!\}$ and the system Hamiltonian H_{sys}

$$\{\!|\eta_1|\!\}[Z_1](\Delta_1, \Omega_1, \phi_1) = \frac{\Delta_1}{2}, \qquad H_{\text{sys}}[Z_1](\vec{x}) = -\frac{C_6}{4|x_1 - x_2|^6} - \frac{C_6}{4|x_1 - x_3|^6}. \tag{2.3.2}$$

We want a set of instruction executions letting the coefficient of Z_1 be 0. Since instruction η_1 is optionally executed, an indicator variable $s_1 \in \{0, 1\}$ is declared to represent whether η_1 is executed. Then we establish an equation

$$H_{\text{sys}}[Z_1] + \{\!|\eta_1|\!\}[Z_1] \cdot s_1 = H_{\text{Ising}}[Z_1] \quad \Leftrightarrow \quad -\frac{C_6}{4|x_1 - x_2|^6} - \frac{C_6}{4|x_1 - x_3|^6} + \frac{\Delta_1}{2} s_1 = 0. \tag{2.3.3}$$

In general, we declare $s_j \in \{0, 1\}$ for each instruction η_j and establish an equation system

$$\forall P \neq I, \quad H_{\text{sys}}[P] + \sum\nolimits_j \{\!|\eta_j|\!\}[P] \cdot s_j = H_{\text{Ising}}[P] \tag{2.3.4}$$

by enumerating every P to match all coefficients of product Hamiltonians in H_{Ising}. Figure 5 shows other established equations, and the full equation system is in the extended version [Peng et al. 2023b] Appendix.

We employ a numerical solver to search for a solution to the non-linear mixed binary equation system. An approximate solution to the equation system is displayed in Figure 5. We interpret the solution as an instruction schedule: it specifies a collection of instruction executions according to the solutions to s_j and local variables. The solution in Figure 5 can then be interpreted: set the positions of atoms at $x = [0, 10.52, 21.04]\mu m$, set laser beams configuration $(\Delta(t), \Omega(t), \phi(t)) \equiv (2.032, 2, 0)$ for atom 1 and 3 and $(\Delta(t), \Omega(t), \phi(t)) \equiv (4, 2, 0)$ for atom 2, and evolve the system for Tms. These configurations can be translated to a Bloqade or Braket program to execute on QuEra devices.

With the above procedure, we successfully simulate the evolution under H_{Ising} on the ideal Rydberg device with the help of SimuQ. In practice, hardware providers design AAIS and implement the analog instructions for their specific devices. Front-end users only need to program H_{Ising} and employ SimuQ to generate executable code to send to backend devices. SimuQ breaks the knowledge barriers for frontend users to exploit analog quantum simulators easily. The following sections will explicate SimuQ components and technical details.

3 DOMAIN-SPECIFIC LANGUAGES

SimuQ is the first framework to tackle quantum simulation with Hamiltonian-level compilation to analog quantum simulators. It includes a collection of novel abstractions and domain-specific languages (DSL). We propose two DSLs in SimuQ: Hamiltonian Modeling Language (HML) for

$$A \in \text{Site}, \qquad r \in \mathbb{R}, \tau \in \mathbb{R}^+$$

$$\begin{aligned}
R \in \text{Operator} \quad & ::= I \mid X \mid Y \mid Z \\
S \in \text{Scalar} \quad & ::= S_1 + S_2 \mid S_1 \cdot S_2 \mid S_1 - S_2 \mid S_1/S_2 \\
& \quad \mid \exp(S) \mid \cos(S) \mid \sin(S) \mid r \\
M \in \text{Hermitian} \quad & ::= M_1 + M_2 \mid M_1 \cdot M_2 \mid S \cdot M \mid A.R \\
E \in \text{Evolution} \quad & ::= \mathbf{nil} \mid (M, \tau); E
\end{aligned}$$

(a) Abstract syntax of HML.

$$\begin{aligned}
h_{A.R} &= R_A \\
h_{S \cdot M} &= \text{eval}(S) \cdot h_M, \\
h_{M_1+M_2} &= h_{M_1} + h_{M_2}, \\
h_{M_1 \cdot M_2} &= h_{M_1} \cdot h_{M_2}, \\
[\![\mathbf{nil}]\!] &= I, \\
[\![(M, \tau); E]\!] &= [\![E]\!] \cdot e^{-i\tau h_M}.
\end{aligned}$$

(b) Semantics of HML.

Fig. 6. Syntax and denotational semantics of HML. Here Site contains system sites. h_M translates to the Hermitian matrix described by M. R_A is a Hermitian matrix where operator R applies to site A and I applies to other sites. Function eval evaluates scalar expression S to a real number.

front-end users to depict their target quantum systems and AAIS Specification Language (AAIS-SL) to specify analog abstract instruction sets (AAISs) of analog quantum simulators.

3.1 Hamiltonian Modeling Language

HML is a DSL designed to describe the physical structure of many-body quantum systems that introduces many abstractions, including quantum sites and site-based representations of Hamiltonians. We implement this language in Python, with its abstract syntax and denotational semantics formally defined in Figure 6.

Abstract Syntax of HML. The first-class objects in HML are *sites of quantum systems*. A site is an abstraction for any quantized 2-level physical entity, like atoms with two energy levels, whose mathematical description is a qubit. In HML, site identifiers are collected in a set Site, each representing a site of the system. Four operators, I, X, Y, and Z, are defined to represent the Pauli operators, and they are *site operators*. We denote the X operator of qubit q as $q.X$ and other operators similarly.

A time-independent Hamiltonian is effectively a Hermitian matrix programmed by algebraic expressions. The basic elements are site operators $A.R$. Expressions for Hermitian are constructed using site operators and scalar expressions, consisting of common matrix operations and scalar operations. An evolution E in HML is a sequence of pairs (M, τ), representing a sequential evolution where each segment is governed by a time-independent Hamiltonian h_M and for time τ.

Remark 3.1. *Beyond sites representing qubits, sites representing fermionic and bosonic modes can be defined together with their annihilation and creation operators. These are characterized by different types of sites in our implementation. Each type of site contains specific site operators, and the operator algebras are symbolically implemented. We omit formal discussions of them in this paper for simplicity.*

Remark 3.2. *HML can generally deal with Hamiltonians with continuous-time coefficients by introducing an additional identifier t in scalars. We choose sequences of time-independent evolution for numerical convenience in the compilation stage and leave this possibility for the future.*

Semantics of HML. The denotational semantics of a program E in HML is interpreted as a unitary matrix by $[\![E]\!]$ in Figure 6b. We let h_M translate program M into Hermitian matrices by evaluating the expressions. Then $[\![E]\!]$ is the product of unitary matrices $e^{-i\tau h_M}$, each representing the solution to the Schrödinger equation under $H(t) = h_M$ for time duration τ. This is the solution to the Schrödinger equation governed by the piecewise-constant Hamiltonian programmed in E.

Implementation of HML. We implement HML in Python to ensure accessibility to physicists and other common users. For a quantum system, we store the sites in a list. A product Hamiltonian

P is then stored as a list of site operators using the same order of the site list. We also employ a Python dictionary to store a time-independent Hamiltonian H where the key-value pairs are made of a product Hamiltonian P and its coefficient denoted by $H[P]$. Mathematically, $H[P] = \mathrm{Tr}\{H \cdot P\}$. We only store those P with non-zero $H[P]$ to compactly store Hamiltonians. For example, H_{Ising} in Section 2.3 is represented by a dictionary $\{Z_1Z_2 : 1,\ Z_2Z_3 : 1,\ X_1 : 1,\ X_2 : 1,\ X_3 : 1\}$.

To deal with the algebraic operations of Hermitian matrices, we symbolically implement an algebraic group for site operators (the Pauli group), and then Hermitian expressions are evaluated accordingly. For example, $H_1 + H_2$ is effectively implemented by enumerating P appearing in the keys of H_1's and H_2's dictionary, and construct $(H_1 + H_2)[P] = H_1[P] + H_2[P]$. Another example is multiplication, where $H_1 \cdot H_2$ is implemented by enumerating P_j in H_j's dictionary keys. Since the site operators of different sites commute and those of the same sites are in a finite group, $P_1 \cdot P_2$ is a product Hamiltonian P with an additional scalar multiplier p (i.e., $(X_1X_2) \cdot (Y_1Y_2) = -1 \cdot Z_1Z_2$). We add $p \cdot H_1[P_1] \cdot H_2[P_2]$ to the coefficient $(H_1 \cdot H_2)[P]$. Then we represent the evolution E as a list of tuples (H, τ) encompassing Hermitian matrix H and the evolution time τ of an evolution segment.

Input Discretization Error. In many-body physics systems, Hamiltonians are commonly continuous, taking form $H_{\mathrm{tar}}(t) = \sum_{k=1}^{K} \alpha_k(t) H_k$. In HML, these Hamiltonians are discretized into a series of piecewise time-independent Hamiltonians in the input. Let the evolution duration be T and the discretization number be D. We discretize $H_{\mathrm{tar}}(t)$ over time steps $\{t_d\}_{d=1}^{D}$ where $0 < t_1 < \ldots < t_D < T$ and use the left endpoint of each interval as its approximation. Formally, $H_{\mathrm{tar}}(t)$ is approximated by

$$\tilde{H}(t) = \sum\nolimits_{k=1}^{K} \tilde{\alpha}_k(t) H_k, \qquad \tilde{\alpha}_k(t) = \sum\nolimits_{d=1}^{D} \alpha_k(t_d) \mathbb{1}_{[t_d, t_{d+1})}(t), \tag{3.1.1}$$

where $\mathbb{1}_{[a,b)}$ is the indicator function of set $[a, b)$. We assume $\|H_k\| = 1$ where $\|\cdot\|$ is the spectral norm of matrices, $\alpha_k(t)$ are piecewise M-Lipschitz functions, and $\{t_d\}_{d=1}^{D}$ include all partitioning points of the piecewise Lipschitz coefficients $\alpha_k(t)$. Then we can derive the error bound induced by discretization by the following lemma.

Lemma 3.1 ([Nielsen and Chuang 2002]). *The difference between the unitary $U(T)$ of evolution under $H_{\mathrm{tar}}(t)$ for duration T and the unitary $\tilde{U}(T)$ of evolution under $\tilde{H}(t)$ is bounded by*

$$\left\| U(T) - \tilde{U}(T) \right\| \le C_1 D^{-1} M K T^2. \tag{3.1.2}$$

Here $C_1 > 0$ is a constant, D is the discretization number, K is the number of terms in $H_{\mathrm{tar}}(t)$, and L is the Lipschitz constant for $\alpha_k(t)$.

This lemma shows that when we increase the discretization number D, the evolution error in the approximation can be arbitrarily small, justifying the discretization. The proof is routine in quantum information and hence omitted.

3.2 Abstract Analog Instruction Set and AAIS Specification Language

3.2.1 Abstract Analog Instruction Set. An AAIS conveys the functionality of an analog quantum simulator in the form of instructions and system Hamiltonians, including necessary device information for synthesizing target quantum systems.

We present the AAIS design and their physics correspondences in Table 1. An *analog instruction* η of an AAIS contains configurable parameters $\vec{v}$ and generates an *instruction Hamiltonian* $H_\eta(\vec{v})$ on the device when executed. These parameters are *local variables* of η, modeling the device parameters that can change over time. The instruction Hamiltonian $H_\eta(\vec{v})$ takes the following form where $u_P(\vec{v})$ is a real function depending on the local variables $\vec{v}$:

$$H_\eta(\vec{v}) = \sum\nolimits_{P} u_P(\vec{v}) \cdot P. \tag{3.2.1}$$

$$A \in \text{Site},\ v[q] \in \text{Var}^q \text{ for } q \in \{L, G\},\ r \in \mathbb{R}$$

$$\begin{aligned}
R \in \text{Operator} \quad & ::= I \mid X \mid Y \mid Z \\
S^q \in \text{Para. Scalar}^q \quad & ::= S_1^q + S_2^q \mid S_1^q \cdot S_2^q \mid S_1^q - S_2^q \mid S_1^q / S_2^q \\
& \quad \mid \exp(S^q) \mid \cos(S^q) \mid \sin(S^q) \mid r \mid v[q] \\
M^q \in \text{Para. Herm.}^q \quad & ::= M_1^q + M_2^q \mid M_1^q \cdot M_2^q \mid S^q \cdot M^q \mid A.R \\
D \in \text{Device} \quad & ::= M^G \mid M^L; D
\end{aligned}$$

(a) Syntax of AAIS specification language.

$$\begin{aligned}
h_{A.R} &= R_A, \\
h_{S^q \cdot M^q} &= \overline{\text{eval}}(S^q) \cdot h_{M^q}, \\
h_{M_1^q + M_2^q} &= h_{M_1^q} + h_{M_2^q}, \\
h_{M_1^q \cdot M_2^q} &= h_{M_1^q} \cdot h_{M_2^q}, \\
\{\!|M^G|\!\} &= h_{M^G}, \\
\{\!|M^L; D|\!\} &= h_{M^L}; \{\!|D|\!\}
\end{aligned}$$

(b) Semantics of AAIS programs.

Fig. 7. Abstract syntax and denotational semantics of AAIS-SL. Here Site contains the sites of the device. Var^L and Var^G contain the local and global variables correspondingly. $\overline{\text{eval}}(S)$ evaluates S as a real function.

Table 1. Comparison among physics concepts, Rydberg devices instances, and AAIS abstraction designs.

Physics	Signal carriers	Pulse signals	Signal effects	Device evolution
Rydberg devices	Laser emitters	Time-dependent lasers	$H_{\text{laser}}^{(j)}$	Obeys $H_{\text{Rydberg}}(t)$
AAIS	Signal lines	Instructions	Instruction Hamiltonians	Total Hamiltonian

Additionally, a *system Hamiltonian* $H_{\text{sys}}(\vec{v}_{\text{glob}})$ with a similar form of (3.2.1) applies an always-on effect on the device. A vector $\vec{v}_{\text{glob}}$ of time-independent configurable parameters, called *global variables*, belongs to it. These global variables are configured before executing any instructions and stay unchanged during the execution.

3.2.2 AAIS Specification Language. To specify AAISs with programs, we propose and implement AAIS-SL and present its abstract syntax and denotational semantics in Figure 7.

Abstract Syntax of AAIS-SL. To characterize the Hamiltonians of instructions, sites are declared with identifiers stored in a set Site, and site operators are defined as objects of sites by default.

Compared to HML, the major difference in the syntax is variables. Two types of variables whose identifiers are stored in Var^G and Var^L represent global variables and local variables, respectively. They are terms in parameterized scalars and consist of parameterized Hermitians. Then an AAIS for a device is effectively a collection of instruction Hamiltonians as parameterized Hermitian matrices, along with the system Hamiltonian.

Denotational Semantics of AAIS-SL. We interpret an AAIS D characterizing a device as a list of instructions along with the system Hamiltonian. Similar to the HML semantics, we employ a translation h for expressions S to obtain parameterized Hermitians. Function $\overline{\text{eval}}$ evaluates a parameterized scalar expression S as a real function taking a valuation of variables and outputting a real number. Hence h translates parameterized Hermitian expressions to Hamiltonians in the form of (3.2.1). Without ambiguity, we use $\{\!|\eta|\!\}(\vec{v})$ to represent the instruction Hamiltonian of η.

Implementation of AAIS-SL. We also provide a Python implementation of AAIS-SL. We store sites and Hermitian matrices similarly to the implementation of HML. The difference is that instead of storing real numbers as coefficients, we store Python functions taking global variable and local variable valuations as inputs. We build function algebraic operations (i.e., $(f_1 + f_2)(x) = f_1(x) + f_2(x)$ and $(f_1 \cdot f_2)(x) = f_1(x) \cdot f_2(x)$) to deal with expressions and establish the parameterized Hermitian matrix expressions. As described in Figure 7, an AAIS is effectively represented by a system Hamiltonian and a list of instruction Hamiltonians.

3.2.3 Examples of AAIS. Through AAISs, we provide a general framework to characterize the programmability of analog quantum simulators. Here we show how we design AAISs for QuEra,

IonQ, and IBM devices. The design of AAIS abstraction pursues a balance between expressiveness and implementation hardness on real devices: to simulate more complicated quantum systems, more complicated instructions are needed, requiring more advanced technologies in their implementation.

Rydberg AAIS. The Rydberg AAIS designed for ideal Rydberg atom devices is introduced in Section 2.3. However, current QuEra devices do not support local laser addressing, meaning only a global laser interacts with every atom simultaneously. We use a variant for QuEra devices, called the global Rydberg AAIS, where there is only one instruction η with the instruction Hamiltonian

$$\{\!|\eta|\!\}(\Delta, \Omega, \phi) = -\Delta \sum_{j=1}^{m} \hat{n}_j + \frac{\Omega}{2} \sum_{j=1}^{m} (\cos(\phi) X_j - \sin(\phi) Y_j). \tag{3.2.2}$$

Heisenberg AAIS. The IonQ and IBM devices, though using different platforms, share similar capabilities for constructing interactions. For both platforms, the Heisenberg AAIS is designed and implemented, which contains 1-site instructions $\eta_{j,P}$ and 2-site instructions $\eta_{j,k,PP}$ for $j, k \in \{1, ..., n\}$ and $P \in \{X, Y, Z\}$, where n is the number of sites. Each instruction possesses one local variable, and their instruction Hamiltonians are:

$$\{\!|\eta_{j,P}|\!\}(a) = a \cdot P_j, \qquad \{\!|\eta_{j,k,PP}|\!\}(a) = a \cdot P_j P_k. \tag{3.2.3}$$

Here, the 1-site instructions $\eta_{j,P}$ are defined for every site in the system, and the 2-site instructions $\eta_{j,k,PP}$ are only defined when $(j, k) \in E$ for an undirected connectivity graph E representing the connectivity of the detailed device. For ion trap devices, E is a complete graph with an edge between each site pair. Superconducting devices typically have limited connectivity, and we let E be the connectivity graph of the IBM devices.

The Heisenberg AAIS can simulate a family of Heisenberg models [Auerbach 1998] covering the Ising models. A variant of the Heisenberg AAIS called the 2-Pauli AAIS extends the 2-site interactions to $P_j Q_k$ interactions for $P, Q \in \{X, Y, Z\}$, is capable of simulating more quantum systems, and is realizable on IonQ and IBM devices with specific connectivity.

IBM-Native AAIS. Besides the Heisenberg AAIS, for the IBM devices, we can also model their native effects in an IBM-native AAIS. Its 2-site instructions are $\eta_{j,k,CR}$ for $(j, k) \in E$ where

$$\{\!|\eta_{j,k,CR}|\!\}(\Omega) = \omega_{ZX} \Omega Z_j X_k + \omega_{ZZ} Z_j Z_k + \omega_{IX} \Omega X_k + \omega_{ZI} \Omega^2 Z_j, \tag{3.2.4}$$

where $\omega_{ZX}, \omega_{ZZ}, \omega_{IX}$, and ω_{ZI} are device-dependent constants. Instruction $\eta_{j,k,CR}$ and $\eta_{l,k,CR}$ can be simultaneously executed on IBM devices because of platform features. However, since it contains multiple terms with limited freedom of control, only a few quantum systems can be directly simulated by the IBM-native AAIS. For the systems that can be simulated, a much shorter pulse duration can be produced. A more detailed analysis is in Section 5.2.

4 INTERMEDIATE REPRESENTATIONS AND COMPILATION

Compiling a target quantum system to an analog quantum simulator is computationally hard in most cases, especially when we aim at a general framework. In this section, we build the first compiler for quantum simulation on general analog quantum simulators and several novel intermediate representations to conquer various challenges in the overall compilation.

The overall compilation workflow is presented in Figure 8. Since this is the first exploration of compilation to heterogeneous analog quantum simulators, our proposal intuitively decomposes the problem into several natural sub-problems that are rarely encountered in prior works and applies straightforward solutions to each step. Much space for optimizing our workflow within the scope of our approach is left for future work, which is discussed in Section 6.

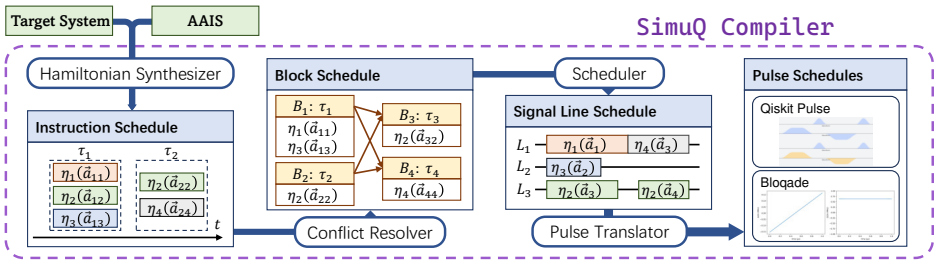


Fig. 8. An illustration of the SimuQ compilation process.

4.1 Instruction Schedules and Hamiltonian Synthesizer

The first intermediate representation is instruction schedules that describe the execution of instructions on the device. We will also introduce a Hamiltonian synthesizer to create an instruction schedule that simulates a target quantum system.

4.1.1 Instruction Schedules. An instruction execution $(\eta, \vec{a}, \tau_s, \tau_e)$ specifies an instruction η, a valuation $\vec{v} \mapsto \vec{a}$ of η's local variables, and evolution starting time τ_s and ending time τ_e. It applies a Hamiltonian $H_\eta(\vec{a})$ to the device during $[\tau_s, \tau_e)$. In later cases when the absolute starting time and ending time are unimportant, we also use duration $\tau_d = \tau_e - \tau_s$ in instruction executions.

An instruction schedule includes a valuation $\vec{g}$ to the global variables and a set of instruction executions $\{(\eta_j, \vec{a}_j, \tau_{s,j}, \tau_{e,j})\}$. At the time t, the instruction executions satisfying $\tau_{s,j} \le t < \tau_{e,j}$ generate effects on the device. Executing the instruction schedule evolves the device, governed by:

$$H(t) = H_{\text{sys}}(\vec{g}) + \sum\nolimits_{j:\tau_{s,j} \le t < \tau_{e,j}} H_{\eta_j}(\vec{a}_j). \tag{4.1.1}$$

We use a more succinct representation of the instruction schedules generated by our Hamiltonian synthesizer. We characterize the set of instruction executions as a list $\mathcal{S} = [(C_j, \tau_j)]_{j=1}^m$ where $C_j = \{(\eta_{jk}, \vec{a}_{jk})\}_k$. $\mathcal{S}$ denotes a sequential evolution of simultaneous instruction executions in C_j for time duration τ_j. Let $T_j = \sum_{k \le j} \tau_j$ and assume $T_0 = 0$. The absolute starting and ending time of instruction execution $(\eta_{jk}, \vec{a}_{jk}) \in C_j$ are then T_{j-1} and T_j. Mathematically, the Hamiltonian $H(t)$ governing the evolution of the device at time $t \in [T_{j-1}, T_j)$ is $H(t) = H_{\text{sys}}(\vec{g}) + \sum_k \{\!|\eta_{jk}|\!\}(\vec{a}_{jk})$. As a solution to the Schrödinger equation, the execution of instruction schedule $(\mathcal{S}, \vec{g})$ results in an evolution of the device described by a unitary matrix

$$U(T_m) = \prod\nolimits_{j=m}^{1} e^{-i\tau_j \left(H_{\text{sys}}(\vec{g}) + \sum_k \{\!|\eta_{jk}|\!\}(\vec{a}_{jk})\right)}. \tag{4.1.2}$$

4.1.2 Quantum Simulation by Executing Instruction Schedules. We formally define the task of compiling quantum simulations to a quantum device described by an AAIS. Consider a target quantum system described by a Hamiltonian $H_{\text{tar}}(t)$ and evolution time interval $[0, T)$. Compilation of a quantum simulation asks for a site layout L and an instruction schedule $(\mathcal{S}, \vec{g})$. A site layout L is an injective mapping from each site in the target system to a site in the device system. We call the Hilbert space of the sites mapped to by L the layout subspace of the device Hilbert space. A layout L induces a mapping $\mathcal{L}$ from the target system's Hilbert space to the layout subspace. When limiting $\mathcal{L}(H)$ in the layout subspace where H is a Hermitian matrix in the target Hilbert space, one can relabel the sites of $\mathcal{L}(H)$ according to L^{-1} and recover H. When $H(t)$ is a time-dependent Hamiltonian of the target Hilbert space, we write $\mathcal{L}(H)$ as a Hamiltonian of the device Hilbert space satisfying $\mathcal{L}(H)(t) = \mathcal{L}(H(t))$. Let the execution of the instruction schedule $(\mathcal{S}, \vec{g})$ produce a unitary matrix U and let the evolution under $\mathcal{L}(H_{\text{tar}})$ for time interval $[0, T)$ be $\mathcal{L}(U_{\text{tar}})$. We say that a site layout L and the instruction schedule $(\mathcal{S}, \vec{g})$ simulate $H_{\text{tar}(t)}$ if U approximates $\mathcal{L}(U_{\text{tar}})$.

Algorithm 1 Equation builder for Hamiltonian synthesis.

Inputs: site layout mapping $\mathcal{L}$, target quantum system $(H_{\text{tar},j}, \tau_j)$ for $1 \le j \le N$, AAIS $D = [\eta_1, ..., \eta_M, H_{\text{sys}}]$, equation system variables $\{\vec{a}_{k,j}\}, \{s_{k,j}\}, \vec{g}, \{t_j\}$
Output: a system of equations Υ

```
Υ ← {}
for j ∈ {1, ..., N} do
    G ← {ℒ(H_{tar,j})}_{j=1}^{N} ∪ {H_sys}
    Q ← [P | ∃H ∈ G, H[P] ≠ 0]
    i ← 0
    while i < |Q| do
        P ← Q.getitem(i)
        i ← i + 1
        if P = I then
            continue
        e ← H_{tar,j}[P](g⃗)
        for k ∈ {1, ...M} do
            if {|η_k|}[P] ≠ 0 then
                e ← e + {|η_k|}[P](a⃗_{k,j}) · s_{k,j}
                for P' ∉ Q : {|η_k|}[P'] ≠ 0 do
                    Q.append(P')
        Υ.add(t_j · e = τ_j · ℒ(H_{tar,j})[P])
```

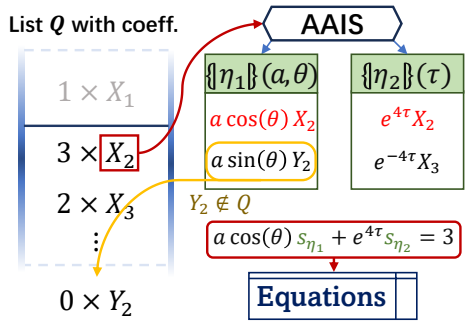


Fig. 9. An example illustrating the equation builder. X_2 is searched for in AAIS, where η_1 and η_2 are found and used in equation for X_2.

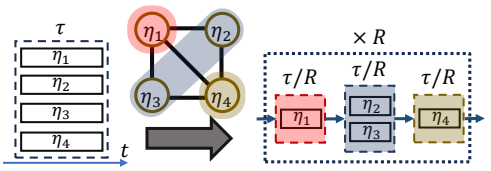


Fig. 10. An example of Trotterization from an instruction schedule to a block schedule, with a conflict graph and its grouping.

4.1.3 Hamiltonian Synthesizer. Since HML discretizes continuous Hamiltonians with small errors, in this step, we consider a target quantum system described by a sequence of evolution under $H_{\text{tar},j}$ for time duration τ_j indexed by $j \in \{1, ..., N\}$. We want to synthesize an instruction schedule simulating the target quantum system on a device described by an AAIS $D = [\eta_1; ...; \eta_M; H_{\text{sys}}]$. Our Hamiltonian synthesizer follows a three-step loop: (1) propose a site layout L; (2) build a coefficient equation system; (3) solve the mixed-binary equation system. If the solver does not find an approximate solution, we repeat this process until a timeout condition is met.

Site layout proposer. The first step of the synthesizer loop proposes a site layout L and later steps check its feasibility. To the best of our knowledge, although layout synthesis for quantum circuits is thoroughly studied [Tan and Cong 2020], there is no prior work on the layout synthesis for Hamiltonian-oriented quantum computing. The main difference between them is the unavailability of swap gates for many analog quantum devices, i.e., QuEra's Rydberg atom arrays.

We employ a search with pruning as a general solution to a layout proposer. The pruning strategy is to abort the search when there exists a product Hamiltonian P and j where $H_{\text{tar},j}[P] \neq 0$ and $\mathcal{L}(P)$ does not have a non-zero coefficient expression in any η_k and H_{sys}. This abort condition can be met halfway through the search. For a partial layout L (where several sites are not assigned in L yet) and a product Hamiltonian P, we can map it to a product Hamiltonian $\mathcal{L}(P)$ of the device with holes on several sites. When searching for $\mathcal{L}(P)$ in an AAIS, holes can match any site operator. If none is found, the current search branch is aborted.

After proposing a layout, we proceed to steps (2) and (3) to check its feasibility. If rejected, the above search process returns and proceeds to other search branches to propose another layout. If all possibilities are not feasible, the compiler will report no solution and fail the process.

Coefficient equation builder. We synthesize instruction executions by a system of mixed-binary non-linear equations to match coefficients of product Hamiltonian in the target quantum system.

Given a site layout L, a set of equations is constructed to match the coefficients in $H_{\text{tar},j}$ for every $1 \le j \le N$. We create time variables t_j to represent the evolution time for instruction executions synthesizing evolution of $H_{\text{tar},j}$ for time τ_j, with constraints $t_j > 0$. For instruction η_k in the AAIS, we create an indicator variable $s_{k,j} \in \{0,1\}$ to indicate whether η_k is selected to be executed in the synthesis of $H_{\text{tar},j}$. Assuming that η_k has local variables $\vec{v}_k$ of dimension $|\vec{v}_k|$, we create $|\vec{v}_k|$ new equation system variables stored in a vector $\vec{a}_{k,j}$. For global variables, we create a vector $\vec{g}$ of dimension $|\vec{v}_{\text{glob}}|$ of the AAIS, which is independent of j. In total, we have created $|\vec{v}_{\text{glob}}| + N\sum_k |\vec{v}_k| + N$ real variables for global variables, local variables, and time variables respectively, and NM indicator variables.

Then we establish a coefficient equation for each product Hamiltonian P to match $H_{\text{tar},j}$:

$$(\forall j), (\forall P \neq I): \quad t_j \cdot H_{\text{sys}}(\vec{g})[P] + \sum_{k=1}^{M} t_j \cdot \{\!|\eta_k|\!\}[P](\vec{v}_{k,j}) \cdot s_{k,j} = \tau_j \cdot \mathcal{L}(H_{\text{tar},j})[P]. \tag{4.1.3}$$

Here the left-hand-side calculates the summed effects of P from each instruction and the system Hamiltonian and the right-hand-side calculates the effect of P in the target quantum system.

There are typically many trivial equations in this system having 0 on both sides since only a few P appear in either $H_{\text{tar},j}$ or $\{\!|\eta_k|\!\}$ with respect to the exponentially many possible combinations of site operators. We propose Algorithm 1 to find all non-trivial equations. This algorithm starts with a list Q containing all the product Hamiltonians with non-zero coefficients in H_{sys} and $\mathcal{L}(H_{\text{tar},j})$. It then enumerates the list Q and establishes coefficient equations for each P by enumerating instructions η_k in AAIS. During this process, it may encounter instruction Hamiltonians $\{\!|\eta_k|\!\}$ who contain product Hamiltonians P' that never appears in Q. These product Hamiltonians may lead to non-trivial equations, so we add them to Q. An example of this procedure is illustrated in Figure 9.

Mixed equation solver. The established coefficient equation system is mixed-binary and non-linear. A solver is applied to obtain approximate solutions which correspond to instruction schedules.

We provide several options for the solver. The first is dReal [Gao et al. 2013] based on δ-complete decision procedures, which supports real variables, binary variables, and algebraic functions in HML and AAIS-SL. It performs well when the coefficient expressions are close to linear (the Heisenberg AAIS), while poorly when highly non-linear (the Rydberg AAIS).

As another option, we construct a least-squares-based solver. This solver uses a relaxation-rounding scheme. We apply a continuous relaxation to loosen the value range of indicator variables from $s_{k,j} \in \{0,1\}$ to $\hat{s}_{k,j} \in [0,1]$, substitute $\hat{s}_{k,j}$ for $s_{k,j}$ in the equation system, and solve the equation system by least-squares methods via an implementation in SciPy [Virtanen et al. 2020]. We then round the indicator variables $s_{k,j}$ according to the solution. The criterion sets $s_{k,j}$ to 1 if there is $\sum_P t_j \left\{\!\left|\eta_{k,j}\right|\!\right\}[P](\vec{v}_{k,j})\hat{s}_{k,j} > \delta$ for a pre-defined tolerance parameter δ, and sets to 0 otherwise. This criterion evaluates how much error the solution will induce if we set $s_{k,j}$ to 0. We then solve the equation system again to obtain a more precise solution.

The solver generates an approximate solution with error e, defined by

$$e = \sum\nolimits_{k,j,P} |\tau_j \{\!|\eta_k|\!\}[P](\vec{v}_{k,j}) - t_j \mathcal{L}(H_{\text{tar},j})[P]|. \tag{4.1.4}$$

If $e < \epsilon$ where ϵ is a pre-defined tolerance, the solution is accepted. Otherwise, we return to step (1) to generate another layout and check feasibility. An accepted solution induces an instruction schedule $(\mathcal{S} = \{(C_j, t_j)\}, \vec{g})$ where $C_j = \{(\eta_k, \vec{a}_{k,j}) : s_{k,j} = 1\}$.

4.1.4 Error Induced by Hamiltonian Synthesizer. Now we bound the error in the evolution induced by the approximation in the equation solving of the Hamiltonian synthesizer since our solver

generates approximate numerical solutions. Let $\hat{U}(T)$ be the unitary of executing generated instruction schedule $(\mathcal{S}, \vec{g})$, and $\tilde{U}'(T) = \mathcal{L}(\tilde{U}(T))$ be the unitary of the evolution of the discretized target system after site layout mapping $\mathcal{L}$. We can conclude the error induced by the Hamiltonian synthesizer is bounded by tolerance ϵ in the equation solving and the proof is routine and omitted.

Lemma 4.1 ([Nielsen and Chuang 2002]). *The error of evolution induced by equation solving is bounded by a constant $C_2 > 0$ and error bound ϵ with the following inequality:*

$$\left\|\tilde{U}'(T) - \hat{U}(T)\right\| \leq C_2\epsilon. \tag{4.1.5}$$

Remark 4.1. *In general, compiling a target system is computationally hard. Finding a site layout for machines with specific topology can be as hard as the sub-graph isomorphism problem, an NP-complete problem. Besides, since the design of AAIS does not pose strict restrictions on the expressions, pathological functions may emerge in the coefficients, which complicates the equation-solving process. Our solutions to these problems may not be optimal but are intuitive, feasible, and efficient enough for most cases (also refer to Section 5 for detailed case studies).*

4.2 Block Schedules and Conflict Resolver

Instruction schedules are oversimplified descriptions of what can be executed on the devices. Mainly, there are two realistic restrictions not captured by instruction schedules. First, some instructions on real devices can not be executed simultaneously. For example, on an IonQ device, $\eta_{1,2,XX}$ cannot be simultaneously executed with $\eta_{1,2,ZZ}$ since they use the same interaction process with different bases. Second, instruction execution implementations may take longer than the scheduled execution time. We propose a flexible generalization to the instruction schedules called block schedules and implement a conflict resolver to compile generated instruction schedules to block schedules.

4.2.1 Block Schedules. A block schedule is a temporal graph whose vertices are blocks of instruction executions, together with the valuation of the global variables. An instruction block B contains a collection of instruction executions whose evolution duration is τ. The block schedule is then a directed acyclic graph where an edge $(B_j \longrightarrow B_k)$ is a restriction: instructions in B_k should start simultaneously after instruction executions in B_j end. Instruction schedules generated by our Hamiltonian synthesizer are special cases of block schedules where the temporal graph forms a chain and blocks are the collections of instruction executions.

When executing a block schedule, we first decide the execution order $\gamma : (B_1, ..., B_r)$ of the blocks and then evolve the system by γ sequentially. Let the B_j contain $\{(\eta_{j,k}, \vec{a}_{j,k})\}_k$ with evolution time τ_j. The evolution will generate a unitary transformation

$$U_\gamma = \prod_{j=r}^{1} e^{-i\tau_j (H_{\text{sys}}(\vec{g}) + \sum_k \{|\eta_{j,k}|\}(\vec{a}_{j,k}))}. \tag{4.2.1}$$

Our next step is to generate a block schedule where instructions in each block are simultaneously executable and approximate the execution of the instruction schedule.

4.2.2 Instruction Decorations. In general, the conflict relation of instructions forms a graph F: $(\eta_j, \eta_k) \in F$ means that η_j and η_k cannot be executed simultaneously. To ease the description of F, we introduce decorations to instructions to specify properties like categories of instructions. More decorations can be added based on the detailed hardware restrictions accordingly.

Signal Lines. Physical pulses are sent to devices through signal carriers like electronic wires or arbitrary waveform generators (AWG). A natural conflict is that if two instructions require the same signal carrier, they cannot be executed simultaneously. We abstract the concept of signal carriers as *signal lines* and assign each instruction η to a signal line denoted by $SL(\eta)$. If $SL(\eta_j) = SL(\eta_k)$, instructions η_j, η_k conflict with each other.

Nativeness. Another aspect leading to conflicts is whether instruction implementations employ compound pulses to approximate an effective Hamiltonian. For example, IBM devices generate $\{\!\{\eta_{j,k,CR}\}\!\}$ by direct microwave controls of a cross-resonance pulse [Malekakhlagh et al. 2020]. Hence the IBM-native AAIS for IBM devices has $\eta_{j,k,CR}$ as *native* instructions: they can be simultaneously executed with other native instructions. To effectively realize $\{\!\{\eta_{j,k,ZZ}\}\!\}$, a compound sequence of microwaves including two cross-resonance pulses is applied to approximate a Z_jZ_k interaction [Alexander et al. 2020]. Simultaneously applying other pulses on site j or k will break the approximation. Hence $\eta_{j,k,ZZ}$ are *derived* instructions in the IBM-native AAIS.

Let $\mathsf{inf}(H)$ be the sites on which Hamiltonian H acts non-trivially (when limited on these sites, H is not identity). We assume that implementing a derived instruction η only affects $\mathsf{inf}(\{\!\{\eta\}\!\})$. Then a derived instruction η_1 conflicts with η_2 if $\mathsf{inf}(\{\!\{\eta_1\}\!\}) \cap \mathsf{inf}(\{\!\{\eta_2\}\!\})$ is not empty.

4.2.3 Conflict Resolver via Trotterization. Given a conflict graph and an instruction schedule $(S, \vec{g})$, we implement a conflict resolver to generate a block schedule without conflicts in each block.

A well-studied technique in quantum information to simulate summed Hamiltonians in quantum simulation is Trotterization. Let Hamiltonian $H = \sum_{j=1}^{L} H_j$ where L Hamiltonians evolve the system simultaneously. We assume we have a device supporting evolving single H_j for any duration t, realizing unitary matrix e^{-itH_j}, while there is no evolution under $\sum_{j=1}^{L} H_j$. Trotterization (also known as the product formula algorithm) [Lloyd 1996] makes use of the Lie-Trotter formula

$$e^{-it\sum_j H_j} = \lim_{n\to\infty} \left(\prod\nolimits_j e^{-i\frac{t}{n}H_j}\right)^n \approx \left(\prod\nolimits_j e^{-i\frac{t}{N}H_j}\right)^N. \tag{4.2.2}$$

By choosing a large N, the above formula shows that we can approximate the evolution under H for time T by repeating for N times a sequential evolution for $j \in \{1, ..., L\}$ under H_j for time t/N. Each segment of evolution realizes a unitary transformation $e^{-i(t/N)H_j}$ as in the formula.

First, we consider the case where $H_{\text{sys}} = 0$. Each (C_d, τ_d) in $\mathcal{S}$ is considered independently. Let $C_d = \{(\eta_j, \vec{a}_j)\}_j$ and the conflict graph of these instructions be F. To accommodate Trotterization in a conflict resolver, we first categorize the instructions into groups without conflict. The grouping is effectively a coloring of vertices in F where no edge connects monochromatic vertices. We employ a greedy graph coloring algorithm from NetworkX [Hagberg et al. 2008] to find a feasible grouping $\{G_j\}_{j=1}^{L}$ with L colors where G_j contains instruction executions in the j-th group.

A temporal graph in a block schedule can depict the process in (4.2.2). Let H_j be the Hamiltonian of simultaneous instruction executions in G_j, $H_j = \sum_{(\{\!|\eta_k|\!\},\vec{a}_k)\in G_j} \{\!\{\eta_k\}\!\}(\vec{a}_k)$, and R be the Trotterization number specified by users. An evolution of H_j for time τ_d/R corresponds to a block $B_j = (G_j, \tau_d/R)$. Then a sequential evolution of $\{H_j\}_{j=1}^{L}$ forms a chain $B_1 \to \cdots \to B_L$. We create R copies of this chain and connect them sequentially to represent the Trotterization process.

Additionally, we deal with the cases where the system Hamiltonian H_{sys} is non-zero. Let $\tilde{L}$ be the maximal coloring number L in the above process. We assume that there exists $\vec{g}_{\tilde{L}}$ such that $H_{\text{sys}}(\vec{g}_{\tilde{L}}) = H_{\text{sys}}(\vec{g})/\tilde{L}$. Some devices may not support this assumption, but it is rarely used since only a few devices with non-zero system Hamiltonian have conflicting instructions. We then augment the number of groups to $\tilde{L}$ for each $(C_j, \tau_j) \in \mathcal{S}$ by adding empty sets in groupings. Now we create a block schedule with $\vec{g}_{\tilde{L}}$ and a temporal graph constructed on the augmented groupings. Executing this block schedule approximates the execution of the given instruction schedule.

4.2.4 Error Induced by Conflict Resolver. The Trotterization resolves conflicts while also introducing errors. We denote the instruction schedule where $S = \{(\{(\eta_{d,j}, \vec{a}_{d,j})\}_j, \tau_d)\}_{d=1}^{D}$ and its evolution as $\hat{U}(T)$. For segment d of evolution in S, we assume the grouping is $\{G_j^d\}_{j=1}^{L_d}$ and the evolution by executing the block schedule as $\bar{U}(T)$.

Lemma 4.2 ([Childs et al. 2018]). *The difference between $\hat{U}(T)$ and $\bar{U}(T)$ of evolution after resolving conflicts by Trotterization is bounded by*

$$\left\|\hat{U}(T) - \bar{U}(T)\right\| \leq \frac{(\Lambda T)^2}{DR} e^{\frac{\Lambda T}{DR}}. \tag{4.2.3}$$

Here $\Lambda = \max_{d,j} L_d \left\| \sum_{(\eta,\vec{a})\in G_j^d} \{|\eta|\}(\vec{a}) \right\|$, D and R are the discretization and Trotterization numbers.

As implied by this lemma, in ideal cases, increasing the Trotterization number reduces the induced error to arbitrarily small. However, it also increases the total number of instruction executions. Due to the non-negligible error accumulations in each instruction execution on devices, there is a trade-off over the Trotterization number R depending on the real-time parameters of the device, where we leave the freedom to user specification.

Optimization techniques for Trotterization are also well-developed in theory [Childs et al. 2021], and we discuss their implementation in the extended version [Peng et al. 2023b] Appendix.

4.3 Signal Line Schedules and Scheduler

The eventual output of SimuQ contains the pulses sent through signal carriers for devices to execute. We propose another intermediate representation, called a *signal line schedule*, to depict the concrete instruction executions sent through each signal line abstracted in Section 4.2.2 before generating platform-dependent executable pulses. For signal line l, it contains a list of instruction executions $(\eta, \vec{a}, \tau_s, \tau_e)$ with absolute starting and ending times and satisfying $SL(\eta) = l$.

To obtain a signal line schedule, we build a scheduler to traverse the temporal graph of the block schedule via a topological sort and generate a valid execution order of block schedules. It employs a first-arrive-first-serve principle for each signal line. The scheduler first extracts information about how long implementing each instruction execution takes from real devices. Next, it arranges instruction executions on the signal lines at the earliest possible starting time obeying the order.

Remark 4.2. *The scheduling process may be independently configured and optimized, and the scheduler may use other criteria to determine the traversal order of instruction blocks or the alignment of blocks within the scheduled order as long as the hardware permits. This freedom in the scheduling process may be leveraged to reduce cross-talk [Murali et al. 2020] between the blocks or save small implementation overheads. We illustrate only a basic strategy and leave the exploitation for the future.*

4.4 Pulse Schedules and Pulse Translator

In its final stage, the SimuQ compiler translates a signal line schedule into a pulse schedule using hardware providers' domain languages and APIs.

We extract the pulse shapes from the devices for each platform to implement instruction executions. We substitute the instruction execution on each signal line for pulse shapes configured by the valuations of local variables via the format specified by a pulse-enabled quantum device provider.

4.4.1 Translation to Hardware APIs. There are few pulse-enabled quantum device providers, and programming pulses is a challenging endeavor that requires extensive platform knowledge of various hardware and software engineering considerations. We demonstrate the effectiveness of SimuQ using QuEra, IBM, and IonQ devices.

QuEra's Rydberg atom devices. Two APIs to QuEra devices are supported by SimuQ for the global Rydberg AAIS: Bloqade [QuEra 2022] programs and Amazon Braket programs. We set the atom positions according to the valuation of global variables and laser configurations as piecewise constant functions according to the valuations of local variables. Since the detuning Δ and amplitude Ω generate linear effects, piecewise linear laser configurations are also supported as an option. For

Amazon Braket programs, $\Omega(t)$ should start and end at amplitude 0, so we add short (0.1ms) time intervals to the pulses' beginning and end with linear ramping. We also scale the pulse schedules to a total length of around 3.5ms to fit in the 4ms duration limit of the device.

IBM's superconducting devices. For IBM devices, SimuQ can generate Qiskit Pulse programs for the Heisenberg AAIS and the IBM-native AAIS. For single-site instructions, the IBM device supports implementations of native X and Y instructions and derived Z instructions. We build up DRAG pulses [Motzoi et al. 2009] to realize X and Y instructions and free Z rotations [McKay et al. 2017] to realize Z instructions, which are standard superconducting device techniques. Two-qubit instructions in the Heisenberg AAIS are realized through the Z_jX_k interactions created by echoed cross resonance pulses [Malekakhlagh et al. 2020] together with single-qubit evolution to change bases. We follow Earnest et al. [2021] and realize interaction-based gate implementations, whose benefits are further explained in Section 5.3. Additionally, we extract cross-resonance pulses from Qiskit and compose pulses to realize native $\eta_{j,k,CR}$ in the IBM-native AAIS.

IonQ's trapped-ion devices. SimuQ supports both IonQ cloud and Qiskit circuit programs for IonQ devices with the Heisenberg AAIS. Unlike QuEra and IBM devices, IonQ does not provide pulse-level programmability for their ion trap devices. However, we can still exploit their native gate set to generate a quantum circuit with precise control of the execution on their devices. With the support of partially entangling Mølmer-Sørenson gate [Sørensen and Mølmer 2000], we can implement instructions of the Heisenberg AAIS with higher fidelity. More details are explained in our case studies in Section 5.3.

4.4.2 Semantics of Pulse Schedules and Errors in Instruction Implementation. Abstractly, a pulse schedule includes a time-dependent function $\vec{f_l}(t)$ (pulses) for signal line l, generating the effective Hamiltonian $H_l(t)$ physically. For example, instruction execution $(\eta, \vec{a}, \tau_s, \tau_e)$ for signal line l in the signal line schedule should be translated into pulses $\vec{f_l}(t)$ that effectively generate $H_l(t) = \{|\eta|\}(\vec{a})$ for $\tau_s \le t < \tau_e$. Collectively, the Hamiltonian on the device is $H_{\text{dev}}(t) = H_{\text{sys}} + \sum_l H_l(t)$, and the semantics of executing a pulse schedule is the unitary evolution under $H_{\text{dev}}(t)$. Yet, the implementation of instructions on real devices may be imperfect. We assume that there is a implementation error threshold Δ such that the on-device $\check{H}_l(t)$ and $\check{H}_{\text{sys}}$ satisfies $\max_{t,l} \left\|\check{H}_l(t) - H_l(t)\right\| \le \Delta$ and $\left\|\check{H}_{\text{sys}} - H_{\text{sys}}\right\| \le \Delta$, forming on-device evolution under $\check{H}_{\text{dev}}(t) = \check{H}_{\text{sys}} + \sum_l \check{H}_l(t)$. Since the signal line scheduler does not alter the semantics of block schedules, we bound the implementation error on the device.

Lemma 4.3 ([Nielsen and Chuang 2002]). *The difference between the unitary $\check{U}(T)$ on the device and the unitary $\bar{U}(T)$ of executing the block schedule generated by the conflict resolver is bounded by*

$$\left\|\bar{U}(T) - \check{U}(T)\right\| \le C_3 S \Delta \Gamma T, \tag{4.4.1}$$

where $C_3 > 0$ is a constant, S is the number of signal lines and system Hamiltonians, and $\Gamma = \max_d L_d$ is the maximal number of groups in the conflict resolver.

With a faithful implementation of instructions on real devices, the pulse translator produces negligible errors. The proof is routine in quantum information and is therefore omitted. We remark that other forms of device errors (e.g., high-energy space leakage) can be analyzed similarly.

4.5 Semantics Preservation of SimuQ Compiler

If compilation succeeds, the SimuQ compiler generates executable pulse schedules from programmed quantum systems with bounded errors. We conclude the approximate semantics preservation theorem of the SimuQ compilation process using Lemma 3.1, Lemma 4.1, Lemma 4.2, and Lemma 4.3.

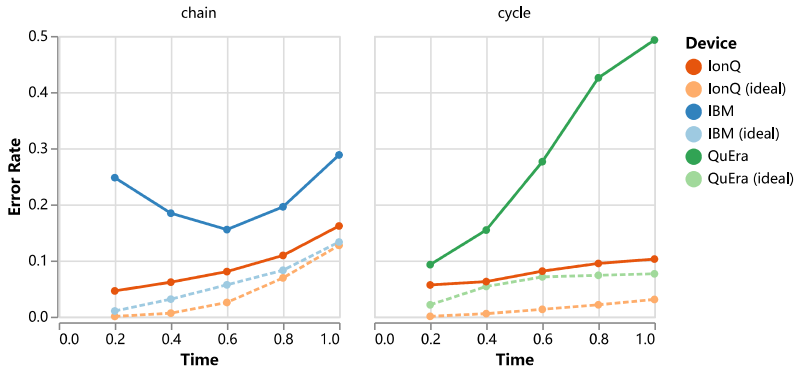


Fig. 11. The simulation errors of the 6-site Ising models on multiple platforms. Ideal results are obtained by compiling with SimuQ and executing on noiseless simulators.

Theorem 4.4 (Semantics Preservation). *Given a Hamiltonian* $H_{\text{tar}}(t) = \sum_{k=1}^{K} \alpha_k(t) H_k$ *where* α_k *is piecewise M-Lipschitz and* $\|H_k\| = 1$*, if the compilation succeeds, SimuQ generates a site layout* L *and an executable pulse schedule. Let the unitary* $U(T)$ *represent the evolution under* $H_{\text{tar}}(t)$ *for duration* $[0, T]$ *and* $\check{U}(T)$ *for the evolution executing the pulse schedule on the device. We have*

$$\left\| \mathcal{L}(U(T)) - \check{U}(T) \right\| \leq C_1 D^{-1} MKT^2 + C_2 \epsilon + (\Lambda T)^2 D^{-1} R^{-1} e^{\frac{\Lambda T}{DR}} + C_3 S \Delta \Gamma T. \tag{4.5.1}$$

Here, T *is the evolution time,* $\mathcal{L}$ *is the site layout mapping of layout* L*,* C_1, C_2, C_3 *are constants,* D *is the discretization number,* ϵ *is the error threshold in Hamiltonian synthesizer,* R *is the Trotterization number,* Δ *is the instruction implementation error threshold,* S *is the number of signal lines and system Hamiltonians on the device, and* Λ *and* Γ *depend on the Trotterization strategy in the compilation.*

Tuning D, ϵ, R and improving the implementation to decrease δ can reduce errors induced by the SimuQ compiler to arbitrarily small. The evolution under H_{tar} is hence simulated on the device.

5 CASE STUDIES

We conduct several case studies highlighting SimuQ's portability and the advantages of Hamiltonian-oriented compilation, including native instructions and interaction-based gates. We also establish a small benchmark of quantum simulation to evaluate the SimuQ compiler performance.

5.1 Multiple-Platform Compatability

We compile and execute the Ising model on multiple supported devices of SimuQ. The following experiments show the portability of SimuQ on heterogeneous analog quantum simulators. We only need to program the target quantum systems once and apply the SimuQ compiler to generate code for different platforms and deploy and execute them on multiple real devices.

We focus on the simulation of the Ising model introduced in Section 2.2. We demonstrate two instances: a 6-site cycle and a 6-site chain, mathematically depicted by

$$H_{\text{chain}} = \sum_{j=1}^{5} Z_j Z_{j+1} + \sum_{j=1}^{6} X_j, \quad H_{\text{cycle}} = H_{\text{chain}} + Z_1 Z_6. \tag{5.1.1}$$

The target quantum system is to simulate H_{cycle} and H_{chain} for $T = 1$. When Trotterization is utilized, we set the Trotterization number to be 4, which is empirically selected based on experiment results.

SimuQ successfully compiles H_{cycle} on QuEra devices using the global Rydberg AAIS, both H_{cycle} and H_{chain} on IonQ devices using the Heisenberg AAIS, and H_{chain} on IBM devices using the Heisenberg AAIS. We send the generated code to execute on corresponding devices. Since QuEra devices do not support state tomography, we evaluate the results on these platforms by a metric

based on measurements supported by all devices in our experiments. We obtain the frequency of reading a bit-string s in a measurement instantly after the simulation finishes as a distribution $\mathbb{P}_{\text{exp}}[s]$ and numerically calculate the ground truth distribution $\mathbb{P}_{\text{GT}}[s]$ of obtaining s. We utilize the total variation distance $TV(\mathbb{P}_{\text{exp}}, \mathbb{P}_{\text{GT}}) = \frac{1}{2}\sum_{s\in\{0,1\}^6}|\mathbb{P}_{\text{exp}}[s] - \mathbb{P}_{\text{GT}}[s]|$ to evaluates the errors. We present the classical simulation of devices and real device execution results in Figure 11.

The minor errors in the classical simulations indicate the correctness of our framework. In ideal cases, the errors are induced by uncancellable non-neighboring Z_jZ_k interactions and short ramping times for the global Rydberg AAIS and by Trotterization errors for the Heisenberg AAIS. The real device execution results show higher errors than classical simulation because of device noises, while they are valid quantum simulation results. The errors on the IBM device are non-monotone, likely because the large state preparation and measurement errors affect more heavily the cases where the states deviate only a little from the initial state.

SimuQ fails to compile H_{chain} on QuEra devices since it requires different local detuning parameters for different sites, which current QuEra devices and the global Rydberg AAIS do not support. It also fails to compile H_{cycle} on IBM devices since there is no 6-vertex cycle in IBM devices.

5.2 Hamiltonian-Oriented Compilation with Native Instructions

The most significant benefit of enabling Hamiltonian-level programming is to gain fine-grained and multi-site control via native operations. Near-term quantum devices have short coherence times: quantum states will deteriorate and lose their quantumness quickly. Generating shorter pulses to achieve the same effects is one of the crucial tasks for compilers of modern quantum devices. In this case study, we showcase the advantage in the lengths of pulse schedules enabled by Hamiltonian-oriented compilation using the IBM-native AAIS.

Our target quantum system evolves under $H_{2ZX} = Z_1X_2 + X_2Z_3$ for time $T = 1$, a small 3-site system. The IBM-native AAIS contains two native instructions $\eta_{1,2,CR}$ and $\eta_{3,2,CR}$ with Z_1X_2 and X_2Z_3 interactions respectively. Following Greenaway et al. [2022], the simultaneous execution of them can be realized by simultaneously applying two cross-resonance pulses on IBM devices. By automatically compensating the other terms in SimuQ with native instructions $\eta_{2,X}$ and derived instructions $\eta_{1,Z}$ and $\eta_{2,Z}$ (their effects commute with $\{\!|\eta_{1,2,CR}|\!\}$ and $\eta_{3,2,CR}$ so no Trotterization is needed), the uncancellable remains are Z_1Z_2 interactions and Z_2Z_3 interactions. Fortunately, they can be reduced to one magnitude smaller than Z_1X_2 and X_2Z_3 interactions when selecting a relatively large Ω, and are considered small errors in the compilation. The pulse schedule to realize H_{2ZX}, displayed in Figure 12, is around 280ns long.

H_{2ZX} can also be compiled on IBM devices by a circuit-based compilation with the help of Qiskit. It first decomposes the simulation into a circuit with two gates $R_{Z_1X_2}(2)R_{X_2Z_3}(2)$ where $R_{Z_jX_k}(\theta) = e^{-i(\theta/2)Z_jX_k}$. It then invokes Qiskit's transpiler to decompose each $R_{Z_jX_k}(\theta)$ into two CNOT gates and several single qubit gates and generates a Qiskit pulse schedule, which is displayed in Figure 13 and is around 1660ns long. This is around six times longer than the pulse schedule generated by SimuQ using the IBM-native AAIS.

5.3 Hamiltonian-Oriented Compilation with Interaction-based Gates

For some devices that lack the support of simultaneous instruction executions by native operations, we can still exploit the capability of realizing gates based on evolving interaction for various time periods. By interaction-based gates, we refer to quantum gates of form $R_H(t) = e^{-itH}$, where the time duration of the pulse shapes implementing them is strongly correlated with t. These gates are common on platforms supporting universal gates like IBM devices and IonQ devices but are not exploited in their provided compiler due to the hardness in calibration. Under the conventional

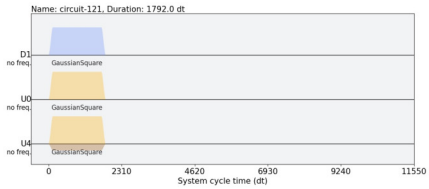


Fig. 12. Pulses generated by SimuQ for evolution under H_{2ZX} for duration 1.

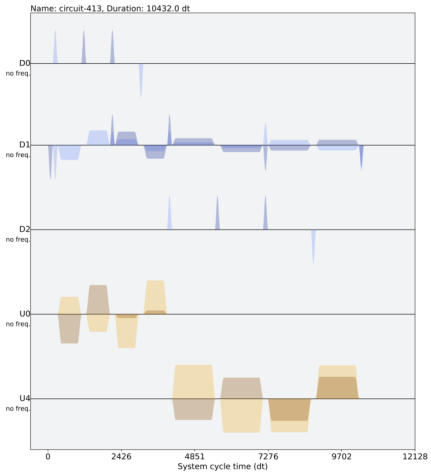


Fig. 13. Pulses generated by Qiskit compiler for evolution under H_{2ZX} for duration 1.

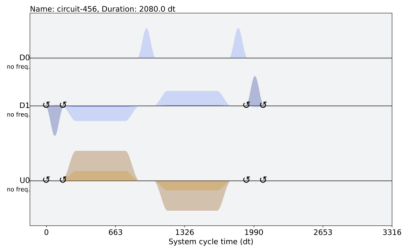


Fig. 14. Pulses generated by SimuQ for evolving Z_0Z_1 for $T = 1$.

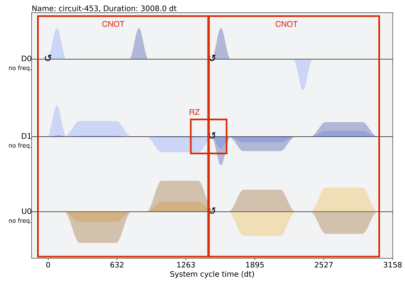


Fig. 15. Pulses generated by Qiskit for evolving Z_0Z_1 for $T = 1$.

Table 2. The differences between the ideal $C(s)$ and the measured $C(s)$ on devices are displayed for different layers p with SimuQ and Qiskit CNOT-based compiler and are better when lower.

p	IBM		IonQ	
	SimuQ	Qiskit	SimuQ	Qiskit
1	**0.724**	0.855	0.240	**0.114**
2	**1.365**	1.929	**0.453**	0.474
3	**2.082**	3.166	**0.518**	0.715

circuit-oriented compilation where quantum programs are compiled to a gate set with fixed number 2-qubit gates (typically, only CNOT gates), interaction-based gates are decomposed using multiple 2-qubit gates for the convenience of calibration, like $R_{Z_jX_k}(\theta)$ gates that are decomposed using 2 CNOT gates by Qiskit and translated to a pulse schedule of long and fixed duration. Although interaction-based gates cannot be simultaneously applied on devices when overlapping sites exist, exploiting them can still significantly reduce the duration of generated pulse schedules and increase the fidelity of simulations as observed in [Earnest et al. 2021; Stenger et al. 2021].

In this section, we implement the quantum approximate optimization algorithm (QAOA) in SimuQ and execute it on IBM and IonQ devices. The QAOA algorithm is a classical-quantum Hamiltonian-oriented algorithm designed to solve combinatorial problems. We omit the algorithm analysis and refer interested readers to [Farhi et al. 2014]. We consider the quantum simulation part of a typical case of the QAOA algorithm, where the target quantum system evolves under a length-p sequence of alternative evolution between H_1 and H_2 where

$$H_1 = Z_1Z_N + \sum_{j=1}^{N-1} Z_jZ_{j+1}, \qquad H_2 = \sum_{j=1}^{N} X_j. \tag{5.3.1}$$

Here $N = 12$ is the problem size. Two pre-defined parameter lists $\{\theta_j\}_{j=1}^{p}$ and $\{\gamma_j\}_{j=1}^{p}$ of length p describe the time of each evolution segment. I.e., the j-th segment first lets the system evolve under H_1 for time θ_j and then lets the system evolve under H_2 for time γ_j. Ultimately, we measure the

sites and store the results in a bit-string s. The more precisely we simulate the system, the larger the evaluation function $C(s) = |\{j : 1 \le j \le N, s_j \neq s_{j+1}\}|$ will be (assuming $s_{N+1} = s_1$).

When compiling the target system for the Heisenberg AAIS, $\eta_{j,j+1,ZZ}$ are executed frequently. We take an execution $\eta_{1,2,ZZ}$ for $t = 1$ as an example, which effectively realizes the gate $e^{-iZ_1Z_2}$. Qiskit decomposes it with 2 CNOT gates and single qubit rotation gates, and the generated pulse duration takes a constant 662ns independent of t, as illustrated in Figure 15. An alternative solution is to create Z_1X_2 interaction constructed by echoed cross-resonance pulses for a duration positively correlated to t with short pulses implementing Hadamard gates to effectively realize the Z_1Z_2 interaction, as illustrated in Figure 14. The pulse schedule is around $(330t + 130)$ns long to execute $\left\{\left|\eta_{j,j_1,ZZ}\right|\right\}(1)$ for time t. When $t = 1$, it is 462ns long, 30% shorter than the Qiskit compiler. Interaction-based gates are especially beneficial when a program requires many short instruction executions, like compiled simulations with a large Trotterization number.

We then compile and execute the QAOA system simulation to IBM and IonQ devices for cases $p = 1, 2, 3$. Similarly, we reproduce this problem in Qiskit and compile it with the Qiskit compiler (CNOT-based decomposition is applied). On IBM devices, for $p = 3$, the pulse schedule generated by SimuQ is 3.35μs long. In contrast, the one generated by Qiskit is 7.48μs long, which is more than two times longer[2]. On average, SimuQ generates pulse schedules 59% percent shorter than Qiskit. We then execute the generated programs on IBM devices and IonQ devices. The differences of the evaluation function $C(s)$ measured on devices and ground truth values are present in Table 2. We observe that, on average, the pulse schedules generated by SimuQ reduce errors (the difference to ideal results) by 34% on IBM devices and 28% on IonQ devices for $p = 3$ compared to pulse schedules from Qiskit. The advantage is less significant for shallow cases where $p = 1, 2$ because of state preparation and measurement errors on real devices [Tannu and Qureshi 2019]. These experiments demonstrate the advantage and the necessity of Hamiltonian-oriented compilation using interaction-based gates on devices not supporting simultaneous instruction executions.

5.4 Benchmarking Quantum Simulation Compilation

To illustrate SimuQ's capability of dealing with various quantum simulation problems, we craft a small benchmark containing models collected from multiple domains like condensed matter physics, high-energy physics, particle physics, and optimization. The diversity of the cases in this benchmark of different topologies, time dependency, and system sizes exhibits our compiler's feasibility and efficiency in dealing with significant simulation problems.

We present the benchmark in Table 3, and further illustrations of its quantum systems are in the extended version [Peng et al. 2023b] Appendix. We report each quantum system's number of sites and lines of code to implement them with HML in SimuQ. Most systems can be programmed within 20 lines, showing the user-friendliness of programming quantum systems in SimuQ.

For each target quantum system, we compile it on the platforms supported by SimuQ using their most capable devices in the possible future. The compilation time is averaged over 5 runs on a laptop with Intel Core i7-8705G CPU. SimuQ compiler reports no solution (No sol.) in several cases due to complicated interactions beyond the hardware capability of QuEra devices and the limited connectivity of IBM devices. Limited connectivity on large IBM devices also complicates the site layout search, making a case exceed a pre-set compilation time limit of 3600 seconds, which is marked as a time out.

Pulse schedule duration for IBM devices using SimuQ and Qiskit to compile is reported. On average, Qiskit's default compilation passes generate 29.3 times longer pulse schedules than the

[2]IonQ devices do not support reporting pulse schedule duration.

Table 3. A benchmark of quantum simulation problems. We program and compile the models in SimuQ to obtain pulse schedules for QuEra and IBM devices and quantum circuits for IonQ devices. We record the compilation time (comp. time), the pulse duration (P.D.), and the 2-qubit gate count for the generated circuits. No. sol. represents cases where the SimuQ compiler reports no solution because of hardware constraints, such as limited interactions of QuEra devices and machine topology for IBM devices. Time out is reported when the compilation takes more than an hour, which happens in the search for a 64-qubit cycle on IBM devices.

System name	LoC	# of sites	QuEra	IBM			IonQ	
			Comp. time (s)	Comp. time (s)	P.D. (μs) SimuQ	P.D. (μs) Qiskit	Comp. time (s)	# of 2q-gate
ising_chain	13	6	0.177	0.224	2.06	8.69	0.155	20
		32	39.3	54.6	3.24	39.2	47.2	124
		64	663	257	3.15	81.2	680	252
		96	2298	1086	3.26	450	3568	380
ising_cycle	13	6	0.585	No. sol.			0.13	24
		12	3.47	1.49	2.05	37.8	1.37	48
		32	114	483	3.35	144	53.8	128
		64	3454	Time out			907	256
heis_chain	15	32	No. sol.	143	10.1	119	138	372
qaoa_cycle	19	12	No. sol.	0.503	0.83	37.6	1.5	36
qhd	16	16	No. sol.	No. sol.			66.3	480
mis_chain	22	12	5.45	19.1	18.9	94	25.2	440
		24	53.1	328	18.9	162	278	920
mis_grid	29	16	28.4	No. sol.			85.4	960
		25	141	No. sol.			489	1600
kitaev	13	18	4.67	15.6	2.12	21.2	8.74	68
schwinger	18	10	No. sol.	No. sol.			1.09	28
o3nlσm	19	30	No. sol.	No. sol.			77.7	588

SimuQ compiler over cases successfully compiled. We also report the number of partially entangling Mølmer-Sørenson gates when compiling on IonQ's devices to indicate the total duration.

6 CONCLUSION AND FUTURE DIRECTIONS

The domain-specific language SimuQ described in this paper is the first framework to consider quantum simulation and compilation to multiple platforms of analog quantum simulators. We propose HML for front-end users to program their target quantum systems intuitively. We also design abstract analog instruction sets to depict the programmability of analog quantum simulators and the AAIS-SL to program them. Furthermore, the SimuQ compiler is the first compiler to generate pulse schedules of analog quantum simulators for desired quantum simulation.

Since this is the first feasibility demonstration of programming analog quantum simulators, there is much optimization space for our compiler. First, since different devices have different properties crucial to the compiler's efficiency, we can develop compilation passes specifically for each platform. Second, this paper employs a brute-force search with heuristics to find a site layout where more pruning techniques are desired. Third, The hand-crafted mixed-binary equation solver can also be optimized according to the structure of the problem. Furthermore, with a better understanding of hardware, we can design more powerful AAISs. Lastly, we can add more compilation techniques like [Clinton et al. 2021] to synthesize product Haimltonians not appearing directly in the given AAIS with a combination of instruction executions.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their constructive feedback and thank Robert Rand, Kesha Hietala, Jens Palsberg, Frederic Chong, Cedric Lin, Peter Komar, Cody Wang, Jean-Christophe Jaskula, Murphy Niu, Lei Fan, and Yufei Ding for their helpful discussions. Y.P., J.Y., and X.W. were partially funded by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research, Quantum Testbed Pathfinder Program under Award Number DE-SC0019040, Air Force Office of Scientific Research under award number FA9550-21-1-0209, the U.S. National Science Foundation grant CCF-1942837 (CAREER), and a Sloan research fellowship. This research used resources of the Oak Ridge Leadership Computing Facility, which is a DOE Office of Science User Facility supported under Contract DE-AC05-00OR22725.

DATA AVAILABILITY STATEMENT

Our code is available at https://github.com/PicksPeng/SimuQ.
A project website of SimuQ is available at https://pickspeng.github.io/SimuQ/.
An evaluated artifact [Peng et al. 2023a] is available at https://zenodo.org/doi/10.5281/zenodo.8423709.

REFERENCES

Ali J Abhari, Arvin Faruque, Mohammad J Dousti, Lukas Svec, Oana Catu, Amlan Chakrabati, Chen-Fu Chiang, Seth Vanderwilt, John Black, and Fred Chong. 2012. *Scaffold: Quantum programming language.* Technical Report. Princeton Univ NJ Dept of Computer Science.

Sara Achour and Martin Rinard. 2020. Noise-Aware Dynamical System Compilation for Analog Devices with Legno. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) *(ASPLOS '20)*. Association for Computing Machinery, New York, NY, USA, 149–166. https://doi.org/10.1145/3373376.3378449

Sara Achour, Rahul Sarpeshkar, and Martin C. Rinard. 2016. Configuration Synthesis for Programmable Analog Devices with Arco. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Santa Barbara, CA, USA) *(PLDI '16)*. Association for Computing Machinery, New York, NY, USA, 177–193. https://doi.org/10.1145/2908080.2908116

Gadi Aleksandrowicz, Thomas Alexander, Panagiotis Barkoutsos, Luciano Bello, Yael Ben-Haim, David Bucher, F Jose Cabrera-Hernández, Jorge Carballo-Franquis, Adrian Chen, Chun-Fu Chen, et al. 2019. Qiskit: An open-source framework for quantum computing. *Accessed on: Mar* 16 (2019). https://doi.org/10.5281/zenodo.2562110

Thomas Alexander, Naoki Kanazawa, Daniel J Egger, Lauren Capelluto, Christopher J Wood, Ali Javadi-Abhari, and David C McKay. 2020. Qiskit pulse: Programming quantum computers through the cloud with pulses. *Quantum Science and Technology* 5, 4 (2020), 044006. https://doi.org/10.1088/2058-9565/aba404

Assa Auerbach. 1998. *Interacting electrons and quantum magnetism.* Springer Science & Business Media. https://doi.org/10.1007/978-1-4612-0869-3

John Backus. 1978. *The History of Fortran I, II, and III.* Association for Computing Machinery, New York, NY, USA, 25–74. https://doi.org/10.1145/800025.1198345

Lindsay Bassman, Connor Powers, and Wibe A. De Jong. 2022. ArQTiC: A Full-Stack Software Package for Simulating Materials on Quantum Computers. *ACM Transactions on Quantum Computing* 3, 3, Article 17 (jun 2022), 17 pages. https://doi.org/10.1145/3511715

Hannes Bernien, Sylvain Schwartz, Alexander Keesling, Harry Levine, Ahmed Omran, Hannes Pichler, Soonwon Choi, Alexander S Zibrov, Manuel Endres, Markus Greiner, et al. 2017. Probing many-body dynamics on a 51-atom quantum simulator. *Nature* 551, 7682 (2017), 579–584. https://doi.org/10.1038/nature24622

Colin D Bruzewicz, John Chiaverini, Robert McConnell, and Jeremy M Sage. 2019. Trapped-ion quantum computing: Progress and challenges. *Applied Physics Reviews* 6, 2 (2019). https://doi.org/10.1063/1.5088164

Yudong Cao, Jonathan Romero, Jonathan P Olson, Matthias Degroote, Peter D Johnson, Mária Kieferová, Ian D Kivlichan, Tim Menke, Borja Peropadre, Nicolas PD Sawaya, et al. 2019. Quantum chemistry in the age of quantum computing. *Chemical reviews* 119, 19 (2019), 10856–10915. https://doi.org/10.1021/acs.chemrev.8b00803

Bikas K Chakrabarti, Amit Dutta, and Parongama Sen. 2008. *Quantum Ising phases and transitions in transverse Ising models.* Vol. 41. Springer Science & Business Media. https://doi.org/10.1007/978-3-642-33039-1

Andrew M Childs. 2010. On the relationship between continuous-and discrete-time quantum walk. *Communications in Mathematical Physics* 294 (2010), 581–603. https://doi.org/10.1007/s00220-009-0930-1

Andrew M Childs. 2017. Lecture notes on quantum algorithms. *Lecture notes at University of Maryland* (2017).

Andrew M. Childs, Dmitri Maslov, Yunseong Nam, Neil J. Ross, and Yuan Su. 2018. Toward the first quantum simulation with quantum speedup. *Proceedings of the National Academy of Sciences* 115, 38 (2018), 9456–9461. https://doi.org/10.1073/pnas.1801723115

Andrew M Childs, Yuan Su, Minh C Tran, Nathan Wiebe, and Shuchen Zhu. 2021. Theory of trotter error with commutator scaling. *Physical Review X* 11, 1 (2021), 011020. https://doi.org/10.1103/PhysRevX.11.011020

Andrew M Childs and Nathan Wiebe. 2012. Hamiltonian simulation using linear combinations of unitary operations. *arXiv preprint arXiv:1202.5822* (2012). https://doi.org/10.48550/arXiv.1202.5822

Laura Clinton, Johannes Bausch, and Toby Cubitt. 2021. Hamiltonian simulation algorithms for near-term quantum hardware. *Nature communications* 12, 1 (2021), 4989. https://doi.org/10.1038/s41467-021-25196-0

Andrew Cross. 2018. The IBM Q experience and QISKit open-source quantum computing software. In *APS March meeting abstracts*, Vol. 2018. L58–003.

Andrew Cross, Ali Javadi-Abhari, Thomas Alexander, Niel De Beaudrap, Lev S. Bishop, Steven Heidel, Colm A. Ryan, Prasahnt Sivarajah, John Smolin, Jay M. Gambetta, and Blake R. Johnson. 2022. OpenQASM 3: A Broader and Deeper Quantum Assembly Language. *ACM Transactions on Quantum Computing* 3, 3, Article 12 (sep 2022), 50 pages. https://doi.org/10.1145/3505636

Shantanu Debnath, Norbert M Linke, Caroline Figgatt, Kevin A Landsman, Kevin Wright, and Christopher Monroe. 2016. Demonstration of a small programmable quantum computer with atomic qubits. *Nature* 536, 7614 (2016), 63–66. https://doi.org/10.1038/nature18648

Nathan Earnest, Caroline Tornow, and Daniel J Egger. 2021. Pulse-efficient circuit transpilation for quantum applications on cross-resonance-based hardware. *Physical Review Research* 3, 4 (2021), 043088. https://doi.org/10.1103/PhysRevResearch.3.043088

Sepehr Ebadi, Tout T Wang, Harry Levine, Alexander Keesling, Giulia Semeghini, Ahmed Omran, Dolev Bluvstein, Rhine Samajdar, Hannes Pichler, Wen Wei Ho, et al. 2021. Quantum phases of matter on a 256-atom programmable quantum simulator. *Nature* 595, 7866 (2021), 227–232. https://doi.org/10.1038/s41586-021-03582-4

Edward Farhi, Jeffrey Goldstone, and Sam Gutmann. 2014. A quantum approximate optimization algorithm. *arXiv preprint arXiv:1411.4028* (2014). https://doi.org/10.48550/arXiv.1411.4028

Richard P Feynman. 1982. Simulating physics with computers. *International journal of theoretical physics* 21, 6/7 (1982), 467–488. https://doi.org/10.1007/BF02650179

WMC Foulkes, Lubos Mitas, RJ Needs, and Guna Rajagopal. 2001. Quantum Monte Carlo simulations of solids. *Reviews of Modern Physics* 73, 1 (2001), 33. https://doi.org/10.1103/RevModPhys.73.33

Sicun Gao, Soonho Kong, and Edmund M Clarke. 2013. dReal: An SMT solver for nonlinear theories over the reals. In *Automated Deduction–CADE-24: 24th International Conference on Automated Deduction, Lake Placid, NY, USA, June 9-14, 2013. Proceedings 24*. Springer, 208–214. https://doi.org/10.1007/978-3-642-38574-2_14

Alexey Vyacheslavovich Gorshkov, M Hermele, V Gurarie, C Xu, Paul S Julienne, J Ye, Peter Zoller, Eugene Demler, Mikhail D Lukin, and AM Rey. 2010. Two-orbital SU (N) magnetism with ultracold alkaline-earth atoms. *Nature physics* 6, 4 (2010), 289–295. https://doi.org/10.1038/nphys1535

Daniel Gottesman. 2010. An introduction to quantum error correction and fault-tolerant quantum computation. In *Quantum information science and its contributions to mathematics, Proceedings of Symposia in Applied Mathematics*, Vol. 68. 13–58. https://doi.org/10.48550/arXiv.0904.2557

Alexander S. Green, Peter LeFanu Lumsdaine, Neil J. Ross, Peter Selinger, and Benoît Valiron. 2013. Quipper: A Scalable Quantum Programming Language. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Seattle, Washington, USA) *(PLDI '13)*. Association for Computing Machinery, New York, NY, USA, 333–342. https://doi.org/10.1145/2491956.2462177

Sean Greenaway, Adam Smith, Florian Mintert, and Daniel Malz. 2022. Analogue Quantum Simulation with Fixed-Frequency Transmon Qubits. *arXiv preprint arXiv:2211.16439* (2022). https://doi.org/10.48550/arXiv.2211.16439

Aric Hagberg, Pieter Swart, and Daniel S Chult. 2008. *Exploring network structure, dynamics, and function using NetworkX*. Technical Report. Los Alamos National Lab.(LANL), Los Alamos, NM (United States).

Kesha Hietala, Robert Rand, Shih-Han Hung, Xiaodi Wu, and Michael Hicks. 2021. A Verified Optimizer for Quantum Circuits. *Proc. ACM Program. Lang.* 5, POPL, Article 37 (jan 2021), 29 pages. https://doi.org/10.1145/3434318

Walter Hofstetter and Tao Qin. 2018. Quantum simulation of strongly correlated condensed matter systems. *Journal of Physics B: Atomic, Molecular and Optical Physics* 51, 8 (2018), 082001. https://doi.org/10.1088/1361-6455/aaa31b

J.R. Johansson, P.D. Nation, and Franco Nori. 2012. QuTiP: An open-source Python framework for the dynamics of open quantum systems. *Computer Physics Communications* 183, 8 (2012), 1760–1772. https://doi.org/10.1016/j.cpc.2012.02.021

A Yu Kitaev. 1997. Quantum computations: algorithms and error correction. *Russian Mathematical Surveys* 52, 6 (1997), 1191. https://doi.org/10.1070/RM1997v052n06ABEH002155

Christoph Kloeffel and Daniel Loss. 2013. Prospects for spin-based quantum computing in quantum dots. *Annu. Rev. Condens. Matter Phys.* 4, 1 (2013), 51–81. https://doi.org/10.1146/annurev-conmatphys-030212-184248

Henning Labuhn, Daniel Barredo, Sylvain Ravets, Sylvain De Léséleuc, Tommaso Macrì, Thierry Lahaye, and Antoine Browaeys. 2016. Tunable two-dimensional arrays of single Rydberg atoms for realizing quantum Ising models. *Nature* 534, 7609 (2016), 667–670. https://doi.org/10.1038/nature18274

David Lauvergnat, Sophie Blasco, Xavier Chapuisat, and André Nauts. 2007. A simple and efficient evolution operator for time-dependent Hamiltonians: the Taylor expansion. *The Journal of chemical physics* 126, 20 (2007), 204103. https://doi.org/10.1063/1.2735315

Jiaqi Leng, Ethan Hickman, Joseph Li, and Xiaodi Wu. 2023. Quantum Hamiltonian Descent. *arXiv preprint arXiv:2303.01471* (2023). https://doi.org/10.48550/arXiv.2303.01471

Gushu Li, Anbang Wu, Yunong Shi, Ali Javadi-Abhari, Yufei Ding, and Yuan Xie. 2022. Paulihedral: A Generalized Block-Wise Compiler Optimization Framework for Quantum Simulation Kernels. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) *(ASPLOS '22)*. Association for Computing Machinery, New York, NY, USA, 554–569. https://doi.org/10.1145/3503222.3507715

Seth Lloyd. 1996. Universal quantum simulators. *Science* 273, 5278 (1996), 1073–1078. https://doi.org/10.1126/science.273.5278.1073

Guang Hao Low and Isaac L Chuang. 2017. Optimal Hamiltonian simulation by quantum signal processing. *Physical review letters* 118, 1 (2017), 010501. https://doi.org/10.1103/PhysRevLett.118.010501

Moein Malekakhlagh, Easwar Magesan, and David C McKay. 2020. First-principles analysis of cross-resonance gate operation. *Physical Review A* 102, 4 (2020), 042605. https://doi.org/10.1103/PhysRevA.102.042605

Jarrod R McClean, Nicholas C Rubin, Kevin J Sung, Ian D Kivlichan, Xavier Bonet-Monroig, Yudong Cao, Chengyu Dai, E Schuyler Fried, Craig Gidney, Brendan Gimby, et al. 2020. OpenFermion: the electronic structure package for quantum computers. *Quantum Science and Technology* 5, 3 (2020), 034014. https://doi.org/10.1088/2058-9565/ab8ebc

David C McKay, Christopher J Wood, Sarah Sheldon, Jerry M Chow, and Jay M Gambetta. 2017. Efficient Z gates for quantum computing. *Physical Review A* 96, 2 (2017), 022330. https://doi.org/10.1103/PhysRevA.96.022330

Felix Motzoi, Jay M Gambetta, Patrick Rebentrost, and Frank K Wilhelm. 2009. Simple pulses for elimination of leakage in weakly nonlinear qubits. *Physical review letters* 103, 11 (2009), 110501. https://doi.org/10.1103/PhysRevLett.103.110501

Prakash Murali, David C McKay, Margaret Martonosi, and Ali Javadi-Abhari. 2020. Software mitigation of crosstalk on noisy intermediate-scale quantum computers. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. 1001–1016. https://doi.org/10.1145/3373376.3378477

Benjamin Nachman, Davide Provasoli, Wibe A De Jong, and Christian W Bauer. 2021. Quantum algorithm for high energy physics simulations. *Physical review letters* 126, 6 (2021), 062001. https://doi.org/10.1103/PhysRevLett.126.062001

Michael A Nielsen and Isaac Chuang. 2002. Quantum computation and quantum information. https://doi.org/10.1017/CBO9780511976667

Kristen Nygaard and Ole-Johan Dahl. 1978. *The Development of the SIMULA Languages*. Association for Computing Machinery, New York, NY, USA, 439–480. https://doi.org/10.1145/800025.1198392

Jeremy L. O'brien, Akira Furusawa, and Jelena Vučković. 2009. Photonic quantum technologies. *Nature Photonics* 3, 12 (2009), 687–695. https://doi.org/10.1038/nphoton.2009.229

Jennifer Paykin, Robert Rand, and Steve Zdancewic. 2017. QWIRE: a core language for quantum circuits. *ACM SIGPLAN Notices* 52, 1 (2017), 846–858. https://doi.org/10.1145/3009837.3009894

Yuxiang Peng, Jacob Young, Pengyu Liu, and Xiaodi Wu. 2023a. *Artifact for SimuQ: a Framework for Programming Quantum Hamiltonian Simulation with Analog Compilation.* https://doi.org/10.5281/zenodo.8423710

Yuxiang Peng, Jacob Young, Pengyu Liu, and Xiaodi Wu. 2023b. SimuQ: A Framework for Programming Quantum Hamiltonian Simulation with Analog Compilation (Extended Version). *arXiv preprint arXiv:2303.02775* (2023). https://doi.org/10.48550/arXiv.2303.02775

Connor Powers, Lindsay Bassman, Thomas M. Linker, Ken ichi Nomura, Sahil Gulania, Rajiv K. Kalia, Aiichiro Nakano, and Priya Vashishta. 2021. MISTIQS: An open-source software for performing quantum dynamics simulations on quantum computers. *SoftwareX* 14 (2021), 100696. https://doi.org/10.1016/j.softx.2021.100696

John Preskill. 2018. Quantum computing in the NISQ era and beyond. *Quantum* 2 (2018), 79. https://doi.org/10.22331/q-2018-08-06-79

QuEra. 2022. Bloqade: a Julia package for quantum computation and quantum dynamics based on neutral-atom architectures. https://queracomputing.github.io/Bloqade.jl/dev/

Mark Saffman. 2016. Quantum computing with atomic qubits and Rydberg interactions: progress and challenges. *Journal of Physics B: Atomic, Molecular and Optical Physics* 49, 20 (2016), 202001. https://doi.org/10.1088/0953-4075/49/20/202001

Peter Schauss. 2018. Quantum simulation of transverse Ising models with Rydberg atoms. *Quantum Science and Technology* 3, 2 (2018), 023001. https://doi.org/10.1088/2058-9565/aa9c59

Albert T Schmitz, Nicolas PD Sawaya, Sonika Johri, and AY Matsuura. 2021. Graph optimization perspective for low-depth Trotter-Suzuki decomposition. *arXiv preprint arXiv:2103.08602* (2021). https://doi.org/10.48550/arXiv.2103.08602

Ulrich Schollwöck. 2005. The density-matrix renormalization group. *Reviews of modern physics* 77, 1 (2005), 259. https://doi.org/10.1103/RevModPhys.77.259

Ulrich Schollwöck. 2011. The density-matrix renormalization group in the age of matrix product states. *Annals of physics* 326, 1 (2011), 96–192. https://doi.org/10.1016/j.aop.2010.09.012

G. Semeghini, H. Levine, A. Keesling, S. Ebadi, T. T. Wang, D. Bluvstein, R. Verresen, H. Pichler, M. Kalinowski, R. Samajdar, A. Omran, S. Sachdev, A. Vishwanath, M. Greiner, V. Vuletić, and M. D. Lukin. 2021. Probing topological spin liquids on a programmable quantum simulator. *Science* 374, 6572 (2021), 1242–1247. https://doi.org/10.1126/science.abi8794

Yunong Shi, Pranav Gokhale, Prakash Murali, Jonathan M. Baker, Casey Duckering, Yongshan Ding, Natalie C. Brown, Christopher Chamberland, Ali Javadi-Abhari, Andrew W. Cross, David I. Schuster, Kenneth R. Brown, Margaret Martonosi, and Frederic T. Chong. 2020. Resource-Efficient Quantum Computing by Breaking Abstractions. *Proc. IEEE* 108, 8 (2020), 1353–1370. https://doi.org/10.1109/JPROC.2020.2994765

Henrique Silvério, Sebastián Grijalva, Constantin Dalyac, Lucas Leclerc, Peter J. Karalekas, Nathan Shammah, Mourad Beji, Louis-Paul Henry, and Loïc Henriet. 2022. Pulser: An open-source package for the design of pulse sequences in programmable neutral-atom arrays. *Quantum* 6 (Jan. 2022), 629. https://doi.org/10.22331/q-2022-01-24-629

Anders Sørensen and Klaus Mølmer. 2000. Entanglement and quantum computation with ions in thermal motion. *Physical Review A* 62, 2 (2000), 022311. https://doi.org/10.1103/PhysRevA.62.022311

John PT Stenger, Nicholas T Bronn, Daniel J Egger, and David Pekker. 2021. Simulating the dynamics of braiding of Majorana zero modes using an IBM quantum computer. *Physical Review Research* 3, 3 (2021), 033171. https://doi.org/10.1103/PhysRevResearch.3.033171

Bochen Tan and Jason Cong. 2020. Optimal layout synthesis for quantum computing. In *Proceedings of the 39th International Conference on Computer-Aided Design*. 1–9. https://doi.org/10.1145/3400302.3415620

Swamit S Tannu and Moinuddin K Qureshi. 2019. Mitigating measurement errors in quantum computers by exploiting state-dependent bias. In *Proceedings of the 52nd annual IEEE/ACM international symposium on microarchitecture*. 279–290. https://doi.org/10.1145/3352460.3358265

Ewout Van Den Berg and Kristan Temme. 2020. Circuit optimization of Hamiltonian simulation by simultaneous diagonalization of Pauli clusters. *Quantum* 4 (2020), 322. https://doi.org/10.22331/q-2020-09-12-322

Pauli Virtanen, Ralf Gommers, Travis E Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, et al. 2020. SciPy 1.0: fundamental algorithms for scientific computing in Python. *Nature methods* 17, 3 (2020), 261–272. https://doi.org/10.1038/s41592-019-0686-2

Göran Wendin. 2017. Quantum information processing with superconducting circuits: a review. *Reports on Progress in Physics* 80, 10 (2017), 106001. https://doi.org/10.1088/1361-6633/aa7e1a

Bing Yang, Hui Sun, Robert Ott, Han-Yi Wang, Torsten V Zache, Jad C Halimeh, Zhen-Sheng Yuan, Philipp Hauke, and Jian-Wei Pan. 2020. Observation of gauge invariance in a 71-site Bose–Hubbard quantum simulator. *Nature* 587, 7834 (2020), 392–396. https://doi.org/10.1038/s41586-020-2910-8

Erez Zohar, J Ignacio Cirac, and Benni Reznik. 2015. Quantum simulations of lattice gauge theories using ultracold atoms in optical lattices. *Reports on Progress in Physics* 79, 1 (2015), 014401. https://doi.org/10.1088/0034-4885/79/1/014401

Received 2023-07-11; accepted 2023-11-07