SuperGrad: a differentiable simulator for superconducting processors

Ziang Wang[1], Feng Wu[2], Hui-Hai Zhao[2], Xin Wan[1], and Xiaotong Ni[3]

[1]Zhejiang Institute of Modern Physics and Zhejiang Key Laboratory of Micro-nano Quantum Chips and Quantum Control, Zhejiang University, Hangzhou 310027, China
[2]Zhongguancun Laboratory, Beijing, China
[3]Quantum Science Center of Guangdong-Hong Kong-Macao, Shenzhen 518045, China

One significant advantage of superconducting processors is their extensive design flexibility, which encompasses various types of qubits and interactions. Given the large number of tunable parameters of a processor, the ability to perform gradient optimization would be highly beneficial. Efficient backpropagation for gradient computation requires a tightly integrated software library, for which no open-source implementation is currently available. In this work, we introduce SuperGrad, a simulator that accelerates the design of superconducting quantum processors by incorporating gradient computation capabilities. SuperGrad offers a user-friendly interface for constructing Hamiltonians and computing both static and dynamic properties of composite systems. This differentiable simulation is valuable for a range of applications, including optimal control, design optimization, and experimental data fitting. In this paper, we demonstrate these applications through examples and code snippets. The code is available at `https://github.com/iqubit-org/supergrad`.

Contents

Hui-Hai Zhao: zhaohuihai@iqubit.org
Xiaotong Ni: xiaotong.ni@gmail.com

1 Introduction

Superconducting quantum processors have made significant strides recently, with their performance steadily improving and approaching the fault-tolerance threshold [1, 2, 3, 4, 5, 6, 7, 8]. These processors have not only demonstrated successful simulations of various quantum systems [9, 10, 11, 12, 13, 14] but have also shown promise in running other applications [15, 16]. As we inch closer to realizing the full potential of these quantum devices, it is natural to seek ways to optimize their parameters to maximize performance across various tasks. However, as discussed in [17, 6], simulating and optimizing superconducting quantum processors for error correction is a challenging endeavor. These processors are intrinsically quantum many-body systems with numerous optimizable parameters, making the optimization process computationally demanding and complex.

To address these challenges, we introduce SuperGrad [18], a differentiable simulator for superconducting quantum processors. It is written in Python and offers a simple and intuitive interface for describing various types of superconducting qubits and their interactions. By also providing information about qubit control and the desired objective function, the library can efficiently compute the gradients with respect to the processor and control parameters through backpropagation. Intuitively, the gradients contain information about how each parameter should be changed to improve the processors, and this is the idea behind various optimizers based on gradient descent. In general, the multiplicative speedup achieved by using backpropagation to compute the gradients is proportional to the number of parameters. Therefore, for a time-consuming simulation with dozens of parameters, backpropagation can offer a decent multiplicative speedup and potentially a very large absolute speedup. After the gradients are computed, they can help us perform joint optimization of these parameters [19], which is beneficial because these parameters can be heavily intertwined in the Hamiltonian. The efficient gradient computation is a result of considering the simulation from constructing superconducting qubits to dynamics as a whole. Our library can be viewed as a combination of previous libraries such as SCQubits [20] and Qiskit Dynamics [21]. However, since they each focus on a specific part of the simulation, they cannot perform backpropagation of gradients between them.

SuperGrad can be used to optimize gate fidelities and logical error rates of quantum processors [17]. Furthermore, we foresee that our library will be useful for optimizing bosonic code processors [22, 23], where complex control schemes and scaling to more modes are ongoing challenges. Additionally, a very common task in experiments is extracting parameters from the Hamiltonian (see, for example [24]). This task can also be viewed as an optimization problem, where the objective function is the difference between experimental data and simulation data. The gradients computed using SuperGrad can again make the optimization process faster and more stable. This extends previous studies [25] by the ability to fit parameters using both time dynamics data and energy spectra data. We provide an example of fitting qubit parameters from experimental data in Sec. 3.5.

SuperGrad is based on the JAX library [26], leveraging its ability to compute gradients of NumPy-like programs and accelerate computations with GPUs. The main challenge in designing SuperGrad was managing the various parameters in a scalable way while maintaining compatibility with JAX's automatic differentiation (auto-diff) functionality. We employed the Haiku library [27] to achieve this task, as discussed in Sec. 5.1.

In the current SuperGrad library, we do not focus extensively on the efficiency of numerical solvers, as simulating many-body processors to an accuracy suitable for predicting quantum error correction performance remains an open problem. The main time evolution solver is based on Trotterization [28, 29, 30]. It is known that naively computing the gradients through backpropagation will lead to a large memory cost proportional to the number of steps for discretizing the time interval. The common countermeasure to save memory costs is using the continuous adjoint method [31, 32, 33]. In the library, we implement an improved variant of the continuous adjoint method, which utilizes the locality of the Hamiltonians.

The remainder of this paper is structured as follows. Sec. 2 presents background information on superconducting quantum processors and introduces our approach to Hamiltonian simulation and optimization. Sec. 3 provides a hands-on tutorial for designing a quantum processor and showcases examples of Super-

Grad's capabilities. Sec. 4 benchmarks SuperGrad's performance for differentiable simulation. Sec. 5 shows the extensibility of SuperGrad and future work.

2 Implementation

2.1 Hamiltonians of superconducting processors

In general, a multi-qubit superconducting circuit comprises numerous qubit modes that should thus be treated as a many-body system. Within the realm of superconducting quantum processors, the dynamics of this system are efficiently captured by 1-body and 2-body Hamiltonians. External control pulses can be applied to individual qubit operators, preserving the structure of the time-dependent Hamiltonian as outlined in Eqn. 1. The Hamiltonian of the composited system $\mathcal{H}(t)$ that acts on N qubits can be expressed as a sum of local Hamiltonians, denoted as

$$\mathcal{H}(t) = \sum_i H_i(t) + \sum_{e(G)} H_{ij}, \tag{1}$$

where the 1-body Hamiltonian $H_i(t)$ and 2-body Hamiltonian H_{ij} act non-trivially only on the corresponding qubits i and j. Here, $e(G)$ represents the edges of the graph G, which indicate the interacting pairs within the quantum processor. Notably, these couplings can be arbitrarily long-ranged in the graph.

The initial step in simulating a quantum processor involves calculating the low-energy basis for each qubit. Following this, we derive the Hamiltonian for the composite quantum system. In circuit quantum electrodynamics (circuit-QED), the Hamiltonian of superconducting qubits involves device parameters such as Josephson energy (E_J), charging energy (E_C), inductive energy (E_L), and the coupling strengths (J_C, J_L).

The next step of simulating a quantum processor's dynamics necessitates solving the time-dependent Schrödinger equation [34, 21, 35]. In scenarios where the quantum processor is influenced by external control pulses, the driving frequency (ω_d) and the pulse amplitude (ϵ_d) become relevant control parameters. These parameters are crucial for characterizing quantum processors in both simulation and experimental setups. Furthermore, our library supports the computation of gradients with respect to these parameters.

2.2 Library architecture

We suggest incorporating the above steps to compute the dynamics within a unified and differentiable framework to streamline the simulation process. This approach offers two primary advantages:

1. Enable the computation of gradients with respect to both control parameters and device parameters.
2. Defining a unified data structure to bridge the quantized Hamiltonian and the ODE solver which facilitates the adoption of advanced solvers.

Consequently, this method allows for the optimization of quantum processor designs through direct time evolution simulations, offering a comprehensive solution for differentiable quantum processor simulation. An overview of the SuperGrad library is illustrated in Fig. 1. A brief description of each component is as follows:

- **SCGraph** provides a comprehensive representation of the superconducting quantum processor structure, incorporating time-dependent control pulses.
- **CircuitLCJ and InteractingSystem**: **CircuitLCJ** computes the Hamiltonians and operators for each qubit, whereas **InteractingSystem** computes the Hamiltonian for the quantum processor.
- **KronObj and LindbladObj** store the Hamiltonian and operators in their local forms and are designed to solve the Schrödinger equation and the Lindblad master equation, respectively.
- **ODE solvers** such as Trotterization and Dormand-Prince can compute gradients by the continuous adjoint method.
- **Helper** processes **SCGraph** to generate differentiable functions for specific computations. This component includes Evolve (for evaluating time evolution) and Spectrum (for assessing static properties).

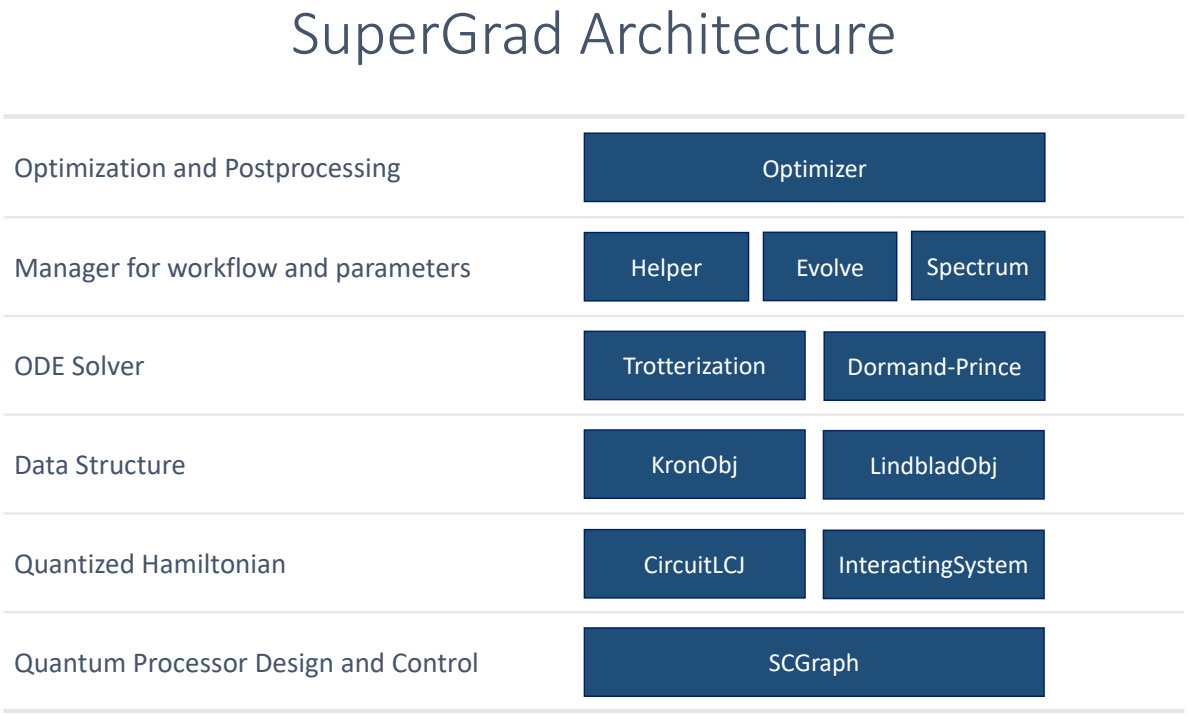


Figure 1: The overview of the SuperGrad library with the components described in Sec. 2.2.

2.3 Trotterization solver

An efficient method for simulating time evolution utilizes Trotterization. This technique decomposes the time evolution operator into a sequence of local operators, facilitating the efficient computation of time evolution using 1-body and 2-body operators. Consequently, memory costs are primarily influenced by quantum states, with the memory cost scaling as $O(d^N)$ where N is the number of qudits, and d represents the dimension of each qudit subsystem. From an alternative perspective, the quantum state may be regarded as a high-rank state tensor. Wherein the product of a local operator and state is calculated through tensor contraction, with the time cost scaling as $O(d^{N+2})$.

We define the time evolution operator by solving the time-dependent Schrödinger equation:

$$U(t_0, t_g) = \mathcal{T} \exp\left\{-i \int_{t_0}^{t_g} \mathcal{H}(t) dt\right\}, \tag{2}$$

where $\mathcal{T}$ exp denotes the time-ordered exponential. This operator $U(t_0, t_g)$ describes the time evolution from the initial time t_0 to the final time t_g.

We utilize the Suzuki-Trotter decomposition (STD) to restrict the matrix exponential to each local Hamiltonian separately. Initially, we employ the product formula to approximate the time evolution operator, dividing it into a sequence of operators over δt time intervals, ensuring that

$$\begin{aligned} U(t_0, t_g) = U(t_g - \delta t, t_g) U(t_g - 2\delta t, t_g - \delta t) \cdots \\ U(t_0 + \delta t, t_0 + 2\delta t) U(t_0, t_0 + \delta t). \end{aligned} \tag{3}$$

Additionally, we apply the n-th Suzuki-Trotter decomposition as a further approximation of the time evolution operator, which is denoted as

$$U(t, t + \delta t) = Q_n(\delta t) + O(\delta t^{n+1}), \tag{4}$$

where $Q_n(\delta t)$ is recursively constructed as

$$Q_n(\delta t) = \prod_{j=1}^{r} Q_{n-1}(p_{n,j} \delta t). \tag{5}$$

The parameters $p_{n,j}$ satisfy the decomposition conditions $\sum_{j=1}^{r} (p_{n,j})^n = 0$ with $\sum_{j=1}^{r} p_{n,j} = 1$. For a Hamiltonian of the form $\mathcal{H} = \sum H_k$, the first-order

Suzuki-Trotter decomposition is simply

$$Q_1(\delta t) = \prod \exp\{-iH_k \delta t\}. \tag{6}$$

We derive the formulations for higher-order STD in Appendix A. Notably, the product of the STD operator with the quantum state corresponds to the contraction of a tensor network. Details of our implementation can be found in Appendix B.

2.4 Efficiency of computing gradients

To compare different gradient computation methods, we can look at the time overhead O_t and memory overhead O_m

$$O_t = \frac{T_{\text{grad}}}{T_{\text{forward}}}$$
$$O_m = \frac{M_{\text{grad}}}{M_{\text{forward}}},$$

where T_{forward}, T_{grad} is the time needed for computing f and ∇f, and similarly M is the memory needed for the computation. We care about the overhead instead of the time or memory costs because T_{forward} and M_{forward} often have exponentially scaling themselves in quantum simulators. O_t and O_m depend heavily on the concrete implementation of software and hardware, but we can still discuss their scaling with respect to some parameters. For example, O_t is proven to be smaller than 5 for a given set of elementary operations [36]. In practice, for the reverse-mode auto-diff done by libraries such as JAX or PyTorch, O_t is indeed around 5 for algorithms which are mostly matrix multiplications. However, O_m is unbounded when using the reverse-mode auto-diff. For example, O_m is proportional to the depth of feedforward neural networks with auto-diff, because all values of intermediate hidden neurons need to be stored. The same issue exists if we try to use auto-diff for Schrödinger's equation. Moreover, the overhead has a larger impact when simulating quantum systems because of the exponential memory cost of storing a n-qubit state.

2.5 Continuous adjoint method

To address the large memory overhead O_m, we explore the continuous adjoint method, a technique that significantly reduces memory costs associated with differentiating through ODEs. This approach treats the backpropagation problem as the task of solving an augmented ODE for the adjoint state. We also propose the local continuous adjoint method, which in addition utilizes the locality of terms in the Hamiltonian.

Here, we will briefly explain these methods. For the more rigorous derivation of the continuous adjoint method, we refer the reader to [33]. The main idea of the method is that we can simply do a reverse time evolution on the quantum states to avoid the need to store them. This is illustrated in Fig. 2. On the other hand, to understand this, it suffices to look at the time evolution after discretization. We can rewrite Eqn. 3 as

$$|\psi(t_g, \theta)\rangle = U_g(\theta)U_{g-1}(\theta)\cdots U_0(\theta)|\psi(t_0)\rangle. \tag{7}$$

Here, θ are the tunable parameters from the Hamiltonian $\mathcal{H}(\theta)$. For brevity, let us assume all quantities in the above equation are real numbers[1]. Then, we have

$$\frac{\partial|\psi(t_g,\theta)\rangle}{\partial\theta} = \sum_i U_g(\theta)\cdots\frac{\partial U_i(\theta)}{\partial\theta}\cdots U_0(\theta)|\psi(t_0)\rangle. \tag{8}$$

We note that the above equation already provides one way to compute the gradient with a memory cost independent of g, since we only need to keep track of the products of unitaries before and after $\frac{\partial U_i(\theta)}{\partial\theta}$. But with more careful treatment, we will show that we only need to store states with size d^N instead of matrices with size $d^N \times d^N$, where N is the number of qudits.

To do this, let us examine the i-th term in the summation of the above equation. We will use the simplified notation

$$|\psi_i\rangle = U_i(\theta)\cdots U_0(\theta)|\psi(t_0)\rangle. \tag{9}$$

We will also consider an objective function $\mathcal{L}(|\psi(t_g,\theta)\rangle)$ which only depends on the final state. By rewriting $\mathcal{L}$ as a function of $|\psi_i\rangle$

$$\mathcal{L}(|\psi(t_g,\theta)\rangle) = \mathcal{L}(U_g(\theta)\cdots U_{i+1}(\theta)|\psi_i\rangle), \tag{10}$$

we can consider the partial derivative $a_i = \frac{\partial\mathcal{L}}{\partial\psi_i}$ which is a column vector of size d^N. One can verify that the i-th term is

$$a_i^T \frac{\partial U_i(\theta)}{\partial\theta}|\psi_{i-1}\rangle, \tag{11}$$

[1] In general, the expression $\frac{\partial\mathcal{L}}{\partial z}$ is not well-defined for a complex number z. Therefore, we need to convert z to a pair of real numbers. See [37] for more details about how JAX handles complex numbers.

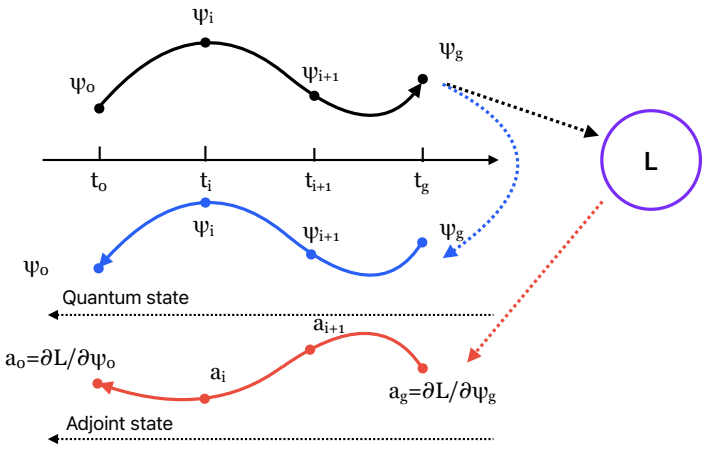


Figure 2: Outline of the continuous adjoint method. The black lines represent time evolution in the forward pass. After the final state ψ_g and the objective function $\mathcal{L}$ are computed, we can use the continuous adjoint method to compute the gradient. To do that, we can evolve ψ_g and a_g backward in time, which means we do not need to store them during the forward pass. During the backward evolution, we can accumulate the gradient according to Eqn. 8 and Eqn. 11.

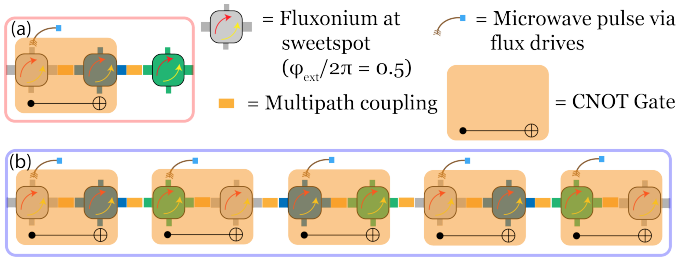


Figure 3: Sketch of a chain-like fluxonium quantum processor with fluxonium qubits biased at the sweetspot, and multipath coupling is used between neighboring qubit pairs. (a) The fluxonium 3-qubit chain arranged in a frequency pattern is highlighted in the red rectangle. Microwave pulses manipulate the gray qubit via its flux drive in Sec. 3.3. (b) 10-qubit chain-like quantum processor with simultaneous gate control. The fluxonium qubits are periodically arranged in a 3-qubit chain pattern, while their parameters have additional small deviations (see Sec. 3.4).

where we use the transpose since we already assume all vectors here are real. Therefore, we only need to keep track of a_i and ψ_i during backpropagation. By using the chain rule, we have

$$a_i = U_{i+1}^T a_{i+1}. \tag{12}$$

To compute ψ_i and a_i from ψ_{i+1} and a_{i+1}, we can use an approximation such as Suzuki-Trotter decomposition to write U_{i+1} as a product of 2-body terms. Then, we can compute $\frac{\partial U_i(\theta)}{\partial \theta}|\psi_{i-1}\rangle$ using an expansion similar to the R.H.S of Eqn. 8. This way, we can also avoid handling full $d^N \times d^N$ matrices during the computation of gradient $\frac{\partial \mathcal{L}}{\partial \theta}$. This is what we call the local continuous adjoint method in the following text.

3 Using the Library: Hands-On Examples

In this section, we provide a detailed guide on the use of SuperGrad, by using the multipath coupling scheme [38] in a fluxonium qubit chain as the example. The two-fluxonium cross-resonance gate has been successfully realized in the laboratory [39, 40], which uses a similar architecture. Our initial task involves optimizing the coupling strengths in a fluxonium 3-qubit system. Subsequently, we show how to construct a longer chain of fluxonium qubits based on a frequency pattern. Finally, we optimize this frequency pattern to enhance the fidelity of the simultaneous CNOT$^{\otimes 5}$ gate.

The tutorials are based on the current version of SuperGrad 0.2.3. Future releases may introduce breaking changes, and we will ensure that the documentation is kept up to date on GitHub [18].

3.1 Create a 3-qubit chain

This section includes a hands-on example using SuperGrad. We explore a 3-qubit chain consisting of three varied fluxonium qubits (highlighted by the red-lined rectangle with label (a) in Fig. 3). Both capacitive and inductive couplings exist between qubits, configured such that the left and right pairs exhibit identical coupling strengths. The device parameters for each fluxonium are sourced from multipath coupling reference designs [38]. The time-independent part of the 3-qubit chain Hamiltonian is

$$\mathcal{H}(0) = \sum_{1\leq i\leq 3} H_{f,i} + H_{12} + H_{23}, \tag{13}$$

where $H_{f,i}$ is the Hamiltonian for the individual fluxonium qubit. It takes the form as

$$H_{\mathrm{f},i} = 4E_{\mathrm{C},i}\hat{n}_i^2 + \frac{1}{2}E_{\mathrm{L},i}(\hat{\varphi}_i + \varphi_{\mathrm{ext},i})^2 - E_{\mathrm{J},i}\cos(\hat{\varphi}_i), \tag{14}$$

where $\hat{\varphi}_i$ is the phase operator, and $\hat{n}_i$ is the conjugate charge operator. $\varphi_{\mathrm{ext},i}$ represents the external flux.

We will keep $\varphi_{\text{ext},i} = \pi$ to set the fluxonium qubits at their sweetspot. The multipath coupling terms satisfy

$$H_{ij} = J_{\text{C}} \hat{n}_i \hat{n}_j - J_{\text{L}} \hat{\varphi}_i \hat{\varphi}_j, \tag{15}$$

where J_{C} and J_{L} are the strengths of capacitive and inductive coupling.

```python
import numpy as np

fluxonium_1 = {
    "ec": 1.0 * 2 * np.pi,
    "ej": 4.0 * 2 * np.pi,
    "el": 0.9 * 2 * np.pi,
    "shared_param_mark": "grey",
    "phiext": np.pi,
    "system_type": "fluxonium",
}

fluxonium_2 = {
    "ec": 1.0 * 2 * np.pi,
    "ej": 4.0 * 2 * np.pi,
    "el": 1.0 * 2 * np.pi,
    "shared_param_mark": "blue",
    "phiext": np.pi,
    "system_type": "fluxonium",
}

fluxonium_3 = {
    "ec": 1.0 * 2 * np.pi,
    "ej": 4.0 * 2 * np.pi,
    "el": 1.1 * 2 * np.pi,
    "shared_param_mark": "green",
    "phiext": np.pi,
    "system_type": "fluxonium",
}

coupling = {
    "capacitive_coupling": {
        "strength": 0.02 * 2 * np.pi},
    "inductive_coupling": {
        "strength": -0.002 * 2 * np.pi},
}
```

Listing 1: Python Dictionaries for parameters of a 3-qubit chain example. Each parameter is defined using specific keywords.

The parameters of each qubit and their couplings are managed via Python Dictionaries and can be exported as JSON files for subsequent utilization. We now explore the structure of the dictionary used in our implementation, detailed in Listing 1. The dictionary defines each qubit type as fluxonium and sets the external flux to bias the fluxonium at its sweetspot. In this work, we will employ the specified fluxonium parameters in Listing 1 that form a frequency pattern. Each parameter set is uniquely labeled with distinct colors, such as **grey**, **blue** and **green**. Those colors should be declared in **shared_param_mark** for parameter sharing.

In addition, we depict the topology of the quantum processor using a graph-based approach. In this representation, nodes correspond to qubits while edges represent the two-qubit interactions. This graph-based methodology facilitates the adaptation of various quantum processor configurations into a graph, leveraging the data structures and algorithms provided by NetworkX [41]. We develop a class **SCGraph** based on NetworkX's graph. In Listing 2, we show how to define a 3-qubit chain using **SCGraph** class. This class effectively encapsulates all relevant details about our quantum system, including the Hamiltonian parameters and the couplings.

```python
from supergrad.scgraph.graph import SCGraph

class MultipathThreeQubit(SCGraph):
    def __init__(self):
        super().__init__()

        # nodes represent qubits
        self.add_node("q1", **fluxonium_1)
        self.add_node("q2", **fluxonium_2)
        self.add_node("q3", **fluxonium_3)
        # edges represent two-qubit
    interactions
        self.add_edge("q1", "q2",**coupling)
        self.add_edge("q2", "q3",**coupling)
```

Listing 2: The **SCGraph** stores the structure and parameters of a quantum processor. We attach parameter dictionaries to nodes and edges of the graph.

We provide tools in the **Helper** subpackage to compute common quantities such as static spectra and the time evolution unitary matrices based on the provided graphs. More concretely, **Helper** can turn these computations into pure functions, which are functions without any side effects. This is a prerequisite for function transformation in JAX (see Sec. 5.1 for more discussion). We will sometimes refer to such functions as models. To demonstrate the **Helper** utility in converting **SCGraph** into models, we engage it to compute the eigenenergies of the composite quantum system and identify the dressed states. The output of this eigenenergy computation is a tensor with N indices, where each index is associated with a label

of the corresponding bare state. The Python code snippet Listing 3 exemplifies this process.

```
from supergrad.helper import Spectrum

chain_3q = MultipathThreeQubit()
spec = Spectrum(chain_3q, share_params=True,
    unify_coupling=True)
params = spec.all_params
energy = spec.energy_tensor(params)
dressed_freq_q1 = (energy[1, 0, 0] - energy
    [0, 0, 0])
dressed_freq_q2 = (energy[0, 1, 0] - energy
    [0, 0, 0])
print(dressed_freq_q1 / 2 / np.pi)
# ω̄01/2π = 0.499 GHz
print(dressed_freq_q2 / 2 / np.pi)
# ω̄01/2π = 0.582 GHz
```

Listing 3: The code block about computing eigenenergies of the composite quantum system and identifying the dressed states.

In the code Listing 3, the class property **spec.all_params** is responsible for extracting all relevant parameters from the graph. These parameters are subsequently passed as arguments to the pure function **spec.energy_tensor**. The pure function can be transformed by JAX to compute gradients or other functions if needed. Users can further customize or expand the functionality by writing their own **Helper**.

Following [38], let us analyze the static longitudinal coupling rate. The rate of the left qubit pair is

$$\zeta_{\mathrm{ZZI}} = \omega_{|000\rangle} + \omega_{|110\rangle} - \omega_{|100\rangle} - \omega_{|010\rangle}. \tag{16}$$

Here, ω_{ijk} is the dressed state's eigenenergy where the dressed state is close to the bare state $|ijk\rangle$. Ideally, we want ζ_{ZZI} to be 0. Let us keep the rightmost qubit in its ground state, and optimize the capacitive coupling parameter J_C by sweeping it. This is done in Listing 4. We adopt a simplified approach named unify-coupling, which shares parameters between all coupling configurations. This is enabled by setting **share_params** and **unify_coupling** to True.

```
import jax
import jax.numpy as jnp

def compute_static_longitudinal_coupling(jc)
    :
    params = spec.all_params
    params["capacitive_coupling_all_unify"].
    update({"strength": jnp.array(jc)})
    energy = spec.energy_tensor(params)
    zeta_zzi = (
        energy[0, 0, 0] + energy[1, 1, 0] -
    energy[0, 1, 0] - energy[1, 0, 0]
    )
    return (zeta_zzi / 2 / np.pi * 1e6)**2

jc_sweep = jnp.linspace(0.0, 80e-3, 1000)
zeta_sweep = jax.vmap(
    compute_static_longitudinal_coupling)(
    jc_sweep * 2 * jnp.pi)
```

Listing 4: The code block about computing static longitudinal coupling rate while sweeping the capacitive coupling parameters.

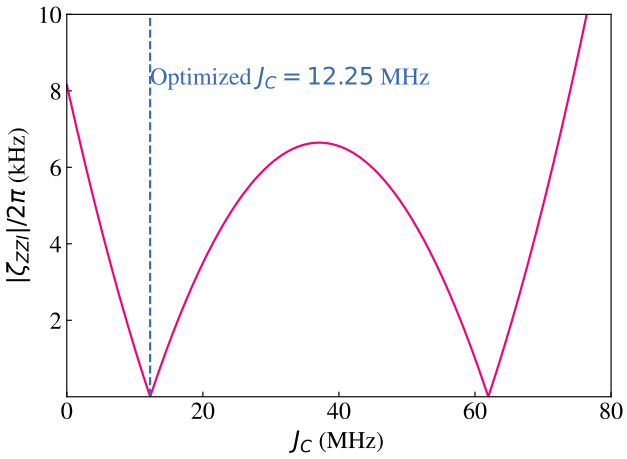


Figure 4: Static longitudinal coupling rate ζ_{ZZI} as a function of the capacitive coupling strength J_C. The dotted blue line labels optimized J_C found in Listing 5.

3.2 Optimization via JAX Framework

The JAX framework offers efficient computation of objective function values and gradients through automatic differentiation, which can be used to speed up optimization.

In Listing 5, we perform the minimization of ζ_{ZZI} over J_C when J_L is fixed at 2 MHz. We use the BFGS optimizer provided by SciPy. In this example, we use **jax.grad** to transform **compute_static_longitudinal_coupling** function to its gradient function by performing automatic differentiation. Then, the gradient function is passed to the minimizer.

```python
from scipy.optimize import minimize

func = lambda x0:
    compute_static_longitudinal_coupling(*x0)
res = minimize(func, 0.1, method='bfgs', jac
    =jax.grad(func))
jc_opt = res.x / 2 / np.pi * 1000
print(jc_opt)
# J_C = 12.25 MHz
```

Listing 5: Code block demonstrating the optimization of the static longitudinal coupling rate using a gradient-based algorithm.

The multipath coupling incorporates both an inductive coupling term $-J_L\hat{\varphi}_A\hat{\varphi}_B$ and a capacitive coupling term $J_C\hat{n}_A\hat{n}_B$ to suppress residual static ZZ interactions. When the inductive coupling constant J_L is fixed at 2 MHz, our goal is to fine-tune the capacitive coupling to minimize the longitudinal coupling rate ζ_{ZZI}. We define ζ_{ZZI} in Eqn. 16 as the objective function as detailed in Listing 5. The optimization yields $J_C = 12.25$ MHz, delineated by the blue line in Fig. 4, which corroborates the optimal value identified through a parametric sweep of J_C.

3.3 Workflow for Time Evolution

This subsection describes the procedure for simulating the time evolution of a quantum system using the **Evolve** class within the **Helper** package, specifically focusing on the implementation of a Cross-Resonance (CR) pulse in a 3-qubit chain. First, we initialize pulse parameters for the time evolution, as detailed in the code block Listing 6. This step involves setting the length of the CR pulse and computing the effective coupling strength J_{eff} that is extracted from the multipath coupling term Eqn. 15, as well as the detuning between the dressed $0-1$ transition frequencies of qubits 1 and 2. The driving amplitude ϵ_d of the pulse is then computed based on the detuning and the effective coupling strength.

```python
from supergrad.helper import Evolve
from supergrad.utils import tensor,
    compute_fidelity_with_1q_rotation_axis
from supergrad.utils.gates import cnot

length = 100.0
detuning = jnp.abs(dressed_freq_q1 -
    dressed_freq_q2)
j_eff = 0.01 * 2 * np.pi
tau_eps_drive = np.pi / 2.0 * detuning /
    j_eff

cr_pulse = {
    "pulse": {
        "amp": tau_eps_drive / length,
        "omega_d": dressed_freq_q2,
        "phase": 0.0,
        "length": length,
        "pulse_type": "cos",
        "operator_type": "phi_operator",
        "delay": 0.0,
    }
}

cr_chain_3q = MultipathThreeQubit()
cr_chain_3q.add_node("q1", **cr_pulse)
```

Listing 6: Implementing a CNOT gate via a cross-resonance pulse. The computation of the initial guess is based on the dressed states' eigenenergies and effective coupling rate.

Subsequent to setting the initial parameters, the **Evolve** class is employed to simulate the time evolution of the system, detailed in the code block Listing 7. Hyperparameters are declared for the simulation, such as truncated dimension (the number of qudit's energy levels), parameter sharing (compute gradient with respect to the frequency pattern), and compensation options (the type of errorless single-qubit rotations before and after the time evolution operator). The unitary evolution of the CR pulse is computed by utilizing the **eigen_basis** method, which performs the time evolution simulation within the multi-qubit eigenbasis. This method employs a basis transform technique that computes the evolution within the eigenbasis while benefiting from the local Hamiltonian in the product basis. The detail is shown in Appendix D.

```python
target_unitary = tensor(cnot(), jnp.eye(2))
evo = Evolve(cr_chain_3q, truncated_dim=3,
    share_params=True, unify_coupling=True,
    compensation_option='no_comp')
params = evo.pulse_params
cr_unitary = evo.eigen_basis(evo.all_params)
fidelity, res_unitary = compute_fidelity_
    with_1q_rotation_axis(target_unitary,
    cr_unitary, compensation_option='
    arbit_single')
print(fidelity)  # 0.98824
```

Listing 7: Performing time evolution via the Evolve class and calculating the fidelity of the simulated result against the ideal CNOT gate. Arbitrary single-qubit rotations are added before and after the time evolution to implement the CNOT⊗I gate.

The CR pulse introduces an effective ZX term, which can lead to an implementation of a CNOT gate when augmented by single-qubit gates. Given the typically lower error rates of single-qubit gates, here we assume they are perfect. These single-qubit compensations are optimized inside the **compute_fidelity_with_1q_rotation_axis** method. The fidelity of the resultant unitary is measured against the target unitary and achieves a high value (0.98824), and the high fidelity affirms the successful implementation of the CNOT gate with the initial guess of pulse parameters. Additionally, we optimize pulse parameters, as illustrated in the accompanying code block Listing 8. This process employs backpropagation to calculate the gradients effectively. The function **scipy_optimize** is a wrapper to achieve the same functionality shown in Listing 5. After optimization, the fidelity of the CNOT $\otimes$ I gate is substantially improved (achieving 99.99%).

```
import haiku as hk
from supergrad.utils.optimize import
    scipy_minimize

def infidelity(params):
    params = hk.data_structures.merge(evo.
    all_params, params)
    cr_unitary = evo.eigen_basis(params)
    fidelity, res_unitary =
    compute_fidelity_with_1q_rotation_axis(
    target_unitary, cr_unitary,
    compensation_option='arbit_single')
    return jnp.abs(1 - fidelity)

scipy_minimize(infidelity, params, method='L
    -BFGS-B', logging=True)

#  message: CONVERGENCE: REL_REDUCTION_OF_F_
    <=_FACTR*EPSMCH
#  success: True
#   status: 0
#      fun: 1.716181357702684e-05
```

Listing 8: Code block showing the objective function and performing pulse parameter optimization.

3.4 Device and control gradient optimization

In this section, we explore a one-dimensional chain of fluxonium qubits with the optimized multipath coupling designs in Sec. 3.2. The qubits are characterized by nine frequency pattern parameters in Table 1 that form a frequency pattern along the chain. For our example, we employ a chain-like quantum processor comprising ten qubits. This setup is illustrated in Fig. 3. The initial values of the frequency pattern parameters are provided in Table 1. We vary E_L by ± 0.1GHz to create differences in the qubit spectrums. Notably, the precision fabrication of superconducting qubits presents challenges, leading to deviations in the final device parameters of the quantum processor post-fabrication. Interestingly, as mentioned in [42], these deviations can enhance the localization and addressability of the qubits. We model these deviations by assuming that the device parameters follow a normal distribution. The mean of this distribution corresponds to the values $E_{C/J/L}$ in Table 1, and the standard deviation is $0.01 \times E_{C/J/L}$.

Qubit	E_J (GHz)	E_C (GHz)	E_L (GHz)
A (Grey)	4.0	1.0	0.9
B (Blue)	4.0	1.0	1.0
C (Green)	4.0	1.0	1.1

Table 1: The frequency pattern for the fluxonium chain. The colors of qubits are referring to the colors in Fig. 3 and in Listing 1.

We extend the time evolution workflow as described in Sec. 3.3, to simulate systems of ten fluxonium qubits. This is achieved by applying simultaneous control pulses to execute CNOT$^{\otimes 5}$ quantum gates. SuperGrad provides a multitude of options for configuring time evolution, each affecting the simulation's accuracy and the performance of the solver. These options include the number of time steps, the Trotter order of the Trotterization solver, and the basis for the unitary matrix. The code template for implementing the CNOT$^{\otimes 5}$ gate is presented in Listing 9. The class **MPCFluxonium** is a **SCGraph** with many custom methods. For example, we use **create_cr_pulse** to add CR pulses in a way similar to Listing 6. We add random deviations to the device parameters by setting **add_random** to True. We enable the parameter sharing so that the gradient computed in Table 2 is with respect to the parameters in the frequency pattern.

```
from supergrad.scgraph.
    graph_mpc_fluxonium_1d import
    MPCFluxonium1D

chain = MPCFluxonium1D(n_qubit=10, periodic=
    False, seed=0)
```

```
chain.create_cr_pulse(
    ix_control_list=[0, 2, 4, 6, 8],
    ix_target_list=[1, 3, 5, 7, 9],
    tg_list=[100.0, 100.0, 100.0, 100.0,
    100.0], add_random=True)
evo = Evolve(chain, truncated_dim=2,
    share_params=True, unify_coupling=True,
    options={'astep': 2000,
             'trotter_order': 4})
# Compute the time evolution unitary in the
    eigenbasis.
sim_u = evo.eigen_basis(evo.all_params)
```

Listing 9: The code template for constructing a 10-qubit chain and performing simultaneous CNOT$^{\otimes 5}$ gates. The initial values for the pulses are set inside the **MPCFluxonium1D** class, in the same way as in Listing 7. The argument **astep** is the number of time steps during time evolution, which is equal to t_g/δ_t.

Given the substantial differences in optimal learning rates for each variable [17], here we will only demonstrate a proof of principle optimization loop. We first optimize the control parameters, then perform a single-step update of the fluxonium qubit parameters $E_{C/J/L}$. Finally, we execute another control optimization with the newly updated qubit parameters and reinitialized control parameters. The multipath coupling strengths remain unchanged in this optimization. We use $1-\mathcal{F}$ as the objective function to optimize the performance of the simultaneous CNOT$^{\otimes 5}$ gate. This optimization is carried out using the gradient-based algorithm L-BFGS-B [43]. The objective function during the optimization process is shown in Fig. 5. Following the optimization of the control parameters within the original frequency pattern, the gradients with respect to the device and control parameters are detailed in Table 2.

Our ultimate goal is to facilitate simultaneous gate operations within the quantum processor. To achieve this, we modify the frequency pattern based on the gradient table detailed in Table 2. Selecting appropriate learning rates for various parameters with disparate units can pose a challenge (for an in-depth discussion, refer to the appendix of [17]). In an ideal scenario, effective learning rates could be determined by calculating the Hessian matrix. However, in this tutorial, the coupling parameters remain constant, and $E_{C/J/L}$ are optimized according to the following learning rate:

$$E_{\mathrm{C/J/L},i} \to E_{\mathrm{C/J/L},i} - 0.01\mathrm{GHz}^2 \times \frac{\partial \mathcal{L}}{\partial E_{\mathrm{C/J/L},i}}. \quad (17)$$

This process results in a more favorable frequency pattern. The optimization processes for the control parameters for both the original and one-step optimized frequency patterns are recorded in Fig. 5. As the number of minimization steps (epochs) increases, we observe the objective function gradually converging, indicating an improvement in gate performance with the optimized frequency pattern compared to the original one. The fidelity of the simultaneous CNOT$^{\otimes 5}$ gate exhibits an increase following the one-step frequency pattern optimization.

Parameters and Gradients for Frequency Patterns			
Type	Parameter	Value (GHz)	Gradient
A (Grey)	$E_{C,A}$	1.000e+00	-4.280e-01
	$E_{J,A}$	4.000e+00	1.323e-01
	$E_{L,A}$	9.000e-01	-3.866e-01
B (Blue)	$E_{C,B}$	1.000e+00	8.286e-01
	$E_{J,B}$	4.000e+00	-2.582e-01
	$E_{L,B}$	1.000e+00	7.027e-01
C (Green)	$E_{C,C}$	1.000e+00	-9.257e-01
	$E_{J,C}$	4.000e+00	3.076e-01
	$E_{L,C}$	1.100e+00	-8.193e-01

Table 2: Gradients table for the objective function with the original frequency pattern and optimized control parameters.

3.5 Fitting experimental data

In this section, we give an example of using SuperGrad to characterize quantum processors through fitting experimental data. Our example is based on the spectrum measurement of a single fluxonium qubit. Our objective is to deduce the parameters in Eqn. 14 by measuring the energy eigenvalues when the fluxonium is at different external flux φ_{ext} in experiments. For example, we can measure the energy difference between the two lowest eigenvalues $\varepsilon_{01} = E_1(\varphi_{\mathrm{ext}}) - E_0(\varphi_{\mathrm{ext}})$ at varying φ_{ext}. We could also calculate the theoretical energy difference $f_{01}(\varphi_{\mathrm{ext}})$ by diagonalizing the Hamiltonian that contains learnable parameters $E_C, E_J, E_L, \varphi_{\mathrm{ext}}$. The caveat is that we cannot directly manipulate φ_{ext} in the experiment. Instead, we apply a voltage $x(\mathrm{V})$ that is approximately linear to φ_{ext}.

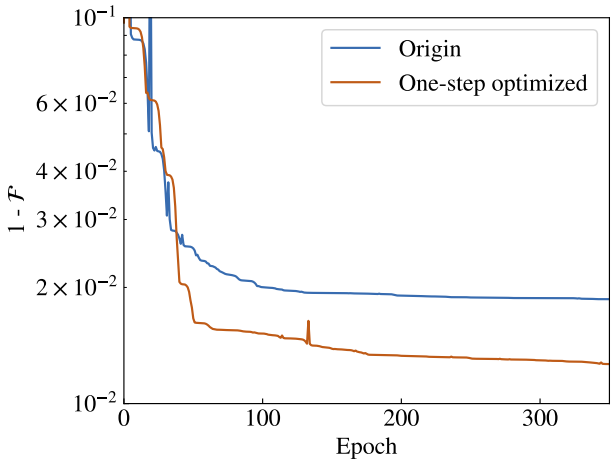


Figure 5: The optimization processes for the control parameters for both the original and one-step optimized frequency patterns are recorded. The x-axis represents the number of minimization steps (Epochs). The fidelity of the simultaneous $\mathrm{CNOT}^{\otimes 5}$ gate exhibits an increase following the one-step frequency pattern optimization.

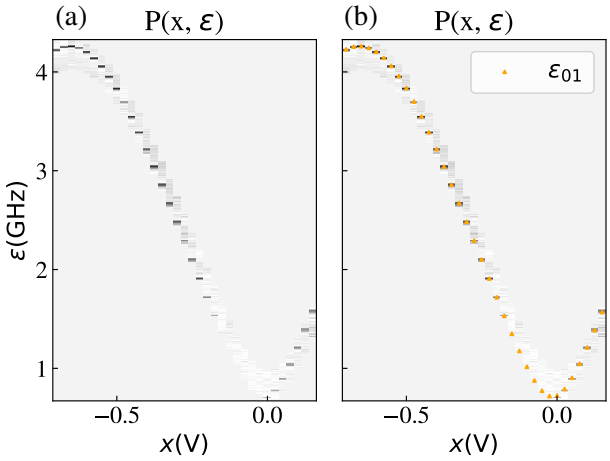


Figure 6: The experimental data and the fitting for the spectra measurement of a fluxonium qubit. The data are normalized as a probability distribution and then plotted as a color mesh from white to gray. The label of the x-axis is voltage, which is approximately linear with external flux φ_{ext}. After optimizing the KL-DIV objective function, we plot the result f_{01} as orange dots. (a) The experiment spectra measurement with proper normalization. (b) The fitted spectra dots together with the original measured data.

However, extracting ε_{01} from the experimental data is not always straightforward, particularly when the signal-to-noise ratio is low. Moreover, we might miss some signals if we are scanning a small part of the parameter space. For example, Fig. 6 shows the experimental data and a fitting that we will obtain at the end of this tutorial. If we simply locate peaks in every column, the columns without a signal might produce wrong ε_{01} values.

Here, we propose a novel way [44] to improve the robustness of the fitting. We will first normalize the data to be positive and sum to 1, and then treat it as a probability distribution $P(x, \varepsilon)$ on 2D. We construct the target probability distribution $Q(x, \varepsilon)$ based on numerical simulation of the fluxonium Hamiltonian. Q has learnable parameters including "linewidth" parameter λ, x-axis linear transformation parameters a, b, and a background noise parameter c. The $Q(x, \varepsilon)$ satisfies

$$Q(x,\varepsilon) = \frac{1}{Z}([\frac{\lambda}{(\varepsilon - f_{01}(ax+b))^2 + \lambda^2}] + c), \quad (18)$$

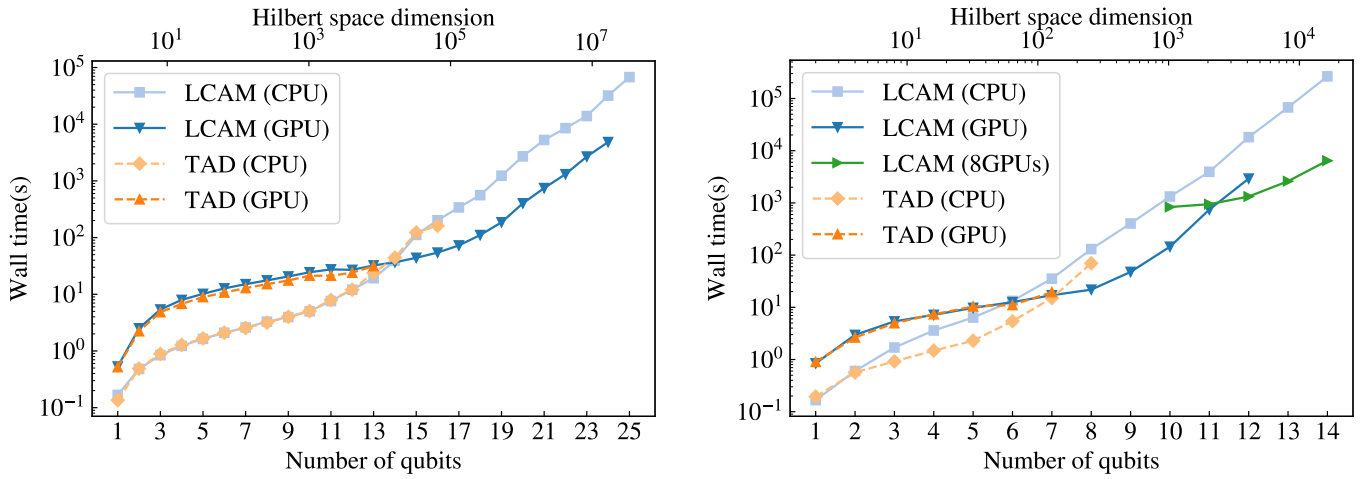
where Z is the normalization factor. We choose the objective function to be the Kullback-Leibler divergence (KL-DIV)

$$\mathcal{C}(P,Q) = \sum_{x,\varepsilon} P \ln(\frac{P}{Q}). \quad (19)$$

With SuperGrad, we can then compute the gradient and Hessian matrix of the objective function $\mathcal{C}$ utilizing higher-order automatic differentiation provided by the JAX framework. We employ the Newton-CG optimizer to minimize the objective function. The fitting result is depicted in Fig. 6.

4 Benchmarks

In this section, we benchmark SuperGrad performances for differentiable simulation, which comprises cost function evaluation and the corresponding backpropagation for gradient computation. We examine how the time and memory costs scale with the dimension of the Hilbert space by performing time evolution on N fluxonium qubits with multipath coupling, which is an extension of the Hamiltonian in Eqn. 13. In the time evolution, each qubit simultaneously undergoes an X gate operation. The detailed descriptions of the gate scheme are provided in Appendix C.



(a) Computing final state and gradients ($\delta t = 0.01, T = 50$)

(b) Computing unitary and gradients ($\delta t = 0.01, T = 50$)

Figure 7: Benchmarking comparison of the auto-diff of Trotterization (TAD) and our local continuous adjoint method (LCAM) focused on the wall time required to evaluate (a) final state fidelity and (b) gate fidelity, as well as gradients with respect to control and device parameters, respectively. The process wall time was measured against the number of qubits, and the results were minimized across three runs. For state differentiable simulation (a), the TAD incurs a significantly higher memory cost, approximately 38 times greater than LCAM for a chain of 16 qubits (in CPU). The TAD exceeds our GPU's memory limitation (32GB) for qubit chains larger than 13 qubits, while the LCAM method remains feasible until qubit chains exceed 24 qubits, at which point it also surpasses GPU memory limits. For unitary differentiable simulation (b), the memory cost of TAD is 30 times greater than LCAM for an 8-qubit chain (in CPU). TAD's memory cost is out of GPU memory for qubit chains larger than 7 qubits; in contrast, the LCAM method remains within GPU memory up to a qubit chain size of 12. By performing the unitary differentiable simulation in an 8GPUs configuration, the LCAM method remains available until qubit chains exceed 14 qubits. We note that this memory overhead of TAD is dependent on the time step size δt, which is chosen to be 0.01 ns.

Our experimental setup consists of an AMD® EPYC® 7513 CPU running at 3.2 GHz (32-core/64-threads), paired with 256 GB of random-access memory (RAM), and eight NVIDIA® Tesla® V100 GPUs, each equipped with 32 GB of memory. The operating system is Ubuntu 22.04 LTS, with CUDA 12.3 and JAX 0.4.33. The SuperGrad version is 0.2.3, and the source code for the benchmarks presented in this section is accessible at [18].

4.1 Benchmark backpropagation algorithm

For computing the time evolution unitaries, we will always use the Trotterization solver detailed in Sec. 2.3. Only the gradient computation method will be changed and compared.

We present benchmarks for the wall time across various numbers of qubits. For the differentiable simulation of the state as shown in Fig. 7a, we employ the target-state infidelity $\mathcal{L}(|\psi_g\rangle) = 1 - |\langle\psi_t|\psi_g\rangle|^2$ as the cost function, where $|\psi_t\rangle$ is the target final state. We measure the wall time including evaluating the final state and computing the gradients with respect to control and device parameters.

In Fig. 7b, we benchmark for optimizing the evolution unitary. It's achieved by computing the time evolution of a set of initial states $\{|\psi_0^s\rangle\}(s = 1, 2, \ldots, S)$. For this task, we use the composite state-transfer cost function [45], which takes the form as

$$\mathcal{L}(U(t_0, t_g)) = 1 - |\frac{1}{S}\sum_s \langle\psi_t^s|\psi_g^s\rangle|^2, \qquad (20)$$

where the set of target states $\{|\psi_t^s\rangle = U_t|\psi_0^s\rangle\}(s = 1, 2, \ldots, S)$ represents the column vectors of the target unitary U_t. This composite state-transfer cost function is equivalent to the gate fidelity when used over a complete basis. Note that the cost function could be computed using distributed methods to support multi-GPU configurations.

We computed gradients by the auto-diff of Trotterization (TAD) and the local continuous adjoint

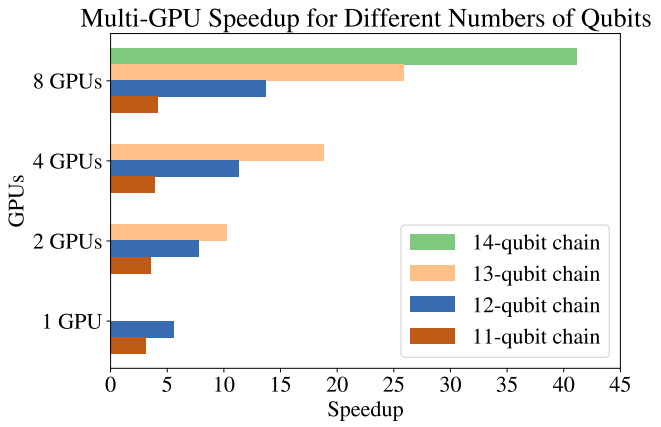


Figure 8: Benchmark of the multi-GPU speedup when computing gradients using LCAM with NVIDIA® Tesla® V100 32GB GPUs over CPU implementations on AMD® EPYC® 7513 CPU (32-core/64-threads) at 3.2GHz. The multi-GPU implementation could surpass the memory limitation of a single GPU by distributing unitary across multiple GPU instances. During our benchmark, 2 GPUs were able to simulate up to 13 qubits, while 8 GPUs could handle 14 qubits, both of them exceeding the memory capacity of a single GPU.

method (LCAM), denoted as $\mathbf{g}^{\text{TAD}}$ and $\mathbf{g}^{\text{LCAM}}$ respectively. To ensure the time evolution computation is accurate, we employ a time step size of $\delta t = 0.01$ ns and utilize a second-order Trotter decomposition to achieve a converged time evolution, where LCAM uses the same step size and Trotter order. Convergence is confirmed through comparisons between the time evolution unitary produced by the Trotterization solver with the unitary computed by Qiskit Dynamics using a very small ODE solver tolerance (dataset is shown in [46]). We evaluate the accuracy of LCAM by computing the relative error for each component in the gradient, which is listed in Table 5. We can see that the relative errors are all smaller than 10^{-7}. The LCAM offers significant memory efficiency (e.g., one-thirtieth of memory usage compared to TAD for computing eight qubits evolution unitary) as it only needs to store a single copy of the quantum state and the adjoint state, and it does not need to represent the Hamiltonian as a dense matrix for evaluating $\partial\mathcal{H}(t,\theta)/\partial\theta$.

4.2 Leveraging multi-GPU configuration

We have implemented multi-GPU support for time evolution unitary simulations. In our time evolution solvers, the largest memory cost is for storing the quantum states array, which scales as $O(d^{2N})$. Where the quantum processor has N qudits, and each qudit subsystem has a dimension d. By distributing the unitary across multiple devices, we can simulate larger quantum processors in parallel.

We benchmark the speedup of the multi-GPU backend over the CPU backend by performing the same time evolution for unitary as described in Sec. 4.1 on an N-qubit chain, where gradients are computed using the local continuous adjoint method. The speedup results shown in Fig. 8 are based on the minimal wall time across three runs. This benchmark indicates that larger Hilbert space sizes better harness the floating-point computational capabilities of the GPU. Additionally, employing multi-GPU configurations allows us to mitigate the memory limitations of individual GPUs. However, the number of qubits that can be simulated remains constrained by the exponential growth of the memory cost associated with a single quantum state. To address this limitation, we plan to explore the use of tensor networks in future work.

4.3 Comparison with other simulation software

In this subsection, we compare SuperGrad against similar software packages, such as SCQubits, QuTiP, and Qiskit Dynamics. We list their features in Table 3. SuperGrad stands out as the only library that integrates Hamiltonian construction and time evolution for superconducting processors. Therefore, it is capable of computing the gradients of control and device parameters in a single backpropagation. Qiskit Dynamics is the only one that supports perturbation-theory-based solvers, such as the Dyson series and Magnus expansion [47]. Qibo stands out for its advanced hardware accelerator support, which takes full advantage of GPU and multi-GPU devices [34].

To get a benchmark for the time and memory cost of these libraries, we will need to use a combination of SCQubits and QuTiP / Qiskit Dynamics to be near feature parity with SuperGrad. We again run the time evolution of $X^{\otimes 6}$ as the benchmark. The result is listed in Table 4.

	SuperGrad	SCQubits	QuTiP	Qiskit Dynamics	Qibo
Version	0.2.3	4.3.0	5.1.1	0.5.1	0.2.16
Hamiltonian construction	✓	✓	✗	✗	✗
Time evolution	✓	✗	✓	✓	✓
Gradients of device parameters	✓	✗	✗	✗	✗
Gradients of control parameters	✓	✗	Pre-Alpha	✓	✓
GPU acceleration	✓	✓	Pre-Alpha	✓	✓
Multi-GPU support	✓	✗	✗	✗	✓

Table 3: A comparison of the capabilities of SuperGrad and other packages reveals notable differences. All tools were benchmarked using the version listing in the table. SuperGrad stands out by providing the full simulation pipeline, from superconducting processor Hamiltonian construction to time evolution, whereas SCQubits is limited to Hamiltonian construction. To use the time evolution solver of QuTiP and Qiskit Dynamics, one needs to supply the Hamiltonians, e.g., from SCQubits. This entire process simulation support in SuperGrad enables efficient differentiable simulation, while Qiskit Dynamics and Qibo only support backpropagation for the time evolution part. QuTiP provides partial support for differentiable simulations and GPU acceleration via optional plug-ins (QuTiP-JAX and QuTiP-CuPy), but these features are in the pre-alpha stage. Both Qibo and SuperGrad support distributed time evolution across multiple GPUs, which is crucial for simulating larger quantum systems. Additionally, Qiskit Dynamics offers perturbation-theory-based solvers, which deliver a significant speed advantage over traditional solvers, though at the cost of increased initial compilation time [47].

For fairness, the ODE solver and its error control parameters (atol $= 1 \times 10^{-8}$, rtol $= 1 \times 10^{-8}$) were kept the same for both QuTiP and Qiskit Dynamics. We used the JAX backend of Qiskit Dynamics to perform computations. For SuperGrad, we employ a time step size of $\delta t = 0.01$ ns and utilize second-order Trotter decomposition. We observed the unitary infidelity $1-\mathcal{F}$ between SuperGrad and Qiskit Dynamics is smaller than 1×10^{-6}. Note that this does not imply that the results they computed have the same accuracies. Therefore, the time overheads for computing gradients are more important than the absolute times in Table 4.

All benchmarks were run on the CPU since some of these packages do not have GPU backends. The minimal wall time across three runs was recorded, and memory usage was defined by the maximum memory resident set size, as reported by the psutil library during computation. Due to Python's lazy garbage collection strategy, unused memory may not be released immediately, which affects the accuracy of memory usage benchmarks. This can lead to an overestimation of memory consumption, particularly in FDM computations. In theory, the memory usage for FDM computation should be equivalent to that of the corresponding forward computation.

One of SuperGrad's key contributions is its support for the differentiable simulation of both control and device parameters. On the other hand, Qiskit Dynamics can only compute gradients with respect to control parameters. We can still estimate the gradients with respect to the device parameters using the finite difference method (FDM). For FDM, we used forward differences, which require one additional computation per parameter, with a step size of 1×10^{-4} for finite differences. The results of the computation wall time and memory usage are shown in Table 4. These benchmarks demonstrate that SuperGrad computes gradients more efficiently than similar software, particularly when dealing with a large number of parameters.

5 Discussion

5.1 Library design

The first question we need to think about when designing the library is how to be compatible with JAX. The main requirement is that the functions which we want to compute the gradients are pure. A pure function is a function that, given the same input, will always return the same output and does not produce any side effects. On the other hand, it is natural to construct the library with some elements of object-oriented programming (OOP) in most modern languages such as Python, C++, etc. We have also adopted OOP for SuperGrad. The fundamental features of OOP, such as inheritance and object composition, provide a suitable framework for describing superconducting quantum processors. In our software

		SuperGrad	SCQubits + Qiskit Dynamics		SCQubits + QuTiP
ODE solver		Trotterization	Dormand-Prince		
Differential method		LCAM	CAM	FDM	
Gradient computation scope		Device and control	Control	Device and control	
Time	Forward f	2.02 s	9.28 s	9.28 s	173.81 s
	Gradients ∇f	8.55 s	14.73 s	554.66 s	10151.47 s
	Overhead O_t	4.23	1.59	59.77	58.41
Memory	Forward f	2.01 GB	1.96 GB	1.96 GB	1.83 GB
	Gradients ∇f	2.74 GB	2.97 GB	8.54 GB	6.31 GB
	Overhead O_m	1.36	1.52	4.36	3.45

Table 4: Benchmark of the performance of SuperGrad and other packages, focusing on the wall time and memory usage required to compute the simultaneous $X^{\otimes 6}$ gate fidelity and its gradient. For the case that the gradient computation scope is "control", gradients are computed only for control parameters (24 parameters in total). In contrast, in the cases where the gradient computation scope is "device and control", gradients are computed for both control and device parameters (58 parameters in total). Time and memory overheads are calculated relative to the corresponding forward simulation. The Dormand-Prince solver in Qiskit Dynamics supports the continuous adjoint method, which allows for efficient backpropagation of time evolution with a small time overhead, as it bypasses gradient computation for the device Hamiltonian. The case is labeled as FDM using the standard finite difference method (FDM), where time overheads are approximated to the number of parameters. Due to Python's garbage collection strategy, the memory usage is likely overestimated.

architecture, we conceptualize the local Hamiltonian terms, such as qubits, couplings, and control fields, as objects in OOP. Here are some examples for which OOP is useful:

1. Qubits can be categorized into specific types based on the structure of their Hamiltonians. This mirrors the OOP requirement for treating an object as an instance of a particular class.
2. As a quantum system, each qubit's eigenvalues could be computed in the same way (diagonalization). This is similar to the OOP class, which could inherit methods from other classes.
3. The quantum processor can be viewed as a complex composition of various objects, analogous to the concept of object composition in OOP, which allows for the assembly of complex entities from simpler ones.

In the OOP paradigm, objects have methods that define procedures for the object's behavior, allowing us to compute outputs by running object methods. However, object methods may not be pure functions that satisfy JAX's transformable requirements. For example, consider a fluxonium qubit biased at the sweetspot while external flux φ_{ext} is an inherited parameter. JAX's auto-diff engine will fail to compute the gradient with respect to φ_{ext} when given the object method. This occurs because JAX does not trace inherited parameters. This incompatibility between the OOP paradigm and JAX's pure function model is something that we should always keep in mind.

Neural network libraries such as Flax [48] and Haiku [27] ensure flexible usage of the JAX framework, mitigating the conflict between OOP and JAX's requirement for pure functions. In SuperGrad, we use Haiku to transform the quantum processor composed of objects into pure functions. Haiku can help us collect all parameters declared in a computation into a Python dictionary and pass the dictionary through a transformed pure function. In some quantum processor optimization tasks, it is reasonable to partition the parameters into trainable and non-trainable sets, and Haiku provides utilities for manipulating the parameters dictionary.

However, we are considering replacing Haiku with customized utilities. This is because Haiku is designed primarily for neural network tasks, e.g., sequentially transforming a tensor by neural network layers. This is not the case for superconducting processor simulation. For example, there are few intermediate results we want to extract or modify when computing time evolution from **SCGraph**. Therefore, we only need to make sure such functions starting from SCGraph can be auto-differentiated.

5.2 SuperGrad Enhancement Proposals

The usability of our current version SuperGrad 0.2.3 can be enhanced in several respects, particularly concerning the user's interaction and the system's flexibility:

1. Users are required to construct a "graph-like" structure to input all necessary data. This process could be streamlined with the development of a graphical user interface (GUI), which would also help in detecting typos.
2. Gradient computation and optimization processes require users to specify variable sharing and fix certain parameters. This selection process can be cumbersome. A well-designed GUI could include features to facilitate these selections, enhancing user interaction and system usability.
3. We are considering making **SCGraph** a JAX-differentiable data structure. The transformed pure functions will use **SCGraph** as input data, and the gradient of the objective function will take the same data structure as **SCGraph**. Moreover, the enhanced **SCGraph** will support functions for handling random deviations and parameter sharing, which further expands its functionality.
4. Beyond standard operations like computing eigenvalues or simulating time evolution, users may need to perform unconventional calculations. The current object-oriented programming (OOP) structure aims to understand the codebase; however, it still needs developer experience.

5.3 Role of SuperGrad in experiments

SuperGrad, like other simulation software, can be an invaluable tool in design optimization, cost-efficient development acceleration, pre-experimental planning, and post-experimental validation for fabricating superconducting processors, as is the case in other fields, such as aircraft, robotics, and semiconductor chip designs. While the exact simulation of a complex processor is generally unfeasible, the software complements expensive design iteration and device testing, and helps plan and understand the experimental measurements of device behavior. To showcase the multifaceted capabilities of SuperGrad, we provide the following two scenarios.

First, we often need to prototype novel processor designs with new types of qubits and couplings. For this purpose, we need to estimate important quantities such as qubit frequencies and 2-qubit gate fidelities. We then need to optimize the device design for fabrication. The experimental data obtained by measuring the fabricated devices can be employed to refine the simulation, progressively enhancing the predictive power of simulation models. As a result, the trial-and-error iterations can be substantially accelerated through this synergy between simulations and experiments.

Secondly, most of the time, we are looking for robust features in simulations and experiments. For example, in the experimental spectra data in Fig. 6, the transition frequencies f_{01} can have small shifts due to two-level systems and other sources in the processors. However, for a working qubit, the overall shape of the spectra is close to the theoretical curve $f_{01}(\varphi_{\text{ext}})$, and therefore we can extract a good approximation of qubit parameters. Another example is the inaccuracy from control pulse distortions, which means that the qubits never experience the exact same control pulses as the simulation. However, we can use the simulation software to verify that for a gate scheme, the pulse parameters can be calibrated to compensate for pulse distortions. The average fidelities after calibration are thus a more robust quantity that can be extracted from simulation.

5.4 Realization of other superconducting qubit architectures

As demonstrated in the previous section, SuperGrad is capable of simulating and optimizing a multipath coupling fluxonium processor. Naturally, SuperGrad also supports transmon qubits. For example, we also reproduced schemes such as transmon qubits with tunable couplers [49] and fluxonium qubits mediated by transmon couplers (FTF) [2]. These codes could be found in Zenodo [46]. The extensible design of SuperGrad allows the integration of novel gate schemes, enabling users to implement their quantum processors accordingly.

One class of processors that are not currently supported by SuperGrad are qubits with tunable frequencies. More specifically, it is difficult to simulate the performance when idling and interaction frequencies are optimized directly in experiments (e.g., calibration done in [6]). One potential solution for these processors is simulating the optimization process in

numerical simulation.

5.5 Adding a Tensor Network Backend

Quantum many-body systems pose a numerical challenge due to the large Hilbert space, which grows exponentially with system size. Various numerical methods such as Matrix Product States (MPS) have been developed to navigate this challenge. Typically, high-rank state tensors can be represented as MPS, which consist of chains of rank-3 tensors truncated by a bond dimension. The time evolution simulations of quantum processors could be improved by employing tensor network algorithms (e.g., the algorithm in [50]). We should be careful that a large bond dimension may be required if we want to accurately represent a state with highly entangled qubits as a tensor network.

At times, it's necessary to compute correlated errors with the highest precision [17], and we wish to compute exact time evolution in this case. With careful design, many superconducting qubit systems can exhibit predominantly local couplings, meaning that simulations in a subset of quantum processors are sufficient to capture the correlated errors. Analyzing these errors is useful for optimizing both the fabrication and control of superconducting qubits, leading to improved quantum processor performance and reduced error rates.

However, for many common operations, the entanglement is very localized. Therefore, it is expected that tensor network algorithms can accurately approximate these kinds of time evolutions. We believe that efficiently simulating larger quantum processors has the potential to significantly advance the device design and control methods, pushing them to the next stage. Hence, we can gain deeper insights into the behavior of the scalable quantum processors.

Our Trotterization solver has already incorporated some tensor network features. It can transform the time evolution computation into a tensor network format and contract the tensor network through an optimized path. However, we have yet to implement tensor network-based time evolution algorithms in the SuperGrad, which we aim to incorporate in future releases.

6 Conclusion

As a versatile library, SuperGrad is specifically designed for the simulation and optimization of quantum processors. SuperGrad seamlessly integrates the quantum processor device and control optimization within a gradient-based framework. In this work, we have shown that the Trotterization solver serves as an efficient differentiable time evolution engine. With the help of SuperGrad's components, users can easily simulate their quantum processors, even with novel gate schemes. We want to emphasize that SuperGrad supports a wide range of gate schemes and is not limited to the specific showcases presented in this article.

We acknowledge that efficiently solving both forward and backward problems for time evolution in the many-body quantum system remains an open and challenging question. We are committed to continuing our exploration of viable solutions and improvements in this area. Our SuperGrad library is open source on GitHub [18]. The SuperGrad library is continually evolving to meet the escalating demands for simulating larger quantum processors, and future releases may introduce significant changes.

7 Acknowledgement

The authors would like to thank Wangwei Lan for constructive discussions. ZAW and XW acknowledge the partial support from the National Key Research and Development Program of China (2021YFA1401902). XTN acknowledges the partial support by Guangdong Provincial Quantum Science Strategic Initiative (GDZX2203001, GDZX2403001) and NSFC (No.92476206). FW and HHZ are supported by Zhongguancun Laboratory.

References

[1] F. Bao, H. Deng, D. Ding, R. Gao, X. Gao, C. Huang, X. Jiang, H.-S. Ku, Z. Li, X. Ma, X. Ni, J. Qin, Z. Song, H. Sun, C. Tang, T. Wang, F. Wu, T. Xia, W. Yu, F. Zhang, G. Zhang, X. Zhang, J. Zhou, X. Zhu, Y. Shi, J. Chen, H.-H. Zhao, and C. Deng, Phys. Rev. Lett. **129**, 010502 (2022).

[2] L. Ding, M. Hays, Y. Sung, B. Kannan, J. An, A. Di Paolo, A. H. Karamlou, T. M. Hazard, K. Azar, D. K. Kim, B. M. Niedzielski,

A. Melville, M. E. Schwartz, J. L. Yoder, T. P. Orlando, S. Gustavsson, J. A. Grover, K. Serniak, and W. D. Oliver, Phys. Rev. X **13**, 031035 (2023).

[3] Y. Sung, L. Ding, J. Braumüller, A. Vepsäläinen, B. Kannan, M. Kjaergaard, A. Greene, G. O. Samach, C. McNally, D. Kim, A. Melville, B. M. Niedzielski, M. E. Schwartz, J. L. Yoder, T. P. Orlando, S. Gustavsson, and W. D. Oliver, Phys. Rev. X **11**, 021058 (2021).

[4] J. Stehlik, D. M. Zajac, D. L. Underwood, T. Phung, J. Blair, S. Carnevale, D. Klaus, G. A. Keefe, A. Carniol, M. Kumph, M. Steffen, and O. E. Dial, Phys. Rev. Lett. **127**, 080505 (2021).

[5] H. Zhang, C. Ding, D. Weiss, Z. Huang, Y. Ma, C. Guinn, S. Sussman, S. P. Chitta, D. Chen, A. A. Houck, J. Koch, and D. I. Schuster, PRX Quantum **5**, 020326 (2024).

[6] R. Acharya, I. Aleiner, R. Allen, T. I. Andersen, M. Ansmann, F. Arute, K. Arya, A. Asfaw, J. Atalaya, R. Babbush, D. Bacon, J. C. Bardin, J. Basso, A. Bengtsson, S. Boixo, G. Bortoli, A. Bourassa, J. Bovaird, L. Brill, M. Broughton, B. B. Buckley, D. A. Buell, T. Burger, B. Burkett, N. Bushnell, Y. Chen, Z. Chen, B. Chiaro, J. Cogan, R. Collins, P. Conner, W. Courtney, A. L. Crook, B. Curtin, D. M. Debroy, A. Del Toro Barba, S. Demura, A. Dunsworth, D. Eppens, C. Erickson, L. Faoro, E. Farhi, R. Fatemi, L. Flores Burgos, E. Forati, A. G. Fowler, B. Foxen, W. Giang, C. Gidney, D. Gilboa, M. Giustina, A. Grajales Dau, J. A. Gross, S. Habegger, M. C. Hamilton, M. P. Harrigan, S. D. Harrington, O. Higgott, J. Hilton, M. Hoffmann, S. Hong, T. Huang, A. Huff, W. J. Huggins, L. B. Ioffe, S. V. Isakov, J. Iveland, E. Jeffrey, Z. Jiang, C. Jones, P. Juhas, D. Kafri, K. Kechedzhi, J. Kelly, T. Khattar, M. Khezri, M. Kieferová, S. Kim, A. Kitaev, P. V. Klimov, A. R. Klots, A. N. Korotkov, F. Kostritsa, J. M. Kreikebaum, D. Landhuis, P. Laptev, K.-M. Lau, L. Laws, J. Lee, K. Lee, B. J. Lester, A. Lill, W. Liu, A. Locharla, E. Lucero, F. D. Malone, J. Marshall, O. Martin, J. R. McClean, T. McCourt, M. McEwen, A. Megrant, B. Meurer Costa, X. Mi, K. C. Miao, M. Mohseni, S. Montazeri, A. Morvan, E. Mount, W. Mruczkiewicz, O. Naaman, M. Neeley, C. Neill, A. Nersisyan, H. Neven, M. Newman, J. H. Ng, A. Nguyen, M. Nguyen, M. Y. Niu, T. E. O'Brien, A. Opremcak, J. Platt, A. Petukhov, R. Potter, L. P. Pryadko, C. Quintana, P. Roushan, N. C. Rubin, N. Saei, D. Sank, K. Sankaragomathi, K. J. Satzinger, H. F. Schurkus, C. Schuster, M. J. Shearn, A. Shorter, V. Shvarts, J. Skruzny, V. Smelyanskiy, W. C. Smith, G. Sterling, D. Strain, M. Szalay, A. Torres, G. Vidal, B. Villalonga, C. Vollgraff Heidweiller, T. White, C. Xing, Z. J. Yao, P. Yeh, J. Yoo, G. Young, A. Zalcman, Y. Zhang, N. Zhu, and Google Quantum AI, Nature **614**, 676 (2023).

[7] Y. Zhao, Y. Ye, H.-L. Huang, Y. Zhang, D. Wu, H. Guan, Q. Zhu, Z. Wei, T. He, S. Cao, F. Chen, T.-H. Chung, H. Deng, D. Fan, M. Gong, C. Guo, S. Guo, L. Han, N. Li, S. Li, Y. Li, F. Liang, J. Lin, H. Qian, H. Rong, H. Su, L. Sun, S. Wang, Y. Wu, Y. Xu, C. Ying, J. Yu, C. Zha, K. Zhang, Y.-H. Huo, C.-Y. Lu, C.-Z. Peng, X. Zhu, and J.-W. Pan, Phys. Rev. Lett. **129**, 030501 (2022).

[8] S. Krinner, N. Lacroix, A. Remm, A. Di Paolo, E. Genois, C. Leroux, C. Hellings, S. Lazar, F. Swiadek, J. Herrmann, G. J. Norris, C. K. Andersen, M. Müller, A. Blais, C. Eichler, and A. Wallraff, Nature **605**, 669 (2022).

[9] I. Georgescu, S. Ashhab, and F. Nori, Reviews of Modern Physics **86**, 153 (2014).

[10] Q. Zhu, Z.-H. Sun, M. Gong, F. Chen, Y.-R. Zhang, Y. Wu, Y. Ye, C. Zha, S. Li, S. Guo, H. Qian, H.-L. Huang, J. Yu, H. Deng, H. Rong, J. Lin, Y. Xu, L. Sun, C. Guo, N. Li, F. Liang, C.-Z. Peng, H. Fan, X. Zhu, and J.-W. Pan, Phys. Rev. Lett. **128**, 160502 (2022).

[11] Google Quantum AI and Collaborators, T. I. Andersen, Y. D. Lensky, K. Kechedzhi, I. K. Drozdov, A. Bengtsson, S. Hong, A. Morvan, X. Mi, A. Opremcak, R. Acharya, R. Allen, M. Ansmann, F. Arute, K. Arya, A. Asfaw, J. Atalaya, R. Babbush, D. Bacon, J. C. Bardin, G. Bortoli, A. Bourassa, J. Bovaird, L. Brill, M. Broughton, B. B. Buckley, D. A. Buell, T. Burger, B. Burkett, N. Bushnell, Z. Chen, B. Chiaro, D. Chik, C. Chou, J. Cogan, R. Collins, P. Conner, W. Courtney, A. L. Crook, B. Curtin, D. M. Debroy, A. Del Toro Barba, S. Demura, A. Dunsworth, D. Eppens, C. Erickson, L. Faoro, E. Farhi, R. Fatemi, V. S. Ferreira, L. F. Burgos, E. Forati, A. G. Fowler, B. Foxen, W. Giang, C. Gidney, D. Gilboa,

M. Giustina, R. Gosula, A. G. Dau, J. A. Gross, S. Habegger, M. C. Hamilton, M. Hansen, M. P. Harrigan, S. D. Harrington, P. Heu, J. Hilton, M. R. Hoffmann, T. Huang, A. Huff, W. J. Huggins, L. B. Ioffe, S. V. Isakov, J. Iveland, E. Jeffrey, Z. Jiang, C. Jones, P. Juhas, D. Kafri, T. Khattar, M. Khezri, M. Kieferová, S. Kim, A. Kitaev, P. V. Klimov, A. R. Klots, A. N. Korotkov, F. Kostritsa, J. M. Kreikebaum, D. Landhuis, P. Laptev, K.-M. Lau, L. Laws, J. Lee, K. W. Lee, B. J. Lester, A. T. Lill, W. Liu, A. Locharla, E. Lucero, F. D. Malone, O. Martin, J. R. McClean, T. McCourt, M. McEwen, K. C. Miao, A. Mieszala, M. Mohseni, S. Montazeri, E. Mount, R. Movassagh, W. Mruczkiewicz, O. Naaman, M. Neeley, C. Neill, A. Nersisyan, M. Newman, J. H. Ng, A. Nguyen, M. Nguyen, M. Y. Niu, T. E. O'Brien, S. Omonije, A. Petukhov, R. Potter, L. P. Pryadko, C. Quintana, C. Rocque, N. C. Rubin, N. Saei, D. Sank, K. Sankaragomathi, K. J. Satzinger, H. F. Schurkus, C. Schuster, M. J. Shearn, A. Shorter, N. Shutty, V. Shvarts, J. Skruzny, W. C. Smith, R. Somma, G. Sterling, D. Strain, M. Szalay, A. Torres, G. Vidal, B. Villalonga, C. V. Heidweiller, T. White, B. W. K. Woo, C. Xing, Z. J. Yao, P. Yeh, J. Yoo, G. Young, A. Zalcman, Y. Zhang, N. Zhu, N. Zobrist, H. Neven, S. Boixo, A. Megrant, J. Kelly, Y. Chen, V. Smelyanskiy, E.-A. Kim, I. Aleiner, and P. Roushan, Nature **618**, 264 (2023).

[12] Y.-H. Shi, R.-Q. Yang, Z. Xiang, Z.-Y. Ge, H. Li, Y.-Y. Wang, K. Huang, Y. Tian, X. Song, D. Zheng, K. Xu, R.-G. Cai, and H. Fan, Nature Communications **14**, 3263 (2023).

[13] Z.-C. Xiang, K. Huang, Y.-R. Zhang, T. Liu, Y.-H. Shi, C.-L. Deng, T. Liu, H. Li, G.-H. Liang, Z.-Y. Mei, H. Yu, G. Xue, Y. Tian, X. Song, Z.-B. Liu, K. Xu, D. Zheng, F. Nori, and H. Fan, Nature Communications **14**, 5433 (2023).

[14] S. Xu, Z.-Z. Sun, K. Wang, L. Xiang, Z. Bao, Z. Zhu, F. Shen, Z. Song, P. Zhang, W. Ren, X. Zhang, H. Dong, J. Deng, J. Chen, Y. Wu, Z. Tan, Y. Gao, F. Jin, X. Zhu, C. Zhang, N. Wang, Y. Zou, J. Zhong, A. Zhang, W. Li, W. Jiang, L.-W. Yu, Y. Yao, Z. Wang, H. Li, Q. Guo, C. Song, H. Wang, and D.-L. Deng, Chinese Physics Letters **40**, 060301 (2023).

[15] D. Ristè, M. P. Da Silva, C. A. Ryan, A. W. Cross, A. D. Córcoles, J. A. Smolin, J. M. Gambetta, J. M. Chow, and B. R. Johnson, npj Quantum Inf. **3**, 16 (2017).

[16] M. Gong, S. Wang, C. Zha, M.-C. Chen, H.-L. Huang, Y. Wu, Q. Zhu, Y. Zhao, S. Li, S. Guo, H. Qian, Y. Ye, F. Chen, C. Ying, J. Yu, D. Fan, D. Wu, H. Su, H. Deng, H. Rong, K. Zhang, S. Cao, J. Lin, Y. Xu, L. Sun, C. Guo, N. Li, F. Liang, V. M. Bastidas, K. Nemoto, W. J. Munro, Y.-H. Huo, C.-Y. Lu, C.-Z. Peng, X. Zhu, and J.-W. Pan, Science **372**, 948 (2021).

[17] X. Ni, Z. Wang, R. Chao, and J. Chen, (2024), arXiv:2312.04186 [quant-ph] .

[18] S. contributors, "SuperGrad: Differentiable simulator for superconducting quantum processors," (2024).

[19] X. Ni, H.-H. Zhao, L. Wang, F. Wu, and J. Chen, npj Quantum Inf. **8**, 106 (2022).

[20] P. Groszkowski and J. Koch, Quantum **5**, 583 (2021).

[21] D. Puzzuoli, C. J. Wood, D. J. Egger, B. Rosand, and K. Ueda, Journal of Open Source Software **8**, 5853 (2023).

[22] F.-M. L. Régent, C. Berdou, Z. Leghtas, J. Guillaud, and M. Mirrahimi, Quantum **7**, 1198 (2023).

[23] C. Chamberland, K. Noh, P. Arrangoiz-Arriola, E. T. Campbell, C. T. Hann, J. Iverson, H. Putterman, T. C. Bohdanowicz, S. T. Flammia, A. Keller, G. Refael, J. Preskill, L. Jiang, A. H. Safavi-Naeini, O. Painter, and F. G. Brandão, PRX Quantum **3**, 010329 (2022).

[24] T. Zhao, D. Chen, T. Lyu, and J. Koch, "Qfit: Interactive parameter fitting for superconducting circuits," (2024).

[25] S. Krastanov, S. Zhou, S. T. Flammia, and L. Jiang, Quantum Science and Technology **4**, 035003 (2019).

[26] J. Bradbury, R. Frostig, P. Hawkins, M. J. Johnson, C. Leary, D. Maclaurin, G. Necula, A. Paszke, J. VanderPlas, S. Wanderman-Milne, and Q. Zhang, "JAX: composable transformations of Python+NumPy programs," (2018).

[27] T. Hennigan, T. Cai, T. Norman, L. Martens, and I. Babuschkin, "Haiku: Sonnet for JAX," (2020).

[28] H. F. Trotter, Proceedings of the American Mathematical Society **10**, 545 (1959).

[29] M. Suzuki, Communications in Mathematical Physics **51**, 183 (1976).
[30] M. Suzuki, Physics Letters A **146**, 319 (1990).
[31] L. S. Pontryagin, Trudy Mat. Inst. Steklov. **169**, 119 (1985).
[32] R. T. Q. Chen, Y. Rubanova, J. Bettencourt, and D. K. Duvenaud, in *Advances in Neural Information Processing Systems*, Vol. 31 (Curran Associates, Inc., 2018).
[33] M. Blondel and V. Roulet, (2024), arXiv:2403.14606 [cs.LG] .
[34] S. Efthymiou, S. Ramos-Calderer, C. Bravo-Prieto, A. Pérez-Salinas, D. García-Martín, A. Garcia-Saez, J. I. Latorre, and S. Carrazza, Quantum Science and Technology **7**, 015018 (2022).
[35] J. Johansson, P. Nation, and F. Nori, Computer Physics Communications **183**, 1760 (2012).
[36] W. Baur and V. Strassen, Theor. Comput. Sci. **22**, 317 (1983).
[37] A. Wiltschko and M. J. Johnson, "The Autodiff Cookbook," (2018).
[38] L. B. Nguyen, G. Koolstra, Y. Kim, A. Morvan, T. Chistolini, S. Singh, K. N. Nesterov, C. Jünger, L. Chen, Z. Pedramrazi, B. K. Mitchell, J. M. Kreikebaum, S. Puri, D. I. Santiago, and I. Siddiqi, PRX Quantum **3**, 037001 (2022).
[39] E. Dogan, D. Rosenstock, L. Le Guevel, H. Xiong, R. A. Mencia, A. Somoroff, K. N. Nesterov, M. G. Vavilov, V. E. Manucharyan, and C. Wang, Phys. Rev. Applied **20**, 024011 (2023).
[40] W.-J. Lin, H. Cho, Y. Chen, M. G. Vavilov, C. Wang, and V. E. Manucharyan, PRX Quantum **6**, 010349 (2025).
[41] A. A. Hagberg, D. A. Schult, and P. J. Swart, in *Proceedings of the 7th Python in Science Conference* (Pasadena, CA USA, 2008) pp. 11 – 15.
[42] C. Berke, E. Varvelis, S. Trebst, A. Altland, and D. P. DiVincenzo, Nature Communications **13**, 2495 (2022).
[43] R. H. Byrd, P. Lu, J. Nocedal, and C. Zhu, SIAM Journal on Scientific Computing **16**, 1190 (1995).
[44] W. Lei, Private communication (2023).
[45] N. Leung, M. Abdelhafez, J. Koch, and D. Schuster, Phys. Rev. A **95**, 042318 (2017).
[46] Z. Wang and X. Ni, Zenodo (2024).
[47] D. Puzzuoli, S. F. Lin, M. Malekakhlagh, E. Pritchett, B. Rosand, and C. J. Wood, Journal of Computational Physics **489**, 112262 (2023).
[48] J. Heek, A. Levskaya, A. Oliver, M. Ritter, B. Rondepierre, A. Steiner, and M. van Zee, "Flax: A neural network library and ecosystem for JAX," (2023).
[49] Y. Xu, J. Chu, J. Yuan, J. Qiu, Y. Zhou, L. Zhang, X. Tan, Y. Yu, S. Liu, J. Li, F. Yan, and D. Yu, Phys. Rev. Lett. **125**, 240503 (2020).
[50] G. Vidal, Phys. Rev. Lett. **98**, 070201 (2007).
[51] D. G. a. Smith and J. Gray, Journal of Open Source Software **3**, 753 (2018).
[52] J. Gray and S. Kourtis, Quantum **5**, 410 (2021).
[53] B. O'Gorman, in *14th Conference on the Theory of Quantum Computation, Communication and Cryptography (TQC 2019)*, Vol. 135 (2019) pp. 10:1–10:19.
[54] S. Kourtis, C. Chamon, E. Mucciolo, and A. Ruckenstein, SciPost Physics **7**, 060 (2019).
[55] D. C. McKay, C. J. Wood, S. Sheldon, J. M. Chow, and J. M. Gambetta, Phys. Rev. A **96**, 022330 (2017).
[56] C. Rigetti and M. Devoret, Phys. Rev. B **81**, 134507 (2010).
[57] P. C. de Groot, J. Lisenfeld, R. N. Schouten, S. Ashhab, A. Lupaşcu, C. J. P. M. Harmans, and J. E. Mooij, Nature Physics **6**, 763 (2010).
[58] J. C. Pommerening and D. P. DiVincenzo, Phys. Rev. A **102**, 032623 (2020).
[59] M. A. Nielsen, Physics Letters A **303**, 249 (2002).
[60] T. F. Havel, Journal of Mathematical Physics **44**, 534 (2003).

A Higher-order Suzuki-Trotter decomposition

Consider a Hamiltonian represented as $\mathcal{H} = \sum_{k=1}^{N_H} H_k$, where each term H_k denotes a local Hamiltonian that acts non-trivially on the subsystem k, N_H is the number of local Hamiltonian terms. The first-order decomposition employs the product of local time evolution operators $e^{-iH_k\delta t}$, which can be expressed as:

$$Q_1(\delta t) = \prod_{k=1}^{N_H} e^{-iH_k\delta t}, \tag{21}$$

where each k corresponds to a specific subset of the quantum system.

The second-order decomposition, derived from a symmetric (palindromic) sequence of $Q_1(\delta t)$ is given by

$$Q_2(\delta t) = \prod_{k=1}^{N_H} e^{-iH_k\frac{\delta t}{2}} \prod_{k=N_H}^{1} e^{-iH_k\frac{\delta t}{2}}. \tag{22}$$

The complex fourth-order Suzuki-Trotter decomposition, which satisfies Eqn. 23 with $p = (3 - \sqrt{3}i/6)$.

$$Q_{4j}(\delta t) = \prod_{k=1}^{N_H} e^{-ipH_k\frac{\delta t}{2}} \prod_{k=N_H}^{1} e^{-ipH_k\frac{\delta t}{2}} \prod_{k=1}^{N_H} e^{-ip^*H_k\frac{\delta t}{2}} \prod_{k=N_H}^{1} e^{-ip^*H_k\frac{\delta t}{2}}, \tag{23}$$

where the star operator $*$ indicates the complex conjugate. The real fourth-order Suzuki-Trotter decomposition, which includes more terms compared to the complex version, is formulated as Eqn. 24 with $p = 1/(2 - \sqrt[3]{2})$.

$$Q_4(\delta t) = \prod_{k=1}^{N_H} e^{-ipH_k\frac{\delta t}{2}} \prod_{k=N_H}^{1} e^{-ipH_k\frac{\delta t}{2}} \prod_{k=1}^{N_H} e^{-i(1-2p)H_k\frac{\delta t}{2}} \prod_{k=N_H}^{1} e^{-i(1-2p)H_k\frac{\delta t}{2}} \prod_{k=1}^{N_H} e^{-ipH_k\frac{\delta t}{2}} \prod_{k=N_H}^{1} e^{-ipH_k\frac{\delta t}{2}} \tag{24}$$

These decompositions offer varying levels of accuracy and computational efficiency, catering to the precision requirements and the specific property of the quantum system being modeled.

B Tensor Network Contraction

We represent the quantum state of the composite Hilbert space as a rank-N tensor, denoted by $\psi_{1,2,3,\ldots,N}$. To address the dynamics of this state, we consider the product of the time evolution operator and the tensor state. Specifically, we focus on the contraction between the first-order STD decomposition $Q_1(\delta t)$ and $\psi_{1,2,3,\ldots,N}$ in the context of time evolution. In the simplest case that only involves a single two-body local time evolution operator, we define the operator as $u_{ij,i'j'} = \exp\{-iH_{ij,i'j'}\delta t\}$, where δt represents a small time interval. The resulting tensor contraction can be expressed as

$$\psi'_{1,2,3,\cdots,i',j',\cdots,N} = \sum_{i,j} u_{ij,i'j'}\psi_{1,2,3,\cdots,i,j,\cdots,N}. \tag{25}$$

This computation is performed using tensor network contraction along an optimized path, performed by efficient packages [51, 52]. In this section, we specifically consider the first-order Suzuki-Trotter decomposition to illustrate the integration of the Suzuki-Trotter decomposition with the tensor network contraction. It is important to note that the computation methodology for higher-order Suzuki-Trotter decompositions is analogous.

Assuming a quantum processor with N qudits and each qudit subsystem having a dimension d, the computational cost of tensor contraction is determined by the size of the k-body evolution operator $dim(U_X) = d^{2k}$ and the dimension of the tensor state $dim(\psi) = d^N$. This cost is represented as $\frac{|dim(\psi)||dim(U_X)|}{|dim(\psi)\cap dim(U_X)|}$, where X

labels subsets of the local quantum system. The time complexity for evolving the quantum state with a k-body evolution operator is $O(d^{N+k})$. This is lower than the time complexity of matrix-vector multiplication in the Hilbert space, which is $O(d^{2N})$. Thus, using tensor contraction can significantly enhance computational efficiency, especially when leveraging GPU with cuTENSOR or hipTensor.

After introducing the Suzuki-Trotter decomposition, the time evolution over a given interval can be represented as a tensor network contraction problem. This requires identifying an efficient contraction path to minimize the size of intermediate tensors and the computational cost, measured in floating-point operations per second (FLOPS). Optimally determining such a path is an NP-hard problem [53]. Nonetheless, the Greedy algorithm provides an effective method for finding contraction paths, especially when dealing with a multitude of low-rank tensors [52, 54]. The algorithm follows these principal steps:

1. Compute Hadamard products of tensors in an arbitrary order.
2. Contract pairs of remaining tensors, specifically choosing those pairs that minimize the resultant tensor's size and computational cost.
3. Compute any necessary pairwise outer products.

Once established, this contraction path can be repeatedly used for both the evolution of quantum states and their adjoint states, thereby justifying a higher initial computational investment to determine the most optimal path possible.

C Simultaneous Gate Simulation Details

A commonly implemented single-qubit gate is the Rabi π rotation. To execute a single-qubit gate, a drive pulse is applied through the diplexed flux control line of the qubit i. The associated driving Hamiltonian is given by

$$H_\epsilon(t) = \mathcal{E}(t)\cos(\omega_d t + \phi)\hat{\varphi}_i, \tag{26}$$

where the pulse envelope $\mathcal{E}(t)$ is specifically shaped as

$$\mathcal{E}(t) = \frac{\epsilon_d}{2}\left[1 - \cos\left(\frac{2\pi t}{\tau_{\text{gate}}}\right)\right]. \tag{27}$$

Considering the presence of higher energy levels and adjacent qubits, the optimal frequency ω_d is derived by analyzing the composited quantum system. Furthermore, arbitrary Z-rotations are incorporated both before and after the gate operations. We also allow arbitrary Z-rotations before and after the gates, since they can be essentially free [55].

For two-qubit operations, we employ the Cross-Resonance (CR) pulse [56, 57]. There is one of the two-qubit gate schemes discussed in [38]. The CR pulse requires the application of a microwave tone to the control qubit, the microwave tuned approximately to the frequency ω_{10} of the target qubit. The drive employs the same form as the Hamiltonian defined in Eqn. 26, with the envelope $\mathcal{E}(t)$ designed to have a ramp time τ_{ramp} equal to half the gate duration $\tau_{\text{gate}}/2$.

By tuning ω_d to approximate the $0-1$ transition frequency of the neighboring qubit, the resultant two-qubit gate U_{CR} can effectively emulate a CNOT gate, with the addition of some single-qubit unitary transformations. To simplify the simulation of U_{CNOT} based on U_{CR}, we incorporate ideal single-qubit rotations before and after the driving of simultaneous pulses.

D Details of time evolution computation

The commonly used basis for multi-qubit processors is the product basis (the tensor product of single-qubit eigenbasis) or the multi-qubit eigenbasis. This latter basis is derived by diagonalizing the idling Hamiltonian matrix in the product basis and then adjusting the phases of its eigenstates [17]. The product basis offers the

advantage of storing the local Hamiltonian in the qubit subspace, facilitating efficient time evolution through the use of a series of 2-body operators. However, the local Hamiltonian in the multi-qubit eigenbasis tends to occupy the full Hilbert space, which significantly degrades the performance during time evolution.

Computing in a different basis is almost equivalent, but some information will be lost when we truncate the matrices to the computational subspace. The multi-qubit eigenbasis is considered to be a good approach for the experimental basis [58]. We suggest a method that utilizes the multi-qubit eigenbasis to represent the final state while employing the product basis for time evolution calculations. The primary computational cost here involves the diagonalization of the idling Hamiltonian. The time complexity for diagonalization is $O(d^{3N})$, where N represents the number of qudits and d denotes the dimension of the qudit subsystem. This complexity becomes prohibitive for large quantum processors. Therefore, finding an efficient method to approximate the eigenstates of the idling Hamiltonian remains a question for future study. Alternatively, employing the product basis to represent the final state may reduce the diagonalization overhead but increase the error.

We begin the process by diagonalizing the idling Hamiltonian and using the eigenbasis for the time evolution unitary matrices. We denote the eigenstates by the matrix V, which is written in the product basis B_{product}. In this work, the Hamiltonians are always written in the product basis since we want to use Trotter decomposition. Therefore, the time evolution unitary in the eigenbasis is

$$U(t_0, t_g) \mapsto V^\dagger \left(\mathcal{T} \exp \left\{ -i \int_{t_0}^{t_g} \mathcal{H}(t) dt \right\} \right) V. \tag{28}$$

We can absorb $V^\dagger$ and V into the quantum state by evolving the initial state according to $|\psi'_{\text{initial}}\rangle \mapsto V|\psi_{\text{initial}}\rangle$ in the product basis. Subsequently, we transform the final state back to the eigenbasis via $|\psi_{\text{final}}\rangle \mapsto V^\dagger|\psi'_{\text{final}}\rangle$. The method for transforming the basis and calculating the objective function comprises the following steps:

1. For each idling qubit, compute its single-qubit eigenbasis and represent the subsystem Hamiltonian, $\hat{n}$, and $\hat{\varphi}$ operators within this eigenbasis. This method achieves faster convergence with an increasing number of subsystem levels due to the more information captured in the low-energy spectrum.
2. Determine the eigenstate of the idling system's Hamiltonian.
3. To model the unitary for simultaneous driving, simulate the evolution of all possible initial states in the computational basis in parallel.
4. Employ a Trotterization solver to compute the time evolution, using a time step size of $\delta t = 0.05$ ns. We used a fourth-order complex Suzuki-Trotter decomposition to simulate simultaneous two-qubit gates.
5. Truncate the final state to the computational basis and assemble the time evolution operator U_{sim}. We assume that the leakage error is negligible, which allows the unitary to be written in B_{eigen} as U_{sim}.
6. Apply SU(4) unitary compensation to U_{sim} by incorporating errorless single-qubit gates before and after the simultaneous pulses, thus aligning U_{sim} as closely as possible with the target unitary U_{target}.
7. Evaluate the fidelity metric and use it as the objective function for optimization. The formula is detailed in Eqn. 29.
8. Optimize using the L-BFGS-B method, leveraging backpropagation to compute the gradient.

The objective function is defined as $\mathcal{L} = 1 - \mathcal{F}$, and the formula for the fidelity metric [59] satisfies

$$\mathcal{F}(\mathcal{E}, I) = \frac{\sum_j \text{tr}(U_j^\dagger \mathcal{E}(U_j)) + D^2}{D^2(D+1)}, \tag{29}$$

where D is the dimension of the system and the unitary operators U_j form an orthonormal operator basis.

It's valuable to note that the dynamics of a density matrix could be characterized using a superoperator, which is computed by solving the vectorized Lindblad master equation [60]. In this way, SuperGrad migrated the Trotterization solver and backpropagation method to the open quantum systems.

E Miscellaneous Data

Device parameters				Control parameters			
Parameter	Value (GHz)	Gradient [TAD]	Relative Error [LCAM]	Parameter	Value	Gradient [TAD]	Relative Error [LCAM]
$E_{C,0}$	9.945e-01	-5.253e-04	5.498e-11	ϵ_d (Q_0)	8.995e-03 GHz	-2.443e-04	-6.275e-12
$E_{J,0}$	3.978e+00	1.636e-04	5.939e-11	t_{single} (Q_0)	5.000e+01 ns	-1.138e-03	-1.755e-08
$E_{L,0}$	8.950e-01	-4.703e-04	9.996e-11	$\omega_d/2\pi$ (Q_0)	4.859e-01 GHz	-3.821e-02	-9.866e-13
$E_{C,1}$	9.967e-01	3.492e-04	-2.101e-11	ϵ_d (Q_1)	9.433e-03 GHz	-7.167e-04	-1.786e-12
$E_{J,1}$	3.987e+00	-1.106e-04	-2.148e-11	t_{single} (Q_1)	5.000e+01 ns	-1.121e-03	-1.781e-08
$E_{L,1}$	9.967e-01	3.064e-04	-8.477e-11	$\omega_d/2\pi$ (Q_1)	5.913e-01 GHz	1.019e-02	-1.503e-12
$E_{C,2}$	1.005e+00	1.929e-03	-6.928e-12	ϵ_d (Q_2)	9.745e-03 GHz	-1.840e-04	-9.815e-12
$E_{J,2}$	4.021e+00	-6.279e-04	-6.326e-12	t_{single} (Q_2)	5.000e+01 ns	-1.125e-03	-1.775e-08
$E_{L,2}$	1.106e+00	1.654e-03	-1.781e-11	$\omega_d/2\pi$ (Q_2)	6.710e-01 GHz	6.646e-03	7.352e-13
$E_{C,3}$	1.014e+00	-1.672e-03	4.218e-13	ϵ_d (Q_3)	9.119e-03 GHz	-1.113e-03	1.287e-11
$E_{J,3}$	4.056e+00	5.166e-04	1.422e-12	t_{single} (Q_3)	5.000e+01 ns	-1.091e-03	-1.830e-08
$E_{L,3}$	9.126e-01	-1.521e-03	1.420e-11	$\omega_d/2\pi$ (Q_3)	5.348e-01 GHz	4.803e-02	-3.982e-13
$E_{C,4}$	9.863e-01	1.089e-03	-1.270e-11	ϵ_d (Q_4)	9.446e-03 GHz	-1.628e-03	9.624e-12
$E_{J,4}$	3.945e+00	-3.492e-04	-1.272e-11	t_{single} (Q_4)	5.000e+01 ns	-1.115e-03	-1.790e-08
$E_{L,4}$	9.863e-01	9.692e-04	-3.276e-11	$\omega_d/2\pi$ (Q_4)	5.843e-01 GHz	-6.521e-02	2.924e-13
$E_{C,5}$	1.010e+00	1.786e-03	-9.624e-12	ϵ_d (Q_5)	9.694e-03 GHz	-2.304e-03	4.820e-12
$E_{J,5}$	4.041e+00	-5.796e-04	-8.902e-12	t_{single} (Q_5)	5.000e+01 ns	-1.036e-03	-1.927e-08
$E_{L,5}$	1.111e+00	1.519e-03	-2.144e-11	$\omega_d/2\pi$ (Q_5)	6.637e-01 GHz	-8.102e-02	4.253e-14
$E_{C,6}$	1.001e+00	-8.966e-04	-3.022e-12	ϵ_d (Q_6)	9.007e-03 GHz	-4.521e-04	2.623e-11
$E_{J,6}$	4.003e+00	2.779e-04	-5.447e-13	t_{single} (Q_6)	5.000e+01 ns	-1.059e-03	-1.885e-08
$E_{L,6}$	9.006e-01	-8.016e-04	2.326e-11	$\omega_d/2\pi$ (Q_6)	4.958e-01 GHz	6.750e-02	-5.386e-13
$E_{C,7}$	1.010e+00	1.071e-03	4.939e-12	ϵ_d (Q_7)	9.396e-03 GHz	-5.403e-04	-6.577e-12
$E_{J,7}$	4.041e+00	-3.382e-04	4.827e-12	t_{single} (Q_7)	5.000e+01 ns	-1.153e-03	-1.731e-08
$E_{L,7}$	1.010e+00	9.346e-04	-1.536e-11	$\omega_d/2\pi$ (Q_7)	5.947e-01 GHz	4.621e-02	-7.426e-13
$J_{C,(0,1)}$	2.000e-02	-1.321e-05	6.382e-12				
$J_{L,(0,1)}$	-2.000e-03	-2.940e-03	5.496e-12				
$J_{C,(1,2)}$	2.000e-02	-3.149e-05	-1.993e-12				
$J_{L,(1,2)}$	-2.000e-03	-5.407e-03	-1.771e-12				
$J_{C,(2,3)}$	2.000e-02	-2.055e-05	-2.378e-12				
$J_{L,(2,3)}$	-2.000e-03	-3.753e-03	-1.977e-12				
$J_{C,(3,4)}$	2.000e-02	-2.748e-05	2.024e-12				
$J_{L,(3,4)}$	-2.000e-03	-5.455e-03	2.385e-12				
$J_{C,(4,5)}$	2.000e-02	-5.965e-05	-1.502e-12				
$J_{L,(4,5)}$	-2.000e-03	-1.019e-02	-1.508e-12				
$J_{C,(5,6)}$	2.000e-02	3.242e-06	1.378e-11				
$J_{L,(5,6)}$	-2.000e-03	1.521e-04	4.129e-11				
$J_{C,(6,7)}$	2.000e-02	-3.169e-05	-8.270e-13				
$J_{L,(6,7)}$	-2.000e-03	-7.191e-03	5.958e-14				

Table 5: The table contains device and control parameters for an 8-qubit chain conducted in Fig. 7. The gradients listed are $\nabla\mathcal{L}(\vec{\theta}_{\text{ham}})$ calculated through the auto-diff of Trotterization (TAD), where $\mathcal{L}$ is the objective function containing simultaneous $X^{\otimes 8}$ gate fidelity defined in Eqn. 29. The units of gradients are the reciprocal of the units of their corresponding values. The relative errors listed are determined using gradients computed by the local continuous adjoint method (LCAM) for each parameter component.