YQuantum 2025 QuEra WriteUp

John Zhuoyang Ye[1] **Mu Niu**[1] **Victor Yu**[1] **Harry Wang**[2] **Hanyu Wang**[1] **Team QuBruin**

[1] *University of California, Los Angeles*

[2] *Brown University*

E-mail: lucaniu2003@g.ucla.edu, yezhuoyang@cs.ucla.edu, haochen_wang@brown.edu

Abstract:

In this challenge, we explore two different area to demonstrate the power of new language feature, as well as software-hardware codesign. First, we implement error correction code such as repetition code and surface code. With the help of **modularized** design and function reusing, we show that we can also implement simple logical algorithm with surface code. We also use **recursion** to implement **fault-tolerant** syndrome measurement. Second, we implement a second-order Trotterized time evolution simulation of the transverse-field Ising model (TFIM)—a paradigmatic strongly correlated system exhibiting two distinct phases determined by the competition between spin-spin interactions and external transverse fields.By adiabatically tuning the Hamiltonian parameters, our simulations successfully detect the crossing of the critical point separating these phases. Furthermore, we optimize the quantum circuit to minimize the number of required pulses, demonstrating an efficient quantum simulation protocol for the TFIM. We have also put a lot of effort in figuring out the compilation automation pipeline, such that all of our algorithm can be compiled down to atom movements, we also use video to demonstrate our compilation. In the contest, we find some potential bugs in the provided packages.

Contents

1 Introduction

This is the writeup report for the YQuantum 2025 QuEra challenge.

Strongly correlated systems host some of the most exotic and intriguing states of matter, including the fractional quantum Hall state, quantum spin liquids, and various quasiparticle excitations. Accurately simulating such systems—especially their time evolution—remains a major challenge in modern physics. Classical methods such as tensor networks have demonstrated impressive results for ground state simulations in one dimension, but they continue to face limitations in higher dimensions and in capturing real-time dynamics.

Recently, controlled quantum systems, particularly neutral atom arrays, have shown great promise for simulating strongly correlated materials. For example, researchers have successfully used a 219-atom programmable quantum simulator to realize a topological spin liquid—highlighting the platform's growing capabilities [1].

QuEra, a company developing neutral atom quantum hardware, has recently released two software packages: Bloqade and Kirin. These tools not only enable the implementation of classical logic operations such as regression and logical gates, but also support parallel circuit processing, offering a more seamless connection between user-defined algorithms and the actual hardware operations [2].

In this study, we demonstrate how to use these tools to simulate the Trotterized time evolution of the transverse-field Ising model (TFIM) with time-dependent parameters. In the first part of the report, we present results from a slow adiabatic sweep of the Hamiltonian across its critical point, capturing the phase transition. In the second part, we describe how the circuit was optimized to minimize the number of required pulses while maintaining high fidelity in the simulation outcome.

2 Implement surface code and fault-tolerant measurement with new Kirin syntax features

QuEra has recently developed a new compilation tool-chain Kirin, and a new version of Bloqade. Kirin support quantum circuit development with classical programming language features such as *if while* statement, *for loop*. It is even possible to implement recursion. Kirin can generate a lower level assembly language, which extends QASM2.0, and is tailored for neutral atom compilation.

We will show how to implement arbitrary surface code with all logical Clifford operations with the help of the new language and software tool. We will also explain how to use *modularization* to organize the code design and apply *recursion* for fault-tolerant stabilizer measurement.

First, we implement the repetition code with *if* statement.

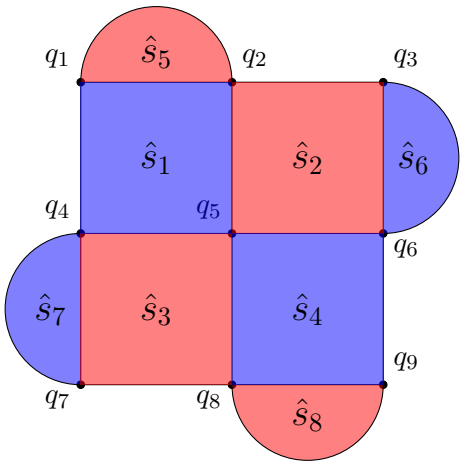
Listing 1. Repetition code

```
@qasm2.extended
def repetition_code():

    qreg = qasm2.qreg(5)
    creg = qasm2.creg(5)
    '''
    Srror propagation and syndrome extraction
    '''
    qasm2.cx(qreg[0], qreg[3])
    qasm2.cx(qreg[1], qreg[3])
    qasm2.cx(qreg[1], qreg[4])
    qasm2.cx(qreg[2], qreg[4])

    qasm2.measure(qreg[0], creg[0])
    qasm2.measure(qreg[1], creg[1])
    qasm2.measure(qreg[2], creg[2])
    qasm2.measure(qreg[3], creg[3])
    qasm2.measure(qreg[4], creg[4])
    '''
    Decoding and error correction
    '''
    if creg[3] == 0 and creg[4] == 1:
        qasm2.x(qreg[2])
    if creg[3] == 1 and creg[4] == 1:
        qasm2.x(qreg[1])
    if creg[3] == 1 and creg[4] == 0:
        qasm2.x(qreg[0])

    return creg
```



To implement the Surfacecode, we first calculate the number of qubit and the coordinates of all XZ stabilizers. For each stabilizer with fixed index,type and qubits, we directly

use the function *add_X_syndrome_circuit* and *add_Z_syndrome_circuit* functions, redundent codes are prevented by this design.

For example, *add_X_syndrome_circuit (qreg , creg ,9 ,0 ,(0 ,1 ,3 ,4))* means add the 0th stabilizer $X_1X_2X_4X_5$ to our circuit.

We isolate the implementation of two syndrome measurement circuits and the implementation of surface code memory, thanks to the feature in Kirin.

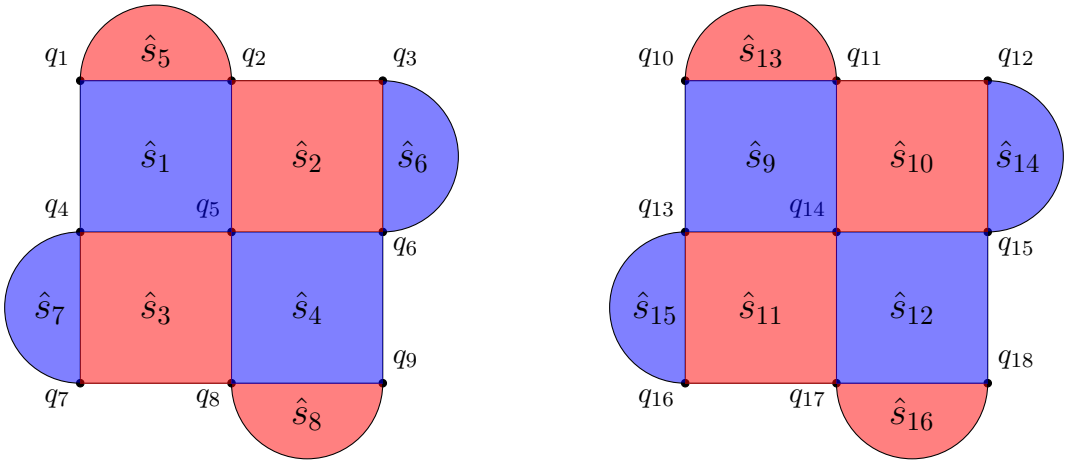
Listing 2. Implementation of 3 by 3 surface code memory

```
@qasm2.extended
def surface_code_d3_circuit():
    qreg = qasm2.qreg(2*3**2-1)
    creg = qasm2.creg(3**2-1)
    add_X_syndrome_circuit(qreg,creg,9,0,(0,1,3,4))
    add_X_syndrome_circuit(qreg,creg,9,1,(3,4,6,7))

    add_X_syndrome_circuit(qreg,creg,9,2,(1,2))
    add_X_syndrome_circuit(qreg,creg,9,3,(7,8))

    add_Z_syndrome_circuit(qreg,creg,9,4,(1,2,4,5))
    add_Z_syndrome_circuit(qreg,creg,9,5,(4,5,7,8))


    add_Z_syndrome_circuit(qreg,creg,9,6,(0,3))
    add_Z_syndrome_circuit(qreg,creg,9,7,(2,5))
```



We also implement a simple logical circuit on two distance 3 surface code. In this logical circuit, we apply a logical X gate on the first logical qubit, and then apply a transversal logical CNOT gate. Finally, we measure the logical Z on the second logical qubit.

Listing 3. Implementation of 3 by 3 surface code

```
@qasm2.extended
def surface_code_d3_transversal_cnot():
    qreg = qasm2.qreg(2*2*3**2-2)
```

```
creg = qasm2.creg(2*3**2-2+3)


for i in range(2*3**2-2):
    qasm2.reset(qreg[i])

'''
Logical X gate
'''
qasm2.x(qreg[9])
qasm2.x(qreg[10])
qasm2.x(qreg[11])


'''
Logical transversal CNOT gate
'''
for i in range(9):
    qasm2.cx(qreg[i],qreg[i+9])


'''
Syndrome measurement on first logical qubit
'''
add_X_syndrome_circuit(qreg,creg,18,0,(0,1,3,4))
add_X_syndrome_circuit(qreg,creg,18,1,(3,4,6,7))

add_X_syndrome_circuit(qreg,creg,18,2,(1,2))
add_X_syndrome_circuit(qreg,creg,18,3,(7,8))

add_Z_syndrome_circuit(qreg,creg,18,4,(1,2,4,5))
add_Z_syndrome_circuit(qreg,creg,18,5,(4,5,7,8))


add_Z_syndrome_circuit(qreg,creg,18,6,(0,3))
add_Z_syndrome_circuit(qreg,creg,18,7,(2,5))

'''
Syndrome measurement on second logical qubit
'''
add_X_syndrome_circuit(qreg,creg,18,8,(9,10,12,13))
add_X_syndrome_circuit(qreg,creg,18,9,(12,13,15,16))

add_X_syndrome_circuit(qreg,creg,18,10,(10,11))
add_X_syndrome_circuit(qreg,creg,18,11,(16,17))

add_Z_syndrome_circuit(qreg,creg,18,12,(10,11,13,14))
add_Z_syndrome_circuit(qreg,creg,18,13,(13,14,16,17))
```

```
    add_Z_syndrome_circuit(qreg,creg,18,14,(9,12))
    add_Z_syndrome_circuit(qreg,creg,18,15,(11,14))

    '''
    Measuring the logical qubit
    '''
    qasm2.measure(qreg[9],creg[16])
    qasm2.measure(qreg[10],creg[17])
    qasm2.measure(qreg[11],creg[18])
    return creg
```

We explore the idea of using recursion to compile fault-tolerant measurement. In syndrome measurement circuit, quantum error might occur at syndrome qubit. To get more confidence of syndrome outcome, people normally apply a fault-tolerant measurement gadget, where they repeat the same measurement multiple times until they are all the same. The idea of fault-tolerant measurement is similar to *repeat-until-success* semantic introduced in the example. So we figure out a recursive algorithm, which implement the *repeat-until-success* fault-tolerant measurement gadget.

Listing 4. Recursive implemention of fault-tolerant measurement

```
@qasm2.extended
def add_X_syndrome_circuit_recursive(result:int,round:int,qreg: qasm2.QReg,
    creg:qasm2.CReg,ndataqubits:int,stabindex:int,index_tuple:tuple[int,
    ...]):
    if round==3:
        return

    qasm2.h(qreg[ndataqubits+stabindex])
    for i in range(len(index_tuple)):
        qasm2.cx(qreg[ndataqubits+stabindex],qreg[index_tuple[i]])
    qasm2.h(qreg[ndataqubits+stabindex])
    qasm2.measure(qreg[ndataqubits+stabindex],creg[stabindex])


    if creg[stabindex]==result:
        return add_X_syndrome_circuit_recursive(result,round+1,qreg,creg,
            ndataqubits,stabindex,index_tuple)

    qasm2.reset(qreg[ndataqubits+stabindex])
    return add_X_syndrome_circuit_recursive(1-result,1,qreg,creg,
        ndataqubits,stabindex,index_tuple)

@qasm2.extended
def add_Z_syndrome_circuit_recursive(result:int,round:int,qreg: qasm2.QReg,
    creg:qasm2.CReg,ndataqubits:int,stabindex:int,index_tuple:tuple[int,
    ...]):
    if round==3:
        return
```

```
    for i in range(len(index_tuple)):
        qasm2.cx(qreg[index_tuple[i]],qreg[ndataqubits+stabindex])
    qasm2.measure(qreg[ndataqubits+stabindex],creg[stabindex])

    if creg[stabindex]==result:
        return add_Z_syndrome_circuit_recursive(result,round+1,qreg,creg,
            ndataqubits,stabindex,index_tuple)

    qasm2.reset(qreg[ndataqubits+stabindex])
    return add_Z_syndrome_circuit_recursive(1-result,1,qreg,creg,
        ndataqubits,stabindex,index_tuple)
```

The idea for the recursion:

1. We keep two integer *result*, and *round*. *result* means the last measurement result. *round* means we have already measure the same result for that many rounds of the same stabilizer.

2. In the new call, if *round* is already 3(Or other user specified constant), then the function directly return. This is where we are certain that the measurement is fault-tolerant.

3. If the new measurement match with the previous result, we set $round \to round + 1$ and keep recursive call with the same *result*.

4. If the new measurement doesn't match with the previous result, we set $round \to 0$, set $result \to 1 - result$ and start over again by recursive call!

Despite our brilliant idea, the above two functions, cannot be compiled by Kirin. The compile stuck in infinite loop of parsing AST tree, although we are convinced that the algorithm must be correct.

3 Quantum Simulation: Trotterized Evolution and Variational Approach

3.1 Transverse Field Ising Model

The Hamiltonian of the TFIM is given by

$$H = -J \sum_{\langle i,j \rangle} Z_i Z_j - h \sum_i X_i,$$

where J characterizes the strength of the spin–spin interaction term $Z_i Z_j$ where Z_i is the Pauli Z operator on the i^{th} site, and h denotes the strength of the external transverse field in the x-direction, represented by X_i, the Pauli X operator on the i^{th} site.

The TFIM exhibits two distinct quantum phases. When $h > J$, the transverse field dominates, and the ground state tends to align along the x-direction (i.e. $|+\rangle = \frac{1}{\sqrt{2}}(|\uparrow\rangle + |\downarrow\rangle)$). In this regime, the system is disordered in the z-basis, and the total z-magnetization has

zero expectation value. Conversely, when $J > h$, the spin–spin interaction dominates, favoring an ordered ground state in the z-basis. The system spontaneously breaks symmetry and settles into one of the two ferromagnetic states: $|\uparrow\uparrow\cdots\rangle$ or $|\downarrow\downarrow\cdots\rangle$, we call this phase the ordered phase.

3.2 Trotterized Time Evolution

Trotterized time evolution is based on the Lie-Trotter formula. When the time step Δt is small, the time evolution operator for a Hamiltonian composed of multiple non-commuting terms can be approximated as:

$$e^{-i(H_1+H_2)\Delta t} \approx e^{-iH_1\Delta t}e^{-iH_2\Delta t} + \mathcal{O}(\Delta t^2) \tag{3.1}$$

In the TFIM, we identify the Hamiltonian terms as $H_1 = -J\sum_{\langle i,j\rangle} Z_iZ_j$ and $H_2 = -h\sum_i X_i$. The Trotterized time evolution operator over a small time step Δt can then be written as:

$$U(t+\Delta t, t) \approx \prod_{\langle i,j\rangle} e^{iJZ_iZ_j\Delta t}\prod_k e^{ihX_k\Delta t} \tag{3.2}$$

In `Bloqade`, there is a convenient built-in method for implementing exponentiated multi-qubit gates such as $e^{-iJZ_iZ_j\Delta t}$ (see code listing 5). This is achieved by transferring the parity of the qubits to an ancilla or target qubit, performing a single-qubit Z-rotation, and then uncomputing the parity transfer. We make extensive use of this feature in our simulation to efficiently construct the interaction part of the Trotter step.

Higher order Trotterizations also exist, i.e. the Trotter-Suzuki formula,

$$e^{-i(H_1+H_2)\Delta t} \approx e^{-iH_1\Delta t/2}e^{-iH_2\Delta t}e^{-iH_1\Delta t/2} + \mathcal{O}(\Delta t^3), \tag{3.3}$$

which is accurate to the next order in Δt.

Listing 5. Bloqade feature for multi-qubit correlated time evolution operator

```
@qasm2.extended
def zzzz_gadget(targets: tuple[qasm2.Qubit, ...], gamma: float):
    for i in range(len(targets) - 1):
        qasm2.cx(targets[i], targets[i + 1])

    qasm2.rz(targets[-1], gamma)

    for j in range(len(targets) - 1):
        qasm2.cx(targets[-j - 1], targets[-j - 2])
```

3.3 Evolution with Time Independent Hamiltonian

Using the Trotterized evolution technique we outlined above, we first simulate the evolution of initial state where all spins are up under the TFIM Hamiltonian with constant parameters $J = 0.2$ and $h = 1.2$. We discovered that the state oscillates and slowly converges

to the complete disordered state. Given our knowledge about the two phases of TFIM, we conclude that this parameter choice that parameter choice, the model is in the disordered phase.

We contrast the first-order Lie-Trotter approximation with the second-order Trotter-Suzuki approximation in Fig. 1; we can see that the higher order approximation converges more quickly.

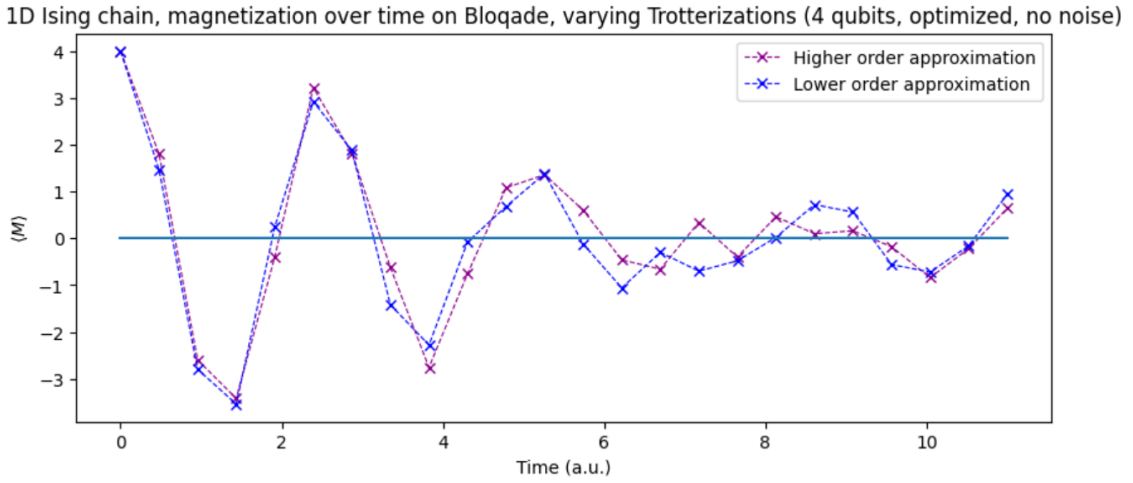


Figure 1. Time evolution of the magnetization under TFIM Hamiltonian with $J = 0.2$, $h = 1.2$ for the first-order and second-order Trotterizations.

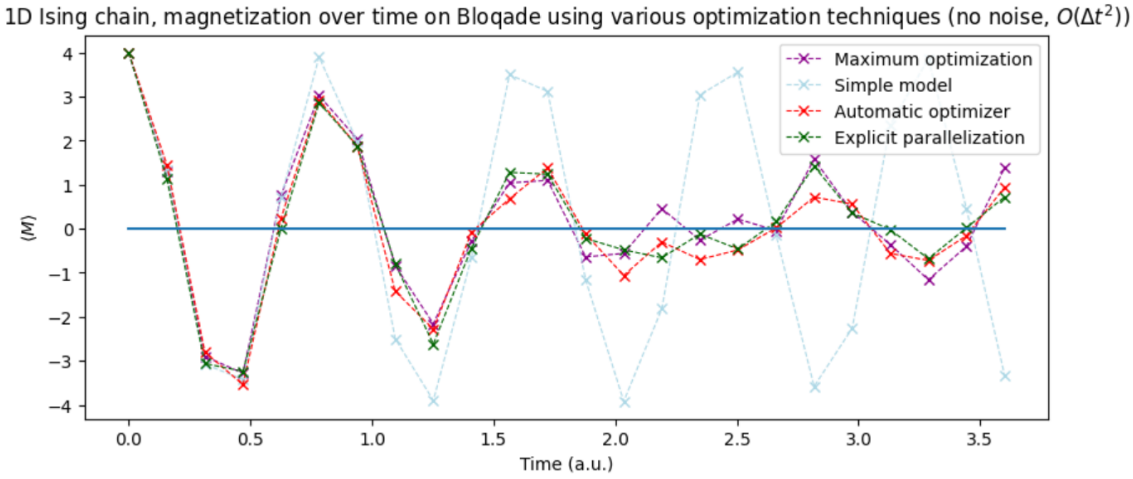


Figure 2. First order approximation to the TFIM with multiple circuit optimization methods.

To achieve co-design between the TFIM and the Bloqade SDK, we made extensive use of the new, high-level features in the Bloqade. For example, we used a for loop to capture the repetitive nature of the Trotterized Hamiltonian; we used parallel gates to decrease both the circuit and execution depth. We contrast several circuit optimizations and their

effects on performance in Fig. 2.

Finally, we swap the values of J and h to confirm that we can see the ordered phase in experiment. Indeed, for 8 qubits, we product the result in Fig. 3. The state does not change over time.

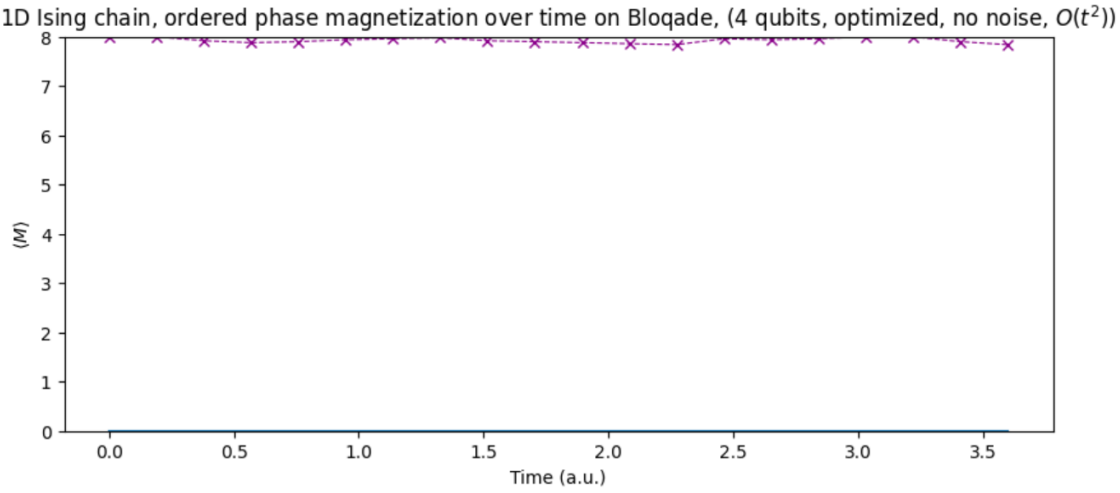


Figure 3. An ordered phase of the TFIM remains invariant under time evolution.

3.4 Evolution with Slowly Varing Time Dependent Hamiltonian

We have found that different phases of TFIM lead to significantly different time evolution behvarior. This observation motivated us to explore how the system responds to a slow, continuous change in parameters—specifically, what happens if we adiabatically vary the Hamiltonian parameters toward the critical point without crossing it, versus what happens when we evolve through the critical point.

According to the adiabatic theorem, if a quantum system begins in its ground state and the Hamiltonian changes slowly while remaining gapped, the system will stay in the instantaneous ground state throughout the evolution. Therefore, if we initialize the system in the all-spin-up state and set the initial Hamiltonian parameters within the ordered phase, the system should remain in that state—as it continues to be the ground state—provided we do not reach the critical point.

In contrast, if the evolution crosses the critical point, the slow change of the Hamiltonian is no longer considered adiabatic because at the critical point, there will be no gap between ground state and excited state. After crossing the critical point, the system's true ground state becomes qualitatively different, and we expect the evolved state to gradually converge toward this new disordered ground state.

To quantify the degree of orderedness in the system, we evaluate the magnetization in

the z-direction after each time-evolution step by repeatedly sampling the circuit. For each subsequent step, we duplicate the previous circuit and apply an additional Trotterized layer of evolution. Our simulation is implemented on an 8-qubit register, so the maximum possible magnetization is $M = 4$, corresponding to a fully ordered state with all spins aligned in the $+z$ or $-z$ direction. A magnetization of $M = 0$ indicates a completely disordered state in the z-basis. As expected, in the case of adiabatic evolution that does not cross the critical point, the magnetization remains close to 4. In contrast, when the critical point is crossed, the system rapidly transitions toward the disordered state, and the magnetization drops accordingly (see Fig. 5).

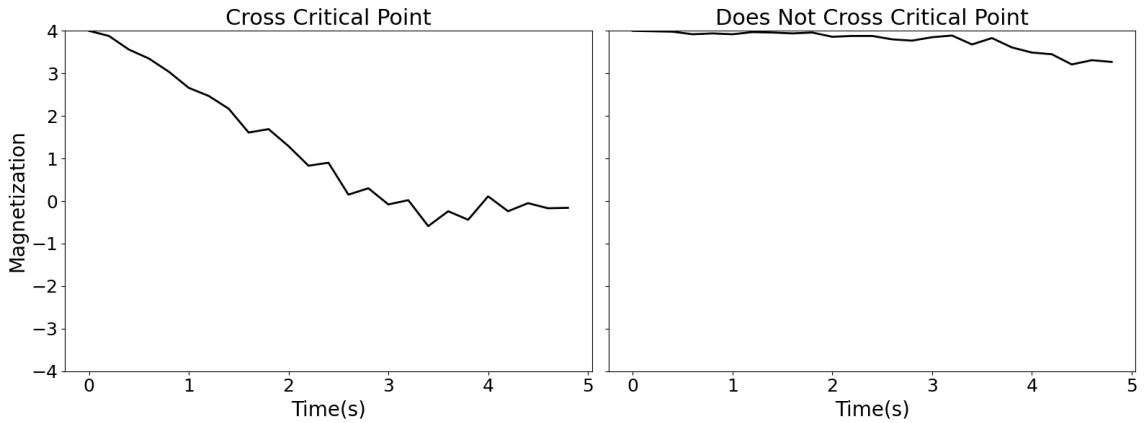


Figure 4. Comparison of two simulated time evolution with slowly changing Hamiltonian. In the left graph, we evolve J from $0,5$ to 0.7 and h from 0.7 ti 0.5, which crosses the critical point $J = h$. In the right graph, we evolve J from 1.2 to 1.0 and h from 1.0 to 1.2.

3.5 Variational Method for Schwinger Model Ground State

In addition to time evolution simulations, we implemented a variational approach to find the ground state of the Schwinger model - a fundamental model in lattice quantum electrodynamics.

$$H_{\text{Schwinger}} = \frac{g^2 a}{2} \sum_{n=1}^{N-1} \left[\frac{1}{2} \sum_{k=1}^{n} (\sigma_k^z + (-1)^k) \right]^2 + \frac{m}{2} \sum_{n=1}^{N} (-1)^n \sigma_n^z + \frac{1}{2a} \sum_{n=1}^{N-1} (\sigma_n^+ \sigma_{n+1}^- + h.c.) \tag{3.4}$$

We designed a parameterized quantum circuit ansatz consisting of alternating layers of single-qubit rotations and entangling operations. Each layer contains:

1. Parameterized rotations (Rx, Ry, Rz) on each qubit
2. A pattern of CNOT and CZ gates to create entanglement between qubits

The workflow follows these steps:

1. Initialize parameters randomly
2. Execute the parameterized circuit on the quantum hardware
3. Measure observables corresponding to terms in the Schwinger Hamiltonian
4. Calculate the total energy expectation value
5. Update parameters using classical optimization (COBYLA method)
6. Repeat until convergence

Our implementation leverages Bloqade's parallel gate capabilities and decomposition to native Rydberg gates. We achieved a final energy of -2.14, which is within 6% of the theoretical ground state energy (-2.27).

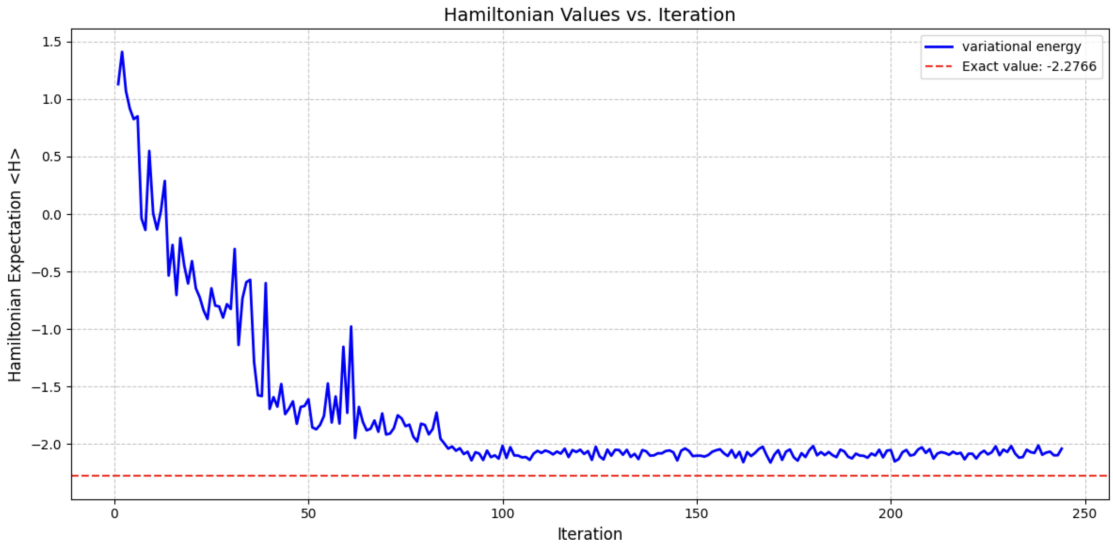


Figure 5. Variation Circuit Schwinger Model Ground State Preparation with Bloqade. N=4, a=g=m=1

This approach demonstrates how the new features in Kirin - especially parallel gates and the ability to create modular code - enable efficient simulation of complex quantum field theories on neutral atom hardware. The variational method complements our Trotterized evolution approach, creating a comprehensive toolkit for quantum simulation with Bloqade.

4 Enhanced Circuit Compilation and Visualization Flow

In the previous sections, we demonstrated how advanced features in `kirin`, such as the `if-else` statement and the `for` loop, simplify algorithm development.

Although `Bloqade` includes several circuit simplification passes (e.g., `UOpToParallel`), we observe that the circuit can be further optimized by adopting more aggressive techniques

that optimize the unitary matrices without affecting the overall functionality. In this section, we introduce our compilation flow specialized for the neutral atom array architecture that further optimizes the circuit to improve the performance.

Additionally, while the `qbraid` package provides an interface for circuit visualization, `bloqade` does not currently offer passes for layout synthesis or for generating placement and routing solutions, gate events, AOD movements, and Rydberg instructions. Consequently, users cannot directly visualize the algorithm using the existing code base. To address this, we exploit an open-source circuit optimization tool [3] into the `bloqade` toolchain to enable comprehensive visualization.

Our key improvements to the existing toolchain are the following:

1. An end-to-end circuit synthesis and visualization flow for the zoned architecture.
2. A circuit rewriting passes based on phase polynomial optimization and commutable gates reordering.

4.1 Circuit Rewriting for More Parallel Single-Qubit Gates

In this subsection, we present a toy example that demonstrates both when `bloqade` can successfully identify parallelizable gates and when it cannot. We then propose a general solution to this problem, and validate its effectiveness through prototyping and experimentation on the algorithm introduced in the previous section.

Listing 6 illustrates a case where `bloqade` correctly identifies the parallel unitary `u(pi/2, 0, pi)`. Since the same unitary is applied to all three qubits in the system, the compiler is able to raise the abstraction level by grouping these instructions into a single `parallel.U`. This grouped instruction can then be directly utilized in the architecture's native instruction set to improve performance.

Listing 6. Example of parallelized unitary

```
@qasm2.extended
def main():
    q = qasm2.qreg(3)
    qasm2.u(q[0], pi/2, 0, pi)
    qasm2.u(q[1], pi/2, 0, pi)
    qasm2.u(q[2], pi/2, 0, pi)
    qasm2.cx(q[0], q[1])

UOpToParallel(dialects=main.dialects)(main)

"""
qreg q[3];
parallel.U(1.5707963267948966, 0, 3.141592653589793) {
  q[0];
  q[1];
  q[2];
}
CX q[0], q[1];
```

```
"""
```

However, `bloqade` does not always group parallelizable gates as expected. For example, Listing 7 shows a case where it fails to do so. The main difference from the previous code is that an inconsistent rotation angle is used when declaring the unitary on qubit `q[1]`, i.e., `U(pi/2, 0, -pi)`. Although this unitary is equivalent to the one used previously (since a rotation of `-pi` is equivalent to `pi`), running the `UOpToParallel` pass on the code with the inconsistent angle results in the compiler failing to recognize the parallelism fully; it only groups the operations on `q[0]` and `q[2]`.

Listing 7. Failed to parallelize untaries due to inconsistent rotation angle

```
@qasm2.extended
def main():
    q = qasm2.qreg(3)
    qasm2.u(q[0], pi/2, 0, pi)
    qasm2.u(q[1], pi/2, 0, -pi)
    qasm2.u(q[2], pi/2, 0, pi)
    qasm2.cx(q[0], q[1])

UOpToParallel(dialects=main.dialects)(main)

"""
qreg q[3];
parallel.U(1.5707963267948966, 0, 3.141592653589793) {
  q[0];
  q[2];
}
U(1.5707963267948966, 0, -3.141592653589793) q[1];
CX q[0], q[1];
"""
```

A more advanced example is shown in Listing 8, where the circuit comprises three Z rotations and a CNOT gate. Note that the Z rotation gate on `q[0]` commutes with the CNOT gate controlled by `q[0]` and targeting `q[1]`. Since swapping these two gates does not affect the overall functionality, rearranging them would result in three consecutive Z rotations that the compiler could group into a single parallel Z rotation. However, `bloqade` fails to perform this simplification because it does not leverage the flexibility of reordering commutable gates.

Listing 8. Failed to parallelize unitaries due to suboptimal ordering of commutable gates

```
@qasm2.extended
def main():
    q = qasm2.qreg(3)
    qasm2.rz(q[2], pi/3)
    qasm2.rz(q[1], pi/3)
    qasm2.cx(q[0], q[1])
    qasm2.rz(q[0], pi/3)
```

```
UOpToParallel(dialects=main.dialects)(main)

"""
qreg q[3];
parallel.RZ(1.0471975511965976) {
  q[0];
  q[1];
}
CX q[0], q[1];
rz (1.0471975511965976) q[0];

"""
```

To mitigate these issues, we employ an algebraic method to simplify single-qubit unitaries and regularize the rotation angles. More specifically, we leverage ZX rewriting rules implemented in `PyZX` to optimize the circuit prior to running the `UOpToParallel` pass.

Table 1. Cost metric for the qualitative comparison

Gate Type	**Cost Weight**
Local 1-qubit	0.2
Local 2-qubit	0.4
Global 1-qubit	0.1
Global 2-qubit	0.4

To demonstrate the effectiveness of our flow, we conducted prototyping and experimentation on the algorithm introduced in the previous section. As shown in Table 1, the cost is defined based on the number of local single-qubit gates, local two-qubit gates, parallel single-qubit gates, and parallel two-qubit gates. We assign a lower cost to the parallel single-qubit gate than to the local ones to encourage the tools to parallelize them. Results in Table 2 show that our flow utilizes more parallelism than directly parsing it through `UOpToParallel`.

Table 2. Qualitative comparison between different circuit rewriting flows

Circuit	**Cost**
Original Ising Model	268.80
Original Ising Model + Qiskit	76.80
Original Ising Model + Qiskit + UOpToParallel	72.10
Original Ising Model + Qiskit + ZX	120.00
Original Ising Model + Qiskit + ZX + UOpToParallel	65.40

4.2 Gate Scheduling for More Parallel Two-Qubit Gates

Besides the techniques introduced in the previous section, we also leverage the zoned architecture to parallelize two-qubit gates, specifically CZ gates. To generate the placement

and routing solution for each intermediate time step, we employ an open-source layout synthesis tool named `ZAC` [3].

The overall procedure consists of the following stages:

1. **Gate scheduling:** Generate the gate dependency graph, where vertices represent CZ gates and edges denote qubit conflicts (i.e., two gates sharing at least one qubit). Since CZ gates acting on different qubit pairs can be executed concurrently, this step maximizes the number of parallel CZ operations by applying a vertex coloring algorithm that uses the minimal number of colors.

2. **Placement:** For each CZ gate, determine the optimal site within the gate zone (specialized for the 1D zoned architecture) by solving a shortest distance bipartite graph maximal matching problem where the first set of vertices represent the site of each qubit before each movement and the second set of vertices represent the candidate destination sites. The edges are weighted by the distance to travel between the two sites.

3. **Qubit routing:** For each *rearrangement* layer, compute the shortest path for each qubit to move from its current location to its destination (either to the gate zone for a Rydberg gate or back to the storage zone). Note that the qubit movements are achieved using AOD, and we need to consider the geometry constraints that two movements cannot overlap.

Due to space constraints, we refer the reader to the original papers for further details [3, 4]. We generate videos to demonstrate the effectiveness of our layout synthesis flow.

The parallel CZ layer is displayed in Figure 6. The blue rectangle represents the Rydberg region in which the adjacent gates are entangled. In this frame, two pairs of qubits stay in the entanglement zone, and these two CZ gates are performed in parallel.

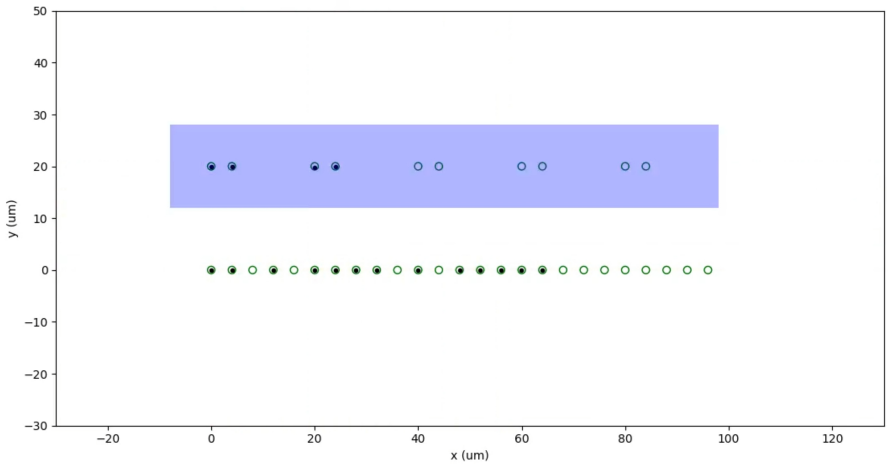


Figure 6. Visualization of a parallel CZ layer.

The movement layer is visualized in Figure 7. AOD is plotted using the vertical and horizontal dashed lines. Three atoms are trapped and moved by the AOD from the storage

zone to the gate zone. Note that the AODs could trap multiple atoms, and these movements are parallelized if their trajectory does not overlap.

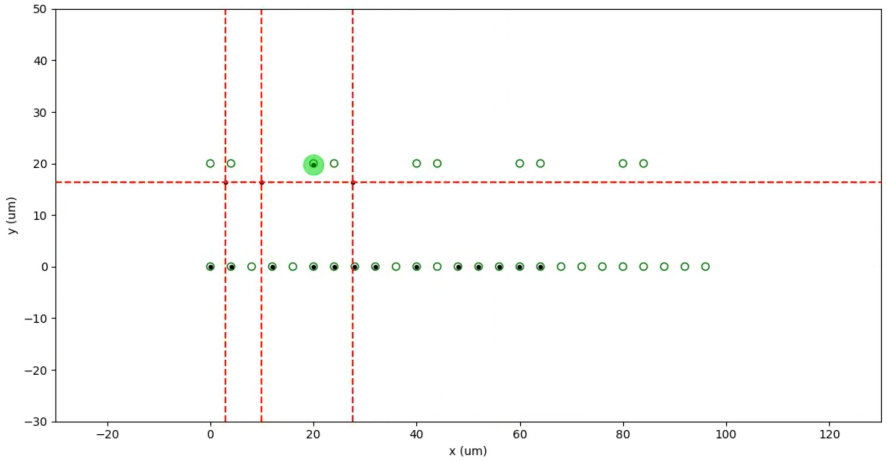


Figure 7. Visualization of a parallel movement layer.

Finally, we exploit advanced features such as qubit reuse. The idea is that if the same qubit is involved in two consecutive layers, it does not need to be returned to the storage zone and fetched again. Instead, the qubit can remain in the gate zone—a strategy we refer to as *qubit reuse*. For more details on how reuse is utilized to reduce qubit movements, please refer to the paper [3].

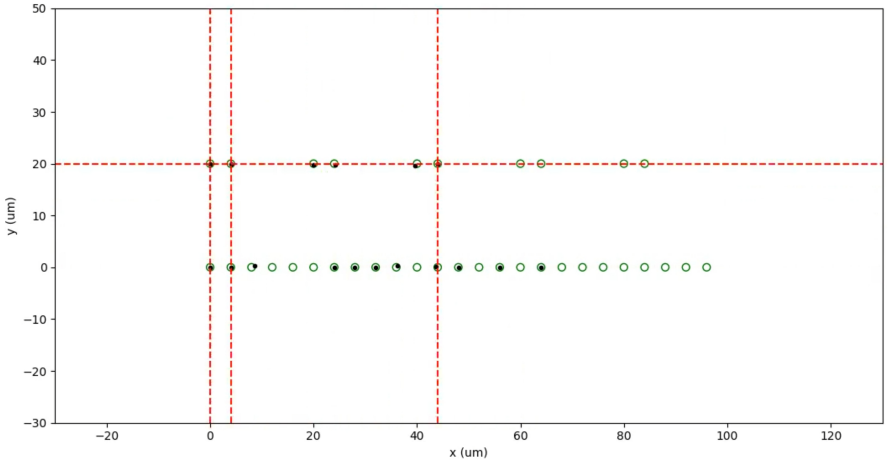


Figure 8. Visualization of qubit reuse.

5 Bug reports

5.1 Infinite loop in compiling recursion

When we try to compile the recursive feature we developed, the program falls into an infinite loop.

5.2 Probability zero events

References

[1] G. Semeghini, H. Levine, A. Keesling, S. Ebadi, T.T. Wang, D. Bluvstein et al., *Probing topological spin liquids on a programmable quantum simulator*, *Science* **374** (2021) 1242.

[2] "Index - the neutral atom sdk."

[3] W.-H. Lin, D.B. Tan and J. Cong, *Reuse-aware compilation for zoned quantum architectures based on neutral atoms*, in *2025 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 127–142, 2025, DOI.

[4] D.B. Tan, W.-H. Lin and J. Cong, *Compilation for dynamically field-programmable qubit arrays with efficient and provably near-optimal scheduling*, in *Proceedings of the 30th Asia and South Pacific Design Automation Conference*, pp. 921–929, 2025.