Circuits 1: Compiling arbitrary single-particle basis rotations in linear depth

Run in Google Colab (https://colab.research.google.com/github/quantumlib/OpenFermion/blob/master/docs/

This is the first of several tutorials demonstrating the compilation of quantum circuits. These tutorials build on one another and should be studied in order. In this tutorial we will discuss the compilation of circuits for implementing arbitrary rotations of the single-particle basis of an electronic structure simulation. As an example, we show how one can use these methods to simulate the evolution of an arbitrary non-interacting fermion model.

Setup

Install the OpenFermion package:

```
try:
    import openfermion
except ImportError:
    !pip install git+https://github.com/quantumlib/OpenFermion.git@master#egg=
```

Background

Second quantized fermionic operators

In order to represent fermionic systems on a quantum computer one must first discretize space. Usually, one expands the many-body wavefunction in a basis of spin-orbitals $\varphi_p = \varphi_p(r)$ which are single-particle basis functions. For reasons of spatial efficiency, all NISQ (and even most error-corrected) algorithms for simulating fermionic systems focus on representing operators in second-quantization. Second-quantized operators are expressed using the fermionic creation and annihilation operators, $a_p^\dagger$ and a_p. The action of $a_p^\dagger$ is to excite a fermion in spin-orbital φ_p and the action of a_p is to annihilate a fermion from spin-

orbital φ_p. Specifically, if electron i is represented in a space of spin-orbitals $\{\varphi_p(r_i)\}$ then $a_p^\dagger$ and a_p are related to Slater determinants through the equivalence,

$$\langle r_0 \cdots r_{\eta-1}|a_0^\dagger \cdots a_{\eta-1}^\dagger|\varnothing\rangle \equiv \sqrt{\frac{1}{\eta!}} \begin{vmatrix} \varphi_0(r_0) & \varphi_1(r_0) & \cdots & \varphi_{\eta-1}(r_0) \\ \varphi_0(r_1) & \varphi_1(r_1) & \cdots & \varphi_{\eta-1}(r_1) \\ \vdots & \vdots & \ddots & \vdots \\ \varphi_0(r_{\eta-1}) & \varphi_1(r_{\eta-1}) & \cdots & \varphi_{\eta-1}(r_{\eta-1}) \end{vmatrix}$$

where η is the number of electrons in the system, $|\varnothing\rangle$ is the Fermi vacuum and $\varphi_p(r) = \langle r|\varphi_p\rangle$ are the single-particle orbitals that define the basis. By using a basis of Slater determinants, we ensure antisymmetry in the encoded state.

Rotations of the single-particle basis

Very often in electronic structure calculations one would like to rotate the single-particle basis. That is, one would like to generate new orbitals that are formed from a linear combination of the old orbitals. Any particle-conserving rotation of the single-particle basis can be expressed as

$$\tilde{\varphi}_p = \sum_q \varphi_q u_{pq} \quad \tilde{a}_p^\dagger = \sum_q a_q^\dagger u_{pq} \quad \tilde{a}_p = \sum_q a_q u_{pq}^*$$

where $\tilde{\varphi}_p$, $\tilde{a}_p^\dagger$, and $\tilde{a}_p^\dagger$ correspond to spin-orbitals and operators in the rotated basis and u is an $N \times N$ unitary matrix. From the Thouless theorem, this single-particle rotation is equivalent to applying the $2^N \times 2^N$ operator

$$U(u) = \exp\left(\sum_{pq} [\log u]_{pq} \left(a_p^\dagger a_q - a_q^\dagger a_p\right)\right)$$

where $[\log u]_{pq}$ is the (p, q) element of the matrix $\log u$.

There are many reasons that one might be interested in performing such basis rotations. For instance, one might be interested in preparing the Hartree-Fock (mean-field) state of a chemical system, by rotating from some initial orbitals (e.g. atomic orbitals or plane waves) into the molecular orbitals of the system. Alternatively, one might be interested in rotating from a basis where certain operators are diagonal (e.g. the kinetic operator is diagonal in the plane wave basis) to a basis where certain other operators are diagonal (e.g. the Coulomb operator is diagonal in the position basis). Thus, it is a very useful thing to be able to apply circuits corresponding to $U(u)$ on a quantum computer in low depth.

Compiling linear depth circuits to rotate the orbital basis

OpenFermion prominently features routines for implementing the linear depth / linear connectivity basis transformations described in Phys. Rev. Lett. 120, 110501 (https://journals.aps.org/prl/abstract/10.1103/PhysRevLett.120.110501). While we will not discuss this functionality here, we also support routines for compiling the more general form of these transformations which do not conserve particle-number, known as a Bogoliubov transformation, using routines described in Phys. Rev. Applied 9, 044036 (https://journals.aps.org/prapplied/abstract/10.1103/PhysRevApplied.9.044036). We will not discuss the details of how these methods are implemented here and instead refer readers to those papers. All that one needs in order to compile the circuit $U(u)$ using OpenFermion is the $N \times N$ matrix u, which we refer to in documentation as the "basis_transformation_matrix". Note that if one intends to apply this matrix to a computational basis state with only η electrons, then one can reduce the number of gates required by instead supplying the $\eta \times N$ rectangular matrix that characterizes the rotation of the occupied orbitals only. OpenFermion will automatically take advantage of this symmetry.

OpenFermion example implementation: exact evolution unde tight binding models

In this example will show how basis transforms can be used to implement exact evolution under a random Hermitian one-body fermionic operator \begin{equation} H = \sum{pq} T{pq} a^\dagger_p aq. \end{equation} That is, we will compile a circuit to implement e^{-iHt} for some time t. Of course, this is a tractable problem classically but we discuss it here since it is often useful as a subroutine for more complex quantum simulations. To accomplish this evolution, we will use basis transformations. Suppose that u is the basis transformation matrix that diagonalizes T. Then, we could implement e^{-iHt} by implementing \(U(u)^\dagger (\prod{k} e^{-i \lambda_k Z_k}) U(u)\) where λ_k are the eigenvalues of T.

Below, we initialize the T matrix characterizing H and then obtain the eigenvalues λ_k and eigenvectors u_k of T. We print out the OpenFermion FermionOperator representation of T.

```
import openfermion
import numpy

# Set the number of qubits in our example.
n_qubits = 3
simulation_time = 1.
random_seed = 8317
```

```
# Generate the random one-body operator.
T = openfermion.random_hermitian_matrix(n_qubits, seed=random_seed)

# Diagonalize T and obtain basis transformation matrix (aka "u").
eigenvalues, eigenvectors = numpy.linalg.eigh(T)
basis_transformation_matrix = eigenvectors.transpose()

# Print out familiar OpenFermion "FermionOperator" form of H.
H = openfermion.FermionOperator()
for p in range(n_qubits):
    for q in range(n_qubits):
        term = ((p, 1), (q, 0))
        H += openfermion.FermionOperator(term, T[p, q])
print(H)
```

```
(0.5367212624097257+0j) [0^ 0] +
(-0.26033703159240107+3.3259173741375454j) [0^ 1] +
(1.3433603748462144+1.544987250567917j) [0^ 2] +
(-0.26033703159240107-3.3259173741375454j) [1^ 0] +
(-2.9143303700812435+0j) [1^ 1] +
(-1.52843836446248+1.3527486791390022j) [1^ 2] +
(1.3433603748462144-1.544987250567917j) [2^ 0] +
(-1.52843836446248-1.3527486791390022j) [2^ 1] +
(2.261633626116526+0j) [2^ 2]
```

Now we're ready to make a circuit! First we will use OpenFermion to generate the basis transform $U(u)$ from the basis transformation matrix u by calling the Bogoliubov transform function (named as such because this function can also handle non-particle conserving basis transformations). Then, we'll apply local Z rotations to phase by the eigenvalues, then we'll apply the inverse transformation. That will finish the circuit. We're just going to print out the first rotation to keep things easy-to-read, but feel free to play around with the notebook.

```
import openfermion
import cirq
import cirq_google

# Initialize the qubit register.
qubits = cirq.LineQubit.range(n_qubits)

# Start circuit with the inverse basis rotation, print out this step.
inverse_basis_rotation = cirq.inverse(openfermion.bogoliubov_transform(qubits,
circuit = cirq.Circuit(inverse_basis_rotation)
```

```
print(circuit)

# Add diagonal phase rotations to circuit.
for k, eigenvalue in enumerate(eigenvalues):
    phase = -eigenvalue * simulation_time
    circuit.append(cirq.rz(rads=phase).on(qubits[k]))

# Finally, restore basis.
basis_rotation = openfermion.bogoliubov_transform(qubits, basis_transformatio
circuit.append(basis_rotation)
```

```
0: ──────────────────────────────────────────────PhISwap(0.25)─────────Rz(0)────
                                                      │
1: ──────────PhISwap(0.25)─────────Z^0.522────PhISwap(0.25)^-0.656──PhISwap
             │                                                        │
2: ──Z^0.762──PhISwap(0.25)^-0.249──Z^-0.519──────────────────────PhISwap
```

Finally, we can check whether our circuit applied to a random initial state with the exact result. Print out the fidelity with the exact result.

```
# Initialize a random initial state.
initial_state = openfermion.haar_random_vector(
    2 ** n_qubits, random_seed).astype(numpy.complex64)

# Numerically compute the correct circuit output.
import scipy
hamiltonian_sparse = openfermion.get_sparse_operator(H)
exact_state = scipy.sparse.linalg.expm_multiply(
    -1j * simulation_time * hamiltonian_sparse, initial_state)

# Use Cirq simulator to apply circuit.
simulator = cirq.Simulator()
result = simulator.simulate(circuit, qubit_order=qubits,
                            initial_state=initial_state)
simulated_state = result.final_state_vector

# Print final fidelity.
fidelity = abs(numpy.dot(simulated_state, numpy.conjugate(exact_state)))**2
print(fidelity)
```

```
1.0000000960717732
```

Thus, we see that the circuit correctly effects the intended evolution. We can now use Cirq's compiler to output the circuit using gates native to near-term devices, and then optimize those circuits. We'll output in QASM 2.0 just to demonstrate that functionality.

```
xmon_circuit = cirq.optimize_for_target_gateset(circuit, gateset=cirq.CZTarget
print(xmon_circuit.to_qasm())
```

```
// Generated from Cirq v1.1.0

OPENQASM 2.0;
include "qelib1.inc";


// Qubits: [q(0), q(1), q(2)]
qreg q[3];


u3(pi*1.5,pi*1.417454382,pi*0.582545618) q[1];
u3(pi*1.5,pi*1.6556038174,pi*0.3443961826) q[2];
cz q[1],q[2];
u3(pi*0.1242949803,pi*1.417454382,pi*0.582545618) q[1];
```


Last updated 2023-01-18 UTC.