Circuits 3: Low rank, arbitrary basis molecular simulations

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

Setup

Install the OpenFermion package:

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

Low rank decomposition of the Coulomb operator

The algorithm discussed in this tutorial is described in arXiv:1808.02625 (https://arxiv.org/abs/1808.02625).

In Circuits 1 (https://quantumai.google/openfermion/tutorials/circuits_1_basis_change) we discussed methods for very compiling single-particle basis transformations of fermionic operators in $O(N)$ depth on a linearly connected architecture. We looked at the particular example of simulating a free fermion model by using Bogoliubov transformations to diagonalize the model.

In Circuits 2 (https://quantumai.google/openfermion/tutorials/circuits_2_diagonal_coulomb_trotter) we discussed methods for compiling Trotter steps of electronic structure Hamiltonian in $O(N)$ depth on a linearly connected architecture when expressed in a basis diagonalizing the Coulomb operator so that

$$H = \sum_{pq} T_{pq} a_p^\dagger a_q + \sum_{pq} V_{pq} a_p^\dagger a_p a_q^\dagger a_q.$$

Here we will discuss how both of those techniques can be combined, along with some insights from electronic structure, in order to simulate arbitrary basis molecular Hamiltonians taking the form

$$H = \sum_{pq} T_{pq} a_p^\dagger a_q + \sum_{pqrs} V_{pqrs} a_p^\dagger a_q a_r^\dagger a_s$$

in depth scaling only as $O(N^2)$ on a linear array of qubits. First, we note that the one-body part of the above expression is easy to simulate using the techniques introduced in Circuits 1 (https://quantumai.google/openfermion/tutorials/circuits_1_basis_change). Thus, the real challenge is to simulate the two-body part of the operator.

We begin with the observation that the rank-4 tensor V, with the values V_{pqrs} representing the coefficient of $a_p^\dagger a_q a_r^\dagger a_s$ can be flattened into an $N^2 \times N^2$ array by making p, q one index and r, s the other. This is the electronic repulsion integral (ERI) matrix in chemist notation. We will refer to the ERI matrix as W. By diagonalizing W, one obtains $W g_\ell = \lambda_\ell g_\ell$ where the eigenvector g_ℓ is a vector of dimension N^2. If we reshape g_ℓ into an $N \times N$ vector, we realize that

$$\sum_{pqrs} V_{pqrs} a_p^\dagger a_q a_r^\dagger a_s = \sum_{\ell=0}^{L-1} \lambda_\ell \left(\sum_{pq} \left[g_\ell\right]_{pq} a_p^\dagger a_q \right)^2 .$$

This is related to the concept of density fitting in electronic structure, which is often accomplished using a Cholesky decomposition. It is fairly well known in the quantum chemistry community that the ERI matrix is positive semi-definite and despite having linear dimension N^2, has rank of only $L = O(N)$. Thus, the eigenvalues λ_ℓ are positive and there are only $O(N)$ of them.

Next, we diagonalize the one-body operators inside of the square so that

$$R_\ell \left(\sum_{pq} \left[g_\ell\right]_{pq} a_p^\dagger a_q \right) R_\ell^\dagger = \sum_p f_{\ell p} a_p^\dagger a_p$$

where the R_ℓ represent single-particle basis transformations of the sort we compiled in Circuits 1 (https://quantumai.google/openfermion/tutorials/circuits_1_basis_change). Then,

$$\sum_{\ell=0}^{L-1} \lambda_\ell \left(\sum_{pq} \left[g_\ell\right]_{pq} a_p^\dagger a_q \right)^2 = \sum_{\ell=0}^{L-1} \lambda_\ell \left(R_\ell \left(\sum_p f_{\ell p} a_p^\dagger a_p \right) R_\ell^\dagger \right)^2 = \sum_{\ell=0}^{L-1} \lambda_\ell \Bigg(R_\ell \Bigg(\sum$$

We now see that we can simulate a Trotter step under the arbitrary basis two-body operator as

$$\prod_{\ell=0}^{L-1} R_\ell \exp\left(-i \sum_{pq} f_{\ell p} f_{\ell q} a_p^\dagger a_p a_q^\dagger a_q\right) R_\ell^\dagger$$

where we note that the operator in the exponential take the form of a diagonal Coulomb operator. Since we can implement the R_ℓ circuits in $O(N)$ depth (see Circuits 1 (https://quantumai.google/openfermion/tutorials/circuits_1_basis_change)) and we can implement Trotter steps under diagonal Coulomb operators in $O(N)$ layers of gates (see Circuits 2 (https://quantumai.google/openfermion/tutorials/circuits_2_diagonal_coulomb_trotter)) we see that we can implement Trotter steps under arbitrary basis electronic structure Hamiltionians in $O(LN) = O(N^2)$ depth, and all on a linearly connected device. This is a big improvement over the usual way of doing things, which would lead to no less than $O(N^5)$ depth! In fact, it is also possible to do better by truncating rank on the second diagonalization but we have not implemented that (details will be discussed in aforementioned paper-in-preparation).

Note that these techniques are also applicable to realizing evolution under other two-body operators, such as the generator of unitary coupled cluster. Note that one can create variational algorithms where a variational parameter specifies the rank at which to truncate the λ_ℓ.

Example implementation: Trotter steps of LiH in molecular orbital basis

We will now use these techniques to implement Trotter steps for an actual molecule. We will focus on LiH at equilibrium geometry, since integrals for that system are provided with every OpenFermion installation. However, by installing OpenFermion-PySCF (https://github.com/quantumlib/OpenFermion-PySCF) or OpenFermion-Psi4 (https://github.com/quantumlib/OpenFermion-Psi4) one can use these techniques for any molecule at any geometry. We will generate LiH in an active space consisting of 4 qubits. First, we obtain the Hamiltonian as an InteractionOperator.

```
import openfermion

# Set Hamiltonian parameters for LiH simulation in active space.
diatomic_bond_length = 1.45
geometry = [('Li', (0., 0., 0.)), ('H', (0., 0., diatomic_bond_length))]
```

```
basis = 'sto-3g'
multiplicity = 1
active_space_start = 1
active_space_stop = 3

# Generate and populate instance of MolecularData.
molecule = openfermion.MolecularData(geometry, basis, multiplicity, descripti
molecule.load()

# Get the Hamiltonian in an active space.
molecular_hamiltonian = molecule.get_molecular_hamiltonian(
    occupied_indices=range(active_space_start),
    active_indices=range(active_space_start, active_space_stop))
print(openfermion.get_fermion_operator(molecular_hamiltonian))
```

```
-6.7698132180879735 [] +
-0.7952726864779313 [0^ 0] +
0.24889540266275176 [0^ 0^ 0 0] +
-0.02307282640154995 [0^ 0^ 0 2] +
-0.023072826401549944 [0^ 0^ 2 0] +
0.005865992881900444 [0^ 0^ 2 2] +
0.24889540266275176 [0^ 1^ 1 0] +
-0.02307282640154995 [0^ 1^ 1 2] +
-0.023072826401549944 [0^ 1^ 3 0] +
0.005865992881900444 [0^ 1^ 3 2] +
0.04614563473199314 [0^ 2] +
-0.02307282640154995 [0^ 2^ 0 0] +
0.005865992881900456 [0^ 2^ 0 2] +
0.11412688446849813 [0^ 2^ 2 0] +
```

We see from the above output that this is a fairly complex Hamiltonian already. Next we will use the **`simulate_trotter`** function from Circuits 1 (https://quantumai.google/openfermion/tutorials/circuits_1_basis_change), but this time using a different type of Trotter step associated with these low rank techniques. To keep this circuit very short for pedagogical purposes we will force a truncation of the eigenvalues λ_ℓ at a predetermined value of **`final_rank`**. While we also support a canned **`LOW_RANK`** option for the Trotter steps, in order to pass this value of **`final_rank`** we will instantiate a custom Trotter algorithm type.

```
import cirq
import openfermion
from openfermion.circuits import trotter

# Trotter step parameters.
```

```
time = 1.
final_rank = 2

# Initialize circuit qubits in a line.
n_qubits = openfermion.count_qubits(molecular_hamiltonian)
qubits = cirq.LineQubit.range(n_qubits)

# Compile the low rank Trotter step using OpenFermion.
custom_algorithm = trotter.LowRankTrotterAlgorithm(final_rank=final_rank)
circuit = cirq.Circuit(
    trotter.simulate_trotter(
            qubits, molecular_hamiltonian,
            time=time, omit_final_swaps=True,
            algorithm=custom_algorithm),
    strategy=cirq.InsertStrategy.EARLIEST)

# Print circuit.
cirq.drop_negligible_operations(circuit)
print(circuit.to_text_diagram(transpose=True))
```

```
0              1                    2                 3
│              │                    │                 │
Rz(π)          Rz(π)                Rz(π)             Rz(π)
│              │                    │                 │
│              PhISwap(0.25)────────PhISwap(0.25)^-1  │
│              │                    │                 │
PhISwap(0.25)──PhISwap(0.25)^0.081  Z^0               │
│              │                    │                 │
Rz(0.337π)     Z^0                  PhISwap(0.25)─────PhISwap(0.25)^-0.081
│              │                    │                 │
Rz(0)          PhISwap(0.25)────────PhISwap(0.25)^-1  Z^0
│              │                    │                 │
│              Rz(0.337π)           Z^0               Rz(0.169π)
│              │                    │                 │
```

We were able to print out the circuit this way but forcing `final_rank` of 2 is not very accurate. In the cell below, we compile the Trotter step with full rank so $L = N^2$ and depth is actually $O(N^3)$ and repeat the Trotter step multiple times to show that it actually converges to the correct result. Since we are not forcing the rank truncation we can use the built-in `LOW_RANK` Trotter step type. Note that the rank of the Coulomb operators is asymptotically $O(N)$ but for very small molecules in small basis sets only a few eigenvalues can be truncated.

```
# Initialize a random initial state.
import numpy
random_seed = 8317
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(molecular_hamiltonian)
exact_state = scipy.sparse.linalg.expm_multiply(
    -1j * time * hamiltonian_sparse, initial_state)

# Trotter step parameters.
n_steps = 3

# Compile the low rank Trotter step using OpenFermion.
qubits = cirq.LineQubit.range(n_qubits)
circuit = cirq.Circuit(
    trotter.simulate_trotter(
            qubits, molecular_hamiltonian,
            time=time, n_steps=n_steps,
            algorithm=trotter.LOW_RANK),
    strategy=cirq.InsertStrategy.EARLIEST)

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

# Print final fidelity.
fidelity = abs(numpy.dot(simulated_state, numpy.conjugate(exact_state))) ** 2
print('Fidelity with exact result is {}.\n'.format(fidelity))

# Print circuit.
cirq.drop_negligible_operations(circuit)
print(circuit.to_text_diagram(transpose=True))
```

```
Fidelity with exact result is 0.9999960568889797.

0           1                     2                3
│           │                     │                │
Rz(π)       Rz(π)                 Rz(π)            Rz(π)
│           │                     │                │
│           PhISwap(0.25)─────────PhISwap(0.25)^-1 │
│           │                     │                │
```

```
PhISwap(0.25)─PhISwap(0.25)^0.081   Z^0             │
│             │                     │               │
Rz(0.112π)    Z^0                   PhISwap(0.25)───PhISwap(0.25)^-0.081
│             │                     │               │
Rz(0)         PhISwap(0.25)─────────PhISwap(0.25)^-1 Z^0
│             │                     │               │
```

Last updated 2023-10-27 UTC.