This document is the start of the quantum hackathon document

Circuit Synthesis mechanism
We did some study into t|ket> quantum compiler internal coding implementation. https://github.com/CQCL/tket/discussions/485 A lot more needs to be done to study the theoretical intuition of circuit synthesis algorithms as well as the effect of different synthesis algorithms on the different performance (noise, number of qubits, depth) metric of the generated quantum circuit.

Python implementation and results:

The main Qupolis notebook and VQE have been executed on QBraid using Kernel qBraid-SDK. The Azure simulations (Azure file) had to be done on Anaconda with Python 3 Kernel.

We have run the H2 Jordan Wigner Mapper qasm file on different backends.
First, we ran it on IBMQ EmulatorBackend and observed the values of these three parameters:

1. Circuit depth
2. CX gates depth
3. CX gates count

Initially, without any optimization process, these values were:

1. Circuit depth = 83
2. CX gates depth = 52
3. CX gates count = 56

These parameters can be measured through *depth(), depth_by_type(OpType.CX)),* and *n_gates_of_type(OpType.CX))*.

Then, we compiled the qasm circuit with an optimization factor = 2 using *get_compiled_circuit()* so the circuit could be executed by the backend. We observed that the 3 parameters defining depth and count, decreased. *Circuit_from_qasm* was used to import the circuit of H2 Jordan Wigner Mapper from the qasm file and convert it to a circuit which can be displayed clearly through *render_circuit_jupyter* (pytket tools).

We experimented by optimizing the circuit further and implementing *FullpeepholeOptimise()*, which lowered the depth and CX count even further. We have done this process for every backend IBMQ EmulatorBackend, IonQ simulator, 29 qubits (AWS Braket), Rigetti Aspen-11 (AWS Braket), IBM AerUnitaryBacked, IonQ (Azure), and Quantinuum emulator quantinuum.hqs-lt-s1-sim (Azure).

We summarize our results in the following.

	IBMQ EmulatorBackend	IonQ (AWS Braket)	Rigetti Aspen-11 (AWS Braket)	IBM AerUnitaryBackend
1.circuit depth 2.cx depth 3.cx count, after compilation	50 19 19	41 16 16	41 17 19	31 17 17
1.circuit depth 2.cx depth 3.cx count, after PeepholeOptimise	28 16 16	30 16 16	30 16 16	31 17 17
Comments				*no change after PeepholeOptimise

	IonQ (Azure)	Quantinuum (Azure)
1.circuit depth 2.cx depth 3.cx count, after compilation (opt=2)	99 33 34	99 33 34
1.circuit depth 2.cx depth 3.cx count, after PeepholeOptimise	43 21 22	66 33 34
1.circuit depth 2.cx depth 3.cx count, 2xPeepholeOptimise	43 21 22	46 25 26

From the tables above, IBM simulator and devices accessed via AWS Braket achieve a good improvement in gate depth and count values after a compilation/optimization=2 and after applying FullPeepholeOptimise() once. IBM Unitary Backend achieves those values just by compilation.

In the Azure case we observe some interesting things. E.g., IonQ achieves different depth/count values after being compiled/optimized through AWS Braket, versus Azure. Furthermore, both IonQ and Quantinuum need two rounds of PeepholeOptimise. IonQ achieves slightly better

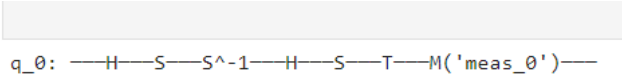
results than Quantinuum although the values vary with the simulation, We can consider other factors such as price (later) and results accuracy. For example, the IonQ simulator produces the correct output “1010” with 100% whereas Quantinuum with 90%.
The IBM emulator on the other hand, achieves lower percentages in generating the accurate output “1010”. For *ibmq_belem* we measured a value of 60% for the correct output (it may vary after each simulation).

Challenges and issues we encountered so far:
The Azure simulations had to be done on Anaconda, because in QBraid they cause a Timeout error.
Also, displaying the counts on AWS Brakend often gives a timeout error, whereas in Azure it runs smoothly. In both, it takes a long time to run real devices, so we have been working with simulators.

Quantum Error Mitigation

We created an Identity-equivalent circuit using Qasm language for the circuit, and Cirq for the mitigation. The reason behind this is that the Mitiq error mitigation methods gave us errors in Qiskit, whereas in Cirq they ran fine. This way we learned more about QASM, Circ, and the Zero Noise Extrapolation method in calculating an approximate of the ideal noiseless expectation value of a variable, from a set of noisy expectation values, where noise is amplified on purpose so that the ideal expectation value can be found through extrapolation.

For a given number of shots, the output value of |0> corresponds to an expectation value of 1. In reality, if we introduce noise, this expectation value will be lower than 1, which means that not every output out of the shots will be |0>. Through the ZNE method, we managed to improve the expectation value. The importance of this method is the possibility of applying it to chemistry circuits, including LiH and H2 Jordan Wigner Mappers. Also, other methods can be implemented like Probabilistic Error Cancellation.



HSSdagHST = Identity

Results of ZNE method.

Default (Richardson Extrapolation Technique):
Unmitigated expectation result 0.652
Mitigated expectation result 0.807

Linear extrapolation with five different (noise) scale factors [1.1, 1.4, 2.1, 2.4, 3.0]:
Mitigated result 0.745

Scale noise by global folding:
Mitigated result 0.817

FermionicOp_to_Qubit Mapping
-> Jordan-Wigner (JW)
-> Parity
-> Bravyi-Kitaev (BK)
-> Bravyi-Kitaev-SuperFast (BKSF)

The gate count to implement BKSF is lower than JW but higher than BK.
JW mapping requires O(N) qubit operations to simulate one electronic operation, while BK mapping only requires just O(log N) qubit operations to simulate one electronic operation.

The theory and derivation for both JW and BK mappers could be found here at A Comparison of the Bravyi–Kitaev and Jordan–Wigner Transformations for the Quantum Simulation of Quantum Chemistry . We only managed to derive the maths proof for the JW mapper which is about anticommute math properties.

When two operators A and B anticommute it means that the following equality holds

$AB = -BA$

This means that if I want to swap the ordering of a product of A and B in some expression I need to multiply by -1 to have the value stay the same. This is something I need to remember to do if I want to manipulate an expression by moving operators around in a product.

This is exactly what we are doing with the a_k^+ and a_1^+ in the following maths proof for JW mapper, which anticommute when k is not 1. I want to move a_k^+ through the product until it reaches the k^{th} place because then I can get rid of it by absorbing it into the $(a\_k^+)^{n-k}$ to get $(a\_k^+)^{n-k+1}$ but to make sure I don't accidentally change the value of the expression I need to include a -1 phase every time I swap it past a creation operator of a different index.

Systems made up of many fermions can essentially be described by a set of states those fermions can be in (often called modes or orbitals) and how many fermions are in each state. An example would be the set of electron orbitals around an atom. You can then describe the state of a many fermion system in terms of the "occupation number" of these states, which is simply how many fermions are occupying each state. Say I have a system where there are 3 states that fermions can exist in and I have one fermion occupying the second state and no others, then I could write the state as

$|0, 1 ,0\rangle$

which is called a Fock State or a Number State. The system can also exist in superposition between many Number States so if the fermion was in the middle of transitioning from the 2nd mode to the 3rd mode then the state could be written like

(|0, 1 ,0> + |0, 0, 1>) / sqrt(2)

in fact, any state of the many fermion systems can be described by number states or superpositions of number states and number states are simply the different combinations of the occupation number for each fermion state[1] .

Due to the Pauli Exclusion Principle, only 1 fermion at a time can be in a given mode so the occupation number for any mode can only be 0 or 1. This means that for a many fermion system with N modes, there are 2^N possible number states and the system has a dimension of 2^N as the number states are orthogonal.

Footnotes:

1. To be precise, number states are defined with respect to a chosen set of orthonormal fermion states, a basis. Different bases can be chosen and a number state w.r.t. one bases may be a superposition of number states w.r.t. another basis.

We find a good way to derive the JW transform is to consider the effect of a creation operator on some arbitrary fermionic occupation state, by which I mean a state where a series of modes are either occupied or unoccupied.

Consider a state of a fermionic system with N modes which are either occupied or unoccupied, label this

|n_1,n_2,...,n_N>

where n_k is the occupation number of mode k (n_k can be 0 or 1). We can define this state in terms of creation operators acting on the fully unoccupied (vacuum) state in a chosen order

$|n_1,n_2,...,n_N\rangle = (a_1^+)^{n_1} (a_2^+)^{n_2} ... (a_N^+)^{n_N} |0,0,...,0\rangle$

here the creation operator a_k^+ is raised to the power of the occupation number n_k, so if n_k=1 it is applied and the mode is filled and if n_k=0 it is not and the mode stays empty.

Now, what happens if we apply a creation operator a_k^+ to the above state? Lets start by writing it down

$a_k^+ (a_1^+)^{n_1} ... (a_N^+)^{n_N} |0,0,...,0\rangle$

If we swap the order of the a_k^+ and the $(a_1^+)^{n_1}$ we need to multiply the product by $(-1)^{n_1}$, because a_k^+ and a_1^+ anticommute. So we have the equivalent expression

$= (-1)^{n_1} (a_1^+)^{n_1} a_k^+ ... (a_N^+)^{n_N} |0,0,...,0\rangle$

Keep swapping the a_k^+ through until you get to the kth place in the product, you'll pick up minus signs in the same manner until you have

$= P (a_1^+)^{n_1} ... (a_k^+)^{n_{k+1}} ... (a_N^+)^{n_N} |0,0,...,0>$

where $P = (-1)^{n_1} (-1)^{n_2} ... (-1)^{n_{k-1}}$.

Recall how the phase factor P above is simply the product of (-1)'s for every occupied mode before k? If you multiply -1 by itself an odd number of times you get -1, an even number of times you get +1. So the parity of the total occupation (sum of occupation numbers) of modes below k decides whether P is +1 or -1.

So we see in the case where $n_k=1$, applying a_k^+ yields 0 (creation ops square to 0) and in the case where $n_k=0$, it fills the mode and multiplies the state by phase factor P which depends on the occupation states of all modes before k in the ordering we chose.

Now we want to define a qubit operator which has the same effect when you associate k^{th} mode occupation with k^{th} qubit state. The part of the action where it multiplies an occupied state by 0 and fills an unoccupied state can be done by the operator

$\sigma^+ = X - iY$

on the k^{th} qubit. To get the phase factor P you have to also apply Z to every qubit with an index lower than k, as that will apply a -1 phase if it's in the 1 ("occupied") state, just as in P. So then the qubit version of the creation operator on mode k will be

$Z_1 Z_2 ... Z_{k-1} (X - iY)_k$

This is the Jordan-Wigner representation.

Besides, we also found that there is some coding that performs Jordan-Wigner to Bravyi-Kitaev transformation in public domain which is not yet implemented in OpenFermion github repo. We have attempted to understand the logic behind the transformation in this user-contributed coding, we also contacted the authors of the OpenFermion github repo to help with this code inspection, and possible code incorporation into github.

ZZ_Hamiltonian
As highlighted in Digital-Analog Quantum Computation paper , ZZ_Hamiltonian is a more {hardware or qubit}-efficient approach compared to using FermionicOp which will increase runtime for the QubitConverter() which is an unnecessary overhead.

VQE implementation

For the VQE, we used the Parity Mapper because it offers the possibility to reduce the number of qubits by 2. We used various modules from qiskit_nature.
We ran VQE on two backends: a noiseless emulator from IBM, and then we introduced a noise model from an IBM device with mitigation.

The known solution for the energy of the ground state is -1.137306035753 (Hartree).

We summarize in a table the different energy values of the electronic ground state and total ground state, found by VQE for different types of ansatz used, when we ran the code using an IBM device with noise model and error mitigation.

	Numpy Exact Solver	Two-Local circuit, SLSQP optimizer	Heuristic ansatz, QN-SPSA optimizer
Electronic ground state energy (Hartree)	-1.857275030202	-1.587190530127	-1.809530046343
Total ground state energy (Hartree)	-1.137306035753	-0.867221535678	-1.089561051894

Let us build the table again when we run a noiseless IBM backend, with no noise model introduced:

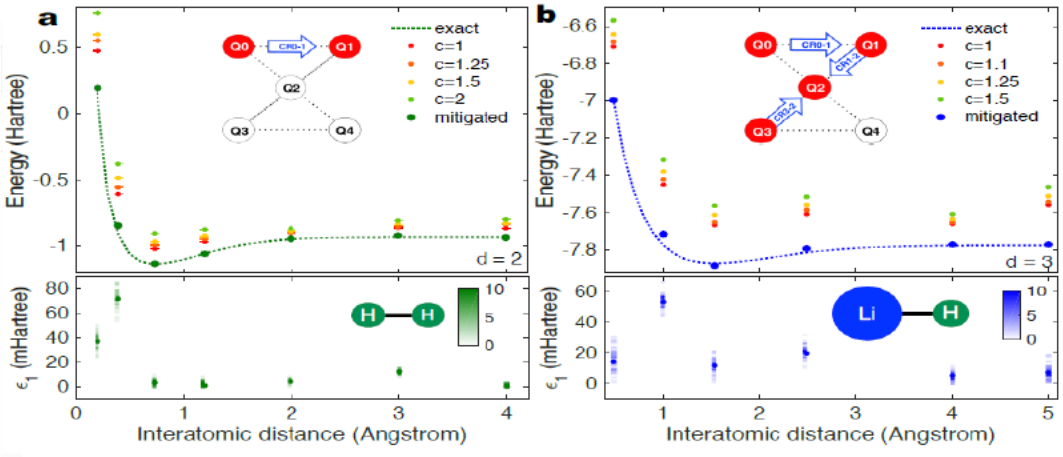
	Numpy Exact Solver	Two-Local circuit, SLSQP optimizer	Heuristic ansatz, QN-SPSA optimizer
Electronic ground state energy (Hartree)	-1.857275030202	-1.857274651021	-1.857265202892
Total ground state energy (Hartree)	-1.137306035753	-1.137305656572	-1.137296208443

Here we observe that for the noiseless backend, the values are closer to the exact numpy solver energy values for the electronic ground state energy, and total ground state energy. Thus, less noise will lead to better approximations in the ground state energy values.

Future perspectives

Considering the skills and the knowledge that we acquired, the next steps for us are to experiment further with other types of compilers and optimizers, such as Transpile, PytChem, FullReduce, and find those combinations which lead to the best optimization in terms of depth and gate count. We already started a combination of get_compiled_circuit with Peephole, which advanced the reduction of depth and gate count. We are curious to find more combinations with other tools.

Another exciting area is of course error mitigation. We experimented with an identity gate equivalent circuit where we applied the Zero Noise Extrapolation method to find the approximate noiseless expectation value. The advantage of ZNE is that it doesn't depend on the nature of noise, but the noise must be constant in time. Another technique Probabilistic Error Cancellation, which tolerates types of noise that change with time, however it requires a good knowledge on the noise model. There is also another recent mitigation technique, the Accreditation protocol and post-selection, which post-selects those circuits whose noise is below a certain threshold. This technique can deal with time-dependent noises and a broad range of noises. Combined with the twp previous techniques that we mentioned, it has the potential to bring advantages in error mitigation, finding applications in VQE and quantum chemistry. In this graph, the authors Kristan Temme, Sergey Bravyi, Jay M. Gambetta have applied the ZNE mitigation technique to receive better accuracy in the variational eigensolver applied to electronic structures of H2 and LiH molecules.



We are invested in theoretical research in the direction of quantum error mitigation, with a recently written report on error mitigation as part of our PhD studies, and would like to take our knowledge further to concrete industrial applications.
https://drive.google.com/file/d/1_tYVu6R9DaZKWJie1tyLULbXY-V63aEu/view

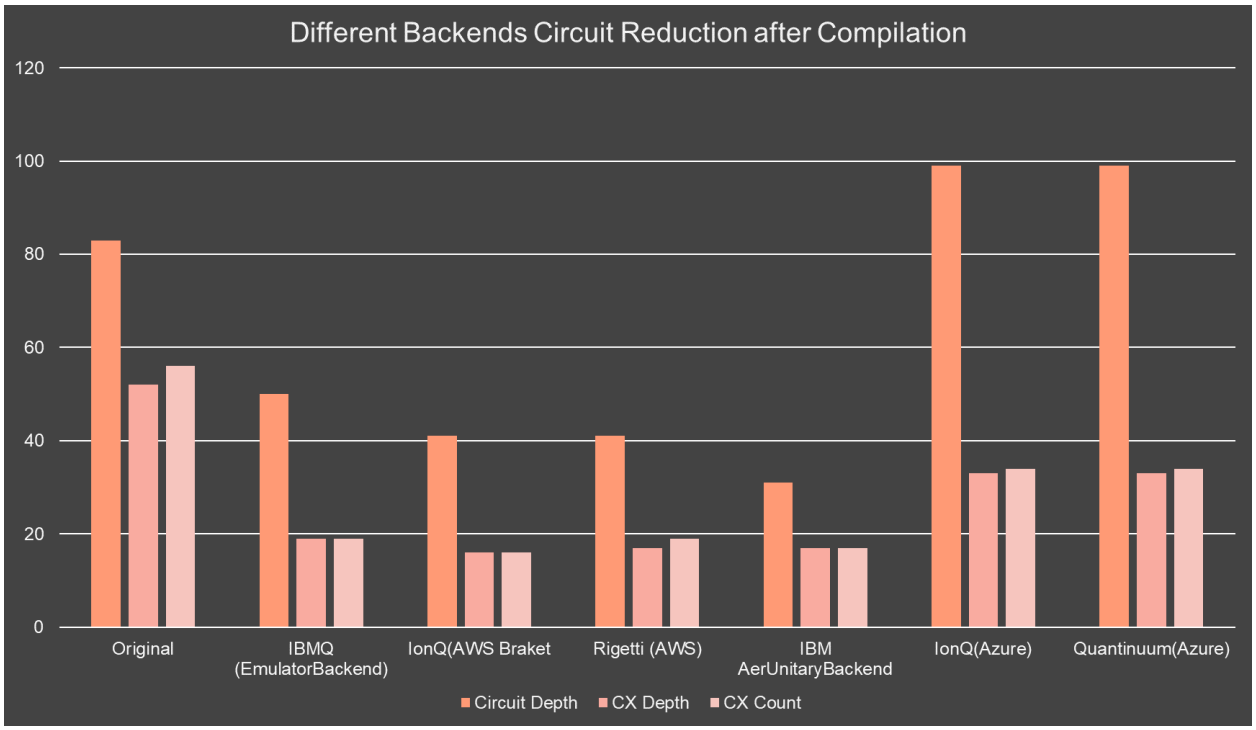
In the quantum chemistry domain, books like Quantum Chemistry and Computing (A. Chance, K.L. Sharkey), and various courses and summer schools, are a real treasure to help us learn more and carry our own projects.
https://drive.google.com/file/d/1_tYVu6R9DaZKWJie1tyLULbXY-V63aEu/view

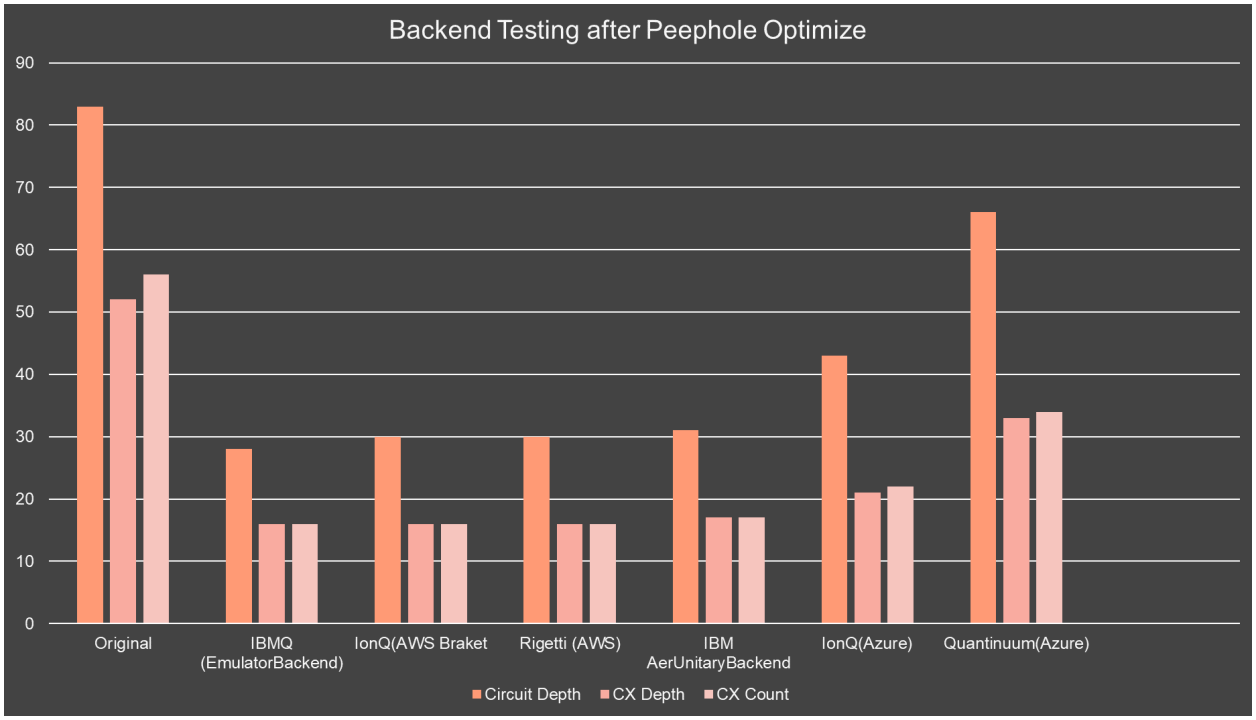
This is also a cost based analysis of which backend would be the most cost efficient to implement. We can confidently say that IonQ(AWS) , while not being the cheapest of the selection of backend implementations, makes up for that by giving 100% “1010” output. So we would recommend using IonQ (AWS) moving forward.

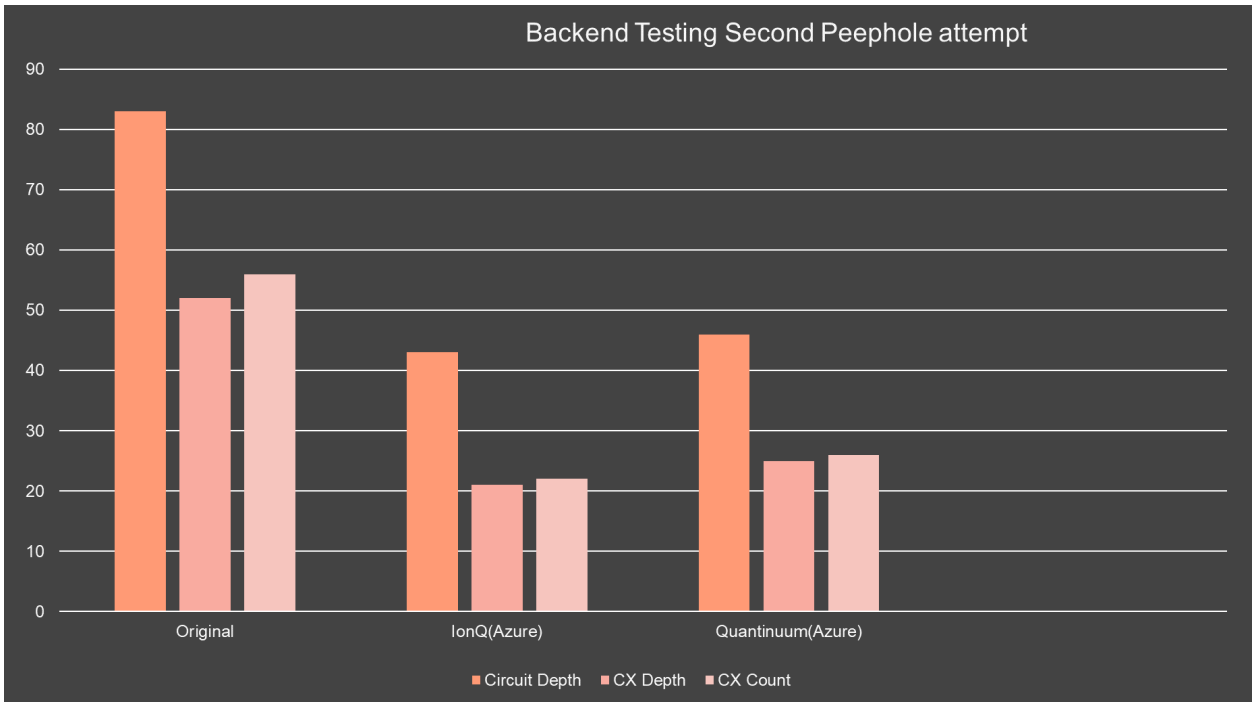
Case number	Backend	Time taken	Fidelity	Optimization process	Price Per Shot	Total cost
1	D-Wave				0.00019	0.0019
2	IonQ (AWS)				0.01000	0.10
3	Rigetti				0.00035	0.0035
4	Quantinium					See Subscription details
5	IBM-Q A				0.00098	0.0098
6	IBM-Q H				0.0003	0.003

Quantinium Pricing:

Pricing	**Standard-$125,000 (USD)/Month + Azure infrastructure costs** **Premium -$175,000 (USD)/Month + Azure infrastructure costs**
Standard	· 10k HQCs for use on the System Model H1 hardware · 100k eHQCs for use on the System Model H1 Emulator
Premium	● 17k HQCs for use on System Model H1 hardware ● 170k eHQCs for use on the System Model H1 Emulato




Backend Testing after Peephole Optimize
90
80
70
60
50
40
30
20
10
0
Original
IBMQ (EmulatorBackend)
IonQ(AWS Braket
Rigetti (AWS)
IBM AerUnitaryBackend
IonQ(Azure)
Quantinuum(Azure)
Circuit Depth
CX Depth
CX Count


Backend Testing Second Peephole attempt
90
80
70
60
50
40
30
20
10
0
Original
IonQ(Azure)
Quantinuum(Azure)
Circuit Depth
CX Depth
CX Count