ECE 592 / CSC 591

Project M1: DAX Transpiler

Final Report

Keith Mellendorf
Alexander Allen

https://github.ncsu.edu/arallen4/qc-ion-trap

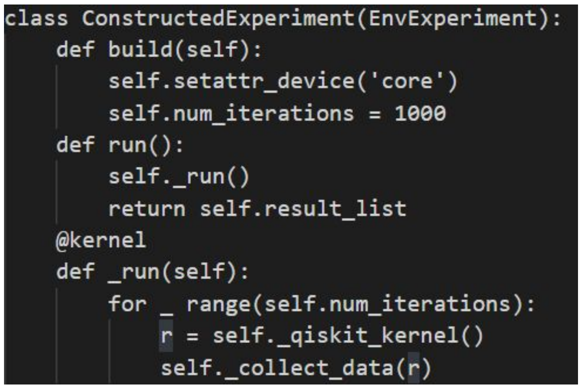
Features

Basic translation

As we showed in our previous report, basic translation of qiskit circuits to DAX code has been achieved, and is shown extensively below in the test cases section.

Shots

The shots argument in the execute function has been implemented to indicate how many times the DAX code should run the Quantum Kernel. We chose to implement it in the parameter type view, of self.num_iterations, as shown in the image below. The maximum quantity of iterations is an easily configurable number, and is currently set to 1024.



```
class ConstructedExperiment(EnvExperiment):
    def build(self):
        self.setattr_device('core')
        self.num_iterations = 1000
    def run():
        self._run()
        return self.result_list
    @kernel
    def _run(self):
        for _ range(self.num_iterations):
            r = self._qiskit_kernel()
            self._collect_data(r)
```

Scheduling

ARTIQ requires explicit scheduling of individual qubit operations and DAX supports this through the constructs `with serial` and `with parallel`

We built a configurable scheduler that considers the resource consumption of a given operation in Qiskit and explicitly schedules the gates as necessary.

To do this, each possible gate in qiskit is assigned a time and a size in a configuration file resources.toml. Time being some integral number of time periods the gate needs to run and size being some integral number of resources required by the gate operation.

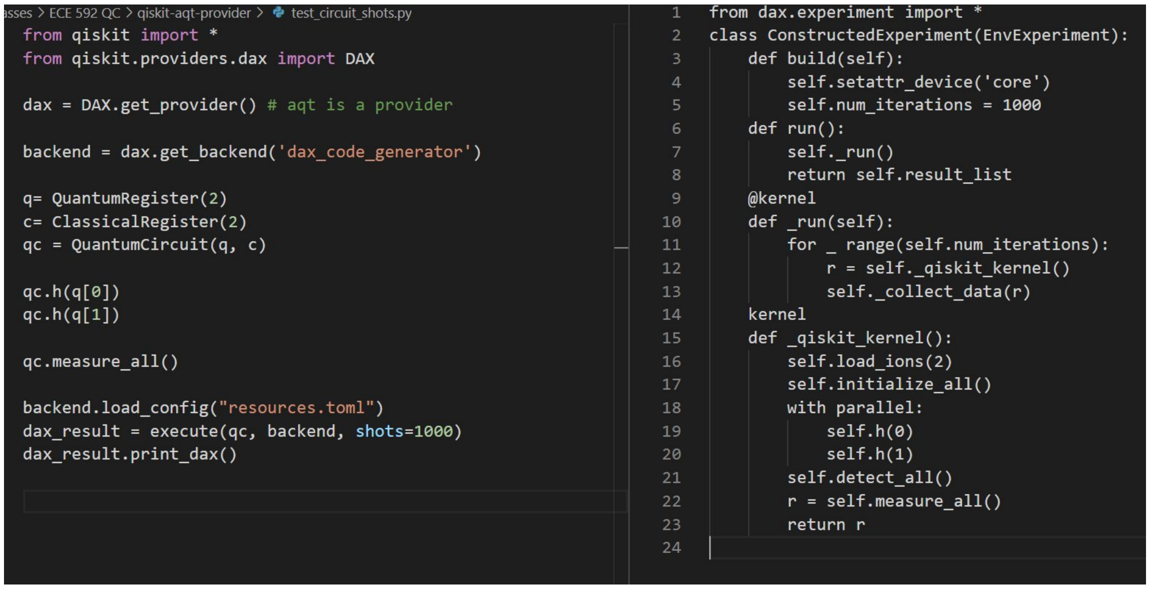
Also defined in the configuration file is an overall capacity for the quantum computer. This limits the number of simultaneous gates based on their individual sizes. We assume, for now, that the size of the gate is equal to the number of qubits the gate operates on and that the machine as a whole can operate on around five qubits at once.

The scheduler is also constrained such that a qubit cannot be operated on by multiple gates at the same time and such that instructions never occur out of order in such a way that the final result is impacted.

Test Cases

Shots

Basic program with customized shots equal to 1000. This value is shown in the num_iterations attribute on the right. That value is then used in a loop to run the quantum program 1000 times, collecting the results.



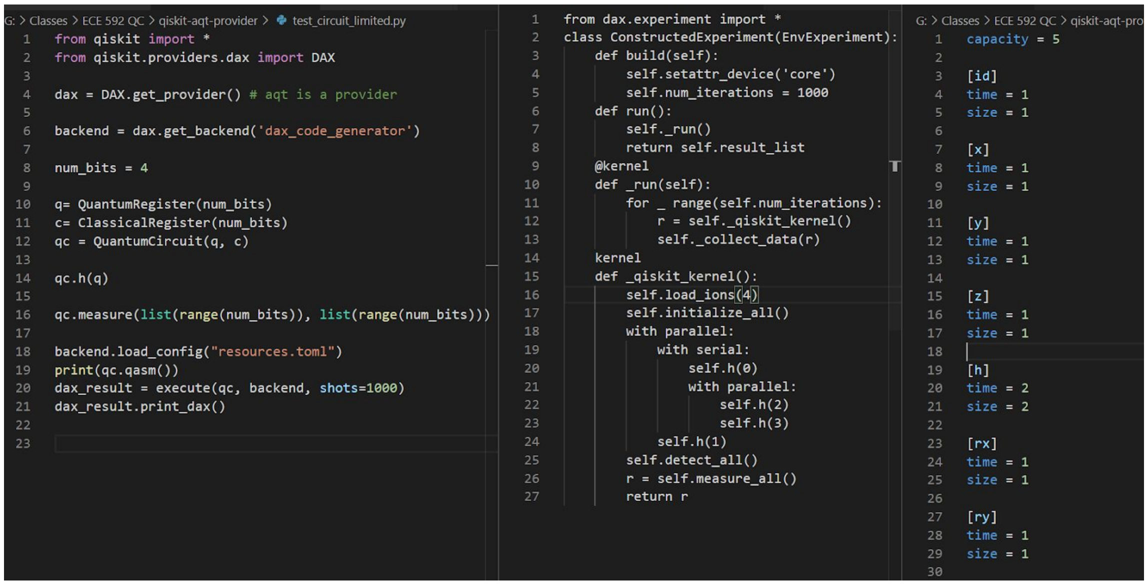
asses > ECE 592 QC > qiskit-aqt-provider > test_circuit_shots.py

```python
from qiskit import *
from qiskit.providers.dax import DAX

dax = DAX.get_provider() # aqt is a provider

backend = dax.get_backend('dax_code_generator')

q= QuantumRegister(2)
c= ClassicalRegister(2)
qc = QuantumCircuit(q, c)

qc.h(q[0])
qc.h(q[1])

qc.measure_all()

backend.load_config("resources.toml")
dax_result = execute(qc, backend, shots=1000)
dax_result.print_dax()
```

```python
from dax.experiment import *
class ConstructedExperiment(EnvExperiment):
    def build(self):
        self.setattr_device('core')
        self.num_iterations = 1000
    def run():
        self._run()
        return self.result_list
    @kernel
    def _run(self):
        for _ range(self.num_iterations):
            r = self._qiskit_kernel()
            self._collect_data(r)
    kernel
    def _qiskit_kernel():
        self.load_ions(2)
        self.initialize_all()
        with parallel:
            self.h(0)
            self.h(1)
        self.detect_all()
        r = self.measure_all()
        return r
```

Resource limited

This simple program shows what parallelism can be achieved while limited by the QC's capabilities. Shown on the far right is the resources.toml which indicates that .h() gates take 2 'ticks' to run, and have a size of 2. This size is described by the capacity value at the top of the .toml which indicates that the concurrent operations may only have a size at or below 5. This in turn means Hadamards can only be parallelized up to two at a time.



G: > Classes > ECE 592 QC > qiskit-aqt-provider > test_circuit_limited.py
G: > Classes > ECE 592 QC > qiskit-aqt-pro

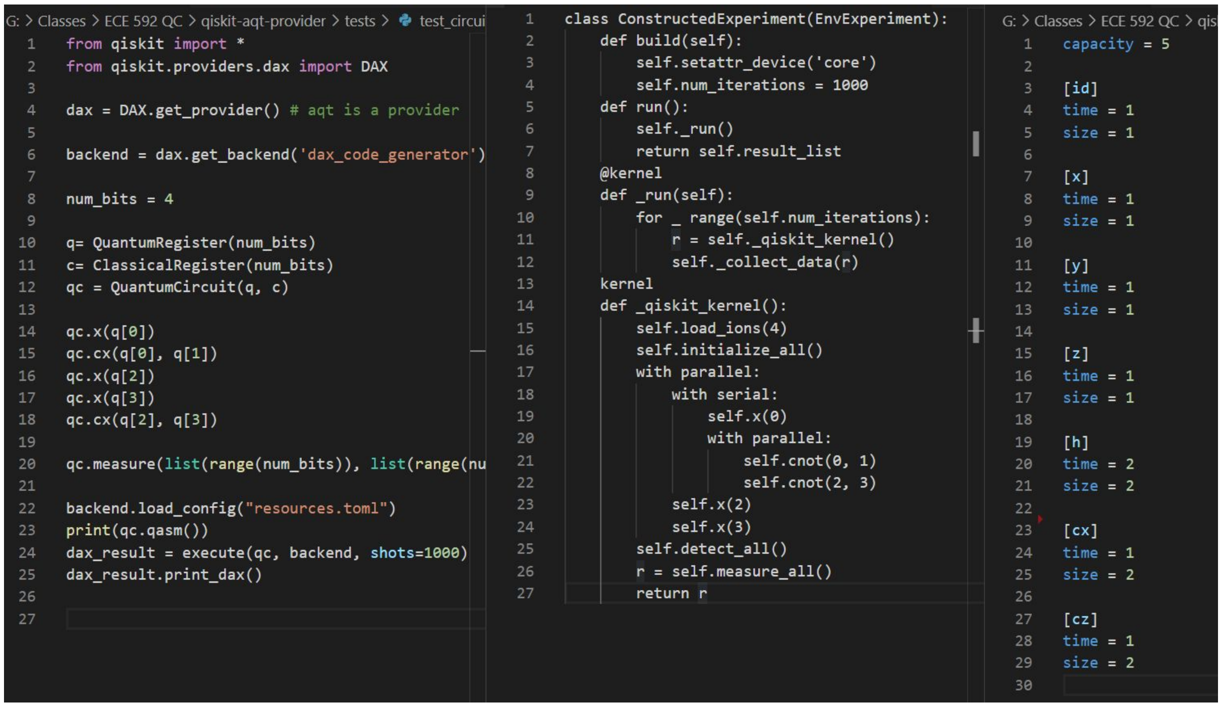
```python
from qiskit import *
from qiskit.providers.dax import DAX

dax = DAX.get_provider() # aqt is a provider

backend = dax.get_backend('dax_code_generator')

num_bits = 4

q= QuantumRegister(num_bits)
c= ClassicalRegister(num_bits)
qc = QuantumCircuit(q, c)

qc.h(q)

qc.measure(list(range(num_bits)), list(range(num_bits)))

backend.load_config("resources.toml")
print(qc.qasm())
dax_result = execute(qc, backend, shots=1000)
dax_result.print_dax()
```

```python
from dax.experiment import *
class ConstructedExperiment(EnvExperiment):
    def build(self):
        self.setattr_device('core')
        self.num_iterations = 1000
    def run():
        self._run()
        return self.result_list
    @kernel
    def _run(self):
        for _ range(self.num_iterations):
            r = self._qiskit_kernel()
            self._collect_data(r)
    kernel
    def _qiskit_kernel():
        self.load_ions(4)
        self.initialize_all()
        with parallel:
            with serial:
                self.h(0)
                with parallel:
                    self.h(2)
                    self.h(3)
            self.h(1)
        self.detect_all()
        r = self.measure_all()
        return r
```

```toml
capacity = 5

[id]
time = 1
size = 1

[x]
time = 1
size = 1

[y]
time = 1
size = 1

[z]
time = 1
size = 1

[h]
time = 2
size = 2

[rx]
time = 1
size = 1

[ry]
time = 1
size = 1
```

Out of Order

In this example the limitations on which gates are run and when is solely based on dependency and timing. In the resource.toml we see that all of our instructions take 1 ‘tick’ to run, which allows us to do this entire program in 2 ticks. Each x gate can happen in the first tick with no problems. The two CNOTs can then run in parallel on the second tick as they do not depend on each other.



G: > Classes > ECE 592 QC > qiskit-aqt-provider > tests > test_circui

```python
from qiskit import *
from qiskit.providers.dax import DAX

dax = DAX.get_provider() # aqt is a provider

backend = dax.get_backend('dax_code_generator')

num_bits = 4

q= QuantumRegister(num_bits)
c= ClassicalRegister(num_bits)
qc = QuantumCircuit(q, c)

qc.x(q[0])
qc.cx(q[0], q[1])
qc.x(q[2])
qc.x(q[3])
qc.cx(q[2], q[3])

qc.measure(list(range(num_bits)), list(range(nu

backend.load_config("resources.toml")
print(qc.qasm())
dax_result = execute(qc, backend, shots=1000)
dax_result.print_dax()
```

```python
class ConstructedExperiment(EnvExperiment):
    def build(self):
        self.setattr_device('core')
        self.num_iterations = 1000
    def run():
        self._run()
        return self.result_list
    @kernel
    def _run(self):
        for _ range(self.num_iterations):
            r = self._qiskit_kernel()
            self._collect_data(r)
    kernel
    def _qiskit_kernel():
        self.load_ions(4)
        self.initialize_all()
        with parallel:
            with serial:
                self.x(0)
                with parallel:
                    self.cnot(0, 1)
                    self.cnot(2, 3)
            self.x(2)
            self.x(3)
        self.detect_all()
        r = self.measure_all()
        return r
```

G: > Classes > ECE 592 QC > qis

```toml
capacity = 5

[id]
time = 1
size = 1

[x]
time = 1
size = 1

[y]
time = 1
size = 1

[z]
time = 1
size = 1

[h]
time = 2
size = 2

[cx]
time = 1
size = 2

[cz]
time = 1
size = 2
```

Future Work

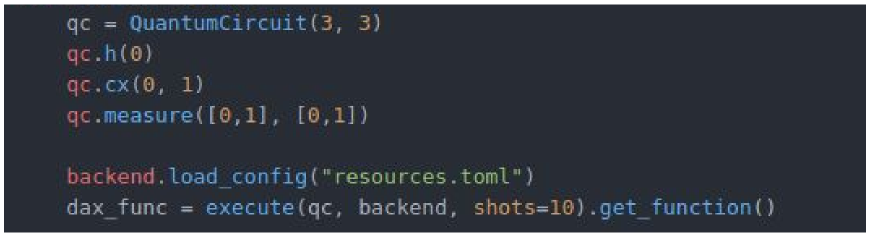
The main component we are currently missing in our qiskit provider are classical constructs.

The ARTIQ system supports running classical code on the FPGA, thus it is in our best interest to encode as much classical information into the quantum circuit encoded in DAX as possible for maximum efficiency. Otherwise, whenever a quantum computation completes, it has to call back to the host machine which has to upload a new kernel to ARTIQ dramatically increasing runtime.

However, Qiskit does not have good support for classical instructions within circuits. Thus, we propose the following extension to Qiskit and our provider to enable support for classical constructs within the DAX code our provider emits.

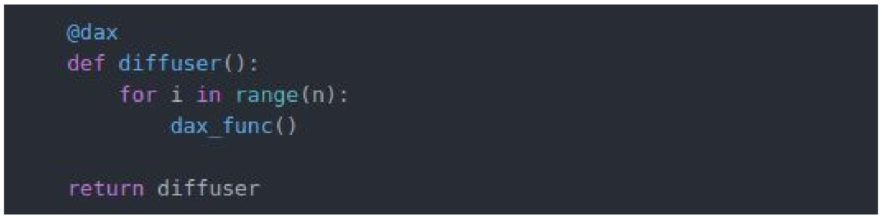
One extension is a “dax” function decorator and the other is a “kernel” quantum gate operation.

We can then construct a small quantum circuit, such as the diffuser in grover's algorithm, and tell qiskit to compile it as demonstrated in this document.



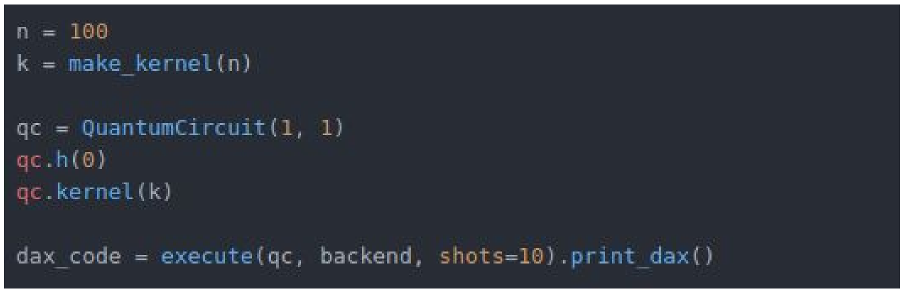
```
qc = QuantumCircuit(3, 3)
qc.h(0)
qc.cx(0, 1)
qc.measure([0,1], [0,1])

backend.load_config("resources.toml")
dax_func = execute(qc, backend, shots=10).get_function()
```

Then, we would take that compiled kernel and then embed it in a custom classical function with the dax decorator in order to integrate the classical operations (in this case repeating it n time as in grovers).



```
@dax
def diffuser():
    for i in range(n):
        dax_func()

return diffuser
```

Then we can simply apply this classical function to a quantum circuit just like a gate and re-execute the circuit to get a final, optimized and integrated DAX kernel.



```
n = 100
k = make_kernel(n)

qc = QuantumCircuit(1, 1)
qc.h(0)
qc.kernel(k)

dax_code = execute(qc, backend, shots=10).print_dax()
```

This allows the developer to use native python constructs to develop their classical kernel operations just as in DAX without compromising the ease or efficiency of their hardware specific qubit operations

The final DAX output is exactly what you would expect from a manually generated kernel seeking to achieve the same goal as the above code.