ESILV

myQLM

Gaëtan Rubez, PhD
Senior expert in Quantum Computing
2022

pyAQASM

pyAQASM – Quantum measurements

- One of the postulate of Quantum mechanics provides a means for describing the effects of measurements on quantum systems.
- So far, we have been measuring our states in the computational basis.
- Meaning that on a single qubit ($|\psi> = \alpha|0> + \beta|1>$) we can obtain two possible outcomes with this measurements:
  - 0 with probability $|\alpha|^2$
  - 1 with probability $|\beta|^2$
- This is acheived by using two measurement operators:

$$M_0=|0><0| \text{ and } M_1=|1><1|$$

- Due to p(0) $= <\psi|M_0^\dagger M_0|\psi> = <\psi|M_0|\psi> = |\alpha|^2$
- After measurement, if we obtained 0, the state is $\frac{\alpha}{|\alpha|}|0>$.

pyAQASM – Observable

- Observable corresponds to a Hermitian operator:

$$\mathrm{A} = A^{\dagger} = \mathrm{A}^{\mathrm{T}*}$$

- On the QLM we can express an observable as the combination of pauli terms (σ_X, σ_Y and σ_Z, named X, Y and Z)

- The goal in myQLM is to provide to users a framework to efficiently deal with observable sampling.

pyAQASM – Observable

```
from qat.core import Observable, Term

my_observable = Observable(4,
                           pauli_terms=[
                               Term(1., "ZZ", [0, 1]),
                               Term(4., "XZ", [2, 0]),
                               Term(3., "ZXZX", [0, 1, 2, 3])
                           ],
                           constant_coeff=23.)

print(my_observable)
```

pyAQASM – Observable

```python
from qat.core import Observable, Term

my_observable = Observable(4,
                    pauli_terms=[
                        Term(1., "ZZ", [0, 1]),
                        Term(4., "XZ", [2, 0]),
                        Term(3., "ZXZX", [0, 1, 2, 3])
                    ],
                    constant_coeff=23.)

print(my_observable)
```

```
23.0 * I^4 +
1.0 * (ZZ|[0, 1]) +
4.0 * (XZ|[2, 0]) +
3.0 * (ZXZX|[0, 1, 2, 3])
```

Atos

pyAQASM – Observable

```
…

job = circuit.to_job(observable= my_observable)

Results = qpu.submit(job)

…
```

The sampling of our observable on the final state produced by a quantum circuit.

pyAQASM – Parametrized circuit

```
from qat.lang.AQASM import *
prog = Program()
#Define your variables
theta = prog.new_var(float, "\\theta")

#Apply a gate with a variable
prog.apply(RY(theta), qubits_reg[0])

#Create and display the circuit
circuit = prog.to_circ()
%qatdisplay circuit
```

pyAQASM – Parametrized circuit

```
from qat.lang.AQASM import *
prog = Program()
#Define your variables
theta = prog.new_var(float, "\\theta")
gamma = prog.new_var(float, "\\gamma")

#Apply a gate with a variable
prog.apply(RY(3 * theta * gamma +1), qubits_reg[0])

#Create and display the circuit
circuit_tetha_gamma = prog.to_circ()
%qatdisplay circuit_tetha_gamma
```

pyAQASM – Bind variables

```
new_circuit = circuit.bind_variables({"\\theta": 0.5})
%qatdisplay new_circuit
```

q[0] – RY[0.5] –

```
new_circuit = circuit_tetha_gamma.bind_variables({"\\theta": 0.5})
%qatdisplay new_circuit
```

q[0] — $\mathrm{RY}[3 * 0.5 * \gamma + 1]$ —

```
new_circuit = new_circuit.bind_variables({"\\gamma": 0.5})
%qatdisplay new_circuit
```

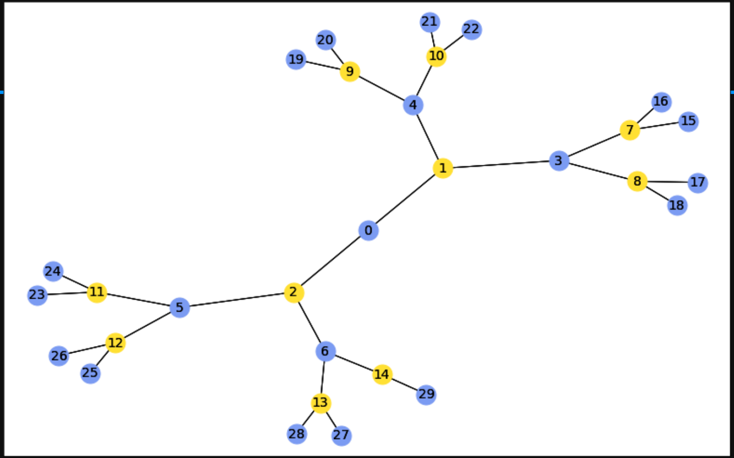
q[0] — RY[2.50] —

```
new_circuit = circuit_tetha_gamma.bind_variables({"\\theta": 0.5, "\\gamma": 0.1})
%qatdisplay new_circuit
```

q[0] – RY[1.15] –

Combinatorial optimization

Usual specification of a *combinatorial problem:*



$$C(z) = \sum \alpha_i C_i(z) \qquad \qquad argmin_z\, C(z)$$

- z is a bitstring
- C is the *target function* (or *cost function*)
- C_i are called *clauses* and are usually $\{0, 1\}$ **valued** and **local**
- α_i are called *weights*

These problems usually correspond to many yes/no decisions giving a value to the cost function we wish to minimize or maximize.

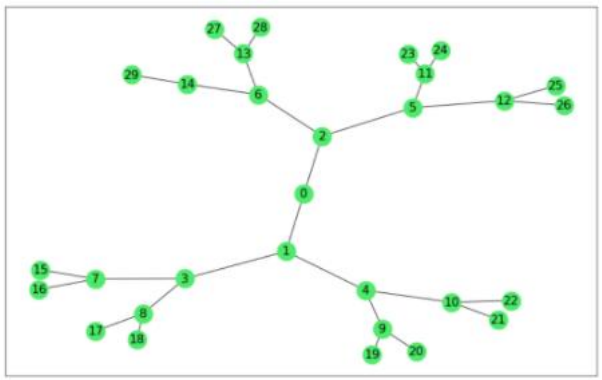
Atos

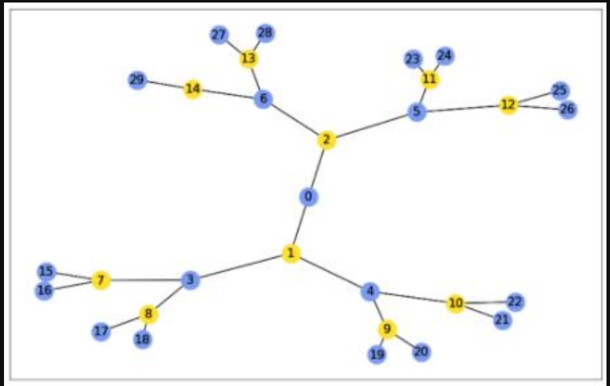
Example: Max Cut

- Given an undirected **graph G(V, E)**
  - V is a set of **vertices**
  - E is a set of **edges**

- The objective of the **Max-cut problem** is to find two sets from V that **maximize** the number of edges between these two sets.

$$H(s_1, \ldots, s_N) = \sum_{(u,v)\in E} s_i s_j$$





pyAQASM
Already implemented combinatorial problems

- Unconstrained Graph Problems:
  - Max cut
  - Graph Partitioning
- Constrained Graph Problems:
  - Graph Colouring
  - K-Clique
  - Vertex Cover
- Other problems:
  - Number Partitioning
  - Binary Integer Linear Programming

pyAQASM – CombinatorialProblem

```
from qat.opt import CombinatorialProblem

problem = CombinatorialProblem("MyProblem")
# Declare two fresh variables
var1, var2 = problem.new_vars(2)
# Add a new clause : logical AND of the two variables
problem.add_clause(var1 & var2)
# Add a new clause : XOR of the two variables
problem.add_clause(var1 ^ var2)

print(problem)
```

```
MyProblem:
 2 variables, 2 clauses
```

pyAQASM – CombinatorialProblem

```python
from qat.opt import CombinatorialProblem

problem = CombinatorialProblem("MyProblem")
# Declare two fresh variables
var1, var2 = problem.new_vars(2)
# Add a new clause : logical AND of the two variables
problem.add_clause(var1 & var2, weight=0.5)

print(problem)
```

```
MyProblem:
 2 variables, 1 clauses
```

pyAQASM – CombinatorialProblem

```
from qat.opt import CombinatorialProblem

problem = CombinatorialProblem("MyProblem")
# Declare two fresh variables
var1, var2 = problem.new_vars(2)
# Add a new clause : logical AND of the two variables
problem.add_clause(var1 & var2, weight=0.5)

obs = problem.get_observable()

print(obs)
```

- A diagonal Hamiltonian encoding the cost function of the problem can be extracted

```
0.125 * I^2 +
-0.125 * (Z|[0]) +
0.125 * (ZZ|[0, 1]) +
-0.125 * (Z|[1])
```

Atos

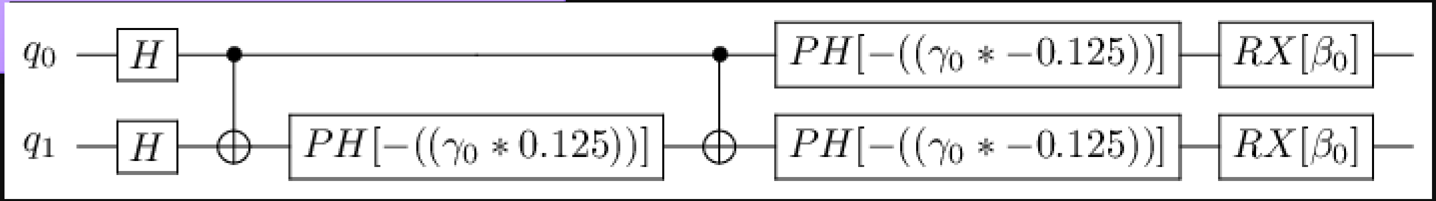
pyAQASM – CombinatorialProblem

```python
from qat.opt import CombinatorialProblem

problem = CombinatorialProblem("MyProblem")
# Declare two fresh variables
var1, var2 = problem.new_vars(2)
# Add a new clause : logical AND of the two variables
problem.add_clause(var1 & var2, weight=0.5)

ansatz = problem.qaoa_ansatz(1)
circuit = ansatz.circuit
%qatdisplay circuit
```

- It is possible to directly generate a QAOA ansatz from a CombinatorialProblem
- The circuit is accessible from the ansatz



pyAQASM – ScipyMinimizePlugin

```
from qat.qpus import get_default_qpu
from qat.plugins import ScipyMinimizePlugin
qpu = get_default_qpu()
stack = ScipyMinimizePlugin(method="COBYLA",
                    tol=1e-2,
                    options={"maxiter":150}) | qpu

result = stack.submit(ansatz)
print("Final energy:", result.value)
```

- Using ScipyMinimizePlugin we can directly minimize our ansatz

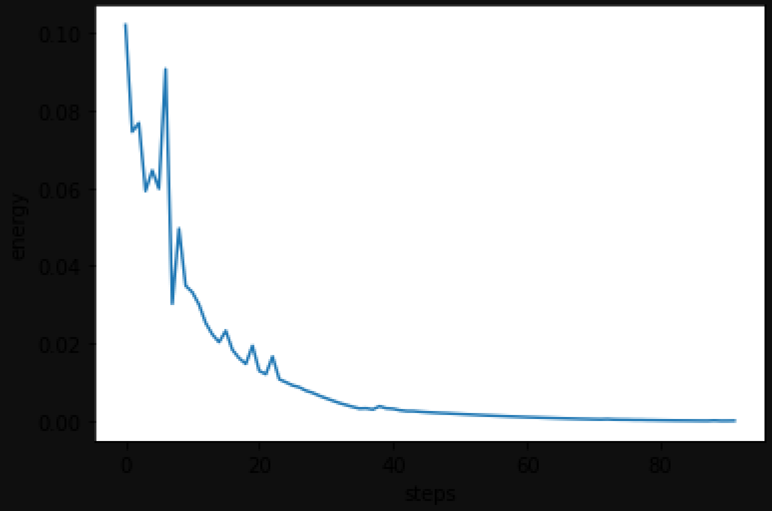
```
Final energy: 7.204089760957932e-05
```

pyAQASM – ScipyMinimizePlugin

```
import matplotlib.pyplot as plt

plt.plot(eval(result.meta_data["optimization_trace"]))
plt.xlabel("steps")
plt.ylabel("energy")
plt.show()
```

- You can print the trace of the optimization



pyAQASM – ScipyMinimizePlugin

```python
from qat.qpus import get_default_qpu
from qat.plugins import ScipyMinimizePlugin
qpu = get_default_qpu()
stack = ScipyMinimizePlugin(method="COBYLA",
                            tol=1e-2,
                            options={"maxiter":150}) | qpu

result = stack.submit(ansatz)
print("Final energy:", result.value)
```

- Multiple methods are already available:

Nelder-Mead	SLSQP
Powell	trust-constr
CG	dogleg
BFGS	trust-ncg
Newton-CG	trust-exact
L-BFGS-B	trust-krylov
TNC	
COBYLA	

Plugins

Goal: Simplify the design of applications.

Plugins can:

- process circuits (or jobs) on their way to a QPU.
- process samples (or values) on their way back.

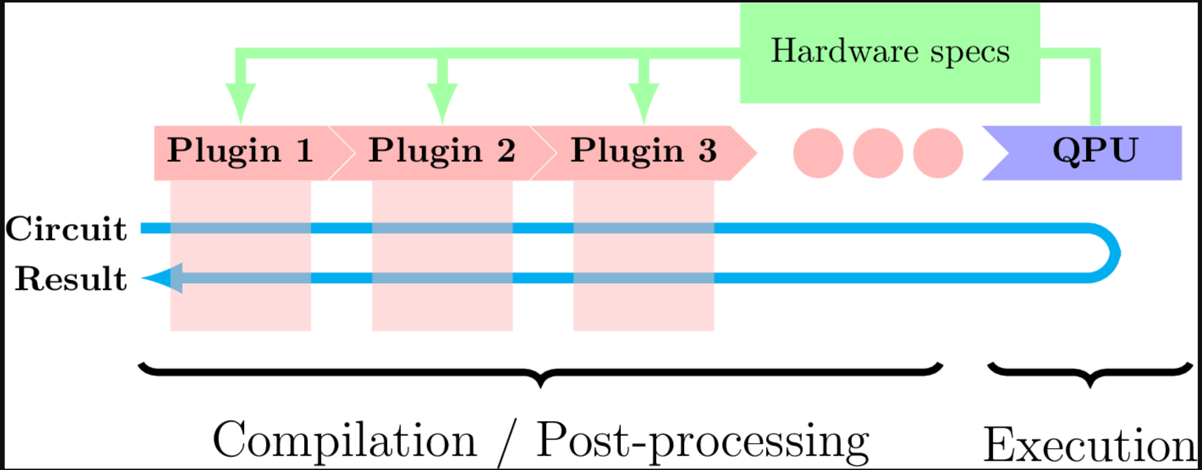
Plugins API is composed of:

- **compile** for the way in. Take a *Batch* with *HardwareSpecs* and return a new *Batch.*
- **post_process** for the way out. Process *BatchResult* and return either a *BatchResult* or a new *Batch* that should go back to the QPU.

Plugins

Creating a stack using plugins:

my_stack = plugin1 | plugin2 | … | my_qpu



Atos

Writing Plugins

```
from qat.core.plugins import AbstractPlugin

class MyPlugin(AbstractPlugin):
    def compile(self, batch, hardware_specs):
    #do something with the batch…
        return batch
    def post_process(self, batch_result):
    #do something with the results…
        return batch_result
    def do_post_process(self):
        return True


MyPlugin()
```

Atos

Using Plugins

```
from qat.qpus import LinAlg

my_stack = MyPlugin() | LinAlg()

from qat.lang.AQASM import Program, H

prog = Program()
for qb in prog.qalloc(3):
    prof.apply(H, qb)

for sample in
my_stack.submit(prog.to_circ().to_job()):
    print(sample)
```

Using Plugins

```
from qat.qpus import LinAlg

my_stack = MyPlugin() | LinAlg()

from qat.lang.AQASM import Program, H

prog = Program()
```

```
Sample(state=|000>, probability=0.1249999, amplitude=(0.35355339+0j), intermediate_measurements=None, err=None)
Sample(state=|100>, probability=0.1249999, amplitude=(0.35355339+0j), intermediate_measurements=None, err=None)
Sample(state=|010>, probability=0.1249999, amplitude=(0.35355339+0j), intermediate_measurements=None, err=None)
Sample(state=|110>, probability=0.1249999, amplitude=(0.35355339+0j), intermediate_measurements=None, err=None)
Sample(state=|001>, probability=0.1249999, amplitude=(0.35355339+0j), intermediate_measurements=None, err=None)
Sample(state=|101>, probability=0.1249999, amplitude=(0.35355339+0j), intermediate_measurements=None, err=None)
Sample(state=|111>, probability=0.1249999, amplitude=(0.35355339+0j), intermediate_measurements=None, err=None)
Sample(state=|011>, probability=0.1249999, amplitude=(0.35355339+0j), intermediate_measurements=None, err=None)
```

Example of Plugin

```
class MyPlugin(AbstractPlugin):
    def compile(self, batch, hardware_specs):
        for i, job in enumerate(batch.jobs):
            print(">> Job #{}".format(i)):
            for op in job.circuit.iterate_simple():
                print(op)
        return batch

    def post_process(self, batch_result):
        for result in batch_result.results:
            print('Result of size', len(result.raw_data))
        return batch_result
```

```
job = prog.to_circ().to_job()
# Let's submit 3 times our job in a
# single go
for sample in my_stack.submit([job]*3):
    print(sample)
```

Atos

Example of Plugin

```
>> Job #0
('H', [], [0])
('H', [], [1])
('H', [], [2])
>> Job #1
('H', [], [0])
('H', [], [1])
('H', [], [2])
>>Job #2
('H', [], [0])
('H', [], [1])
('H', [], [2])
Result of size 8
Result of size 8
Result of size 8
Result(raw_data=[Sample(state=|000>, probability=0.1249999999, amplitude=(0.35355339+0j), intermediate_measure…
Result(raw_data=[Sample(state=|100>, probability=0.1249999999, amplitude=(0.35355339+0j), intermediate_measure…
Result(raw_data=[Sample(state=|000>, probability=0.1249999999, amplitude=(0.35355339+0j), intermediate_measure…
```

List of implemented Plugins

Nnizer: swap insertion plugin
PatternManager: a pattern-based quantum circuit rewriter
Graphopt: pattern & phase polynomial-based circuit optimizer
VariationalOptimizer: a plugin for variational algorithms
ObservableSplitter: turning observable sampling into qubit sampling
CircuitInliner: inlining circuit inside a stack
Display: a console displayer plugin
QuameleonPlugin: emulating hardware constraints via a plugin

Thank you!

For more information please contact:
gaetan.rubez@atos.net