

Advance tools

2021/12/03

Gaëtan Rubez

Quantum Computing Expert for the CEPP

gaetan.rubez@atos.net



Lecture: Optimizers



Some context

- ▶ In order to increase the **fidelity** of outputs of a quantum circuit, some considerations need to be taken into account:
 - Quantum circuits are implemented by real quantum processors with **physical limitations**. These limitations may include a specific, fixed chip topology, or the existence of a preferred set of quantum gates
 - Given a quantum circuit, in general it **is NP-hard to find the shortest** equivalent circuit. Circuit length is specially important for NISQ processors. This problem can approximately solved by using heuristic optimisers

Optimizers

- ▶ We will discuss 3 kinds of circuit optimizer:
 - **NNizer**: Taking as input a quantum circuit and a restricted connectivity graph, it recasts the circuit taking into account the processor topology
 - **PBO**: It allows to find specific patterns in a circuit and replace them by an equivalent user-defined gate. This can be useful when some gates can be implemented natively faster (or at higher fidelities) than others in a given architecture
 - **GraphOpt**: It takes a circuit as input and tries to find a shorter equivalent circuit by performing a local search in the space of unitaries

Optimizers

- ▶ We will discuss 3 kinds of circuit optimizer:
 - **NNizer**: Taking as input a quantum circuit and a restricted connectivity graph, it recasts the circuit taking into account the processor topology
 - **PBO**: It allows to find specific patterns in a circuit and replace them by an equivalent user-defined gate. This can be useful when some gates can be implemented natively faster (or at higher fidelities) than others in a given architecture
 - **GraphOpt**: It takes a circuit as input and tries to find a shorter equivalent circuit by performing a local search in the space of unitaries

Quantum Circuits Topology Optimizer

General description

Optimization

NNizer

- ▶ Depending on the hardware implementation of the qubits, topology constraints for **N**earest **N**eighbors may apply
- ▶ The NNIZER can be provided with any constraint described by the user. It can then observe the input circuit and perform the necessary changes to allow the run using the given constraints

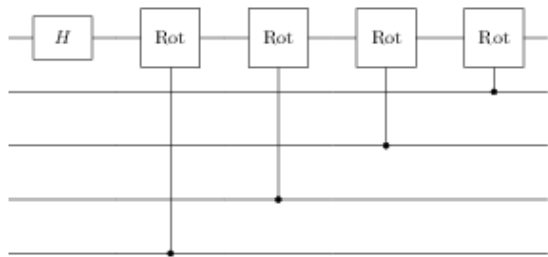
Quantum Circuits Topology Optimizer

NNizer workflow

Optimization

NNizer

User input1: Quantum circuit

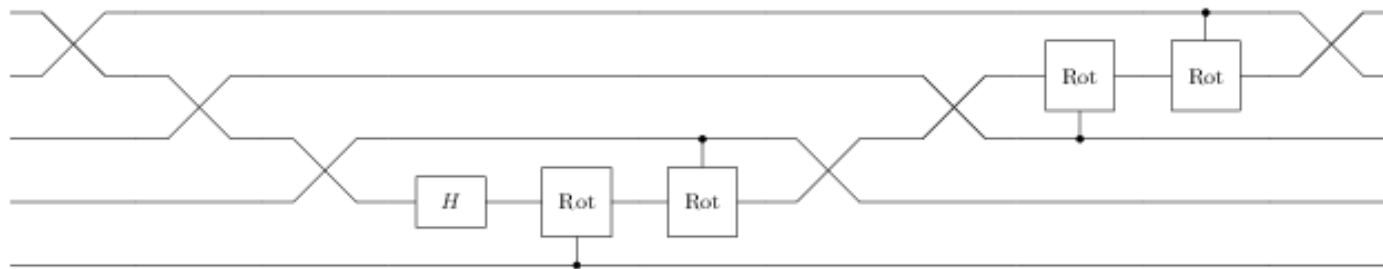
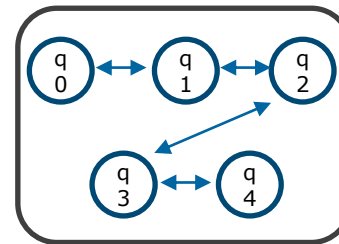


OPTIMIZATION

NNIZER

Topology constraint solver

User input2: Constraint



NNIZER optimizer

Topology & HardwareSpecs

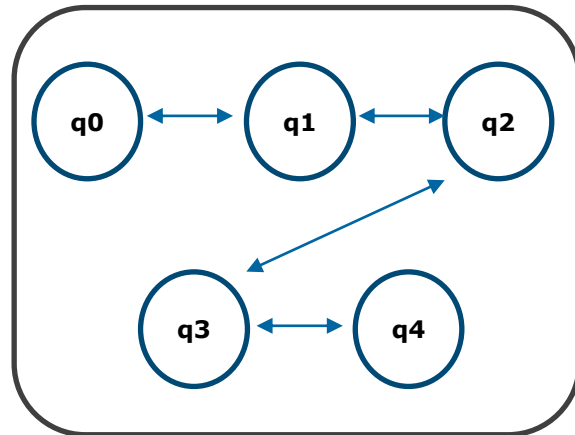
```
from qat.core import Topology
```

```
My_topology=Topology()
```

```
for i, j in [(0, 1), (1, 2), (2, 3), (3, 4)]:  
    my_topology.add_edge(i, j)
```

```
from qat.core import HardwareSpecs
```

```
My_hardware = HardwareSpecs(nbqubits=5,  
                             topology=my_topology)
```



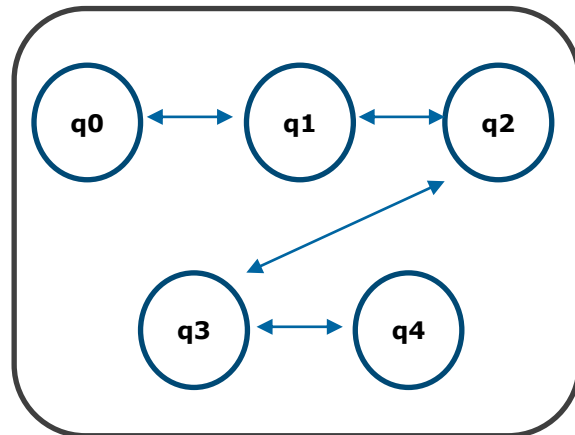
NNIZER optimizer

Topology & HardwareSpecs

- ▶ Another way:

```
from qat.core import Topology, TopologyType, HardwareSpecs
```

```
My_hardware = HardwareSpecs(nbqubits=5,  
                             topology=Topology(type=TopologyType.LNN))
```



NNIZER optimizer

QuameleonPlugin

```
from qat.core import QuameleonPlugin
from qat.qpus import LinAlg

qpu = QuameleonPlugin(specs=My_ hardware) | LinAlg()
```

- ▶ QuameleonPlugin adds constraints coming from the hardware specification created previously.
- ▶ Only compliant circuit will be executed with this stack.

NNIZER optimizer

in the stack

```
from qat.core import QuameleonPlugin
from qat.qpus import LinAlg

qpu = QuameleonPlugin(specs=My_ hardware) | LinAlg()

from qat.plugins import Nnizer

final_stack = Nnizer() | qpu

result = final_stack.submit(job)
```

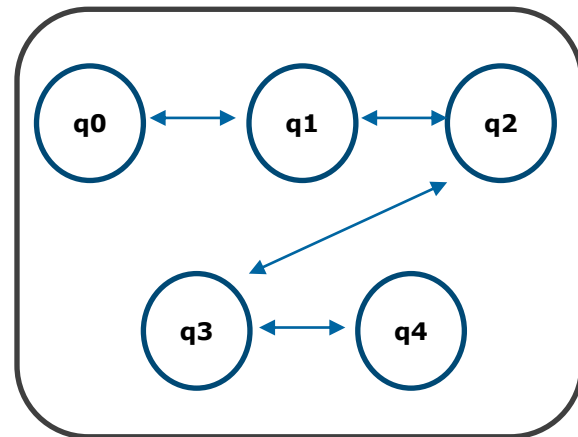
- ▶ Adding the Nnizer plugin to the stack will solve the issue by modifying the circuit accordingly to the topology.

NNIZER optimizer

topology options

- ▶ Example of json file to be passed as argument

```
{ "edges": { "0": [1],  
            "1": [0, 2],  
            "2": [1, 3],  
            "3": [2, 4],  
            "4": [3]} }
```



- ▶ Topology can be directional (i.e for CNOT)

NNIZER optimizer

topology options

- ▶ Few methods are implemented in **Nnize** to solve the swap insertion problem:
 - **atos**: based on a strict generalization of the algorithm described in [An Efficient Method to Convert Arbitrary Quantum Circuits to Ones on a Linear Nearest Neighbor Architecture](#) by *Hirata and al.*
 - **sabre**: implementation of [Tackling the Qubit Mapping Problem for NISQ-Era Quantum Devices](#) by *Gushu Li, Yufei Ding and Yuan Xie*
 - **bka**: implementation of [Efficient mapping of quantum circuits to the IBM QX architectures](#) by *Alwin Zulehner, Alexandru Paler and Robert Wille*
 - **pbn**: based on a strict generalization of the algorithm described in [Synthesis of quantum circuits for linear nearest neighbor architectures](#) by *Mehdi Saeedi, Robert Wille and Rolf Drechsler*

NNIZER optimizer

compile method

```
from qat.plugins import Nnizer
from qat.core import Batch

nnizer = Nnizer(method="atos")

nnized_batch = nnizer.compile(Batch(jobs=[ansatz]),
                              my_hardware)

nnized_ansatz_circuit = nnized_batch.jobs[0].circuit

#Number of gates in the circuit
len(nnized_ansatz_circuit.ops)

#List of the qubits
nnized_batch.jobs[0].qubits
```

Optimizers

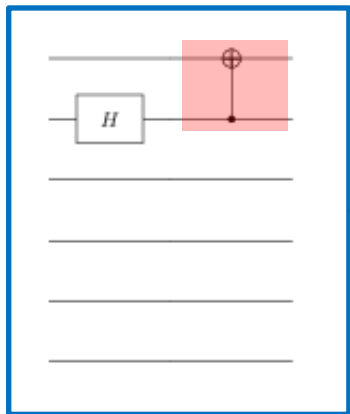
- ▶ We will discuss 3 kinds of circuit optimizer:
 - **NNizer**: Taking as input a quantum circuit and a restricted connectivity graph, it recasts the circuit taking into account the processor topology
 - **PBO**: It allows to find specific patterns in a circuit and replace them by an equivalent user-defined pattern. This can be useful when some gates can be implemented natively faster (or at higher fidelities) than others in a given architecture
 - **GraphOpt**: It takes a circuit as input and tries to find a shorter equivalent circuit by performing a local search in the space of unitaries

Pattern-based optimizer

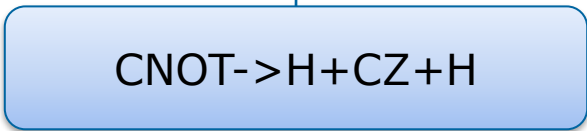
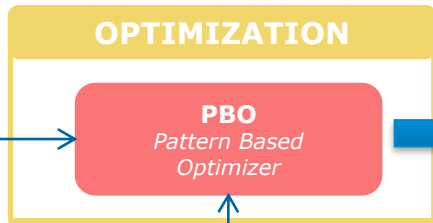
To re-write gates

Optimization

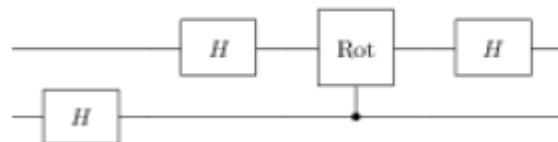
PBO



input circuit



User-defined rules set

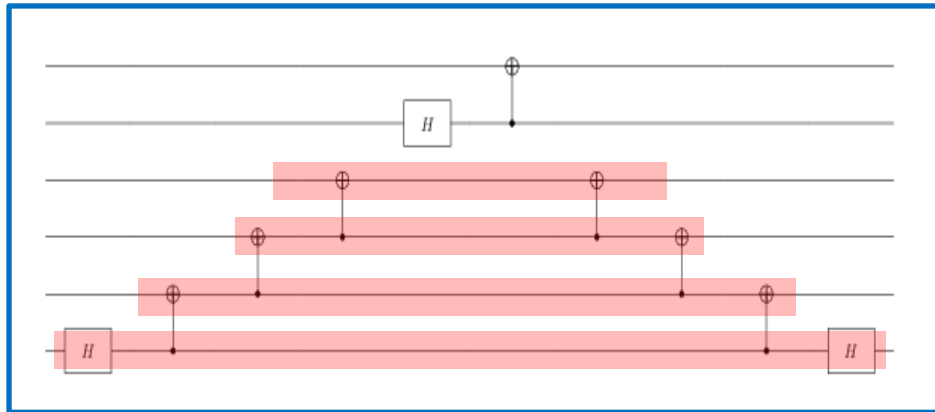


Pattern-based optimizer

To detect redundant computation

Optimization

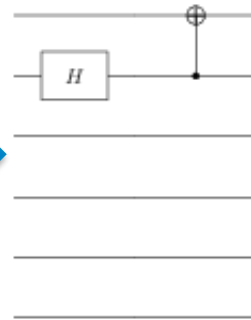
PBO



input circuit

OPTIMIZATION

PBO
Pattern Based
Optimizer



$\text{CNOT} + \text{CNOT} \rightarrow \text{Id}$
 $\text{H} + \text{H} \rightarrow \text{Id}$

User-defined rules set

Pattern based optimizer

Example

```
from qat.lang.AQASM import Program, H, X
```

```
# Define initial circuit (X - H - H circuit)
```

```
prog = Program()
```

```
qubit = prog.qalloc(1)
```

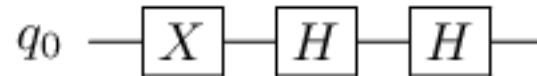
```
prog.apply(X, qubit)
```

```
prog.apply(H, qubit)
```

```
prog.apply(H, qubit)
```

```
circ = prog.to_circ()
```

```
%qatdisplay circ
```



Pattern based optimizer

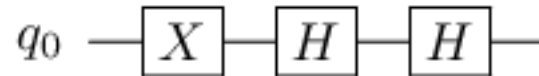
Example

```
from qat.pbo import GraphCircuit
```

```
# Create a graph object and load circuit
```

```
graph = GraphCircuit()
```

```
graph.load_circuit(circ)
```



Pattern based optimizer

Example

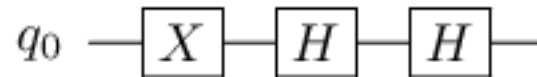
```
from qat.pbo import GraphCircuit
```

```
# Create a graph object and load circuit
```

```
graph = GraphCircuit()  
graph.load_circuit(circ)
```

```
# Define two patterns
```

```
left_pattern = [("H", [0]), ("H", [0])]  
right_pattern = []
```



Pattern based optimizer

Example

```
from qat.pbo import GraphCircuit
```

```
# Create a graph object and load circuit
```

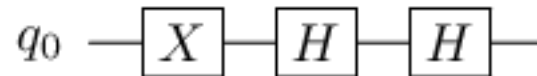
```
graph = GraphCircuit()  
graph.load_circuit(circ)
```

```
# Define two patterns
```

```
left_pattern = [("H", [0]), ("H", [0])]  
right_pattern = []
```

```
# Replace left_pattern by right_pattern
```

```
graph.replace_pattern(left_pattern, right_pattern)
```



Pattern based optimizer

Example

```
from qat.pbo import GraphCircuit
```

```
# Create a graph object and load circuit
```

```
graph = GraphCircuit()  
graph.load_circuit(circ)
```

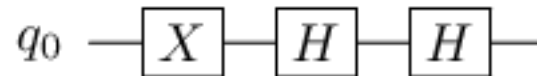
```
# Define two patterns
```

```
left_pattern = [("H", [0]), ("H", [0])]  
right_pattern = []
```

```
# Replace left_pattern by right_pattern
```

```
graph.replace_pattern(left_pattern, right_pattern)
```

```
while graph.replace_pattern(left_pattern, right_pattern):  
    continue
```



Pattern based optimizer

Example

```
from qat.pbo import GraphCircuit
```

```
# Create a graph object and load circuit
```

```
graph = GraphCircuit()  
graph.load_circuit(circ)
```

```
# Define two patterns
```

```
left_pattern = [("H", [0]), ("H", [0])]  
right_pattern = []
```

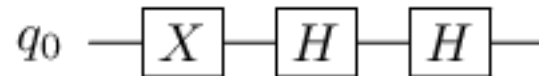
```
# Replace left_pattern by right_pattern
```

```
graph.replace_pattern(left_pattern, right_pattern)
```

```
while graph.replace_pattern(left_pattern, right_pattern):  
    continue
```

```
# Get the optimized circuit
```

```
optimized_circ = graph.to_circ()
```



Pattern based optimizer

Abstract gates

```
# graph.to_circ() won't work since "HZ" gate is not known
```

```
from qat.lang.AQASM import AbstractGate  
graph.add_abstract_gate(AbstractGate("HZ", [], 1))
```

```
# graph.to_circ() is now working  
optimized_circ = graph.to_circ()
```


Optimizers

- ▶ We will discuss 3 kinds of circuit optimizer:
 - **NNizer**: Taking as input a quantum circuit and a restricted connectivity graph, it recasts the circuit taking into account the processor topology
 - **PBO**: It allows to find specific patterns in a circuit and replace them by an equivalent user-defined gate. This can be useful when some gates can be implemented natively faster (or at higher fidelities) than others in a given architecture
 - **GraphOpt**: It takes a circuit as input and tries to find a shorter equivalent circuit by performing a local search in the space of unitaries

GraphOpt

Description

- ▶ The main goal of GraphOpt reduce the total number of gates among
 - H, CNOT, X
 - Rz/Ph

These gates represent a set typically available in real quantum hardware implementations.

- ▶ A preprocessing step has been included in our implementation that splits CPh, CRz and Toffoli gates into subcircuit in the adequate gate set.
- ▶ It is inspired by the result:

Quantum Physics

Automated optimization of large quantum circuits with continuous parameters

[Yunseong Nam, Neil J. Ross, Yuan Su, Andrew M. Childs, Dmitri Maslov](#)

(Submitted on 19 Oct 2017 (v1), last revised 1 Jun 2018 (this version, v2))

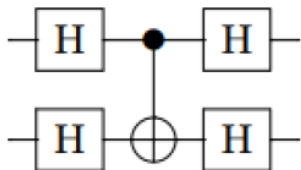
GraphOpt

To optimize circuits

Optimization

GraphOpt

input circuit



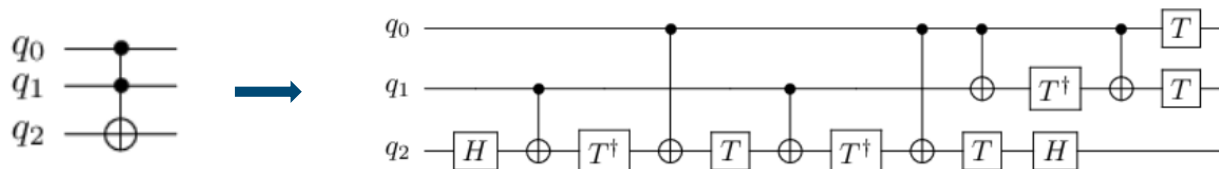
OPTIMIZATION

GRAPHOPT



GraphOpt

- ▶ **First step:** expand the circuit using only the set of gates. This leads to an increase of the number of gates of gates

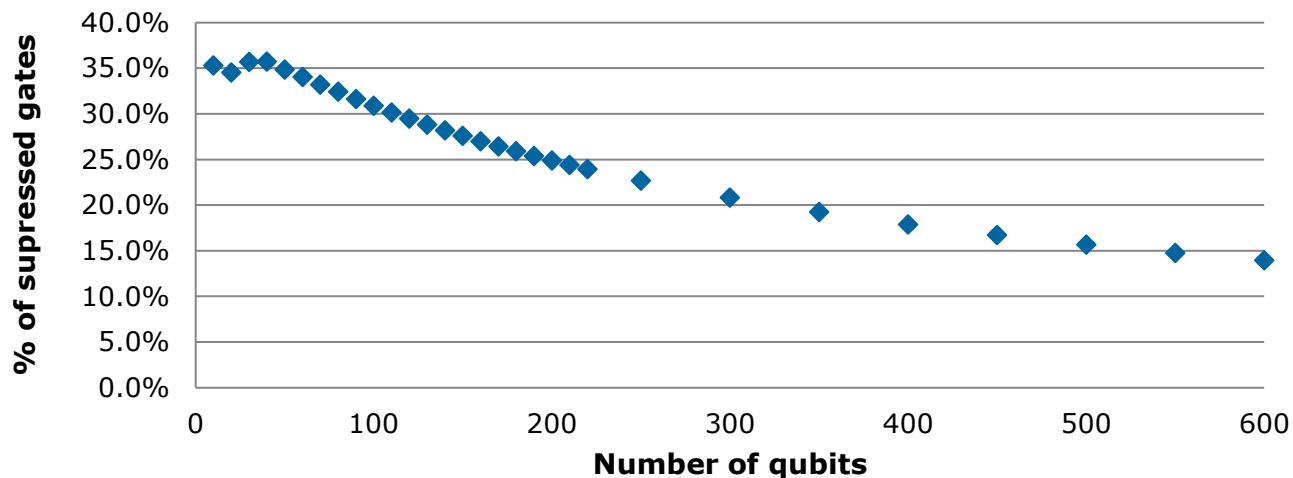


- ▶ **Second step:** optimize the circuit to reduce the number of gate after expansion



GraphOpt

**Fraction of suppressed gates while optimizing
Adder (sum of 2 qubits register)**



- ▶ More than 30% of suppressed gates for # qubits < 100 ! (after expansion)

GraphOpt

```
from qat.plugins import Graphopt
# from qat.graphopt import Graphopt # Equivalently

my_stack = Graphopt() | LinAlg()

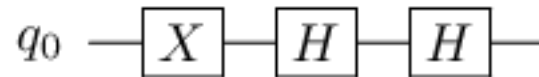
my_stack.submit(job)
```

GraphOpt

```
from qat.plugins import Graphopt  
# from qat.graphopt import Graphopt # Equivalently
```

```
my_stack = Graphopt() | LinAlg()
```

```
my_stack.submit(job)
```



Hands-on 8: Optimizers

Hands-on 8: Optimizers

▶ Log on the QLM

▶ Go to Hands-on 8 directory :

<http://127.0.0.1:8888/tree/notebooks/Hands-on8>

▶ Open and complete the notebook *Optimising-QAOA.ipynb*

Lecture: Noisy Simulation

The Problem of Noise

How Quantum and Classical Fluctuations affect QC

- ▶ Most quantum algorithms can be implemented by circuits consisting on a fixed sequence of gates.
- ▶ Noise can be thought of as introducing **random gates** in a particular circuit. The nature of these gates depends on the source of noise.
 - The quantum advantage gets slightly modified if noise is very small, but can be destroyed if noise is too large or if circuits are too long. Information about this will be contained in the **hardware specifications**.
- ▶ The way to fight noise is to implement **quantum error correction**. This is out of reach for NISQ processors. In this case, postprocessing techniques may be used to undo the effects of noise for a particular task.

The Problem of Noise

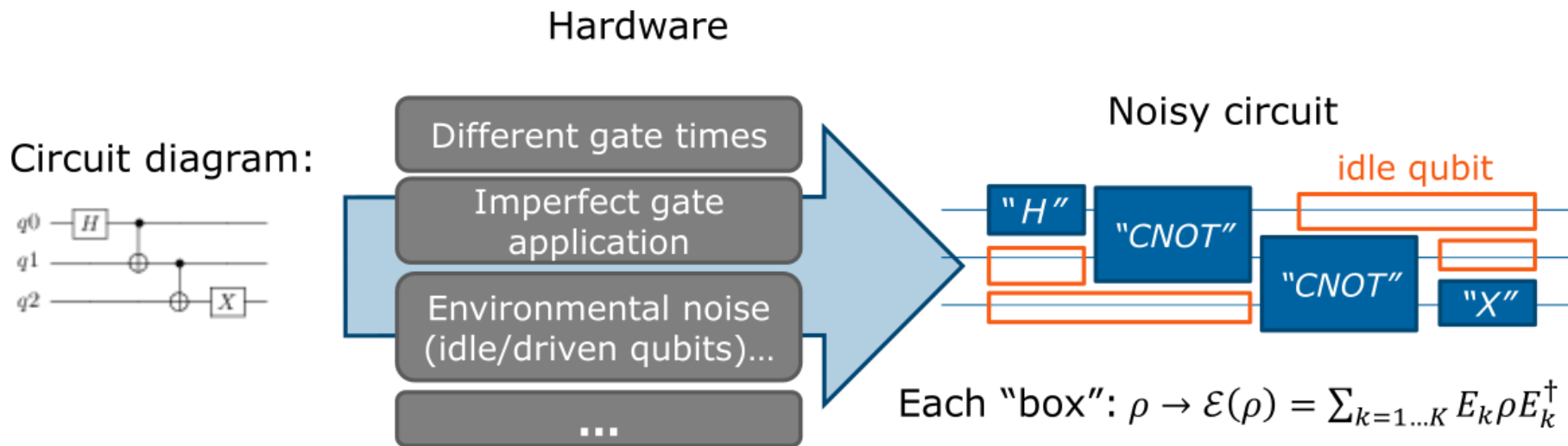
On the QLM

- ▶ A discrete description of the temporal evolution of **a noisy circuit**.
- ▶ Starting from a perfect quantum circuit, we build a noisy quantum circuit:
 - By specifying the meaning of gates
 - By potentially adding new operations that describe the effect of noise (for e.g. the effect of noise during “idling” periods).
- ▶ The resulting circuit is made up of “boxes”, each of which describes an action on the density matrix of the qubits

Noisy Circuit Generation

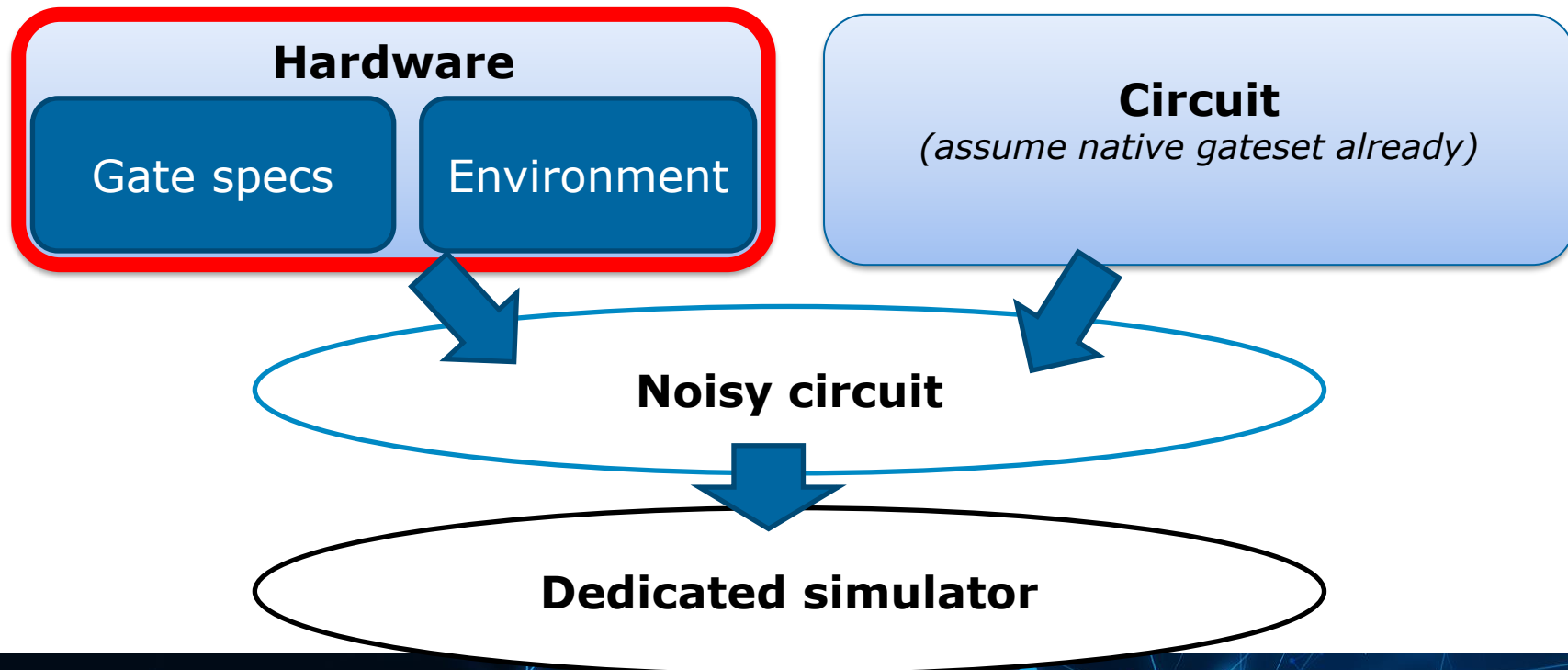
From Noiseless to Noisy Circuits

- ▶ Using a clean circuit and a hardware specification, the Atos QLM generates a noisy circuit object:



Noisy quantum simulation

Overview



Noise models

- ▶ The notion of “**noise model**” refers to the action of the gates and the environment on the qubits’ density matrix ρ .
- ▶ There are several ways to describe such **noise models**.
- ▶ The only common property is that they represent a **linear map**.
- ▶ They sometimes come with extra properties such as **trace preservation (TP) and complete positivity (CP)**.
- ▶ When these two properties are fulfilled, one can speak of **CPTP maps**, or **quantum channels**.

QuantumChannelKraus

Example

```
import numpy as np
from qat.quops import QuantumChannelKraus

H_mat = np.array([[1, 1], [1, -1]])/np.sqrt(2)
p = 0.2
noisy_H = QuantumChannelKraus(
    kraus_operators=[np.sqrt(1-p)*H_mat,
                    np.sqrt(p)*np.identity(2)],
    name="noisy identity")
```

You can define Quantum Channel with a Kraus representation.

In that example, roughly speaking:
80% of the time the Hadamard gate will be applied correctly.

20% of the time nothing will be applied.

Mathematical Description of Noisy Channels

Kraus Representation

Generic mathematical description for quantum processes
(imperfect gates, environmental noise...)

completely positive, trace-preserving maps:

$$\mathcal{E}(\rho) = \sum_{k=1\dots K} E_k \rho E_k^\dagger$$

with

ρ : density matrix

E_k : Kraus operators, with property: $\sum_k E_k^\dagger E_k = 1$

“Ideal” gate:

special case $K = 1$, $E_1 = U$ unitary matrix

Pure state:

special case $\rho = |\psi\rangle\langle\psi|$

Mathematical Description of Noisy Channels

Kraus Representation

Kraus operators can depend on some parameter t

$$E_k \rightarrow E_k(t)$$

Use case: modelling idle noise since the idle times of a qubit can vary during the circuit execution.

Some Noise Models

Noise depends on architecture

- ▶ Noise models depend on the nature of the physical architecture. Some factors to take into account are the **temperature** of operation, the **lifetime** of qubits and the **quality** of applied gates. Classical noise can be modeled as bit-flips, ie. apply the Pauli X operator with a probability p :

$$\rho = |\psi\rangle\langle\psi| \rightarrow \mathcal{E}_{classical}(\rho) = (1 - p)\rho + pX\rho X$$

```
import numpy as np
from qat.quops import QuantumChannelKraus
Pauli_X = np.array([[0.,1.],[1.,0.]])
my_channel = QuantumChannelKraus(kraus_operators=[np.sqrt(0.75)*np.identity(2),
                                                  np.sqrt(0.25)*Pauli_X],
                                name="classical channel")
```

- ▶ Typical Quantum Noise Channels include:
 - **Amplitude Damping**: This arises when qubits lose their energy to the environment
 - **Phase Damping**: This is due to quantum processes which do not imply energy transfer

Noisy Circuit Generation Structure

**H
a
r
d
w
a
r
e**

Gates
specification

gate_times

dict of QuantumChannelKraus (parametric or not)
associated to each gate
(opt.: qubit associated)

initialization

measurement

Gate
noise

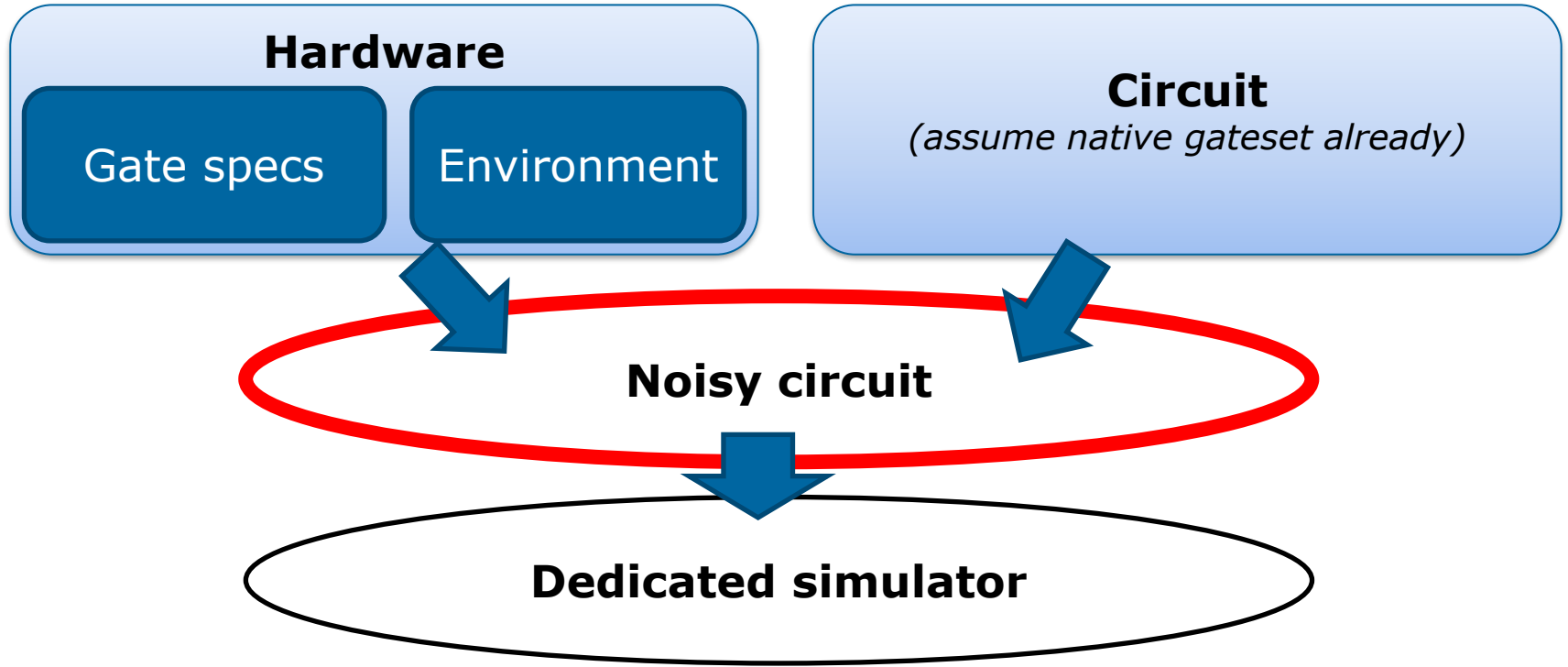
dict of Parametric Quantum Channel
associated to each gate

Idle
noise

list of Parametric Quantum Channel
(opt.: qubit associated)

Noisy quantum simulation

Overview



Noisy Circuit Generation

Example: Gates

- ▶ The definition of gates consists of the specification of (i) **duration** and (ii) **Kraus operators**:

```
gate_times = { "Foo": 10,
               "Bar": lambda theta: theta/5.+5.,
               "RX": lambda theta: theta/10.}

quantum_channels = {
    "Foo": QuantumChannelKraus([np.sqrt(0.75)*foo_mat,
                               np.sqrt(0.25)*np.array([[0,1],[1,0]],
                                                       dtype = np.complex_)]),
    "Bar": lambda theta: QuantumChannelKraus([np.array([[1, 0, 0, 0],
                                                       [0, 1, 0, 0],
                                                       [0, 0, np.exp(theta*1j), 0],
                                                       [0, 0, 0, 1.]],
                                                       dtype = np.complex_)]),
    "RX" : {
        0: lambda theta: QuantumChannelKraus([np.array([[np.cos(theta/2+.01), -1j*np.sin(theta/2+.01)],
                                                       [-1j*np.sin(theta/2+.01), 1j*np.cos(theta/2+.01)]],
                                                       dtype = np.complex_)]),
        1: lambda theta: QuantumChannelKraus([np.array([[np.cos(theta/2+.02), -1j*np.sin(theta/2+.02)],
                                                       [-1j*np.sin(theta/2+.02), 1j*np.cos(theta/2+.02)]],
                                                       dtype = np.complex_)]),
    }
}

gates_spec = GatesSpecification(gate_times, quantum_channels=quantum_channels)
```

Duration

Operator definition

Noisy Circuit Generation

Example: Environment Modeling

- ▶ An environment model consists of a user-defined **noise for gates** (in this example it is obviated) and **environment** (applied to qubits when idle)

```
# custom noise
def my_custom_noise(idling_time):
    error_prob = 1 - np.exp(-idling_time / 400.)
    return QuantumChannelKraus([np.sqrt(1-error_prob)*np.identity(2), np.sqrt(error_prob)* np.array([[0,1],[1,0]])],
                               name = "MyCustomNoise")

# we assume that each qubit experiences a different AD/PD noise, and the same custom noise
idle_noise = {
    0: [ParametricAmplitudeDamping(T_1=210), ParametricPureDephasing(T_phi=105), my_custom_noise],
    1: [ParametricAmplitudeDamping(T_1=208), ParametricPureDephasing(T_phi=95), my_custom_noise],
    2: [ParametricAmplitudeDamping(T_1=199), ParametricPureDephasing(T_phi=101), my_custom_noise],
}

# note: in the case of identical noise parameters for each qubit, we could have written:
# idle_noise = [ParametricAmplitudeDamping(T_1=200), ParametricPureDephasing(T_phi=100), my_custom_noise]

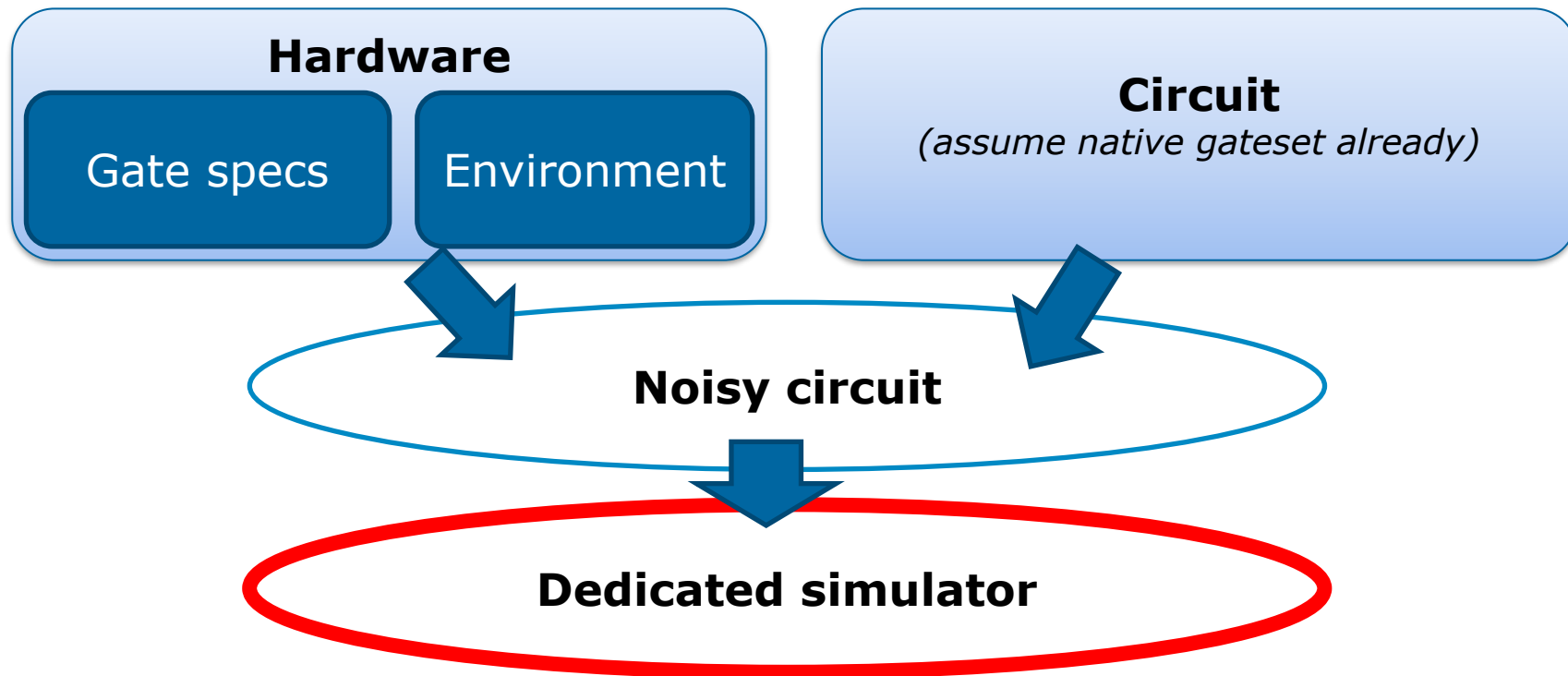
hardware_model = HardwareModel(gates_spec, None, idle_noise)
```

Effect of environment

No noise to be introduced after each gate

Noisy quantum simulation

Overview



Noisy Circuit Simulators

Two Approaches for Noisy Circuit Simulation

► Deterministic simulation of the matrix evolution

$$\mathcal{E}(\rho) = \sum_{k_M} \left(E_{k_M}^{(M)} \dots \left(\sum_{k_1} E_{k_1}^{(1)} \rho E_{k_1}^{(1)\dagger} \right) \dots E_{k_M}^{(M)\dagger} \right)$$

- **Advantages:** returns the full density matrix after evolution
- **Drawbacks:** resource-intensive. Subject to discretization error

► Stochastic sampling of trajectories

$$\mathcal{E}(\rho = |\psi_0\rangle\langle\psi_0|) = \sum_{k_1 \dots k_M} E_{k_M}^{(M)} \dots E_{k_1}^{(1)} |\psi_0\rangle\langle\psi_0| E_{k_1}^{(1)\dagger} \dots E_{k_M}^{(M)\dagger}$$

- **Advantages:** less resource consuming. Not subject to discretization error
- **Drawbacks:** provides estimates of the expectation values. Storage exponential on number of trajectories

Noisy Circuit Simulators

Example

```
import qat.noisy, qat.linalg
# We define a first noisy quantum processor with this hardware model,
# and a deterministic method to simulate the computation of the circuit
noisy_qpu_1 = NoisyQProc(hardware_model = hardware_model, sim_method = "deterministic")

# We define a second noisy quantum processor with the same hardware model,
# but this time with a stochastic method to simulate the computation of the circuit
noisy_qpu_2 = NoisyQProc(hardware_model=hardware_model,
                        sim_method="stochastic",
                        backend_simulator=qat.linalg.LinAlg(),
                        n_samples=10000)
```

a noisy qpu is a pair of
(hardware model,
simulator method)

2 types of noisy
simulators

the task can then be executed as usual

Noisy Circuit Simulators

Summing Up

- ▶ Density matrix simulation is a costly **deterministic** method. It is limited to few qubits.
- ▶ To reach large number of qubits: Monte-Carlo methods i.e **stochastic** sampling:
 - number of samples (hence accuracy) is defined by user
 - The error corresponds to the error bar on the probability, evolving like $1/\sqrt{\text{nb_samples}}$
 - You can also specify the backend simulator (linalg, MPS, Feynman)
- ▶ Note that contrary to ideal circuit simulators, we do not have access to the quantum amplitudes, but merely the probabilities.

Hands-on 9: QAOA noisy simulation

Hands-on 9: noisy simulation


▶ Log on the QLM

▶ Go to Hands-on 9 directory :

<http://127.0.0.1:8888/tree/notebooks/Hands-on9>

▶ Open and complete the notebook *Noisy-QAOA.ipynb*

Lecture: Simulators



Quantum Circuits Simulation

Linalg

Simulation

Linalg

Based on linear algebra. N qubits represented by a 2^N vector. Heavy load on memory bandwidth and memory access latency.



- ▶ general purpose simulator, any gate, any arity
- ▶ linear simulation time in function of number of gates
- ▶ Predictable run-time and memory usage
- ▶ Access to entire amplitude vector



- ▶ Memory usage: 2^N
- ▶ Execution time grows exponentially with number of qubits

Quantum Circuits Simulation

Feynman

Simulation

Feynman

Based on Feynman path integral. Computes all final reachable states and sums the contributions of each path.



- ▶ Depending on the number of “touched” states, memory usage can be very low.
- ▶ Fast for circuits with few dense gates
- ▶ Any gate up to arity 3



- ▶ Run-time exponential in the number of dense gates (like Hadamard)

Quantum Circuits Simulation

MPS

Simulation

MPS

Based on Matrix product states representation

- ▶ MPS is suited to simulate circuit with low entanglement
 - ▶ Simulation time and memory size depend on circuit entanglement.
 - ▶ For low/medium entangled circuit, low memory usage, fast simulation
 - ▶ Up to 1000 qubits could be simulated with low entanglement.
 - ▶ Cutoff threshold on Schmidt coefficients can be specified
-
- ▶ Limited to gates with at most arity 3
 - ▶ Only accepts gates acting on neighbouring qubits. Circuit need to be nnized

Quantum Circuits Simulation

Stabs

Simulation

Stabs

Based on stabilizer formalism



- ▶ Allows to simulate a very large number of qubits
- ▶ Very efficient simulation of Clifford circuits



- ▶ Restricted available gates: **CNOT,H,CZ,S,X,Y,Z,SWAP**
(gate T and Toffoli not accepted)
- ▶ Amplitudes are only determined up to a global phase

Quantum Circuits Simulation

Binary Decision Diagrams

Simulation

BDD

Based on QMDDs (Quantum Multi-valued Decision Diagrams)



- ▶ Accepts any gate, with arbitrary arity (but connectivity restrictions)
- ▶ Allow to simulate more qubits than linalg



- ▶ gates need to act on neighboring qubits, except for controls, which can be taken arbitrarily far from the gate application.

Use cases:

- ▶ Shor, especially Toffoli-based arithmetics
- ▶ circuits containing “a lot of 0,1-valued unitaries” tend to behave well on QMDDs

Hands-on 10: Simulators

Hands-on 10: Simulators

▶ Log on the QLM

▶ Go to Hands-on10 directory :

<http://127.0.0.1:8888/tree/notebooks/Hands-on10>

▶ Open and complete the notebook *Simulators.ipynb*

Thank you.

Gaëtan Rubez

Quantum Computing Expert for the CEPP

gaetan.rubez@atos.net

Atos, the Atos logo, Atos Codex, Atos Consulting, Atos Worldgrid, Bull, Canopy, equensWorldline, Unify, Worldline and Zero Email are registered trademarks of the Atos group. March 2017. © 2017 Atos. Confidential information owned by Atos, to be used by the recipient only. This document, or any part of it, may not be reproduced, copied, circulated and/or distributed nor quoted without prior written approval from Atos.

The Atos logo is displayed in the bottom right corner. It consists of the word "Atos" in a bold, white, sans-serif font. The letter 'o' is stylized with a white circle inside it. The background of the slide features a complex network of glowing blue lines and nodes, resembling a data network or quantum computing structure, which is more prominent on the right side.