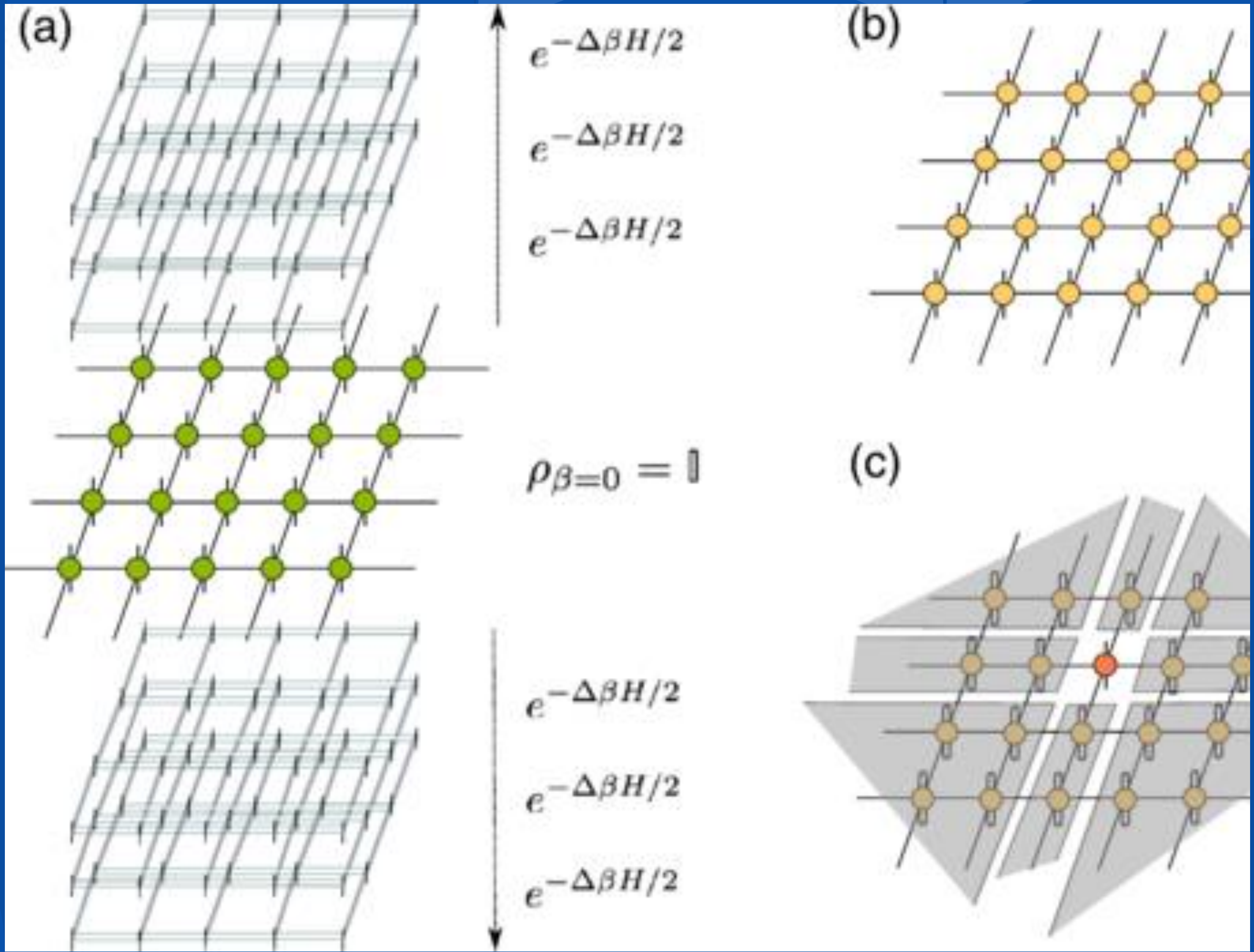
Perfect quantum simulation through tensor networks

GABRIEL ROCHETTE, ALEXANDRE GRAVEREAUX, SEIF ZAAFOURI
PROJET N°1 - FINAL PRESENTATION - 28/03/2025

PLAN

- Introduction – context and theory
- Presentation of the project
- Tensor networks in Quantum Computing
- Application to QML
- Further discussion

Context And Theory

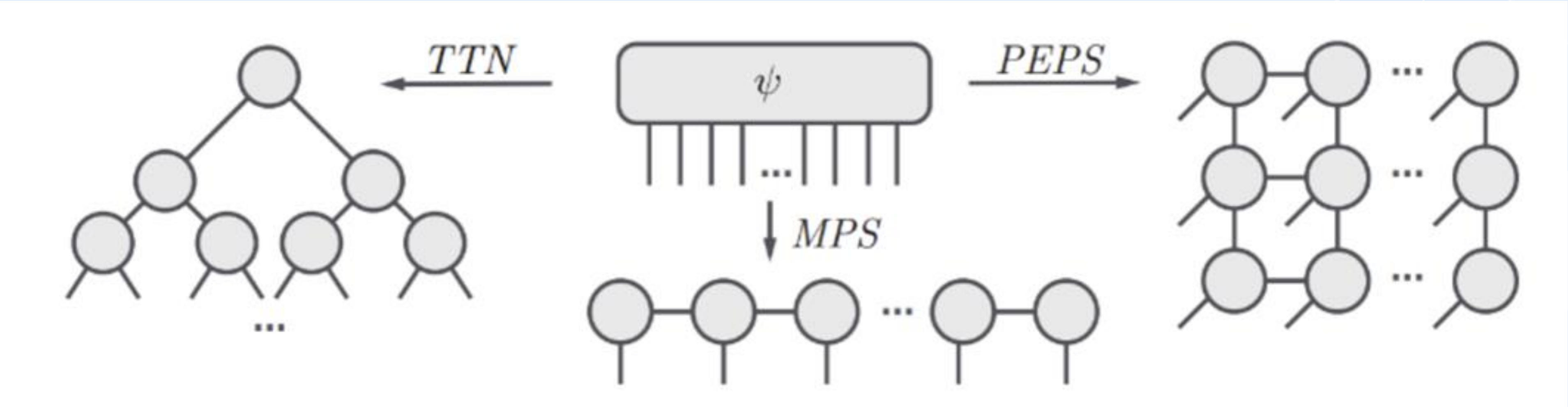


LIMITATIONS OF STATEVECTOR SIMULATION

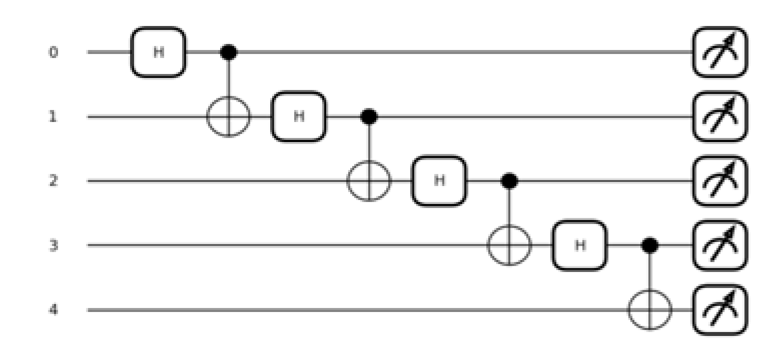
- Exponential increase in space and time complexity with number of qubits
- Turn to tensor networks instead
- Parallelize contraction to gain time
- Cheaper memory and more time efficient

MAIN IDEAS OF TENSOR GRAPHS

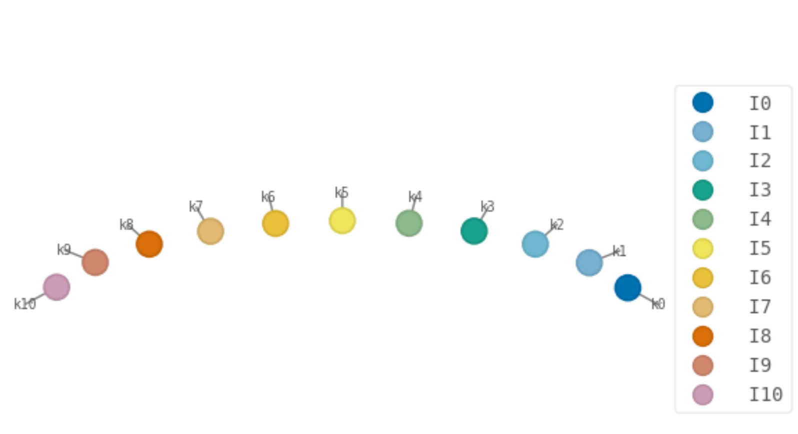
- See quantum gates as linear mappings : <u>tensors</u>
- Multiple methods to turn a quantum circuit into a tensor graph
  - Tree Tensor Network, Matrix Product State, Projected Entangled Pair states

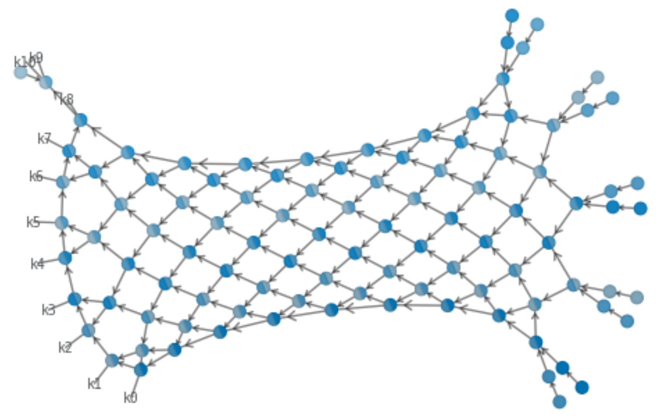


FOR THIS PROJECT: PENNYLANE + JAX




```
qml.device("default.tensor", method="mps",
**kwargs_mps)
```

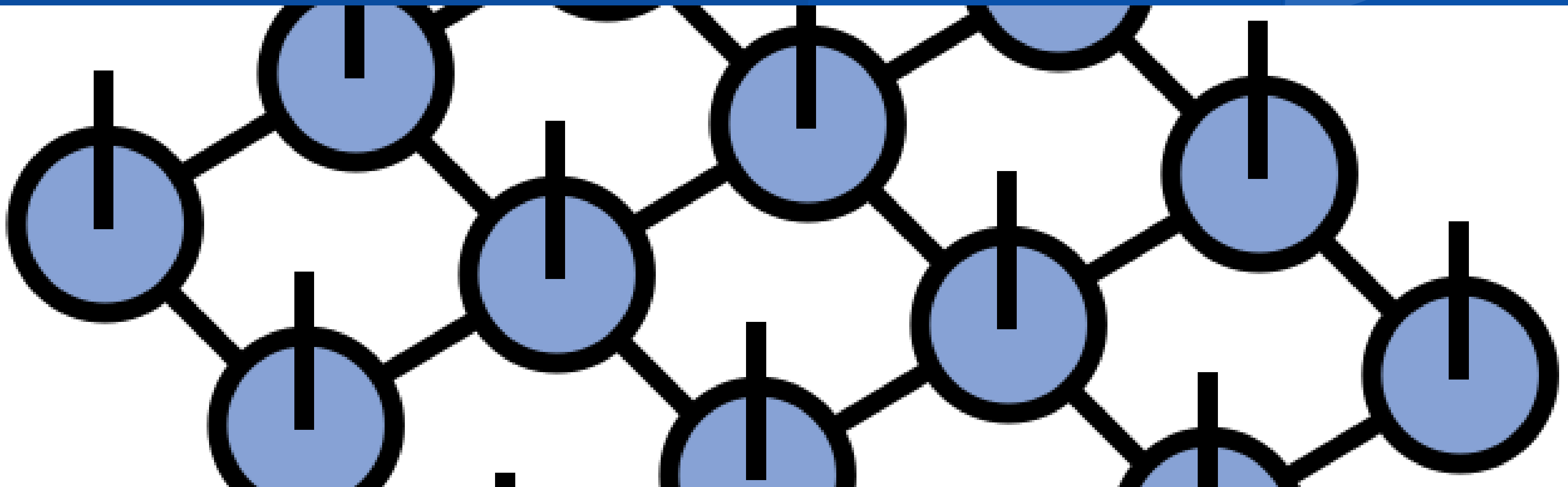




GOAL OF THE PROJECT

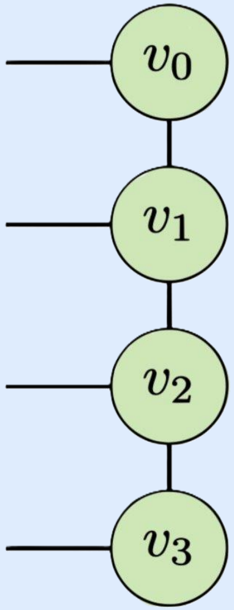
- Objective: look into perfect simulation with high number of qubits through parallel contraction of tensor networks.
- First: characterize efficiency of parallelization.
- Second: apply these methods to QML problems.
- Third: Try on real QPU (IBM)

Tensor networks in quantum computing

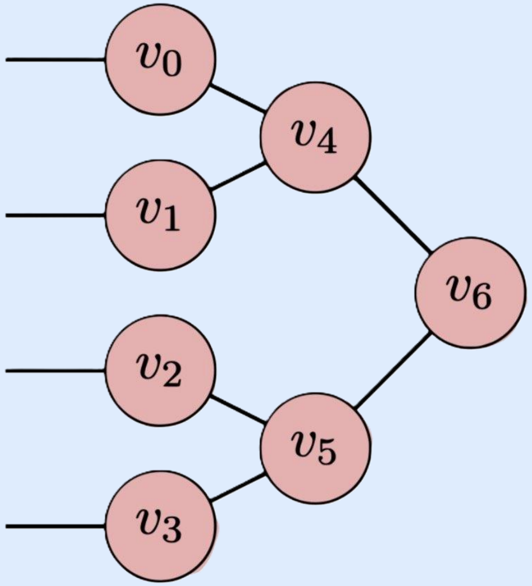


IMPLEMENTATION AND FIRST TESTS

Pennylane module: default.qubit vs default.tensor
Computationnal method: MPS and TN



MPS (approx, low entanglement)

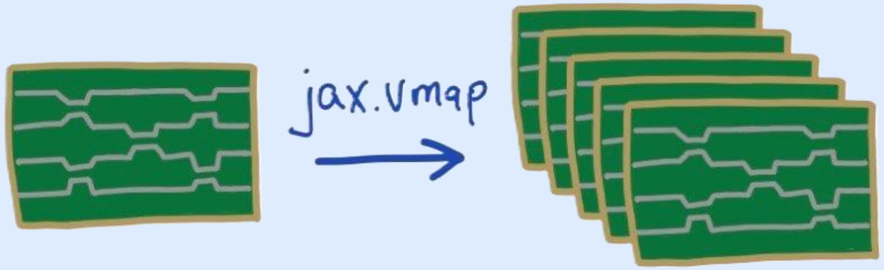


TN (exact result, higher cost)

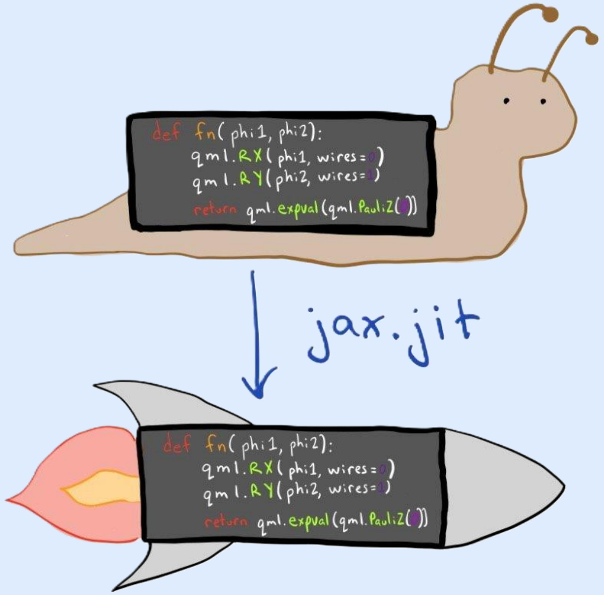
IMPLEMENTATION AND FIRST TESTS

Use of JAX: scientific computing library for parallelisation

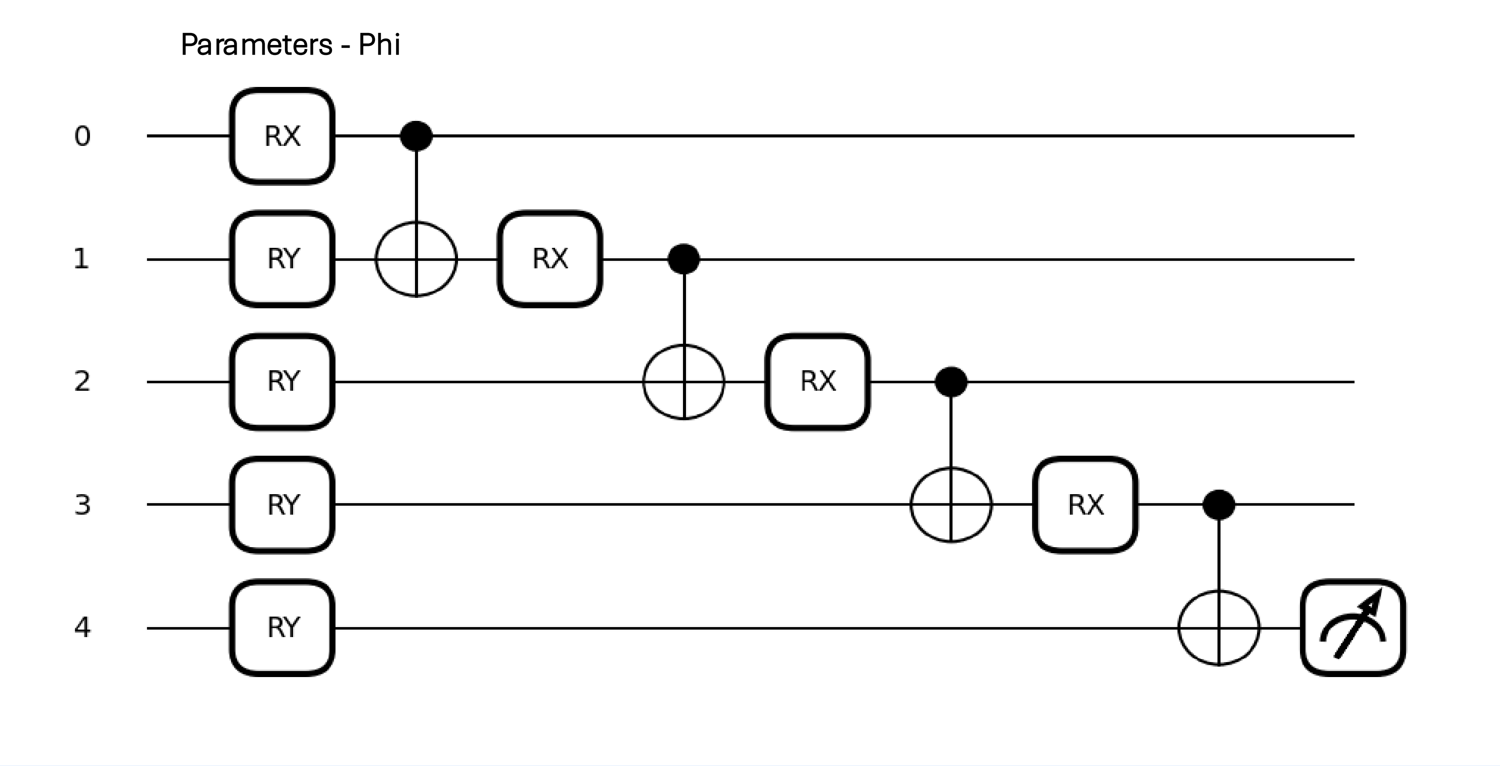
Jax.vmap: run batches of circuits in parallel



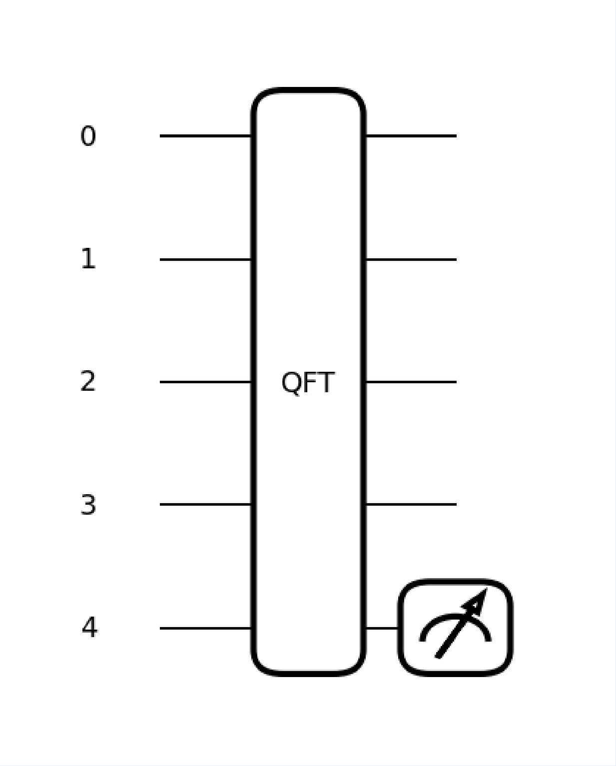
Jax.jit: compile circuit execution to XLA



RESULTS

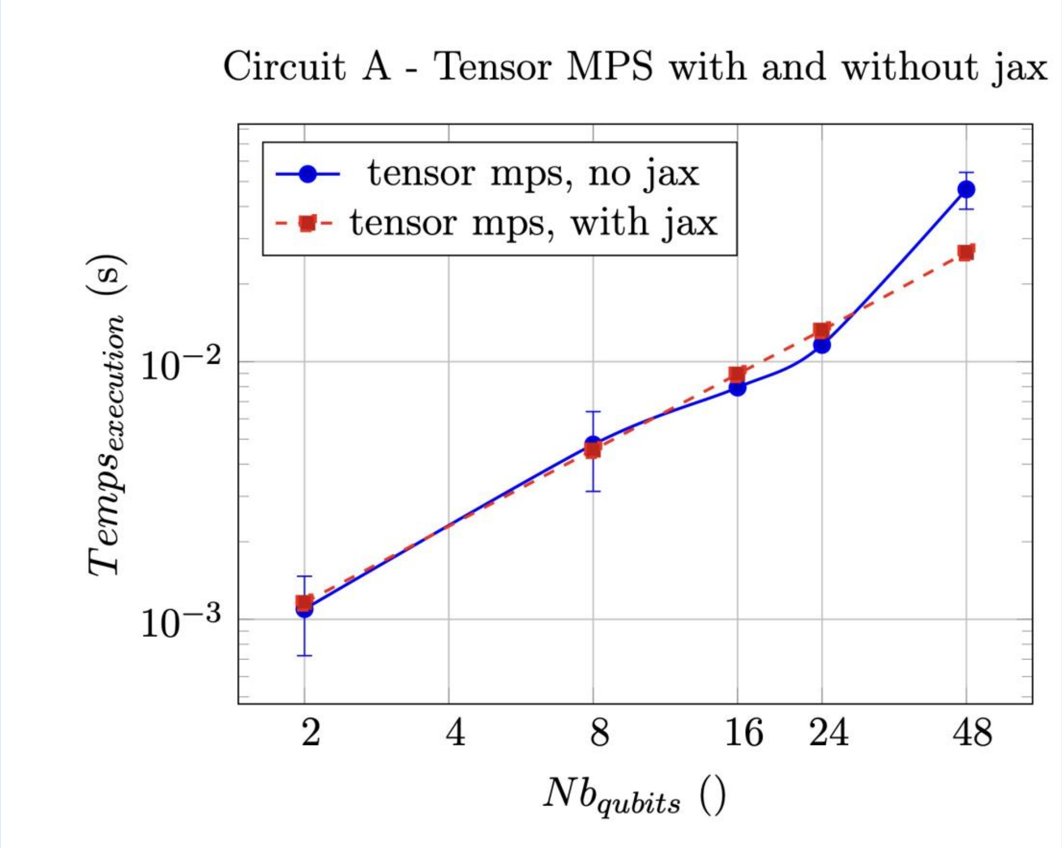


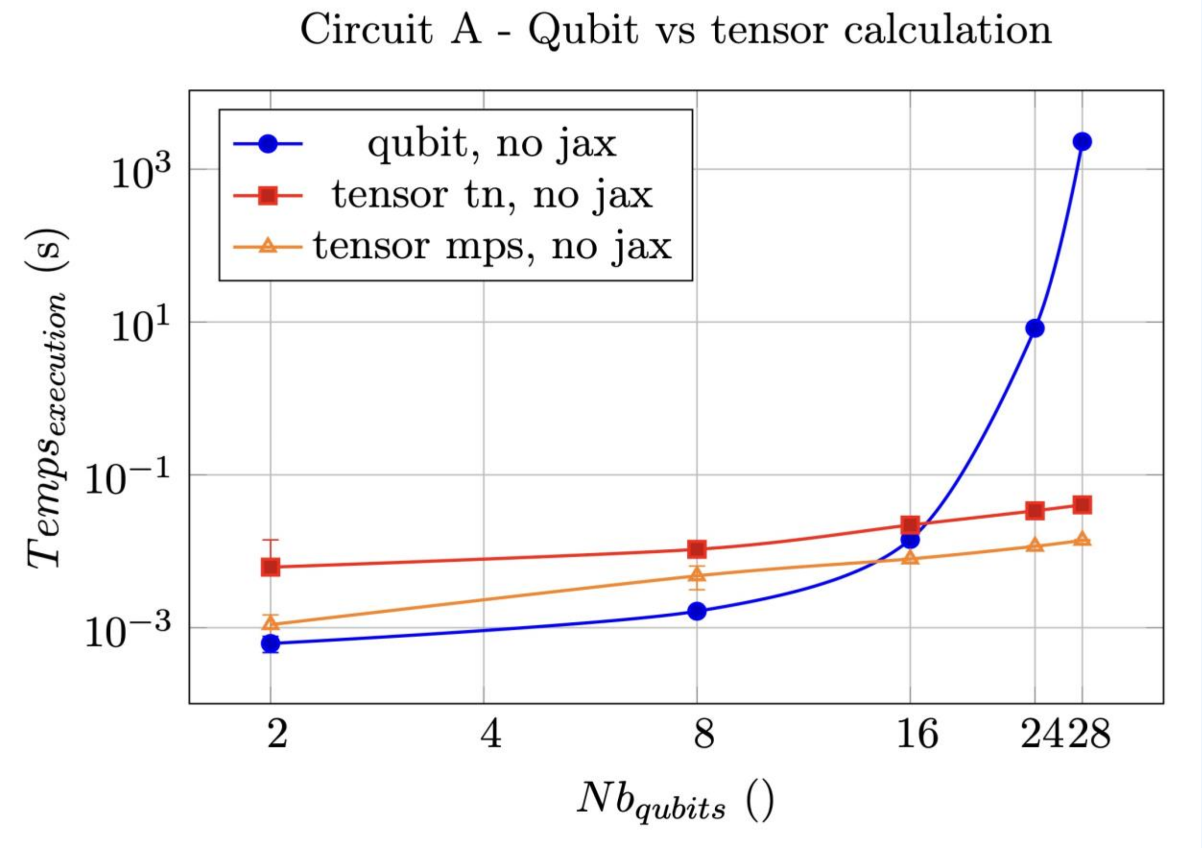
CIRCUIT A



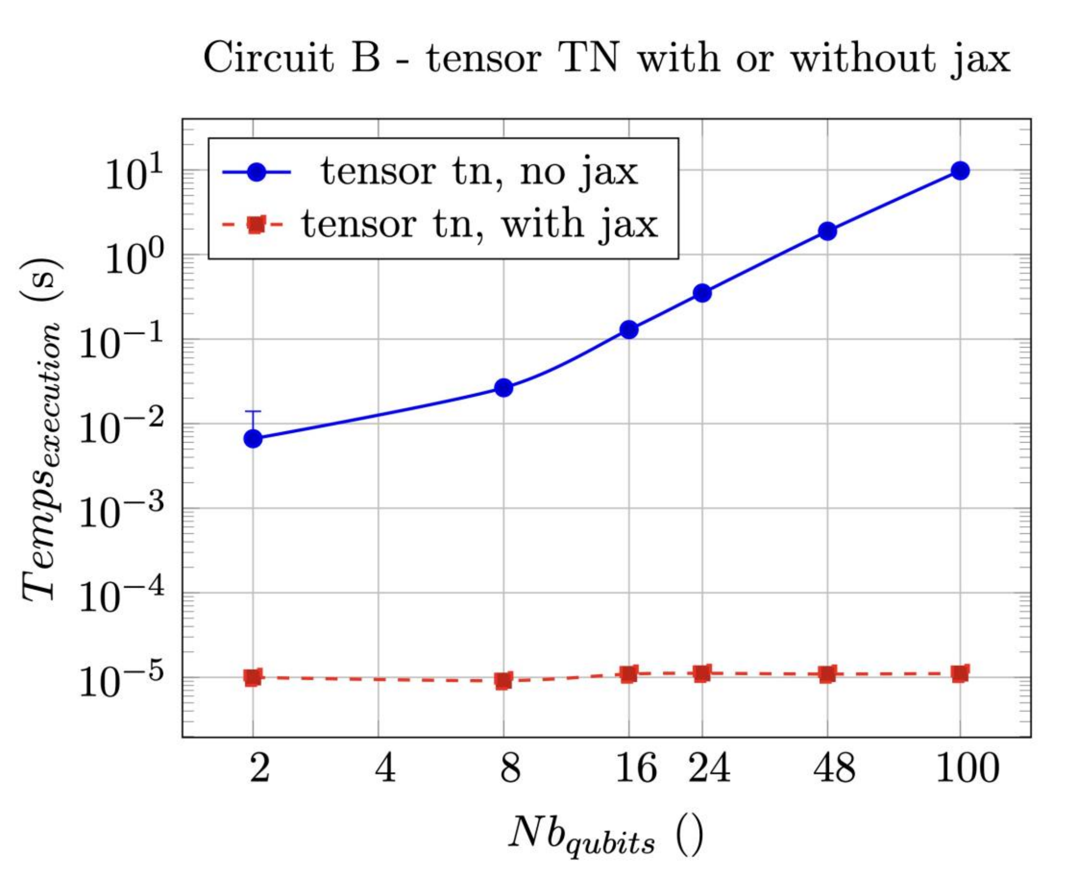
CIRCUIT B

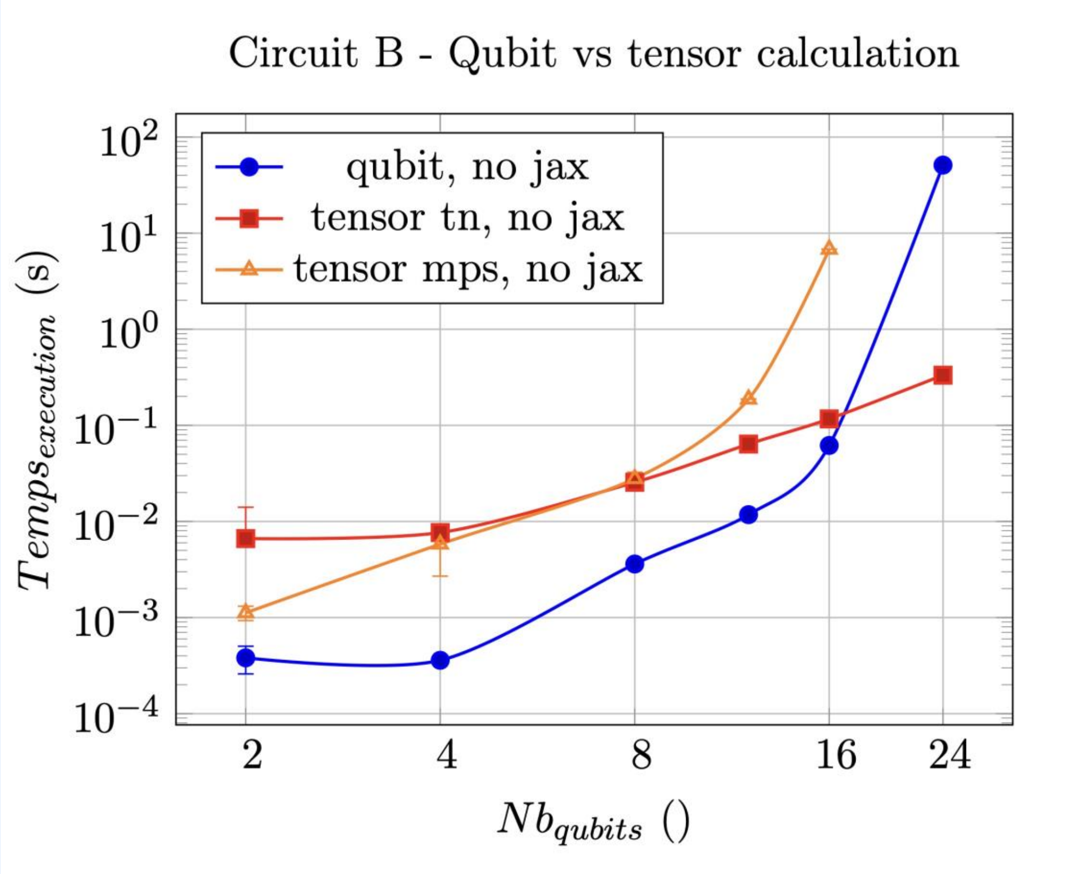
RESULTS – CIRCUIT A - WITH PARAMETERS





RESULTS – CIRCUIT B - NO PARAMETERS

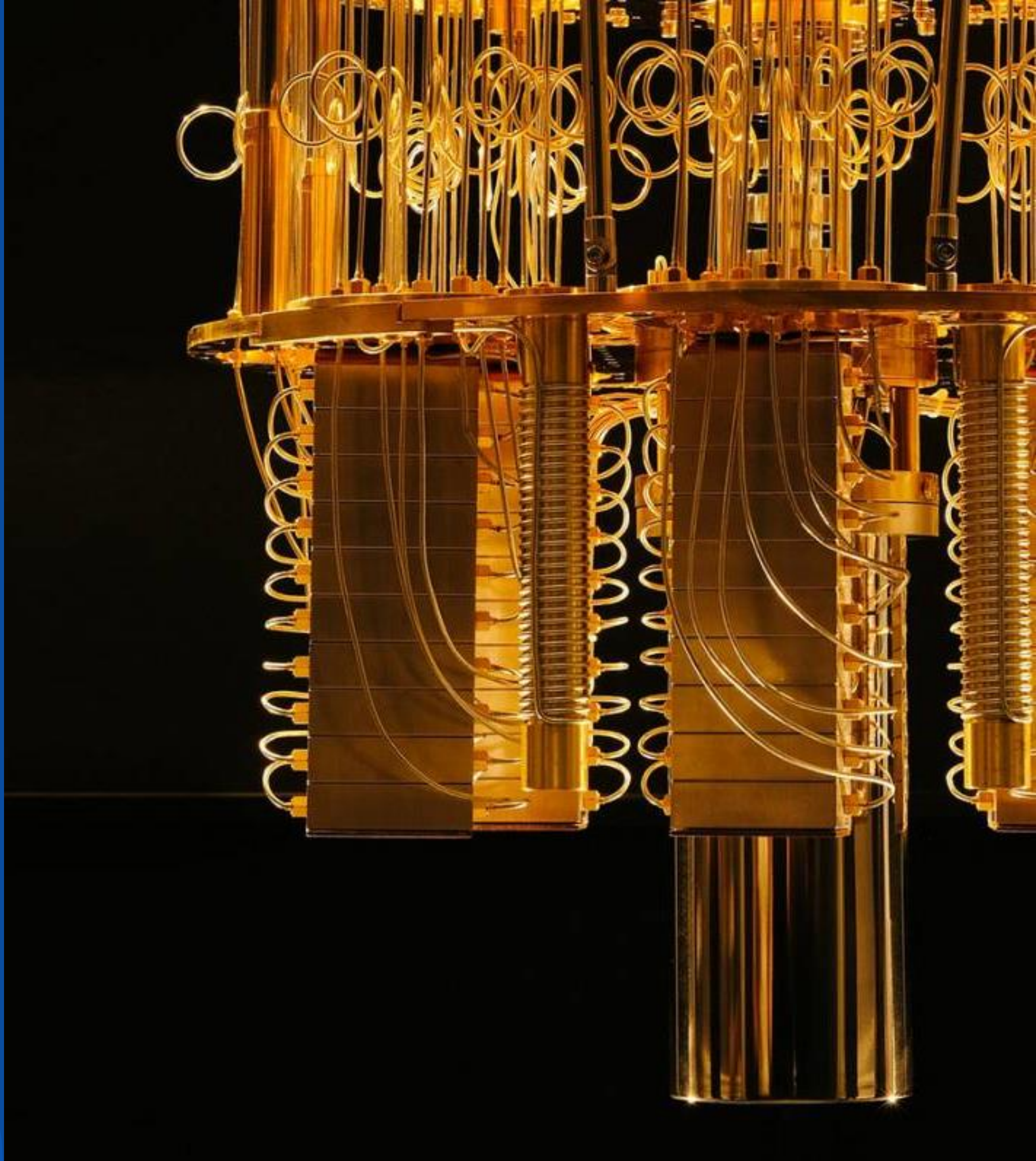




RESULTS – SUMMARY

- **Clear advantage for tensors** on default qubit
- However, it is important to **choose the tensor architecture** accordingly to the circuit design --> MPS or TN or other.
- JAX has a **variable advantage**: we noticed this advantage was clear for circuits with static input --> to take advantage of the XLA compilation.
- We will further show how we used this to **take advantage of JAX**

Application to QML



CLASSIC QML PIPELINE

- Encoding classical data points into quantum states
- Calculating a similarity matrix
- Applying Spectral Clustering Algorithm
- Evaluating with a NMI score

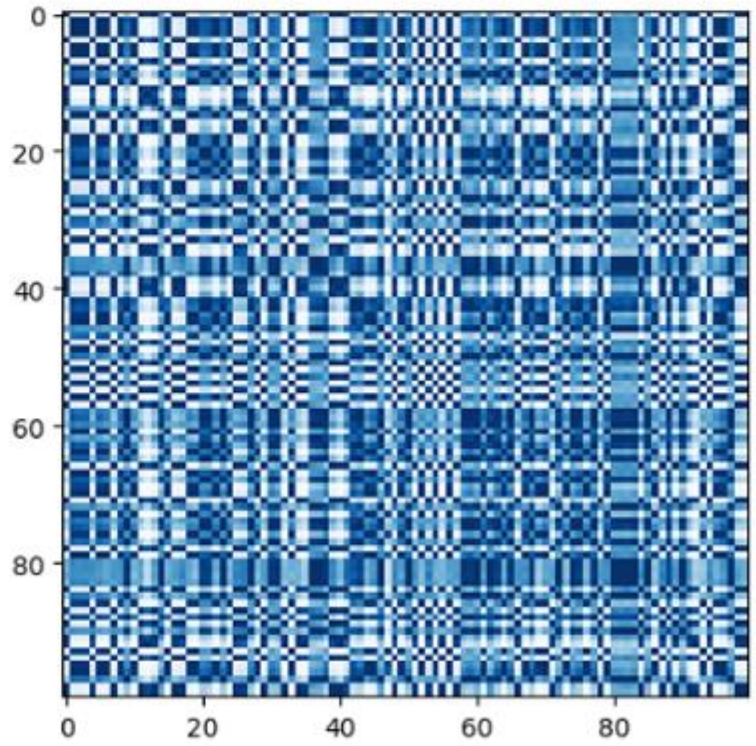


Figure 7: Similarity matrix

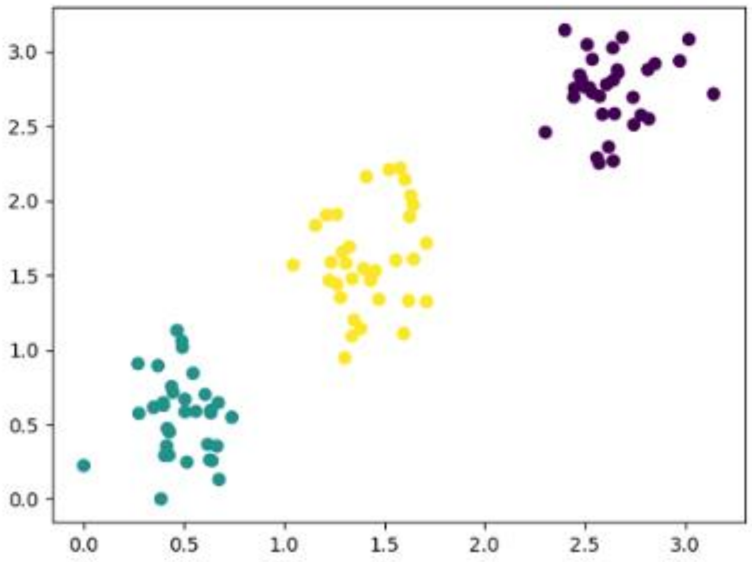


Figure 8: Clustering results

EVALUATING MORE COMPLEX DATASETS

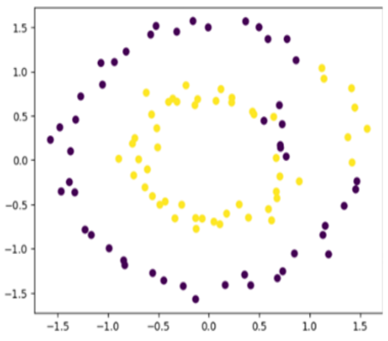


Figure 9: Donuts(NMI=0.44)

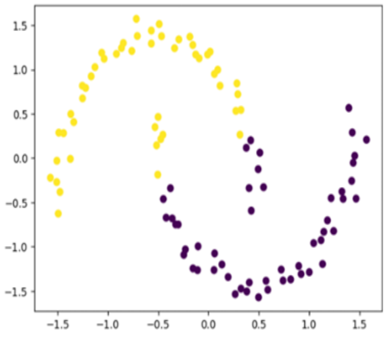


Figure 10: Moons(NMI=0.44)

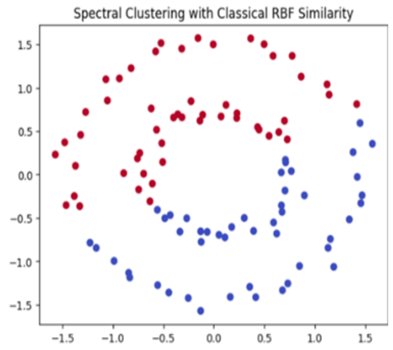


Figure 11: Gamma=2

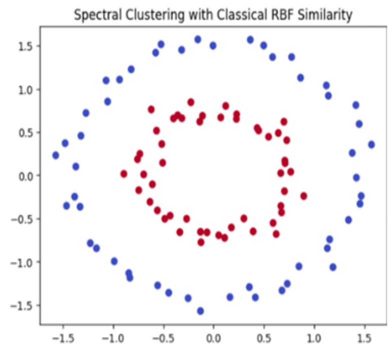


Figure 12: Gamma=50

```
dist = np.exp(similarity_adjoint(6, 2, X_moons_scaled, sim, pm) * 50)
```

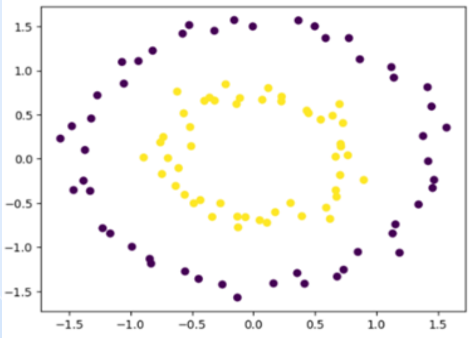


Figure 13: Donuts(NMI=1)

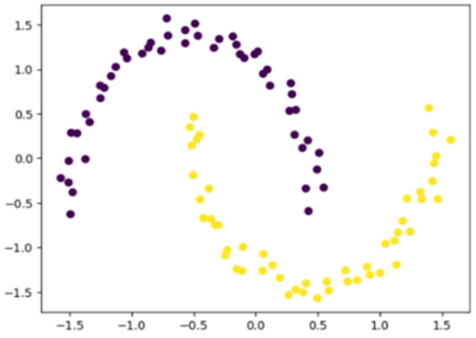


Figure 14: Moons(NMI=1)

QML IMPLEMENTATION USING PENNYLANE-JAX

MAIN DIFFERENCES WITH CLASSICAL IMPLEMENTATION:

1. Calling qml.state() to retrieve the final quantum state vector
2. Compute the fidelity outside the circuit by focusing on the amplitude of the 000…0 basis state.
3. Use a tensor-network simulator (default.tensor)
4. Leverage JAX's just-in-time (JIT) compilation

IMPROVING THE IMPLEMENTATION

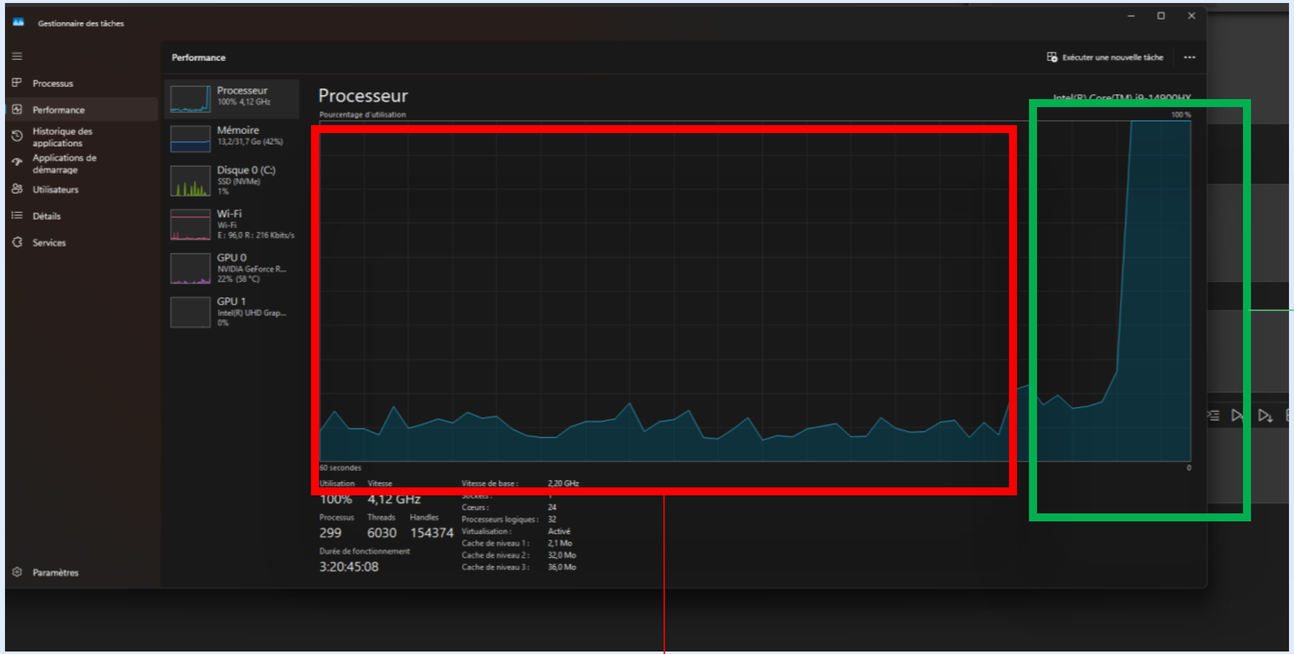
- So far: basic approach with a two parameters circuit

```
def circuit(x1, x2):
    fidelity_adjoint_circuit_pennylane(n_qubits, n_features, x1, x2)
    return qml.state() # The calculation of fidelity is done outside the circuit

circuit = jax.jit(circuit)

for i in range(n_samples):
    for j in range(i):
        state = circuit(X_jax[i], X_jax[j]).block_until_ready()
```

- Switch to only one!

```
for i in range(n_samples):
    # We create the circuit for the given x1
    circuit = x1_fixed_circuit(n_qubits, n_features, X_jax[i])
    circuit = jax.jit(circuit) # Extra cost: we run the compilation of the circuit for each i. But,
        we now only have one dynamic parameter.
    for j in range(i):
        state = circuit(X_jax[j])
```

BUT, INEFFICIENT USE OF CPUS



Qiskit

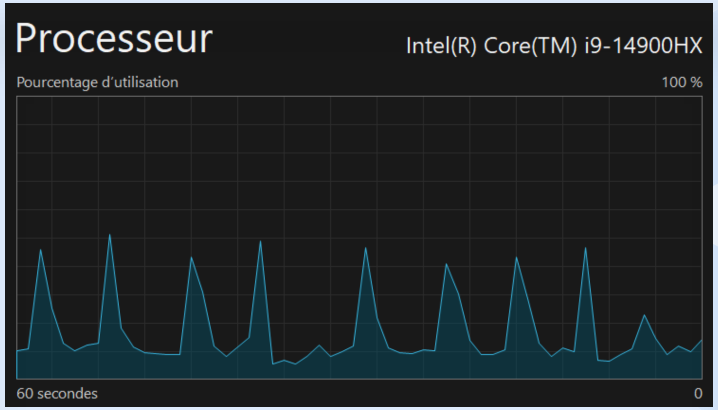
Pennylane + jax.jit

ADDITIONAL LEVEL OF PARALLELISM: PMAP

- Create batch of parameters
- Run the different circuit in parallel (mimick Sampler)

```python
circuit = x1_fixed_circuit(n_qubits, n_features, X_jax[i])
circuit = jax.jit(circuit)
p_circuit = jax.pmap(circuit) # We use the pmap function to adapt it for use with pmax

# We run for X[:i], but we cut it in batch_size for use with pmap
n_iter = i//batch_size + 1 # batch_size is defined by the number of cores allocated for jax

for j in range(n_iter):
    if j == n_iter - 1:
        batch_parameters = X_jax[j*batch_size:i]
    else:
        batch_parameters = X_jax[j*batch_size:(j+1)*batch_size]

    states = p_circuit(batch_parameters)
```

- Amazing increase in performances!
- But now, memory access limits performances




```
Total circuit time: 4.390684366226196
Total space time: 19.668492555618286
```

PENNYLANE VS QISKIT: HIGH NUMBER OF QUBITS AND FEATURES

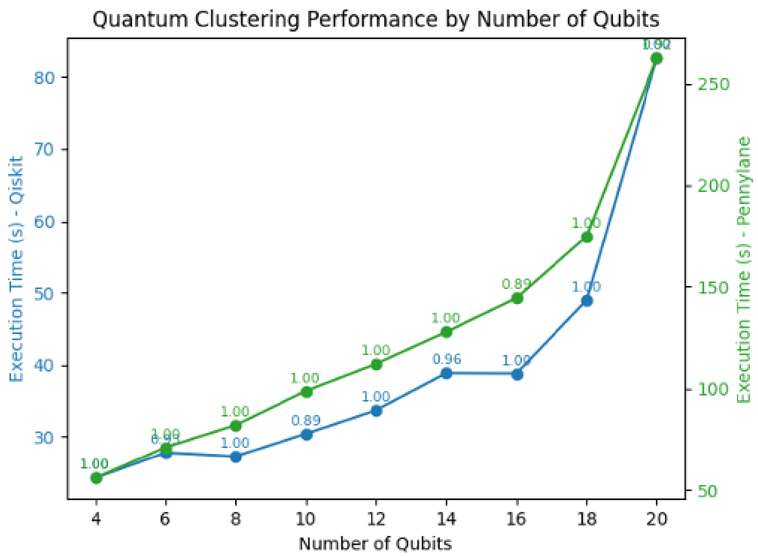


Figure 15: 4 Features

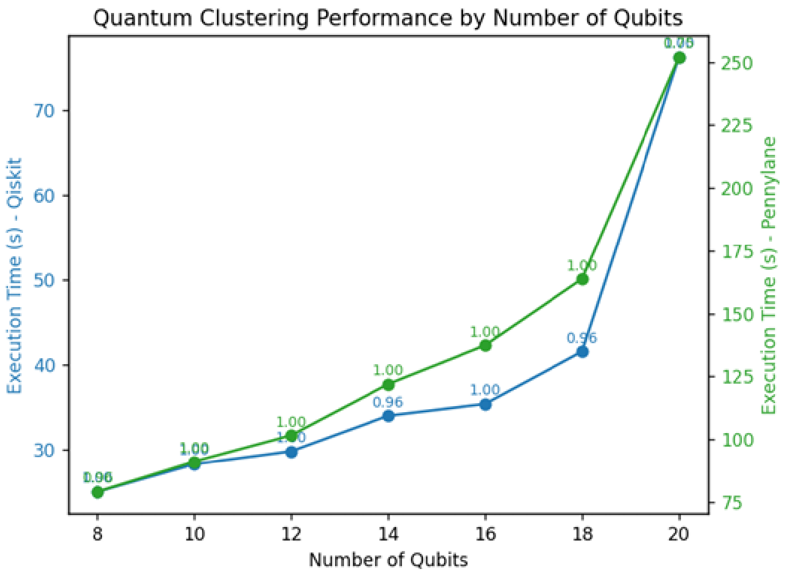


Figure 16: 8 Features

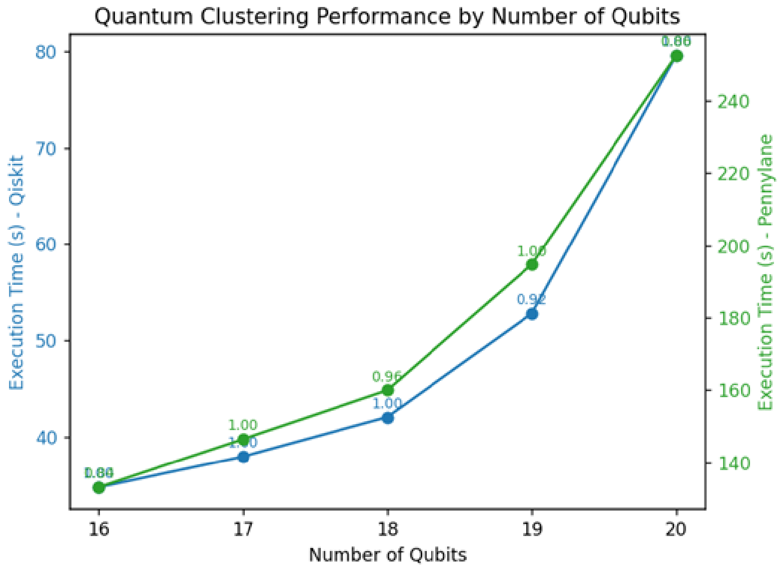


Figure 17: 16 Features

RESULTS ON REAL DATASETS

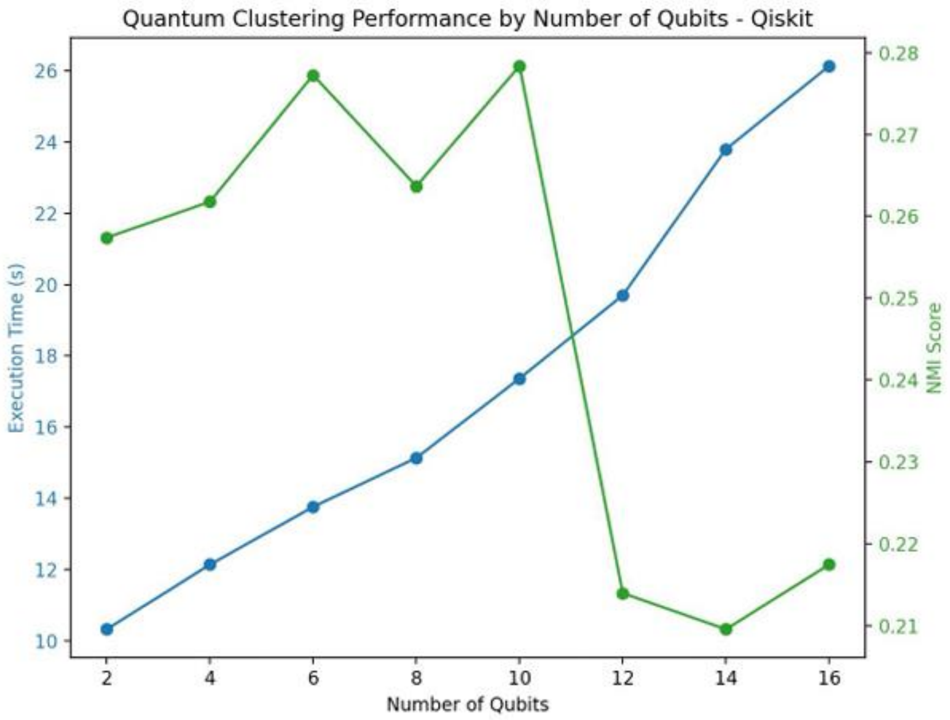


Figure 18: Number of features equal to number of Qubits

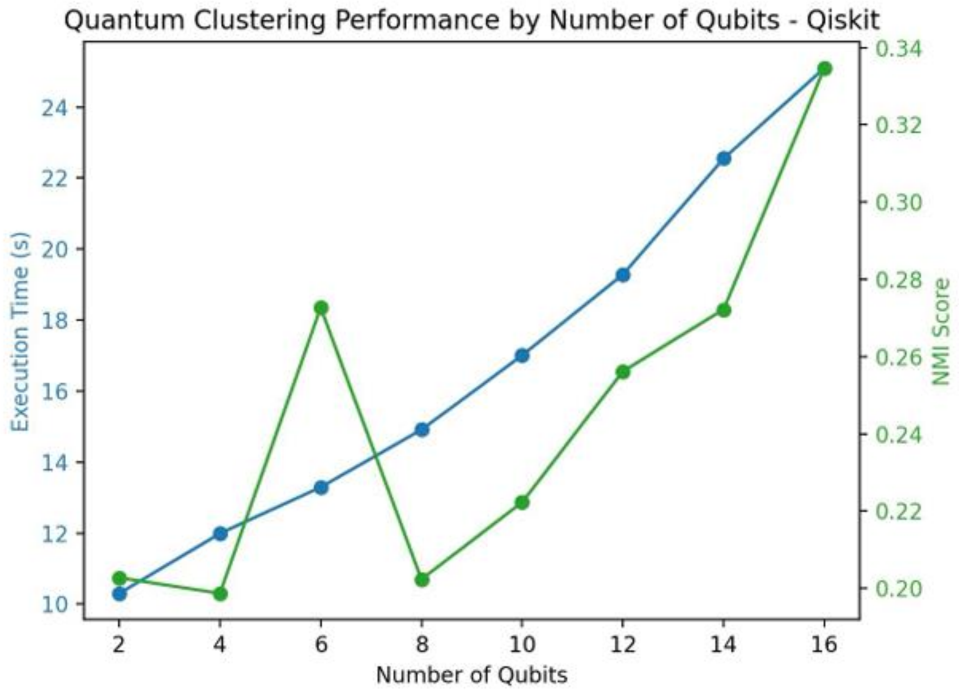


Figure 19: Twice more Qubits than features

Thanks

ANY QUESTIONS ?