QML Tensor Network Simulation

Gabriel Rochette, Alexandre Gravereaux, Seif Zaafouri

28/03/2025

1 Introduction

Quantum simulation plays a key role in developing new quantum algorithms. Indeed, it enables us to tweak different parameters, look at perfect results without being impacted by noise, and it enables the testing of algorithms without having to use an actual QPU, which are hard to come by [2]. However, quantum simulation on classic infrastructures are extremely costly, and the maximum number of qubits one can simulate is upper-bounded by the extreme cost in memory and time efficiency. For instance, in order to simulate 50 qubits, one would require $2^{50} \times 8$ bytes, so 9 PB of memory. This is by no means a small cost, and it becomes particularly important to turn to alternative representations of quantum circuits to enable simulations with a high number of qubits.

Today, the most common approach is the tensor-based approach [4]. The core idea behind this principle is to see gates as tensors, acting on a certain amount of inputs and outputs, and links between tensors as summation on indexes. This enables the contraction of quantum circuits: two neighboring tensors are transformed into an equivalent tensor through the sum of the appropriate indexes [1]. In particular, this approach presents one key benefice: the product of two tensors is an operation very similar to matrix product (particular case for 2 dimensional tensors) that we master extremely well in High Performance Computation, and it can easily be parallelized. This means that, assuming one has a tensor representation of a quantum circuit, one can drastically increase performances of calculation by shrinking the whole circuit to a smaller, equivalent tensor, through parallelized tensor multiplications. Different methods exist to fulfill this task. These methods differ by the way they convert a quantum circuit to a tensor circuit and by the way the choose to parallelize tensor products (in particular, how do they split tensors in different groups).

1.1 Scope of this project

The aim of this project is to look into perfect simulation through tensorized parallelism to address the question of high number of qubits. We will look into the different methods used to parallelize tensors and compare them. Once this first objective has been fulfilled, we will aim to apply these methods of simulation to QML. In particular, we will also look into variational kernel methods, and verify the results experimentally using real QPUs thanks to IBM's services. We will use Scikit included data or basic financial data from Kaggle to provide a real-world application of our solutions. However, the core focus of this project remains the bench-marking of different parallelization methods for tensor networks applied to quantum circuits.

1.2 Different methods of parallelization

When thinking about parallelizing tensor contraction, the main idea to bear in mind is the order in which you wish to contract them. A first approach consists in finding the optimal order for contracting two pairs of neighboring tensors, and to optimize this calculation by parallelizing this two tensor contraction operation, on GPU for instance. The second main approach is to cut the whole circuit into optimal **indexes** slices, and parallelize the contraction of each slice by turning to MPI processes to treat each of them, and then assembling the results of these contractions. A more recent method emerged, using the Girvan-Newman algorithm to detect communities of tensor and deduce optimal contraction plans. These methods have been implemented in different ways, depending on the choice of the method to parallelize and find optimal order or groups for contraction, through some premade modules. [2]

1.3 Modules found for tensor network parallelization

As we searched for modules implementing these contraction methods, we came across three main modules:

1. Jet
2. Jax
3. QXTools

QXTools seems like the most complete and efficient module, offering both the direct construction of tensor networks for quantum circuits and the application of the methods mentioned above. However, having been made for julia, we have chosen to pursue other methods in the first place, hoping to use python instead.
Jet has been specifically designed for quantum-oriented tensor network, and has been optimized to use GPU acceleration for tensor contraction.
Jax is a broader module used for high-speed computation. It showcases one remarkable advantage compared to the two other packages: it provides a function, ‘jit‘, to run just-in-time compilation based on XLA optimization, that works perfectly with another module, Pennylane, designed to build quantum circuits directly in tensor form. Thus, the combination of jax and pennylane offers a user-intuitive, simple interface, to create quantum circuits in tensor form and parallelize the contraction.
At this point of the project, we have mostly explored these two modules in order to obtain some basic results to quantify the advantages of the tensor approach for simulations compared to the classical approach of quantum simulation.

1.4 Basic tensor theory

A tensor is basically the generalization of the concept of matrix. It is a function, represented by an object of dimensions $n_1 \times n_2 \times \cdots \times n_m$ (in the space $\mathbf{C}^{n_1.n_2...n_m}$). A tensor graph, is a succession of tensors with links between their indexes. A link between two tensors is akin to a summation of the specified indexes. This notation enables the generalization of matrice-matrice products and matrice-vectors product. For instance, the latter is represented as a one-dimensional tensor with a connection to a two-dimensional tensor, the index being that of the columns of the two-dimensional tensor (the summation occurs along the columns of each given line) [1].

The main idea to use tensor networks in quantum informatics is to represent a quantum circuit as a tensor network. Multiple methods exist to do so, as represented in figure 1. We have not yet perfectly understood the advantages of each method, but the overall idea is that it depends mostly on the precision you aim for, and especially, the level of intrication of the quantum circuit.

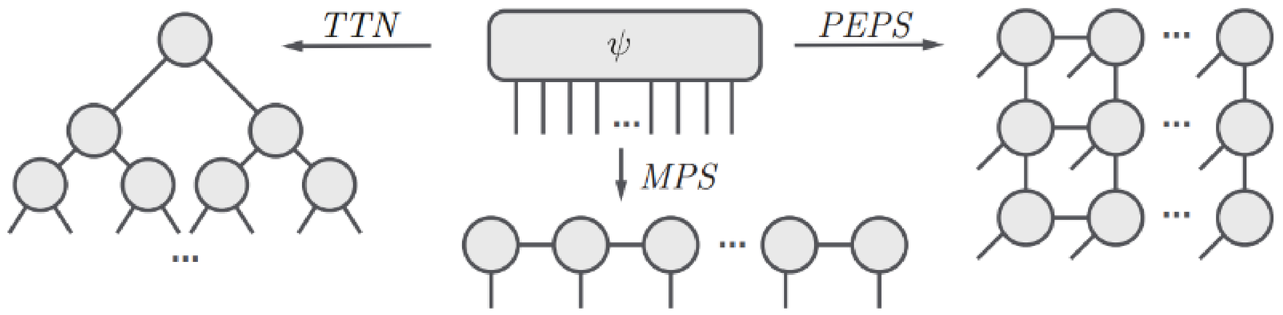


Figure 1: Different methods to go from a quantum gate to a tensor network, reference: Quantum Tensor Networks in a Nutshell

Pennylane offers to use two methods, the MPS and the TTN ones. At this point of the project, we have not yet tried to quantify the impact of using one over the other on the results of calculations. Pennylane implements the possibility to build the tensor graph directly from a quantum circuit, according to the specified method (MPS by default). We have kept this approach to obtain the results presented in the next section.

2 Related work

2.1 Tensor-network quantum circuits

The article "Tensor-network quantum circuits", published in **March 2022** by *Diego Guala, Esther Cruz, Shaoming Zhang and Juan Miguel* on PennyLane, explains how tensor networks can serve as templates for designing quantum circuits that mimic the connectivity of well-known tensor network structures like **matrix product states (MPS) and tree tensor networks (TTN)** . It starts by introducing tensors as multidimensional arrays of scalars, vectors, and matrices, and discusses tensor contraction - key to forming these networks. The tutorial then shows how to translate these ideas into quantum circuits by using PennyLane's templates. Specifically, it demonstrates **how to transform a variational block** (built from RX, RY, and CNOT gates) into the shape of an MPS or TTN, enabling one to craft circuits with the desired entanglement structure.

The article further explores different implementations by varying the depth and width of the blocks—using examples like StronglyEntanglingLayers and SimplifiedTwoDesign—and even shows how to extend the idea to TTN architectures. A compelling part of the tutorial is its **application to a machine learning task**: classifying the bars and stripes dataset. Here, the circuit encodes image data via BasisState preparation, processes it through a TTN-based circuit, and outputs measurements that are used to label the images as either bars or stripes, with training performed using gradient descent.

This work builds upon foundational research such as Huggins [5] and Orús [6], situating itself within the broader effort to leverage tensor networks for efficient simulation and processing of quantum systems.

2.2 Using JAX with PennyLane

The tutorial "JAX Transformations", in **April 2021** by *Chase Roberts*, on PennyLane shows how to leverage JAX's powerful tools to accelerate and differentiate quantum circuits within the PennyLane framework. The article explores **just-in-time (JIT) compilation, automatic differentiation (via grad), and vectorized operations (via vmap)**.

When you decorate a function with JAX's JIT, what happens under the hood is that the function is **traced and compiled into optimized machine code using XLA** (Accelerated Linear Algebra). This means that your Python code, which would normally run through an interpreter, is instead transformed into a highly efficient form that runs much faster on supported hardware like CPUs or GPUs.

In essence, the work demonstrates how to define a quantum circuit using PennyLane and then use JAX's transformations to not only speed up its evaluation but also compute gradients efficiently, which is particularly useful in quantum machine learning applications.

The article doesn't include a formal bibliography, but its approach aligns with research trends in integrating automatic differentiation and JIT compilation—techniques, inspired by the same article on tensors [6]

3 Work done

3.1 Pennylane and tensor representation

We first worked with the Pennylane module. This module is powerful as it provides a DefaultTensor device, based on quimb Python library for tensor network manipulations. The default.tensor device is well-suited for simulations of circuits with tens, hundreds, or even thousands of qubits as long as the degree of entanglement within the circuit remains manageable.

The default.tensor device can simulate quantum circuits using two different computational methods. The first is the matrix product state (MPS) representation, and the second is the full contraction approach using a tensor network (TN). **For the actual work, we used the default MPS representation**, as we will emphasize our work on tensors vs qubits, and on parallelization.

For this first step, we used a simple Had + CNOT entangled circuit, as shown below (with 5 qubits):

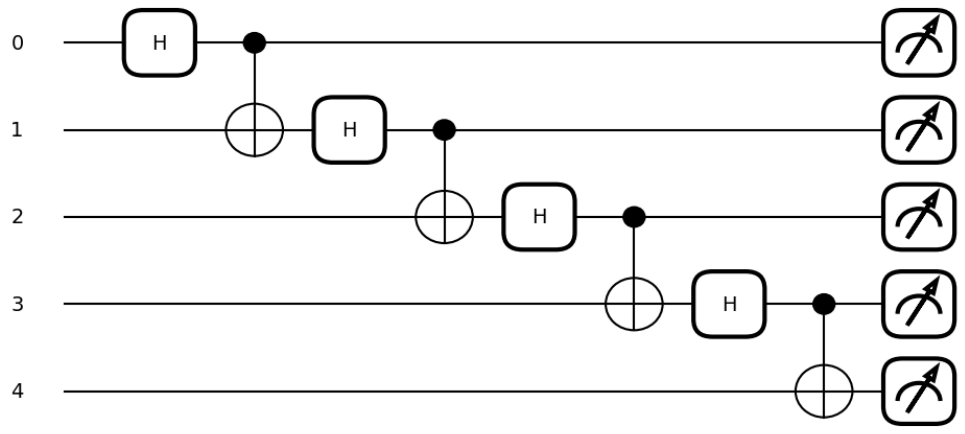


Figure 2: Circuit 1

Then, with the following code, we are able to get the tensor representation in MPS of the circuit. We first set a device, then define the circuit and draw it:

```
[...]
# Instantiate the device with the MPS method and the specified kwargs
dev = qml.device("default.tensor", method="mps", **kwargs_mps)

[...]
@qml.qnode(dev)
def circuit(n):
    for i in range(n):
        qml.Hadamard(wires=i)
        qml.CNOT(wires=[i, i+1])
    return qml.state()

# Drawing the circuit
```

```
dev.draw(circuit, show_inds=True, return_fig=True)
```

We then get the following representation:

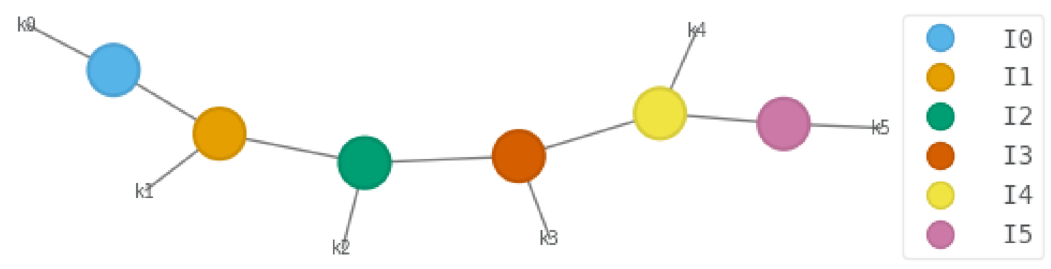
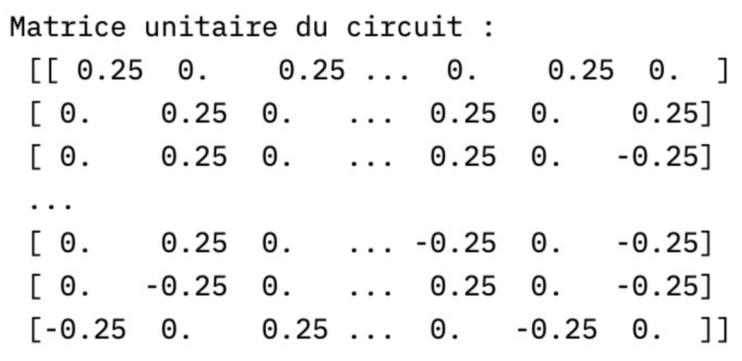


Figure 3: Circuit 1 - tensor representation

We are also able to get the unitary matrix with:

```
unitary = qml.matrix(circuit, wire_order=range(n_qbits+1))(n_qbits)
```

as below:



```
Matrice unitaire du circuit :
 [[ 0.25  0.    0.25 ...  0.    0.25  0.  ]
 [ 0.    0.25  0.    ...  0.25  0.    0.25]
 [ 0.    0.25  0.    ...  0.25  0.   -0.25]
 ...
 [ 0.    0.25  0.    ... -0.25  0.   -0.25]
 [ 0.   -0.25  0.    ...  0.25  0.   -0.25]
 [-0.25  0.    0.25 ...  0.   -0.25  0.  ]]
```

Figure 4: Matrice du circuit 1

3.2 JAX exploration with a simple circuit

We tried 4 different configurations, in order to confirm the advantage offer by the tensor and by jax:

- with default.qubit, but without jax
- with default.tensor, but without jax
- with default.qubit and with jax
- with default.tensor and with jax

Here is an example of the code we used for "default.tensor" case:

```
import pennylane as qml
import numpy as np
from time import time

qubit_range = [20, 24, 28, 30, 100, 500, 1000]
timings = []

for n_qubits in qubit_range:
    weights = np.random.random(size=(n_qubits - 1, 2))

    dev = qml.device("default.tensor", wires=n_qubits)

    @qml.qnode(dev)
    def circuit(weights):
        for i in range(n_qubits - 1):
            qml.RX(weights[i, 0], wires=i)
            qml.RY(weights[i, 1], wires=i+1)
            qml.CNOT (wires=[i, i+1])
        return qml.expval(qml.Identity(0))

    start = time()
    result = circuit(weights)
    duration = time() - start

    print(f"{n_qubits} qubits: {duration: 2f}s")
```

We change the device with default.qubit as follow:

```
dev = qml.device("default.qubit", wires=n_qubits)
```

With jax, we convert the weights in an adapted format, we specifiy the interface for qml and we parallize the circuit through jax.jit:

```
weights_jax = jnp.array(weights)
[..]
@qml.qnode(dev, interface="jax")
[..]
# Circuit parallelization
result = parallel_circuit(weights_jax)

# Circuit already parallelized: the time is only for execution
start = time()
result = parallel_circuit(weights_jax)
duration = time() - start
```

3.3 Circuits used for testing

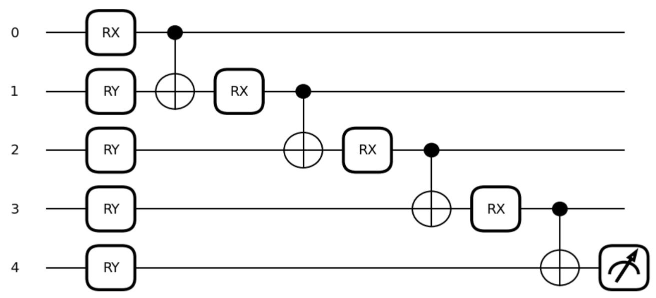


Figure 5: Circuit A - non static imput

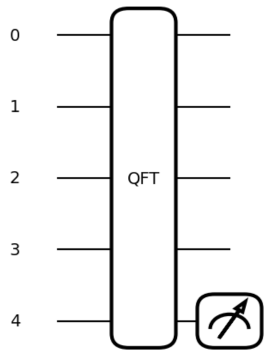


Figure 6: Circuit B - static imput

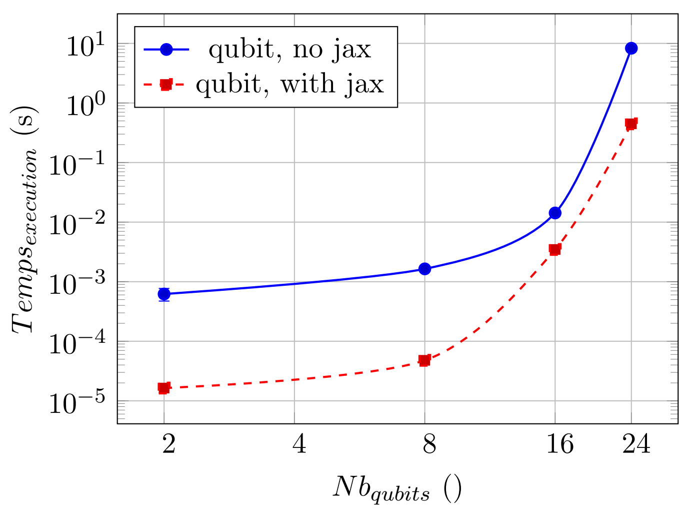
3.4 Performance and Plots

3.4.1 Diverse notes

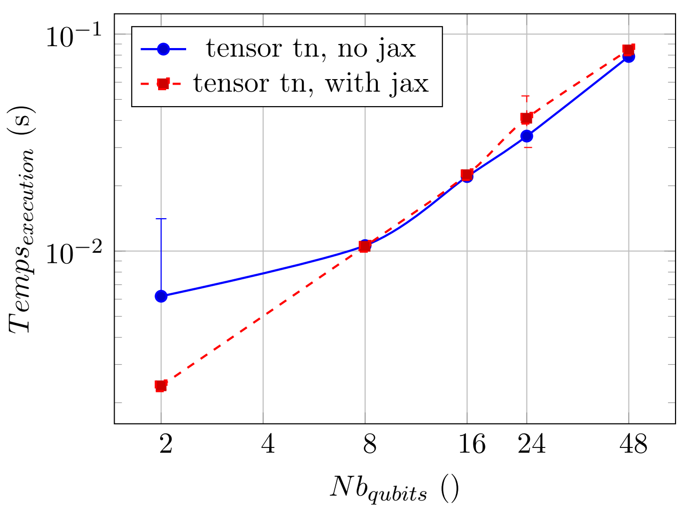
- The measurements were performed on *Python 3.10.12*, using the modules *pennylane*, *jax*, and possibly *numpy;*
- The machine that carried out the measurements is equipped with an *Apple M2 Pro* processor: *ARM architecture, 10 cores, including 4 at 2.42 GHz and 6 at 3.5 GHz;*
- For measurements involving JAX, the recorded time is that of the **second execution of the circuit**. This ensures that only the execution time is measured, without accounting for compilation time.
- For each data point, **10 measurements were performed**. The displayed value corresponds to the arithmetic mean of these measurements, and the uncertainty is calculated using the Type A method with a *Student's coefficient of 2*, corresponding to *a confidence level between 90% and 95%*.
- **Circuit A** has a variable input (weights) whereas **Circuit B** is always the same with static input.
- We get the return of the functions with *qml.expval*(): this returns a scalar product and is faster than using *qml.state*() to get the complete state. In the second part of this project, we'll use rather *qml.state*().

3.4.2 Circuit A - performance plots

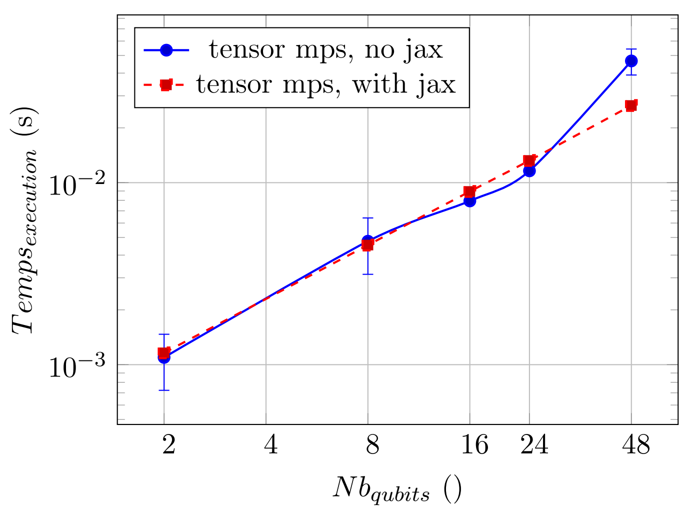
Circuit A - Qubit with and without jax



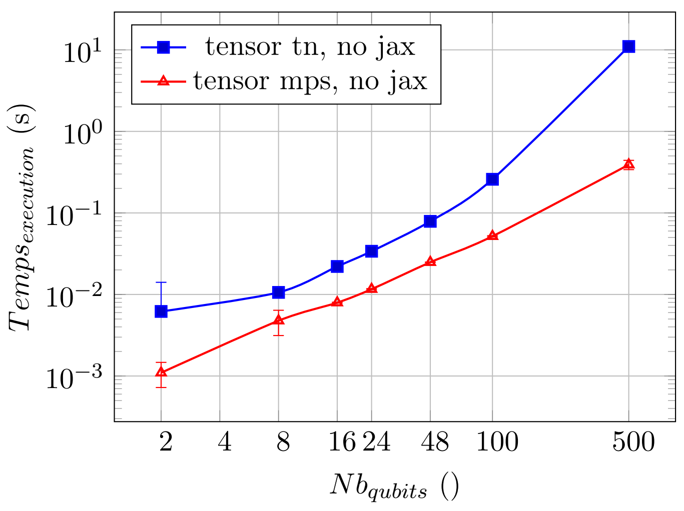
Circuit A - Tensor TN with and without jax



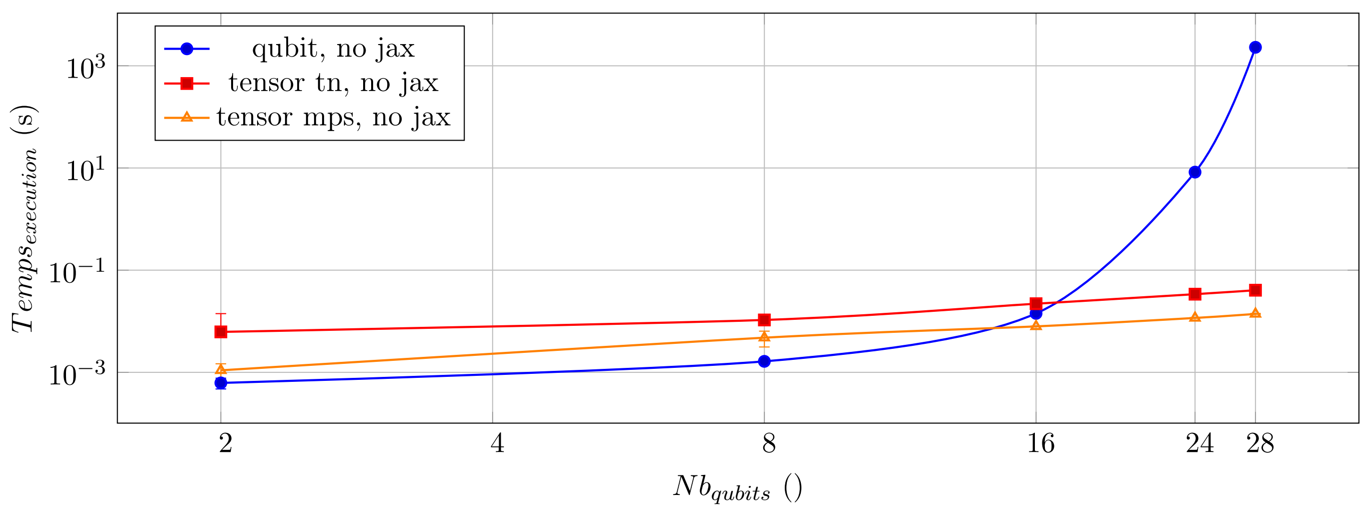
Circuit A - Tensor MPS with and without jax



Circuit A - Tensor TN vs tensor MPS



Circuit A - Qubit vs tensor calculation

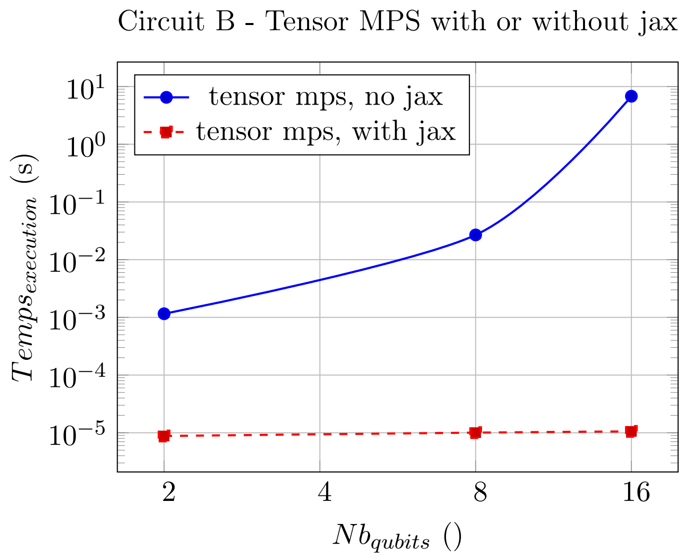


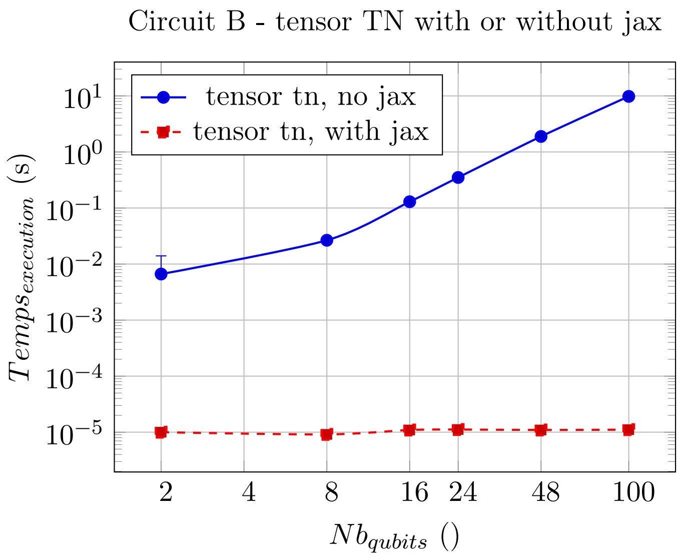
Circuit A - Performance analysis:

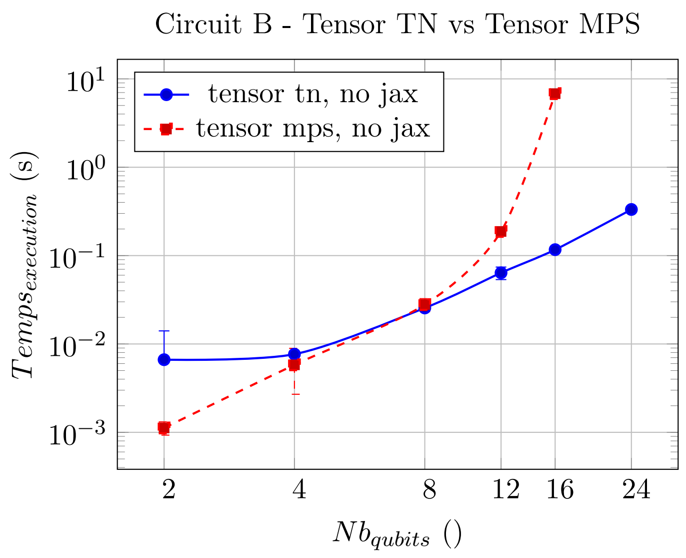
- We can first confirm that we have a great advantage in time using tensor rather than qubit default method. This advantage is clear when $nb_{qubits} > 16$. For example, the time advantage is **more than x800** for 24 qubits (it seems we have an exponential cost for qubit and a linear one with tensor)
- With default.qubit, we notice that the use of jax offers a **slight increase in performance** compared to computation without (variable gain, between x3 and x20).
- With default.tensor (MPS or TN), we can notice that there is **no advantage using jax**. Indeed, it seems that for a circuit with variable input, the compilation advantage offered by jax is gone.
- Last, we observe a **slight increase in time performance** for the default.tensor mps network. However, this network is less precise than the tn one, and this performance advantage really depends on the circuit design (see circuit B)

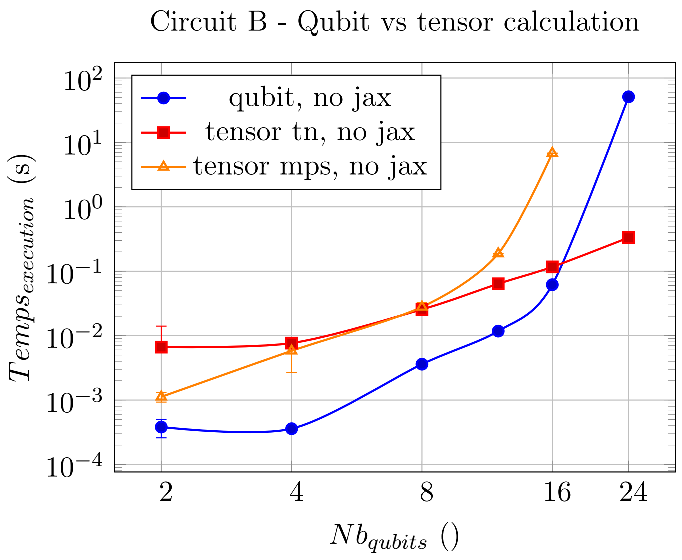
Note: jax is conceived to be used on several CPU or GPU, which was not the case on our machines. This could be another source of performance gain we should keep in mind.

3.4.3 Circuit B - Performance plots









Circuit B - Performance analysis:

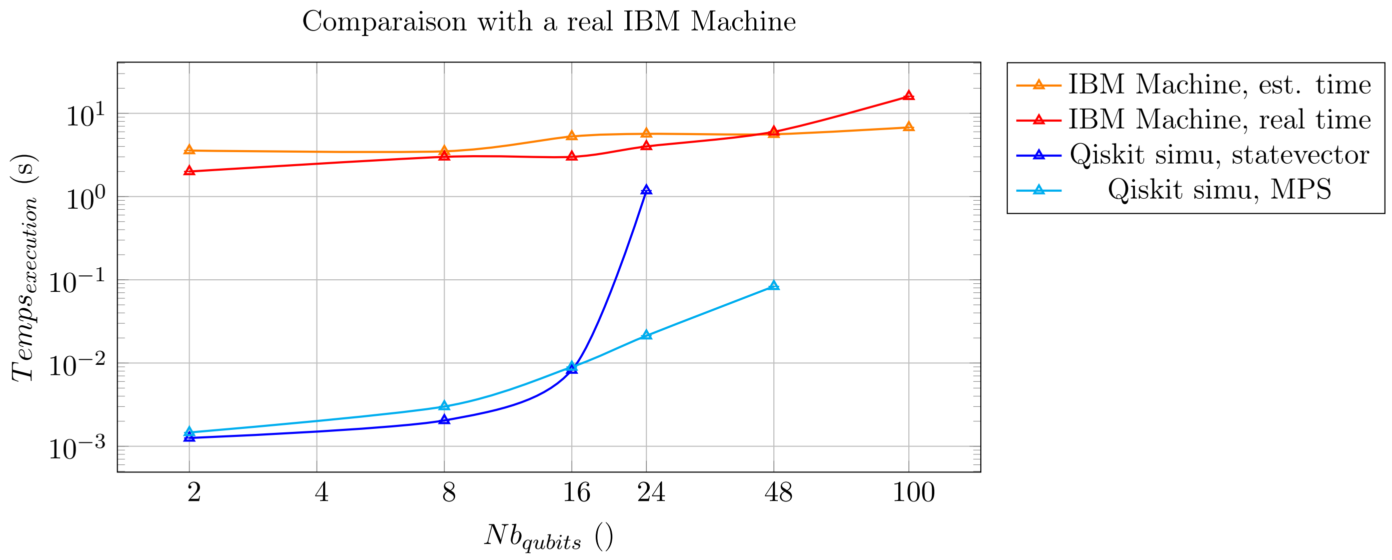
- We notice right away that **jax provide an incredible advantage**, as the circuit is static without variable input. The tensor MPS and tensor TN without jax execution of circuit B is approximately linear (maybe squarred for mps), whereas the **execution time with jax is constant**, staying at about $10^{-5}s$.
- Then, we observe a **clear performance increase for the default.tensor tn** network, which is also the more precise one. This is different from circuit A, and we can thus say that the choice of the network (tn or mps) should really **depends on the circuit design**.
- Last, we oberve that we have an advantage in time using tensor tn rather than tensor mps or qubit default method. This advantage is clear when $nb_{qubits} > 16$. And here, the tensor mps is almost as bad as the default qubit method. For example, the time advantage is **more than x150** for 24 qubits (it seems we have an exponential cost for qubit / tensor mps, and a linear one with tensor tn)

Note: jax is conceived to be used on several CPU or GPU, which was not the case on our machines. This could be another source of performance gain we should keep in mind.

Comparaison with real IBM Machine

- Computations where performed on ibm sherbrooke QPU (127 qubits)

- Computations where performed with circuit B (QFT)



4 Quantum Machine Learning

4.1 Pipeline

In this section, we demonstrate a quantum method for clustering synthetic data using quantum fidelity as a similarity metric. We start by encoding classical data points into quantum states and measuring their overlap, we obtain a quantum-generated similarity matrix. We then apply a classical spectral clustering algorithm on this matrix and evaluate the performance using the Normalized Mutual Information (NMI) metric.

4.1.1 Data Generation and Preprocessing

We use `scikit-learn`'s `make_blobs` function to generate `NB_DATA` points distributed among `NB_CLUSTER` centers in two-dimensional space. Each point has two features.
Since quantum circuits typically encode data into angles (e.g., rotation angles of qubits), we normalize the data into an interval $[0, \pi]$. This is achieved by first scaling the data to $[0, 1]$ then multiplying by π.

4.1.2 Quantum Fidelity as a Similarity Metric

Our quantum circuit uses *embedding* gates (e.g., rotation gates) to load each data point into qubit states. For data points x_i and x_j, we construct two embedding circuits and append the inverse of the second onto the first, measuring the probability of the all-zero bitstring ($|0 \dots 0\rangle$). This probability directly reflects the *fidelity* (overlap) of the two quantum states:

$$F\big(|\psi_i\rangle, |\psi_j\rangle\big) = |\langle\psi_i|\psi_j\rangle|^2. \tag{1}$$

We compute this for every pair (i, j), populating a similarity matrix **dist** of size $n \times n$, where n is the number of data points.

4.1.3 Spectral Clustering and NMI Score

Using this quantum fidelity-based similarity matrix, we apply a classical spectral clustering algorithm. Spectral clustering uses an affinity matrix (here, our quantum fidelity matrix) and partitions the graph formed by data points in a high-dimensional space.

To evaluate the clustering performance against the known labels y, we use the Normalized Mutual Information (NMI) metric:

$$\mathrm{NMI}(labels_{pred}, labels_{true}) \in [0, 1], \tag{2}$$

4.1.4 Results

We can visualize the fidelity-based similarity matrix as a heatmap, where darker colors indicate higher similarity. We then plot the clustering results (each data point colored by its cluster label).

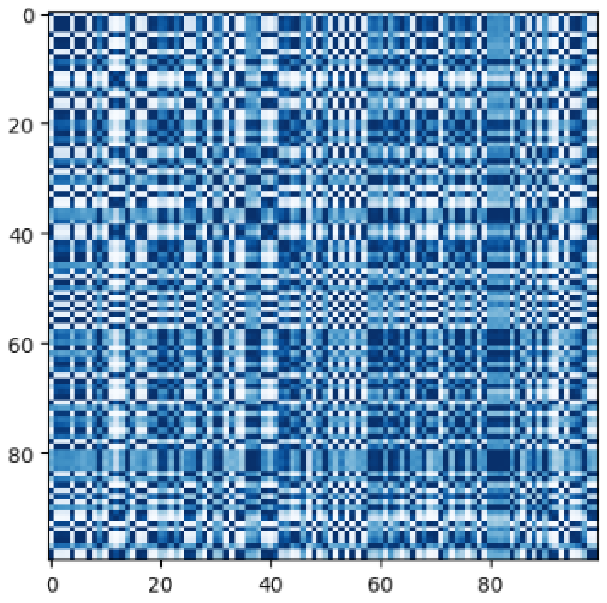


Figure 7: Similarity matrix

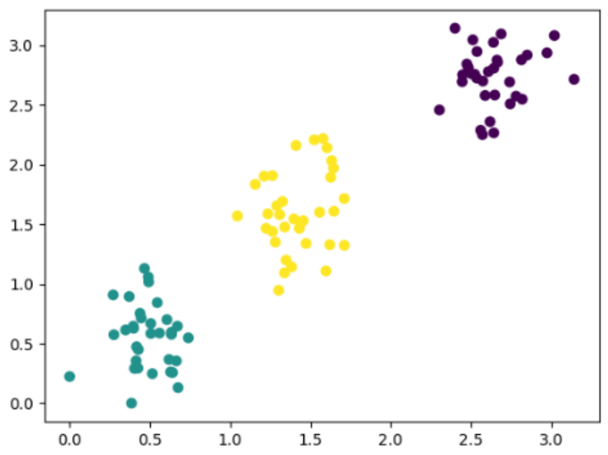


Figure 8: Clustering results

4.2 Trying more complex datasets

4.2.1 Moons and Donuts datasets

In addition to the synthetic blobs dataset, we explored more complex shapes by testing our quantum fidelity approach on both the two-moons dataset and a donut-shaped dataset. When we applied the same quantum settings and spectral clustering, the best NMI score we achieved was around 0.44.

To investigate whether the issue stemmed from the quantum fidelity approach or the clustering algorithm, we replaced our similarity matrix with an RBF kernel (using `gamma=2`). However, the clustering still failed to separate. By tuning the RBF kernel to a much higher `gamma=50`, we dramatically improved clustering performance.

Following this insight, we sharpened our quantum similarity matrix by exponentiating the fidelity values with a large scaling factor:

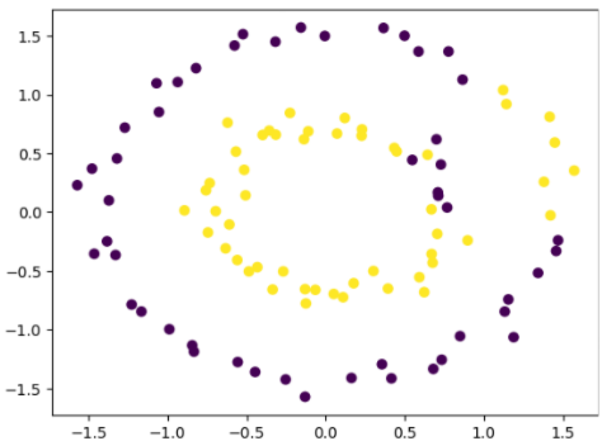


Figure 9: Donuts(NMI=0.44)

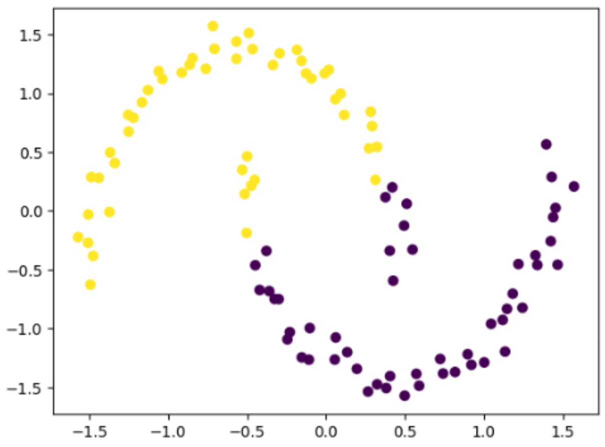


Figure 10: Moons(NMI=0.44)

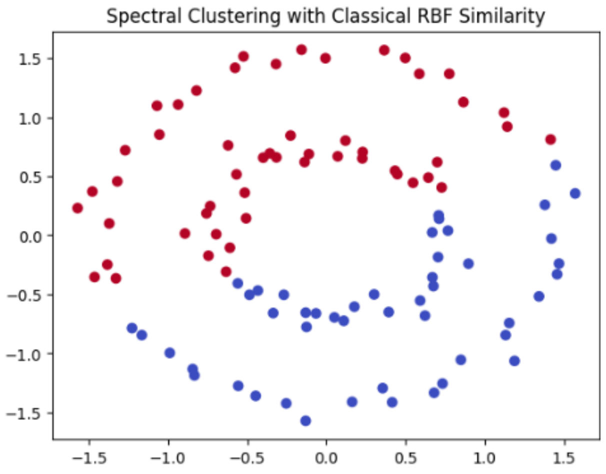


Figure 11: Gamma=2

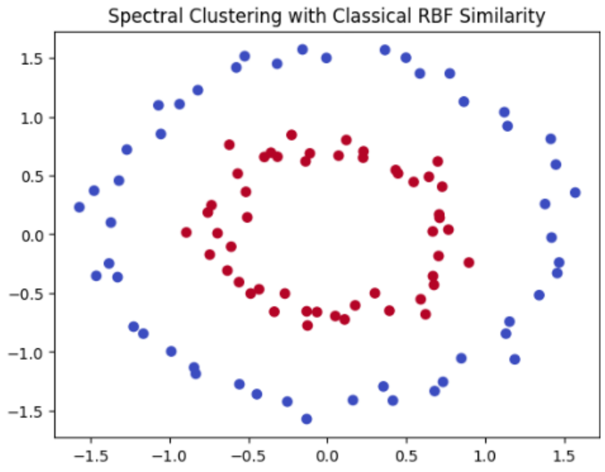


Figure 12: Gamma=50

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

This exponential sharpening enhanced contrast in the similarity matrix and led to a perfect clustering result (NMI = 1). Thus, adjusting the quantum fidelity-based kernel can be essential for capturing more intricate structures in real or synthetic data.

4.3 Implementation in PennyLane-JAX

Our fidelity-based similarity approach can be carried out with the *PennyLane* library in conjunction with *JAX*. The aim is to utilize a tensor-network simulator (`default.tensor`) and leverage JAX's just-in-time (JIT) compilation for performance gains.

- **Defining the circuit:** The first embedding is straightforward (applying Hadamard and rotation gates per feature). The second embedding is then applied *in reverse* (the adjoint operation).
- **Returning the quantum state:** Rather than measuring directly inside the circuit, the PennyLane function calls `qml.state()` to retrieve the final quantum state vector. This allows us to compute the fidelity *outside* the circuit by focusing on the amplitude of the $|000\ldots0\rangle$ basis state.

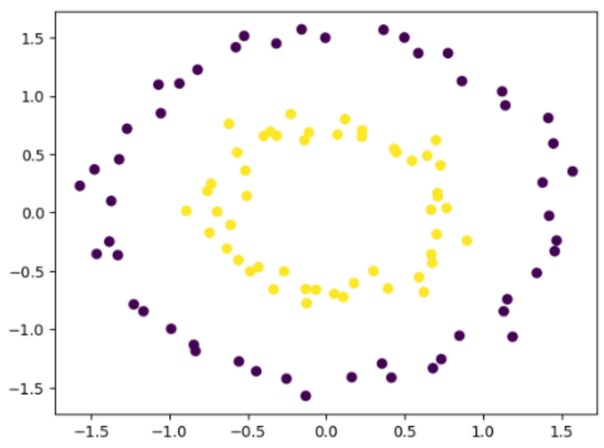


Figure 13: Donuts(NMI=1)

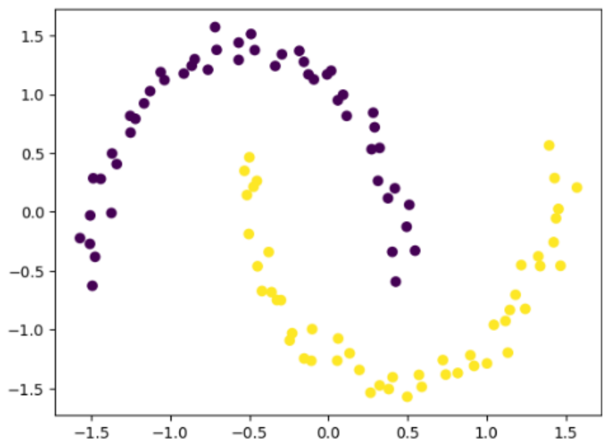


Figure 14: Moons(NMI=1)

- **Constructing the similarity matrix:** We then define a `similarity_matrix` function that:
  1. Converts the classical dataset X into a JAX array.
  2. Creates a PennyLane device with the specified number of qubits
  3. Defines a JAX-jitted quantum node (using `qml.qnode`).
  4. For each pair of data points, runs the quantum node to obtain the full state vector, and extracts the fidelity as the squared amplitude of the all-zero state amplitude.
  5. Fills an $n \times n$ matrix (where n is the number of samples) with these fidelity values.
- **Accelerating with JAX:** The function is wrapped with `@jax.jit`, this JIT compilation is especially helpful when repeatedly calling the same circuit structure for multiple data point pairs, potentially leading to a significant speedup in simulation.
- **Outcome:** The result is a quantum fidelity matrix (like in the Qiskit implementation) that can be used as a kernel in spectral clustering or any other machine learning method requiring a similarity measure.

Embedding circuit in PennyLane:

```
def embedding_circuit(nb_qubits, nb_feature, dataPoint):
    """
        nb_qubits : nomber of qubits in the circuit
        nb_feature : nb_feature of data
        This function return the circuit of embedding
    """
    for i in range(nb_qubits):
        qml.H(wires=i)
        qml.RZ(dataPoint[i % nb_feature], wires=i)
        qml.RY(dataPoint[i % nb_feature], wires=i)
```

Inverse embedding circuit in PennyLane:

```
def inverse_embedding_circuit(nb_qubits, nb_feature, dataPoint):
    """
        nb_qubits : nomber of qubits in the circuit
        nb_feature : nb_feature of data
        This function return the circuit of embedding
    """
    for i in range(nb_qubits):
        qml.RY(-dataPoint[i % nb_feature], wires=i)
        qml.RZ(-dataPoint[i % nb_feature], wires=i)
        qml.H(wires=i)
```

Fidelity adjoint circuit in PennyLane:

```
def fidelity_adjoint_circuit_pennylane(nb_qubits, nb_feature, x1, x2):
    """
        nb_qubits : nomber of qubits in the circuit
        nb_feature : nb_feature of data
        This function insert the data in the circuit for fidelity with adjoint method
    """
    embedding_circuit(nb_qubits, nb_feature, dataPoint=x1)
    inverse_embedding_circuit(nb_qubits, nb_feature, dataPoint=x2)
```

Similarity matrix in PennyLane:

```
def similarity_matrix(X,n_features, n_qubits):
    n_samples = len(X)
    sim_matrix = np.eye(n_samples)
    X_jax = jnp.array(X)

    # We define the tensor circuit
    dev = qml.device("default.tensor", wires=n_qubits)
    @jax.jit
    @qml.qnode(dev)
    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()
            fidelity = np.abs(state[0]) ** 2 # fidelity is the square of the amplitude of the |000> state
            sim_matrix[i, j] = fidelity
            sim_matrix[j, i] = fidelity
    return sim_matrix
```

As we've seen previously, the performances obtained from using Jax depend heavily the nature of the arguments - static, or dynamic. So as to obtain better performances, a different approach was thus considered: instead of defining the quantum circuit for two dynamic parameters (x_1, x_2) and then calling it for all pairs of arguments in the for loops, we recreate and recompile the circuit in each iteration of the i loop, but this time keeping x_1 fixed. This thus reduces the number of dynamic parameters to 1, at the cost of additional compilation time (since the circuit is recompiled in each iteration of the i loop). This resulted in the following circuit. Note that for ease of read, we have directly written the whole circuit in the main function without using two additional functions.

Circuit for a fixed x1 in PennyLane:

```
def x1_fixed_circuit(n_qubits, n_features, x1):
    dev = qml.device("default.tensor", method='mps', wires=n_qubits)
    @jax.jit
    @qml.qnode(dev, interface="jax-jit")
    def circuit(x2): #Note that in this case, the circuit only takes one dynamic parameter: x1 is fixed.
        for i in range(n_qubits):
            qml.H(wires=i)
            qml.RZ(x1[i % n_features], wires=i)
            qml.RY(x1[i % n_features], wires=i)
            qml.RY(-x2[i % n_features], wires=i)
            qml.RZ(-x2[i % n_features], wires=i)
            qml.H(wires=i)

        state = qml.state()
        return state # The calculation of fidelity is done outside the circuit
    return circuit
```

New Similarity Matrix in PennyLane:

```
def similarity_matrix_jax(X,n_features, n_qubits):
    n_samples = len(X)
    sim_matrix = np.eye(n_samples)
    X_jax = jnp.array(X)

    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])
            fidelity = np.abs(state[0]) ** 2 # fidelity is the square of the amplitude of the |000> state
            sim_matrix[i, j] = fidelity
            sim_matrix[j, i] = fidelity
    return sim_matrix
```

After running a quick benchmark, it appears that the gain of performances that stems from reducing the number of dynamic parameters far outweigh that of increasing the compilation time. Thus, for the next benchmarks, we will keep this version of the circuit.

In the attempt to obtain the most optimized circuit, we also decided to push further and add another level of parallelism. Indeed, when executing the previous circuit and monitoring resource usage, we noticed that not all cores were being used. This is a significant problem when comparing our method with Qiskit, as the latter utilizes all cores at nearly 100% of their capacity. To further increase the performances of the pennylane algorithm, we added another layer of parallelism: jax's pmap method. This method takes as input a circuit, and returns a parallelized version of this circuit. Calling this version with a batch of parameters will then compute the circuit results in parallel, ensuring a much more efficient use of computer resources. To implement this, we have re-written the similarity matrix function as follows:

Pmap Similarity Matrix in PennyLane:

```
def similarity_matrix_pmap(X,n_features, n_qubits):
    n_samples = len(X)
    sim_matrix = jnp.eye(n_samples)
    X_jax = jnp.array(X)


    for i in range(n_samples):
        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)
            for k in range(len(batch_parameters)):
                fidelity = np.abs(states[k][0]) ** 2
                sim_matrix[i, j*batch_size + k] = fidelity
                sim_matrix[j*batch_size + k, i] = fidelity
    return sim_matrix
```

Note that the parameter batch_size has been defined in the import section, as shown below. Note that pmap uses one CPU per instance of the circuit, so batch_size is equal to the number of CPUs allocated.
Setting batch_size :

```
import os

batch_size = os.cpu_count() - 2
os.environ["XLA_FLAGS"] = f"--xla_force_host_platform_device_count={batch_size}" # This allocates
    batch_size devices to jax.
```

This time, the gain in performances is astonishing (nearly a factor 5): this use of parallelism is very efficient. In order to understand how we could further improve performances, we looked closer to the time taken by this system to run. As an order of magnitude, running this circuit on the blob dataset for 50 datapoints and 3 clusters, using 15 CPU at 2.20 GHz, we saw that the total time to run all the circuits ('states = p_circuit(batch_parameters)' in the code above) was of approximately 5s, while the time taken to update the matrix was around 20s: the enhancements made were such that memory was now the limiting factor. After some quick research, we did not find any way to substantially improve this. We thus stopped looking into improving calculation efficiency, since memory was dominating the performances obtained.

This final function, using jax, jit and pmap was used to compare with Qiskit in the following benchmarks.

4.4 Results and performance

4.4.1 PennyLane vs Qiskit: High number of qubits and features

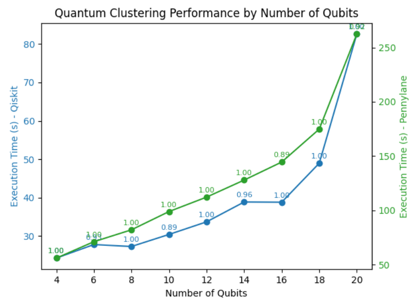


Figure 15: 4 Features

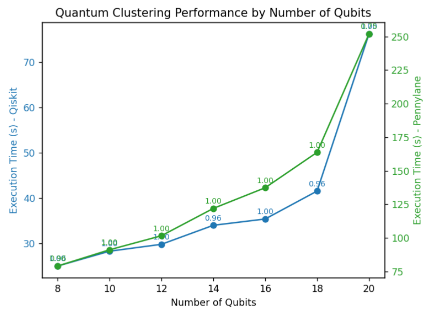


Figure 16: 8 Features

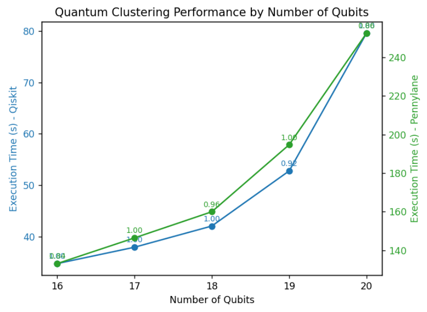


Figure 17: 16 Features

- **Performance(Execution Time):** Qiskit consistently demonstrates lower execution times compared to PennyLane, especially as the number of qubits increases beyond 16. Increasing the feature dimensionality from 4 to 8, and subsequently to 16, substantially increases computational demand for both frameworks.

- **Clustering Quality (NMI Score):** Both frameworks consistently yield high NMI scores (typically around 0.90 to 1.00), indicating stable and accurate clustering performance. Variations in NMI scores across different qubit numbers do not significantly impact the overall high-quality clustering results.

4.4.2 Trying on real datasets

In order to conclude with this project, we looked into some real data using a dataset for breast cancer provided by scikit learn. The dataset has 2 clusters, 100 datapoints and 30 features. As observed before, Qiskit manages to outperform our algorithm. It has to be noted that Qiskit also switches to a tensor-based approach (MPS), and utilizes parallelism as well. Observations yielded that Qiskit also has a much better use of the CPUs than the algorithm crafted from pennylane, jax and jit. Thus, for the following benchmaks, we only recorded Qiskit's performances.

As we can see in figure 19, for an appropriate choice for the number of qubits relative to the number of features, increasing the number of qubits leads to an overall increase in the score. This demonstration concludes on the key question of this project: using tensor enables the simulation in a reasonable time of complex QML circuits, which is particularly important in the NISQ era, especially since using a high number of qubits is important to reach desired performances.

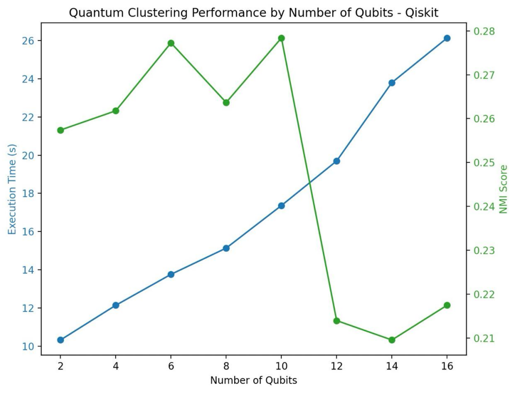


Figure 18: Number of features equal to number of Qubits

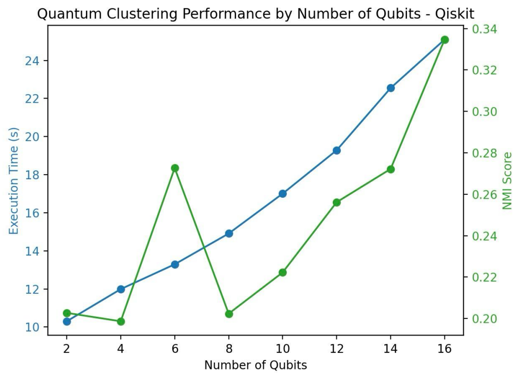


Figure 19: Twice more Qubits than features

5 Bibliographie

References

[1] Jacob Biamonte, Ville Bergholm, *Quantum Tensor Networks in a Nutshell*

[2] Alfred M. Pastor et al., *A community detection-based parallel algorithm for quantum circuit simulation using tensor networks*, January 2, 2025.

[3] Michele Cattelan et al., *Parallel circuit implementation of variational quantum algorithms*

[4] Kuan-Cheng Chen et al., *Validation large-scale quantum machine learning: efficient simulation of quantum support vector machines using tensor networks*, 2025

[5] W. Huggins, P. Patil, B. Mitchell, K. B. Whaley, and E. M. Stoudenmire *Towards quantum machine learning with tensor networks, ISSN 2058-9565, URL http://dx.doi.org/10.1088/2058-9565/aaea94*, 2019

[6] R. Orús *A practical introduction to tensor networks: Matrix product states and projected entangled pair states, ISSN 0003- 4916, URL https://www.sciencedirect.com/science/article/pii/S0003491614001596.*, 2014

All the articles have been uploaded on the teams folder.