

# Introducing Qibo

An open-source full stack API for quantum simulation and hardware control

---

Andrea Pasquale and Stavros Efthymiou on the behalf of the Qibo Team

2nd December 2022



UNIVERSITÀ  
DEGLI STUDI  
DI MILANO



## Outline of this session

- 11:30 - 11:50 Introduction to Qibo
- 11:50 - 12:00 Brief overview of Grover's algorithm
- 12:00 - 12:30 Tutorial on how to use Qibo
- 12:30 - 13:00 Your turn to solve the exercise
- 13:00 - 13:15 Explanation of the solution
- 13:15 - 13:30 Implementation of the solution on GPU

# Introduction to Qibo

---



Institute	TII	CQT	INFN	Qilimajiaro
Quantum Hardware	5 qubits	10 qubits	1 qubit	2 qubits

# Quantum Computing and HPC

---













In order to simulate a quantum circuit with  $n$  qubits we need to be able to manipulate a  $2^n$  components vector.

In Schrödinger's approach each gate is applied to the state via the following matrix multiplication

$$\psi'(\sigma_1, \dots, \sigma_n) = \sum_{\tau'} G(\tau, \tau') \psi(\sigma_1, \dots, \tau', \dots, \sigma_n) \quad (1)$$

where the gate targeting  $n_{\text{tar}}$  qubits is represented by the  $2^{n_{\text{tar}}} \times 2^{n_{\text{tar}}}$  complex matrix  $G(\tau, \tau') = G(\tau_1, \dots, \tau_{n_{\text{tar}}}, \tau'_1, \dots, \tau'_{n_{\text{tar}}})$  and  $\sigma_i, \tau_i \in \{0, 1\}$ .

# A few gates

Operator	Gate(s)	Matrix
Pauli-X (X)	 	$\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$
Pauli-Y (Y)		$\begin{bmatrix} 0 & -i \\ i & 0 \end{bmatrix}$
Pauli-Z (Z)		$\begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}$
Hadamard (H)		$\frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$
Phase (S, P)		$\begin{bmatrix} 1 & 0 \\ 0 & i \end{bmatrix}$
$\pi/8$ (T)		$\begin{bmatrix} 1 & 0 \\ 0 & e^{i\pi/4} \end{bmatrix}$
Controlled Not (CNOT, CX)		$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}$
Controlled Z (CZ)		$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & -1 \end{bmatrix}$
SWAP	 	$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$
Toffoli (CCNOT, CCX, TOFF)		$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$

- 1 qubit gates 2x2 matrices
- 2 qubit gates 4x4 matrices
- 3 qubit gates 8x8 matrices

# Why is it difficult to implement a good quantum simulator?

From the previous slides we can understand that

good **quantum circuits** simulator  $\Leftrightarrow$  good engine to perform **matrix multiplication**

Moreover, when it comes to full-state vector simulation we need to store the full state vector in the RAM.

How much memory is required if we increase the size of the state vector?

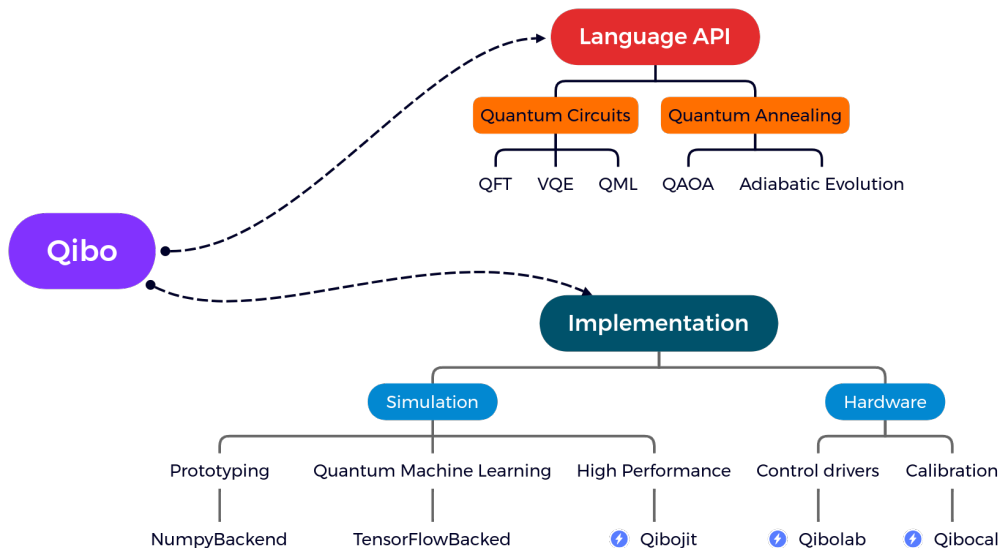
<b>Qubits</b>	single precision	double precision
10	8 kB	16 kB
20	8 MB	16 MB
30	8.59 GB	17.18 GB
40	8.79 TB	17.59 TB



## Introducing Qibo

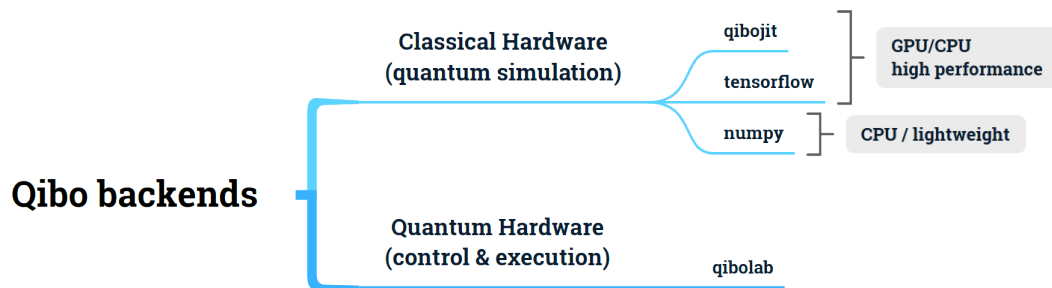
---

Qibo is an **open-source** full stack API for **quantum simulation** and quantum hardware control and calibration.



# Simulation backends in Qibo

Qibo provides multiple backends for simulating quantum circuits:



Qibo has a modular layout that enables to switch between different backends easily.

Let's take a closer look at each backend...

Simulator based on numpy:

- `np.ndarray`
- numpy primitives



## FEATURES

- Cross-architecture (x86, arm64, etc)
- Cross-platform
- Fast for small circuits
- Fast for single-threaded operations

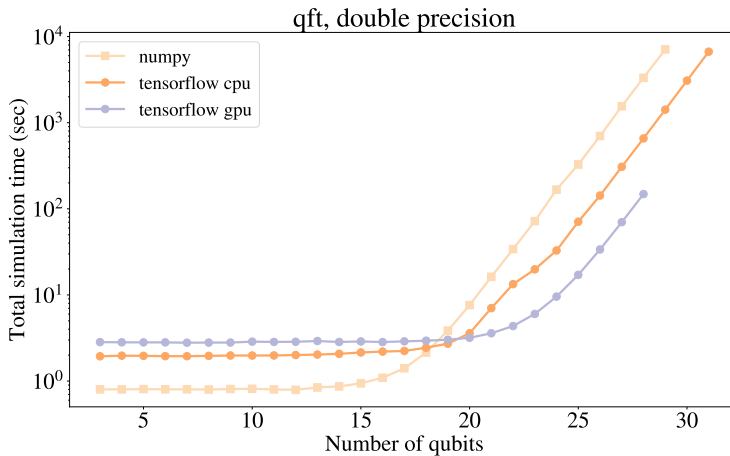
Simulator based on tensorflow primitives:

- `tf.Tensor`
- `tf.matmul` and `tf.matmul`



## FEATURES

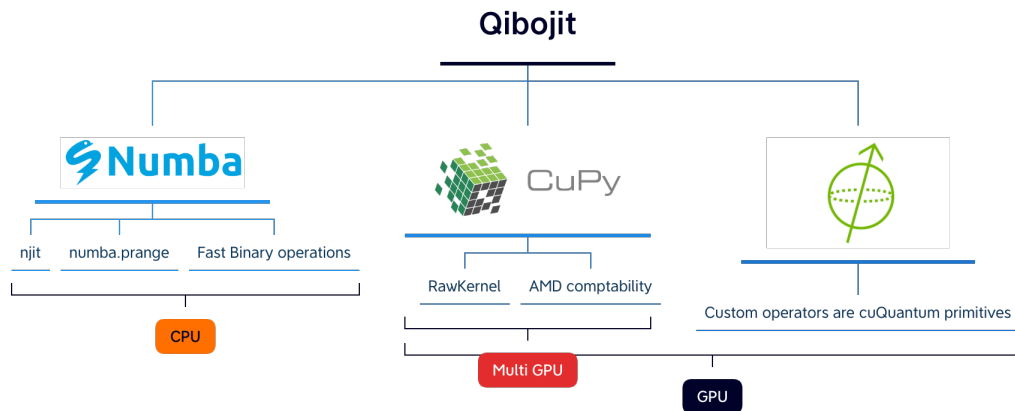
- Multithreading CPU
- Single GPU
- Gradient descent on quantum circuits
- QML using Qibo



- Exponential scaling
- Tensorflow better than numpy
- GPU architecture helps with large number of qubits

# Can we do better than this?

To efficiently simulate circuits with large number of qubits we designed a new backend in Qibo based on JIT compilation: **qibojit**.



## FEATURES

- *in-place* updates
- exploit *sparsity of matrix*
- Just-in-Time compilation
- CuQuantum compatibility

# What is Just-in-Time (JIT) compilation?

JIT: a method for improving the performance of interpreted programs.

When compiling/interpreting a language we have different options:

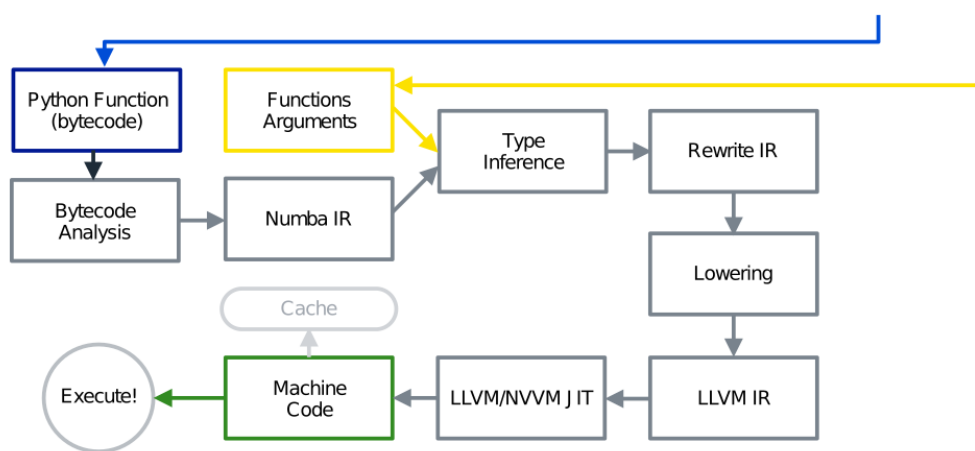
- **Static compiler:** reads a programme, looks at the code and tries to convert it into machine code. Examples: C, C++.
- **Interpreter:** looks at the programme, does not convert to machine code and it executes it almost as it is. Examples: Python, JavaScript
- **Just-in-Time compiler:** starts a programme running an interpreter and dynamically produce machine code based on the observation of the programme. Examples: Python, JavaScript

JIT compilers can be **faster** than static compilers because they can get more information by running the programme instead of just looking at the programme at compile time!



## Example of JIT

```
@cuda.jit  
def axpy(r, a, x, y)  
...  
>>> axpy(r, a, x, y)
```



---

```
from numba import njit, prange

@njit(parallel=True, cache=True)
def apply_gate_kernel(state, gate, target):
    """Operator that applies an arbitrary one-qubit gate.

    Args:
        state (np.ndarray): State vector of size (2 **
        ↪ nqubits,).
        gate (np.ndarray): Gate matrix of size (2, 2).
        target (int): Index of the target qubit.
    """
    k = 1 << target
    # for one target qubit: loop over half states
    nstates = len(states) // 2
    for g in prange(nstates):
        # generate index with fast binary operations
        i1 = ((g >> m) << (m + 1)) + (g & (k - 1))
        i2 = i1 + k
        state[i1], state[i2] = (gate[0, 0] * state[i1] + \
                                gate[0, 1] * state[i2],
                                gate[1, 0] * state[i1] + \
                                gate[1, 1] * state[i2])

    return state
```

---

## Observations

- @njit
- prange
- fast binary operations

Same approach followed using Numba: JIT compilation.

Custom operators implemented using `cupy.RawKernel`:

## 1. Write custom CUDA kernels written in C++

---

```
apply_gate_kernel = (
    """
    #include <cupy/complex.cuh>
    {_apply_gate}"""
    + """
    // C++ implementation of gates.py:apply_gate_kernel()
    extern "C"
    __global__ void apply_gate_kernel(T* state, long tk, int m, const T* gate) {
        const long g = blockIdx.x * blockDim.x + threadIdx.x;
        const long i = ((long)((long)g >> m) << (m + 1)) + (g & (tk - 1));
        _apply_gate(state[i], state[i + tk], gate);
    }
    """
)
```

---

2. At the first invocation the kernel will be compiled using `nvcc`

3. After the first invocation it is cached for each device

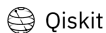
cuQuantum from NVIDIA has a Python API which delivers all the functionalities cuStateVec and cuTensorNet with Python API.

Thanks to the modular layout of Qibo it was possible to create a new backend where the custom operators are the primitives from cuStateVec!

## Framework Integrations

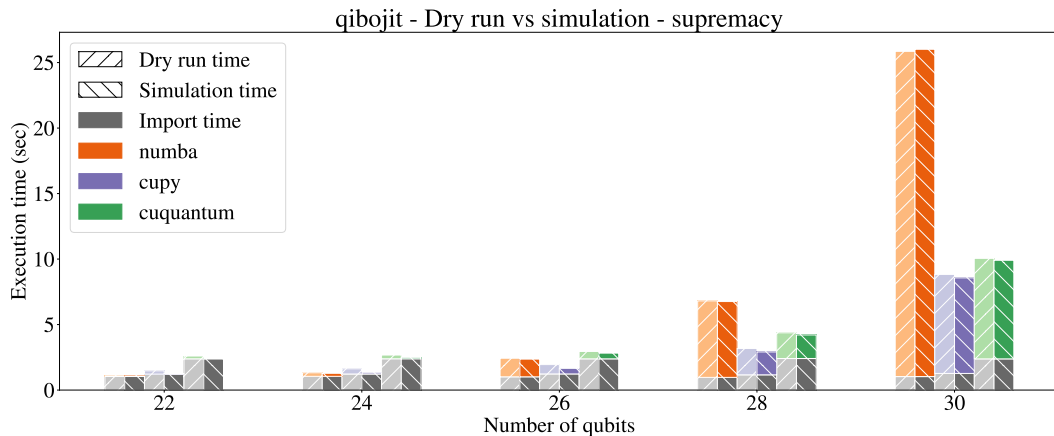
cuQuantum is integrated with leading quantum circuit simulation frameworks.

Download cuQuantum to dramatically accelerate performance using your framework of choice, with zero code changes.

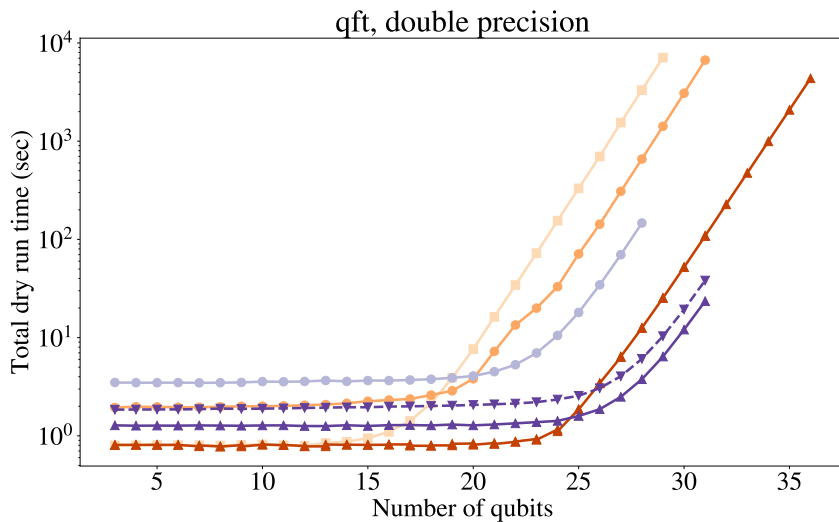


# Benchmarks

When benchmarking libraries which involve justin-time compilation it is important to distinguish the first execution, *dry run* because it will involve a compilation or loading of cached binaries and therefore will be slower than subsequent executions, *simulation* in the same run time.

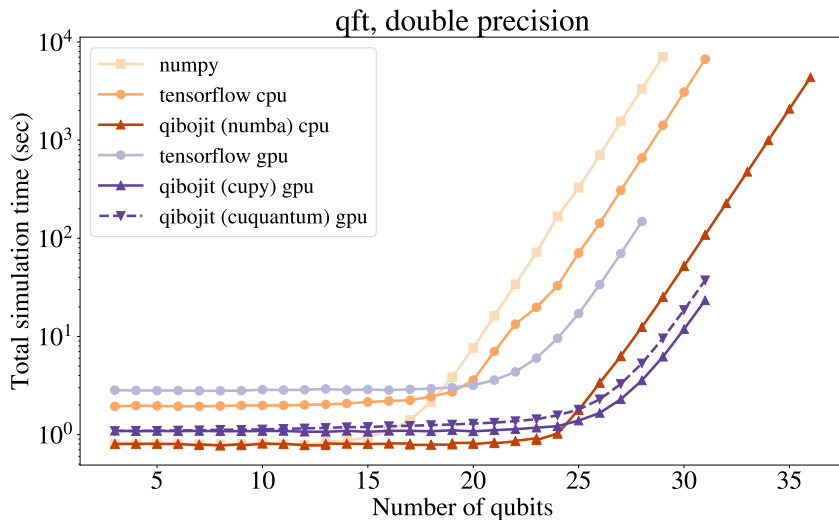


# Benchmarks



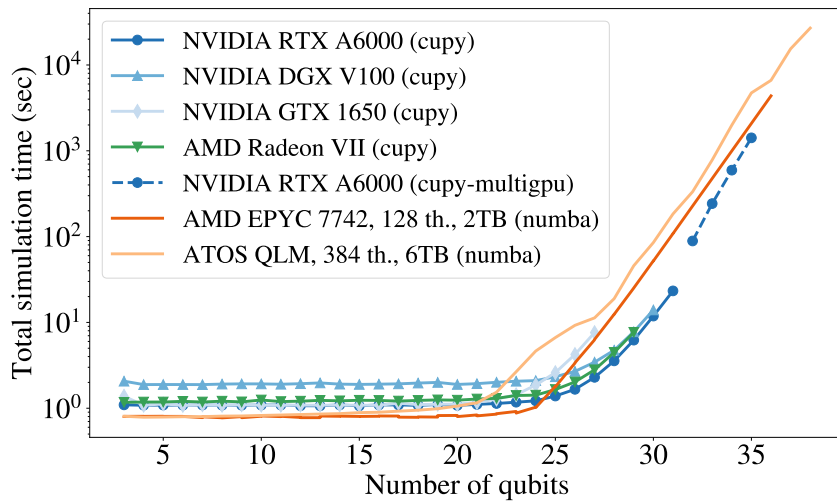
Benchmark library: <https://github.com/qiboteam/qibojit-benchmarks>

# Benchmarks



Benchmark library: <https://github.com/qiboteam/qibojit-benchmarks>

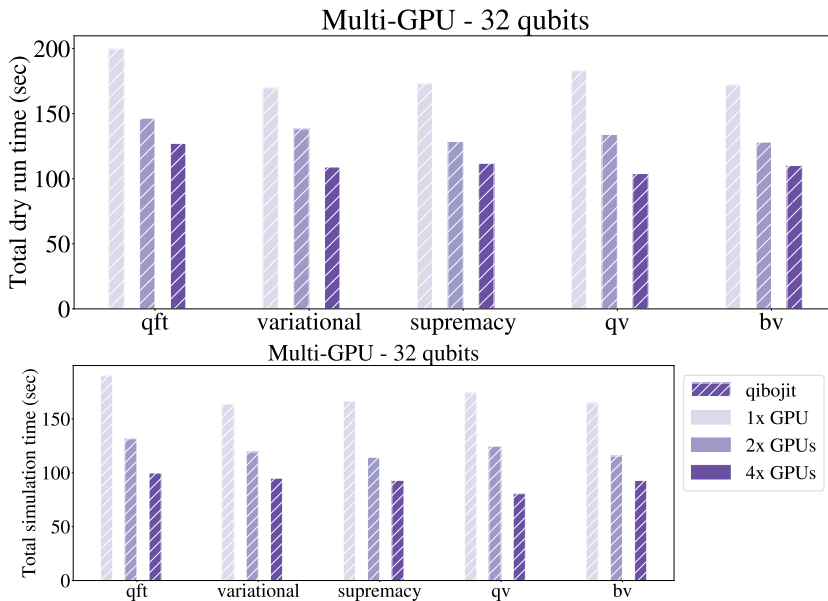
## Benchmark on different devices



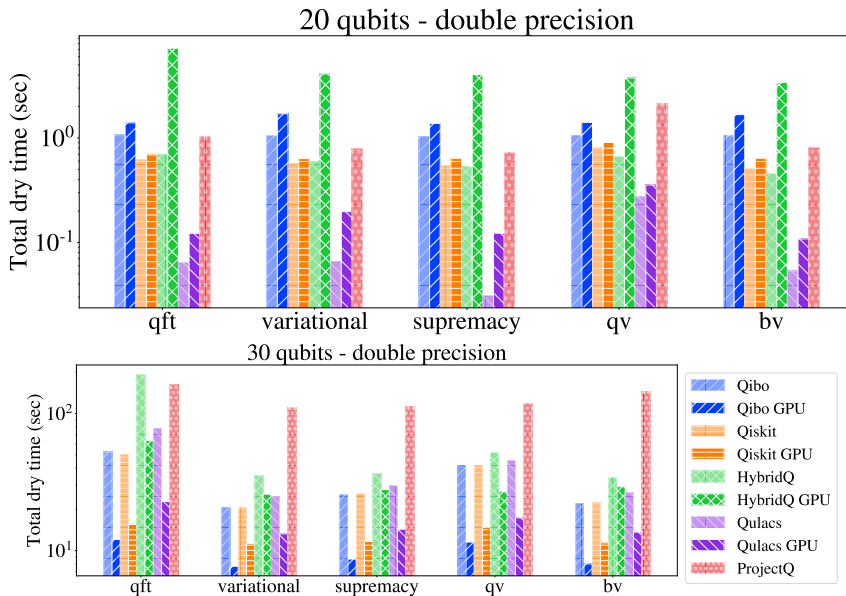


# Multi-GPU support

CupyBackend supports also multi-GPU architectures



# How does Qibo perform against the other libraries?

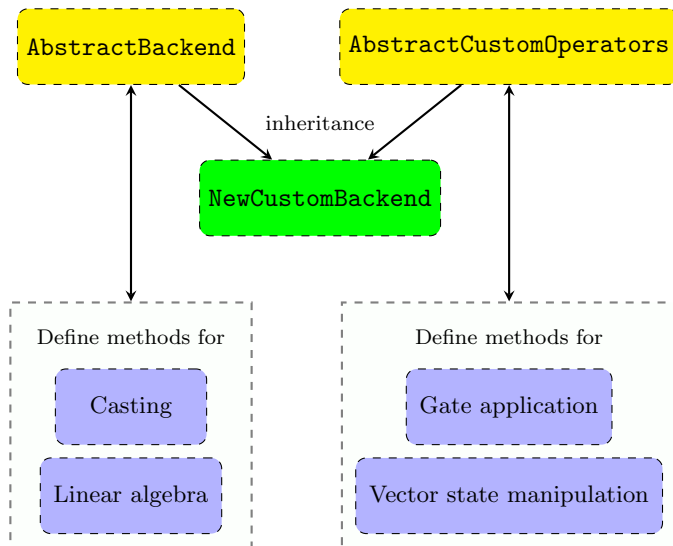


Benchmark library: <https://github.com/qiboteam/qibojit-benchmarks>

## How to add a new backend in Qibo?

Do you think that you can develop a backend with better performances?

Thanks to its modular framework in Qibo it is easy to implement a new backend.



## Outlook

---

We have presented an open full stack API for quantum simulation: Qibo.

✓ High-performance quantum simulation: **qibojit**

What makes Qibo different from other libraries:



- + Public available as an open source project.
- + Modular layout design with possibility of adding
  - a new backend for simulation
- + Community driven effort

Qibo is **more** than a quantum simulator. We are currently developing modules for

- hardware control → `qibolab`
- hardware calibration → `qibocal`

`https://github.com/qiboteam/qibo`

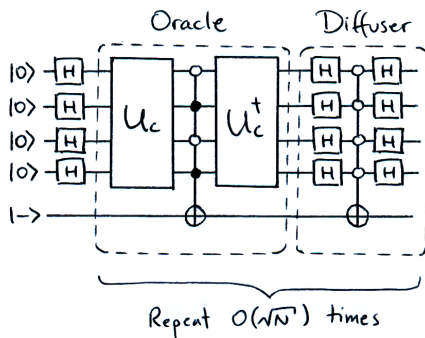
`https://qibo.readthedocs.io/en/stable/`

## Grover's algorithm

---

# What is Grover's Algorithm?

Grover's algorithm is a quantum search algorithm that can search for a value or element in an unsorted set in  $\mathcal{O}(\sqrt{N})$  as opposed to classical search algorithms that at worst will find an element in  $\mathcal{O}(N)$  time.



Quantum advantage originates from:

- **Superposition:** Perform an operation to all possible solutions at the same time.
- **Interference:** Change sign of the amplitude of the correct solution
- **Entanglement:** Non-trivial sharing of information between states

## Important operations

**Welsh-Hadamard transform:** Apply Hadamard gate to every qubit:

$$\mathbf{H} \rightarrow \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}$$

All possible binary strings with equal amplitude

$$(-1)^{\bar{x} \cdot \bar{y}} 2^{-\frac{n}{2}}$$

Plus or minus sign depending on the number of ones in the initial and final state

**Selective phase rotation:** Apply a phase to just some specific states.

$$\begin{pmatrix} e^{i\phi_{00}} & 0 & 0 & 0 \\ 0 & e^{i\phi_{01}} & 0 & 0 \\ 0 & 0 & e^{i\phi_{10}} & 0 \\ 0 & 0 & 0 & e^{i\phi_{11}} \end{pmatrix}$$

Grover's algorithm uses this matrix with  $\phi_i = \pi$  if the state  $i$  fulfills a condition, and  $\phi_j = 0$  otherwise.



Operator that changes the sign of the amplitudes of the quantum states that encode solutions of the problem.

Common way to change the sign once the solution is detected: **use an ancillary qubit.**

Ancilla initialized with an  $X$  gate followed by a Hadamard gate:

$$|\psi_a\rangle = HX|0\rangle = H|1\rangle = \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle)$$

When the  $X$  gate is applied:

$$X|\psi_a\rangle = \frac{1}{\sqrt{2}}(|1\rangle - |0\rangle) = -|\psi_a\rangle$$

Hint: use CNOT gates to change the sign of the solution.

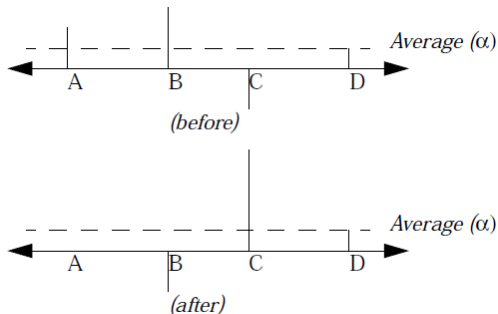
## Diffusion transform

The diffusion transform matrix is a matrix defined as follows:

$$D_{ij} = \begin{cases} \frac{2}{N}, & \text{if } i \neq j \\ -1 + \frac{2}{N}, & \text{if } i = j \end{cases} \quad (2)$$

This can be achieved by applying a Walsh-Hadamard transform on all qubits. Then changing the sign of the  $|000 \dots 000\rangle$  state, and applying once again a Hadamard gate on every qubit.

The diffuser implements an inversion about the average.



## Reason for scaling

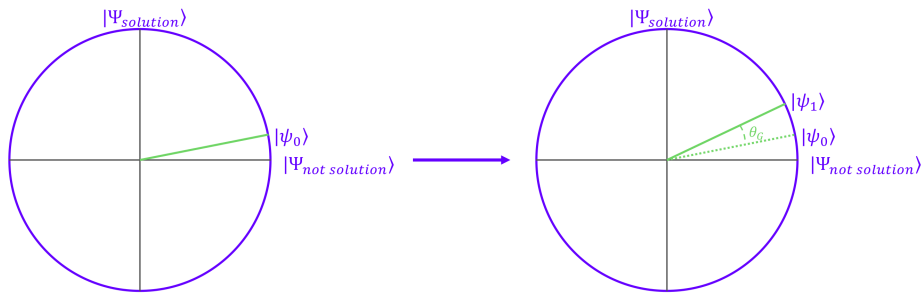
The quantum state can be understood as a superposition of:

$$|\psi_i\rangle = k_i |\Psi_{\text{solution}}\rangle + l_i |\Psi_{\text{not solution}}\rangle$$

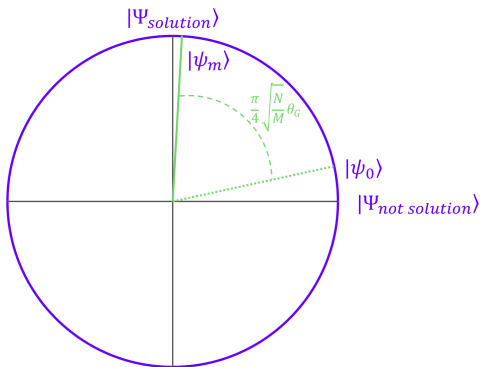
At the start of the algorithm, we can consider  $k_0 = \sin \theta$  and  $l_0 = \cos \theta$  with  $\sin^2 \theta = 1/N$ .

After the  $i$ -th Grover step:

$$k_j = \sin(2j + 1)\theta \quad \text{and} \quad l_j = \frac{1}{\sqrt{N-1}} \cos(2j + 1)\theta$$



## Reason for scaling



In order to achieve  $k_m = 1$  it follows that  $(2m + 1)\theta = \pi/2$

For large number of  $N$ :

$$\theta \approx \sin \theta = 1/\sqrt{N}$$

The number of iterations needed is the closest integer to

$$\frac{\pi}{4}\sqrt{N}$$

in case of a single solution.

This can be extended to  $\frac{\pi}{4}\sqrt{\frac{N}{M}}$  when considering multiple solutions.

**Thanks for listening!**

---