

Introduction to ProjectQ

Damien Nguyen

Huawei 2012 Laboratories

June 21, 2022

damien1@huawei.com



Outline

- 1 The ProjectQ framework
- 2 ProjectQ in details
 - ProjectQ by example
 - Beyond simple gates
 - Backends
 - Summary
- 3 Installation
 - General
 - Step-by-step by operating system
 - URLs
- 4 Time for a break!

The ProjectQ framework

What is ProjectQ?

ProjectQ is an open-source software framework for quantum computing.

It aims at providing tools which facilitate inventing, implementing, testing, debugging, and running quantum algorithms using either classical hardware or actual quantum devices.

Important URLs:

Main website	http://projectq.ch
GitHub	https://github.com/ProjectQ-Framework/ProjectQ
Documentation	https://projectq.readthedocs.io/en/latest/index.html

ProjectQ: Design goals

ProjectQ was designed with four main goals

1. Open & Free: ProjectQ is released under the Apache 2 license
2. Simple learning curve: It is implemented in Python and has an intuitive syntax
3. Easily extensible: Anyone can contribute to the compiler, the embedded domain-specific language, and libraries
4. Code quality: Code reviews, continuous integration testing (unit and functional tests)

Citing ProjectQ

Think about citing the following papers

D. S. Steiger, T. Häner, and M. Troyer. "ProjectQ: An Open Source Software Framework for Quantum Computing". In: (Dec. 2016). DOI: 10.22331/q-2018-01-31-49. arXiv: 1612.08091. URL: <https://arxiv.org/abs/1612.08091>

T. Häner et al. "A Software Methodology for Compiling Quantum Programs". In: (Apr. 2016). DOI: 10.1088/2058-9565/aaa5cc. arXiv: 1604.01401. URL: <http://arxiv.org/abs/1604.01401>

ProjectQ in details

ProjectQ's syntax

Physics

$$A |000\rangle \equiv A (|0\rangle \otimes |0\rangle \otimes |0\rangle)$$

ProjectQ

A | qubits

eDSL in Python

```

1  from projectq import MainEngine
2  from projectq.ops import H, Measure
3
4  eng = MainEngine()
5  qubit = eng.allocate_qubit()
6
7  H | qubit
8  Measure | qubit
9
10 eng.flush()
11 print("Measured {}".format(int(qubit)))

```

```

> python3 test.py
Measured 1

```

Imports

There are four main sub-modules to ProjectQ

```

1  import projectq.backends # contains all known back-ends
2  import projectq.cengines # contains compiler engines
3  import projectq.ops      # contains most quantum gates and operations
4  import projectq.meta     # contains 'meta' operations

```

But there are actually a few more:

```

1  import projectq.setups
2  import projectq.libs
3  import projectq.types

```

Allocating qubits

The MainEngine contains two methods to create qubits:

- `eng.allocate_qubit()` for a single qubit
- `eng.allocate_ureg(N)` for a list of N qubits (also known as a quantum register)

For ProjectQ, quantum registers are nothing more than lists of qubits

```

1  qureg = eng.allocate_ureg(10)
2  qubit_8 = qureg[7]

```

Per convention, qubit allocated using ProjectQ start in the $|0\rangle$ state.

Allocating qubits (contd.)

Once a qubit has been measured, you can access its value by converting it to an `int` or `bool`:

```

1  measured = int(qubit)

```

If a qubit was not already measured when you try to convert it to an `bool` or `int`, an exception will be thrown:

```

> python3 test.py
RuntimeError: Error: Qubit has not been measured / uncomputed! \
  There is most likely a bug in your code.
raised in:
' File ".../backends/_sim/_simulator.py", line 385, in _handle'
'   self._simulator.deallocate_qubit(ID)'

```

Deallocating qubits

Qubit are *automatically* deallocated once they go out of scope.

You can however deallocate a qubit manually by using either syntax:

```
1 del qubit
2 qubit.__del__()
```

For quantum registers:

```
1 del qureg # deletes all qubits in register
```

Important

Qubit may only be de-allocated when in a classical state

Single-qubit gates

Basic single-qubit quantum gates can be used as-is.

$$\begin{array}{ll}
 \text{X, NOT} & \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \\
 \text{Y} & \begin{bmatrix} 0 & -i \\ i & 0 \end{bmatrix} \\
 \text{Z} & \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix} \\
 \text{S} & \begin{bmatrix} 1 & 0 \\ 0 & i \end{bmatrix} \\
 \text{T} & \begin{bmatrix} 1 & 0 \\ 0 & e^{i\pi/4} \end{bmatrix} \\
 \text{Rx} & \begin{bmatrix} \cos(\alpha/2) & -i \sin(\alpha/2) \\ -i \sin(\alpha/2) & \cos(\alpha/2) \end{bmatrix} \\
 \text{Ry} & \begin{bmatrix} \cos(\alpha/2) & -\sin(\alpha/2) \\ \sin(\alpha/2) & \cos(\alpha/2) \end{bmatrix} \\
 \text{Rz} & \begin{bmatrix} e^{-i\alpha/2} & 0 \\ 0 & e^{i\alpha/2} \end{bmatrix} \\
 \text{R} & \begin{bmatrix} 1 & 0 \\ 0 & e^{i\alpha} \end{bmatrix} \\
 \text{Ph} & \begin{bmatrix} e^{i\alpha} & 0 \\ 0 & e^{i\alpha} \end{bmatrix}
 \end{array}$$

```
1 from projectq.ops import (X, NOT, Y, Z, S, T, H,
2                           Rx, Ry, Rz, Ph, R)
3 X | qubit
4 Rx(1.0) | qubit
```

Other standard gates

```
1 from projectq.ops import CNOT, CZ, Toffoli, Swap, Barrier, Measure
2
3 CNOT | (ctrl, tgt)
4 CZ | (ctrl, tgt)
5 Toffoli | (ctrl1, ctrl2, tgt) # (CCNOT gate)
6
7 Swap | (qubit0, qubit1)
8
9 Barrier | qubit
10 Measure | qubit
```

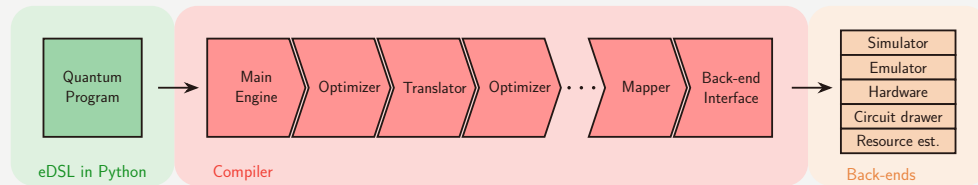
eDSL in Python (again)

```
1 from projectq import MainEngine
2 from projectq.ops import H, Measure
3
4 eng = MainEngine() # create a default compiler
5 qubit = eng.allocate_qubit() # allocate 1 qubit
6
7 H | qubit # apply a Hadamard gate
8 Measure | qubit # measure the qubit
9
10 eng.flush() # flush all gates (and execute measurements)
11 print("Measured {}".format(int(qubit))) # output measurement result
```

Flushing command list

ProjectQ does *not* execute instructions immediately.

Teaser from tomorrow:



Main message

Some of the compiler engines may cache instructions. Therefore, use:

```
eng.flush()
```

Beyond simple gates

ProjectQ supports some *meta*-operations:

Tensor gates (going from single to N-qubit gates)

Control

Dagger

Loop

Compute/Uncompute

Control and tensor gates

```

1 from projectq.ops import (X, Measure,
2                           All, Tensor, C)
3
4 All(Measure) | qureg      # Measure all qubits in 'qureg'
5 Tensor(Measure) | qureg   # Same as above
6
7 C(X) | (ctrl, tgt)        # equivalent to CNOT(ctrl, tgt)
8 C(X, 2) | (ctrl1, ctrl2, tgt) # equivalent to Toffoli(c1, c2, tgt)
  
```

2-qubit Bell state

```

1 from projectq import MainEngine
2 from projectq.ops import All, Measure, CNOT, C, X, H
3
4 eng = MainEngine()
5 qureg = eng.allocate_quireg(2)
6
7 H | qureg[0]
8 CNOT | (qureg[0], qureg[1]) # same as C(X) | (qureg[0], qureg[1])
9 All(Measure) | qureg
10
11 eng.flush()
12 print("Measured {}".format([int(qubit) for qubit in qureg]))
13 # Should print either '[0, 0]' or '[1, 1]' with 0.5 probability

```

General Bell state

```

1 from projectq import MainEngine
2 from projectq.ops import All, Measure, CNOT, H
3
4 N = 10
5 eng = MainEngine()
6 qureg = eng.allocate_quireg(N)
7
8 H | qureg[0]
9 for qb in qureg[1:]:
10     CNOT | (qureg[0], qb)
11 All(Measure) | qureg
12
13 eng.flush()
14 print("Measured {}".format([int(qubit) for qubit in qureg]))

```

Control with $N > 1$

Imagine you are in the following situation:

```

1 from projectq import MainEngine
2 from projectq.ops import H, X, All, C
3
4 eng = MainEngine()
5 ctrl = eng.allocate_quireg(5)
6 qureg = eng.allocate_quireg(4)

```

How to perform some gates controlled on all qubits in ctrl ?

Control with $N > 1$ (contd.)

Bad solution:

```

1 from projectq import MainEngine
2 from projectq.ops import H, X, All, C
3
4 eng = MainEngine()
5 ctrl = eng.allocate_quireg(5)
6 qureg = eng.allocate_quireg(4)
7
8 # Idea 1
9 C(H, 5) | (ctrl[0], ctrl[1], ctrl[2], ctrl[3], ctrl[4], qureg[0])
10 # Idea 2
11 C(H, 1) | (*ctrl, qureg[0])

```

Control with $N > 1$ (contdd.)

Better solution:

```

1 from projectq import MainEngine
2 from projectq.meta import Control
3 from projectq.ops import H, X, All, C
4
5 eng = MainEngine()
6 ctrl = eng.allocate_quireg(5)
7 qureg = eng.allocate_quireg(4)
8
9 with Control(eng, ctrl):
10     H | qureg[0]
11     X | qureg[1]

```

General Bell state (revisited)

```

1 from projectq import MainEngine
2 from projectq.meta import Control
3 from projectq.ops import All, Measure, X, H
4
5 N = 10
6 eng = MainEngine()
7 qureg = eng.allocate_quireg(N)
8
9 H | qureg[0]
10 with Control(eng, qureg[0]):
11     All(X) | qureg[1:]
12 All(Measure) | qureg
13 eng.flush()
14 print("Measured {}".format([int(qubit) for qubit in qureg]))

```

Inverting gates

Bad solution:

```

1 from projectq import MainEngine
2 from projectq.ops import Rx, get_inverse
3
4 eng = MainEngine()
5 qureg = eng.allocate_quireg(2)
6
7 # ...
8 get_inverse(Rx(1.0)) | qureg[0]
9 get_inverse(Rx(2.0)) | qureg[1]
10

```

Not recommended!

Dagger (inverting gates)

Better solution:

```

1 from projectq import MainEngine
2 from projectq.meta import Dagger
3 from projectq.ops import Rx
4
5 eng = MainEngine()
6 qureg = eng.allocate_quireg(2)
7
8 with Dagger(eng):
9     Rx(1.0) | qureg[0]
10    Rx(1.0) | qureg[1]

```

Implementing loops

```

1 from projectq import MainEngine
2 from projectq.meta import Loop
3
4 N = 5; N_iter = 10
5 eng = MainEngine()
6 qureg = eng.allocate_qureg(N)
7
8 # Normal Python loop
9 for _ in range(N_iter):
10     # do something
11
12 # ProjectQ loop
13 with Loop(eng, N_iter):
14     # do something

```

Compute/Uncompute pattern

```

1 from projectq import MainEngine
2 from projectq.ops import H, X
3 from projectq.meta import Compute, Uncompute
4
5 N = 5
6 eng = MainEngine()
7 qureg = eng.allocate_qureg(N)
8
9 # ProjectQ loop
10 with Compute(eng):
11     All(H) | qureg
12 All(X) | qureg[0::2]
13 Uncompute(eng)

```

Compute/Uncompute pattern (contd.)

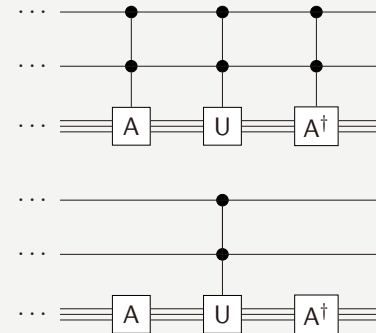
```

1 # ...
2
3 # ProjectQ loop
4 with Compute(eng):
5     # some computations here
6
7 # more computations here
8
9 with CustomUncompute(eng):
10     # do 'better' Uncompute here

```

Benefits of code annotations

More context for the compiler → better chances for optimizations!



```

C(A) | (ctrl, qubit)
C(U) | (ctrl, qubit)
C(iA) | (ctrl, qubit)

```

```

with Control(eng, ctrl):
    with Compute(eng):
        A | qubit
    U | qubit
Uncompute(eng)

```

Quick words about Backends

Default backend

By default, the MainEngine uses the simulator/emulator backend:

```
1 from projectq import MainEngine
2 from projectq.backends import Simulator
3
4 eng = MainEngine(backend=Simulator())
```

Available backends:

Simulator
 ResourceCounter
 CommandPrinter
 CircuitDrawer
 IBMBackend

ResourceCounter

```
1 from projectq import MainEngine
2 from projectq.backends \
3     import ResourceCounter
4
5 backend = ResourceCounter()
6 eng = MainEngine(backend=backend)
7 # ... do something
8 print(backend)
```

```
> python3 test.py
Gate class counts:
  AllocateQubitGate : 1
  HGate : 1
  MeasureGate : 1

Gate counts:
  Allocate : 1
  H : 1
  Measure : 1

Max. width (number of qubits) : 1.
```

CommandPrinter

```
1 from projectq import MainEngine
2 from projectq.backends import CommandPrinter
3
4 eng = MainEngine(backend=CommandPrinter(accept_input=True,
5                                         default_measure=False,
6                                         in_place=False))
7 # ... do something
```

```
> python3 test.py
Allocate | Qureg[0]
H | Qureg[0]
Measure | Qureg[0]
Input measurement result (0 or 1) for qubit 0: 0
```

CircuitDrawer

```

1 from projectq import MainEngine
2 from projectq.backends import CircuitDrawer
3
4 backend = CircuitDrawer()
5 eng = MainEngine(
6     backend=CircuitDrawer(
7         accept_input=False,
8         default_measure=0)
9
10 # ... do something
11
12 with open('/tmp/circuit.tex', 'w') as fd:
13     fd.write(backend.get_latex())

```



IBMBBackend

Submit circuits directly to the IBM Quantum Experience

```

1 from projectq.backends import IBMBBackend
2
3 backend = IBMBBackend(use_hardware=False,
4                       num_runs=1024,
5                       verbose=False,
6                       user=None,
7                       password=None,
8                       device='ibmqx4',
9                       num_retries=3000,
10                      interval=1,
11                      retrieve_execution=None)

```

Summary

What we've seen so far

Qubit allocation/deallocation (`eng.allocate_XXX(...)`)

Single-qubit gates (`X | qureg`, `H`, `Rx(1.0)`, ...)

Measurements (`Measure | qubit`)

Tensor gate application (`All(Measure) | qureg`)

Meta-operations

Control (`with Control(eng, ctrl_qubits)`)

Dagger (`with Dagger(eng, ctrl_qubits)`)

Loop (`with Loop(eng, N_iter)`)

Compute/Uncompute (`with Compute(eng)`)

Backend choices (`Simulator()`, `ResourceCounter()`, `CommandPrinter()`, ...)

Installation

Quick install for users

Simplest method:

```
python3 -m pip install --user projectq
```

Or if you're using a virtualenv:

```
. path/to/virtualenv/bin/activate  
python3 -m pip install projectq
```

Might want to change the C++ compiler:

```
CXX=g++-9 python3 -m pip install -e .
```

Quick install for developers

Checkout the repository

```
git clone https://github.com/ProjectQ-Framework/ProjectQ.git
```

Install *in-place*:

```
python3 -m pip install -e .
```

Ubuntu/Debian

1. Install the required build tools:

```
sudo apt-get install build-essential
```

2. Install Python 3 and the package manager pip

```
sudo apt-get install python3 python3-pip
```

3. Install ProjectQ

```
python3 -m pip install --user projectq
```

Windows

Download and install the latest Python distribution:

64bits: <https://www.python.org/ftp/python/3.10.5/python-3.10.5-amd64.exe>

32bits: <https://www.python.org/ftp/python/3.10.5/python-3.10.5.exe>

Install the compiler and associated libraries. Either of

With Visual Studio <https://visualstudio.microsoft.com/>

Without Visual Studio

Visual C++ Build Tools <http://chürzer.ch/visual-cpp-build-tools>

Microsoft Windows SDK <http://chürzer.ch/windows-10-sdk>

Open a cmd.exe window:

```
> python3 -m pip install --user projectq
```

Mac OSX

Download and install XCode

<https://itunes.apple.com/ch/app/xcode/id497799835>

In a Terminal windows enter:

```
> xcode-select --install
```

Try with the Apple version of clang:

```
> python3 -m pip --user install projectq
```

Mac OSX (contd.)

If it fails, install Homebrew

```
> /bin/bash -c \
"$(&curl -fsSL https://raw.githubusercontent.com/\
Homebrew/install/HEAD/install.sh)"
```

Install the required programs and libraries

```
> brew install python3 llvm
```

Compile and install ProjectQ

```
> CXX=/usr/local/opt/llvm/bin/clang++ python3 -m pip install projectq
```

Small test program

```
1 from projectq import MainEngine
2 from projectq.ops import H
3
4 eng = MainEngine()
5 qubit = eng.allocate_qubit()
6 H | qubit
7
8 Measure | qubit
9 eng.flush()
10
11 print('Measured: {}'.format(int(qubit)))
```

Shortened URLs

Windows:

Python

64bits: <https://www.python.org/ftp/python/3.10.5/python-3.10.5-amd64.exe>

32bits: <https://www.python.org/ftp/python/3.10.5/python-3.10.5.exe>

Visual Studio

<https://visualstudio.microsoft.com/>

Visual C++ Build Tools

<https://visualstudio.microsoft.com/en/visual-cpp-build-tools/>

Microsoft Windows SDK

<https://developer.microsoft.com/en-us/windows/downloads/windows-10-sdk>

Mac OSX:

Homebrew:

<https://raw.githubusercontent.com/Homebrew/install/master/install>

Thank you for your attention

Any questions?

Bring digital to every person, home and organization for a fully connected intelligent world.

Copyright © 2022 Huawei Technologies Switzerland AG. All Rights Reserved.

The information in this document may contain predictive statements including, without limitation, statements regarding the future financial and operating results, future products, portfolio, new technology, etc. There are a number of factors that could cause actual results and developments to differ materially from those expressed or implied in the predictive statements. Therefore, such information is provided for reference purpose only and constitutes neither an offer nor an acceptance. Huawei may change the information at any time without notice.



Time for a break!