$|\mathrm{Lib}\rangle$: A Cross-Platform Programming Framework for Quantum-Accelerated Scientific Computing

Quantum Computing Thematic Track at *virtual* ICCS 2020

SCAN ME

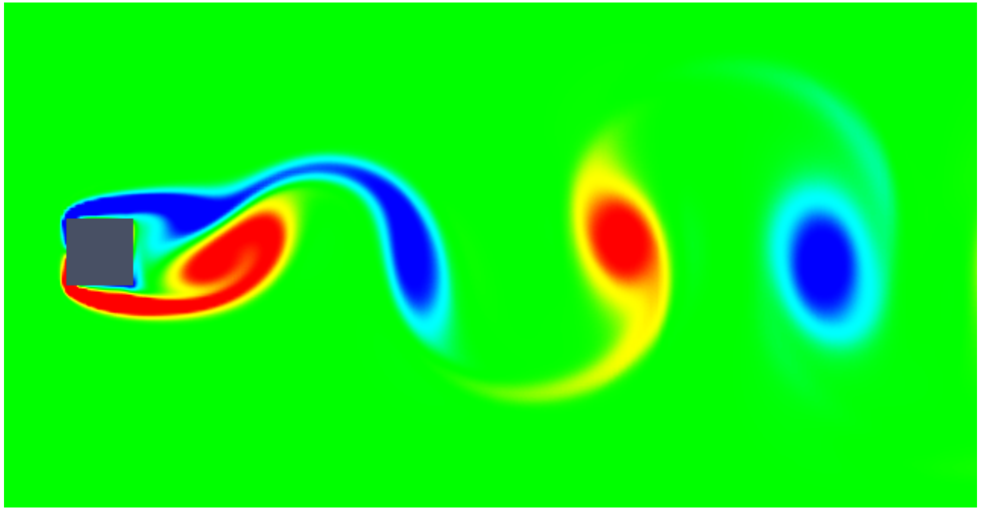
Matthias Möller and Merel Schalkers

Department of Applied Mathematics (DIAM)
Centre for Engineering Education (4TU.CEE)
Delft University of Technology

TU Delft

Numerical Analysis group

Scientific Computing

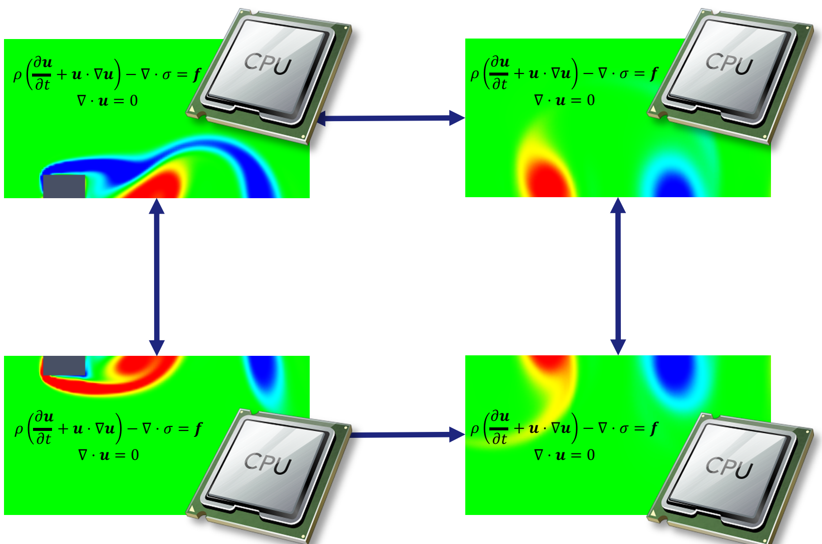


$$\rho\left(\frac{\partial \boldsymbol{u}}{\partial t} + \boldsymbol{u} \cdot \nabla \boldsymbol{u}\right) - \nabla \cdot \sigma = \boldsymbol{f}$$
$$\nabla \cdot \boldsymbol{u} = 0$$

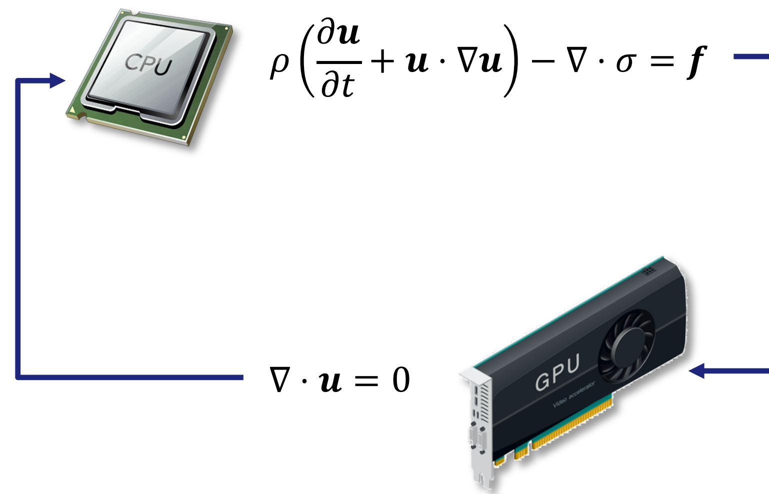
Karman vortex street animation by V. Fuka (https://artax.karlin.mff.cuni.cz/~fukav1am/sqcyl.html)

Scientific Computing

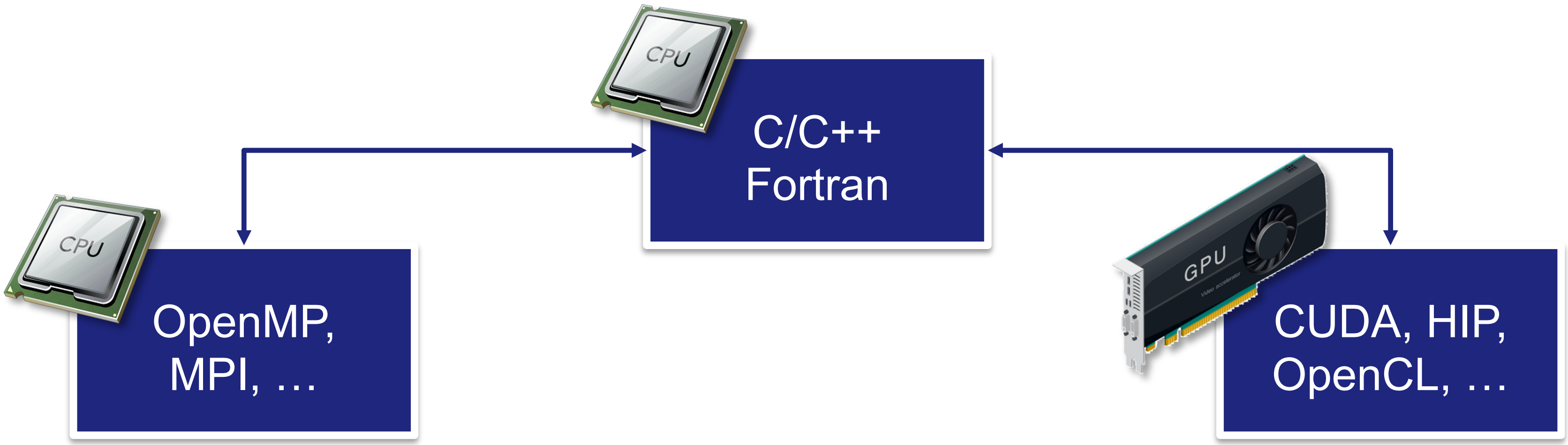
- Divide-and-conquer



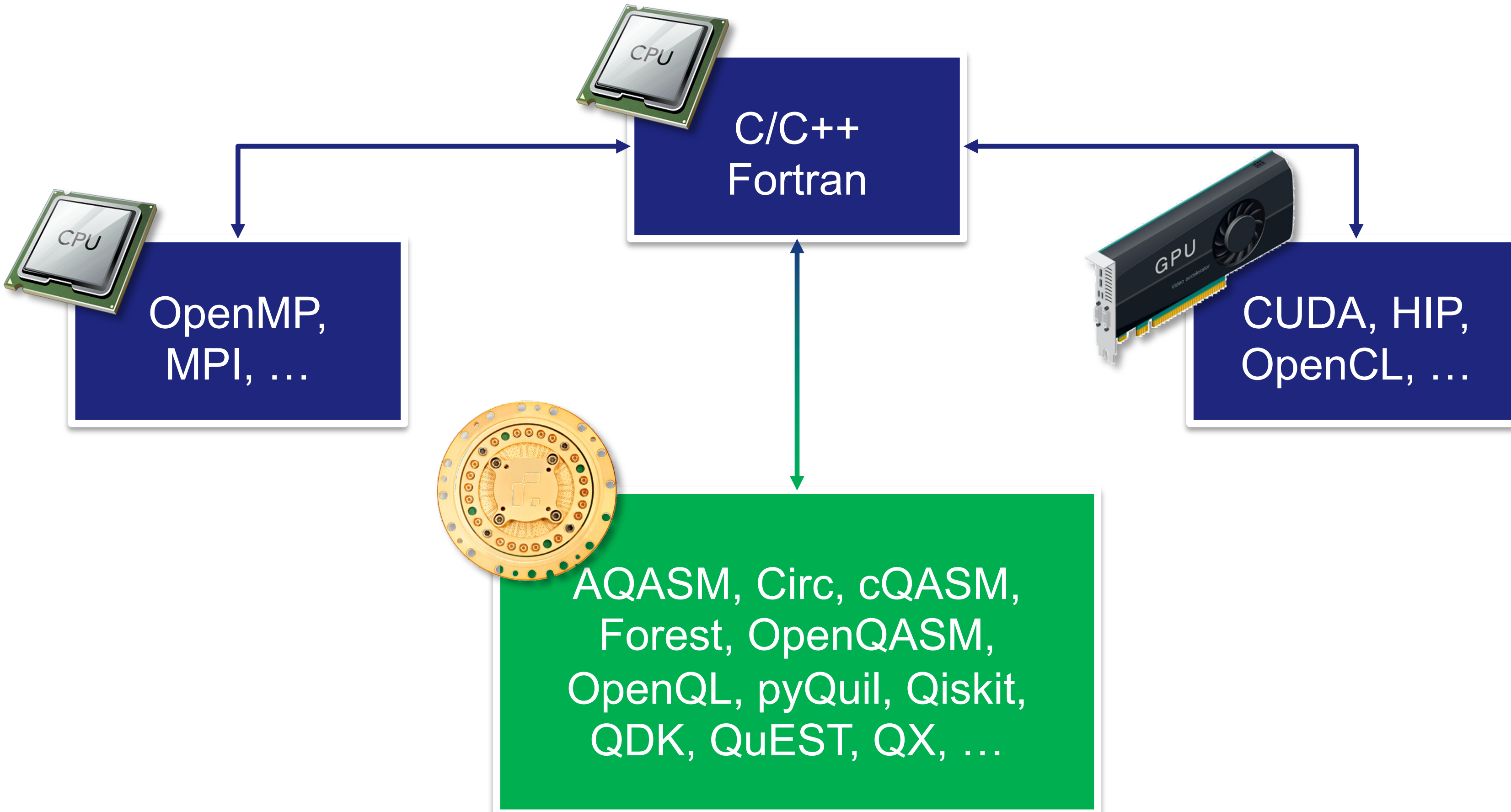
- Offloading



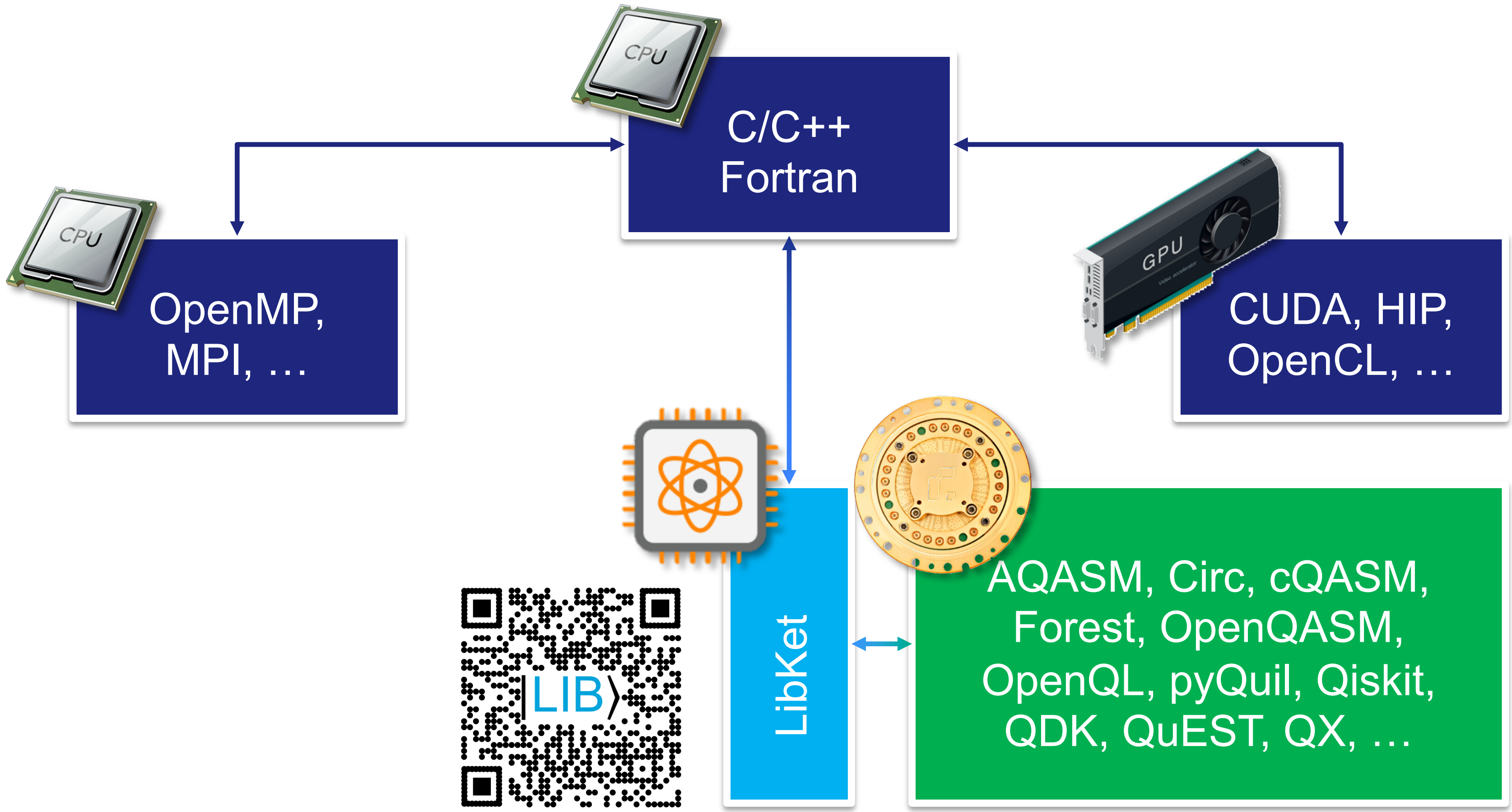
Programming Models



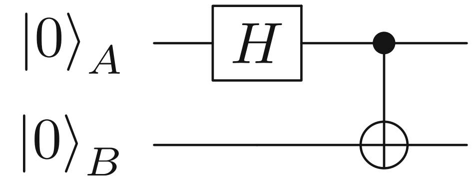
Programming Models



Programming Models

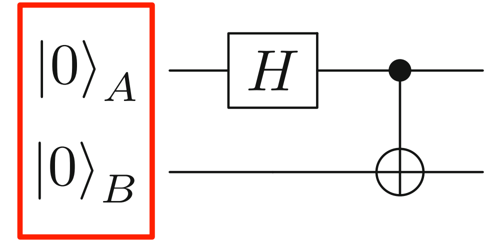


Example: First Bell state



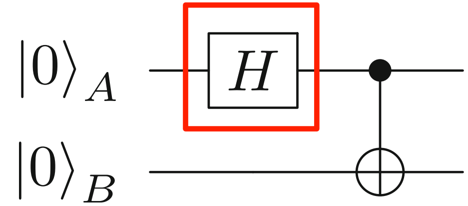

```cpp
#include <LibKet.hpp>

// Create quantum expression
auto expr = cnot(h(sel<1>()),
                 sel<3>(init()));

// Select quantum device
QDevice<ibmq_london, 5> device;

// Populate quantum kernel
device(expr);

// Execute quantum job
auto job = device.execute_async(…, [stream]);

// Wait for job and retrieve result
auto result = job->get();
```

Example: First Bell state



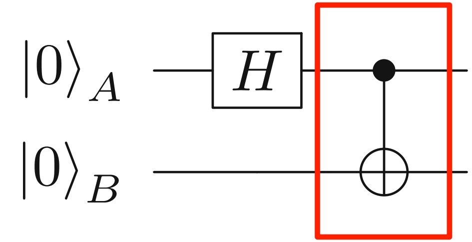

```
#include <LibKet.hpp>

// Create quantum expression
auto expr = cnot(h(sel<1>()),
                 sel<3>(init()));

// Select quantum device
QDevice<ibmq_london, 5> device;

// Populate quantum kernel
device(expr);

// Execute quantum job
auto job = device.execute_async(…, [stream]);

// Wait for job and retrieve result
auto result = job->get();
```

Example: First Bell state



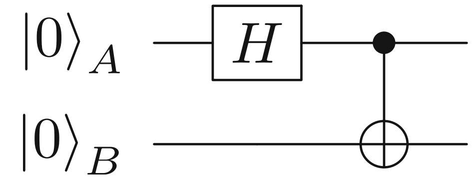

```cpp
#include <LibKet.hpp>

// Create quantum expression
auto expr = cnot(h(sel<1>()),
                 sel<3>(init()));

// Select quantum device
QDevice<ibmq_london, 5> device;

// Populate quantum kernel
device(expr);

// Execute quantum job
auto job = device.execute_async(…, [stream]);

// Wait for job and retrieve result
auto result = job->get();
```

Example: First Bell state




```
#include <LibKet.hpp>

// Create quantum expression
auto expr = cnot(h(sel<1>()),
                 sel<3>(init()));

// Select quantum device
QDevice<ibmq_london, 5> device;

// Populate quantum kernel
device(expr);

// Execute quantum job
auto job = device.execute_async(…, [stream]);

// Wait for job and retrieve result
auto result = job->get();
```

Example: First Bell state

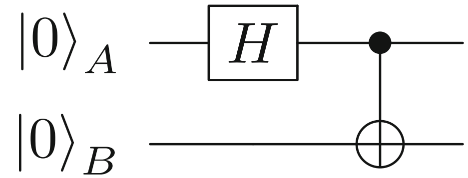



```cpp
#include <LibKet.hpp>

// Create quantum expression
auto expr = cnot(h(sel<1>()),
                 sel<3>(init()));

// Select quantum device
QDevice<ibmq_london, 5> device;

// Populate quantum kernel
device(expr);

// Execute quantum job
auto job = device.execute_async(…, [stream]);

// Wait for job and retrieve result
auto result = job->get();
```

Abstract syntax tree of the quantum expression

```
BinaryQGate
|   gate = QCNOT
| filter = QFilterSelect [ 1 3 ]
|  expr0 = UnaryQGate
|              |   gate = QHadamard
|              | filter = QFilterSelect [ 1 ]
|              |   expr = QFilterSelect [ 1 ]
|  expr1 = UnaryQGate
|              |   gate = QInit
|              | filter = QFilterSelect [ 3 ]
|              |   expr = QFilter
```

Example: First Bell state

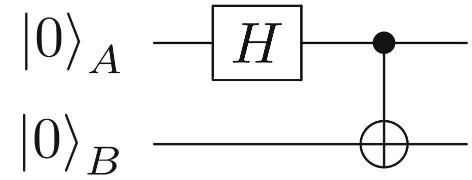



```cpp
#include <LibKet.hpp>

// Create quantum expression
auto expr = cnot(h(sel<1>()),
                 sel<3>(init()));

// Select quantum device
QDevice<ibmq_london, 5> device;

// Populate quantum kernel
device(expr);

// Execute quantum job
auto job = device.execute_async(…, [stream]);

// Wait for job and retrieve result
auto result = job->get();
```

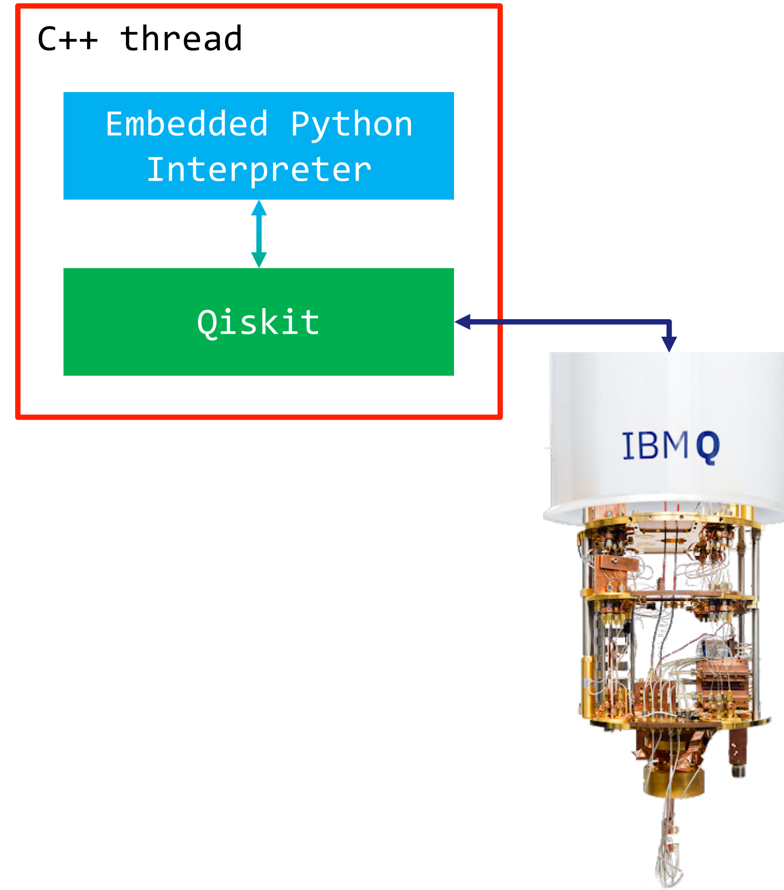
OpenQASM kernel

```
OPENQASM 2.0;
include "qelib1.inc";
qreg q[5];
creg c[5];
h q[1];
cnot q[1], q[3];
```

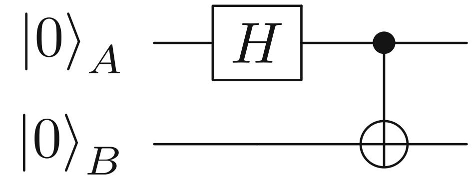
Example: First Bell state



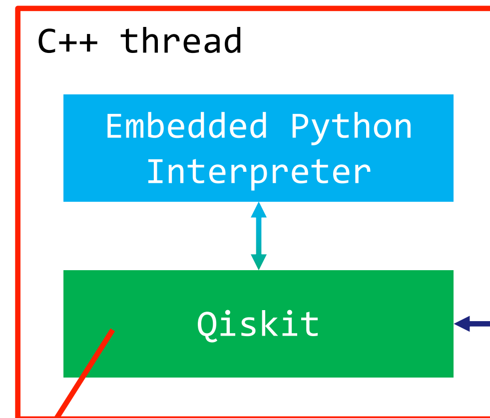

```
#include <LibKet.hpp>

// Create quantum expression
auto expr = cnot(h(sel<1>()),
                 sel<3>(init()));

// Select quantum device
QDevice<ibmq_london, 5> device;

// Populate quantum kernel
device(expr);

// Execute quantum job
auto job = device.execute_async(…, [stream]);

// Wait for job and retrieve result
auto result = job->get();
```



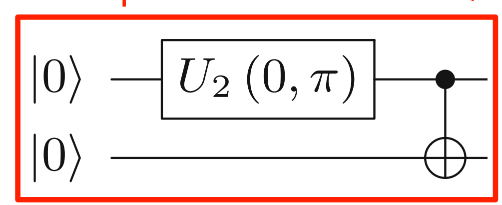
Example: First Bell state



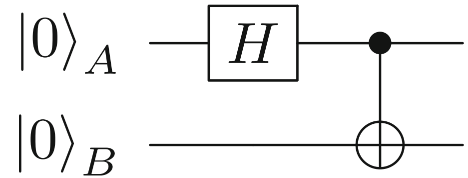
```cpp
#include <LibKet.hpp>

// Create quantum expression
auto expr = cnot(h(sel<1>()),
                 sel<3>(init()));

// Select quantum device
QDevice<ibmq_london, 5> device;

// Populate quantum kernel
device(expr);

// Execute quantum job
auto job = device.execute_async(…, [stream]);

// Wait for job and retrieve result
auto result = job->get();
```



Transpiled circuit for IBM Q

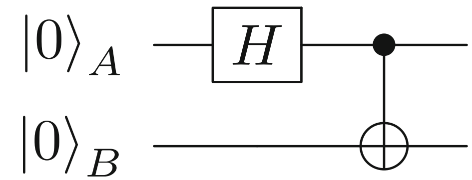


Example: First Bell state

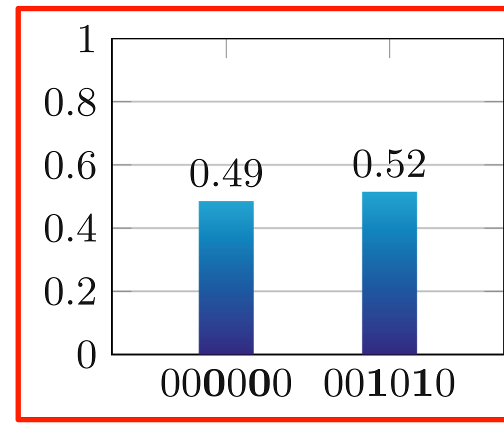



```
#include <LibKet.hpp>

// Create quantum expression
auto expr = cnot(h(sel<1>()),
                 sel<3>(init()));

// Select quantum device
QDevice<ibmq_london, 5> device;

// Populate quantum kernel
device(expr);

// Execute quantum job
auto job = device.execute_async(…, [stream]);

// Do other stuff while waiting

// Wait for job and retrieve result
auto result = job->get();
```

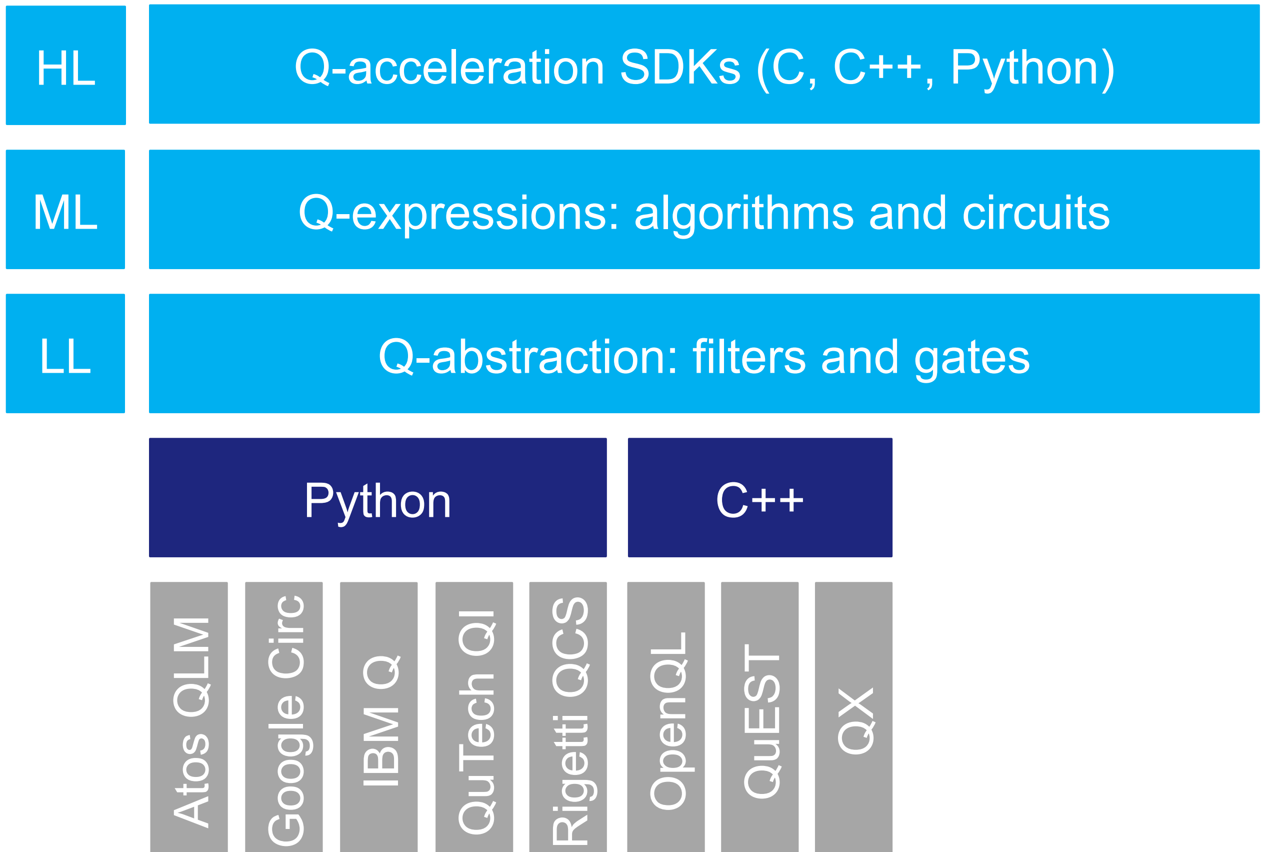
Example: First Bell state



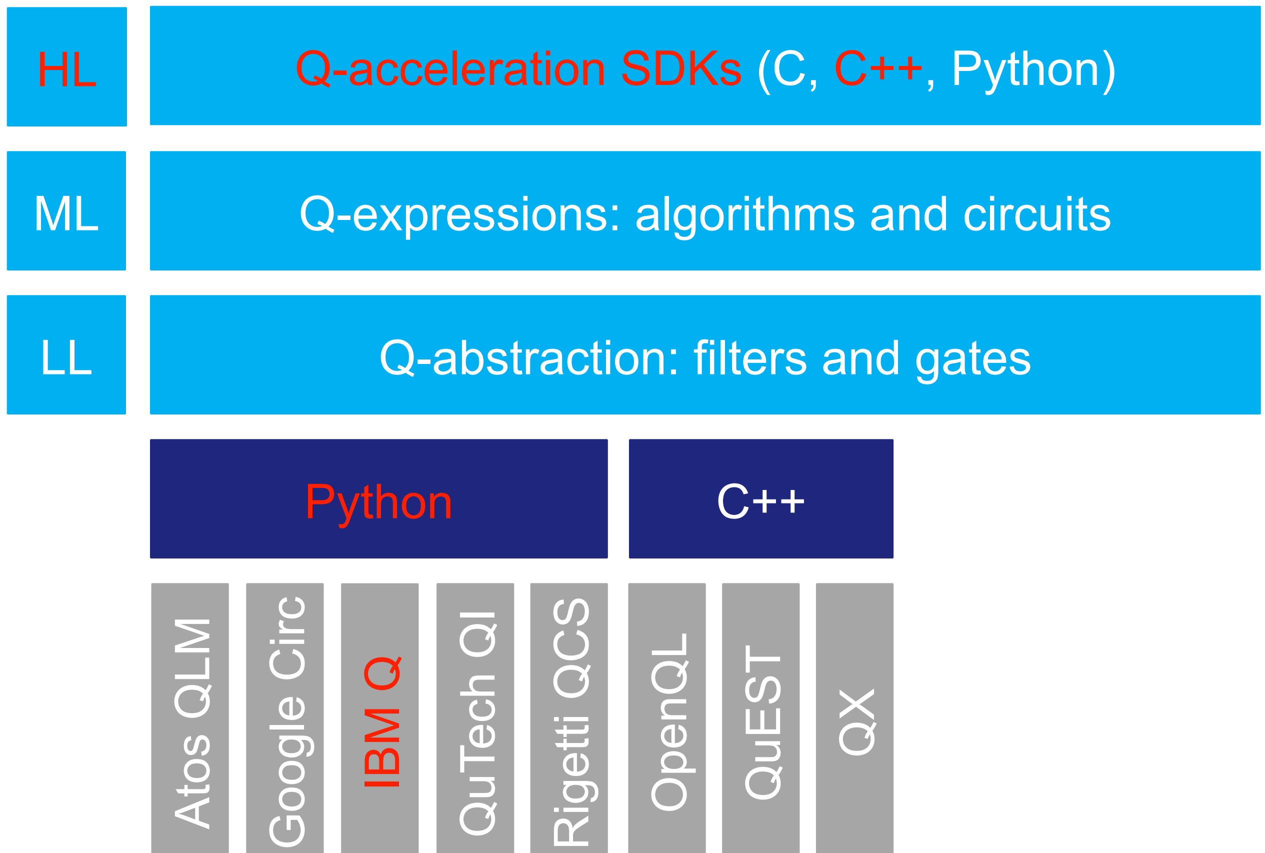

```
#include <LibKet.hpp>

// Create quantum expression
auto expr = cnot(h(sel<1>()),
                 sel<3>(init()));

// Select quantum device
QDevice<ibmq_london, 5> device;

// Populate quantum kernel
device(expr);

// Execute quantum job
auto job = device.execute_async(…, [stream]);

// Do other stuff while waiting

// Wait for job and retrieve result
auto result = job->get();
```



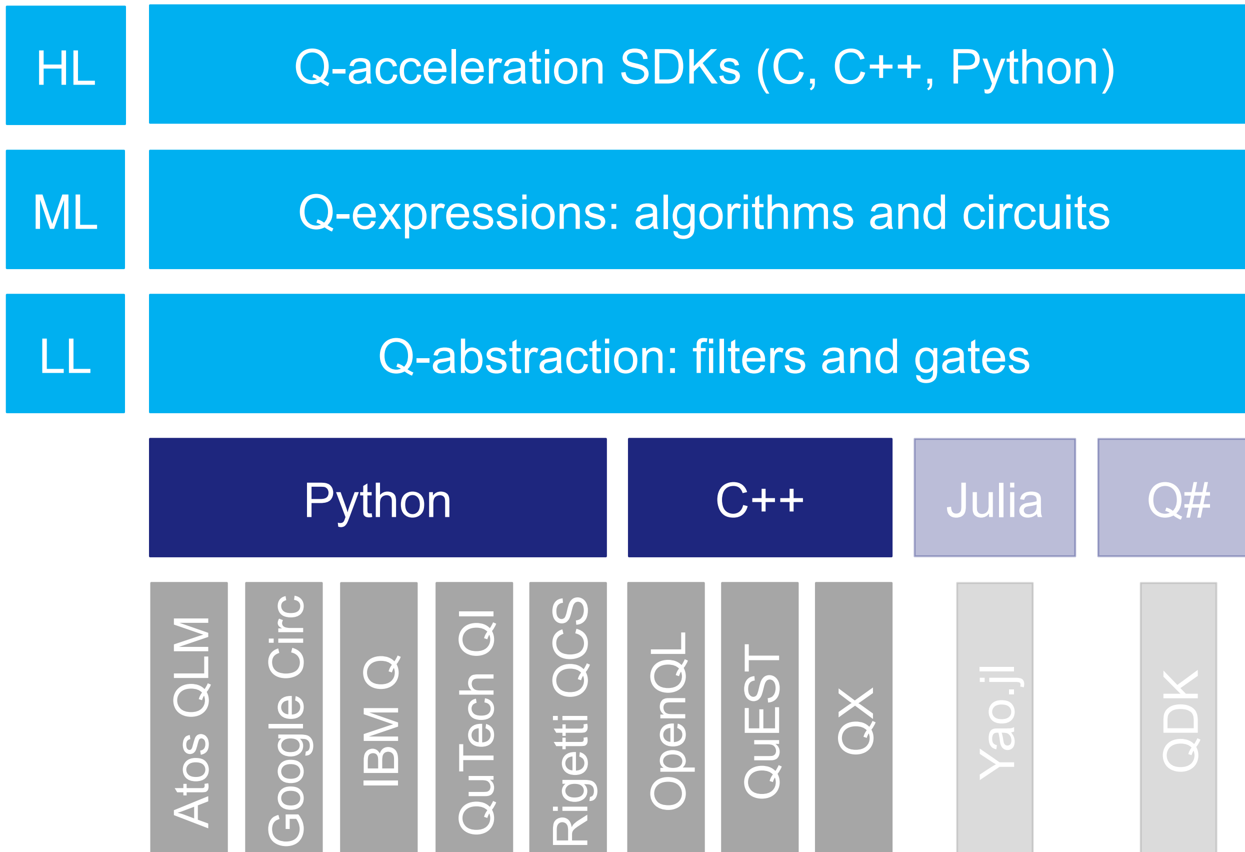
Kwantum expression template Library



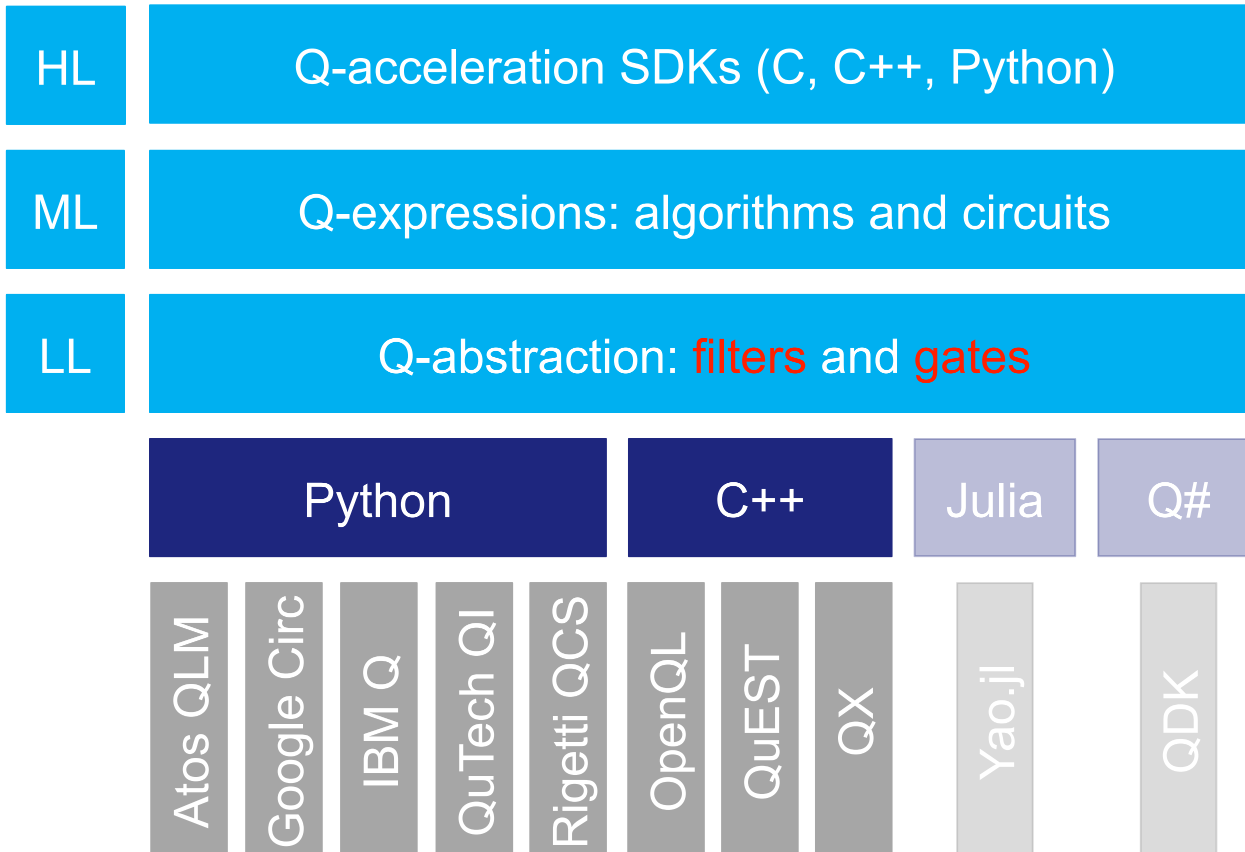
Kwantum expression template Library



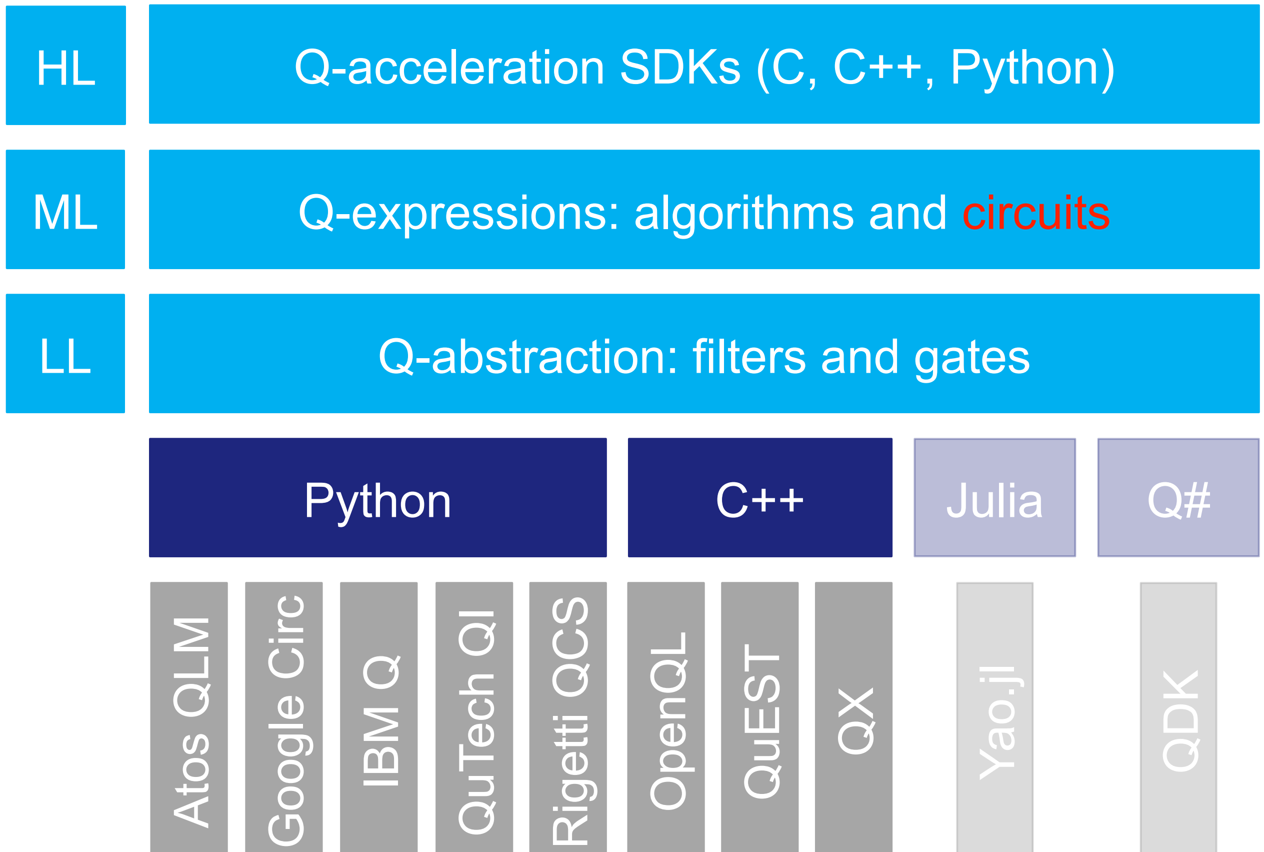
Kwantum expression template Library



Kwantum expression template Library



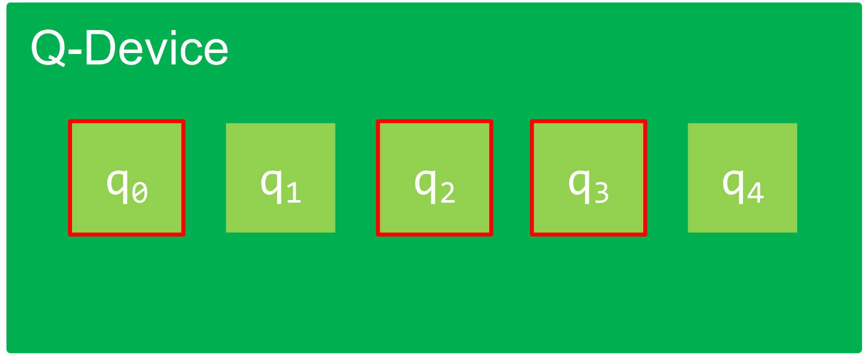
Kwantum expression template Library



Filters – views on the global Q-memory

- Starting from the full Q-memory filters restrict qubits step by step

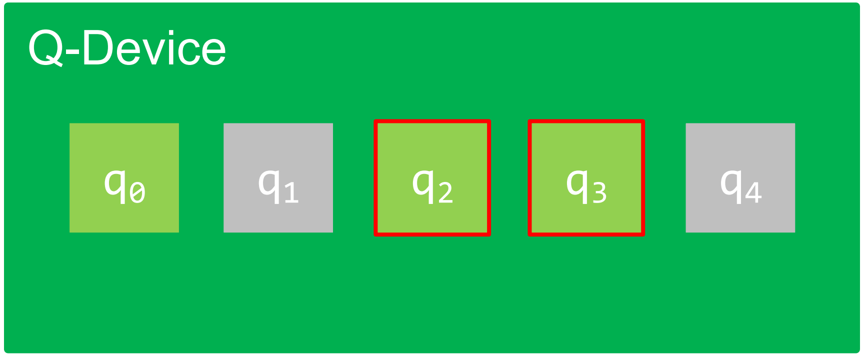
```
auto f0 = select<0,2,3>();
```



Filters – views on the global Q-memory

- Starting from the full Q-memory filters restrict qubits step by step

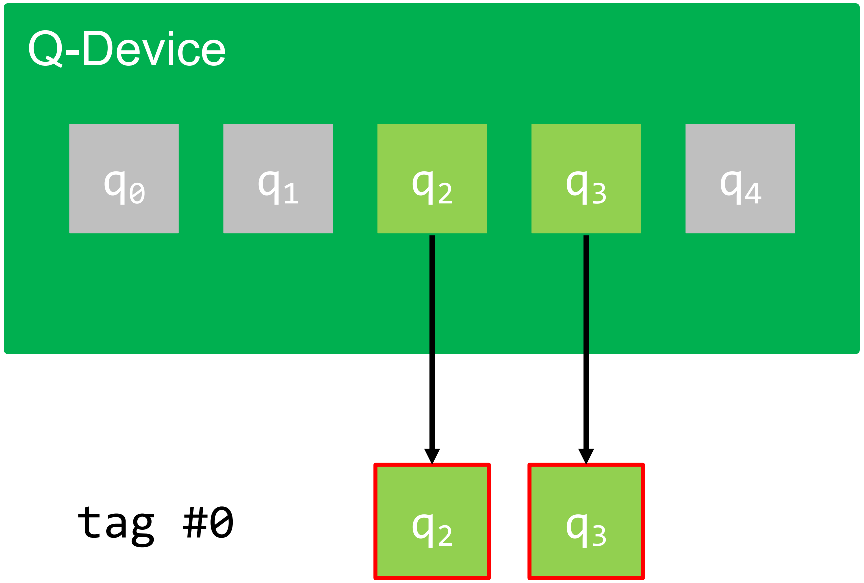
```
auto f0 = select<0,2,3>();
auto f1 = range<1,2>(f0);
```



Filters – views on the global Q-memory

- Starting from the full Q-memory filters restrict qubits step by step

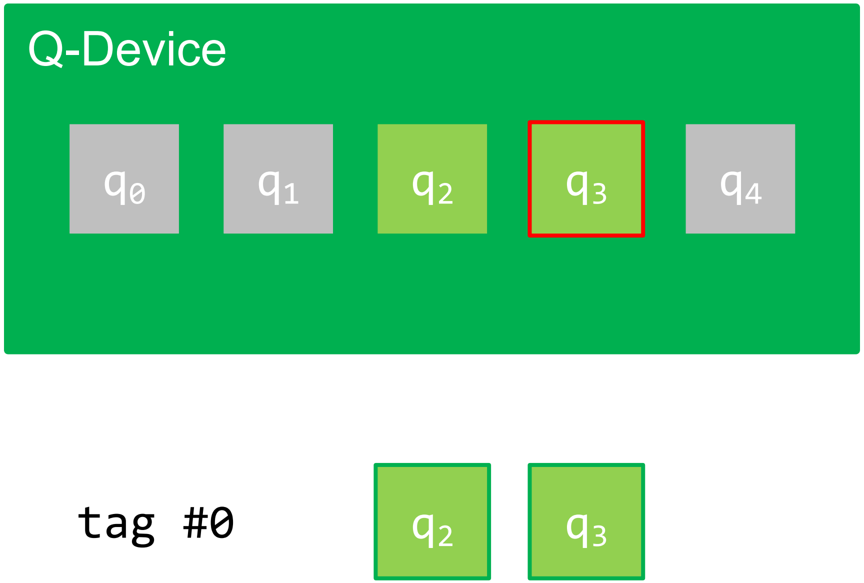
```
auto f0 = select<0,2,3>();
auto f1 = range<1,2>(f0);
auto f2 = tag<0>(f1);
```



Filters – views on the global Q-memory

- Starting from the full Q-memory filters restrict qubits step by step

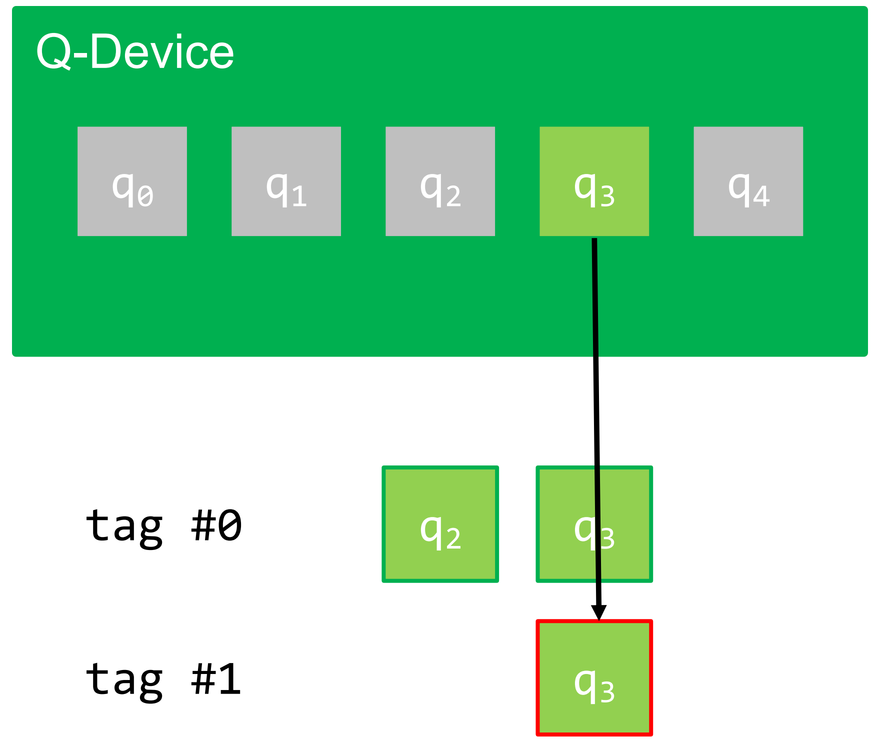
```
auto f0 = select<0,2,3>();
auto f1 = range<1,2>(f0);
auto f2 = tag<0>(f1);
auto f3 = qubit<1>(f2);
```



Filters – views on the global Q-memory

- Starting from the full Q-memory filters restrict qubits step by step

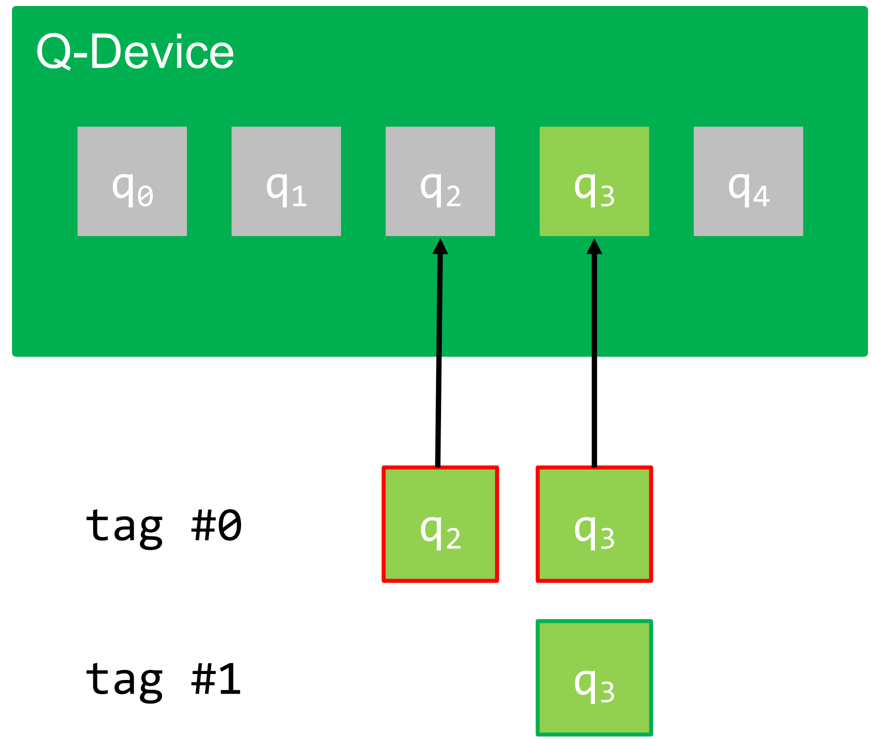
```
auto f0 = select<0,2,3>();
auto f1 = range<1,2>(f0);
auto f2 = tag<0>(f1);
auto f3 = qubit<1>(f2);
auto f4 = tag<1>(f3);
```



Filters – views on the global Q-memory

- Starting from the full Q-memory filters restrict qubits step by step

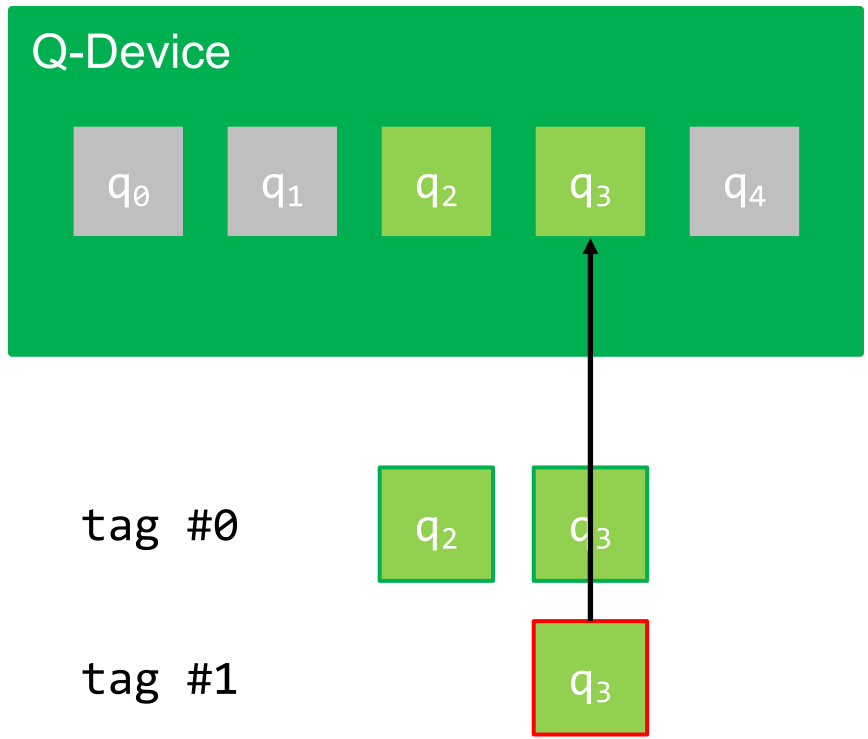
```
auto f0 = select<0,2,3>();
auto f1 = range<1,2>(f0);
auto f2 = tag<0>(f1);
auto f3 = qubit<1>(f2);
auto f4 = tag<1>(f3);
auto f5 = gototag<0>(f4);
```



Filters – views on the global Q-memory

- Starting from the full Q-memory filters restrict qubits step by step

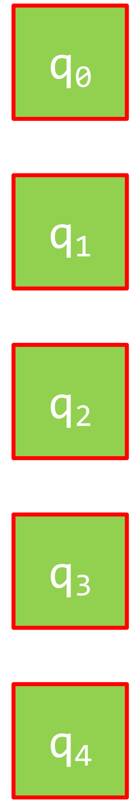
```
auto f0 = select<0,2,3>();
auto f1 = range<1,2>(f0);
auto f2 = tag<0>(f1);
auto f3 = qubit<1>(f2);
auto f4 = tag<1>(f3);
auto f5 = gototag<0>(f4);
auto f6 = gototag<1>(f5);
```



Gates – SIMD-ops

- Gates apply to all qubits of the current filter chain (SIMD-ops)

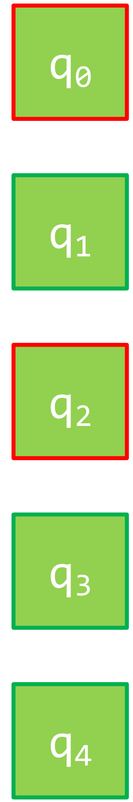
```
auto e0 = init();
```



Gates – SIMD-ops

- Gates apply to all qubits of the current filter chain (SIMD-ops)

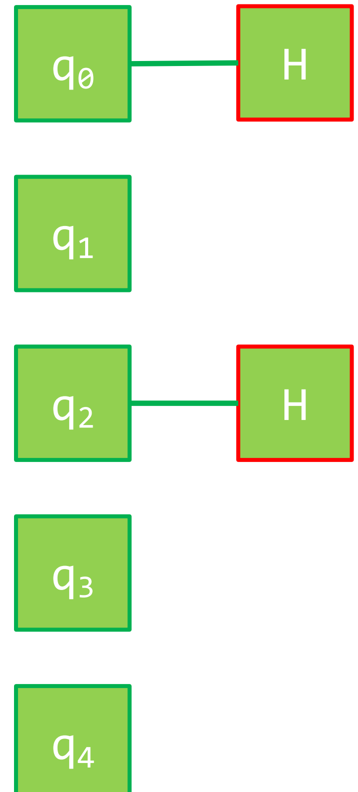
```
auto e0 = init();
auto e1 = sel<0,2>(e0);
```



Gates – SIMD-ops

- Gates apply to all qubits of the current filter chain (SIMD-ops)

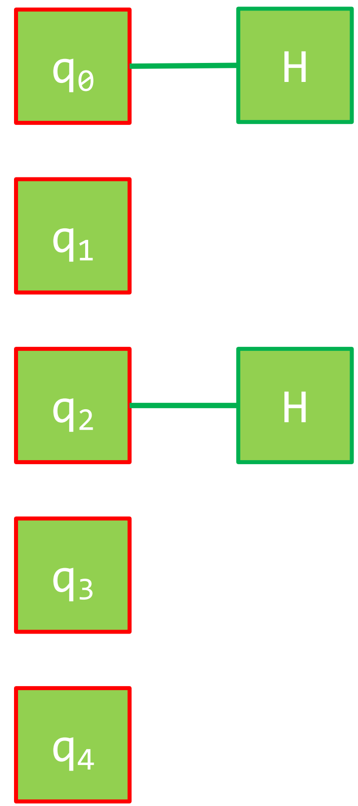
```
auto e0 = init();
auto e1 = sel<0,2>(e0);
auto e2 = h(e1);
```



Gates – SIMD-ops

- Gates apply to all qubits of the current filter chain (SIMD-ops)

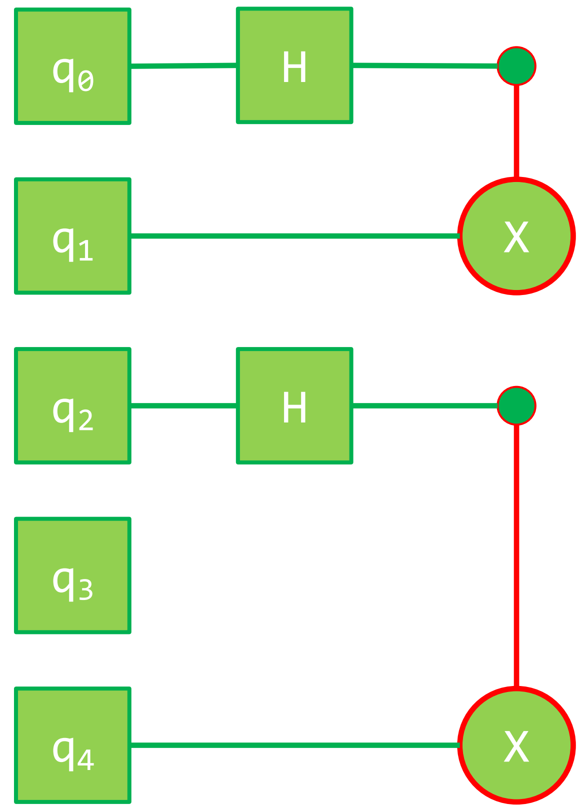
```
auto e0 = init();
auto e1 = sel<0,2>(e0);
auto e2 = h(e1);
auto e3 = all(e2);
```



Gates – SIMD-ops

- Gates apply to all qubits of the current filter chain (SIMD-ops)

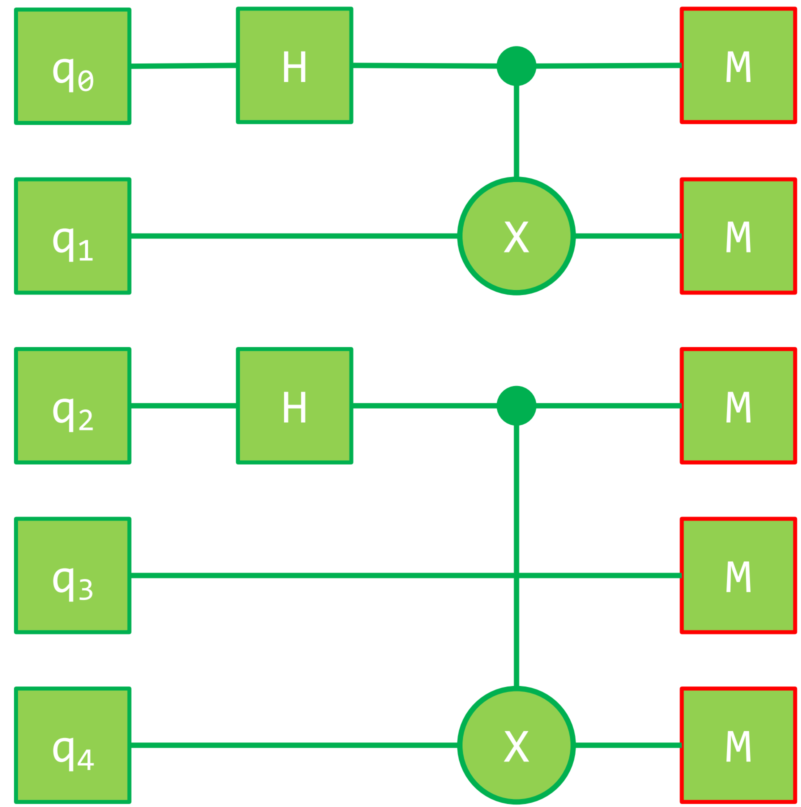
```
auto e0 = init();
auto e1 = sel<0,2>(e0);
auto e2 = h(e1);
auto e3 = all(e2);
auto e4 = cnot(
            sel<0,2>(),
            sel<1,4>(e3)
          );
```



Gates – SIMD-ops

- Gates apply to all qubits of the current filter chain (SIMD-ops)

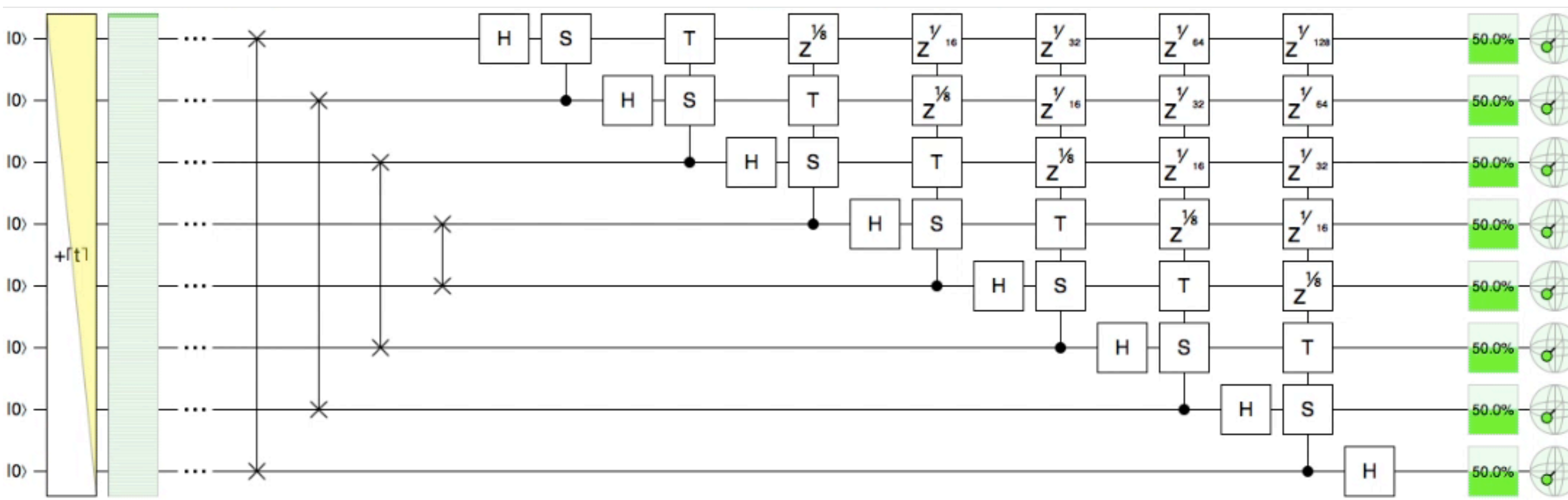
```
auto e0 = init();
auto e1 = sel<0,2>(e0);
auto e2 = h(e1);
auto e3 = all(e2);
auto e4 = cnot(
            sel<0,2>(),
            sel<1,4>(e3)
         );
auto e5 = measure(all(e4));
```



Circuits – pre-cooked quantum building blocks

- Generic algorithms that can be applied to registers of arbitrary size n

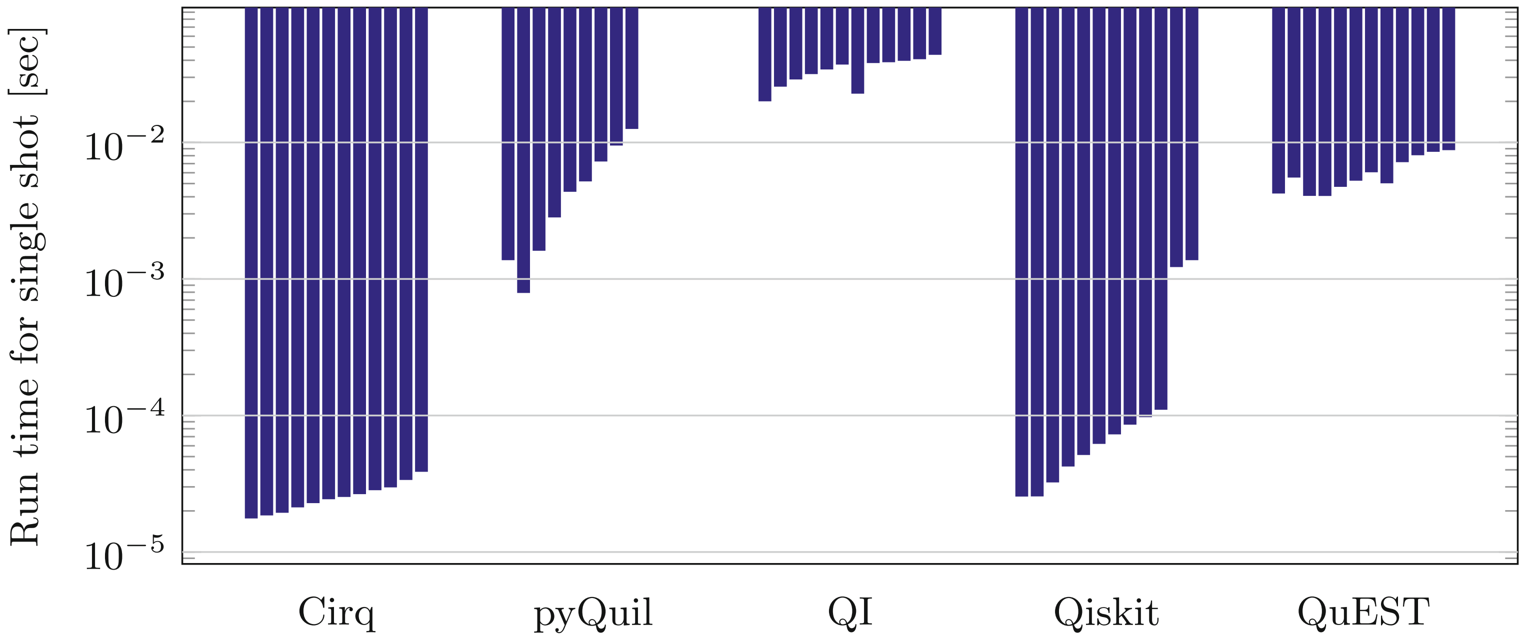
```
auto expr = qft(range<0,n>(init()));
```



Animation created with Quirk https://algassert.com/quirk

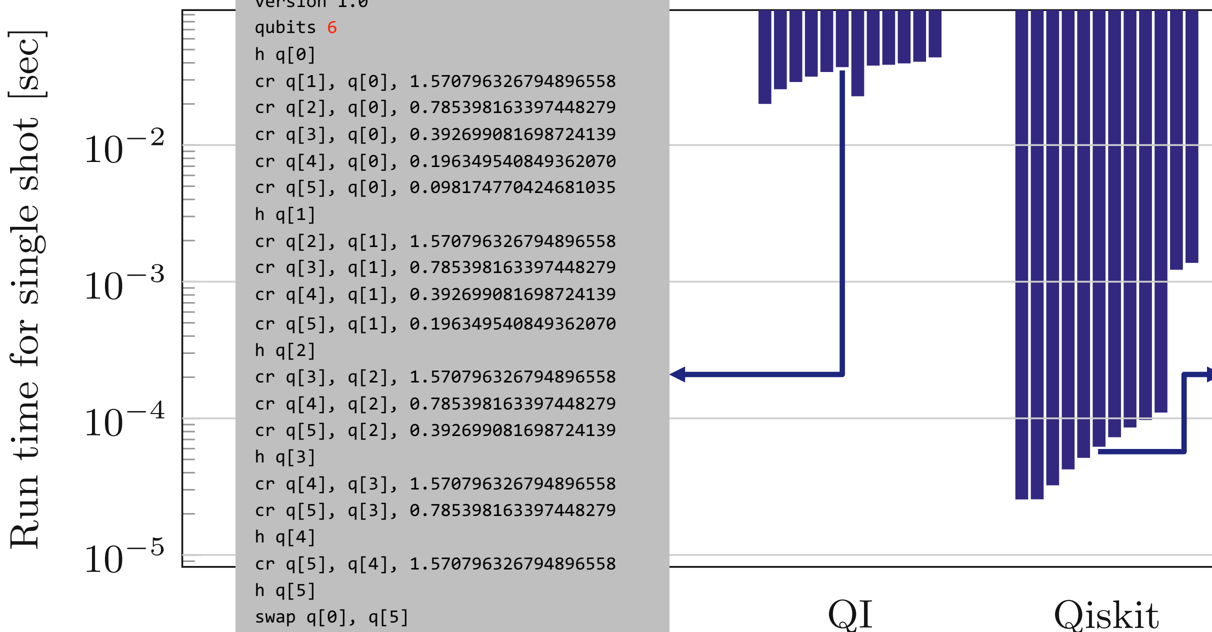
Example: n-qubit QFT benchmark

- Execute n-qubit QFT for `n=1..12` on different quantum simulators



Example: n-qubit QFT benchmark

- Execute n-qubit QFT for `n=1..12` on different quantum simulators




```
version 1.0
qubits 6
h q[0]
cr q[1], q[0], 1.570796326794896558
cr q[2], q[0], 0.785398163397448279
cr q[3], q[0], 0.392699081698724139
cr q[4], q[0], 0.196349540849362070
cr q[5], q[0], 0.098174770424681035
h q[1]
cr q[2], q[1], 1.570796326794896558
cr q[3], q[1], 0.785398163397448279
cr q[4], q[1], 0.392699081698724139
cr q[5], q[1], 0.196349540849362070
h q[2]
cr q[3], q[2], 1.570796326794896558
cr q[4], q[2], 0.785398163397448279
cr q[5], q[2], 0.392699081698724139
h q[3]
cr q[4], q[3], 1.570796326794896558
cr q[5], q[3], 0.785398163397448279
h q[4]
cr q[5], q[4], 1.570796326794896558
h q[5]
swap q[0], q[5]
swap q[1], q[4]
swap q[2], q[3]
```

```
OPENQASM 2.0;
include "qelib1.inc";
qreg q[6];
creg c[6];
h q[0];
cu1(1.570796326794896558) q[1], q[0];
cu1(0.785398163397448279) q[2], q[0];
cu1(0.392699081698724139) q[3], q[0];
cu1(0.196349540849362070) q[4], q[0];
cu1(0.098174770424681035) q[5], q[0];
h q[1];
cu1(1.570796326794896558) q[2], q[1];
cu1(0.785398163397448279) q[3], q[1];
cu1(0.392699081698724139) q[4], q[1];
cu1(0.196349540849362070) q[5], q[1];
h q[2];
cu1(1.570796326794896558) q[3], q[2];
cu1(0.785398163397448279) q[4], q[2];
cu1(0.392699081698724139) q[5], q[2];
h q[3];
cu1(1.570796326794896558) q[4], q[3];
cu1(0.785398163397448279) q[5], q[3];
h q[4];
cu1(1.570796326794896558) q[5], q[4];
h q[5];
swap q[0], q[5];
swap q[1], q[4];
swap q[2], q[3];
```

Advanced features

- Rule-based optimization

$$U \circ U^\dagger = U^\dagger \circ U = id$$

- Compile-time for loops

```
auto expr = static_for<begin,end,step,ftor_body>(…)
```

- User-definable gates

```
QFunctor_alias(Bell, cnot(h(sel<1>()),sel<3>(init())));
auto expr = hook<Bell>(…)
```

- Just-in-time compilation of string expressions

```
device("cnot(h(sel<1>()),sel<3>(init()))");
```

Conclusion

|Lib⟩: **A Cross-Platform Programming Framework for Quantum-Accelerated Scientific Computing,** https://doi.org/10.1007/978-3-030-50433-5_35

- Rapid prototyping and testing of QAs from quantum expression templates
- Seamless integration of QAs into classical scientific computing applications
- Support for Atos, Circ, IBMQ, QX, Quantum Inspire, QuEST, Rigetti, …
- C++14 header-only library with unified C and Python API

Acknowledgements

- Kelvin Loh and Richard Versluis (TNO)
- TNO and 4TU.CEE for financial support