Yao.jl

An **extensible** efficient framework for quantum algorithm design

About me

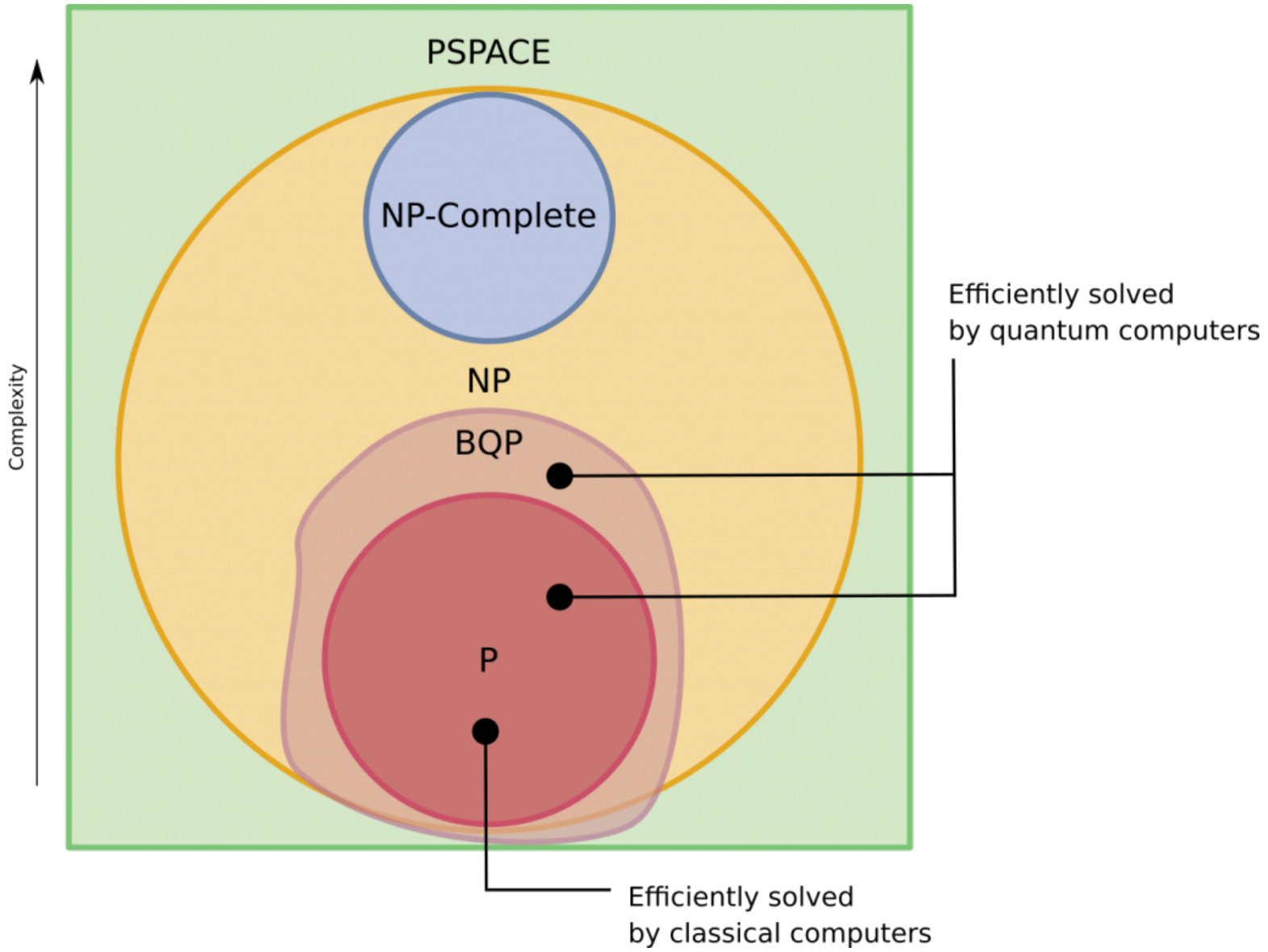
Xiuzhe (Roger) Luo
罗秀哲

Co-author
Jin-guo Liu, Pan Zhang, Lei Wang

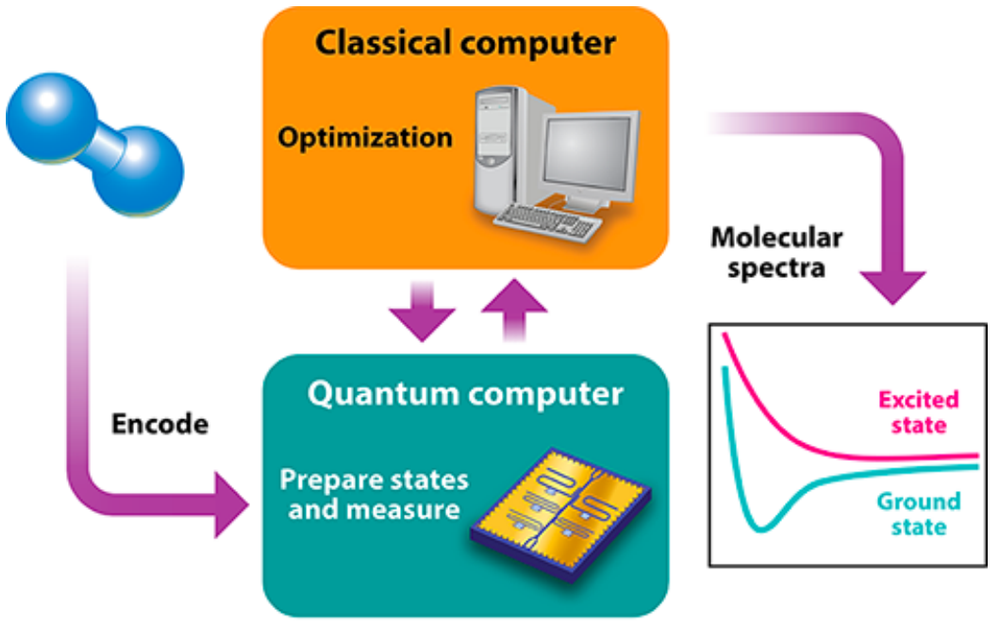
Rogerluo
Roger-luo

QUANTUM INTELLIGENCE LAB

UNIVERSITY OF WATERLOO

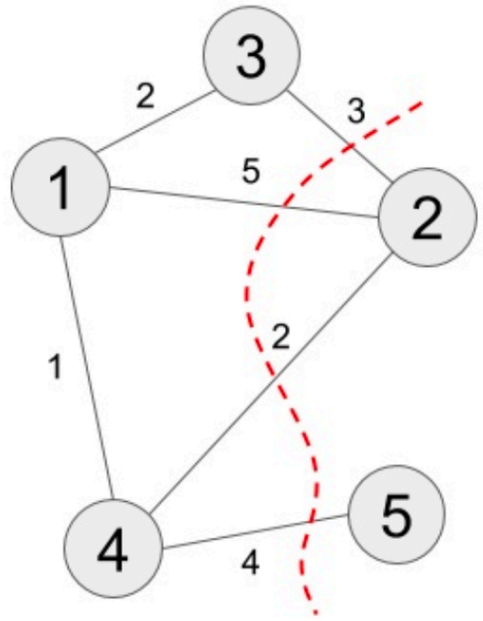


Raz, Ran, and Avishay Tal. "Oracle Separation of BQP and PH." Proceedings of the 51st Annual ACM SIGACT Symposium on Theory of Computing. ACM, 2019.



Chemistry

Sim, Sukin, et al. "Quantum computer simulates excited states of molecule." *Physics* 11 (2018): 14.



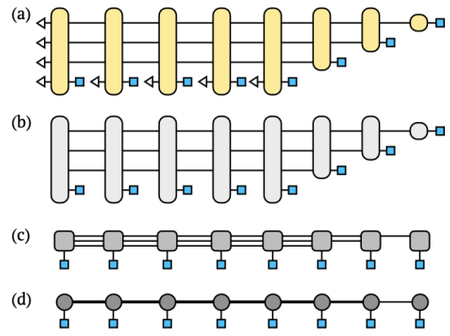
Combinatoric Optimization

Farhi, Edward, Jeffrey Goldstone, and Sam Gutmann. "A quantum approximate optimization algorithm." *arXiv preprint arXiv:1411.4028* (2014).

$$N = p \cdot q$$

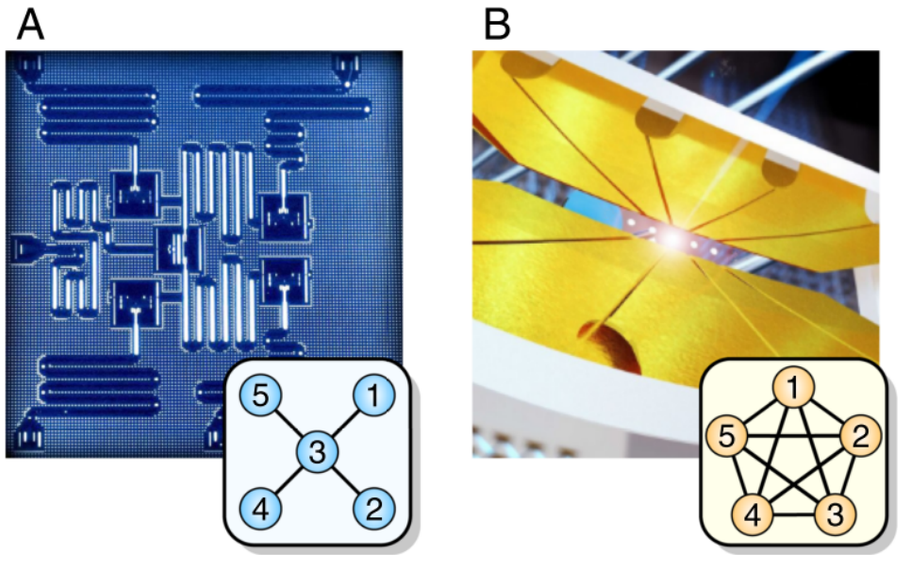
Cryptography

Shor, Peter W. "Algorithms for quantum computation: Discrete logarithms and factoring." *Proceedings 35th annual symposium on foundations of computer science*. Ieee, 1994.



Machine Learning

Huggins, William James, et al. "Towards quantum machine learning with tensor networks." *Quantum Science and technology* (2018).



Experimental comparison of two quantum computing architectures

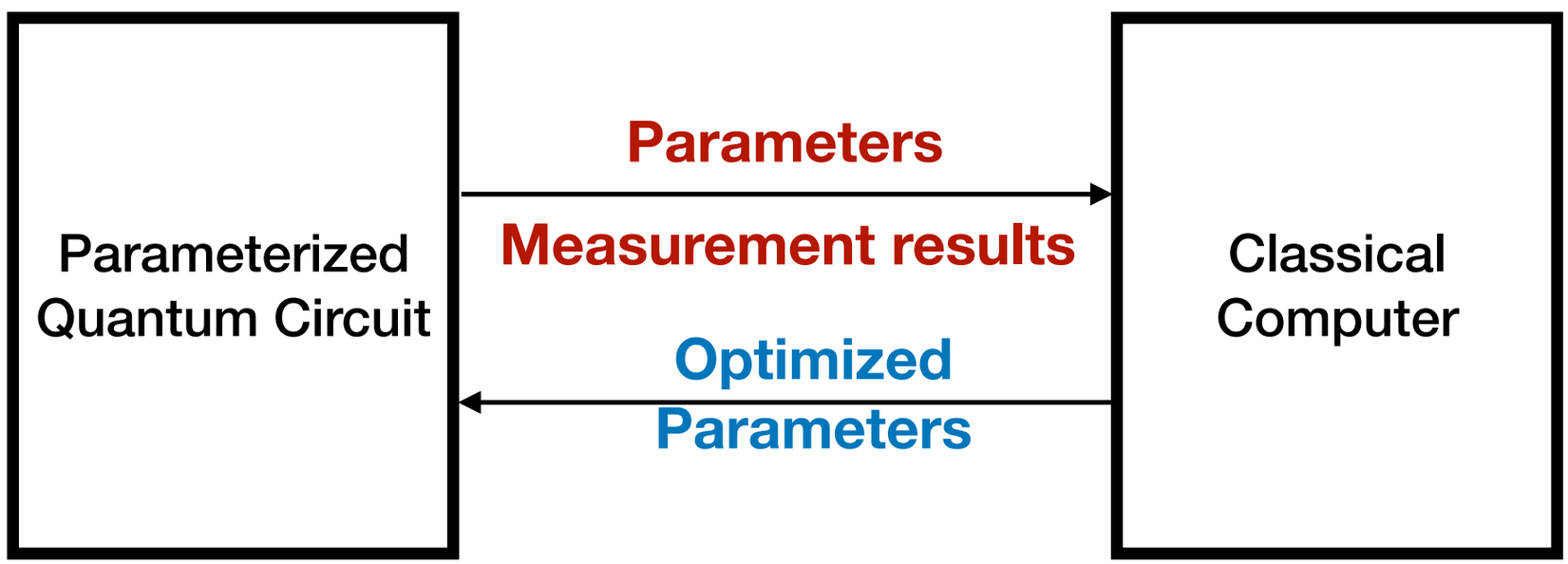
Norbert M. Linke[a,b,1], Dmitri Maslov[c], Martin Roetteler[d], Shantanu Debnath[a,b], Caroline Figgatt[a,b], Kevin A. Landsman[a,b], Kenneth Wright[a,b], and Christopher Monroe[a,b,e,1]

	1 qubit gate T_{gate}(fidelity)	2 qubit gate T_{gate}/(fidelity)	T1	T2	1 qubit readout fidelity	5 qubit readout fidelity
SQUID	130 ns (99.7%)	250–450 ns (96.5%)	60μs	60μs	96%	80%
Ion Trap	20μs (99.1%)	250μs (97%)	∞	0.5 s	99.7% for |0> 99.1% for |1>	95.7%

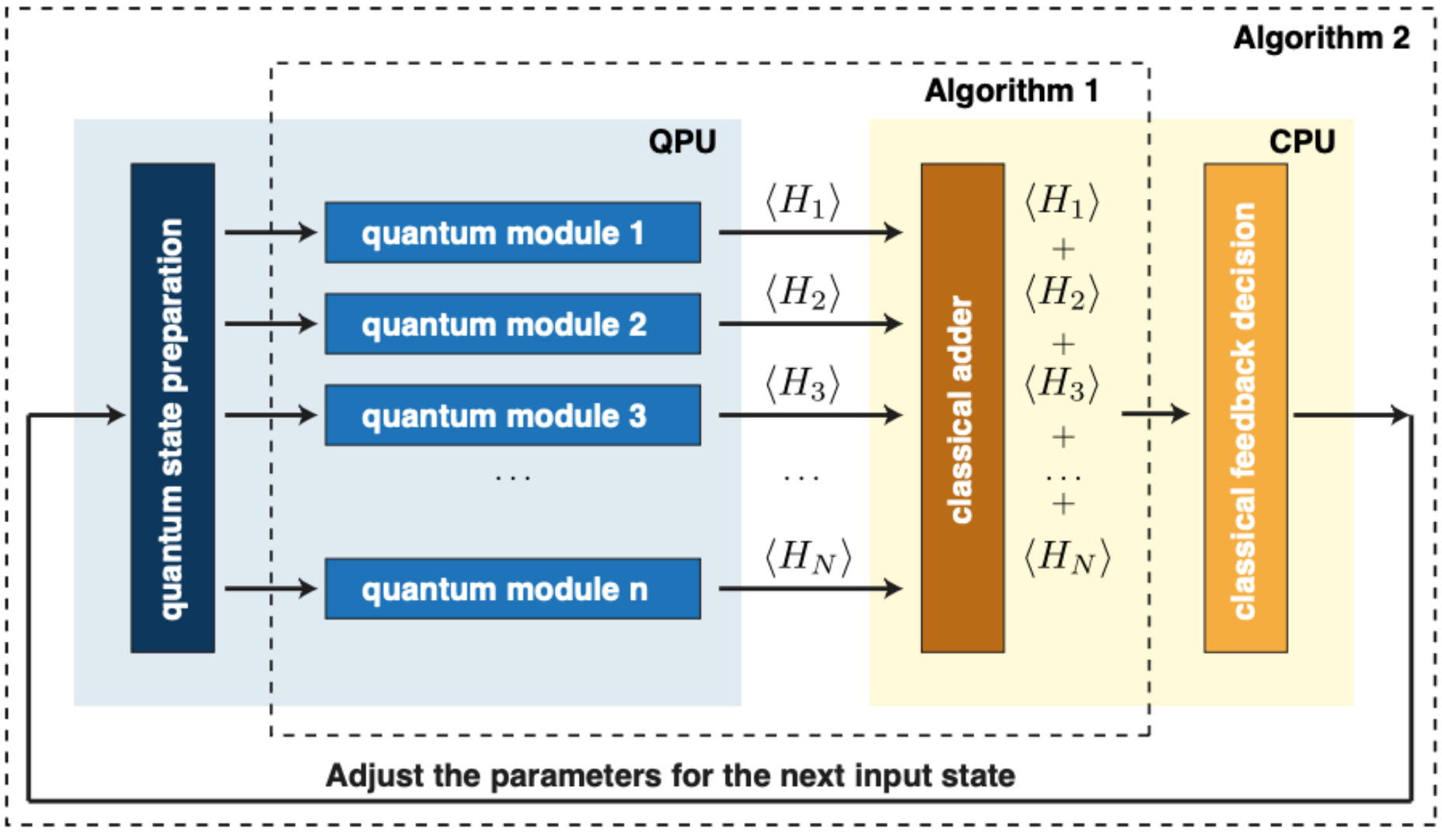
Variational Quantum Algorithms/Circuits

Reference	Name	Developer	Features	Language	Backend
Aleksandrowicz *et al.* [28]	Qiskit Aqua	IBM Research	VQE, QAOA, VQM, QKE	Python	Superconducting, Simulator
Bergholm *et al.* [29]	Pennylane	Xanadu	VQE, VQM, QGAN	Python	Superconducting, Simulator
Luo *et al.* [30]	Yao	QuantumBFS	VQE, QAOA, QCBM	Julia	Simulator

Benedetti, Marcello, Erika Lloyd, and Stefan Sack. "Parameterized quantum circuits as machine learning models." *arXiv:1906.07682* (2019).



Variational Quantum Eigensolver



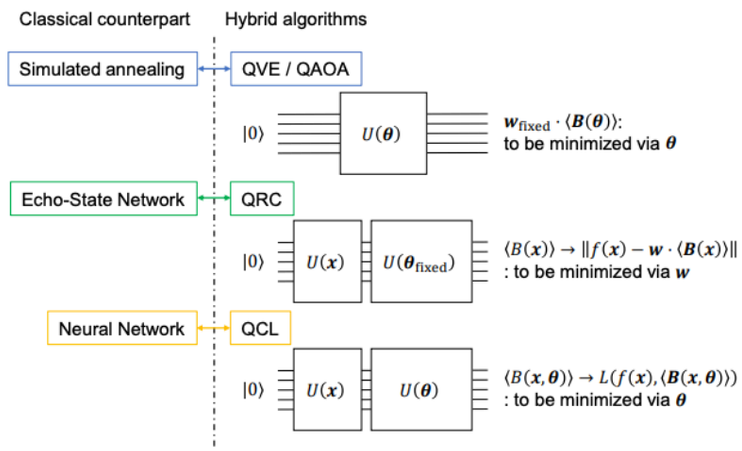
Peruzzo, Alberto, et al. "A variational eigenvalue solver on a photonic quantum processor." *Nature communications* 5 (2014): 4213.

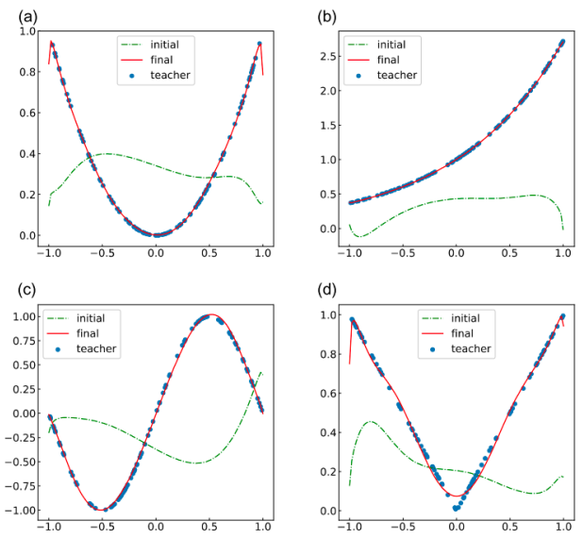
Quantum Circuit Learning

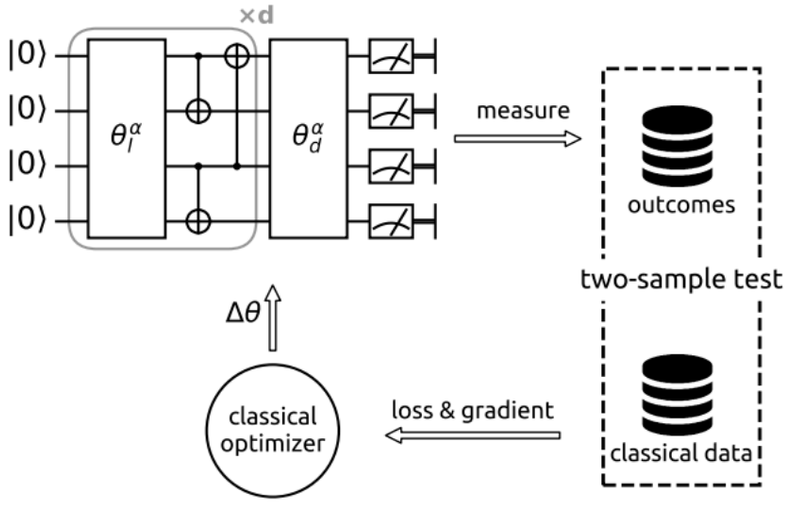
$$\frac{\partial \langle B \rangle}{\partial \eta} = \frac{1}{2}(\langle B \rangle_{\eta \to \eta + \frac{\pi}{2}} - \langle B \rangle_{\eta \to \eta - \frac{\pi}{2}})$$

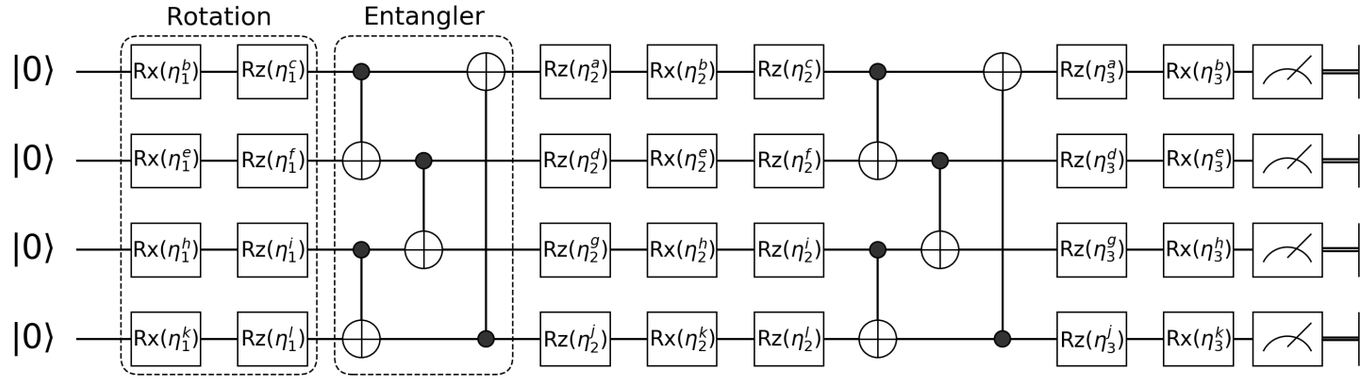
Mitarai, Kosuke, et al. "Quantum circuit learning." *Physical Review A* 98.3 (2018): 032309.

Nakanishi, Ken M., Keisuke Fujii, and Synge Todo. "Sequential minimal optimization for quantum-classical hybrid algorithms." arXiv preprint arXiv:1903.12166 (2019).



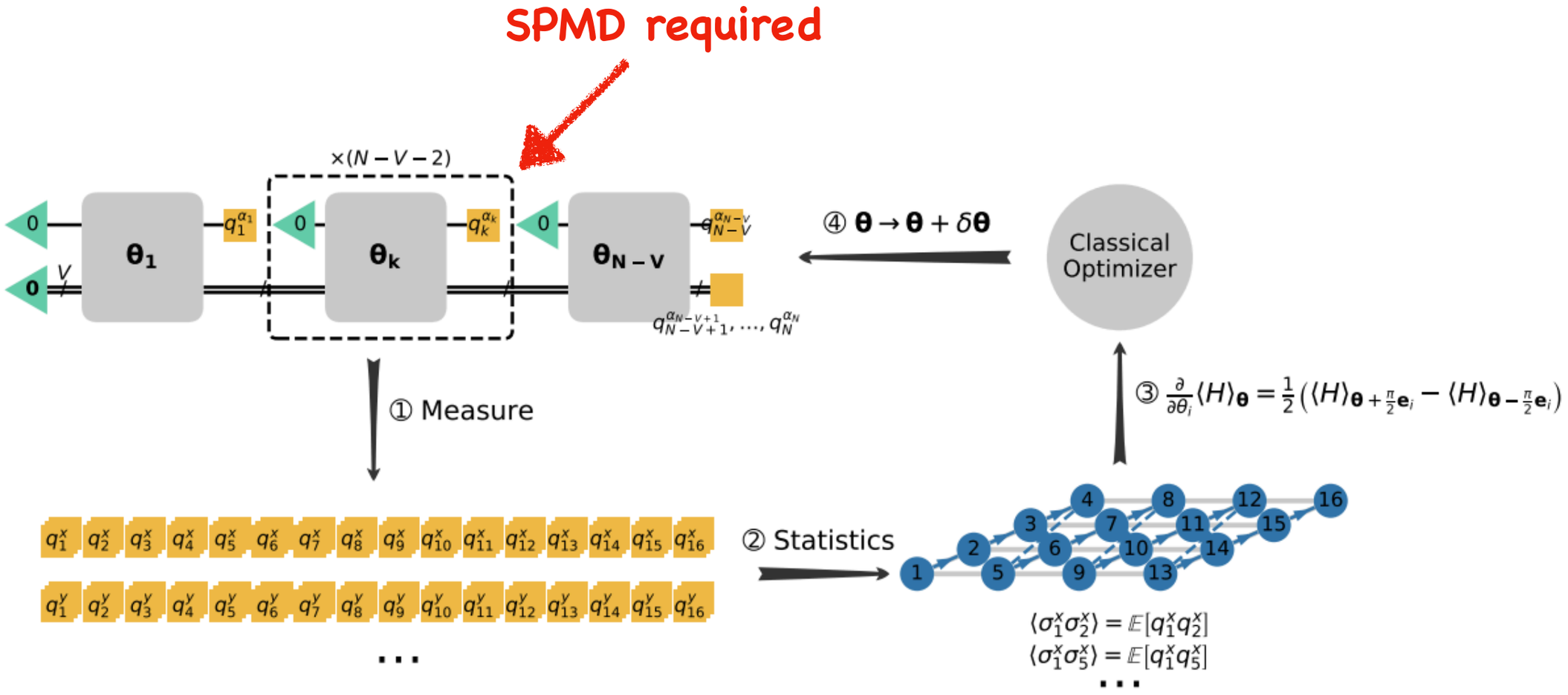






Liu, Jin-Guo, and Lei Wang. "Differentiable learning of quantum circuit born machines." *Physical Review A* 98.6 (2018): 062324.

SPMD required

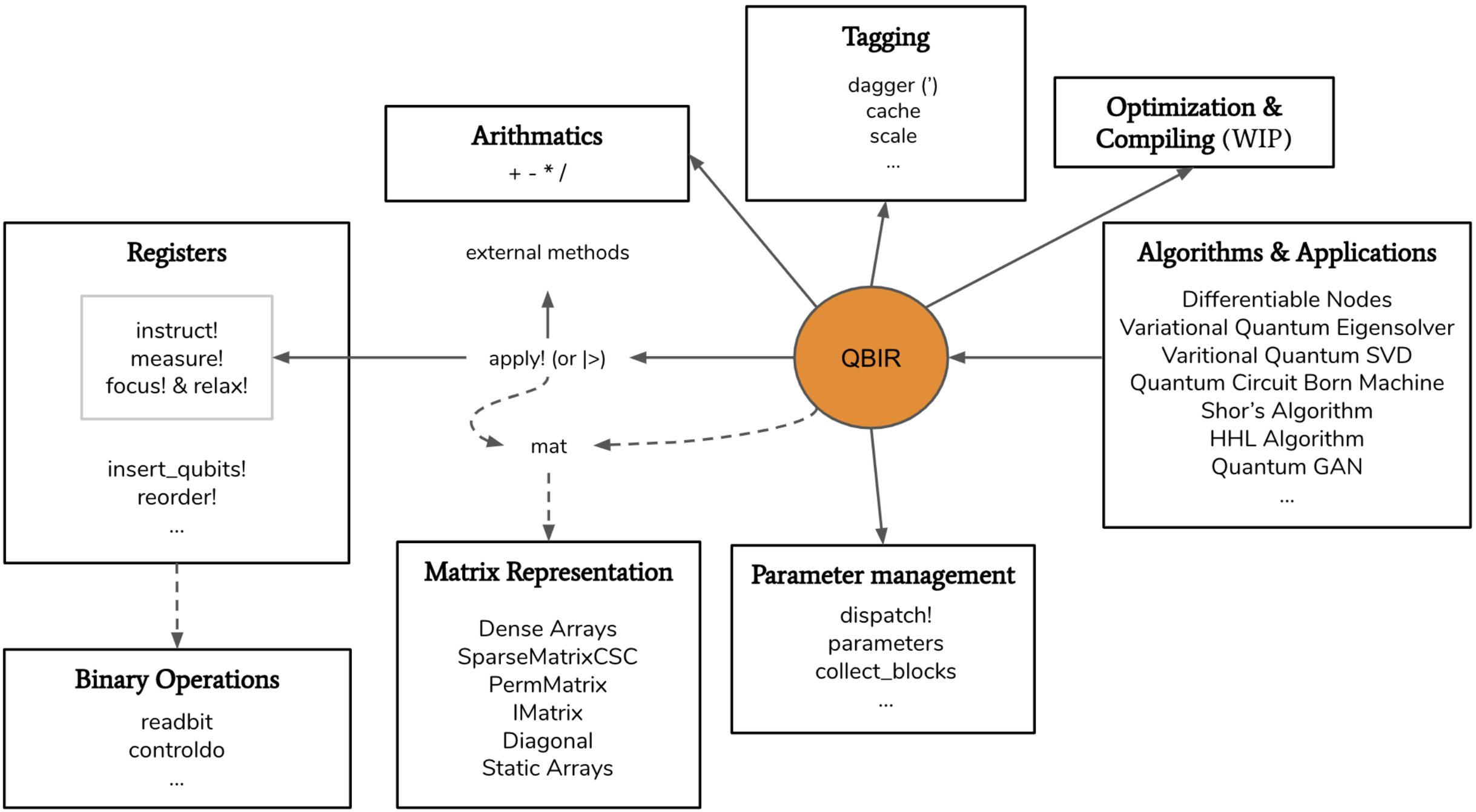


Liu, Jin-Guo, et al. "Variational Quantum Eigensolver with Fewer Qubits." *arXiv preprint arXiv:1902.02663* (2019).

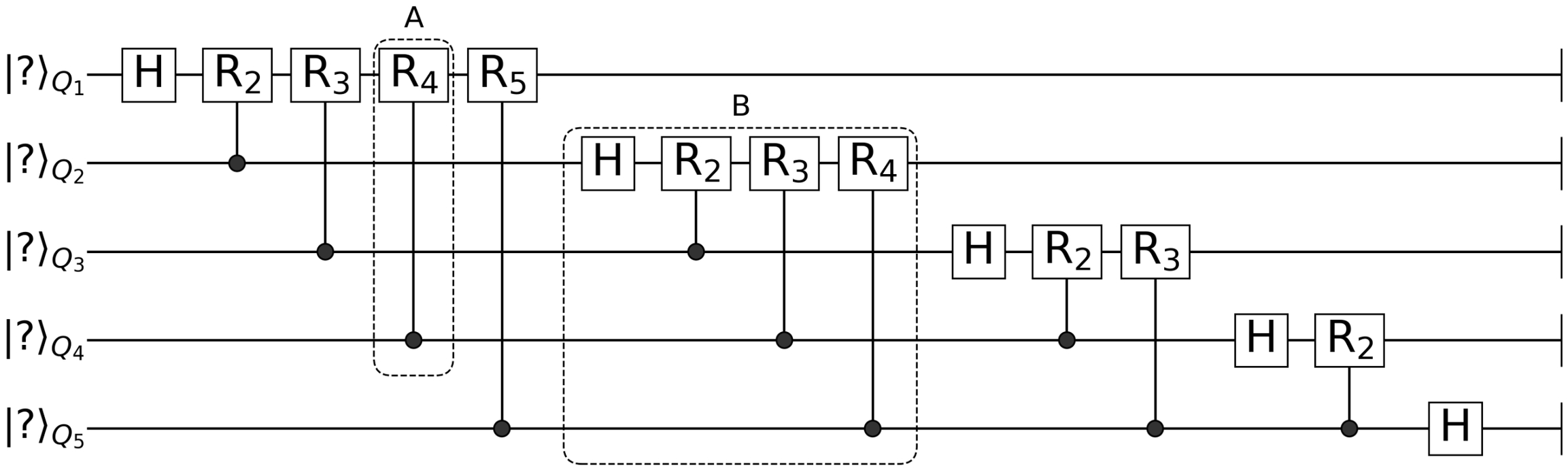
What do we need?

- Parameters Management
- Hardware Free Algorithm Representation
- Reasonable performance
- Single Program Multi Data (SPMD) support

Overview of Yao



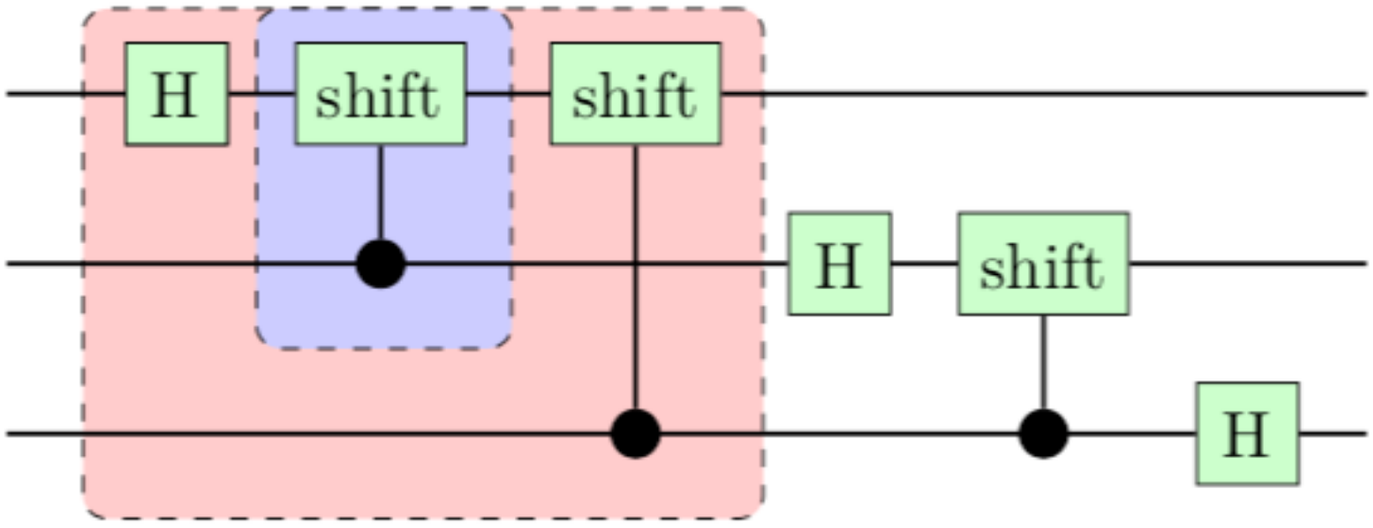
Quantum Fourier Transformation



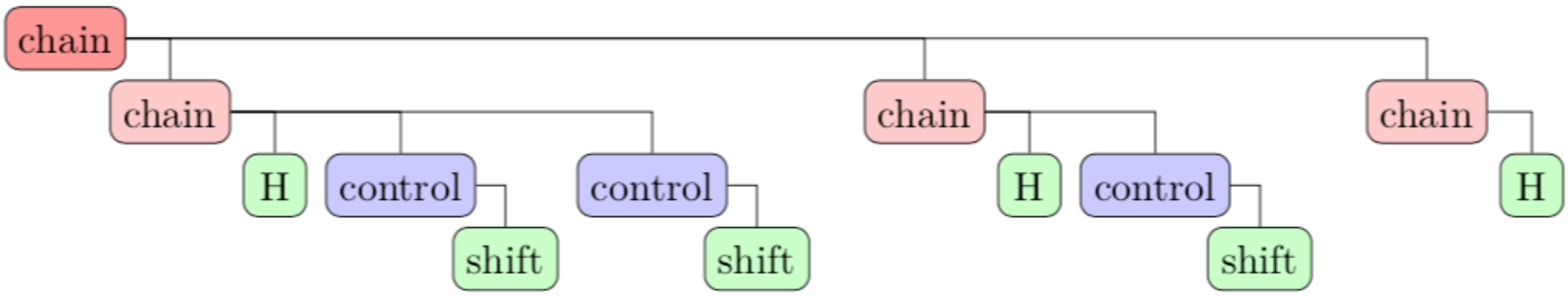
Quantum Fourier Transformation circuit of size 5

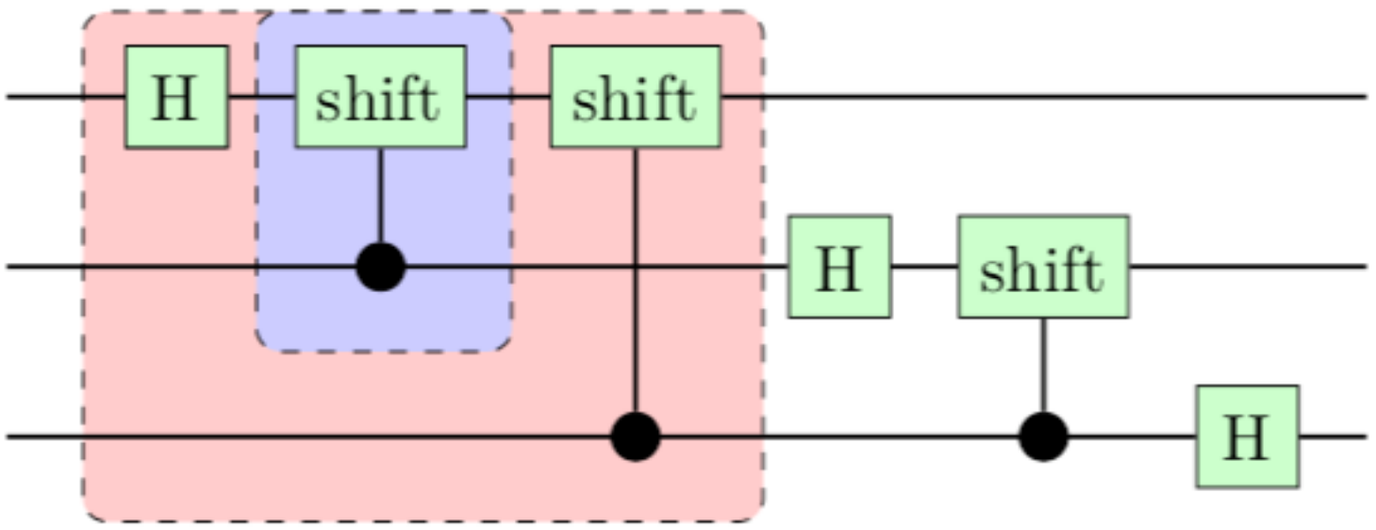
The Quantum Block IR

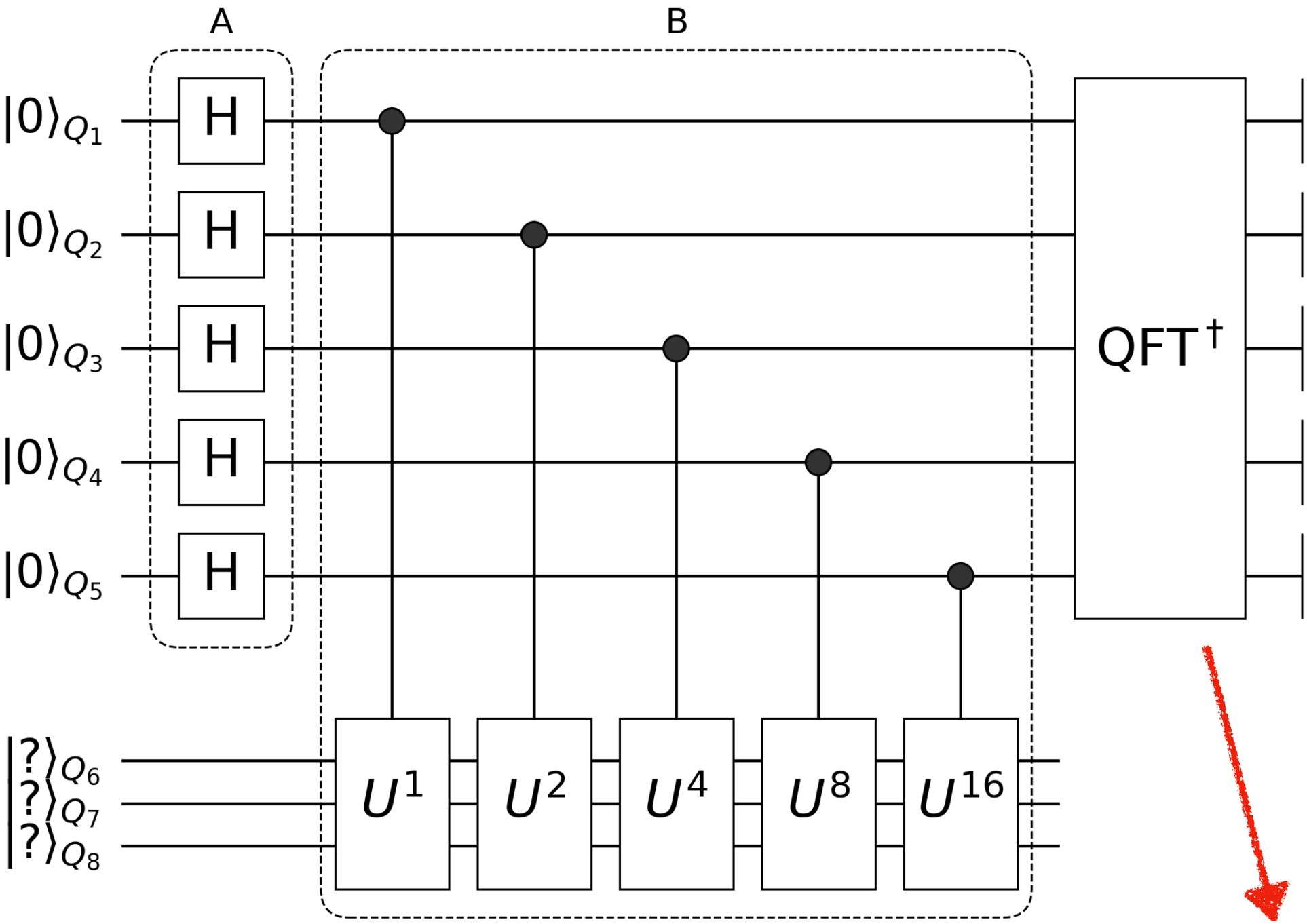
The Quantum Block IR



The Quantum Block IR



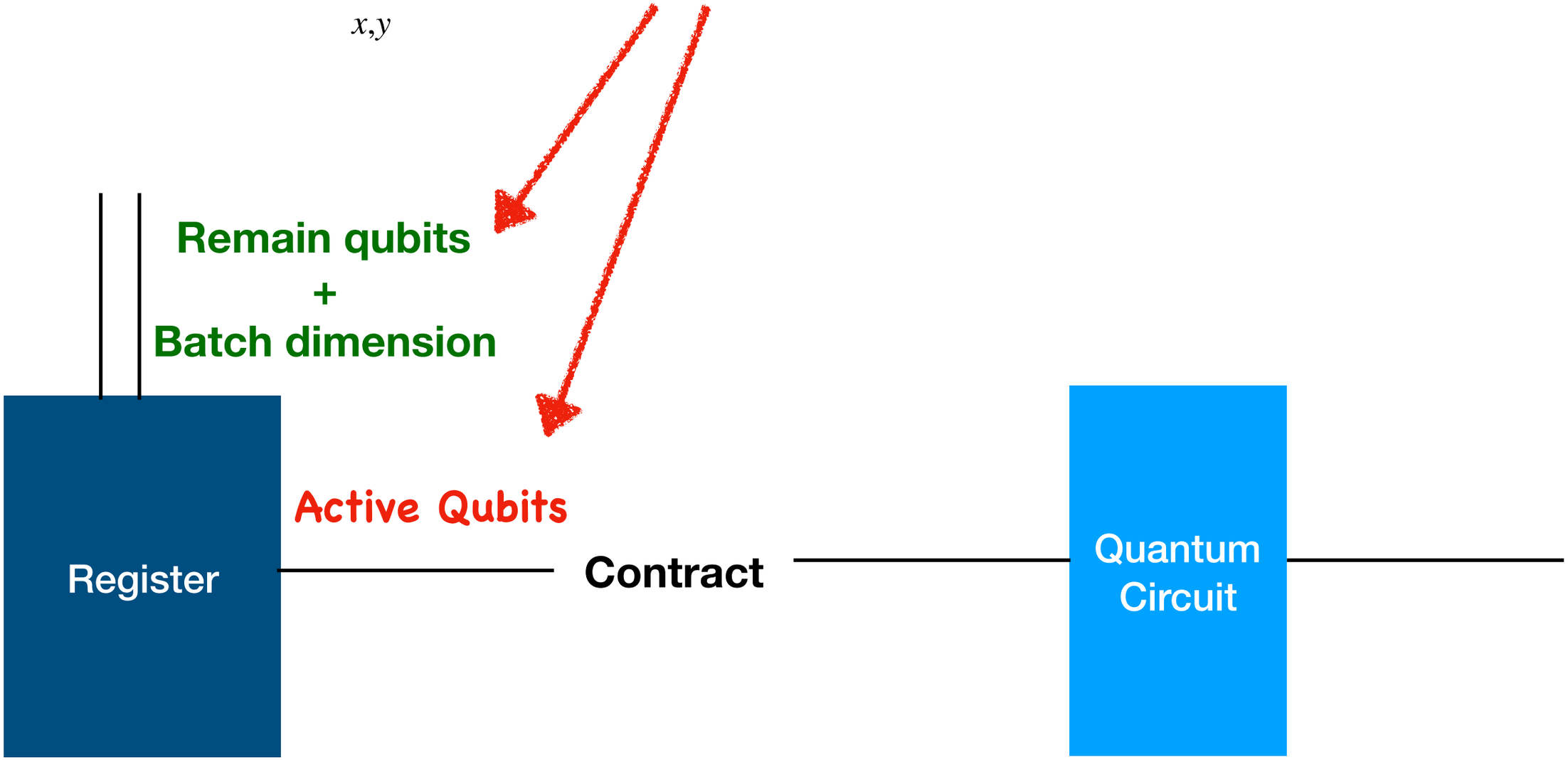




Quantum Circuit for Phase Estimation

Quantum Routine
On local scope

$$|\psi_k\rangle = \sum_{x,y} L(x, y, \dots)\,|j\rangle\,|i\rangle$$



VS

Aleksandrowicz, Gadi, et al. "Qiskit: An open-source framework for quantum computing." *Accessed on: Mar* 16 (2019).

VS

Aleksandrowicz, Gadi, et al. "Qiskit: An open-source framework for quantum computing." *Accessed on: Mar* 16 (2019).

VS

ProjectQ - An open source software framework for quantum computing

build passing coverage 100% docs failing pypi package 0.4.2
python 2.7, 3.4, 3.5, 3.6

ProjectQ is an open source effort for quantum computing.

It features a compilation framework capable of targeting various types of hardware, a high-performance quantum computer simulator with emulation capabilities, and various compiler plug-ins. This allows users to

- run quantum programs on the IBM Quantum Experience chip
- simulate quantum programs on classical computers
- emulate quantum programs at a higher level of abstraction (e.g., mimicking the action of large oracles instead of compiling them to low-level gates)
- export quantum programs as circuits (using TikZ)
- get resource estimates

ProjectQ

Aleksandrowicz, Gadi, et al. "Qiskit: An open-source framework for quantum computing." *Accessed on: Mar* 16 (2019).

VS

ProjectQ - An open source software framework for quantum computing

build passing coverage 100% docs failing pypi package 0.4.2
python 2.7, 3.4, 3.5, 3.6

ProjectQ is an open source effort for quantum computing.

It features a compilation framework capable of targeting various types of hardware, a high-performance quantum computer simulator with emulation capabilities, and various compiler plug-ins. This allows users to

- run quantum programs on the IBM Quantum Experience chip
- simulate quantum programs on classical computers
- emulate quantum programs at a higher level of abstraction (e.g., mimicking the action of large oracles instead of compiling them to low-level gates)
- export quantum programs as circuits (using TikZ)
- get resource estimates

ProjectQ

Aleksandrowicz, Gadi, et al. "Qiskit: An open-source framework for quantum computing." *Accessed on: Mar* 16 (2019).

VS

ProjectQ - An open source software framework for quantum computing

build passing coverage 100% docs failing pypi package 0.4.2
python 2.7, 3.4, 3.5, 3.6

ProjectQ is an open source effort for quantum computing.

It features a compilation framework capable of targeting various types of hardware, a high-performance quantum computer simulator with emulation capabilities, and various compiler plug-ins. This allows users to

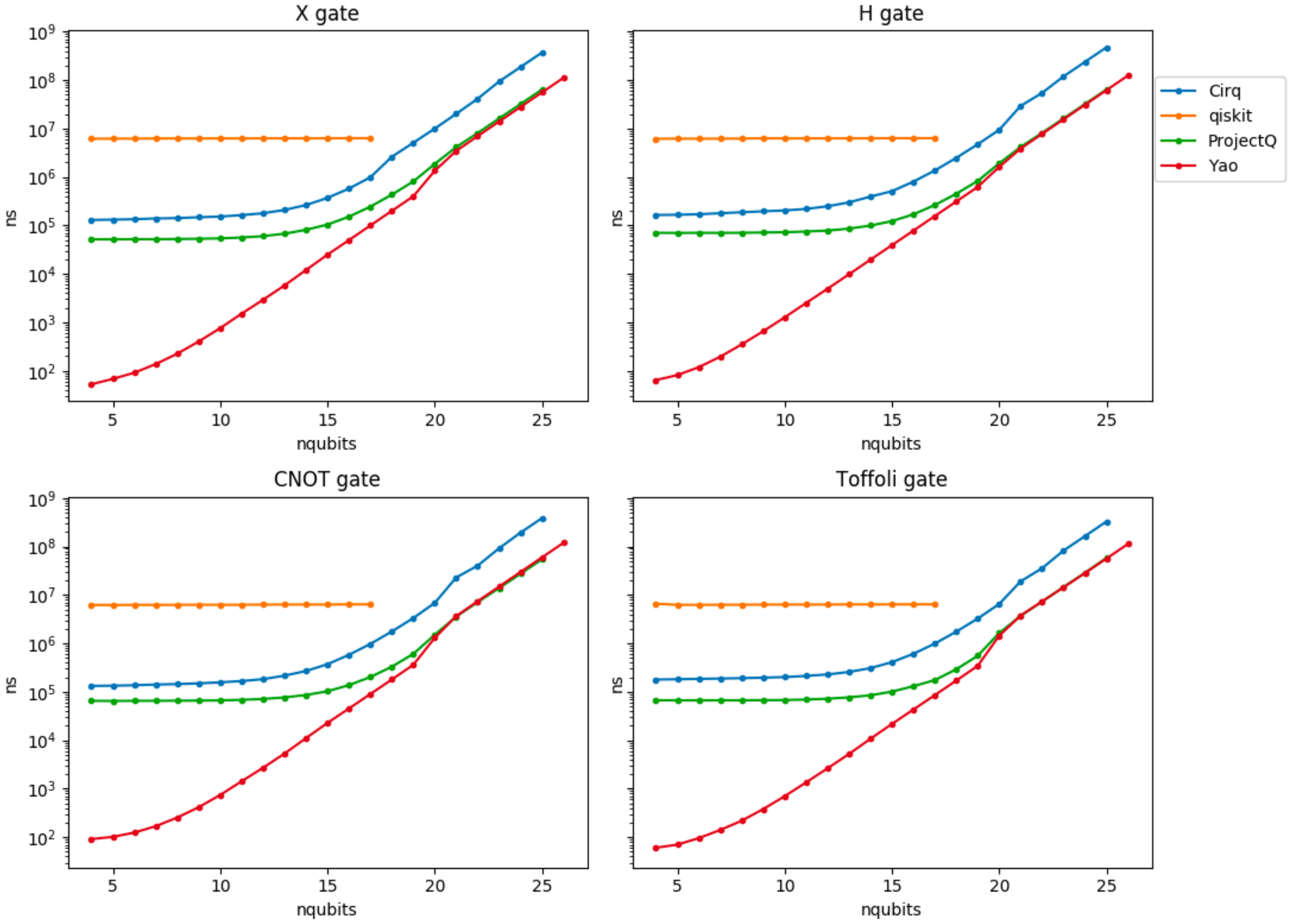
- run quantum programs on the IBM Quantum Experience chip
- simulate quantum programs on classical computers
- emulate quantum programs at a higher level of abstraction (e.g., mimicking the action of large oracles instead of compiling them to low-level gates)
- export quantum programs as circuits (using TikZ)
- get resource estimates

ProjectQ

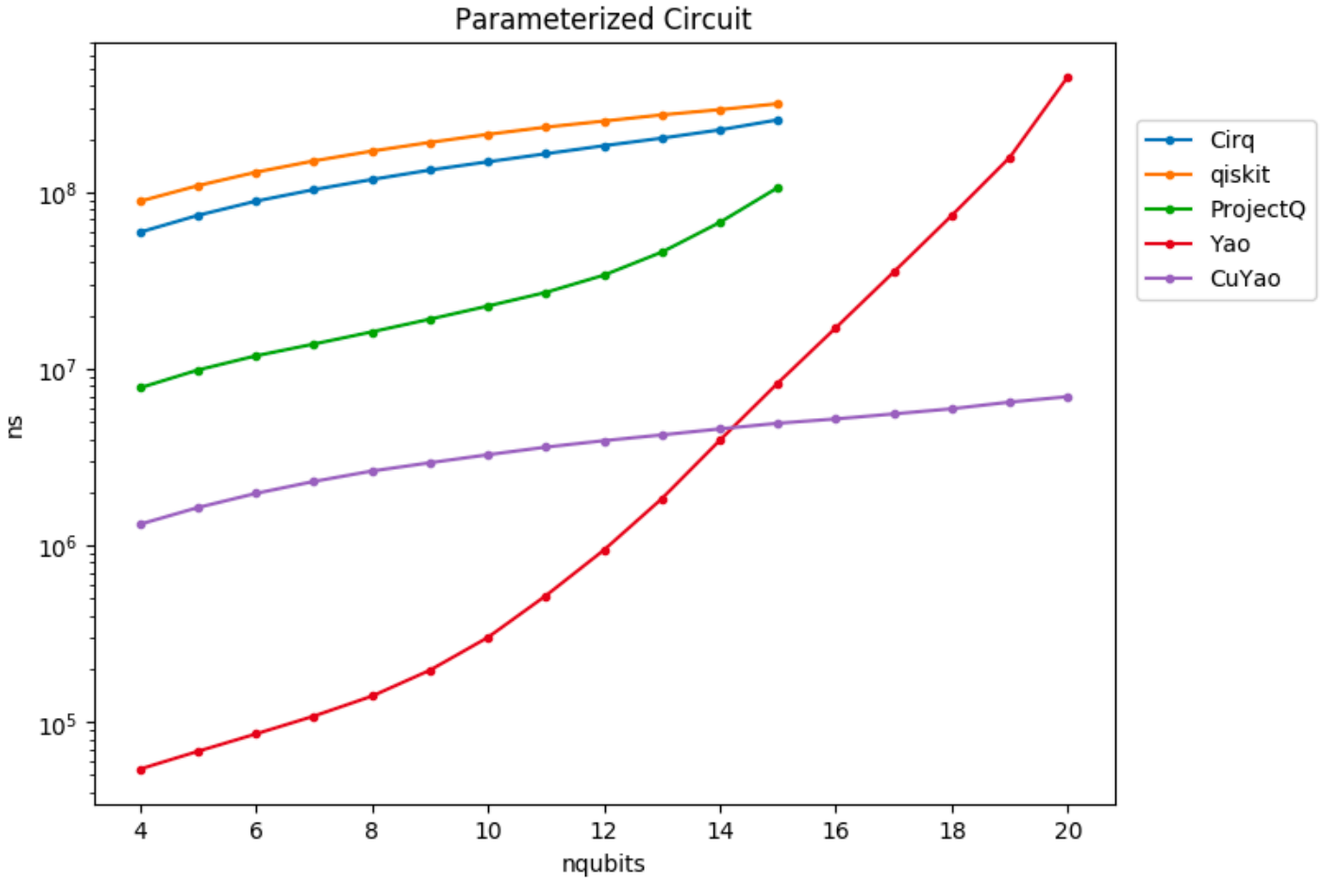
Cirq

Qiskit

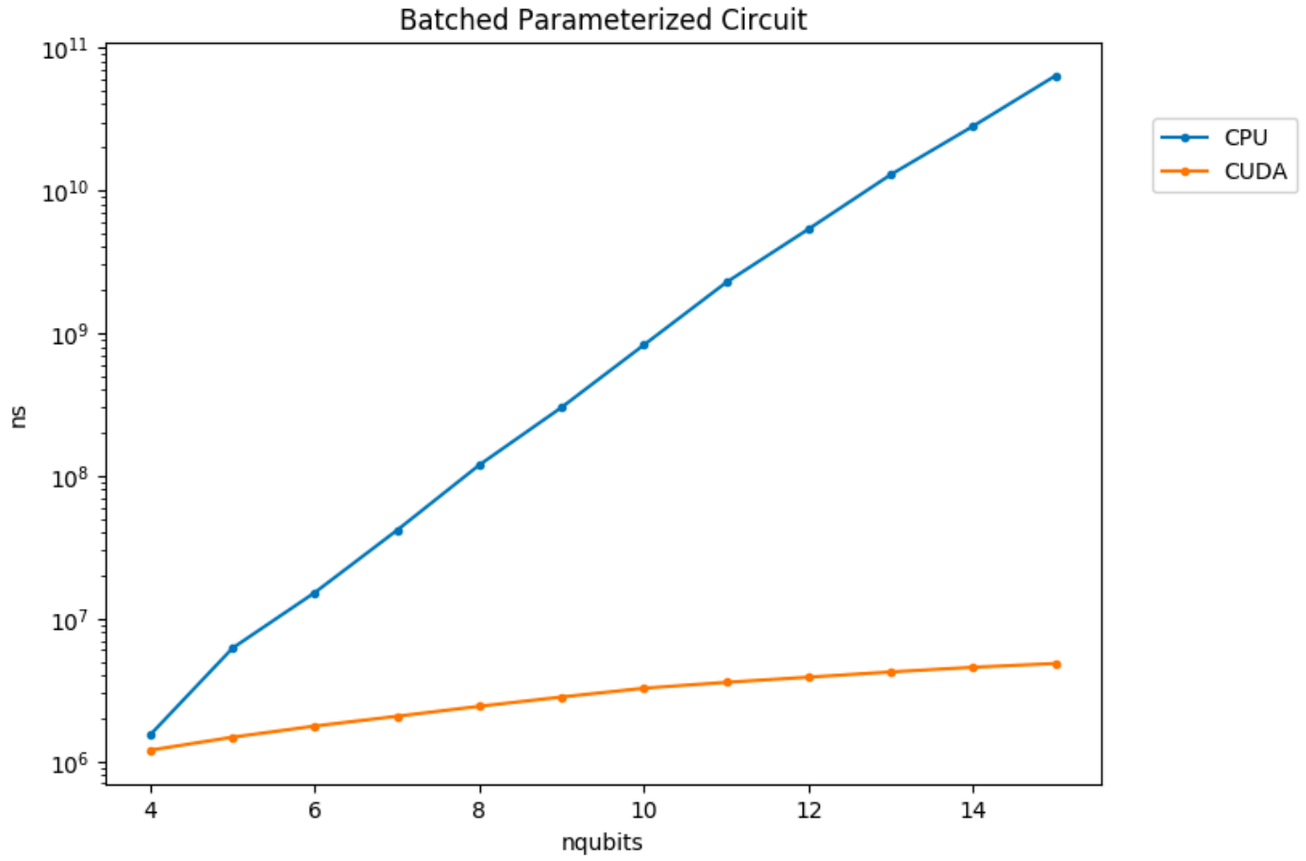
Aleksandrowicz, Gadi, et al. "Qiskit: An open-source framework for quantum computing." *Accessed on: Mar* 16 (2019).


X gate
H gate
CNOT gate
Toffoli gate
ns
nqubits
Cirq
qiskit
ProjectQ
Yao

Single Evaluation



Batched Evaluation (Yao only)

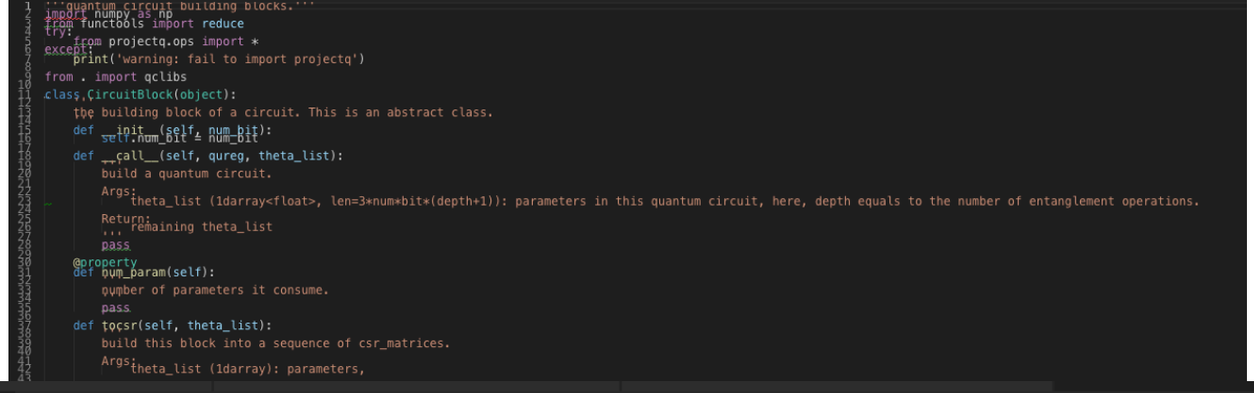


github: Roger-luo/quantum-benchmark

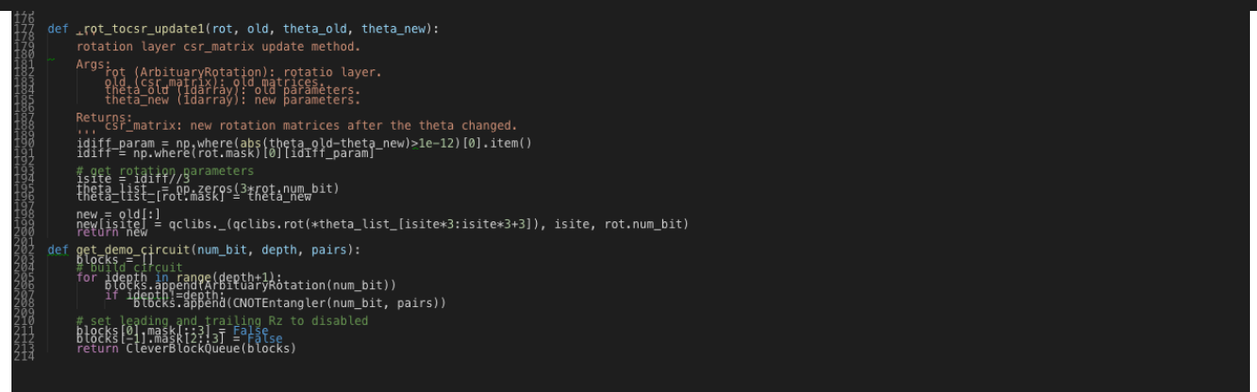
Github: GiggleLiu/QuantumCircuitBornMachine

```python
'''quantum circuit building blocks.'''
import numpy as np
from functools import reduce
try:
    from projectq.ops import *
except:
    print('warning: fail to import projectq')
from . import qclibs

class CircuitBlock(object):
    '''the building block of a circuit. This is an abstract class.'''
    def __init__(self, num_bit):
        self.num_bit = num_bit

    def __call__(self, qureg, theta_list):
        '''
        build a quantum circuit.

        Args:
            theta_list (1darray<float>, len=3*num*bit*(depth+1)): parameters in this quantum circuit, here, depth equals to the number of entanglement operations.

        Return:
            remaining theta_list
        '''
        pass

    @property
    def num_param(self):
        '''
        number of parameters it consume.
        '''
        pass

    def tocsr(self, theta_list):
        '''
        build this block into a sequence of csr_matrices.

        Args:
            theta_list (1darray): parameters,

        Returns:
            list: a list of csr_matrices, apply them on a vector to perform operation.
        '''
        pass

class BlockQueue(list):
    '''
    BlockQueue is a sequence of CircuitBlock instances.
    '''
    @property
    def num_bit(self):
        return self[0].num_bit

    @property
    def num_param(self):
        return sum([b.num_param for b in self])

    def __call__(self, qureg, theta_list):
        for block in self:
            theta_i, theta_list = np.split(theta_list, [block.num_param])
            block(qureg, theta_i)
        np.testing.assert_(len(theta_list)==0)

    def __str__(self):
        return '\n'.join([str(b) for b in self])

class CleverBlockQueue(BlockQueue):
    '''
    Clever Block Queue that keep track of theta_list changing history, for fast update.
    '''
    def __init__(self, *args):
        super().__init__(*args)
        self.theta_last = None
        self.memo = None

    def __call__(self, qureg, theta_list):
        if not isinstance(qureg, np.ndarray):
            return super(CleverBlockQueue, self).__call__(qureg, theta_list)
        # cache? if theta list change <= 1 parameters, then don't touch memory.
        remember = self.theta_last is None or (abs(self.theta_last-theta_list)>1e-12).sum() > 1

        mats = []
        theta_last = self.theta_last
        if remember:
            self.theta_last = theta_list.copy()

        qureg_ = qureg
        for iblock, block in enumerate(self):
            # generate or use a block matrix
            num_param = block.num_param
            theta_i, theta_list = np.split(theta_list, [num_param])
            if theta_last is not None:
                theta_o, theta_last = np.split(theta_last, [num_param])
            if self.memo is not None and (num_param==0 or np.abs(theta_i-theta_o).max()<1e-12):
                # use data cached in memory
                mat = self.memo[iblock]
            else:
                if self.memo is not None and not remember:
                    # update the changed gate, but not touching memory.
                    mat = _rot_tocsr_update1(block, self.memo[iblock], theta_o, theta_i)
                else:
                    # regenerate one
                    mat = block.tocsr(theta_i)
                for mat_i in mat:
                    qureg_ = mat_i.dot(qureg_)
                mats.append(mat)

        if remember:
            # cache data
            self.memo = mats
        # update register
        qureg[...] = qureg_
        np.testing.assert_(len(theta_list)==0)


class ArbituaryRotation(CircuitBlock):
    def __init__(self, num_bit):
        super(ArbituaryRotation, self).__init__(num_bit)
        self.mask = np.array([True] * (3*num_bit), dtype='bool')

    def __call__(self, qureg, theta_list):
        gates = [Rz, Rx, Rz]
        theta_list_ = np.zeros(self.num_bit*3)
        theta_list_[self.mask] = theta_list
        for i, (theta, mask) in enumerate(zip(theta_list_, self.mask)):
            ibit, igate = i//3, i%3
            if mask:
                gate = gates[igate](theta)
                gate | qureg[ibit]

    def __str__(self):
        return 'Rotate[%d]'%(self.num_param)

    @property
    def num_param(self):
        return self.mask.sum()

    def tocsr(self, theta_list):
        '''transform this block to csr_matrix.'''
        theta_list_ = np.zeros(3*self.num_bit)
        theta_list_[self.mask] = theta_list
        rots = [qclibs.rot(*ths) for ths in theta_list_.reshape([self.num_bit,3])]
        res = [qclibs._(rot, [i], self.num_bit) for i,rot in enumerate(rots)]
        return res

class CNOTEntangler(CircuitBlock):
    def __init__(self, num_bit, pairs):
        super(CNOTEntangler, self).__init__(num_bit)
        self.pairs = pairs

    def __str__(self):
        pair_str = ','.join(['%d-%d'%(i,j) for i,j in self.pairs])
        return 'CNOT[%s]'%(pair_str)

    def __call__(self, qureg, *args, **kwargs):
        for pair in self.pairs:
            CNOT | (qureg[pair[0]], qureg[pair[1]])

    @property
    def num_param(self):
        return 0

    def tocsr(self, theta_list):
        '''transform this block to csr_matrix.'''
        i, j = self.pairs[0]
        res = qclibs.CNOT(i, j, self.num_bit)
        for i,j in self.pairs[1:]:
            res = qclibs.CNOT(i,j,self.num_bit).dot(res)
        res.eliminate_zeros()
        return [res]


def _rot_tocsr_update1(rot, old, theta_old, theta_new):
    '''
    rotation layer csr_matrix update method.

    Args:
        rot (ArbituaryRotation): rotatio layer.
        old (csr_matrix): old matrices.
        theta_old (1darray): old parameters.
        theta_new (1darray): new parameters.

    Returns:
        csr_matrix: new rotation matrices after the theta changed.
    '''
    idiff_param = np.where(abs(theta_old-theta_new)>1e-12)[0].item()
    idiff = np.where(rot.mask)[0][idiff_param]

    # get rotation parameters
    isite = idiff//3
    theta_list_ = np.zeros(3*rot.num_bit)
    theta_list_[rot.mask] = theta_new

    new = old[:]
    new[isite] = qclibs._(qclibs.rot(*theta_list_[isite*3:isite*3+3]), isite, rot.num_bit)
    return new

def get_demo_circuit(num_bit, depth, pairs):
    blocks = []
    # build circuit
    for idepth in range(depth+1):
        blocks.append(ArbituaryRotation(num_bit))
        if idepth!=depth:
            blocks.append(CNOTEntangler(num_bit, pairs))

    # set leading and trailing Rz to disabled
    blocks[0].mask[:3] = False
    blocks[-1].mask[2::3] = False
    return CleverBlockQueue(blocks)
```

Github: GiggleLiu/QuantumCircuitBornMachine



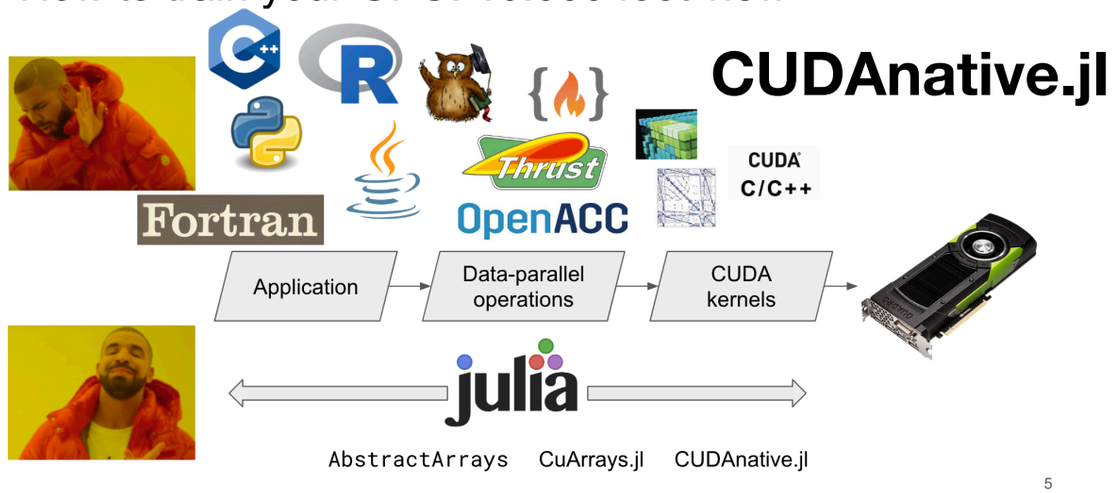

```julia
using Yao, YaoBlocks

layer(nbit::Int, ::Val{:last}) = chain(nbit, put(i=>chain(Rz(0), Rx(0))) for i = 1:nbit)
layer(nbit::Int, ::Val{:mid}) = chain(nbit, put(i=>chain(Rz(0), Rx(0), Rz(0))) for i = 1:nbit);
entangler(pairs) = chain(control(ctrl, target=>X) for (ctrl, target) in pairs);

function build_circuit(n, nlayers, pairs)
    circuit = chain(n)
    push!(circuit, layer(n, :first))
    for i in 2:nlayers
        push!(circuit, cache(entangler(pairs)))
        push!(circuit, layer(n, :mid))
    end
    push!(circuit, cache(entangler(pairs)))
    push!(circuit, layer(n, :last))
    return circuit
end

build_circuit(4, 1, [1=>2, 2=>3, 3=>4])
```



Github: GiggleLiu/QuantumCircuitBornMachine

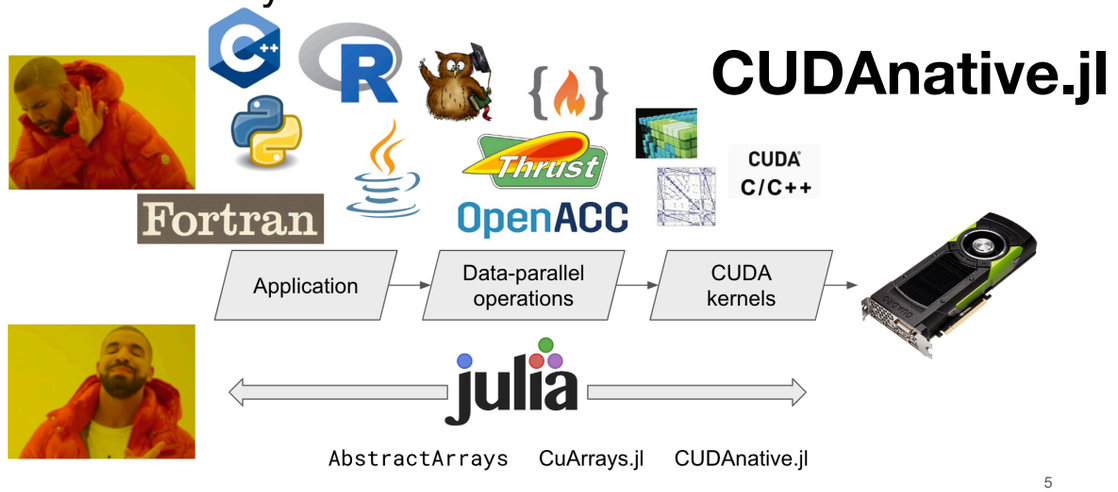
```
function instruct!(state::CuVecOrMat, ::Val{:SWAP}, locs::Tuple{Int,Int})
    b1, b2 = locs
    mask1 = bmask(b1)
    mask2 = bmask(b2)

    X, Y = cudiv(size(state)...)
    function kf(state, mask1, mask2)
        inds = ((blockIdx().x-1) * blockDim().x + threadIdx().x,
                (blockIdx().y-1) * blockDim().y + threadIdx().y)
        b = inds[1]-1
        c = inds[2]
        c <= size(state, 2) || return nothing
        if b&mask1==0 && b&mask2==mask2
            i = b+1
            i_ = b ⊻ (mask1|mask2) + 1
            temp = state[i, c]
            state[i, c] = state[i_, c]
            state[i_, c] = temp
        end
        nothing
    end
    @cuda threads=X blocks=Y kf(state, mask1, mask2)
    state
end
```

How to train your GPU: 10.000 foot view



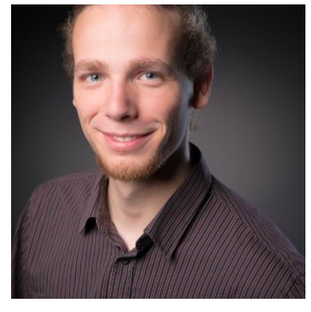

```
function instruct!(state::CuVecOrMat, :
    b1, b2 = locs
    mask1 = bmask(b1)
    mask2 = bmask(b2)

    X, Y = cudiv(size(state)...)
    function kf(state, mask1, mask2)
        inds = ((blockIdx().x-1) * blockDim().x + threadIdx().x,
                (blockIdx().y-1) * blockDim().y + threadIdx().y)
        b = inds[1]-1
        c = inds[2]
        c <= size(state, 2) || return nothing
        if b&mask1==0 && b&mask2==mask2
            i = b+1
            i_ = b ⩲ (mask1|mask2) + 1
            temp = state[i, c]
            state[i, c] = state[i_, c]
            state[i_, c] = temp
        end
        nothing
    end
    @cuda threads=X blocks=Y kf(state, mask1, mask2)
    state
end
```

How to train your GPU: 10.000 foot view




```
function instruct!(state::CuVecOrMat, :
    b1, b2 = locs
    mask1 = bmask(b1)
    mask2 = bmask(b2)

    X, Y = cudiv(size(state)...)
    function kf(state, mask1, mask2)
        inds = ((blockIdx().x-1) * blockDim().x + threadIdx().x,
                (blockIdx().y-1) * blockDim().y + threadIdx().y)
        b = inds[1]-1
        c = inds[2]
        c <= size(state, 2) || return nothing
        if b&mask1==0 && b&mask2==mask2
            i = b+1
            i_ = b ⩛ (mask1|mask2) + 1
            temp = state[i, c]
            state[i, c] = state[i_, c]
            state[i_, c] = temp
        end
        nothing
    end
    @cuda threads=X blocks=Y kf(state, mask1, mask2)
    state
end
```


Tim Besard



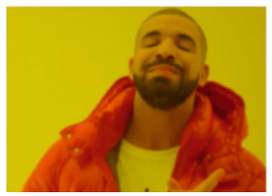
Valentin Churavy

- Parameters Management
- Hardware Free Algorithm Representation
- Reasonable performance
- Single Program Multi Data (SPMD) support

- Parameters Management **dispatch!/parameters**
- Hardware Free Algorithm Representation
- Reasonable performance
- Single Program Multi Data (SPMD) support

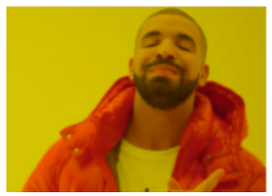
- Parameters Management **dispatch!/parameters**
- Hardware Free Algorithm Representation **QBIR**
- Reasonable performance
- Single Program Multi Data (SPMD) support

- Parameters Management **dispatch!/parameters**
- Hardware Free Algorithm Representation **QBIR**
- Reasonable performance



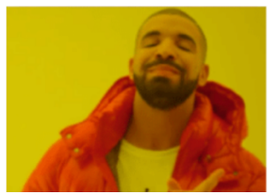
- Single Program Multi Data (SPMD) support

- Parameters Management **dispatch!/parameters**
- Hardware Free Algorithm Representation **QBIR**
- Reasonable performance



- Single Program Multi Data (SPMD) support **batched register**

- Parameters Management **dispatch!/parameters**
- Hardware Free Algorithm Representation **QBIR**
- Reasonable performance



- Single Program Multi Data (SPMD) support **batched register**

Yao.jl: Extensible, Efficient Framework for Quantum Algorithm Design

Xiu-Zhe Luo[1,2], Jin-Guo Liu[3], Pan Zhang[4], and Lei Wang[3,5,6]

Why Julia?

- LuxurySparse.jl: Easy to implement custom matrices type
- CuYao.jl: Easy to write CUDA code while reusing generic functions
- YaoArrayRegister.jl: multiple dispatch -> specialized instructions
- Active community
- Complete benchmarking, testing, documentation tool chain

On going projects

Full AD support on Quantum Circuit

```
using Yao, Zygote, Flux.Optimise
# make a one qubit GHZ state as learning target
# NOTE: this can be an arbitrary one qubit state
t = ArrayReg(bit"0") + ArrayReg(bit"1")
normalize!(t)

# calculate the fidelity with GHZ state
function fid(xs)
    r = zero_state(1)
    U = mat(chain(Rx(xs[1]), Rz(xs[2])))
    return abs(statevec(t)' * U * statevec(r))
end

# simply tell Zygote to get the gradient and start training
function train()
    opt = ADAM()
    xs = rand(2)
    for _ in 1:1000
        println(fid(xs))
        Optimise.update!(opt, xs, -fid'(xs))
    end
    return xs
end

train()
```

QASM & Quil Compilation

thautwarm / RBNF.jl

Unwatch 4 | Unstar 7 | Fork 0

Code | Issues 0 | Pull requests 0 | Projects 0 | Wiki | Security | Insights

A DSL for modern parsing

26 commits | 1 branch | 0 releases | 1 contributor | BSD-2-Clause

QuantumBFS / YaoQASM.jl

Unwatch 7 | Star 1 | Fork 1

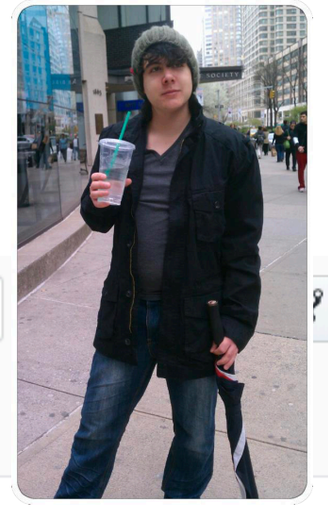
Code | Issues 1 | Pull requests 0 | Projects 0 | Wiki | Security | Insights | Settings

Bidirectional transformation between Yao IR and QASM.

Edit

qasm | yao-ir | quantum-computing | parsing | Manage topics

JSoC 2019: Quantum ODE solver for DiffEq.jl



QuantumBFS / QuDiffEq.jl

Unwatch 3 | Fork 0

Code | Issues 0 | Pull requests 0 | Projects 0 | Wiki | Security | Insights | Settings

Quantum Algorithms for solving differential equations

Christopher Rackauckas Edit

Manage topics

3 commits | 1 branch | 0 releases | 1 contributor | MIT

Branch: master | New pull request | Create new file | Upload files | Find File | Clone or download

dgan181 readme — Latest commit 0d5e843 2 days ago

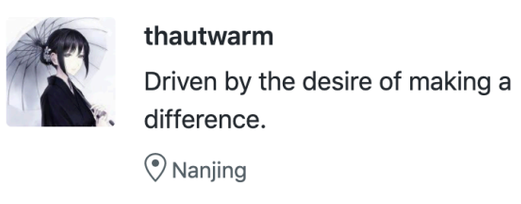
Roadmap

- Quantum circuit architecture optimization （Quon）
- Tensor Network support
- Experiment validation
- Circuit Plotting & visualization

The Yao Team

Creators

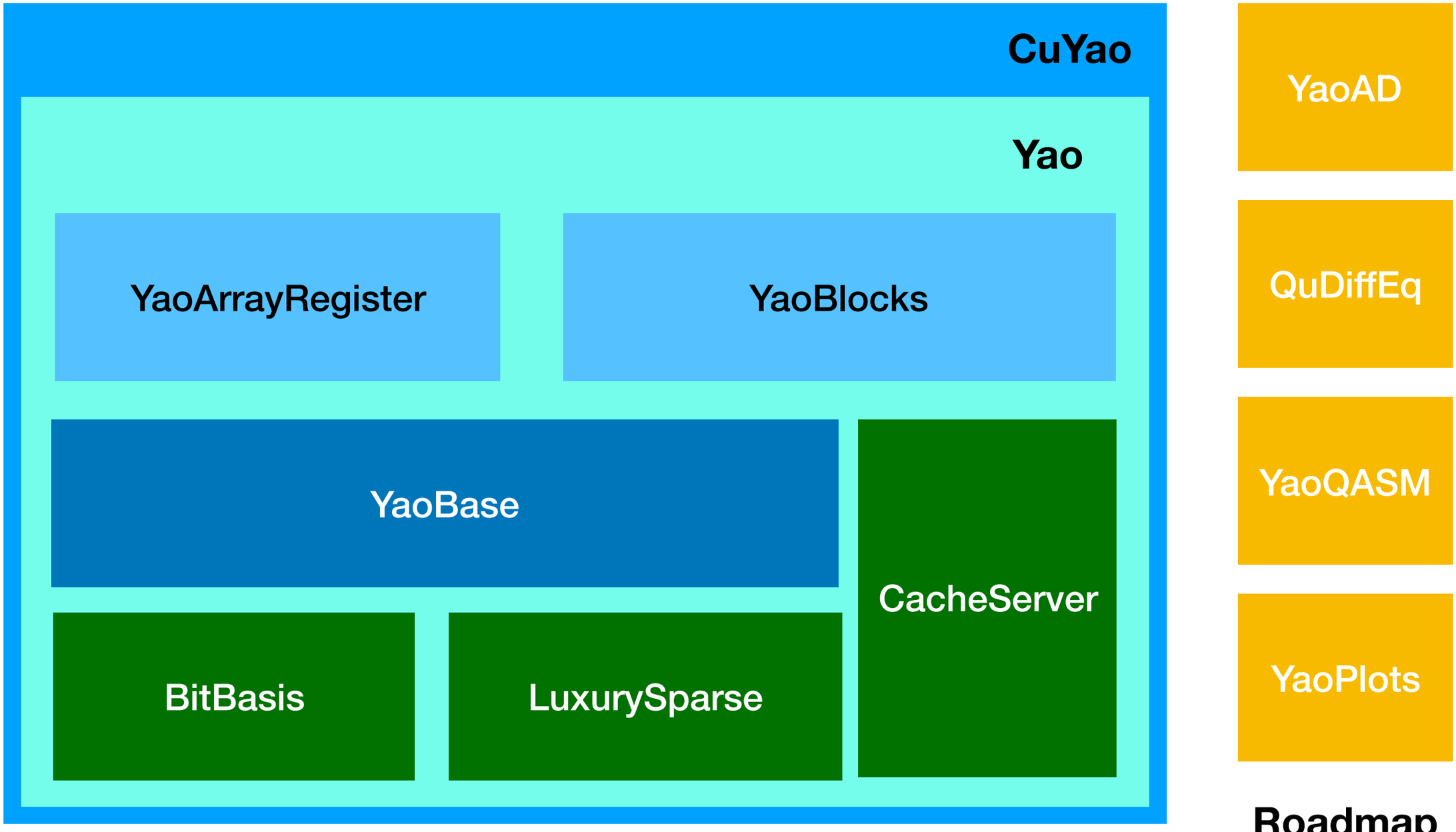
Members



Acknowledge

Special thanks to Mrs. Jin Zhu for our logo design

Ecosystem


CuYao
Yao
YaoArrayRegister
YaoBlocks
YaoBase
CacheServer
BitBasis
LuxurySparse
YaoAD
QuDiffEq
YaoQASM
YaoPlots
Roadmap

Quantum Computing 101

Classical Logic Bits

Value	Bit Index

Value	Bit Index

Classical Logic Bits

Classical Logic Bits

$$0 \rightarrow \begin{pmatrix} 1 \\ 0 \end{pmatrix} \begin{matrix} \textbf{Value} \\ 0 \\ 1 \end{matrix} \quad \begin{matrix} \textbf{Bit Index} \\ 0 \\ 1 \end{matrix}$$

$$1 \rightarrow \begin{pmatrix} 0 \\ 1 \end{pmatrix} \begin{matrix} \textbf{Value} \\ 0 \\ 1 \end{matrix} \quad \begin{matrix} \textbf{Bit Index} \\ 0 \\ 1 \end{matrix}$$

$$00 \rightarrow \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \end{pmatrix} \begin{matrix} 00 \\ 01 \\ 10 \\ 11 \end{matrix}$$

$$01 \rightarrow \begin{pmatrix} 0 \\ 1 \\ 0 \\ 0 \end{pmatrix} \begin{matrix} 00 \\ 01 \\ 10 \\ 11 \end{matrix}$$

Classical Operations (Gates)

$$\mathbf{X} = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \quad X \cdot 0 \rightarrow \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} 1 \\ 0 \end{pmatrix} = \begin{pmatrix} 0 \\ 1 \end{pmatrix}$$

Quantum

Quantum

$$\frac{1}{\sqrt{2}}\begin{pmatrix}1\\ im\end{pmatrix}\begin{matrix}|0\rangle\\ |1\rangle\end{matrix} \rightarrow \frac{1}{\sqrt{2}}(|0\rangle + im\,|1\rangle)$$

Quantum

$$\frac{1}{\sqrt{2}}\begin{pmatrix}1\\ im\end{pmatrix}\begin{matrix}|0\rangle\\ |1\rangle\end{matrix} \rightarrow \frac{1}{\sqrt{2}}(|0\rangle + im\,|1\rangle)$$

$$\mathbf{X} = \begin{pmatrix}0 & 1\\ 1 & 0\end{pmatrix}$$

Quantum

$$\frac{1}{\sqrt{2}}\begin{pmatrix}1\\ im\end{pmatrix}\begin{matrix}|0\rangle\\ |1\rangle\end{matrix} \rightarrow \frac{1}{\sqrt{2}}(|0\rangle + im\,|1\rangle)$$

$$\mathbf{X} = \begin{pmatrix}0 & 1\\ 1 & 0\end{pmatrix} \qquad \mathbf{Y} = \begin{pmatrix}0 & -im\\ im & 0\end{pmatrix}$$

Quantum

$$\frac{1}{\sqrt{2}}\begin{pmatrix}1\\ im\end{pmatrix}\begin{matrix}|0\rangle\\ |1\rangle\end{matrix} \rightarrow \frac{1}{\sqrt{2}}(|0\rangle + im\,|1\rangle)$$

$$\mathbf{X} = \begin{pmatrix}0 & 1\\ 1 & 0\end{pmatrix} \qquad \mathbf{Y} = \begin{pmatrix}0 & -im\\ im & 0\end{pmatrix} \qquad \mathbf{Z} = \begin{pmatrix}1 & 0\\ 0 & -1\end{pmatrix}$$

Quantum

$$\frac{1}{\sqrt{2}}\begin{pmatrix}1\\ im\end{pmatrix}\begin{matrix}|0\rangle\\ |1\rangle\end{matrix} \rightarrow \frac{1}{\sqrt{2}}(|0\rangle + im\,|1\rangle)$$

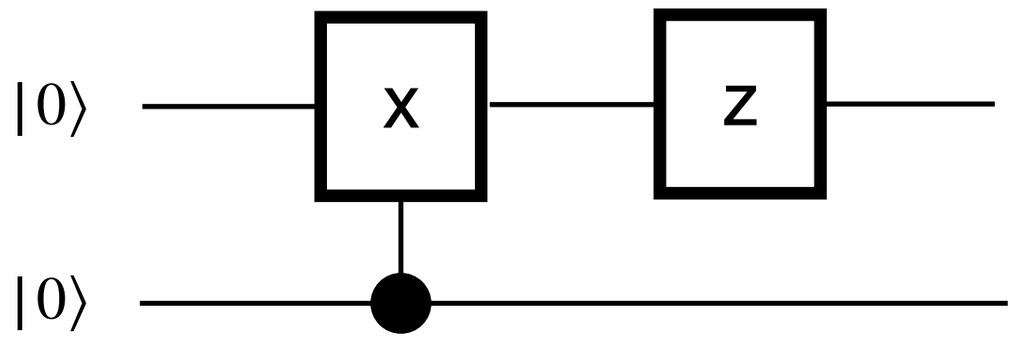
$$\mathbf{X} = \begin{pmatrix}0 & 1\\ 1 & 0\end{pmatrix} \qquad \mathbf{Y} = \begin{pmatrix}0 & -im\\ im & 0\end{pmatrix} \qquad \mathbf{Z} = \begin{pmatrix}1 & 0\\ 0 & -1\end{pmatrix}$$

$$\frac{1}{\sqrt{2}}\begin{pmatrix}1\\ im\end{pmatrix}\begin{matrix}\langle 0|\\ \langle 1|\end{matrix} \rightarrow \frac{1}{\sqrt{2}}(\langle 0| + im\langle 1|)$$

Quantum Circuit

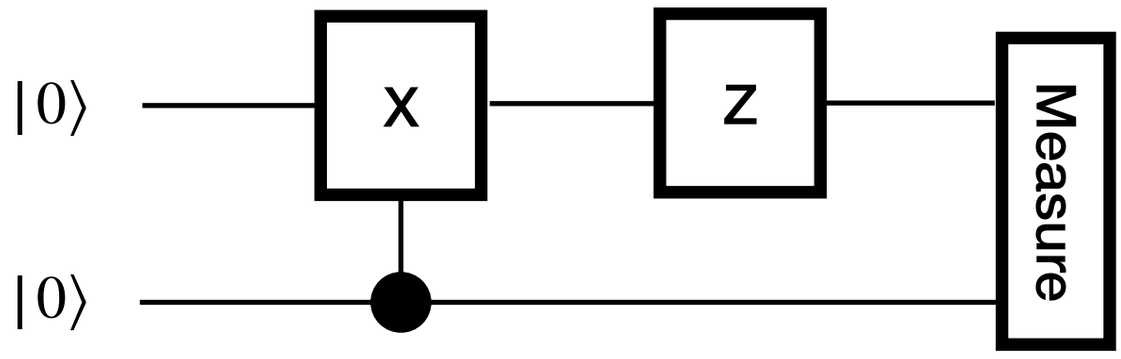
- **Probability based**
- **Unitary**
- **Normalized**

Quantum Circuit



- **Probability based**
- **Unitary**
- **Normalized**

Quantum Circuit



- **Probability based**
- **Unitary**
- **Normalized**

Traditional quantum algorithms

- Define a goal
- Code every step
- Requires error correction
- Achieve goal with high probability

NISQ

- Finite number of qubits
- Finite number of gates
- Low fidelity gates

QuAlgorithmZoo

A curated implementation of quantum algorithms with Yao.jl

Installation

QuAlgorithmZoo.jl is not registered yet, please use the following command:

```
pkg> add https://github.com/QuantumBFS/QuAlgorithmZoo.jl.git
```

Disclaimer: **this package is still under development and needs further polish.**

Contents

- [x] QCBM tutorial
- [x] grover search
- [x] HHL
- [x] QFT
- [x] QuGAN
- [x] QCBM
- [x] Hamiltonian Solver
- [x] QAOA
- [x] Quantum Chemistry
- [x] QuODE