ACCELERATING QUANTUM ALGORITHM RESEARCH WITH CUQUANTUM

HPCQC 2021

ALEX MCCASKEY, QUANTUM COMPUTING SOFTWARE ARCHITECT, NVIDIA

DECEMBER 16, 2021

AGENDA

The need for circuit simulation

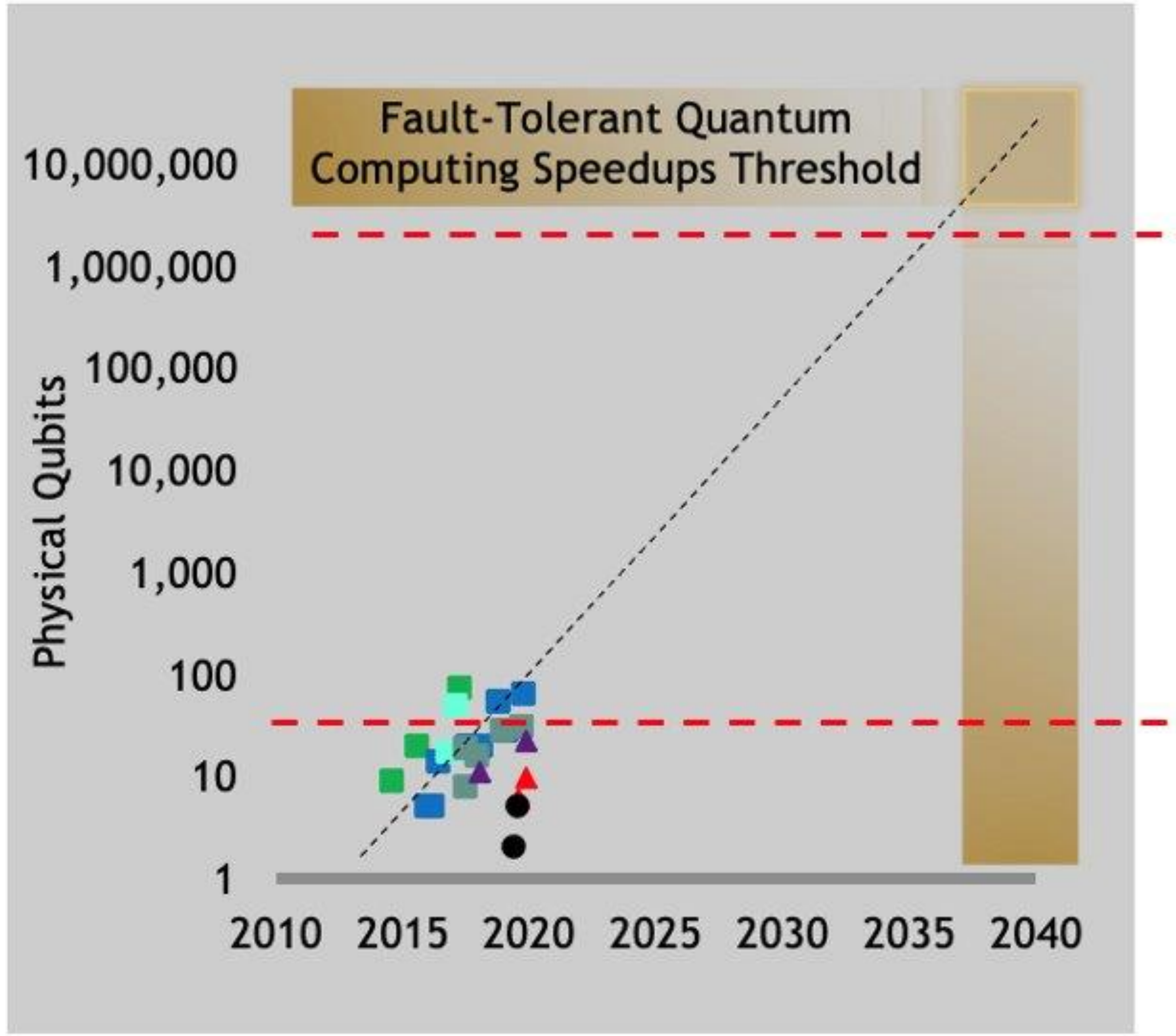
cuQuantum – an overview and performance benchmarks

- cuStateVec and cuTensorNet
- Latest QAOA Maxcut results

Early look at programming cuQuantum for algorithmic research

QC RESEARCH ROADMAP

Large improvements in qubit quantity & quality, error correction, needed for wide adoption



Fault-Tolerant QC Era:

1000:1-10000:1 redundancy for error-corrected *logical* qubits.
[Fowler 2012][Reiher 2016]

Exponential speedups on a limited set of applications with hundreds to thousands of logical qubits (millions of physical qubits).

Active Research: What are the best error correction algorithms?

Noisy Intermediate Scale Quantum (NISQ) Era:

Quantum gates are noisy, errors accumulate. Qubits lose coherence.

QC hardware will mitigate errors by using tens to hundreds of redundant physical qubits per logical qubit to mitigate errors.

Active Research: Will NISQs have quantum advantage on useful workloads?

Quantum Supremacy Threshold: Experimental confirmation of quantum speedup on a well-defined (not necessarily *useful*) problem.

Qubits and quantum gates are very noisy, hardware not very usable.

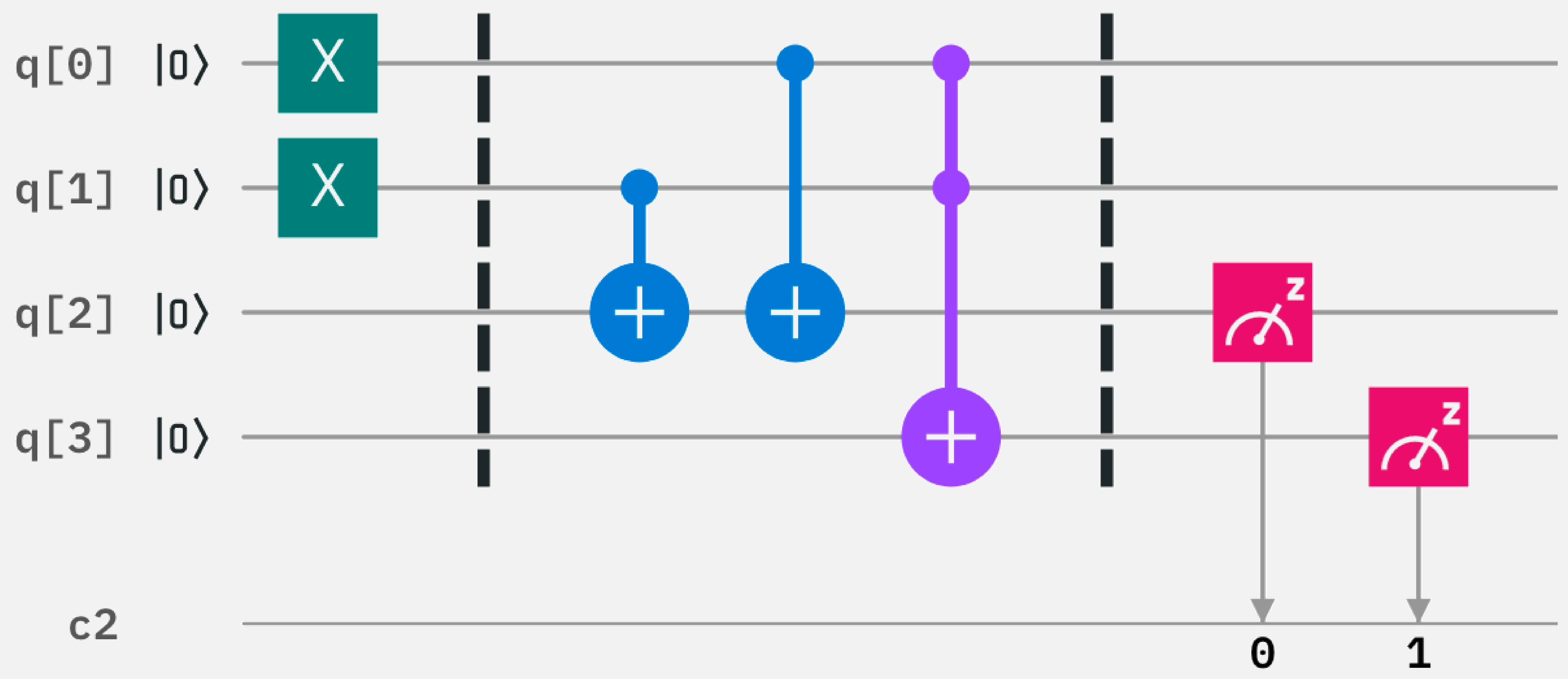
Active Research: Can this be simulated efficiently on GPU supercomputers?

GPU-BASED SUPERCOMPUTING IN THE QC ECOSYSTEM

Researching the Quantum Computers of Tomorrow with the Supercomputers of Today

QUANTUM CIRCUIT SIMULATION

Critical tool for answering today's most pressing questions in Quantum Information Science (QIS):



- What quantum algorithms are most promising for near-term or long-term quantum advantage?
- What are the requirements (number of qubits and error rates) to realize quantum advantage?
- What quantum processor architectures are best suited to realize valuable quantum applications?

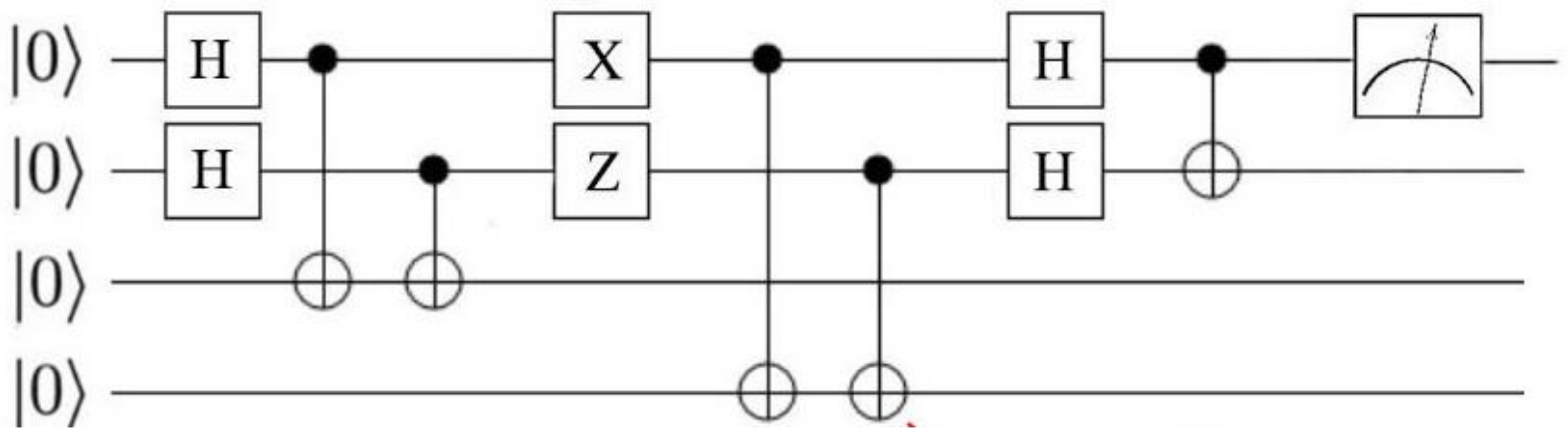
HYBRID CLASSICAL/QUANTUM APPLICATIONS

Impactful QC applications (e.g. simulating quantum materials and systems) will require classical supercomputers with quantum co-processors

+

- How can we integrate and take advantage of classical HPC to accelerate hybrid classical/quantum workloads

QUANTUM CIRCUIT SIMULATION APPROACHES

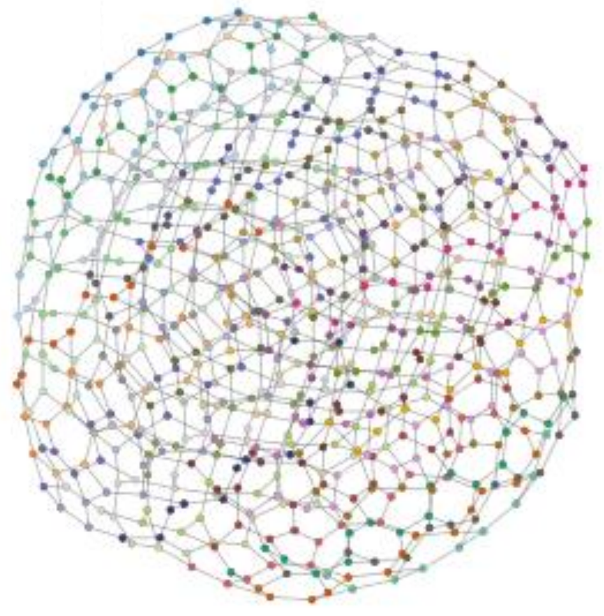


State vector

“Gate-based emulation of a quantum computer”

- Maintain full 2^n qubit vector state in memory
- Update all states every timestep, probabilistically sample n of the states for measurement

Memory capacity & time grow exponentially w/ # of qubits - practical limit around 50 qubits on a supercomputer



Tensor network

“Only simulate the states you need”

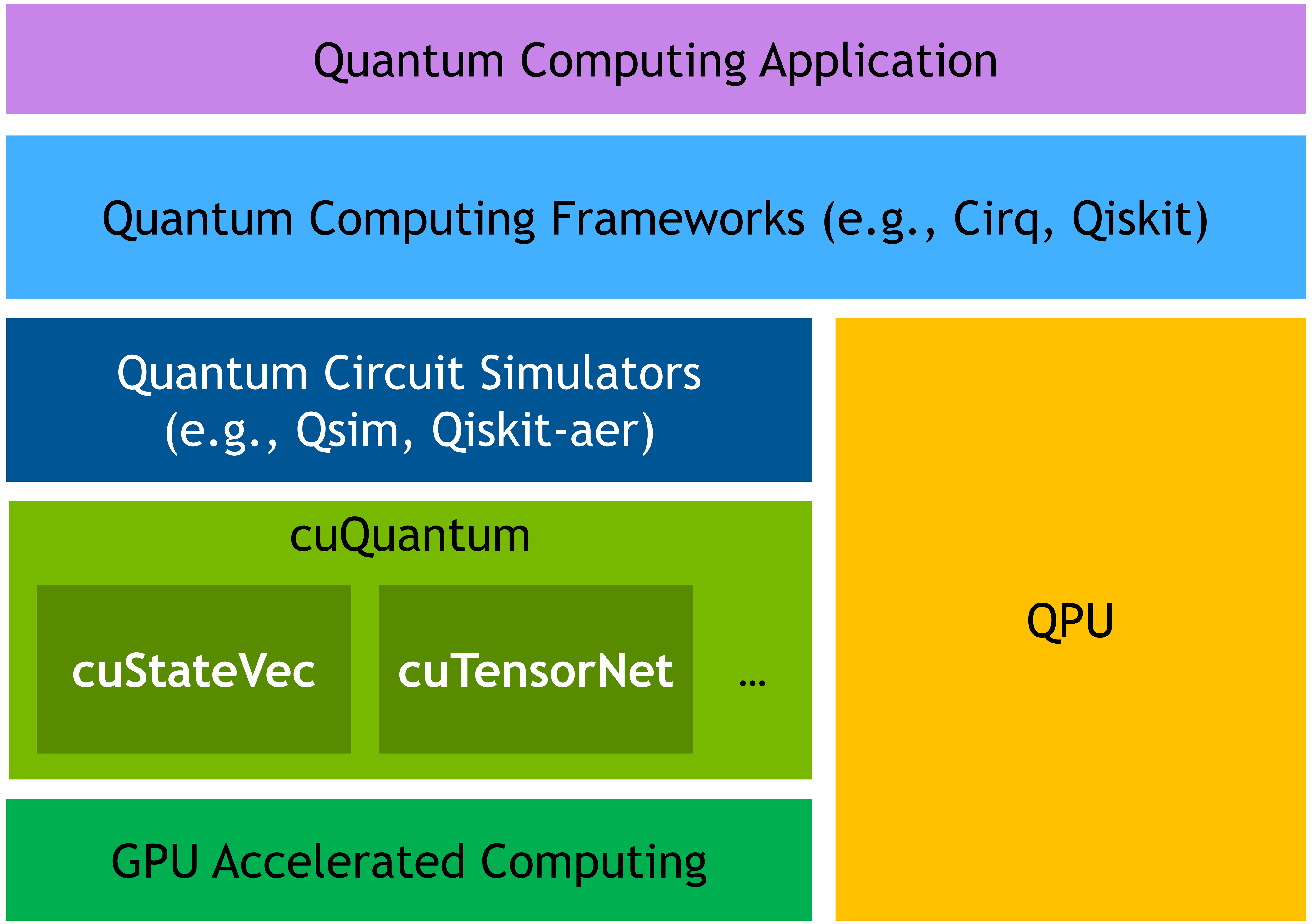
- Use tensor network contractions to dramatically reduce memory for simulating circuits
- Can simulate 100s or 1000s of qubits for many practical quantum circuits

GPUs are a great fit for either approach

NVIDIA

Introducing cuQuantum

- cuQuantum is an SDK of optimized libraries and tools for accelerating quantum computing workflows
- cuQuantum is not a:
  - Quantum Computer
  - Quantum Computing Framework
  - Quantum Circuit Simulator

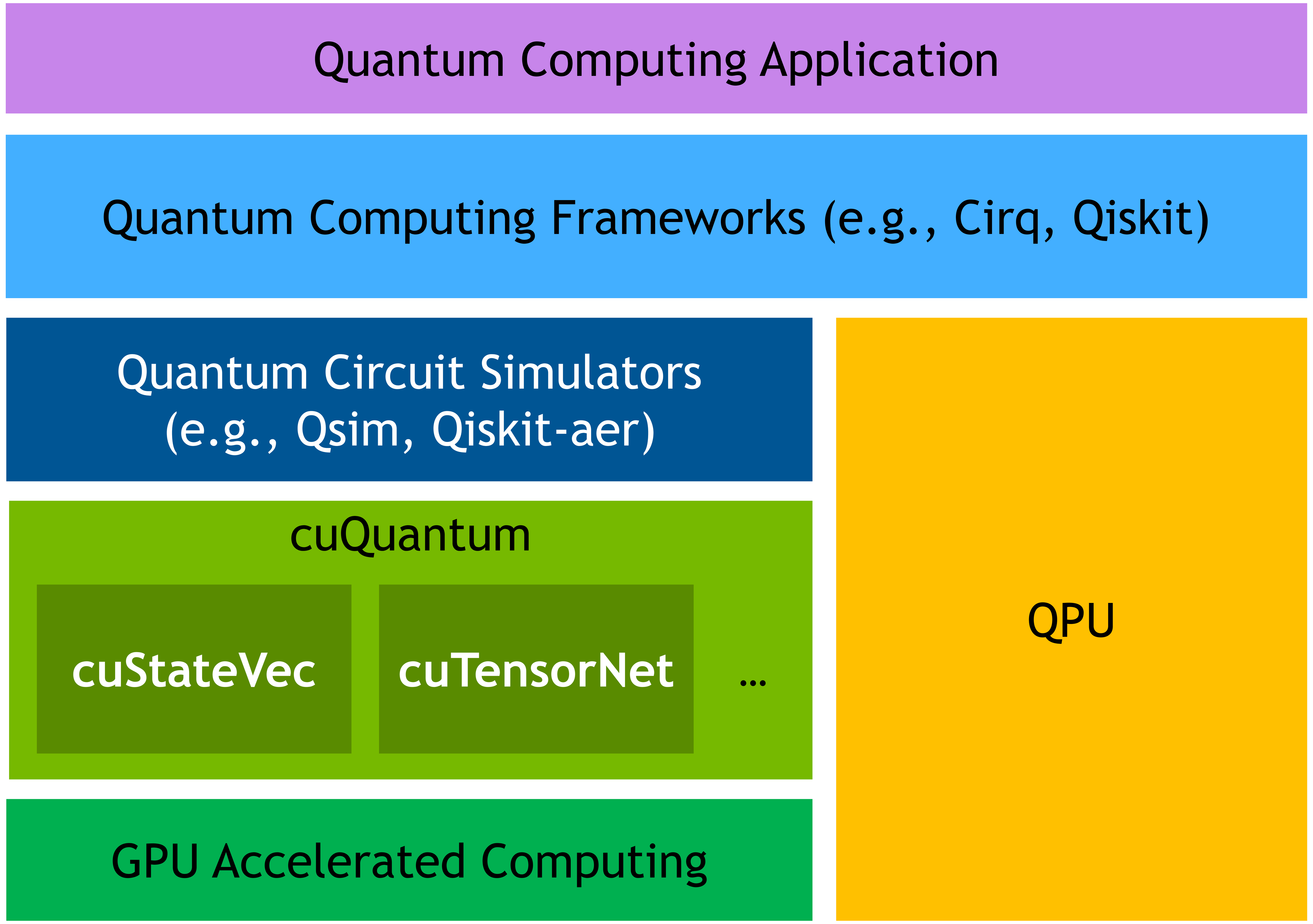


Introducing cuQuantum

- cuQuantum is a platform for quantum computing research
  - Accelerate Quantum Circuit Simulators on GPUs
  - Simulate ideal or noisy qubits
  - Enable algorithms research with scale and performance not possible on quantum hardware, or on simulators today
- Open Beta available now
  - Integrated with Cirq, Qiskit (December), Pennylane (January)

Cirq Qiskit

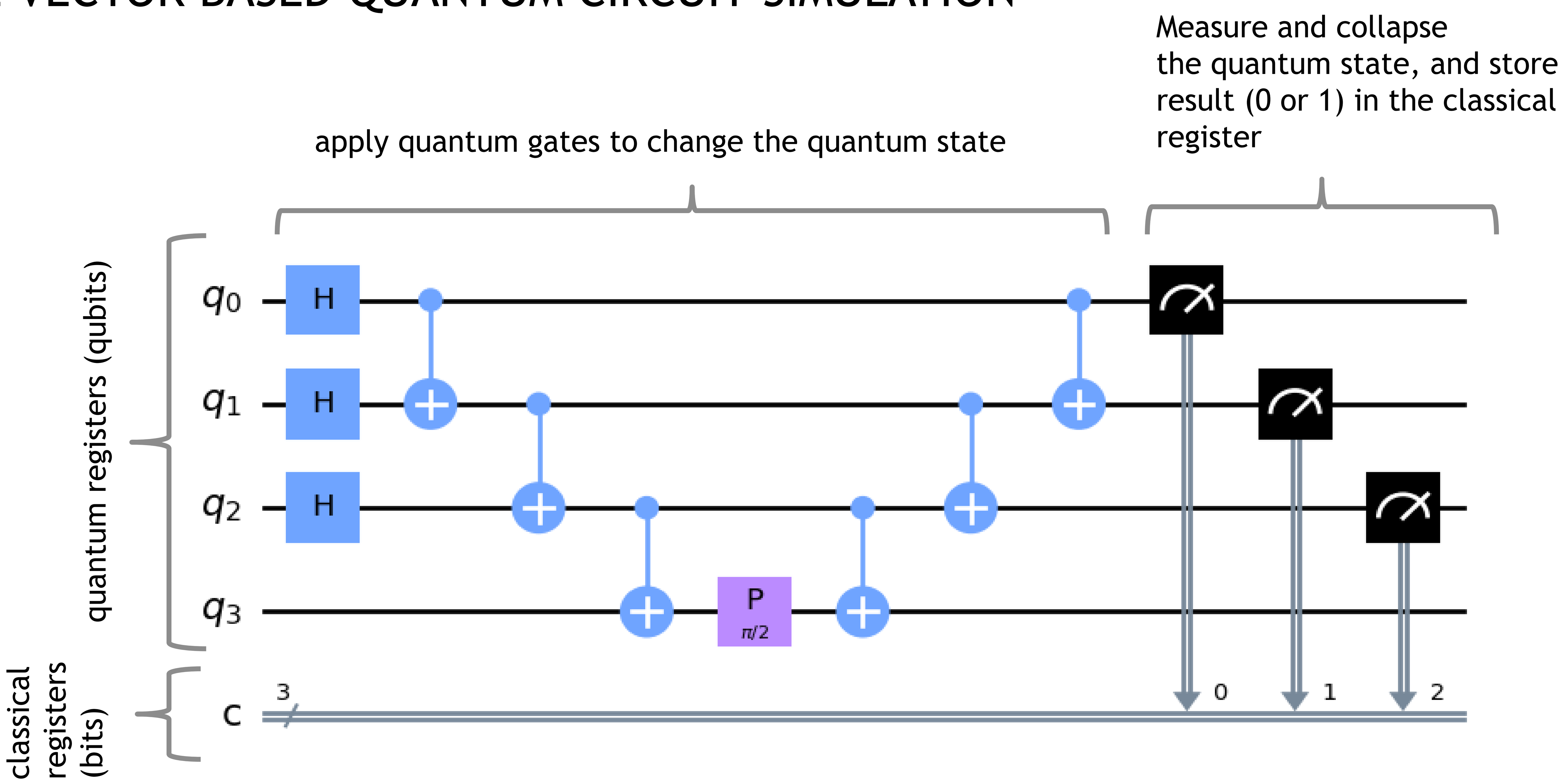
PENNYLANE



cuStateVec

A LIBRARY TO ACCELERATE STATE VECTOR BASED QUANTUM CIRCUIT SIMULATION

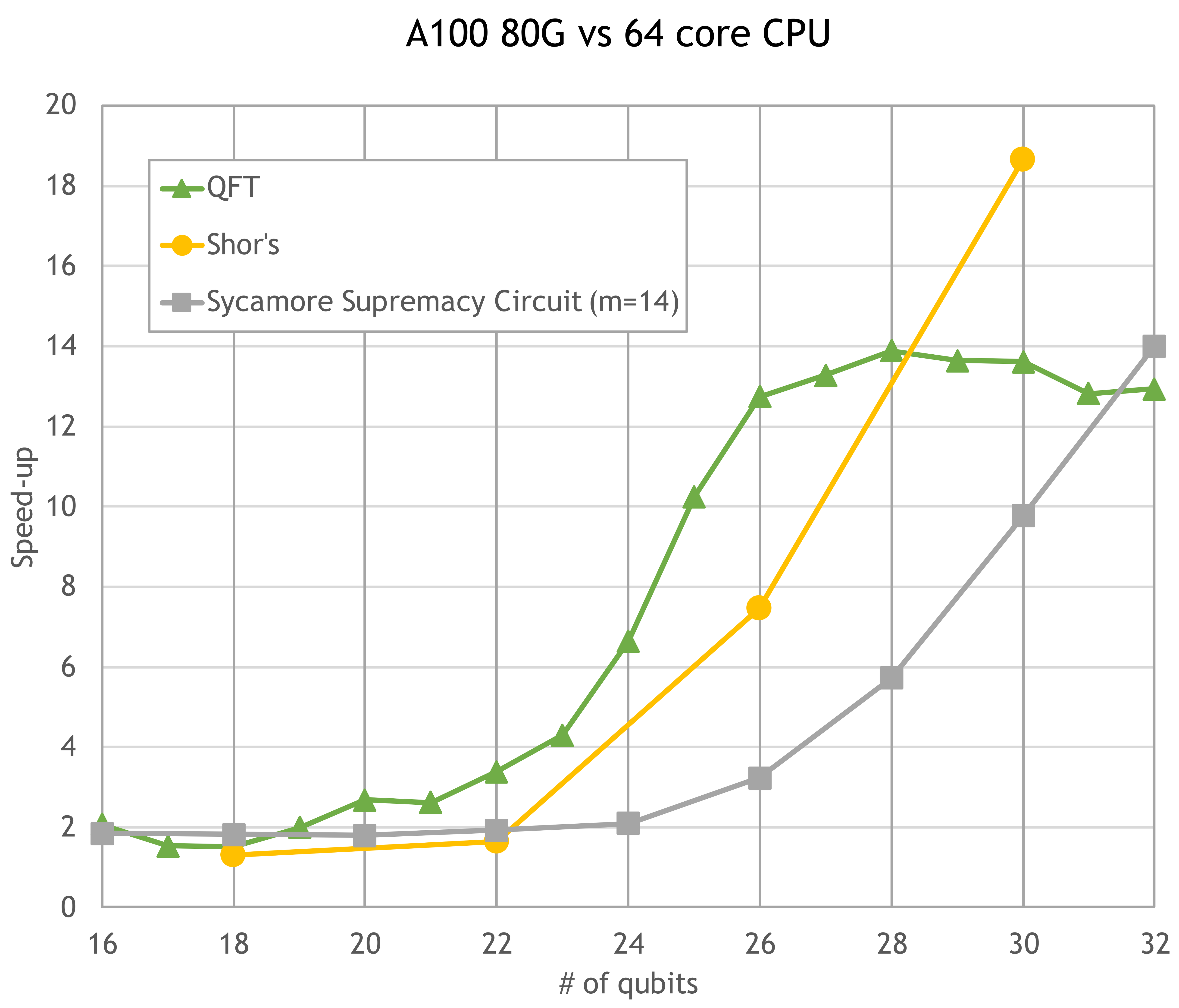
- APIs are specifically designed for state vector simulators, operating 'in-place' to save memory usage
- Preliminary benchmarks show ~10-20x improvement over CPU implementations with a single GPU
- Covers common use cases including:
    1) Measurement on a Z-product basis
    2) Batched single qubit measurement
    3) Apply gate matrix (facilitates gate fusion)
    4) Apply exponential of Pauli matrix product
    5) Expectation using matrix as observable
    6) Sampling
    7) Apply general permutation matrix
    8) Apply diagonal matrix
    9) Expectation on Pauli basis
    10) State vector segment extraction
    11) …

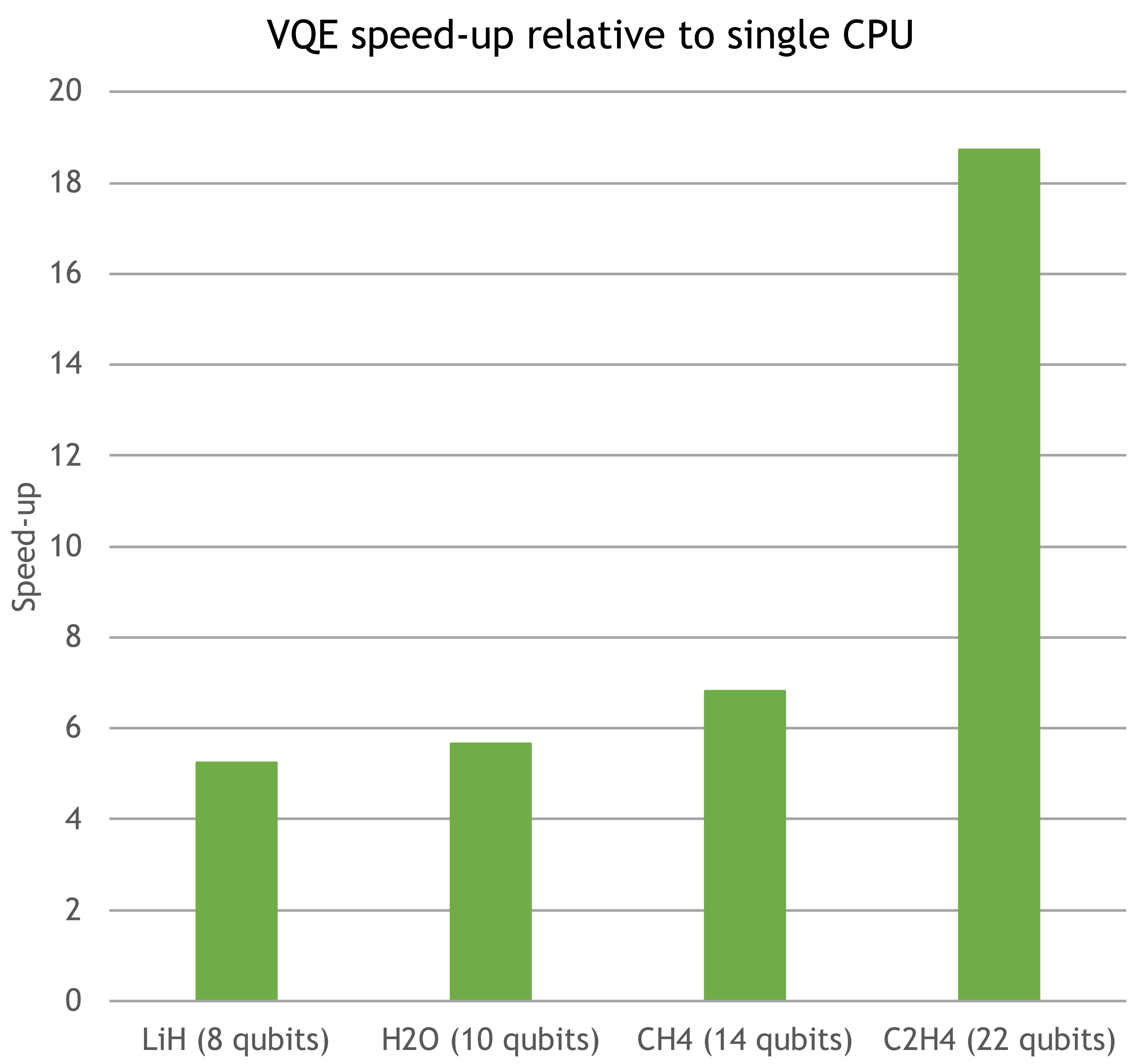



```
custatevecStatus_t
custatevecApplyMatrix(custatevecHandle_t          handle,
                      void*                       sv,
                      cudaDataType_t              svDataType,
                      const uint32_t              nIndexBits,
                      const void*                 matrix,
                      cudaDataType_t              matrixDataType,
                      custatevecMatrixLayout_t    layout,
                      const int32_t               adjoint,
                      const int32_t*              targets,
                      const uint32_t              nTargets,
                      const int32_t*              controls,
                      const uint32_t              nControls,
                      custatevecComputeType_t     computeType,
                      void*                       extraWorkspace,
                      size_t                      extraWorkspaceSizeInBytes);
```

cuStateVec - SINGLE-GPU

PRELIMINARY PERFORMANCE OF Cirq/Qsim + cuStateVec ON THE A100





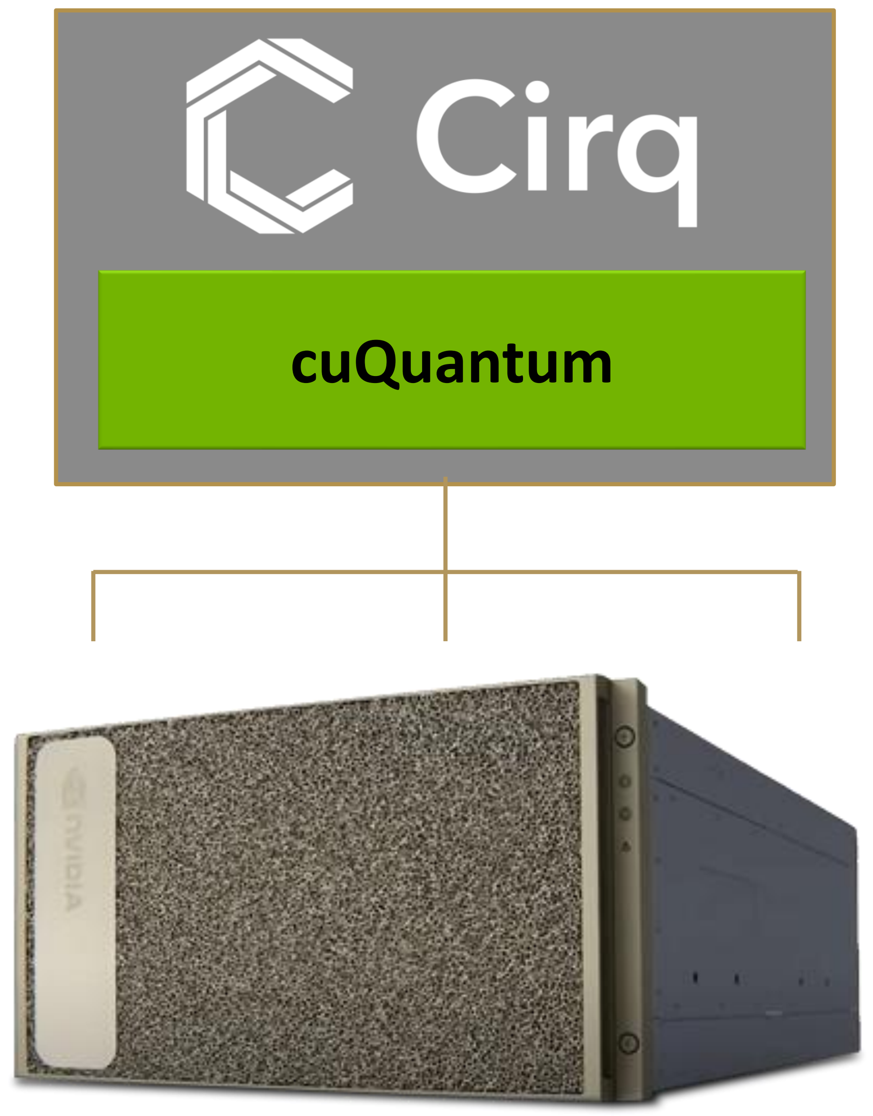
Benchmarks run using Cirq/Qsim with modifications to integrate cuStateVec
CPUs used were AMD EPYC 7742 with 64 cores
QFT circuit with 32 qubits and depth 63
Shor’s circuit with 30 qubit and depth 15560 (integer factorized: 65)
Sycamore supremacy circuit m=14 with 7480 gates

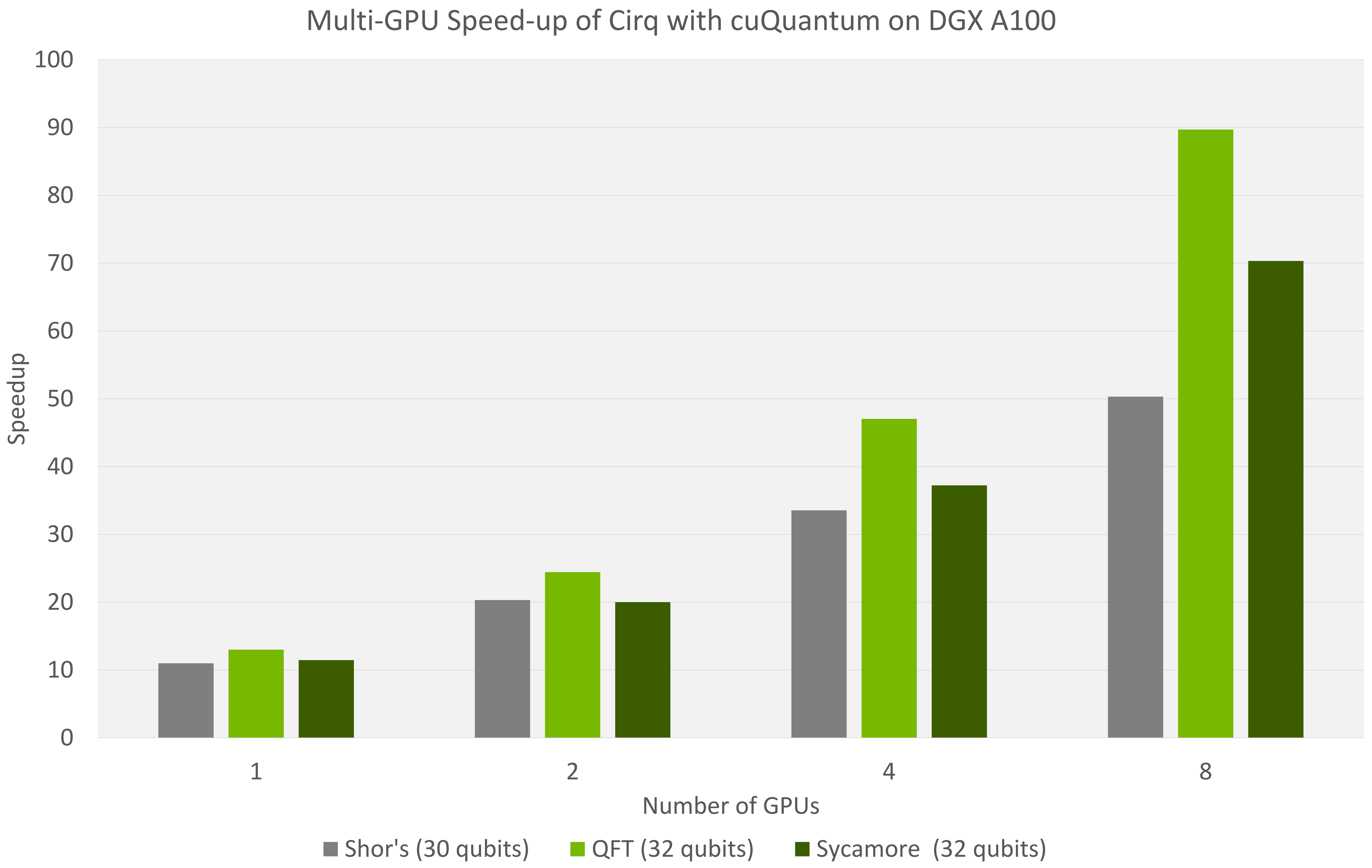
VQE benchmarks have all orbitals and results were measured for the energy function evaluation

Announcing DGX Quantum Appliance

MULTI-GPU CONTAINER WITH CIRQ/QSIM/CUQUANTUM

- Full Quantum Simulation Stack
- World class performance on key quantum algorithms
- Available Q1 2022



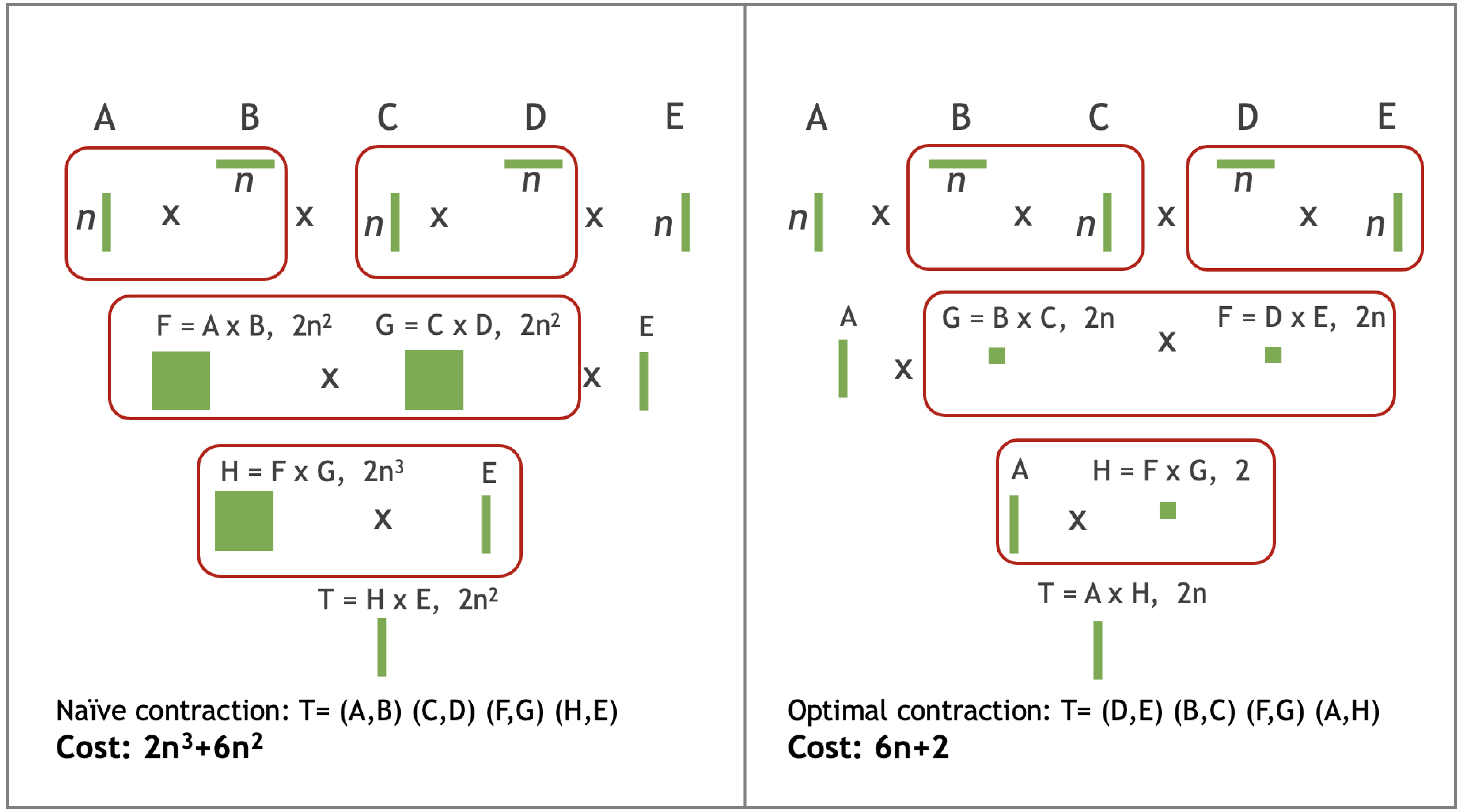


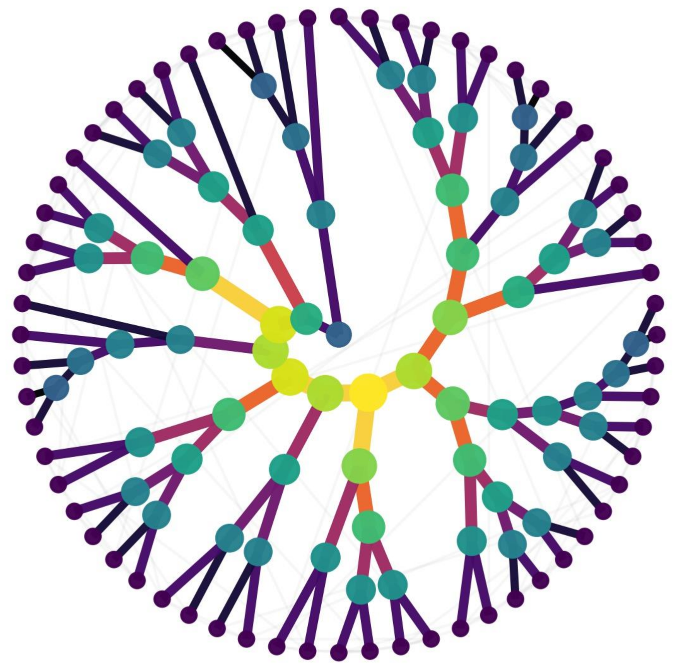
NVIDIA

cuTensorNet

A LIBRARY TO ACCELERATE TENSOR NETWORK BASED QUANTUM CIRCUIT SIMULATION

- The cuTensorNet library initially will provide the following APIs:
    1. Given a tensor network definition calculate optimal contraction path subject to memory constraints and parallelization needs:
        - Hyper-optimization is used to find contraction path with lowest total cost (eg, FLOPS or time estimate)
        - Slicing is introduced to create parallelism or reduce maximum intermediate tensor sizes
    2. Given a contraction path for a Tensor Network calculate an optimized execution plan
        - Leverages cuTENSOR heuristics
    3. Execute the TN contraction
- cuTensorNet depends on the latest cuTENSOR library for executing all pairwise contractions for cuTENSOR

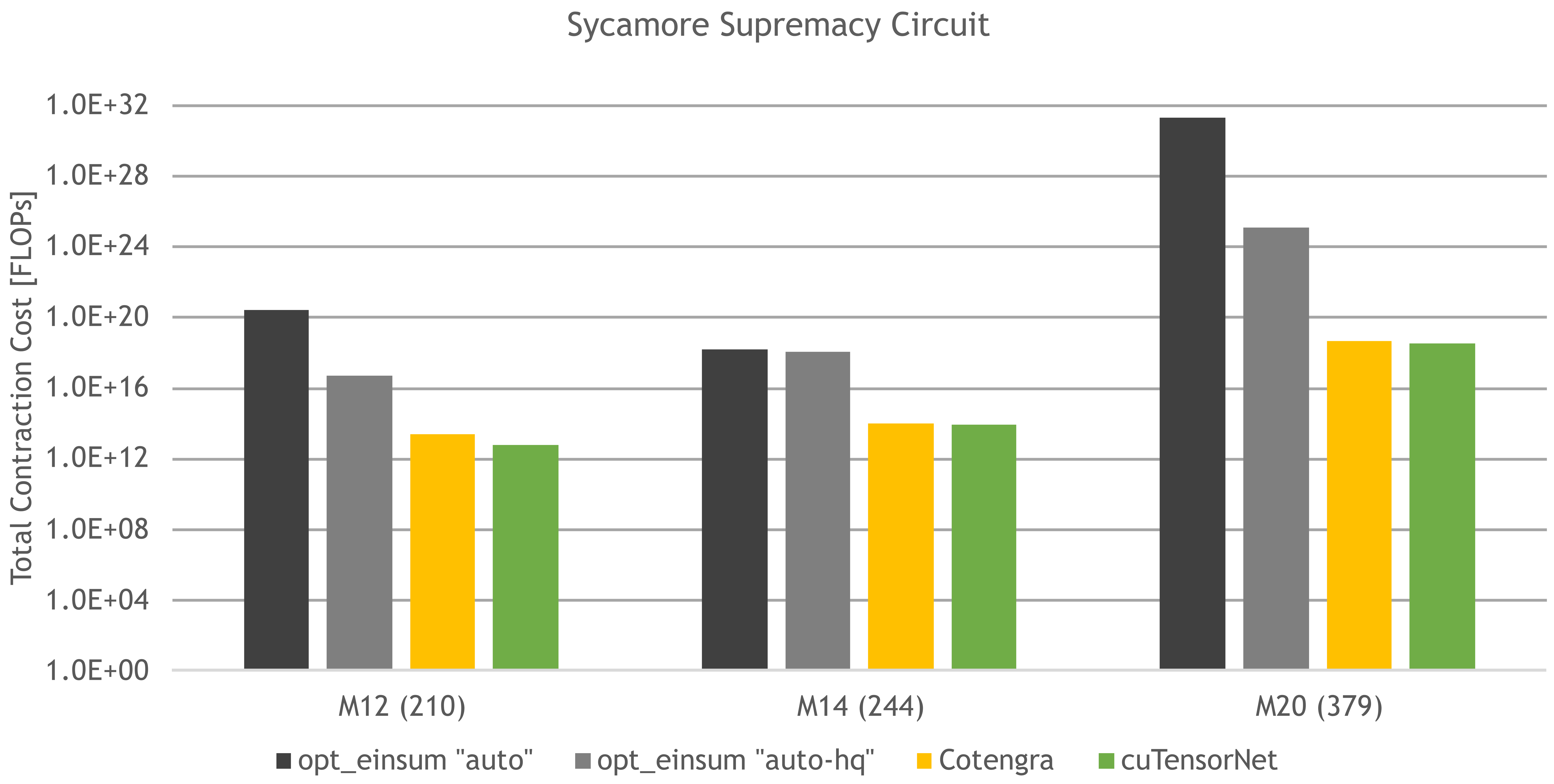




Tensor Network image from Quimb: https://quimb.readthedocs.io/en/latest/index.html

cuTensorNet

PRELIMINARY TENSOR NETWORK PATH OPTIMIZATION PERFORMANCE

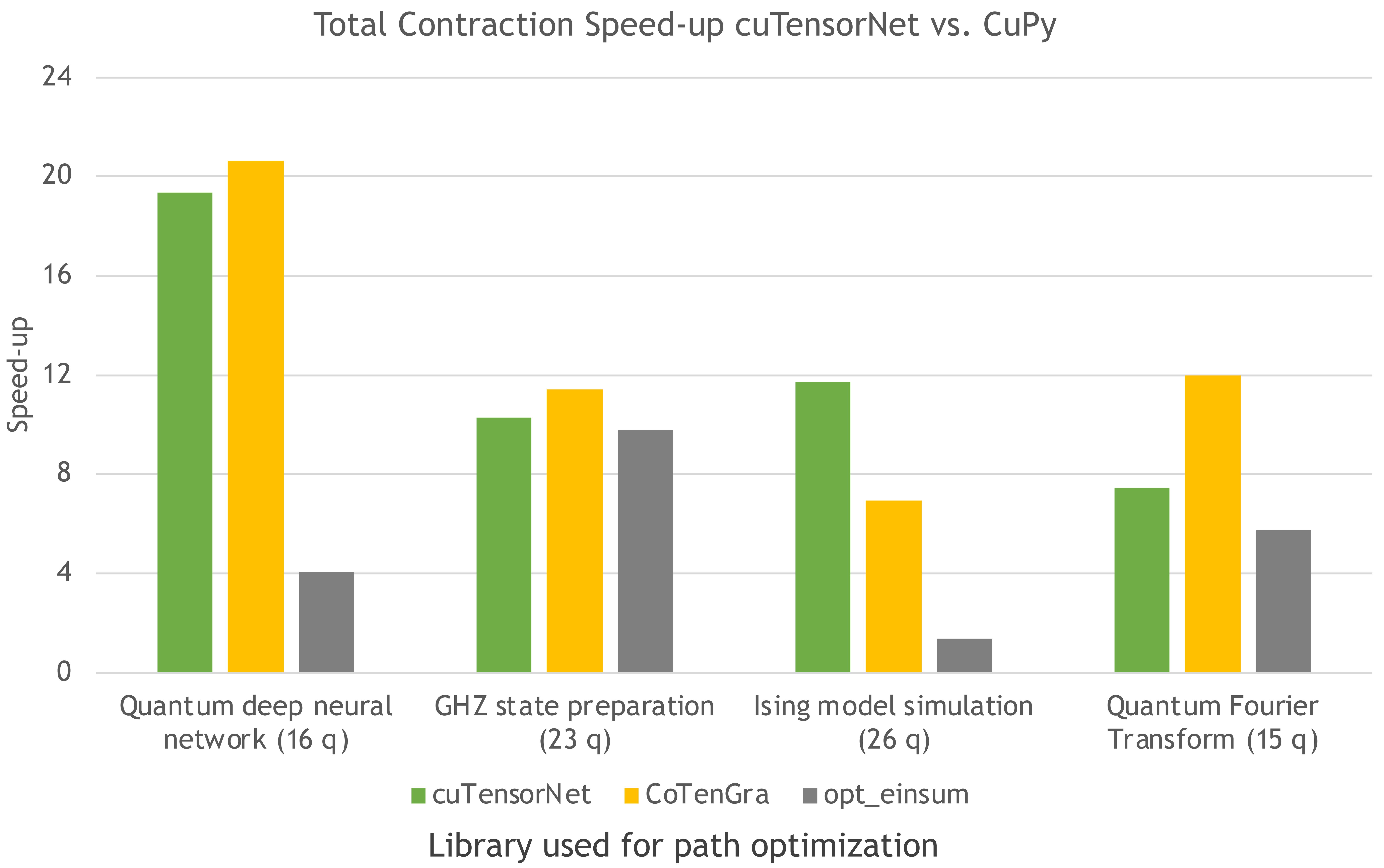


- Path optimization cost is amortized over many TN contractions in a QCS
- Performance optimizations within cuTensorNet allow efficient exploration of the solution space by its hyper-optimizer

[1] Gray & Kourtis, Hyper-optimized tensor network contraction, 2021 https://quantum-journal.org/papers/q-2021-03-15-410/pdf/
[2] opt-einsum https://pypi.org/project/opt-einsum/

cuTensorNet - SINGLE-GPU

PRELIMINARY PERFORMANCE DATA FOR TENSOR NETWORK CONTRACTION



- Switching execution from CuPy to cuTensorNet alone has big impact on performance regardless of which path optimization library is used
- Contraction performance of different paths is also dependent on the ordering
- Performance optimizations within cuTensorNet will allow exploration of a larger solution space by its hyperoptimizer
  - Path optimization cost is amortized over many TN contractions in a QCS

Benchmarks run using Cirq/Qsim with modifications to integrate upcoming multi-GPU APIs in cuStateVec
CPUs used were AMD EPYC 7742 with 64 cores each

The MaxCut Problem

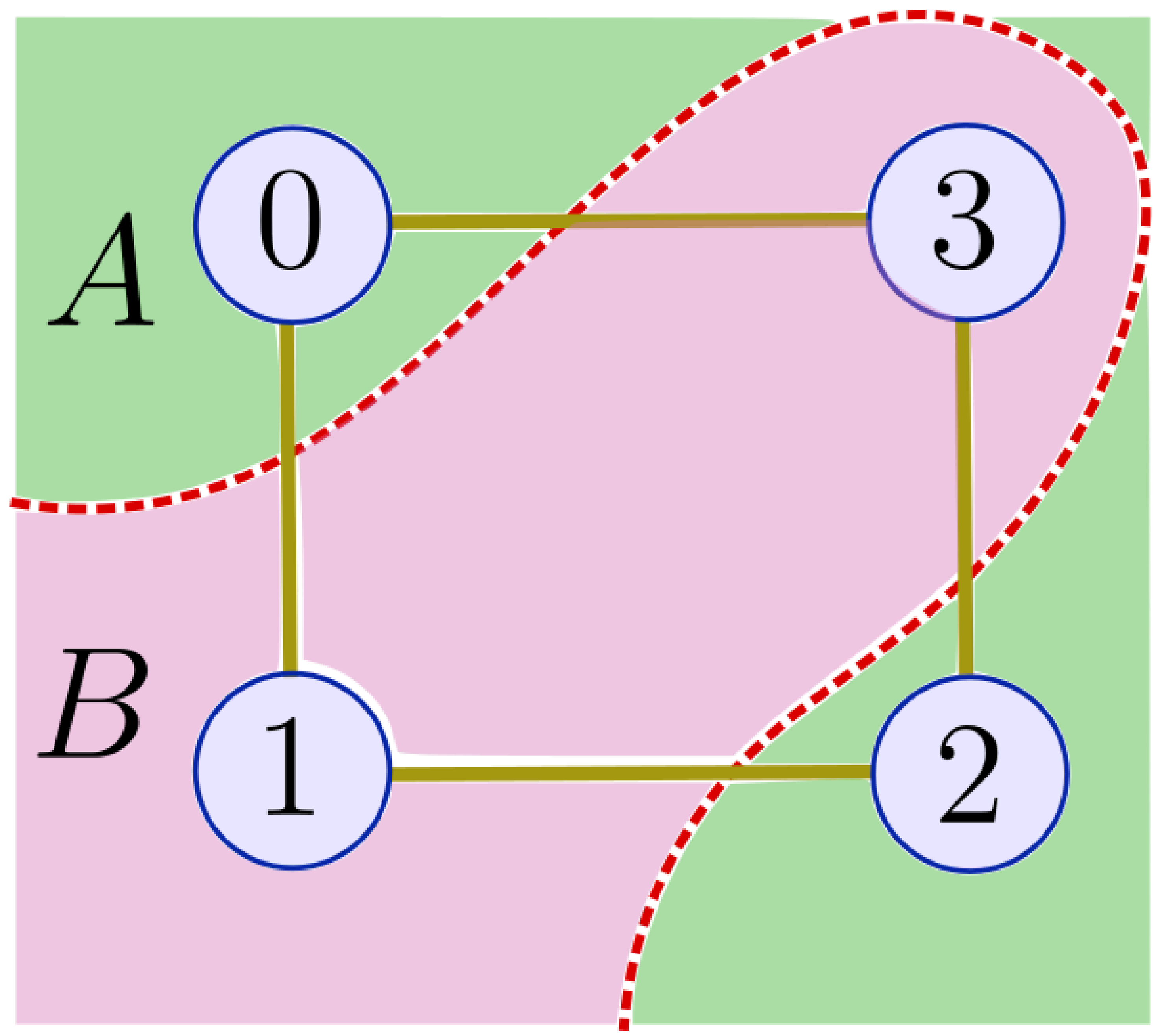
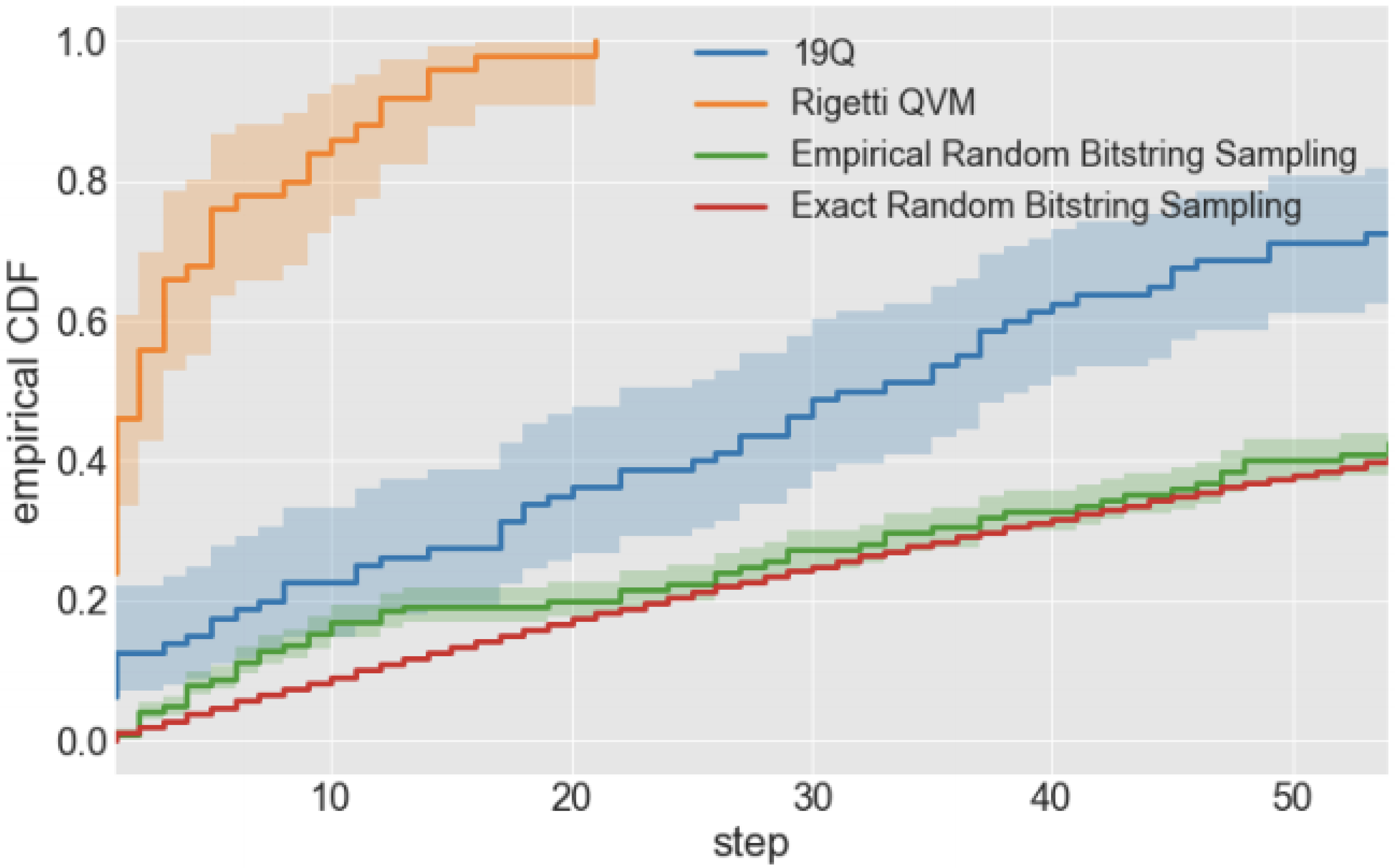


Image courtesy Xanadu Inc.
https://pennylane.ai/qml/demos/tutorial_qaoa_maxcut.html

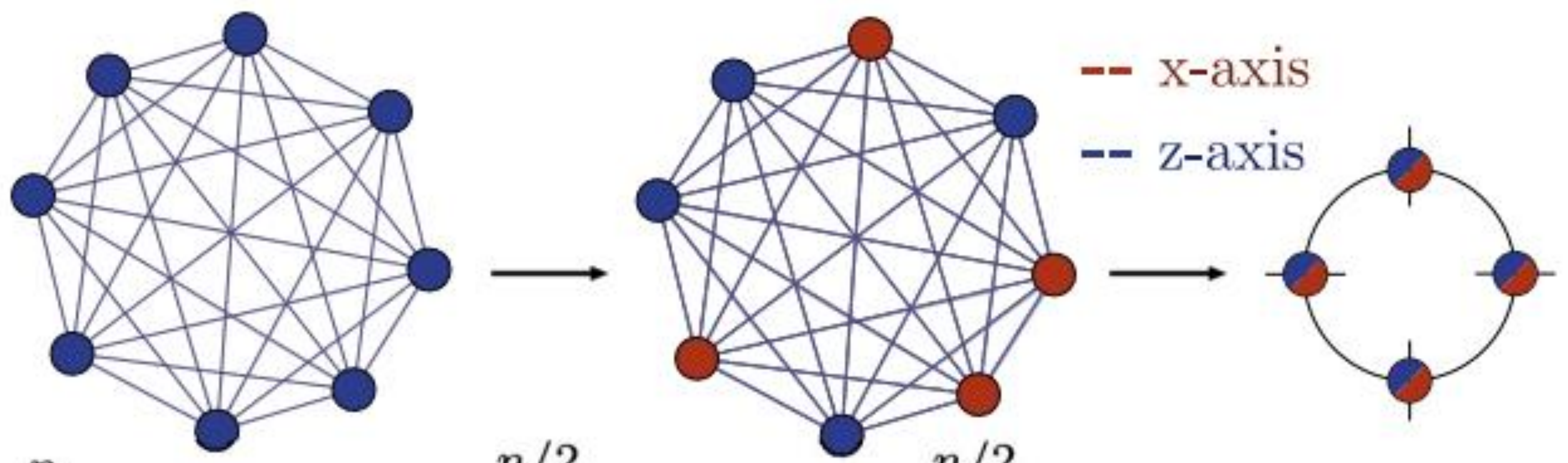


Otterbach et. al. Unsupervised Machine Learning on a Hybrid Quantum Computer. arxiv: 1712.05771

- NP-Complete combinatorial optimization problem
- Applications include clustering, network design, statistical physics, and more
- Early target for hybrid variational quantum algorithms
  - QAOA proposed by Farhi et.al.: arxiv: 1411.4028
- Several HW demonstrations, including on Rigetti 19Q chip in 2017

Simulating MaxCut using Tensor Networks

- Tensor Networks are a natural fit for MaxCut
  - Fried et. al. (2017) https://arxiv.org/abs/1709.03636
  - Huang et. al (2019) https://arxiv.org/pdf/1909.02559.pdf
  - Lykov et. al. (2020) https://arxiv.org/pdf/2012.02430.pdf

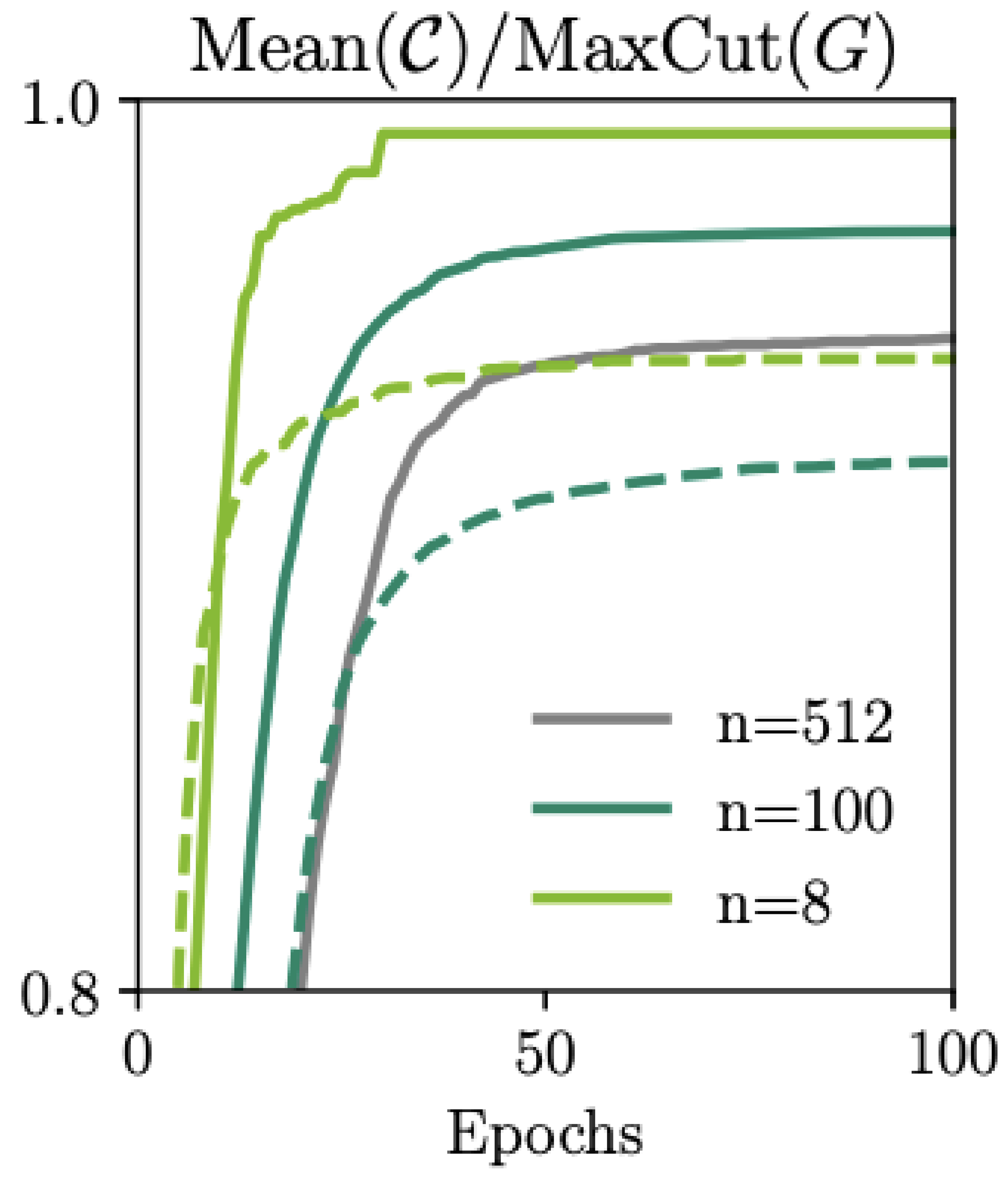


$$\sum_{j<i}^{n} w_{ij}^{zz}\,\sigma_i^z\sigma_j^z \rightarrow \sum_{j<i}^{n/2} w_{ij}^{zz}\,\sigma_i^z\sigma_j^z + \sum_{j<i}^{n/2} w_{ij}^{xx}\,\sigma_i^x\sigma_j^x \rightarrow \mathcal{L}_{\mathrm{MBE}}$$

- Patti et. al.(2021): NVIDIA Research proposes a novel variational quantum algorithm
  - Based on 1D tensor ring representation
  - Multibasis encoding
  - Able to find accurate solution for 512 vertices (256 qubits) on a single GPU

Paper: https://arxiv.org/pdf/2106.13304.pdf

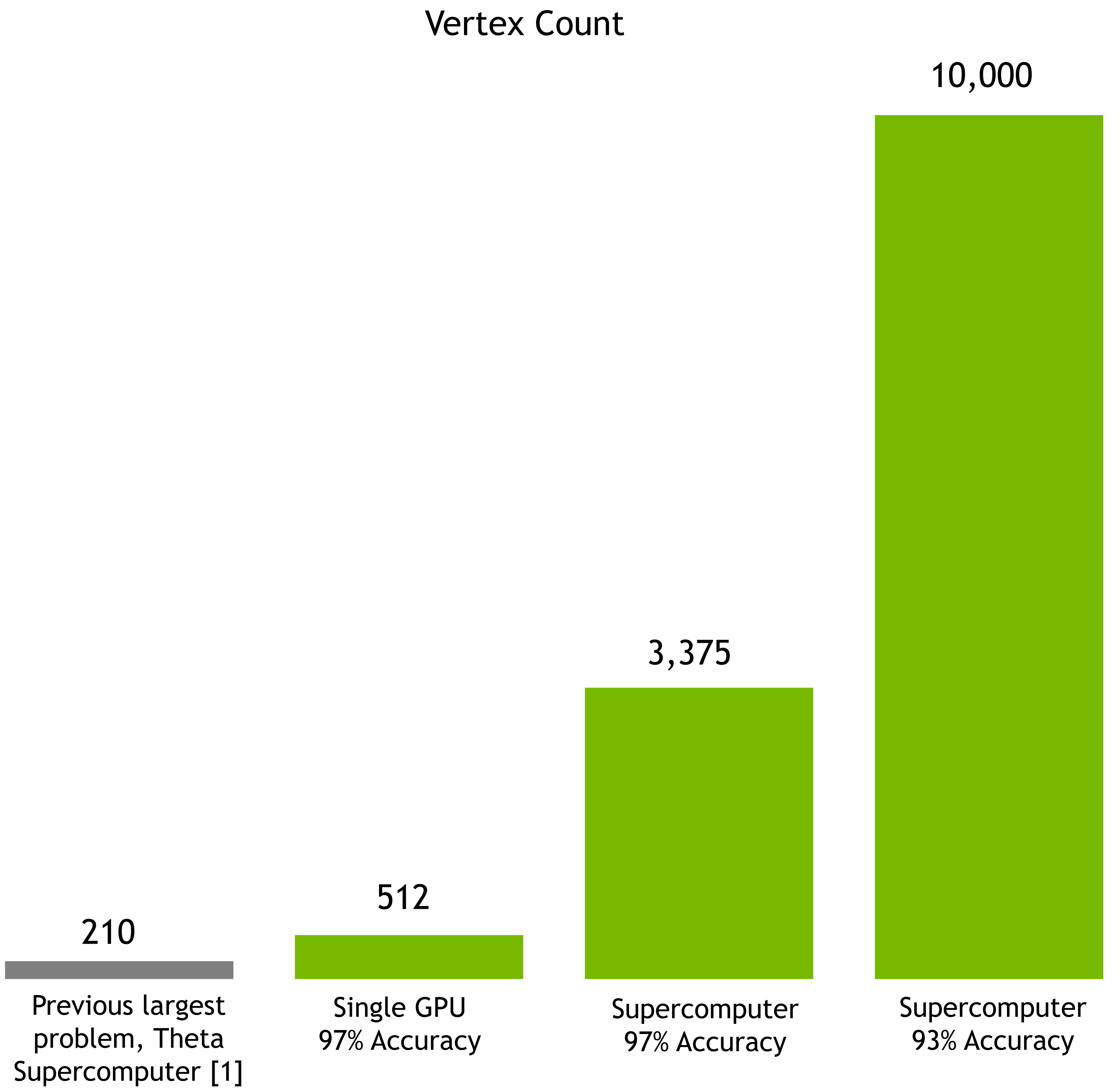
Code: https://github.com/tensorly/quantum



Scaling to a Supercomputer

NVIDIA's Selene DGX SuperPOD based supercomputer

- Using NVIDIA's Selene supercomputer
- Solved a 3,375 vertex problem (1,688 qubits) with 97% accuracy
- Solved a 10,000 vertex problem (5,000 qubits) with 93% accuracy

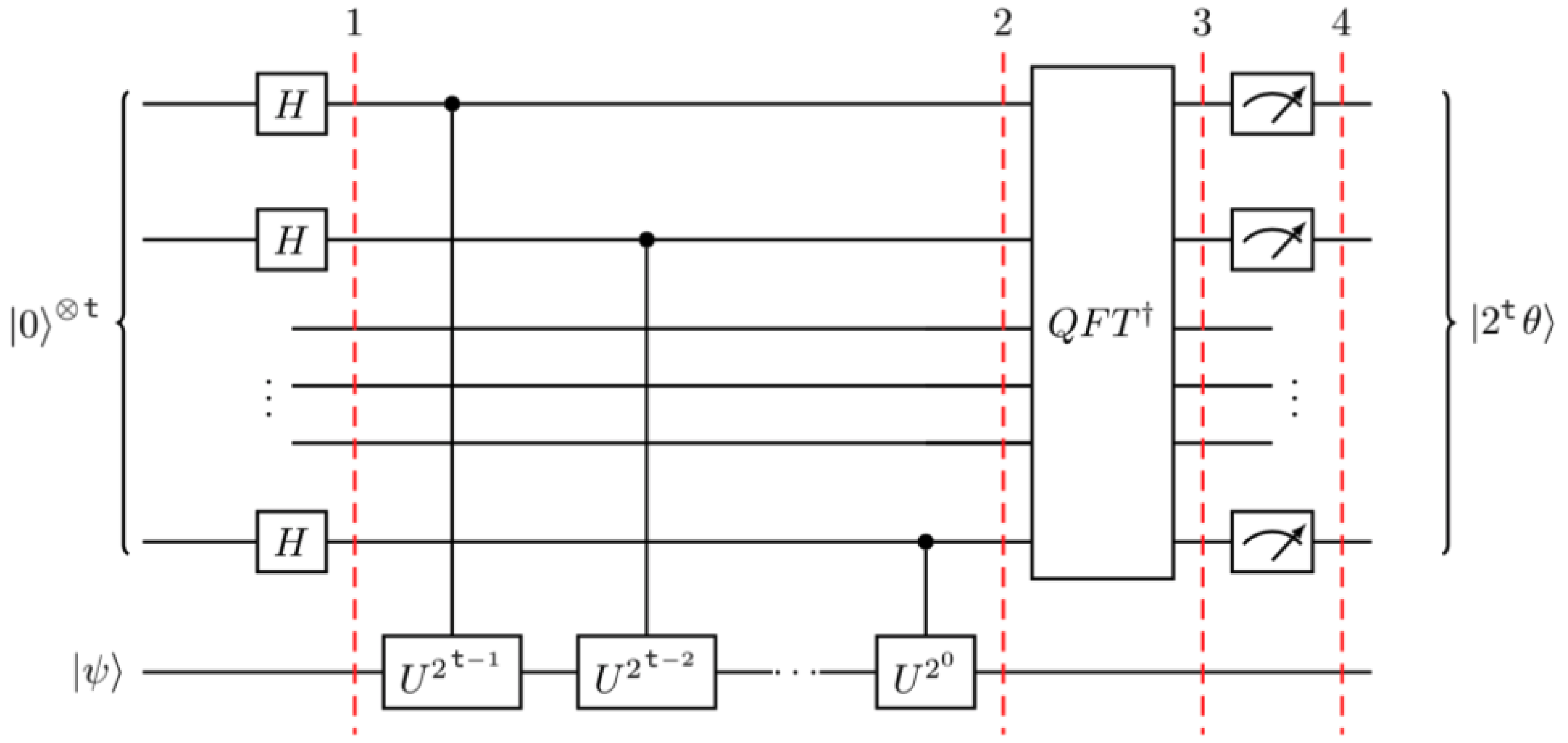


[1] Danylo Lykov et al, Tensor Network Quantum Simulator With Step-Dependent Parallelization, 2020
https://arxiv.org/pdf/2012.02430.pdf

Programming cuQuantum in C++

QCOR: A HPC-READY, C++ COMPILER FOR QUANTUM-CLASSICAL COMPUTING

Extend C++ with quantum kernels that can be compiled to available physical and simulation backends.




```
// Our qpe kernel requires oracles with
// the following signature.
using QPEOracleSignature = KernelSignature<qubit>;

__qpu__ void qpe(qreg q, QPEOracleSignature oracle) {
  // Extract the counting qubits and the state qubit
  auto counting_qubits = q.extract_range({0,3});

  // Get the eigenstate qubit
  auto state_qubit = q[3];

  // Put it in |1> eigenstate
  X(state_qubit);

  // Create uniform superposition on all 3 qubits
  H(counting_qubits);

  // run ctr-oracle operations
  for (auto i : range(counting_qubits.size())) {
    const int nbCalls = 1 << i;
    for (auto j : range(nbCalls)) {
      oracle.ctrl(counting_qubits[i], state_qubit);
    }
  }

  // Run Inverse QFT on counting qubits
  iqft(counting_qubits);

  // Measure the counting qubits
  Measure(counting_qubits);
}

// Oracle I want to consider
__qpu__ void oracle(qubit q) { T(q); }

int main(int argc, char **argv) {
  auto q = qalloc(4);
  qpe(q, oracle);
  q.print();
}
```

```
$ qcor -qpu custatevec qpe.cpp -shots 100
$ ./a.out
```

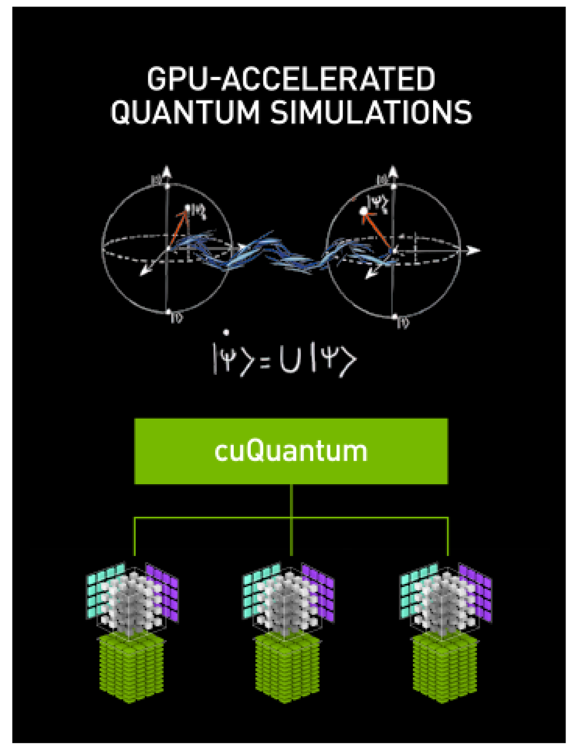
QCOR

QUANTUM COMPILER

Clang preprocessing, map to XACC API calls

XACC

QUANTUM FRAMEWORK



- Language extension to C++
  - Leverage novel plugins to Clang for processing __qpu__ kernels
  - Leverage existing C++ control flow semantics
  - Translate kernels to XACC API calls
- Deployed first Quantum MLIR Dialect enabling language lowering to the LLVM (Quantum Intermediate Representation)
  - Enables OpenQASM 3 programming and compilation to hybrid quantum-classical executable and library code
- Robust API for variational quantum programming
- Available under the new QIR Alliance

NVIDIA

VARIATIONAL ALGORITHMS WITH QCOR

High-level API for defining variational algorithms with user input on the quantum Operator, state preparation circuit, and classical optimization.

- Variational Quantum Eigensolver
  - Used to compute minimal eigenvalue of given Hamiltonian
  - Define a state preparation ansatz, parameterized to explore the Hilbert space and search for ground eigenstate
  - Ansatz measurements dictated by Hamiltonian structure
- In qcor
  - State prep is a quantum kernel function, parameterized by function arguments
  - Easy mechanism for creating Hamiltonian Operator from existing chemistry packages
  - Define optimization functions via standard lambdas.
  - Optimizer extension point, implemented to provide a wide variety of classical optimization routines.

Compile and execute on physical architecture with

```
$ qcor -qpu ibm:ibmq_boeblingen qpe.cpp -shots 100
$ ./a.out
```

Compile and run on cuStateVec with

```
$ qcor -qpu custatevec qpe.cpp -shots 100
$ ./a.out
```

```
__qpu__ void ansatz(qreg q, double theta) {
  X(q[0]);
  X(q[2]);
  compute {
    Rx(q[0], constants::pi / 2);
    for (auto i : range(3)) H(q[i + 1]);
    for (auto i : range(3)) {
      CX(q[i], q[i + 1]);
    }
  }
  action { Rz(q[3], theta); }
}

int main() {
  std::string h2_geom = R"#(H  0.000000   0.0      0.0
H   0.0        0.0  .7474)#";
  auto H =
      createOperator("pyscf", {{"basis", "sto-3g"}, {"geometry", h2_geom}});

  OptFunction opt_function(
      [&](std::vector<double> x) {
        return ansatz::observe(H, qalloc(4), x[0]);
      },
      1);

  auto [energy, opt_params] = createOptimizer("nlopt")->optimize(opt_function);
  print(energy);
}
```

PROGRAMMING CUQUANTUM WITH OPENQASM 3

Build on the MLIR, lower languages to the LLVM adherent to the QIR specification

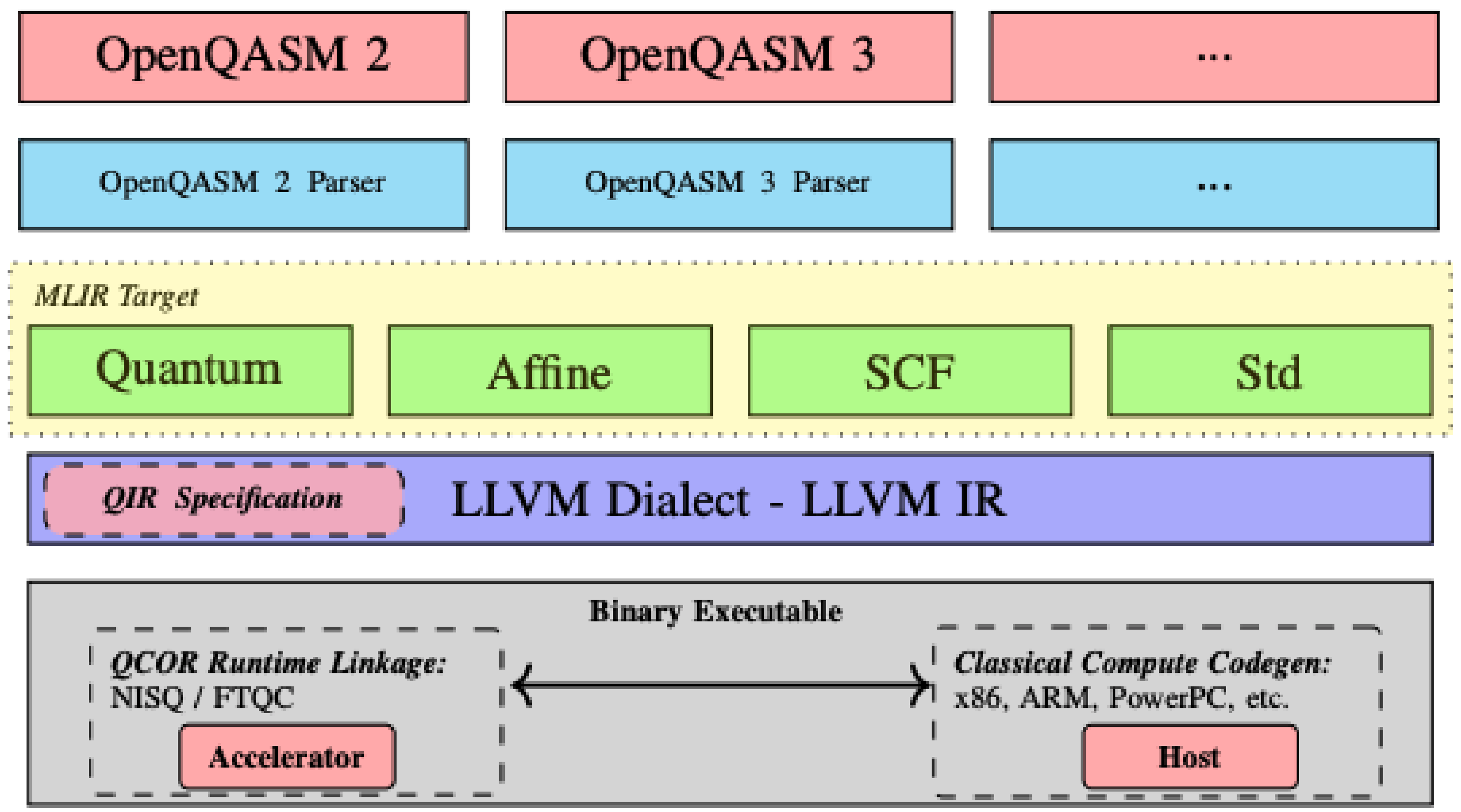
- Quantum Intermediate Representation
  - Unified compiler representation embedded in the LLVM IR
  - Low-level target for language lowering
- MLIR and the Quantum Dialect
  - Language-level representation for quantum computing
  - Progressive lowering from language-level IR to LLVM IR adherent to the QIR specification
  - Can mix dialects (specifically those for classical control flow)
- Parse OpenQASM3, walk parse tree, generate MLIR tree, transform / lower to LLVM IR
- Use LLVM toolchain to generate binary executable, link to QIR implementation library
- cuStateVec backend enabled through existing QCOR/XACC integration

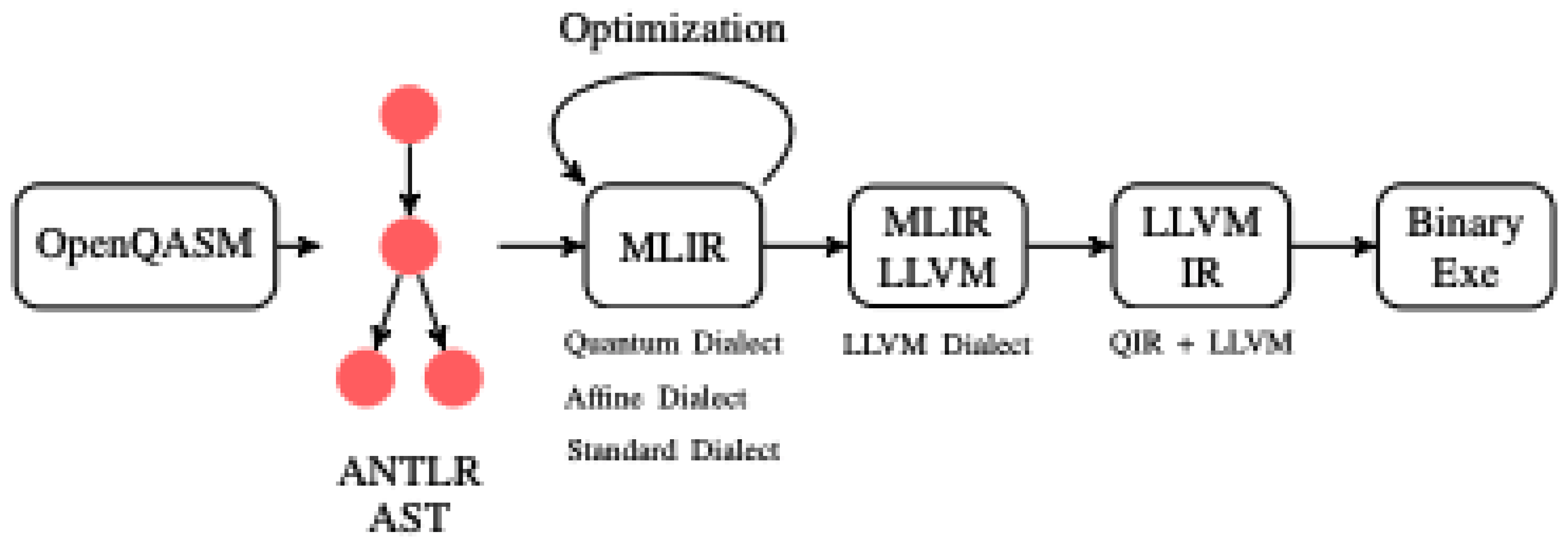
Compile and run on cuStateVec with

```
$ qcor -qpu custatevec rwpe.qasm
$ ./a.out
```

Random-walk Phase Estimation

```
OPENQASM 3;

int n_iterations = 24;
int iteration = 0;

double mu = 0.7951;
double sigma = 0.6065;
double theta;
double c_theta;

qubit target;
qubit aux;
bit result;

x target;

while (iteration < n_iterations) {

    h aux;
    theta = 1 - mu / sigma;
    rz(theta) aux;
    c_theta = 0.25 / sigma;
    rz(c_theta) target;
    cnot aux, target;
    rz(-1 * c_theta) target;
    cnot aux, target;
    h aux;
    measure aux -> result;

    if (result) {
        x aux;
        mu += sigma * 0.6065;
    } else {
        mu -= sigma * 0.6065;
    }
    sigma *= 0.7951;

    iteration += 1;
}

print(mu * 2);
```






NVIDIA