OLCF CUDA-Q Training

August 22, 2024

Presenters:

Justin Lietz

Support:

Alex McCaskey

Thien Nguyen

Dmitry Lyakh

Marwa Farag

Agenda

- Setup and Troubleshooting
- Brief Intro to Quantum
- Intro CUDA-Q
- Hands-on CUDA-Q
- End

Logistics

Let’s debug any setup issues first

- Workshop materials: https://github.com/justinlietz/ornl-cudaq-workshop (including these slides and links)
- Three modes to follow along:
  - OLCF Ascent
  - X-Lab
  - Laptop
- OLCF Ascent
  - Everyone here should have access to TRN024 training account.
  - Ascent user guide: https://docs.olcf.ornl.gov/systems/ascent_user_guide.html
- X-Lab
  - A10 nodes available for us for today.
  - Tested Jupyter notebook environment.
  - You will not be able to access these machines after the workshop.
  - Make account at https://learn.nvidia.com
- Laptop
  - Quick start guide: https://nvidia.github.io/cuda-quantum/latest/using/quick_start.html
  - Can easily install CUDA-Q either via conda, pip, or docker.
  - CPU only unless your laptop has NVIDIA GPU.

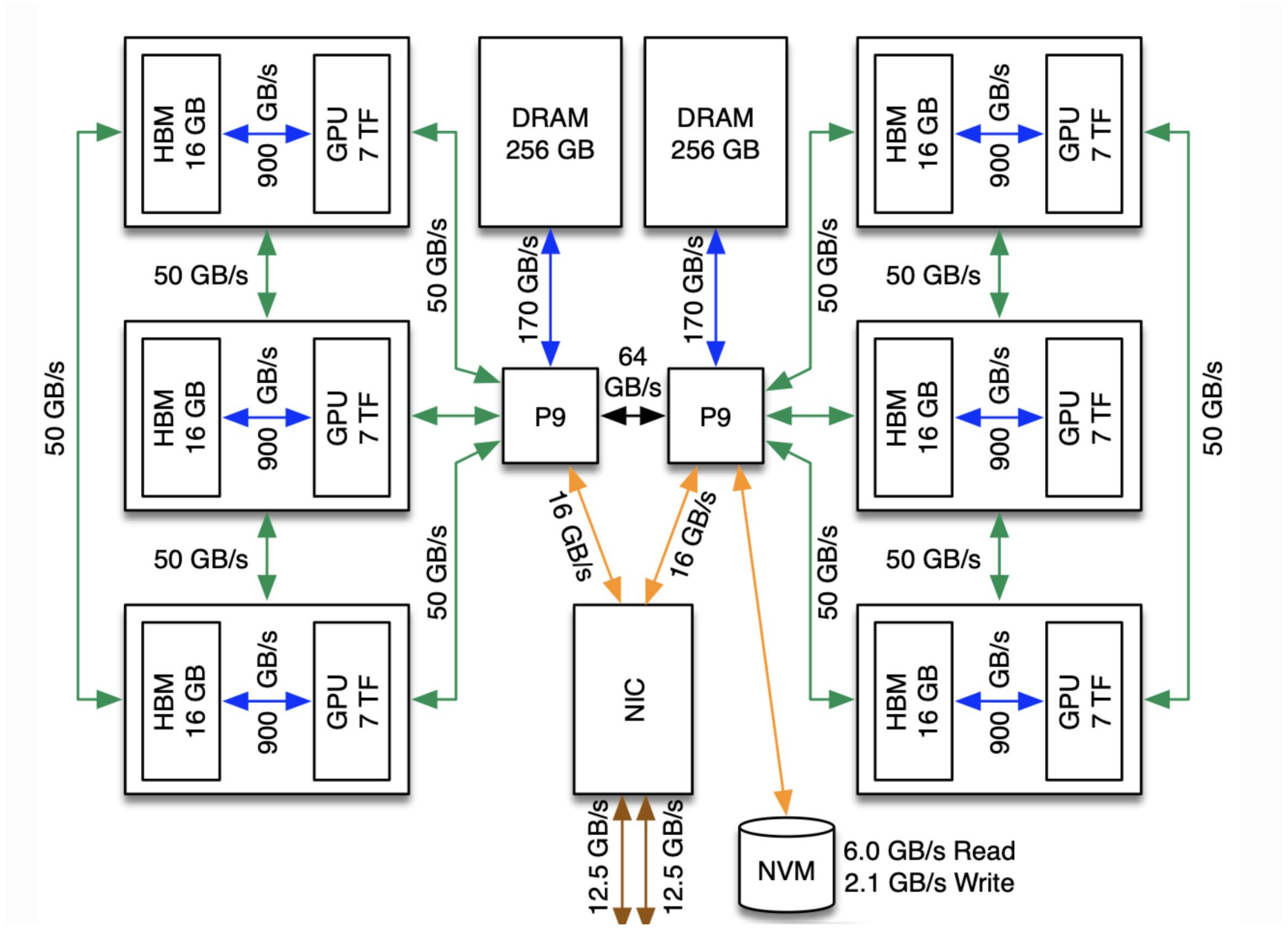
X-Lab

Best to grab these resources now.

- If you haven't already, make an account at: http://learn.nvidia.com/join
- Log in (link below works better if you log in first then click link below)
- Go to https://learn.nvidia.com/dli-event
- Enter code: NVIDIA_XLAB_AU22
- Click "start" to spin up a lab. If you think you'll want to follow along with the Jupyter notebooks.
- Hot labs have already been spun up for us, but we can spin up more later, they may take ~15 minutes though.
- We'll start with `cudaq-Intro/cudaq-introduction.ipynb` after these slides.
- These notebooks can be found at: https://github.com/cudaq-libraries/workshops/tree/202408-cudaq

Ascent Crash Course

- Ascent user guide: https://docs.olcf.ornl.gov/systems/ascent_user_guide.html
- Workshop materials: https://github.com/justinlietz/ornl-cudaq-workshop
- 18 Summit nodes, each with 6 v100 GPUs.
  - Please limit time on interactive nodes so there's enough to share
- Job scripts and environment script is in the \`batch_scripts\` dir
  - Launch environment with \`source load_env.sh\`
  - Submit job with \`bsub –L$SHELL <scriptname.sh>
  - A good test code is \`targets.py\` in the scripts folder
- Everyone will be on the same login node, so only run small tests
- The environment script uses a conda env, so try not to run it twice. If you do, just log out and back in again.



Ascent Crash Course

- Our training account is TRN024
- Example script showing how to use 1, 6, and 12 GPUs.
- Notice resources are requested at the top, but then accessed only with jsrun command (even if only requesting 1 node)
- This example script is in `batch_scripts` and uses a relative path to the script. You can `cd` here as well.

```
#BSUB -P TRN024
#BSUB -W 0:10
#BSUB -nnodes 2
#BSUB -alloc_flags gpudefault
#BSUB -J cudaq_py_nvidia
#BSUB -o cudaq_py_nvidia_%J.output
#BSUB -e cudaq_py_nvidia_%J.error

module purge
module use /gpfs/wolf2/olcf/trn024/proj-shared/modulefiles
module load gcc/11.2.0
module load cudaq/0.8.0
module load spectrum-mpi/10.4.0.3-20210112

# 1 rank with 1 GPU
jsrun -n 1 -a 1 -c 1 -g 1 time python3 ../demos/MQPU-MGPU/observe-qml-mnmgpu.py
# 1 rank with 6 GPUs
jsrun -n 1 -a 1 -c 1 -g 6 time python3 ../demos/MQPU-MGPU/observe-qml-mnmgpu.py
# 2 ranks each with 6 GPUs
jsrun -n 2 -a 1 -c 1 -g 6 time python3 ../demos/MQPU-MGPU/observe-qml-mnmgpu.py
```

Quantum Computing

A two-slide summary

- What is it?
  - Physics at very small scales is accurately described by the theory of quantum mechanics.
  - Quantum computing is an approach to use this theory to do information processing.
- Why do we care?
  - Known asymptotic speedups for a variety of important problems in cryptography, chemistry, materials science, and optimization.
  - Even a few successful implementations of these algorithms in this space could have large economical impact.
- How does progress look so far?
  - Research is exploring a variety of different architectures to implement a quantum computer.
  - Many dimensions which progress can be evaluated, but all are improving (speed, fidelity, number of qubits).
  - Each architecture has its own trade-offs.
    - Superconducting qubit quantum computers have the fastest runtimes and large qubit counts (100-1000's) but have a restrictive connectivity.
    - Trapped ion and neutral atom quantum computers are slower (need to physically move atoms) but have the current best fidelities and increase connectivity.
    - Many more architectures! Topological qubits require cutting edge materials science research, notably studied at ORNL.

Quantum Computing

What resources make quantum computers different?

- Pros:
  - A bit can store 0 or 1, where as a quantum bit (qubit) can be in a superposition of both.
  - We can generate quantum entanglement between qubits, creating an exponentially larger space to perform calculations in.
  - The qubit probability can be constructively/destructively interfered to preferentially select desired solutions from this large space.
- Cons:
  - Hardware challenges, as quantum computers are extremely sensitive to noise. Most quantum computers require extremely low temperatures.
  - Quantum error correction overhead is much larger than classical error correction.
  - Cannot readout quantum data. We get classical data bits at the end of the algorithm, which can be a bottleneck.
  - Probabilistic nature means that algorithms are ran many times and statistics must be collected.

Quantum-Accelerated Supercomputing

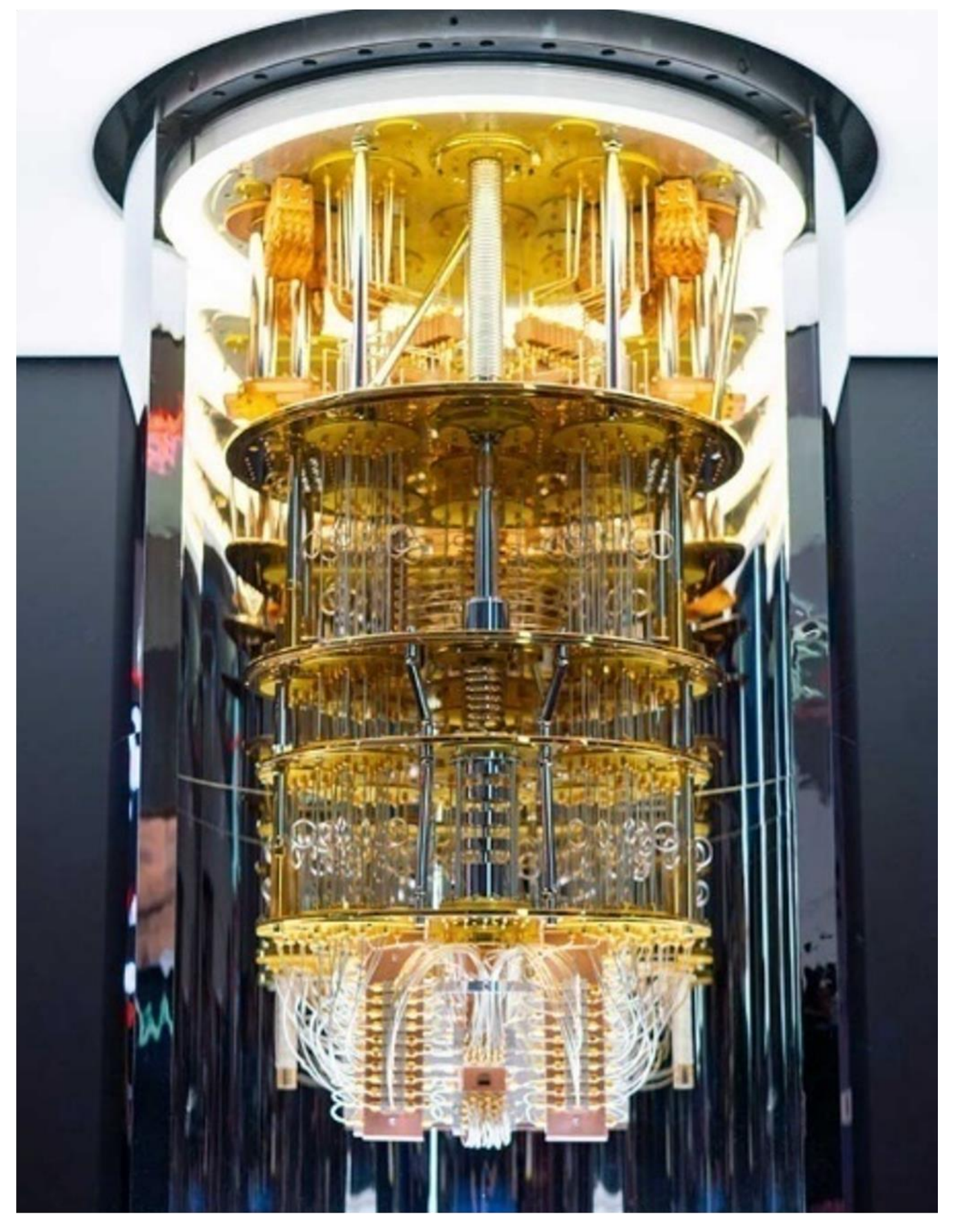
GPU Supercomputers are the foundation of Quantum R&D

Simulation

- Quantum computers are small and error-prone
  - simulation is an essential tool
- **Today:** Powerful simulators enable algorithm and application R&D – new approaches (e.g. tensor networks)
- **Future:** Digital twins of quantum computers for design and architecture optimization

HPC Quantum Integration

- Useful quantum computing will be hybrid
- **Today:** Enable domain scientists to start developing for QPUs, enable quantum researchers to use accelerated computing
- **Future:** quantum computers will integrate tightly with supercomputers as accelerators and be co-programmed

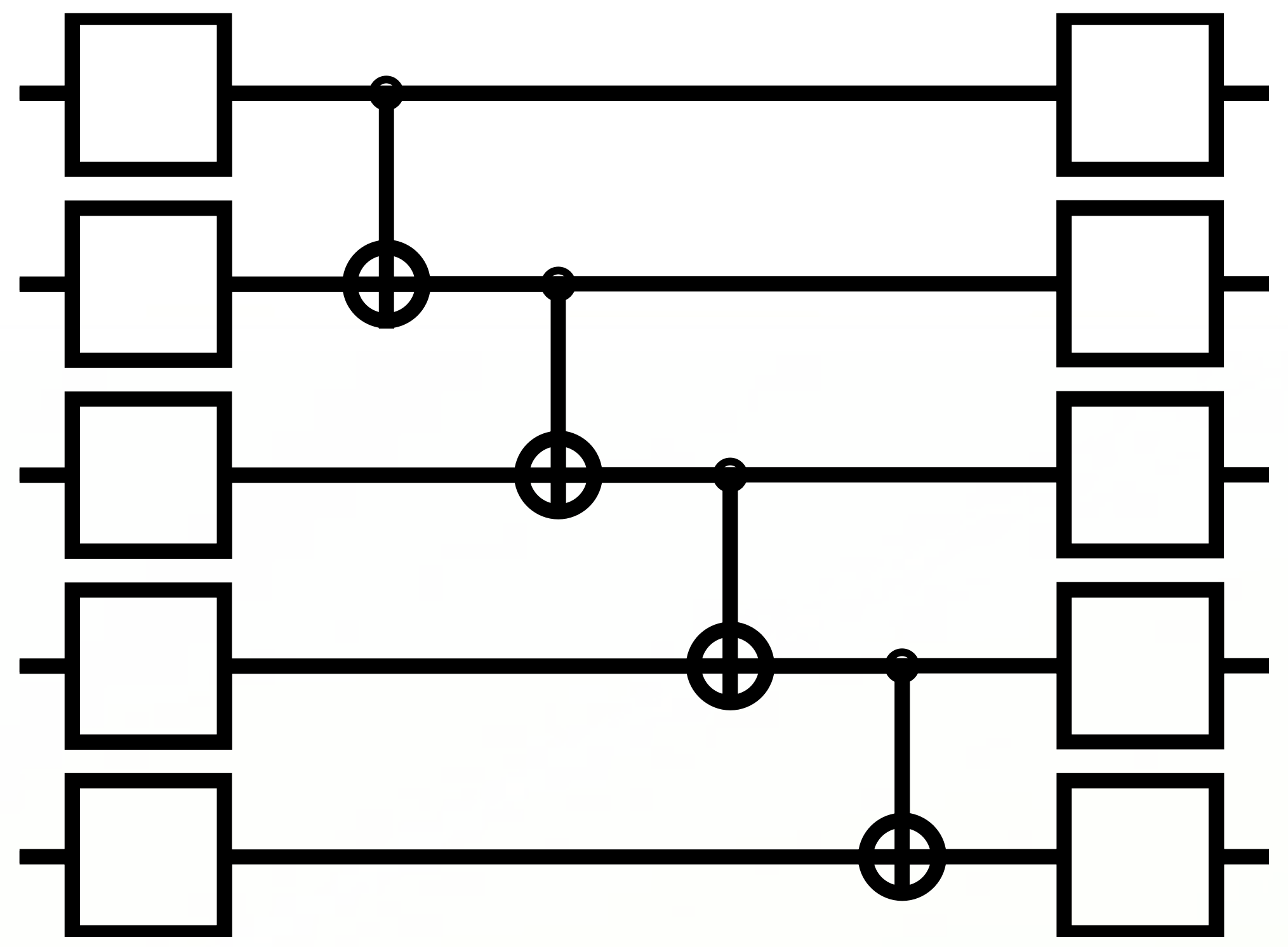


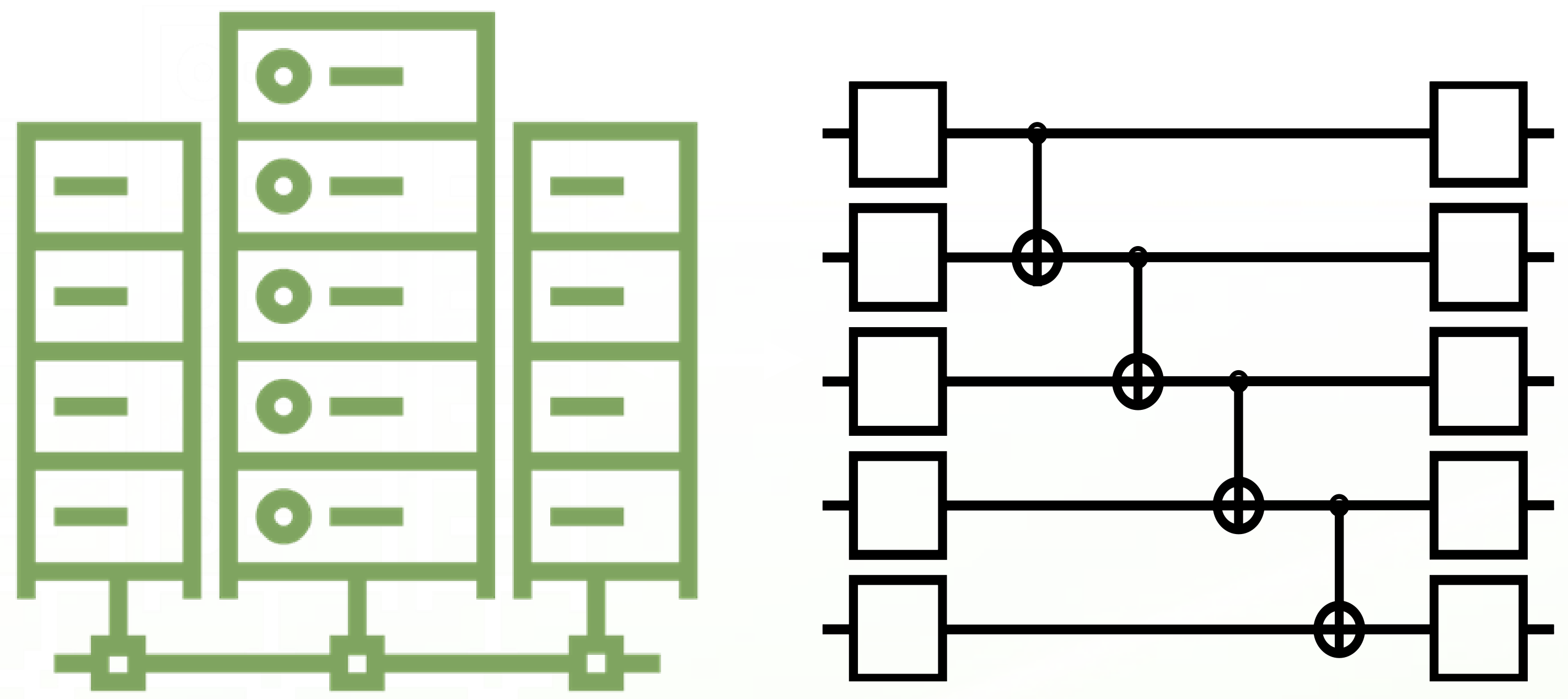
AI for Quantum

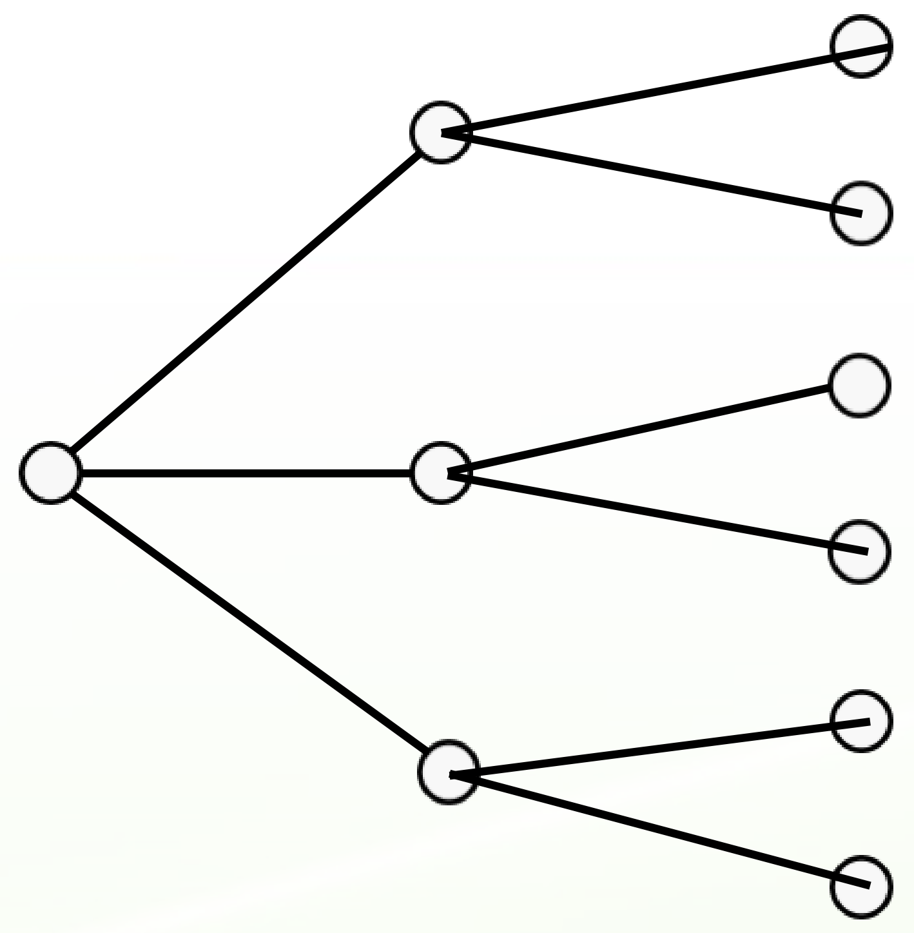
- Error correction, calibration, control, compilation are challenging computationally, real-time compute often needed
- Accelerated computing and AI can solve these problems
- **Today:** Enable AI research for all of the above
- **Future:** Hybrid Quantum+AI supercomputer with low-latency link

NVIDIA Quantum

Powering the Global Quantum Computing Community







Simulation

Algorithm Design, Resource Estimation, QPU Design

HPC Quantum Integration

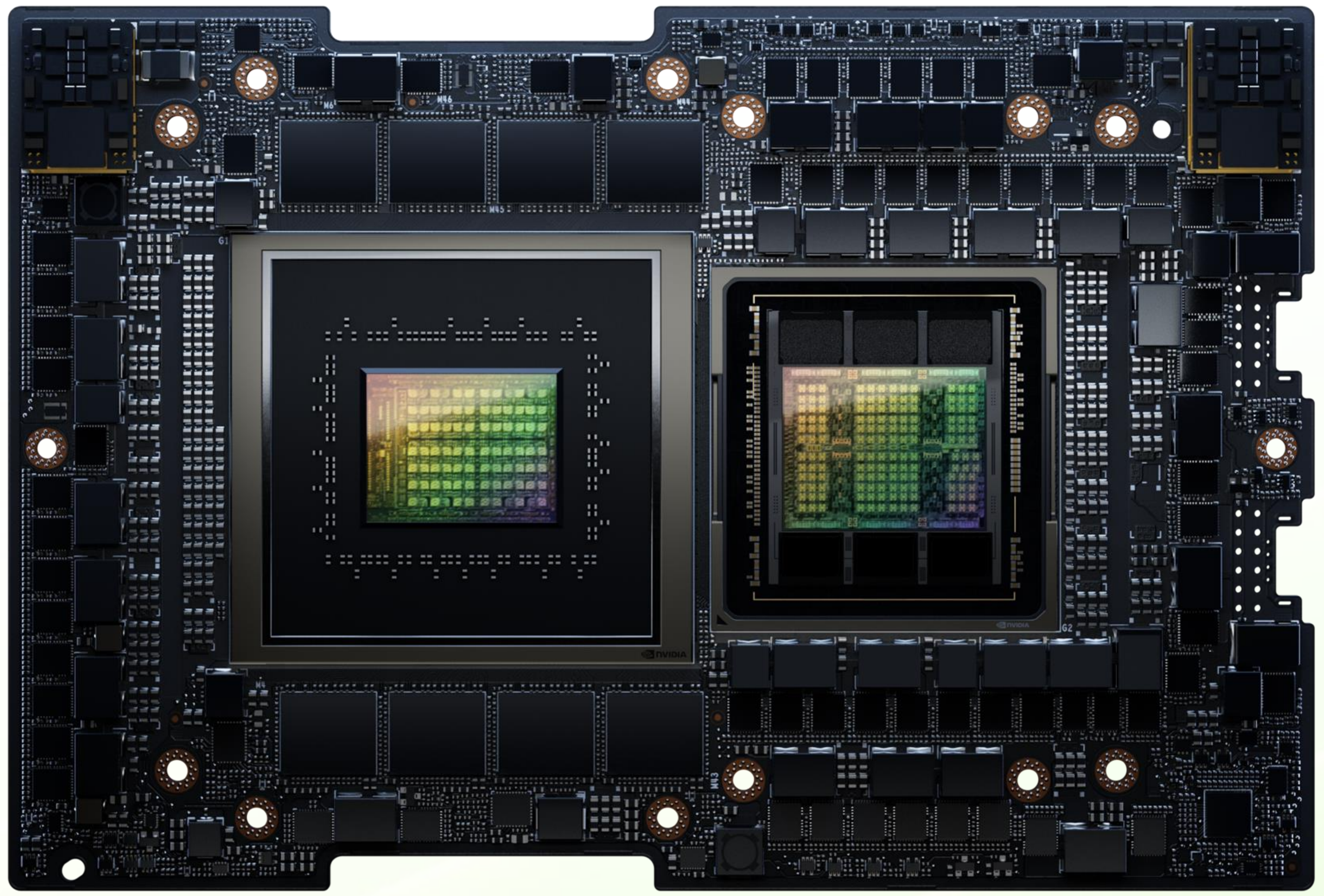
AI for Quantum

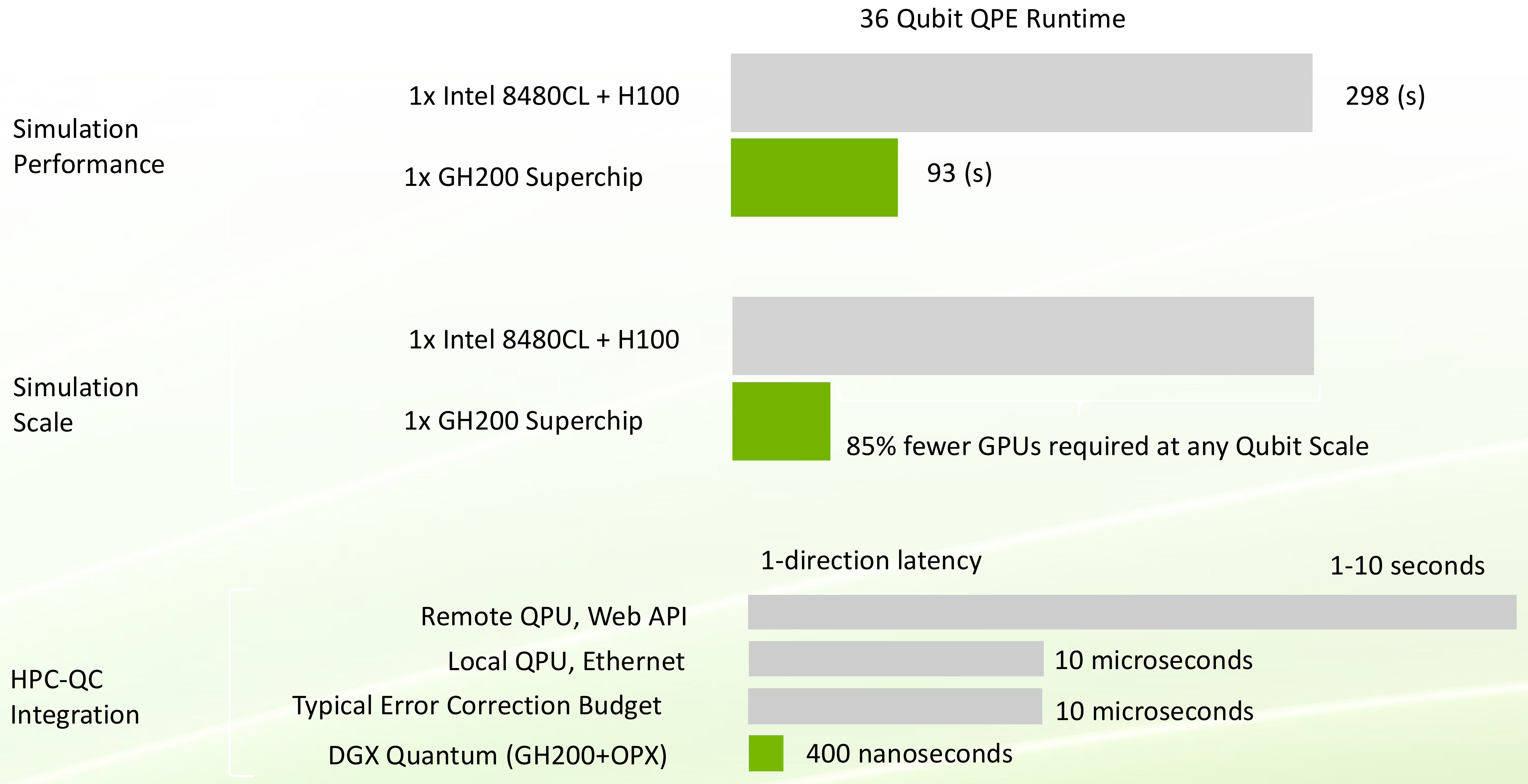
QEC, Calibration, Algorithms

CUDA-Q
Libraries
Programming Model
Tools
Infrastructure
cuQuantum
Quantum Simulation
DGX-Q
Quantum Integrated Computing
GPU Supercomputing

Grace Hopper: The Engine for Quantum Research

New Grace Hopper Deployments for Quantum Research





CUDA-Q
Enabling the QPU-accelerated GPU Supercomputer

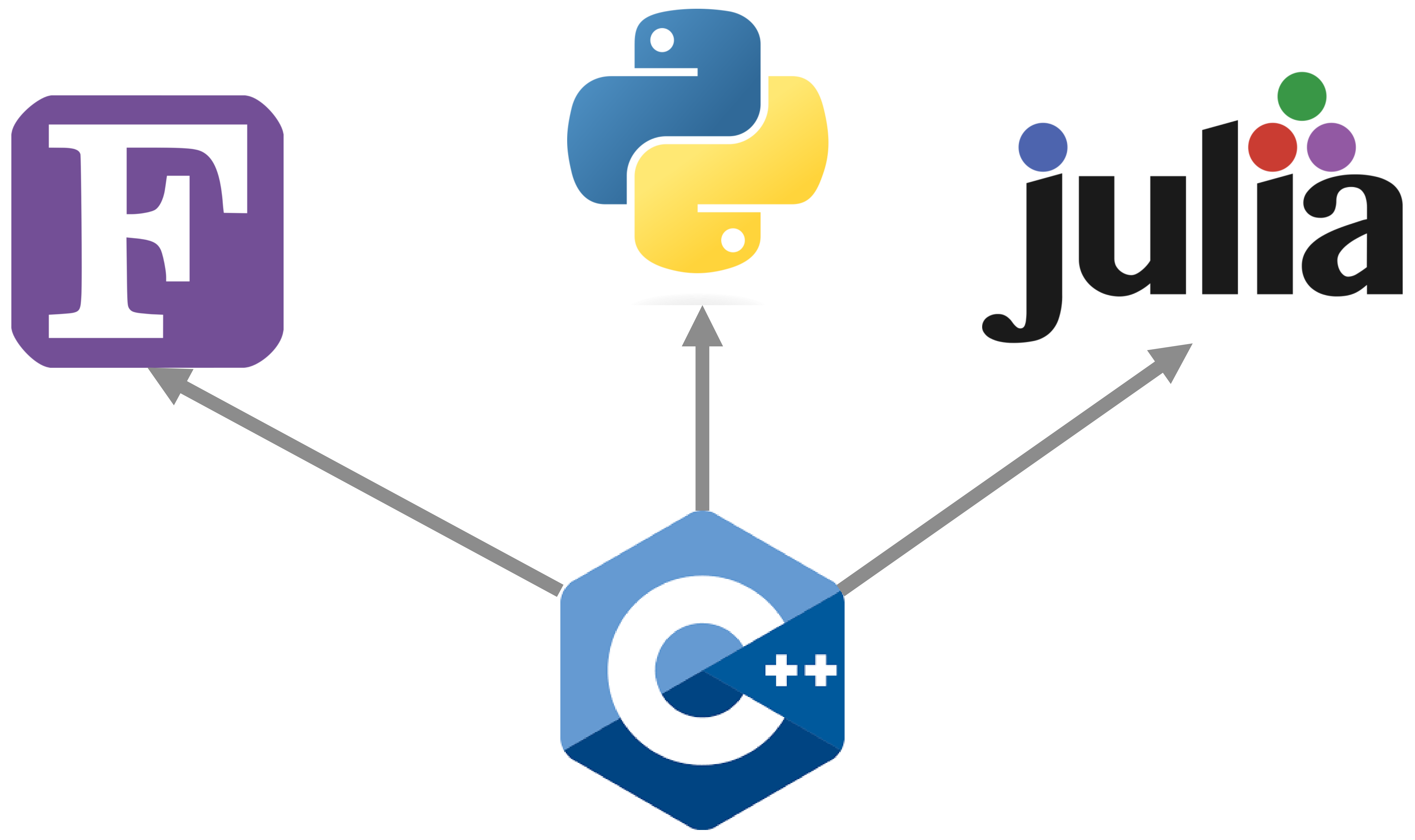
A library-based C++ language extension that compiles directly to the MLIR

Focus – requirements, programming model, architecture

Requirements for Programming the Hybrid Quantum-Classical Node

What can we learn from experience in the purely classical programming space?
How to architect something like CUDA-Q?

- Requirements
  - Performance
  - Familiar Programming Models
  - Integration with existing compilers and runtimes
- C++ as the Least Common Denominator for Programming Languages
  - Leads to optimal performance / control for developers
  - Easily bind to high-level language approaches
  - Most HPC applications are in C++ or Fortran
  - Most AI / ML frameworks are in Python, but APIs are often bound to performant C code (or JIT compiled)
  - Python user-surface is necessary, but only part of solution
- CUDA-like programming models
  - Cleanly separate device and host code
  - Direct vs library-based language extension
  - Split-compilation - map user kernel code to GPU instruction set (PTX)

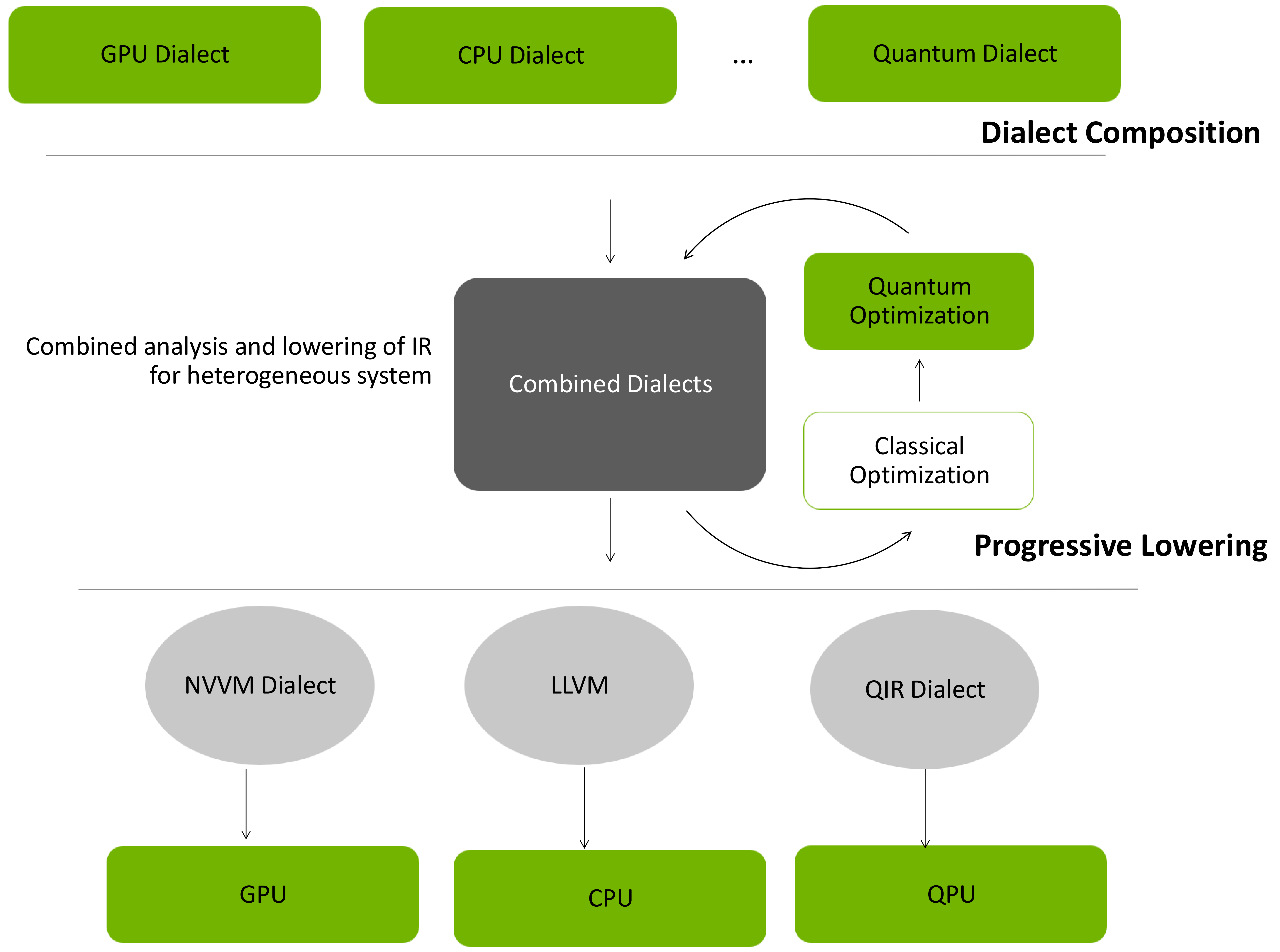


```
// Kernel functions enable clean separation of
// host and device code

__global__ void VecAdd(float* A, float* B, float* C) {
  int i = threadIdx.x;
  C[i] = A[i] + B[i];
}
int main() {
  ...
  // Invoke kernel from host code
  VecAdd<<<1, N>>>(A, B, C);
  ...
}
```

Leveraging Today's Compiler Technologies

Leverage existing state-of-the-art and enable tight quantum-classical integration at IR level

- ***Our goal should be - do not reinvent the wheel…***
- We want quantum extensions to classical
- LLVM as the gold standard…
  - Toolchain for generating executable code
  - Modular
- Control Flow is a solved problem
  - Recursive nature of the core abstractions (regions, blocks, operations)
- MLIR – Framework for creating custom compiler IRs
  - Dialects and Dialect Composition
  - Progressive Lowering
  - Control Flow
  - Optimization, Transformation, and Conversion
    - Language level abstractions



The CUDA-Q Stack

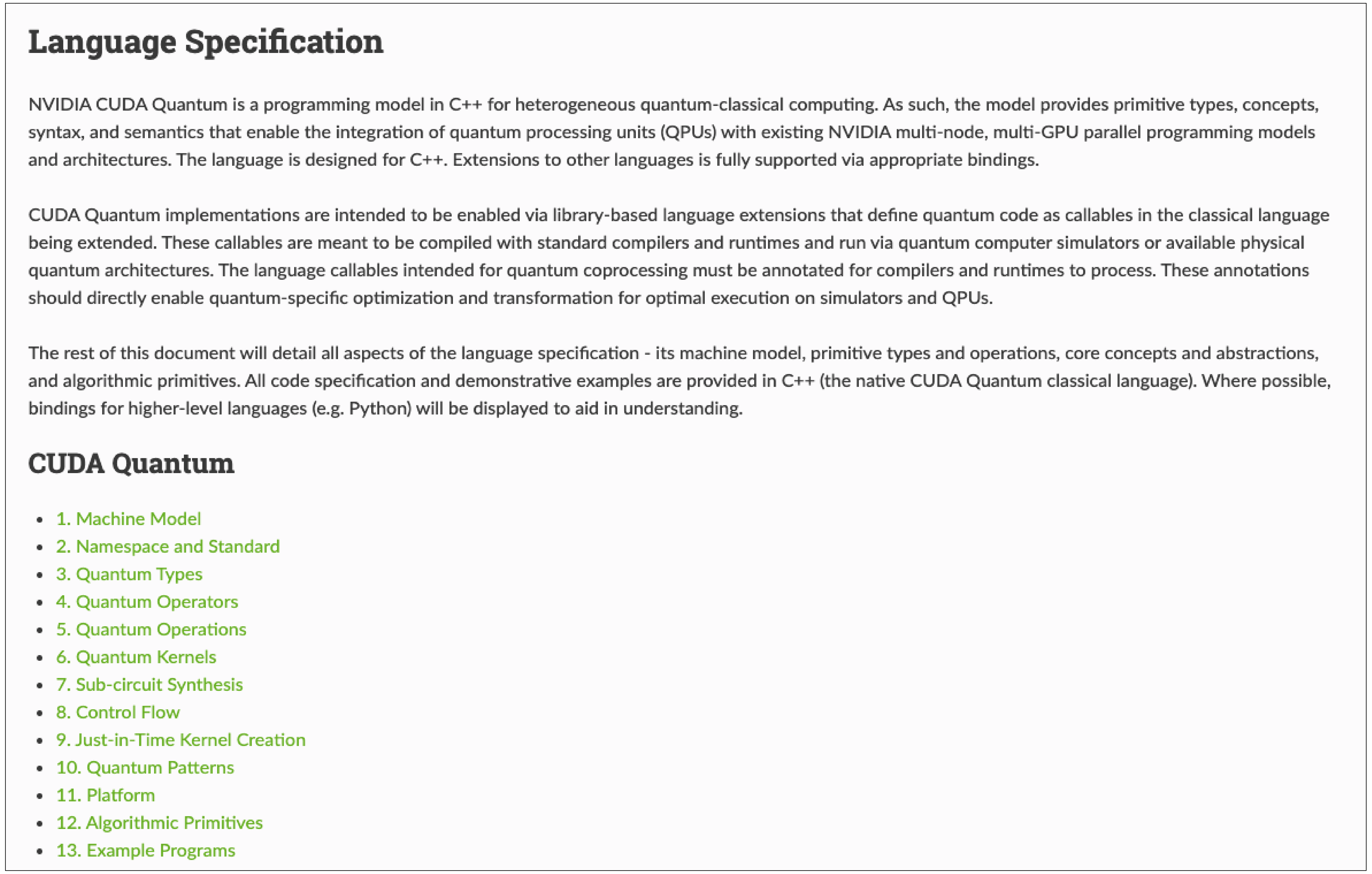
Platform for unified quantum-classical accelerated computing

Frontends
C++
Kernel Expressions
JIT Kernel Expressions
Runtime
Python
Kernel Expressions
JIT Kernel Expressions
Runtime
CUDA-Q Formal Language Specification
Compiler Platform
cudaq-quake
cudaq-opt
cudaq-translate
nvq++ driver
CUDA-Q Intermediate Representation (MLIR)
Quake
(Quantum)
CC
(Classical CFG)
Func
(Kernels)
Math
(Standard Math)
Arith
(Constants)
LLVM
(Lowering Target)
Target Low-Level IR - Quantum Intermediate Representation (QIR, Profiles, LLVM IR), QASM ISA
libnvqir.so
Simulation (MGPU, MNMG, DM, TN)
Physical QPU (Quantinuum, IonQ, IQM, OQC)

The CUDA-Q Language

Formal specification for quantum programming concepts that span classical language embeddings

- CUDA-Q starts with a general language specification.
  - Language implementations must adhere to the specification
- Meant to be classical language agnostic
  - Implementations now in C++ and Python
- Defines common concepts for quantum programming
  - Quantum types and operations
  - Quantum kernels
  - Library APIs
- This is an evolving specification.
  - We are interested in feedback and collaboration
  - There is a formal proposal process for contributions (coming soon)



Language Specification

NVIDIA CUDA Quantum is a programming model in C++ for heterogeneous quantum-classical computing. As such, the model provides primitive types, concepts, syntax, and semantics that enable the integration of quantum processing units (QPUs) with existing NVIDIA multi-node, multi-GPU parallel programming models and architectures. The language is designed for C++. Extensions to other languages is fully supported via appropriate bindings.

CUDA Quantum implementations are intended to be enabled via library-based language extensions that define quantum code as callables in the classical language being extended. These callables are meant to be compiled with standard compilers and runtimes and run via quantum computer simulators or available physical quantum architectures. The language callables intended for quantum coprocessing must be annotated for compilers and runtimes to process. These annotations should directly enable quantum-specific optimization and transformation for optimal execution on simulators and QPUs.

The rest of this document will detail all aspects of the language specification - its machine model, primitive types and operations, core concepts and abstractions, and algorithmic primitives. All code specification and demonstrative examples are provided in C++ (the native CUDA Quantum classical language). Where possible, bindings for higher-level languages (e.g. Python) will be displayed to aid in understanding.

CUDA Quantum

https://nvidia.github.io/cuda-quantum/latest/specification/cudaq.html

Next: let's dive into the specifics of the language…

GHZ State Example

Running on CPU

```python
import cudaq

@cudaq.kernel
def ghz_state(N: int):
    qubits = cudaq.qvector(N)
    h(qubits[0])
    for i in range(N - 1):
        x.ctrl(qubits[i], qubits[i + 1])
    mz(qubits)
```

```python
cudaq.set_target("qpp-cpu")
n = 3
print("Preparing GHZ state for", n,
"qubits.")
counts = cudaq.sample(ghz_state,n)
counts.dump()
```

```
Output:
Preparing GHZ state for 3 qubits.
{ 000:494 111:506 }
```

GHZ State Example

Running on GPU

```python
import cudaq

@cudaq.kernel
def ghz_state(N: int):
    qubits = cudaq.qvector(N)
    h(qubits[0])
    for i in range(N - 1):
        x.ctrl(qubits[i], qubits[i + 1])
    mz(qubits)
```

```python
cudaq.set_target(“nvidia")
n = 29
print("Preparing GHZ state for", n,
"qubits.")
counts = cudaq.sample(ghz_state,n)
counts.dump()
```

```
Output:
Preparing GHZ state for 29 qubits.
{ 00000000000000000000000000000:509
11111111111111111111111111111:491 }
```

What is a CUDA-Q Kernel?

Cleanly separate host code from quantum device code

```cpp
struct phase_estimation {
  template <typename StatePrep, typename Unitary>
  void operator()(int numCountingQubits, int numStateQubits,
                  StatePrep && statePrep,
                  Unitary && oracle) __qpu__ {
    // Allocate a register of qubits
    cudaq::qvector q(numCountingQubits + numStateQubits);

    // Extract sub-registers, one for the counting qubits
    // another for the eigenstate register
    auto& countingQubits = q.front(numCountingQubits);
    auto& stateRegister = q.back(numStateQubits);

    // Prepare the eigenstate
    statePrep(stateRegister);

    // Put the counting register into uniform superposition
    h(countingQubits);

    // Perform `ctrl-U^j`
    for (int i = 0; i < numCountingQubits; ++i)
      for (int j = 0; j < (1 << i); ++j)
        cudaq::control(oracle, countingQubits[i],
                       stateRegister);

    // Apply inverse quantum Fourier transform
    iqft(countingQubits);

    // Measure to gather sampling statistics
    mz(countingQubits);
  }
};
```

C++

```python
@cudaq.kernel
def phase_estimation(numCounting: int, numQubits : int,
       statePrep : Callable[[cudaq.qview], None],
       oracle : Callable[[cudaq.qview], None]):

    # Allocate a register of qubits
    q = cudaq.qvector(numCounting + numQubits)

    # Extract sub-registers, one for the counting qubits
    # another for the eigenstate register
    countingQubits = q.front(numCounting)
    stateRegister = q.back(numQubits)

    # Prepare the eigenstate
    statePrep(stateRegister)

    # Put the counting register into uniform superposition
    h(countingQubits)

    # Perform `ctrl-U^j`
    for i in range(numCounting):
        for j in range(2**i):
            cudaq.control(oracle, countingQubits[i],
                          stateRegister)

    # Apply inverse quantum Fourier transform
    iqft(countingQubits);

    # Measure to gather sampling statistics
    mz(countingQubits);
```

Python

Any callable in the language

Annotated in some way

Quantum memory management

Kernels are composable

Control flow inherited from classical language

Automated multi-control synthesis

Enable generic application libraries

Ultimately a kernel defines a template for quantum circuits. Circuits are concretized / synthesized with runtime information.

What is a CUDA-Q Kernel?

Cleanly separate host code from quantum device code

```cpp
__qpu__ double RWPE(int N, double mu, double sigma) {
  cudaq::qubit q, r;
  x(q);
  while (i < N) { // while loops available
    h(q);
    rz(1-(mu / sigma), q);
    rz(.25 / sigma, r);
    x<cudaq::ctrl>(q, r);
    rz(-.25 / sigma, r);
    x<cudaq::ctrl>(q, r);
    h(q);
    if (mz(q)) { // Condition code on qubit measurements
      x(q);
      mu += sigma * .6065;
    } else {
      mu -= sigma * .6065;
    }

    sigma *= .7951;
    i++;
  }
  return 2 * mu;
}
```

```python
@cudaq.kernel
def RWPE(N: int, mu : float, sigma : float) -> float:
    q, r = cudaq.qubit(), cudaq.qubit()
    x(q)
    while i < N:
        h(q)
        rz(1.-(mu / sigma), q)
        rz(.25 / sigma, r)
        x.ctrl(q, r)
        rz(-.25 / sigma, r)
        x.ctrl(q, r)
        h(q)
        if mz(q): // Condition code on qubit measurements
            x(q)
            mu += sigma * .6065
        else:
            mu -= sigma * .6065

        sigma *= .7951
        i += 1

    return 2 * mu;
```

Dynamic circuits (control dictated by qubit measurement results) and user-defined return types fit into this model too.

Quantum Operations and Control Modifiers

Single qubit operations and compiler-synthesized controlled operations

- Quantum operations are unique functions that take a qubit reference and optional floating-point parameters.
  - CUDA-Q starts with a specification of logical single-target quantum operations
- General multi-control operations are expressed via ***modification*** of these single-target operations
  - Control operations expressed with `cudaq::ctrl` modifier
  - Multi-qubit operations are synthesized by the compiler
  - Control qubits are first `N-1` qubit arguments
  - Control qubits can be also negated with `operator!()`
- Entire kernel expressions can be controlled with `cudaq::control(...)`
- Adjoint kernels can be synthesized with `cudaq::adjoint(...)`

```
__qpu__ void cnotKernel(cudaq::qubit& q, cudaq::qubit& r) {
  x<cudaq::ctrl>(q, r);
}

__qpu__ void toBeAdjointed(cudaq::qubit& q, cudaq::qubit& r) {
  h(q);
  x(r);
  x<cudaq::ctrl>(q, r);
  ry(-M_PI_2, q);
}

__qpu__ void multipleWaysToExpressToffoli(std::size_t N) {
  {
    cudaq::qarray<3> q;

    // Toffoli
    x<cudaq::ctrl>(q[0], q[1], q[2]);
  } // qubits deallocated at scope exit
  {
    cudaq::qvector q(N);

    // Toffoli
    cudaq::control(cnotKernel, {q[0]}, q[1], q[2]);
  }

__qpu__ void showNegationAndAdjoint() {
    cudaq::qvector q(2);

    h<cudaq::ctrl>(!q[0], q[1]); // negated control

    // adjoint modifier
    t<cudaq::adj>(q[0]);
    // compiler synthesizes the adjoint
    cudaq::adjoint(toBeAdjointed, q[0], r[1]);
}
```

Quantum Operations and Control Modifiers

Single qubit operations and compiler-synthesized controlled operations

- Quantum operations are unique functions that take a qubit reference and optional floating-point parameters.
  - CUDA-Q starts with a specification of logical single-target quantum operations
- General multi-control operations are expressed via ***modification*** of these single-target operations
  - Control operations expressed with `cudaq::ctrl` modifier
  - Multi-qubit operations are synthesized by the compiler
  - Control qubits are first `N-1` qubit arguments
  - Control qubits can be also negated with `operator!()`
- Entire kernel expressions can be controlled with `cudaq::control(...)`
- Adjoint kernels can be synthesized with `cudaq::adjoint(...)`

```
@cudaq.kernel
def cnotKernel(q : cudaq.qubit, r : cudaq.qubit):
    x.ctrl(q, r)

@cudaq.kernel
def toBeAdjointed(q : cudaq.qubit, r : cudaq.qubit):
    h(q)
    x(r)
    x.ctrl(q, r)
    ry(-np.pi / 2, q)

@cudaq.kernel
def multipleWaysToExpressToffoli():
    q = cudaq.qvector(3)

    // Toffoli
    x.ctrl(q[0], q[1], q[2]);

    # Toffoli
    cudaq.control(cnotKernel, [q[0]], q[1], q[2]);

@cudaq.kernel
def showNegationAndAdjoint(N : int):
    q = cudaq.qvector(N);

    h.ctrl(!q[0], q[1]); // negated control

    # adjoint modifier
    t.adj(q[0])
    // compiler synthesizes the adjoint
    cudaq.adjoint(toBeAdjointed, q[0], r[1]);
```

CUDA-Q Platform and Asynchronous Execution

Expose the underlying system architecture to the programmer

- The system architecture model considers access to multiple quantum accelerators
- CUDA-Q provides programmatic access to this configuration via the **`quantum_platform`**
- Exposes a native platform that models a virtual QPU for every CUDA device.
- Each CUDA device gets a cuQuantum-based simulator
- Enables experimentation with distributed quantum computing and further simulation scalability
- ***Targets***: CUDA-Q programs are compiled to specific targets which define the multi-QPU granularity (or the physical QPU characteristics).

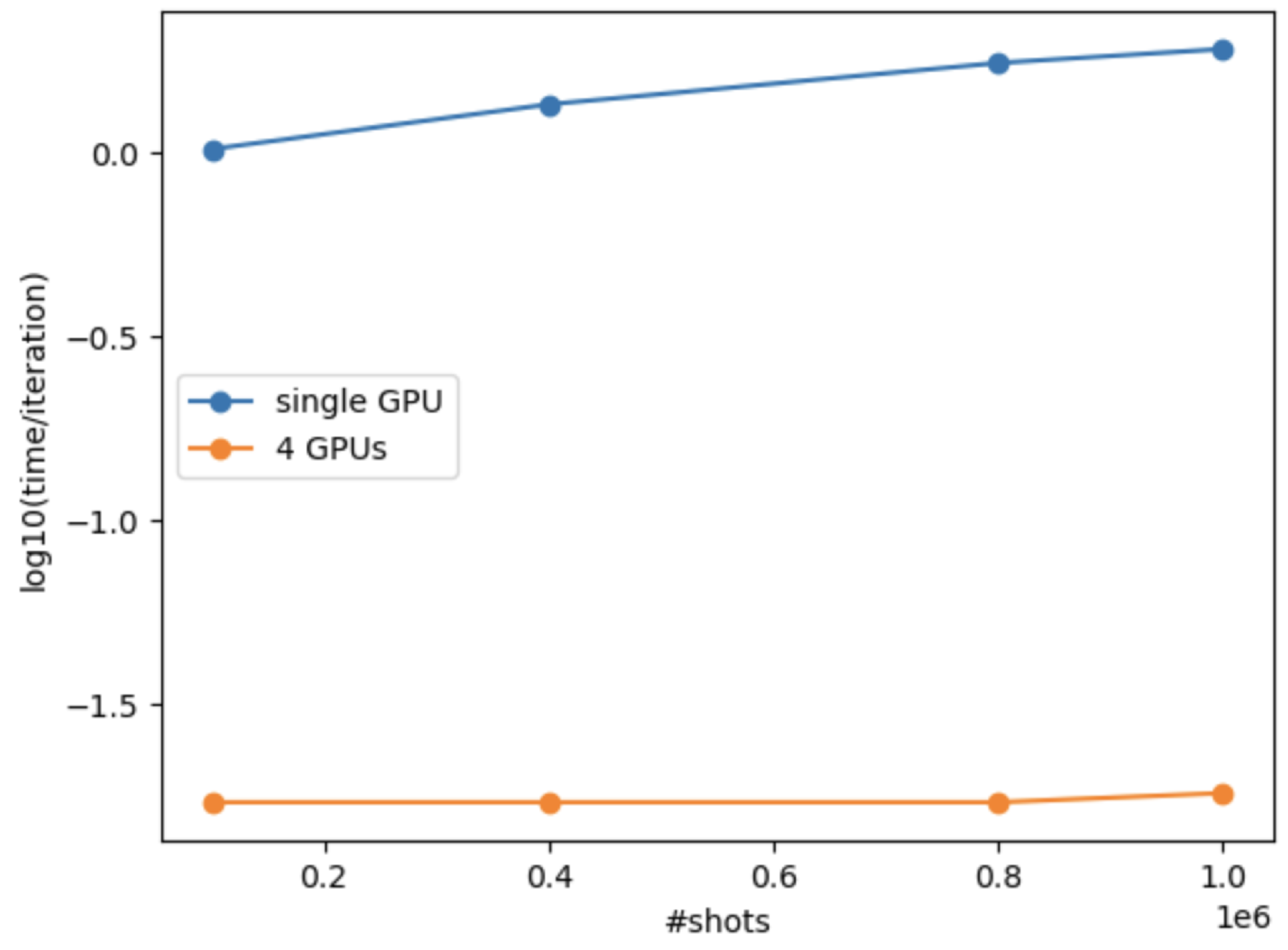
```
// Programmer can query info about the platform
auto& platform = cudaq::get_platform();

// Get number of QPUs available
auto numQpus = platform.num_qpus();

// Get the number of qubits on QPU 1
auto nQ1 = platform.get_num_qubits(1)

// Get QPU 0 connectivity.
auto connectivity = platform.get_connectivity(0);

// Async task execution on available QPUs
std::vector<std::future<double>> subs;
for (auto qpuIdx : cudaq::range(numQpus))
  subs.emplace_back(cudaq::my_async_task(qpuIdx, ...));

auto sum = stdr::reduce(std::execution::par,
                        cudaq::when_all(subs), 0.0);
```

Node 0, MPI Rank 0
vQPU (cuQuantum)
H100
vQPU (cuQuantum)
H100
...
vQPU (cuQuantum)
H100
Node 1, MPI Rank 1
vQPU (cuQuantum)
H100
vQPU (cuQuantum)
H100
...
vQPU (cuQuantum)
H100

CUDA-Q Platform and Asynchronous Execution

Expose the underlying system architecture to the programmer

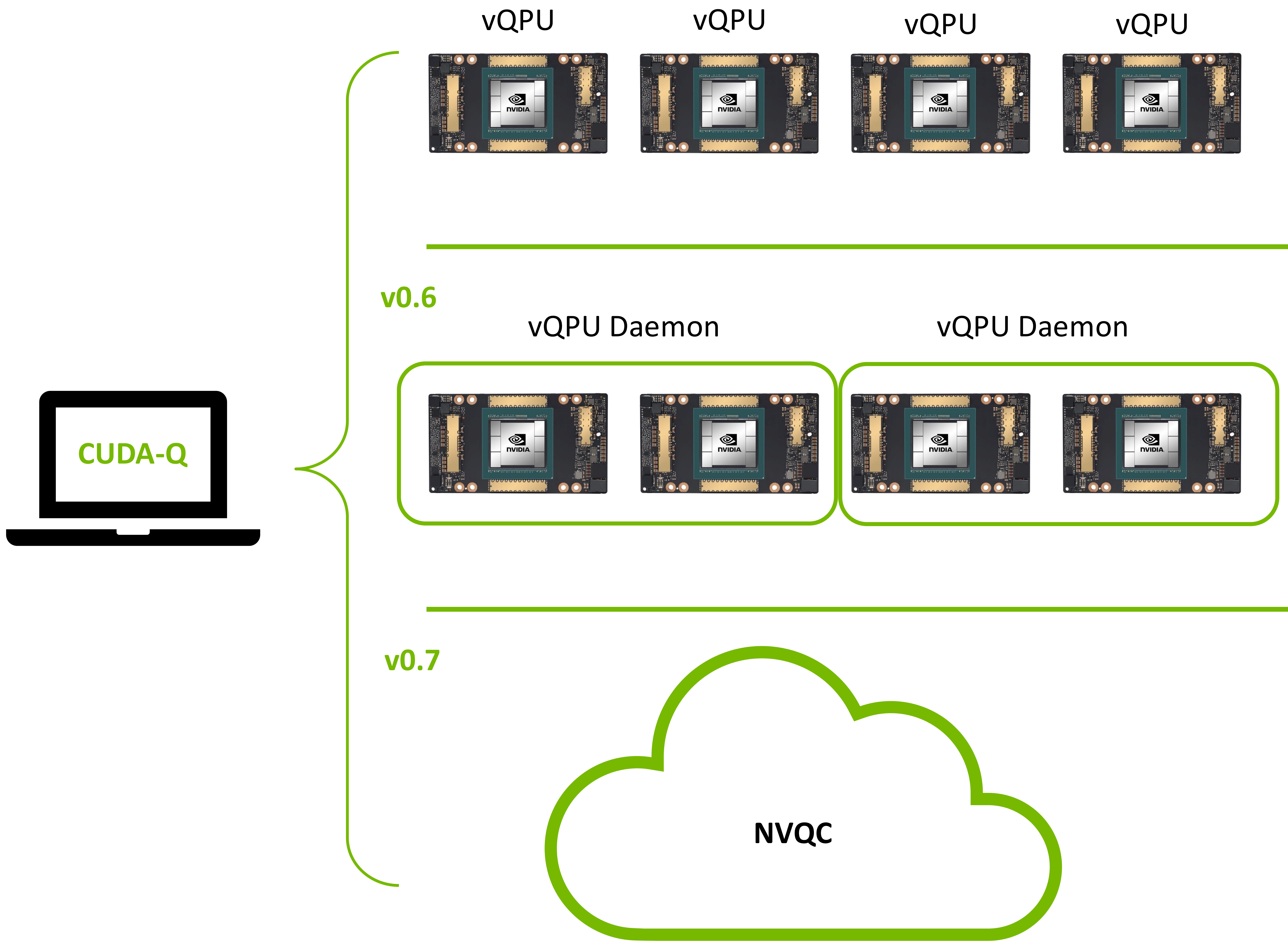
- Example use-case – Sampling a Quantum Restricted Boltzmann Machine model
- Find the ground state wavefunction for electronic Hamiltonian
- Find ground state wavefunction for a Transverse Field Ising (TFS) model and Antiferromagnetic Heisenberg model (AFH)
- Anomaly detection (DOI: https://doi.org/10.1038/s42005-023-01390-y)
- Cybersecurity (arXiv:2011.13996)



Node 0, MPI Rank 0
vQPU (id=0)
H100
vQPU (id=1)
H100
...
vQPU (id=N-1)
H100
Node 1, MPI Rank 1
vQPU (id=0)
H100
vQPU (id=1)
H100
...
vQPU (id=N-1)
H100

CUDA-Q Platform and Asynchronous Execution

Expose the underlying system architecture to the programmer



Local Virtualization (**nvidia-mqpu**)

- Thread-based (NVIDIA device id) or MPI-based resource isolation.
- Single GPU per vQPU.
- Performant (minimal latency/overhead)

Localhost Virtualization (**remote-mqpu**)

- Remote client-server architecture.
- Daemon processes (cudaq-qpud) = virtual QPUs.
- HTTP (REST API) client-qpud communication.
- Flexible GPU resource allocation and isolation → enabling multi-GPU vQPU.
- Communication overhead.

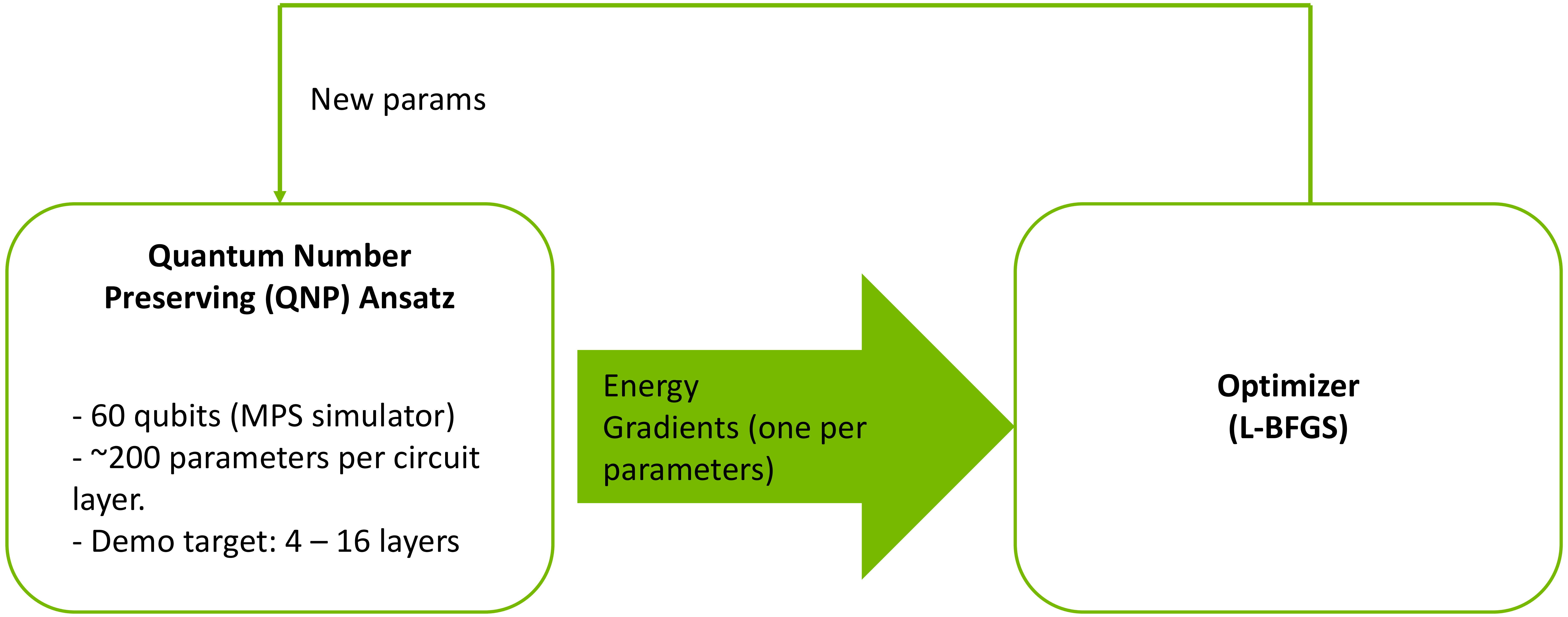
Serverless Virtualization (**nvqc**)

- Remote client-server architecture.
- Similar to remote-mqpu with fully-managed server services.
- Pre-configured daemon services, e.g., 1 GPU, 4 GPUs, or 8 GPUs/vQPU.
- On-demand, unlimited* scalability (number of vQPUs)

* theoretically, depending on NVCF auto-scaling capabilities

CUDA-Q Platform and Asynchronous Execution

Expose the underlying system architecture to the programmer


New params
Quantum Number Preserving (QNP) Ansatz
- 60 qubits (MPS simulator)
- ~200 parameters per circuit layer.
- Demo target: 4 – 16 layers
Energy Gradients (one per parameters)
Optimizer (L-BFGS)

CUDA-Q Solution:

- Parameter-shift gradient calculation: one parameter ➔ 2 circuits (+/- delta).

- Task-based virtualization: each iteration ➔ 1 + 2 * num_params tasks (each task is a circuit with different rotation angles).

- vQPUs grab tasks from the task pool.

- Executed on EOS: 59 nodes – 472 GPUs (each vQPU, aka GPU, only needs to run one task).

- Optimizer check-pointing to fit EOS wall-time limit.

EOS

CUDA-Q Platform and Asynchronous Execution

Expose the underlying system architecture to the programmer

```
// Toy model: H2
cudaq::spin_op H(h2_data, /*nQubits*/ 4);
auto ansatz = [](std::vector<double> thetas) __qpu__ {
  … Quantum Number Preserving kernel.
  qnp_fabric(q, numLayers, thetas);
};

// Get the number of virtual QPUs.
auto numQpus = platform.num_qpus();
auto objectiveDistributeGradients = [&](const std::vector<double> &x,
                                        std::vector<double> &dx) -> double {
  std::vector<std::vector<double>> allParams;
  // Task lists: all parameter sets
  allParams.reserve(2 * x.size() + 1);
  allParams.emplace_back(x);
  for (std::size_t paramId = 0; paramId < x.size(); ++paramId) {
    … Shift params up and down
  }

  // Splice the task list (for QPUs)
  const std::vector<std::vector<std::vector<double>>> paramsForQpus =
      splitVec(allParams, std::min(numQpus, allParams.size()));
 …
  std::vector<cudaq::async_observe_result> expectation_futures;
  // Async. dispatch
  for (std::size_t qpuId = 0; qpuId < numQpus; ++qpuId)
    for (const auto &param : paramsForQpus[qpuId])
      expectation_futures.emplace_back(
          cudaq::observe_async(qpuId, ansatz, H, param));

 … Do something else

  // Retrieve results
  std::vector<double> allData;
  for (auto &fut : expectation_futures)
    allData.emplace_back(fut.get().expectation());

  // Return the <H> energy
  const double energy = allData[0];
  for (std::size_t paramId = 0; paramId < x.size(); ++paramId)
    … Compute the gradients
    dx[paramId] = (shiftPlus - shiftMinus) / 2.0;
  ….
  return energy;
};
```

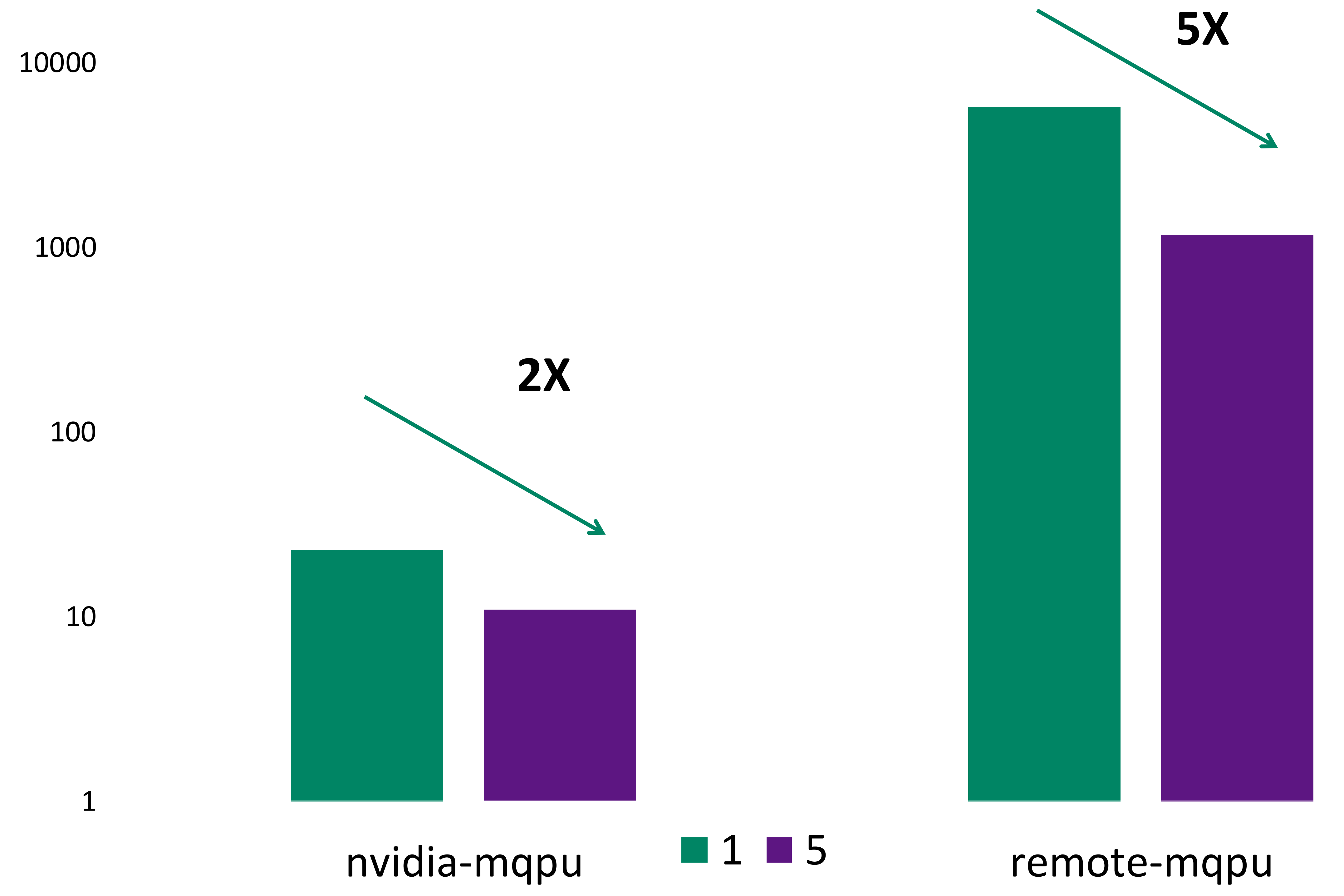
Task List

Async. Dispatch to vQPUs

Gather Data and Compute Gradients

- Platform agnostic programming (vQPU programming model)
- Interchange backend, scale resources up and down

```
> n
```

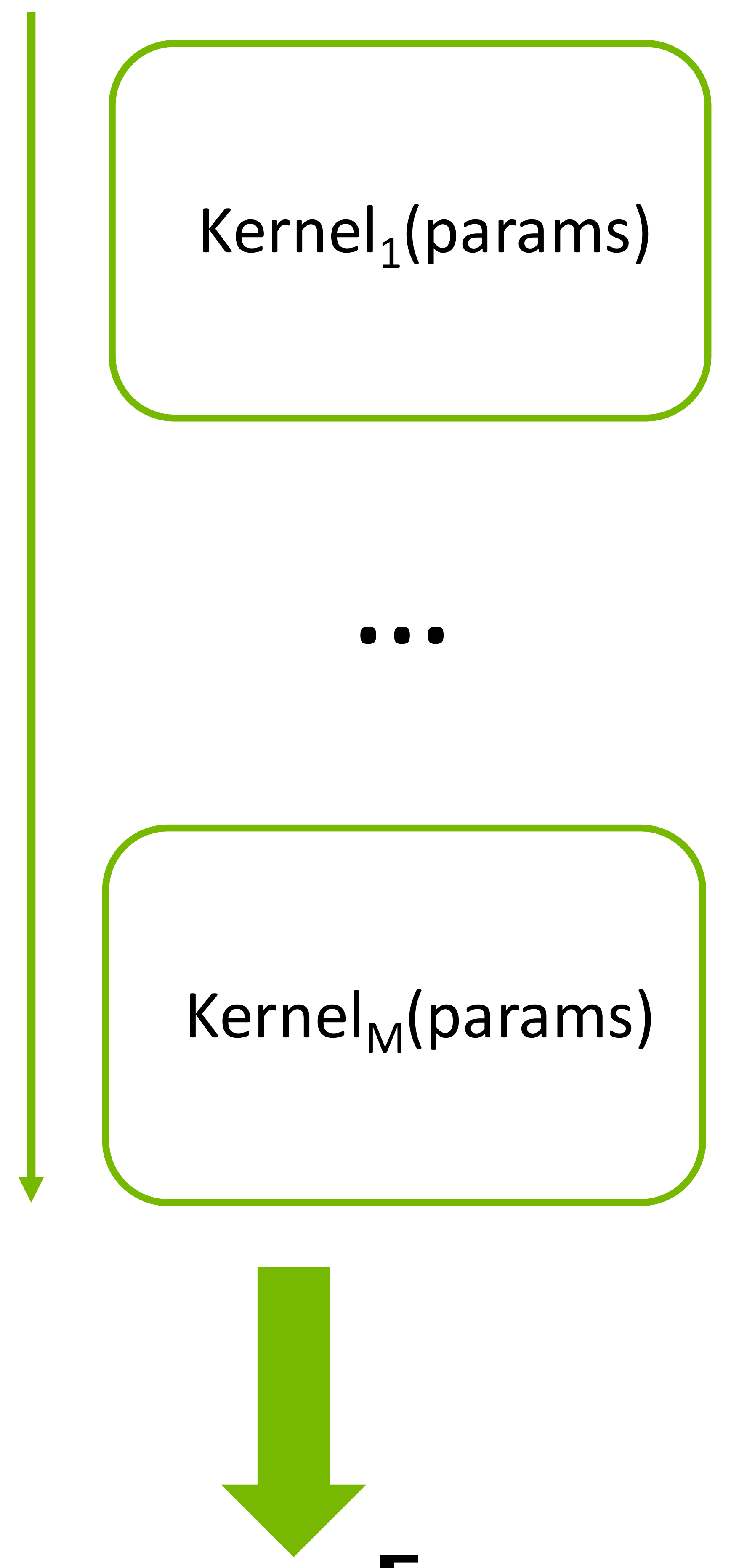


CUDA-Q Platform and Asynchronous Execution

Expose the underlying system architecture to the programmer

Quantum Machine Learning use case

M Quantum Kernels (data samples)



$$Gram's\ Matrix = \begin{bmatrix} \left|\langle \Psi_i | \Psi_j \rangle\right|^2 & \cdots & \\ \vdots & \ddots & \vdots \\ & \cdots & \end{bmatrix}$$

CUDA-Q Solution:

(1) Python-based mpi4py parallelization (user code)

(2) Direct access to GPU memory (tensor network state representation) (**v0.8**)

(3) CUDA integration: cupy and CUDA-aware MPI (data movement)

(1)

```python
import cudaq
import cupy as cp
import mpi4py
from mpi4py import MPI

comm = MPI.COMM_WORLD

# distribute the parameters across the available vQPU
param_vals_split = comm.scatter(param_vals, root = 0)
states_split = comm.scatter(states, root = 0)
```

(2)

```python
# quantum computation (aka. get_state)
for i in range(param_vals_split.shape[0]):
    states_split[i] = cudaq.get_state(kernel, param_vals_split[i])
```

(3)

```python
# gather the results from the different ranks
results = comm.gather(states_split, root = 0)

# Post-processing to calculate the gram matrix which is the input to SVM

if rank == 0:
    kets = cp.concatenate(results)
    bras = cp.transpose(cp.conj(kets))
    gram_matrix = cp.zeros((m,m))

    for i in range(m):
        for j in range(m):
            ith_bra = bras[:,i].reshape(1,-1)
            jth_ket = kets[j].reshape(-1,1)
            gram_matrix[i][j] = np.abs(jth_ket.overlap(ith_bra))**2
```

CUDA-Q Compiler Platform

The compiler is composed of MLIR dialects and a set of discrete tools

The CUDA-Q Compiler Stack

Platform for unified quantum-classical accelerated computing

Compiler Platform
cudaq-quake
cudaq-opt
cudaq-translate
nvq++ driver
CUDA-Q Intermediate Representation (MLIR)
Quake
(Quantum)
CC
(Classical CFG)
Func
(Kernels)
Math
(Standard Math)
Arith
(Constants)
LLVM
(Lowering Target)
Target Low-Level IR - Quantum Intermediate Representation (QIR, Profiles, LLVM IR), QASM ISA

CUDA-Q provides a collection of tools that enables the compilation of language representations to external representations like the QIR.

The `nvq++` driver orchestrates this collection of these tools to produce hybrid quantum-classical executables and library code.

CUDA-Q targets define the architecture and native gate set for the specific QPU, as well as whether it is physical or emulated.

```
$ nvq++ -o simulatedExample.x example.cpp
$ ./simulatedExample.x

$ nvq++ -o gpuAccelerated.x example.cpp --target nvidia –I path/includes
$ ./gpuAccelerated.x

$ nvq++ -o noisyExample.x example.cpp --target density-matrix-gpu
$ ./noisyExample.x

$ nvq++ -o emulateQuantinuum.x example.cpp --target quantinuum --emulate
$ ./emulateQuantinuum.x

$ nvq++ -o quantinuumH2.x example.cpp --target quantinuum
$ ./quantinuumH2.x
```

The CUDA-Q Compiler Stack

Platform for unified quantum-classical accelerated computing

Compiler Platform

- cudaq-quake
- cudaq-opt
- cudaq-translate
- nvq++ driver

CUDA-Q Intermediate Representation (MLIR)

Quake	CC	Func	Math	Arith	LLVM
(Quantum)	(Classical CFG)	(Kernels)	(Standard Math)	(Constants)	(Lowering Target)

Target Low-Level IR - Quantum Intermediate Representation (QIR, Profiles, LLVM IR), QASM ISA

```
__qpu__ double kernel(int N,
         std::vector<double> params) {
   int another = 5;
   return params[0] + params[1];
}
```

```
module {
 func.func @kernel(%arg0: i32, %arg1: !cc.stdvec<f64>) -> f64 {
  %c5_i32 = arith.constant 5 : i32
  %0 = cc.alloca i32
  cc.store %arg0, %0 : !cc.ptr<i32>
  %1 = cc.alloca i32
  cc.store %c5_i32, %1 : !cc.ptr<i32>
  %2 = cc.stdvec_data %arg1 : (!cc.stdvec<f64>) -> !cc.ptr<f64>
  %3 = cc.compute_ptr %2[0] : (!cc.ptr<f64>) -> !cc.ptr<f64>
  %4 = cc.load %3 : !cc.ptr<f64>
  %5 = cc.stdvec_data %arg1 : (!cc.stdvec<f64>) -> !cc.ptr<f64>
  %6 = cc.compute_ptr %5[1] : (!cc.ptr<f64>) -> !cc.ptr<f64>
  %7 = cc.load %6 : !cc.ptr<f64>
  %8 = arith.addf %4, %7 : f64
  return %8 : f64
 }
}
```

- CC Dialect (Classical Computing)
  - Model C++ types and behavior
  - Control flow
  - `Span-like` types, callables, variables
  - Loop normalization and unrolling, lambda lifting, mem2reg, reg2mem, lower to LLVM CFG
  - This dialect will grow over time to support more and more of C++

The CUDA-Q Compiler Stack

Platform for unified quantum-classical accelerated computing

Compiler Platform
cudaq-quake
cudaq-opt
cudaq-translate
nvq++ driver
CUDA-Q Intermediate Representation (MLIR)
Quake
(Quantum)
CC
(Classical CFG)
Func
(Kernels)
Math
(Standard Math)
Arith
(Constants)
LLVM
(Lowering Target)
Target Low-Level IR - Quantum Intermediate Representation (QIR, Profiles, LLVM IR), QASM ISA

```
__qpu__ double kernel(int N,
       std::vector<double> params) {
   int another = 5;
   return params[0] + params[1];
}
```

```
module {
 func.func @kernel(%arg0: i32, %arg1: !cc.stdvec<f64>) -> f64 {
  %c5_i32 = arith.constant 5 : i32
  %0 = cc.alloca i32
  cc.store %arg0, %0 : !cc.ptr<i32>
  %1 = cc.alloca i32
  cc.store %c5_i32, %1 : !cc.ptr<i32>
  %2 = cc.stdvec_data %arg1 : (!cc.stdvec<f64>) -> !cc.ptr<f64>
  %3 = cc.compute_ptr %2[0] : (!cc.ptr<f64>) -> !cc.ptr<f64>
  %4 = cc.load %3 : !cc.ptr<f64>
  %5 = cc.stdvec_data %arg1 : (!cc.stdvec<f64>) -> !cc.ptr<f64>
  %6 = cc.compute_ptr %5[1] : (!cc.ptr<f64>) -> !cc.ptr<f64>
  %7 = cc.load %6 : !cc.ptr<f64>
  %8 = arith.addf %4, %7 : f64                  Reuse other dialects
  return %8 : f64
 }
}
```

- MLIR Dialect Reuse
  - Leverage the work from the community
  - Functions, Math and Constants, and the LLVM Dialects
- Optimizations from the community
  - Function inlining, canonicalization, common subexpression elimination
- Lower to the QIR in MLIR before translating MLIR to LLVM IR (also get that for free)

The CUDA-Q Compiler Stack

Platform for unified quantum-classical accelerated computing

Compiler Platform

- cudaq-quake
- cudaq-opt
- cudaq-translate
- nvq++ driver

CUDA-Q Intermediate Representation (MLIR)

Quake	CC	Func	Math	Arith	LLVM
(Quantum)	(Classical CFG)	(Kernels)	(Standard Math)	(Constants)	(Lowering Target)

Target Low-Level IR - Quantum Intermediate Representation (QIR, Profiles, LLVM IR), QASM ISA

```
__qpu__ double kernel(int N,
           std::vector<double> params) {
   int another = 5;
   return params[0] + params[1];
}
```

```
llvm.func @kernel(%arg0: i32, %arg1: !llvm.struct<(ptr, i64)>) -> f64 {
  %0 = llvm.mlir.constant(5 : i32) : i32
  %1 = llvm.mlir.constant(1 : i32) : i32
  %2 = llvm.alloca %1 x i32 : (i32) -> !llvm.ptr
  llvm.store %arg0, %2 : i32, !llvm.ptr
  %3 = llvm.alloca %1 x i32 : (i32) -> !llvm.ptr
  llvm.store %0, %3 : i32, !llvm.ptr
  %4 = llvm.extractvalue %arg1[0] : !llvm.struct<(ptr, i64)>
  %5 = llvm.load %4 : !llvm.ptr -> f64
  %6 = llvm.extractvalue %arg1[0] : !llvm.struct<(ptr, i64)>
  %7 = llvm.getelementptr inbounds %6[1] : (!llvm.ptr) ->   !llvm.ptr, f64
  %8 = llvm.load %7 : !llvm.ptr -> f64
  %9 = llvm.fadd %5, %8 : f64
  llvm.return %9 : f64
}
```

Fully reuse LLVM dialect to go out to executable code

- MLIR Dialect Reuse
  - Leverage the work from the community
  - Functions, Math and Constants, and the LLVM Dialects
- Optimizations from the community
  - Function inlining, canonicalization, common subexpression elimination
- Lower to the QIR in MLIR before translating MLIR to LLVM IR (also get that for free)

The CUDA-Q Compiler Stack

Platform for unified quantum-classical accelerated computing

Compiler Platform
cudaq-quake
cudaq-opt
cudaq-translate
nvq++ driver
CUDA-Q Intermediate Representation (MLIR)
Quake
(Quantum)
CC
(Classical CFG)
Func
(Kernels)
Math
(Standard Math)
Arith
(Constants)
LLVM
(Lowering Target)
Target Low-Level IR - Quantum Intermediate Representation (QIR, Profiles, LLVM IR), QASM ISA
Quake
Memory Semantic Model
Value Semantic Model
Easier to find optimizations

- Quake (Quantum Kernel Execution) Dialect
  - Model quantum types and operations
- Quake can be in one of 2 forms:
  - Memory semantic model
    - easier to generate, QASM-like
  - Value semantic model
    - easier for optimizations
- Optimizations and Transformations may be better suited for either of these forms
  - Track the use-def chains

Quake

Memory Semantic Model

```
func.func foo(%veq : !quake.veq<2>) {
  // Boilerplate to extract each qubit from the vector
  %c0 = arith.constant 0 : index
  %c1 = arith.constant 1 : index
  %q0 = quake.extract_ref %veq[%c0] :
    (!quake.veq<2>, index) -> !quake.ref
  %q1 = quake.extract_ref %veq[%c1] :
    (!quake.veq<2>, index) -> !quake.ref

  // We apply some operators to those extracted qubits
  // ... bunch of operators using %q0 and %q1 ...
  quake.h %q0 : (!quake.ref) -> ()

  // We decide to measure the vector
  %result = quake.mz %veq : (!quake.veq<2>) -> cc.stdvec<i1>

  // And then apply another Hadamard to %q0
  quake.h %q0 : (!quake.ref) -> ()
  // ...
}
```

Value Semantic Model

Easier to find optimizations

```
func.func @foo() {
  %0 = quake.null_wire
  %1 = quake.null_wire
  %2 = quake.h %0 : (!quake.wire) -> !quake.wire
  %3 = quake.h %2 : (!quake.wire) -> !quake.wire
  %4 = quake.h %3 : (!quake.wire) -> !quake.wire
  %5 = quake.h %4 : (!quake.wire) -> !quake.wire
  %6 = quake.h %5 : (!quake.wire) -> !quake.wire
  %7 = quake.x %6 : (!quake.wire) -> !quake.wire
  %8 = quake.x %7 : (!quake.wire) -> !quake.wire
  %9:2 = quake.x [%8] %1 : (!quake.wire, !quake.wire) ->
      (!quake.wire, !quake.wire)
  %10:2 = quake.x [%9#0] %9#1 : (!quake.wire, !quake.wire) ->
      (!quake.wire, !quake.wire)
  %11:2 = quake.x [%10#0] %10#1 :(!quake.wire, !quake.wire) ->
      (!quake.wire, !quake.wire)
  return
}
```

The CUDA-Q Compiler Stack

Platform for unified quantum-classical accelerated computing

Compiler Platform
cudaq-quake
cudaq-opt
cudaq-translate
nvq++ driver
CUDA-Q Intermediate Representation (MLIR)
Quake
(Quantum)
CC
(Classical CFG)
Func
(Kernels)
Math
(Standard Math)
Arith
(Constants)
LLVM
(Lowering Target)
Target Low-Level IR - Quantum Intermediate Representation (QIR, Profiles, LLVM IR), QASM ISA

- Lower C++ CUDA Quantum Kernels to Quake
- Leverage Clang to build AST, walk the tree and map `__qpu__ FunctionDecls` to MLIR Functions containing Quake and CC operations
- `cudaq-opt` - Transform / Optimize Quake

```
__qpu__ void ghzForLoop(int N) {
  cudaq::qreg q(N);
  h(q[0]);
  for (std::size_t i = 0; i < N - 1; i++)
    x<cudaq::ctrl>(q[i], q[i + 1]);
  mz(q);
}
```

```
cudaq-quake ghz.cpp | cudaq-opt --canonicalize
```

```
func.func ghzForLoop (%arg0: i32) {
 %c0_i64 = arith.constant 0 : i64
 ... (skipped for brevity) ...
 %0 = cc.alloca i32
 cc.store %arg0, %0 : !cc.ptr<i32>
 ... (skipped for brevity) ...
 quake.h %4 : (!quake.ref) -> ()
 %5 = cc.alloca i64
 cc.store %c0_i64, %5 : !cc.ptr<i64>
 cc.loop while {
  ... (skipped for brevity) ...
  cc.condition %11
 } do {
  ... (skipped for brevity) ...
  %11 = quake.extract_ref %3[%10] : (!quake.veq<?>, i64) -> !quake.ref
  quake.x [%8] %11 : (!quake.ref, !quake.ref) -> ()
  cc.continue
 } step {
 ... (skipped for brevity) ...
}
```

The CUDA-Q Compiler Stack

Platform for unified quantum-classical accelerated computing

Compiler Platform
cudaq-quake
cudaq-opt
cudaq-translate
nvq++ driver
CUDA-Q Intermediate Representation (MLIR)
Quake
(Quantum)
CC
(Classical CFG)
Func
(Kernels)
Math
(Standard Math)
Arith
(Constants)
LLVM
(Lowering Target)
Target Low-Level IR - Quantum Intermediate Representation (QIR, Profiles, LLVM IR), QASM ISA

- Translate Quake to external representations
  - **QIR**
  - QIR Profiles
  - OpenQASM 2.0
  - IQM JSON

```
__qpu__ void simple() {
  cudaq::qubit q, r;
  h(r);
  x(q);
}
```

```
cudaq-quake ghz.cpp |
   cudaq-opt --canonicalize |
   cudaq-translate --convert-to=qir
```

```
%Array = type opaque
%Qubit = type opaque

... (skipped for brevity) ...

define void @__nvqpp__mlirgen__function_test._Z4testv() local_unnamed_addr {
  %1 = tail call %Array* @__quantum__rt__qubit_allocate_array(i64 2)
  %2 = tail call i8* @__quantum__rt__array_get_element_ptr_1d(%Array* %1, i64 0)
  %3 = bitcast i8* %2 to %Qubit**
  %4 = load %Qubit*, %Qubit** %3, align 8
  %5 = tail call i8* @__quantum__rt__array_get_element_ptr_1d(%Array* %1, i64 1)
  %6 = bitcast i8* %5 to %Qubit**
  %7 = load %Qubit*, %Qubit** %6, align 8
  tail call void @__quantum__qis__h(%Qubit* %7)
  tail call void @__quantum__qis__x(%Qubit* %4)
  tail call void @__quantum__rt__qubit_release_array(%Array* %1)
  ret void
}
```

The CUDA-Q Compiler Stack

Platform for unified quantum-classical accelerated computing

Compiler Platform

- cudaq-quake
- cudaq-opt
- cudaq-translate
- nvq++ driver

CUDA-Q Intermediate Representation (MLIR)

Quake	CC	Func	Math	Arith	LLVM
(Quantum)	(Classical CFG)	(Kernels)	(Standard Math)	(Constants)	(Lowering Target)

Target Low-Level IR - Quantum Intermediate Representation (QIR, Profiles, LLVM IR), QASM ISA

- Translate Quake to external representations
  - QIR
  - **QIR Profiles**
  - OpenQASM 2.0
  - IQM JSON

```cpp
__qpu__ void simple() {
  cudaq::qubit q, r;
  h(r);
  x(q);
}
```

```
cudaq-quake ghz.cpp |
   cudaq-opt --canonicalize |
   cudaq-translate --convert-to=qir-base
```

```
%Qubit = type opaque

declare void @__quantum__qis__h__body(%Qubit*) local_unnamed_addr
declare void @__quantum__qis__x__body(%Qubit*) local_unnamed_addr
declare void @__quantum__rt__array_end_record_output() local_unnamed_addr
declare void @__quantum__rt__array_start_record_output() local_unnamed_addr

define void @__nvqpp__mlirgen__function_test._Z4testv() local_unnamed_addr #0 {
  tail call void @__quantum__qis__h__body(%Qubit* nonnull inttoptr
                              (i64 1 to %Qubit*))
  tail call void @__quantum__qis__x__body(%Qubit* null)
  tail call void @__quantum__rt__array_start_record_output()
  tail call void @__quantum__rt__array_end_record_output()
  ret void
}

attributes #0 = { "EntryPoint" "requiredQubits"="2" "requiredResults"="0" }
```

The CUDA-Q Compiler Stack

Platform for unified quantum-classical accelerated computing

Compiler Platform
cudaq-quake
cudaq-opt
cudaq-translate
nvq++ driver
CUDA-Q Intermediate Representation (MLIR)
Quake
CC
Func
Math
Arith
LLVM
(Quantum)
(Classical CFG)
(Kernels)
(Standard Math)
(Constants)
(Lowering Target)
Target Low-Level IR - Quantum Intermediate Representation (QIR, Profiles, LLVM IR), QASM ISA

- Translate Quake to external representations
  - QIR
  - QIR Profiles
  - **OpenQASM 2.0**
  - IQM JSON

```
__qpu__ void simple() {
  cudaq::qubit q, r;
  h(r);
  x(q);
}
```

```
cudaq-quake ghz.cpp |
   cudaq-opt --canonicalize |
   cudaq-translate --convert-to=openqasm
```

```
OPENQASM 2.0;

include "qelib1.inc";

qreg var0[1];
qreg var1[1];
h var1[0];
x var0[0];
```

The CUDA-Q Compiler Stack

Platform for unified quantum-classical accelerated computing

Compiler Platform
cudaq-quake
cudaq-opt
cudaq-translate
nvq++ driver

CUDA-Q Intermediate Representation (MLIR)

Quake	CC	Func	Math	Arith	LLVM
(Quantum)	(Classical CFG)	(Kernels)	(Standard Math)	(Constants)	(Lowering Target)

Target Low-Level IR - Quantum Intermediate Representation (QIR, Profiles, LLVM IR), QASM ISA

- nvq++ orchestrates the CUDA Quantum tools
- Replace LLVM kernel function with MLIR one

C++
CPP
nvq++
Clang AST-to-LLVM CodeGen
Transform CUDAQ Kernels via clang::RecursiveASTVisitor subtype
MLIR Gen AST Visitor
MLIR Quake and CC Dialects
Opt Done?
no
yes
QIR (LLVM IR)
object file
Device Code Registration
object file
Link
EXE
Standard LLVM IR for entire Translation Unit
Link-time Override (redirect CUDAQ functions to MLIR functions)
object file

The CUDA-Q Compiler Stack

Platform for unified quantum-classical accelerated computing

Compiler Platform

- cudaq-quake
- cudaq-opt
- cudaq-translate
- nvq++ driver

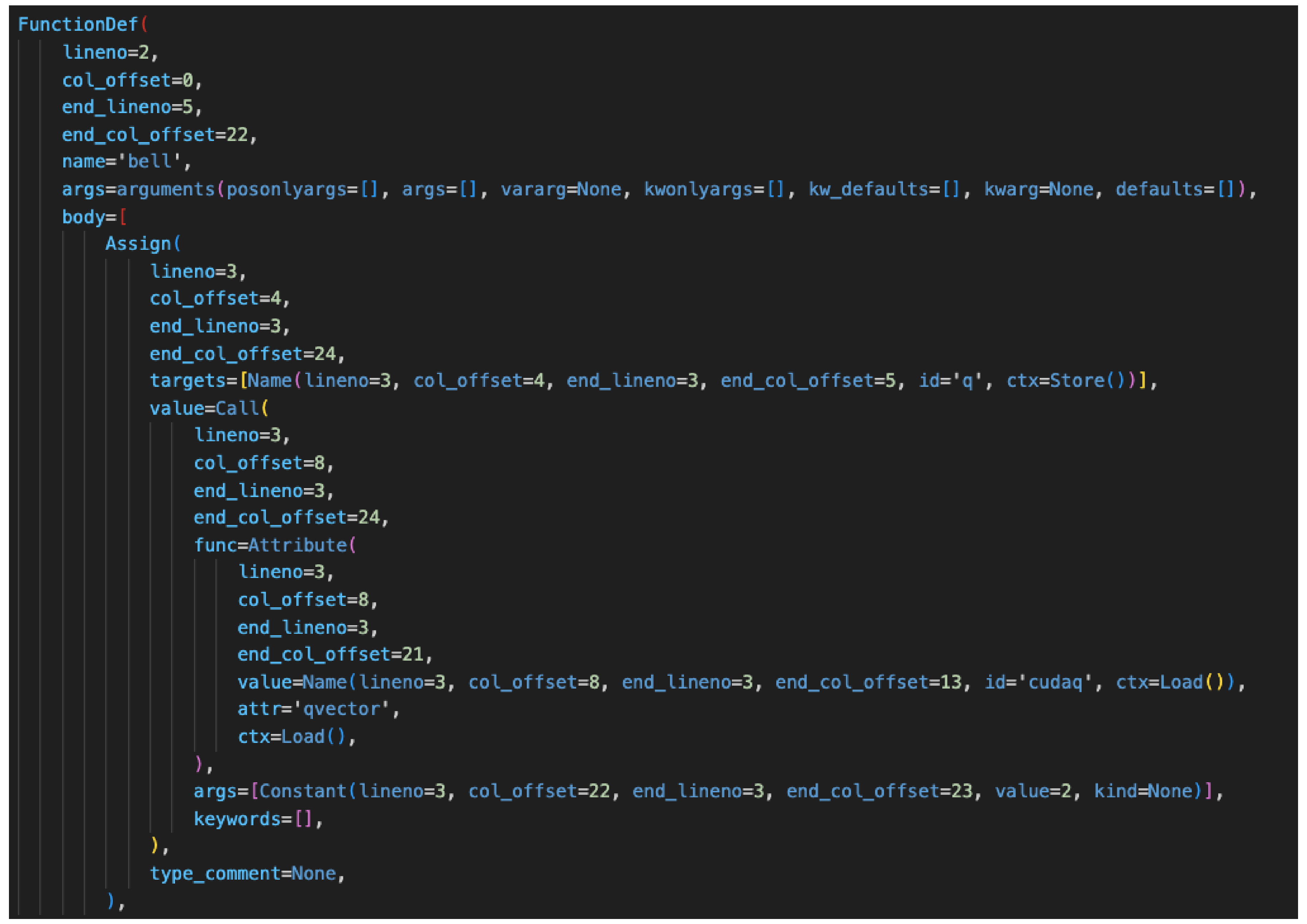
CUDA-Q Intermediate Representation (MLIR)

Quake	CC	Func	Math	Arith	LLVM
(Quantum)	(Classical CFG)	(Kernels)	(Standard Math)	(Constants)	(Lowering Target)

Target Low-Level IR - Quantum Intermediate Representation (QIR, Profiles, LLVM IR), QASM ISA

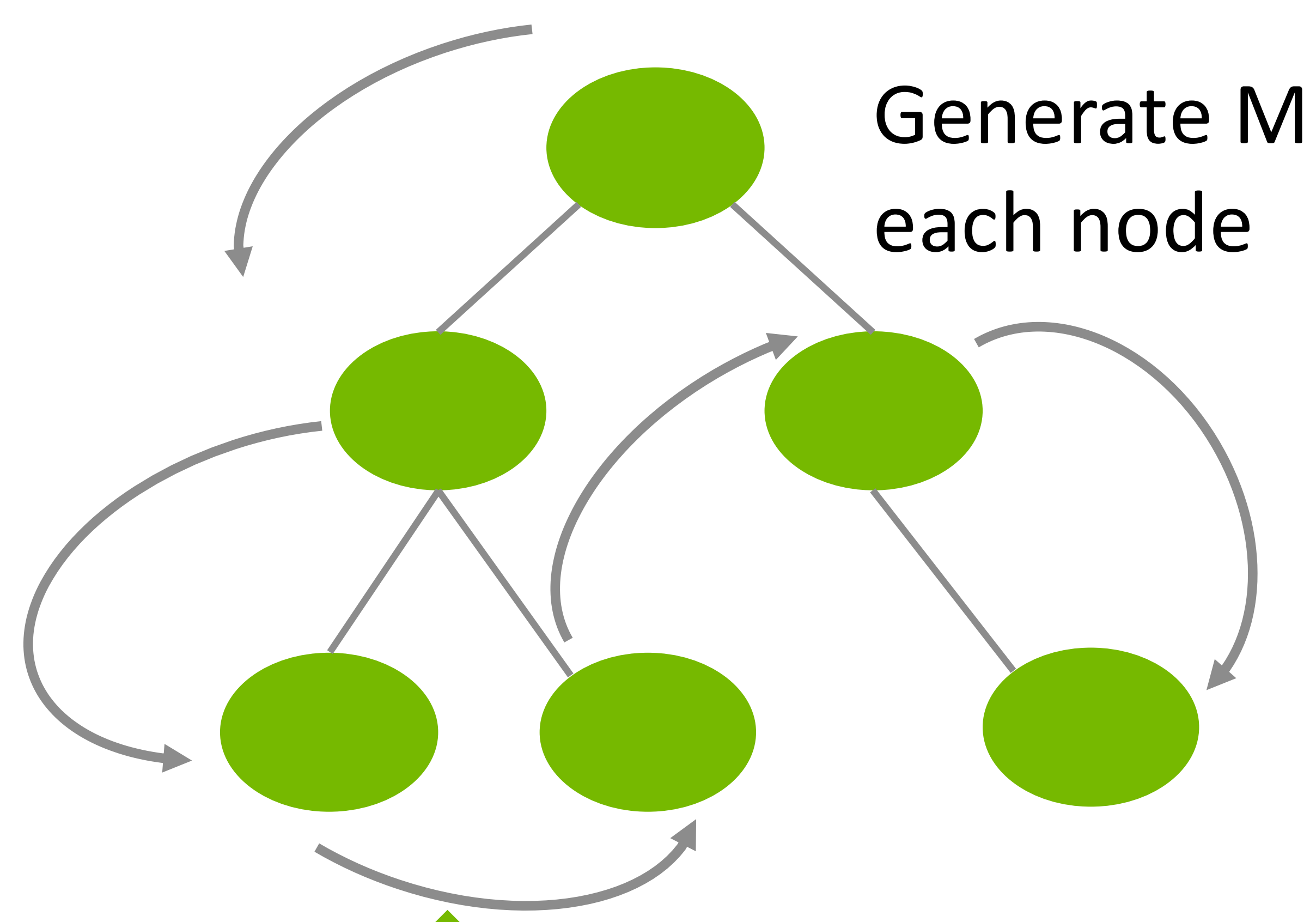
```
@cudaq.kernel
def bell():
   q = cudaq.qvector(2)
   h(q[0])
   x.ctrl(q[0], q[1])
```

Map function to Python AST



```
FunctionDef(
    lineno=2,
    col_offset=0,
    end_lineno=5,
    end_col_offset=22,
    name='bell',
    args=arguments(posonlyargs=[], args=[], vararg=None, kwonlyargs=[], kw_defaults=[], kwarg=None, defaults=[]),
    body=[
        Assign(
            lineno=3,
            col_offset=4,
            end_lineno=3,
            end_col_offset=24,
            targets=[Name(lineno=3, col_offset=4, end_lineno=3, end_col_offset=5, id='q', ctx=Store())],
            value=Call(
                lineno=3,
                col_offset=8,
                end_lineno=3,
                end_col_offset=24,
                func=Attribute(
                    lineno=3,
                    col_offset=8,
                    end_lineno=3,
                    end_col_offset=21,
                    value=Name(lineno=3, col_offset=8, end_lineno=3, end_col_offset=13, id='cudaq', ctx=Load()),
                    attr='qvector',
                    ctx=Load(),
                ),
                args=[Constant(lineno=3, col_offset=22, end_lineno=3, end_col_offset=23, value=2, kind=None)],
                keywords=[],
            ),
            type_comment=None,
        ),
```

AST Visitor

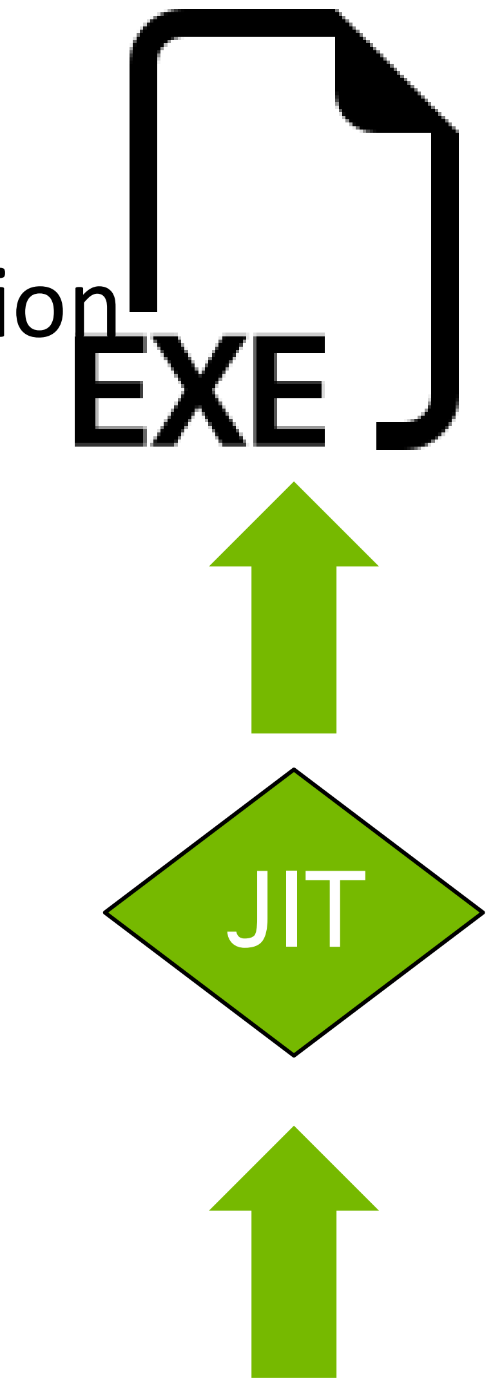


Generate MLIR for each node

Visit / Walk AST nodes

```
func.func @bell() {
  %0 = quake.alloca !quake.veq<2>
  %1 = quake.extract_ref %0[0] : (!quake.veq<2>) -> !quake.ref
  quake.h %1 : (!quake.ref) -> ()
  %2 = quake.extract_ref %0[1] : (!quake.veq<2>) -> !quake.ref
  quake.x [%1] %2 : (!quake.ref, !quake.ref) -> ()
  return
}
```

JIT compile, extract function pointer



CUDA-Q in Action

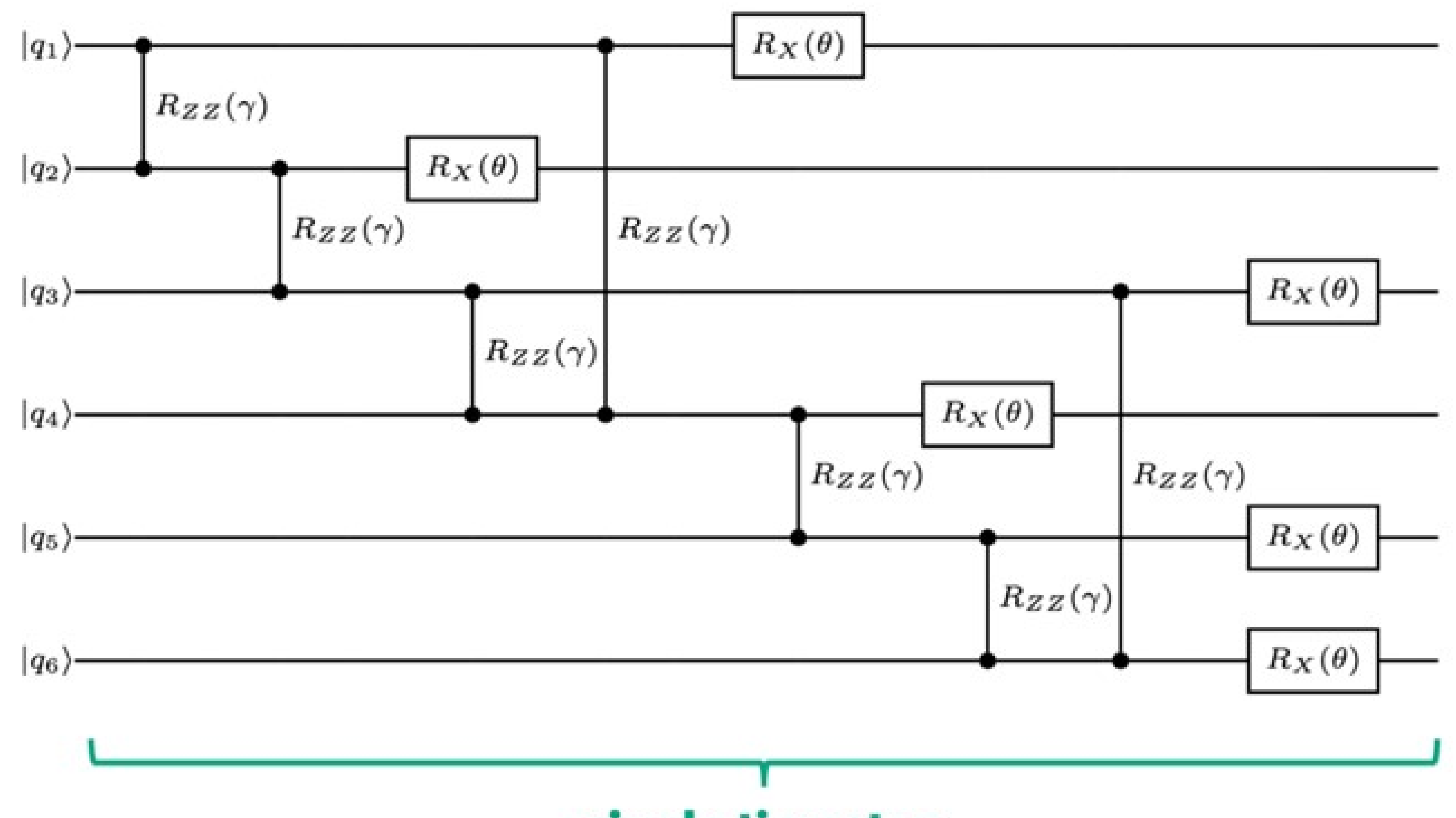
- Dramatic performance improvements
- Hybrid QPU-GPU applications

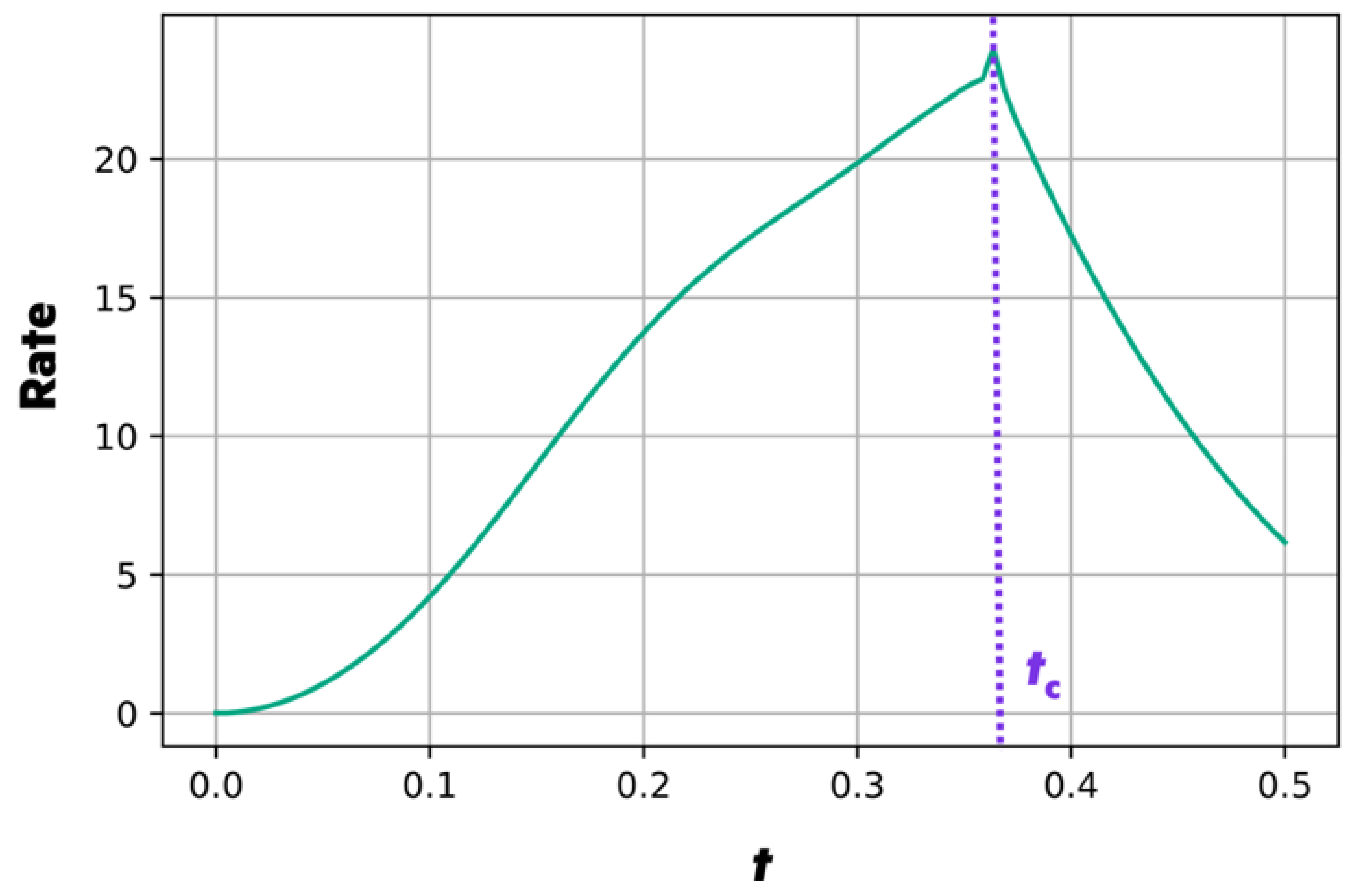
CUDA-Q in Action

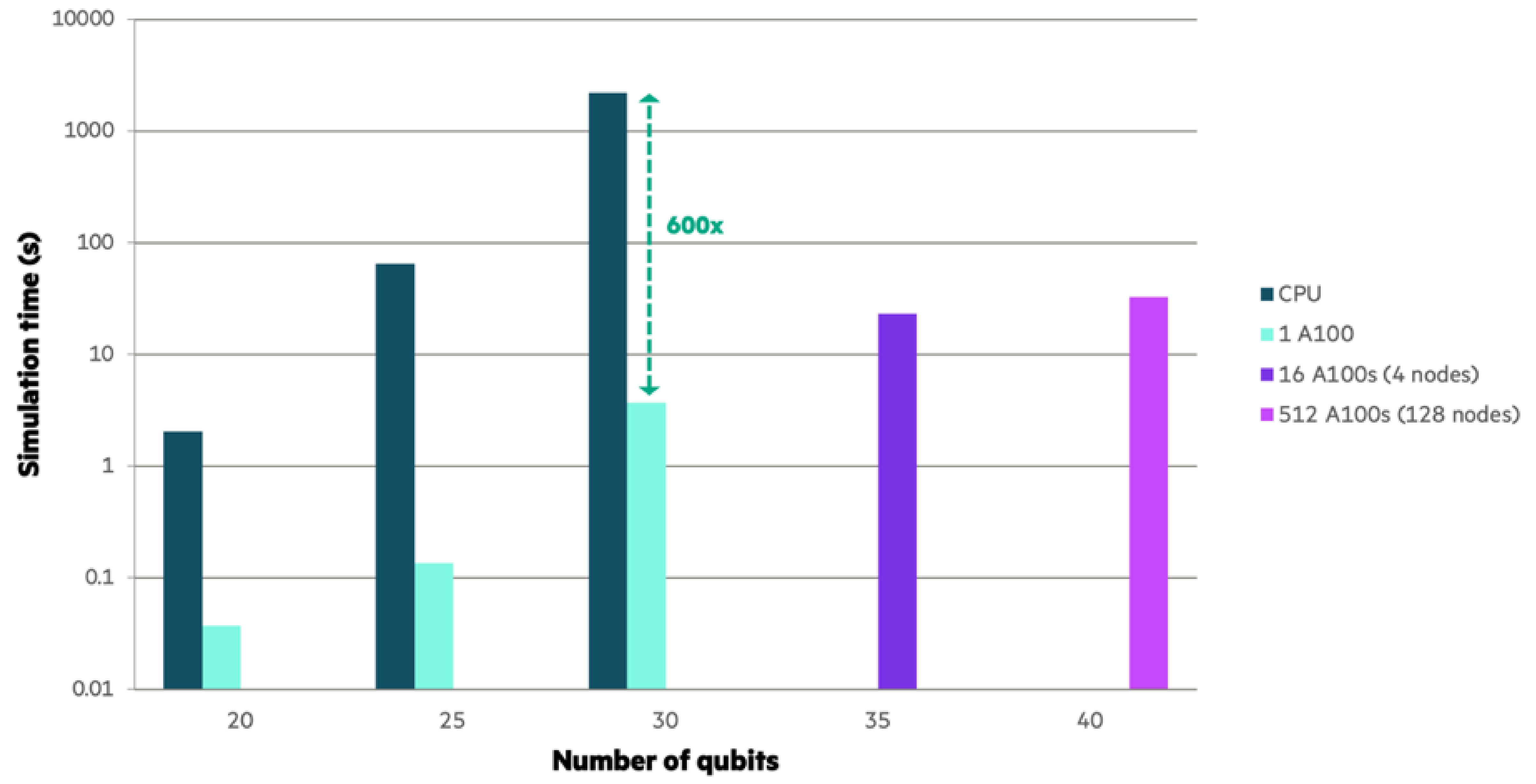
Speed-ups for time-evolution of the transverse field Ising model (TFIM)

- Collaboration with Hewlett Packard Labs
- Study dynamical quantum phase transitions
  - Requires computation of overlap of initial state with time evolved state
- Leverage NVIDIA multi-node, multi-GPU simulation backend.
  - Distributed state-vector simulator
- 600x performance increase over multi-threaded CPU approaches

```
@cudaq.kernel()
def tfimEvolve(timeStep: float, params: list[float]):
    qubits = cudaq.qvector(40)
    ... Circuit, use input params ...

for time in range(finalTime):
    state = cudaq.get_state(tfimEvolve, time, params)
    overlaps.append(state.overlap(initialState))
```



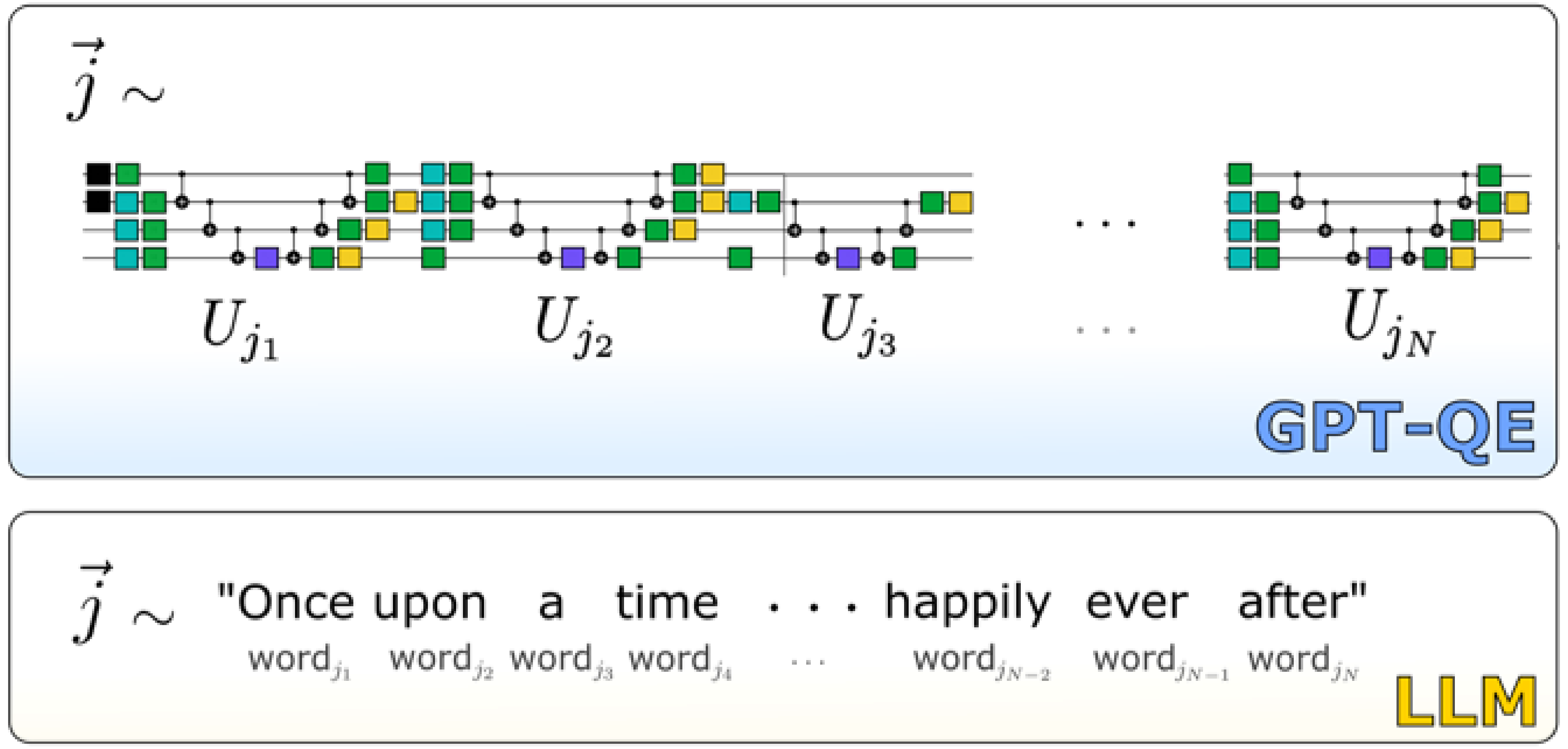


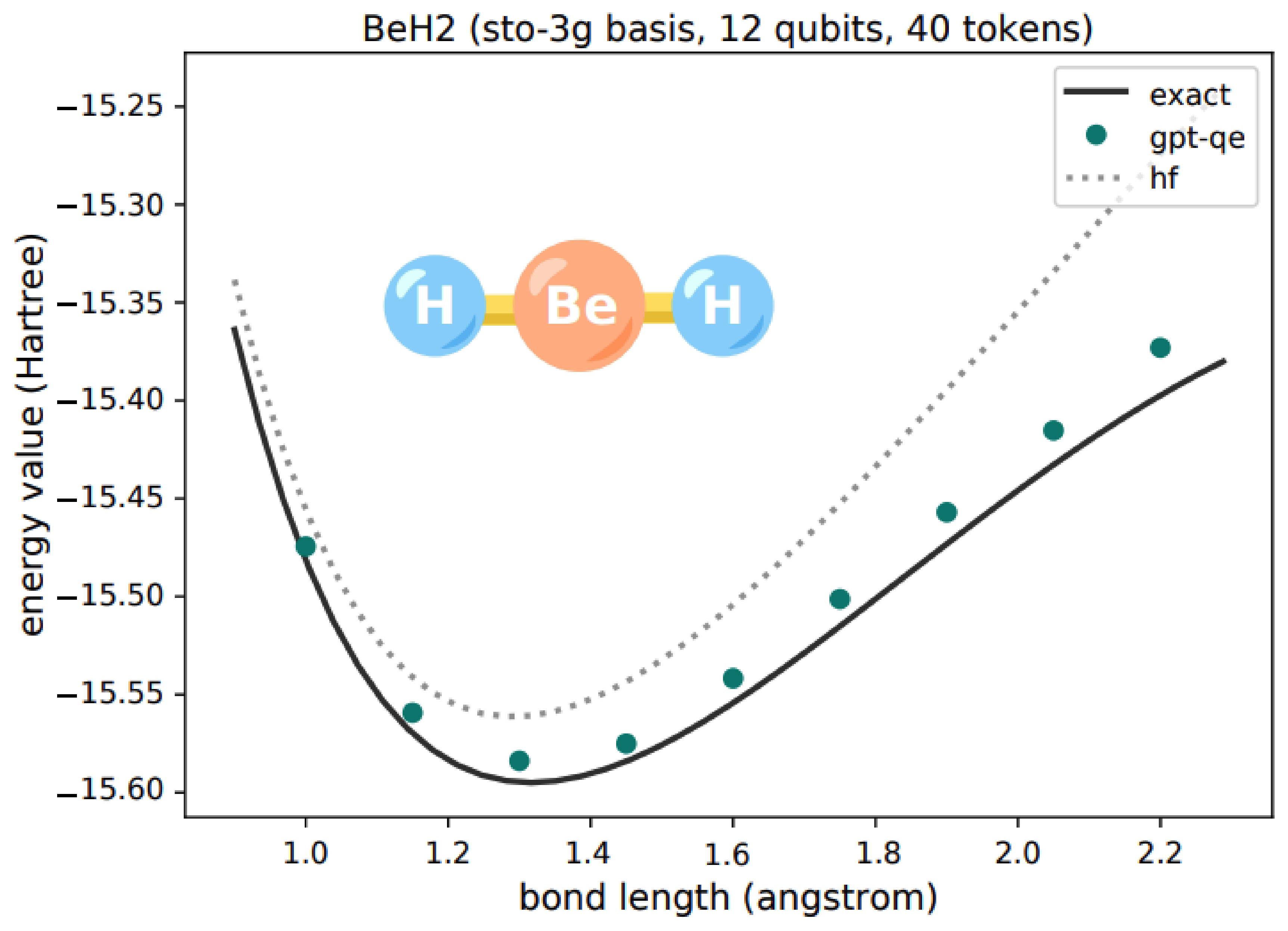


CUDA-Q in Action

GPT-QE - University of Toronto and St. Jude Children's Research Hospital with CUDA-Q

- Developed a novel Generative Pre-Trained Transformer-based (GPT) method for computing the ground-state energy of molecules of interest
- The first demonstration of a GPT-generated quantum circuit in the literature
- A powerful example of leveraging AI to accelerate quantum computing
- Executed using CUDA-Q on A100 GPUs on Perlmutter
- Opens the door to a wide variety of novel Generative Quantum Algorithms (GQAs) for drug discovery, materials science, and environmental challenges


$\vec{j} \sim$
U_{j_1}
U_{j_2}
U_{j_3}
U_{j_N}
GPT-QE
$\vec{j} \sim$ "Once upon a time . . . happily ever after"
word_{j_1} word_{j_2} word_{j_3} word_{j_4} ... $\text{word}_{j_{N-2}}$ $\text{word}_{j_{N-1}}$ word_{j_N}
LLM

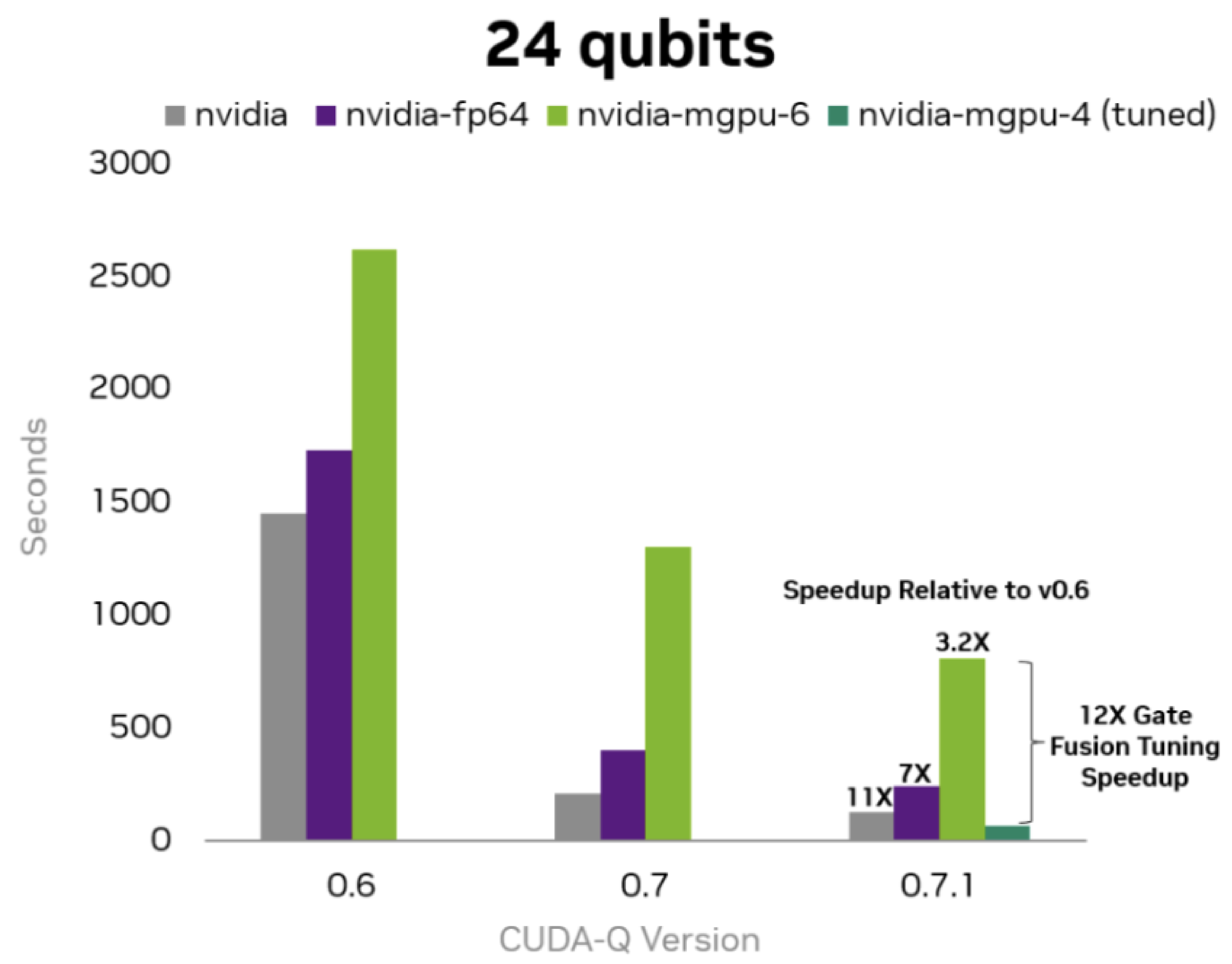


https://arxiv.org/pdf/2401.09253.pdf

New Performance Updates

CUDA-Q performance updates

Improved Gate Fusion



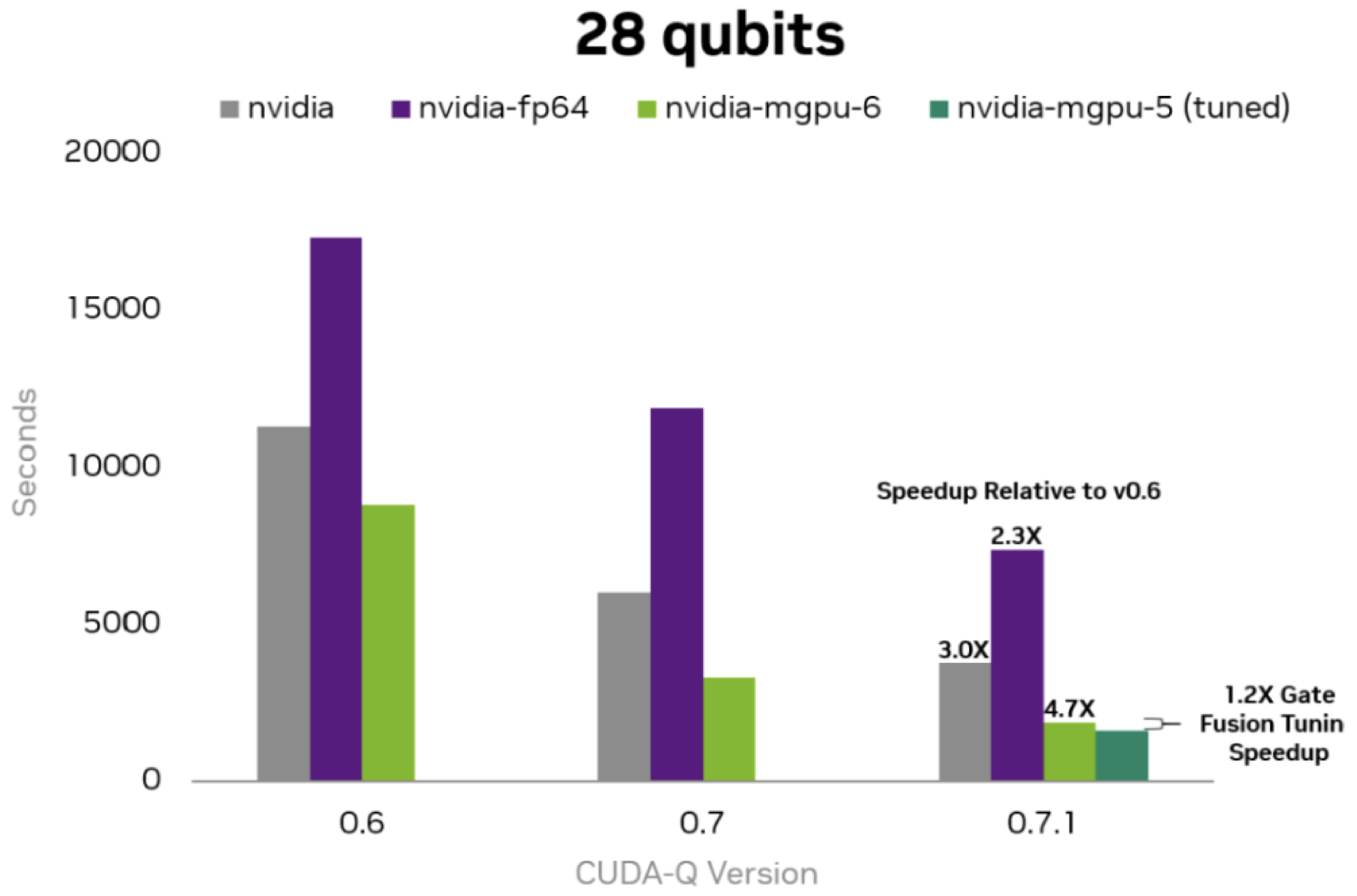


Figure 1. Execution times for 10 `observe` calls in 24 and 28 qubit UCCSD-VQE experiments

https://developer.nvidia.com/blog/new-nvidia-cuda-q-features-boost-quantum-application-performance/

CUDA-Q performance updates

Improved JIT compilation and runtime

v0.6	Baseline
v0.7	JIT path improved from *$O(N^2)$* to *$O(N)$* scaling (Much shorter blue)
v0.7	Improved hashing for JIT Checks (Virtually eliminate gray)
v0.7	Improved logging (Slightly shorter green)
v0.7.1	Optimized gate fusion level/Hamiltonian batching (Much shorter green)

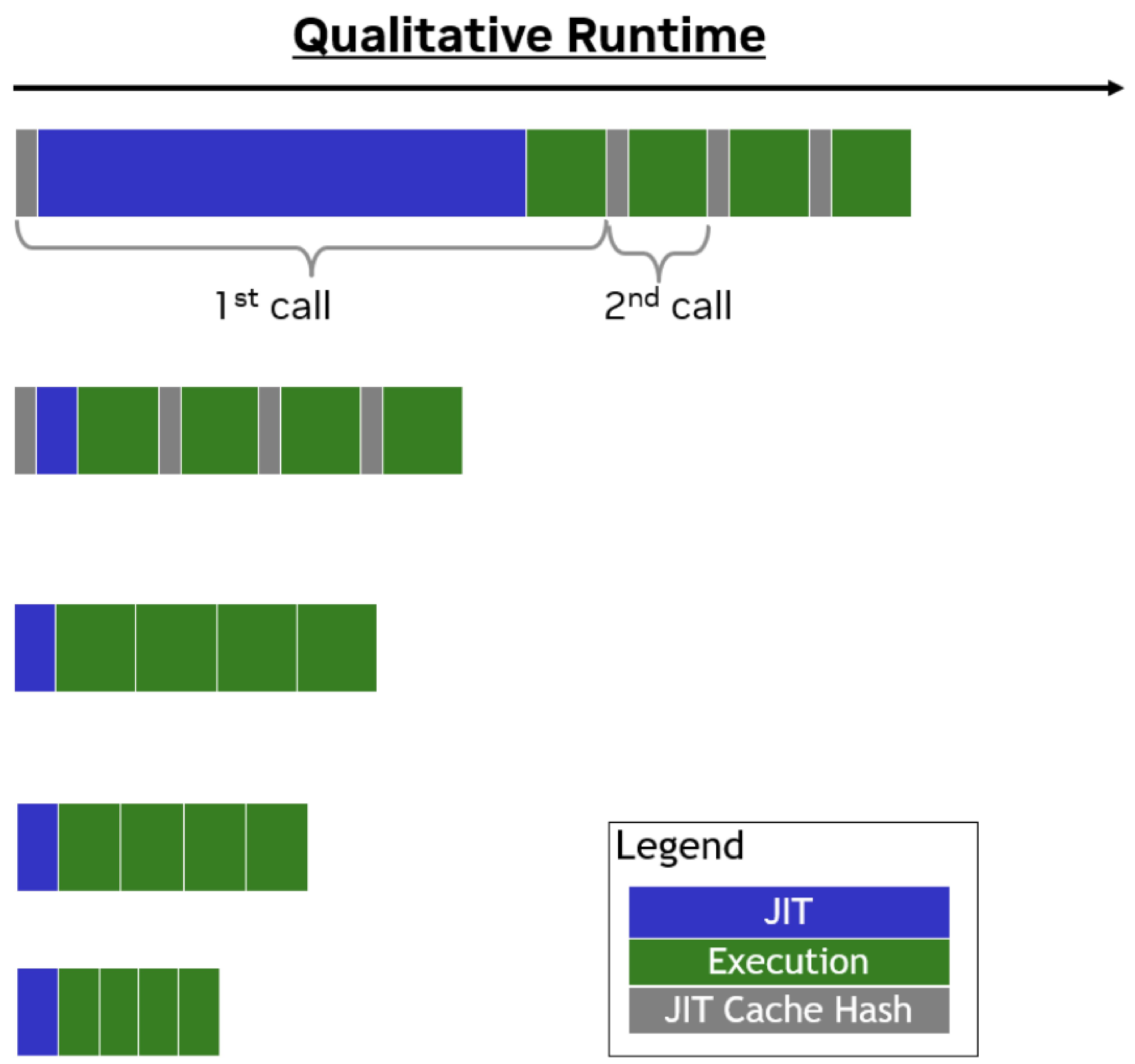


Figure 2. Representation of the changes included in CUDA-Q v0.7 and v0.7.1 and the runtime improvements to four `observe` calls

https://developer.nvidia.com/blog/new-nvidia-cuda-q-features-boost-quantum-application-performance/

CUDA-Q: State Handling Simulator

Connecting to a simulator

- Simulator backend-specific state data representation (`cudaq::SimulationState`)

  Abstract away state data and resources (e.g., simple arrays in GPU or host memory, distributed memory blobs, tensors' data along with the tensor network description, etc.)

`cudaq::SimulationState`

- `getAmplitude`
- `overlap`

  Facilitate simulator-specific data accessor and overlap calculation, e.g., direct GPU memory access, multi-node multi-GPU data access (via MPI), tensor-based amplitude/overlap contraction, etc.

- `toHost`
- `createFromData`

  Data import and export.
  - Flat state vector export if required.
  - Import from a flat state vector in host memory.

- `destroyState`

  Release managed resources, e.g., GPU memory

- `cudaq::state` = user-facing class with simplified life-cycle management (RAII)

CUDA-Q: State Handling in Python

Python syntax and API

- Syntax change connects qubits and state vectors explicitly
  - state expressed as various aggregates of data
    - vector of float, numpy.float32, numpy.float64
    - vector of complex, numpy.complex64, numpy.complex128
    - numpy array with float or complex dtypes
    - cudaq::state (next)
- Design: kernels are converted to MLIR and JIT-ed
  - log(len(state)) qubits are created and initialized with the state
  - Code works for any simulation precision
  - Initializer data type might not match the simulation precision
    - Automatically convert to the correct precision
    - Issue a performance warning if copy of a vector is created
- User code may be portable, simulator precision-agnostic
  - cudaq::complex, cudaq::real types
  - cudaq::amplitudes helper method that creates an array

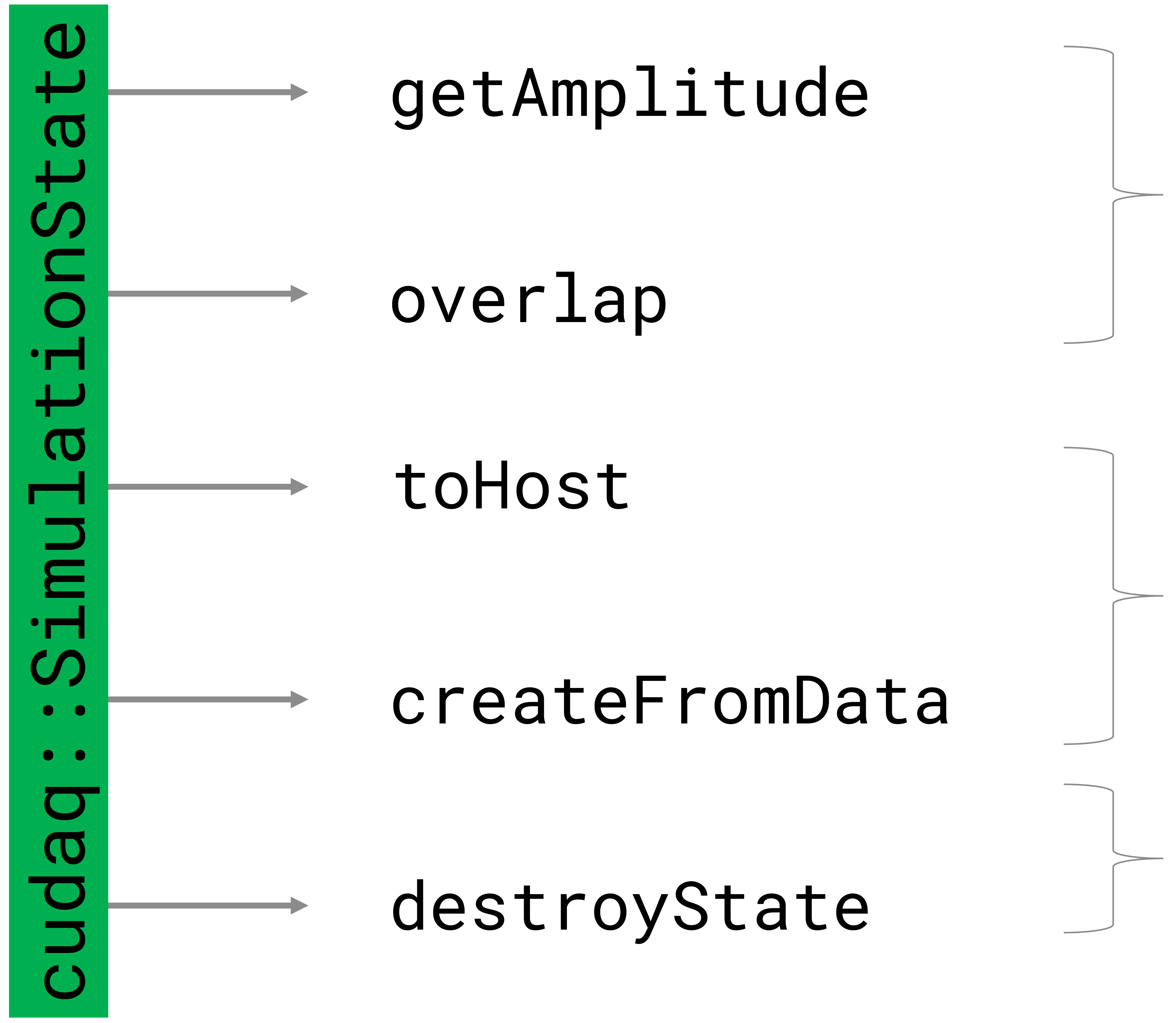
```python
# Passing complex vectors as params
c = [.70710678 + 0j, 0., 0., 0.70710678]
@cudaq.kernel
def kernel(vec: list[complex]):
    q = cudaq.qvector(vec)
counts = cudaq.sample(kernel, c)

# Capturing complex vectors
c = [.70710678 + 0j, 0., 0., 0.70710678]
@cudaq.kernel
def kernel():
    q = cudaq.qvector(c)
counts = cudaq.sample(kernel)

# Using precision-agnostic API
c = np.array([.70710678 + 0j, 0., 0., 0.70710678], dtype=cudaq.complex())
@cudaq.kernel
def kernel(vec: list[complex]):
    q = cudaq.qvector(vec)
counts = cudaq.sample(kernel, c)

c = cudaq.amplitudes([.70710678, 0., 0., 0.70710678])
@cudaq.kernel
def kernel(vec: list[complex]):
    q = cudaq.qvector(vec)
counts = cudaq.sample(kernel, c)
```

CUDA-Q: State Handling Simulator

Connecting to a simulator

- Simulator backend-specific state data representation (`cudaq::SimulationState`)

Abstract away state data and resources (e.g., simple arrays in GPU or host memory, distributed memory blobs, tensors' data along with the tensor network description, etc.)



Facilitate simulator-specific data accessor and overlap calculation, e.g., direct GPU memory access, multi-node multi-GPU data access (via MPI), tensor-based amplitude/overlap contraction, etc.

Data import and export.
- Flat state vector export if required.
- Import from a flat state vector in host memory.

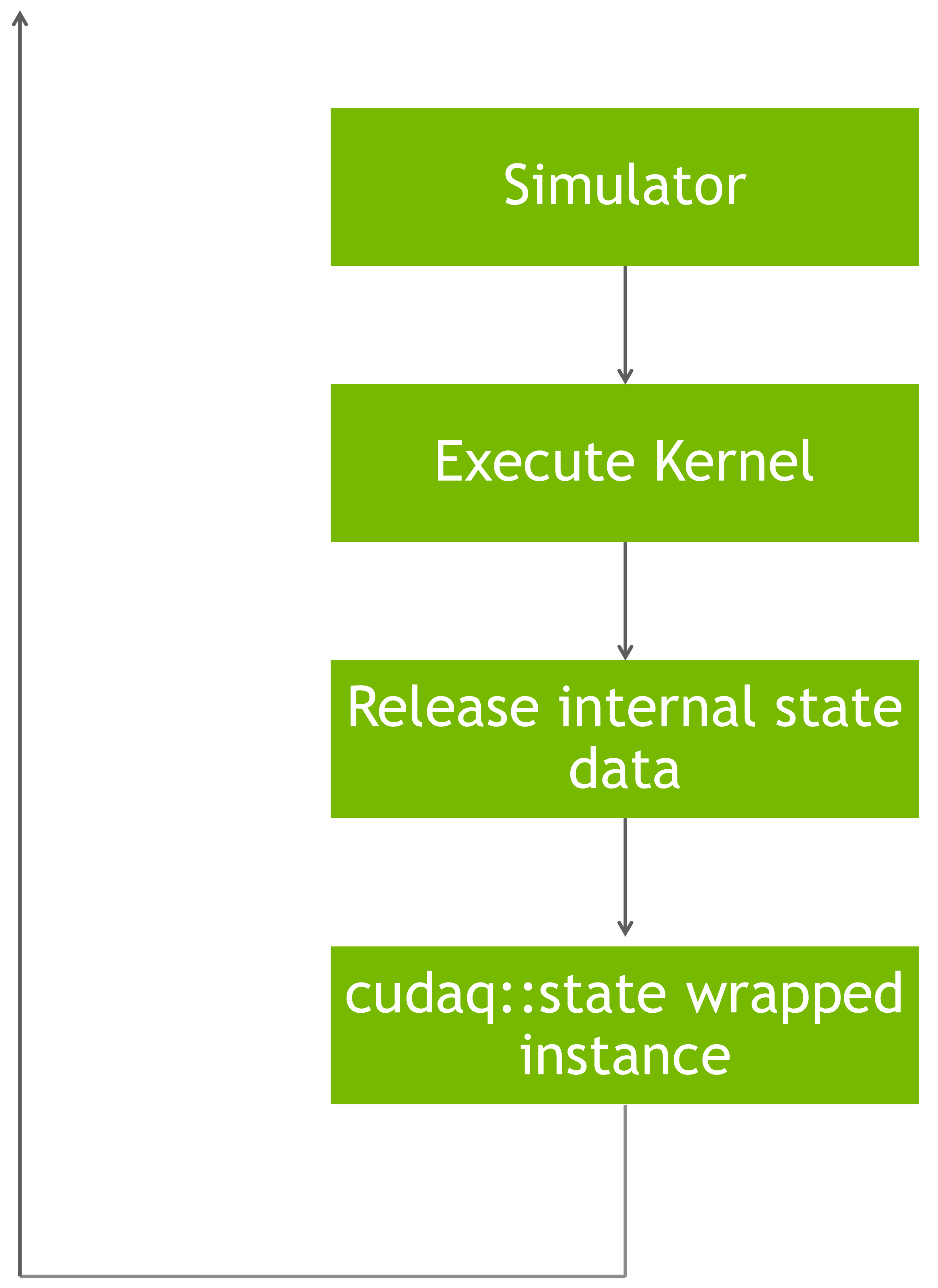
Release managed resources, e.g., GPU memory

- `cudaq::state` = user-facing class with simplified life-cycle management (RAII)

CUDA-Q: State Handling Simulator

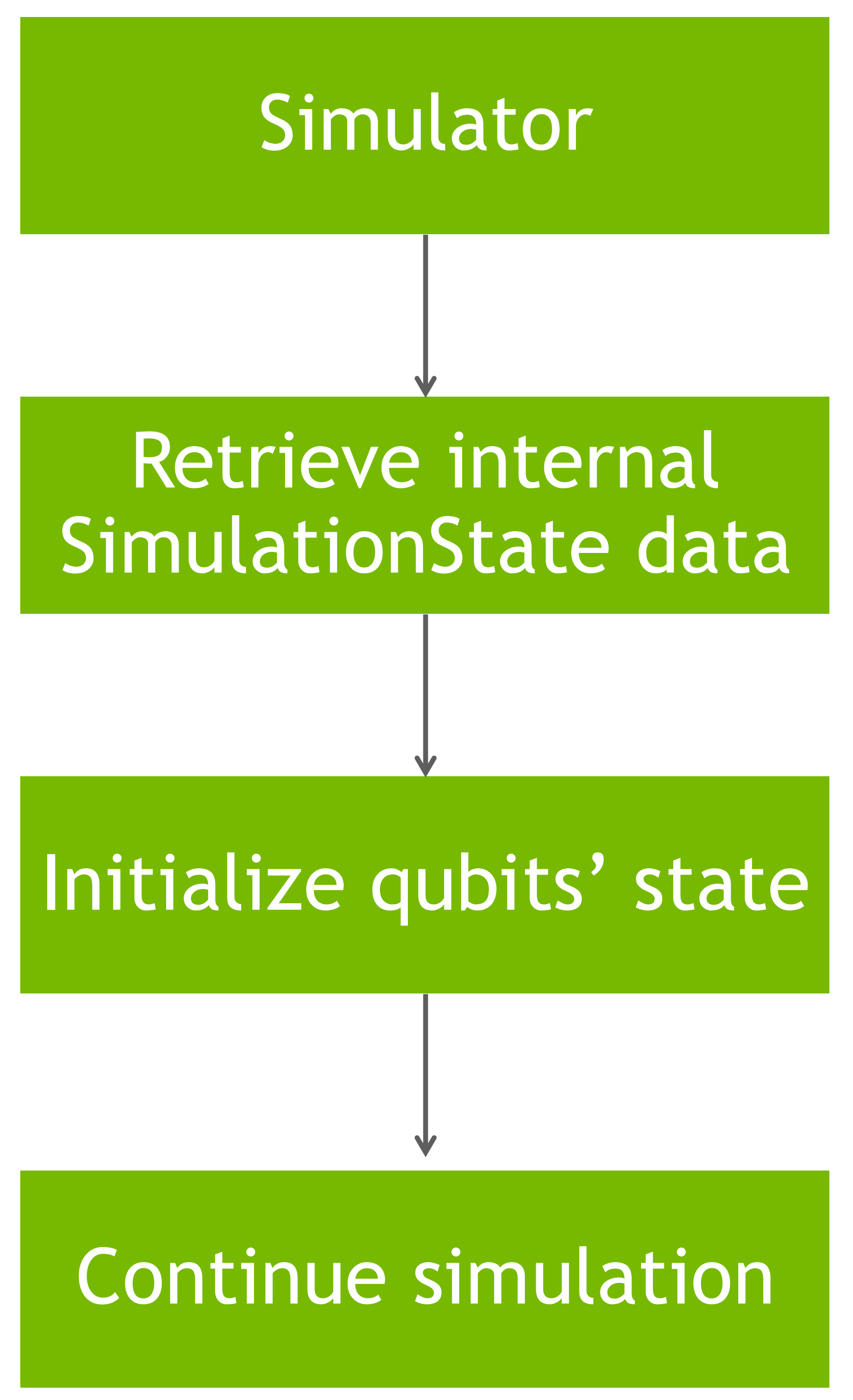
Connecting to a simulator

`cudaq::get_state(Kernel, Args…)`



State Retrieval

`qvector(cudaq::state)`



Native state data to the backend, already in GPU memory, or properly MPI-distributed; tensors in GPU memory, etc.

FAST!

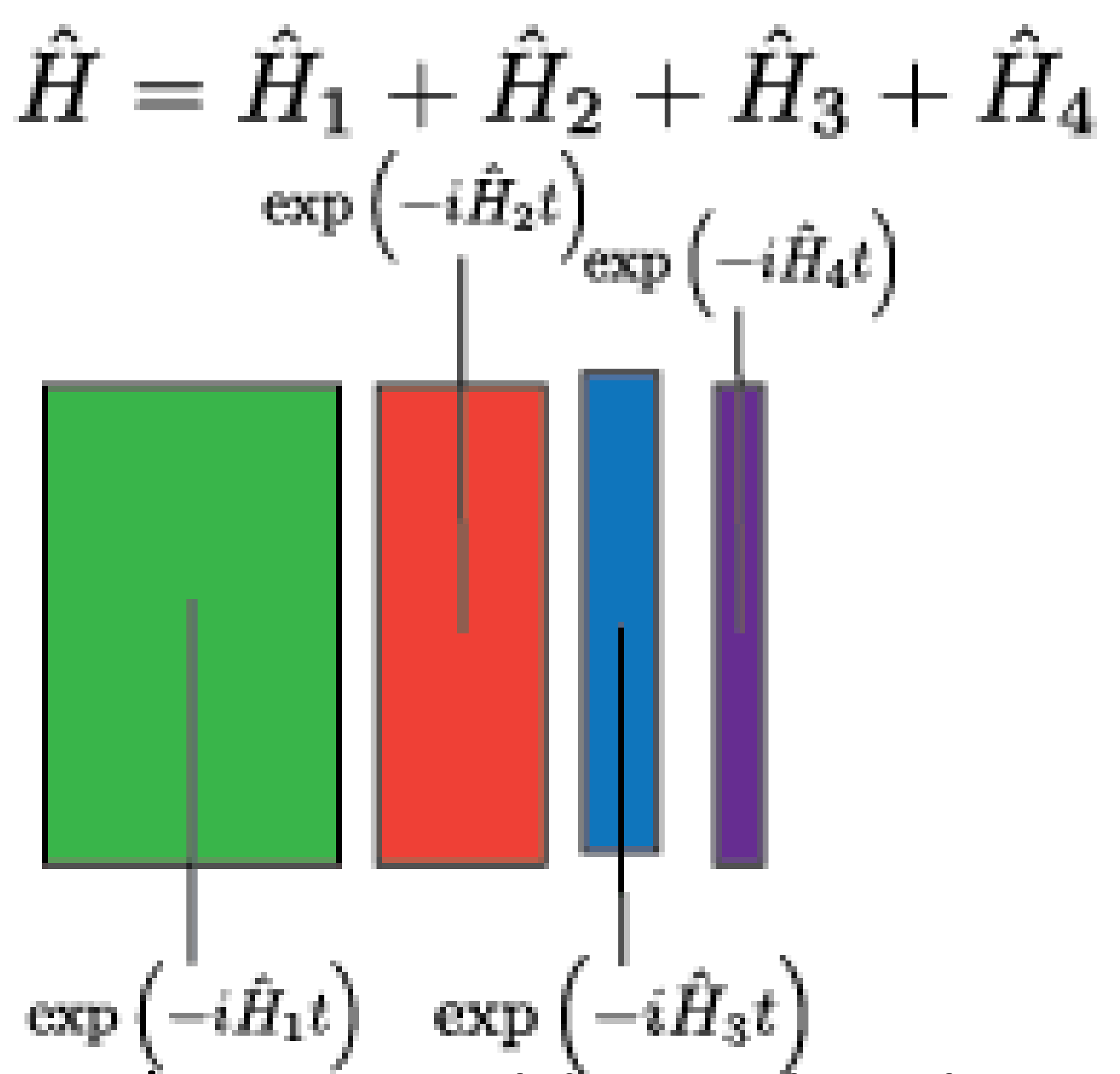
State Initialization

CUDA-Q: State Handling Example

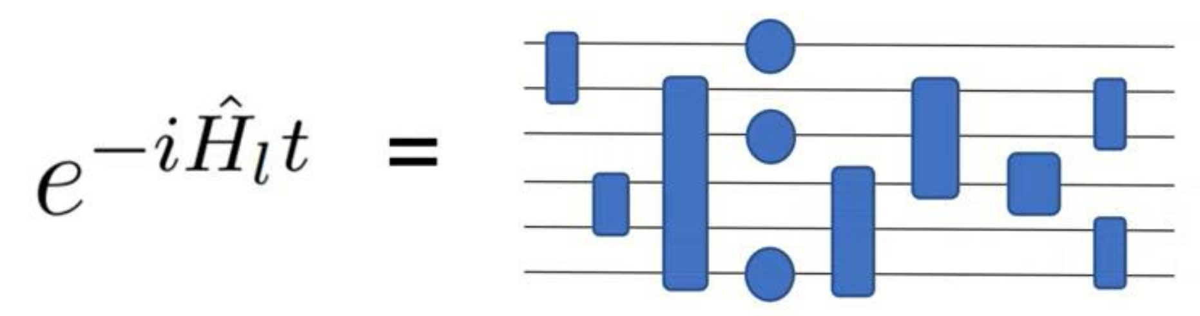
Use Cases

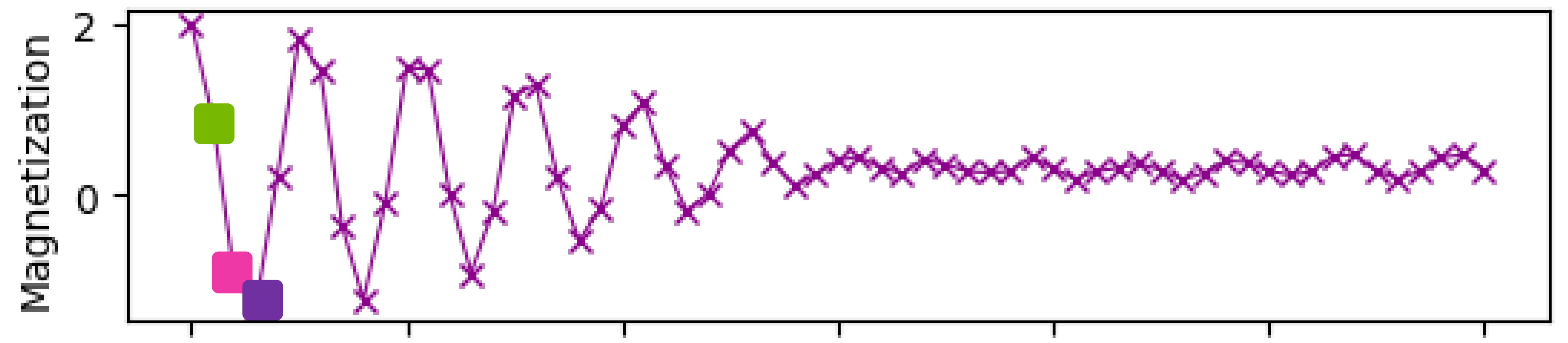
Efficient state vector handling (retrieval and initialization) in simulation is an important requirement for various applications/algorithms

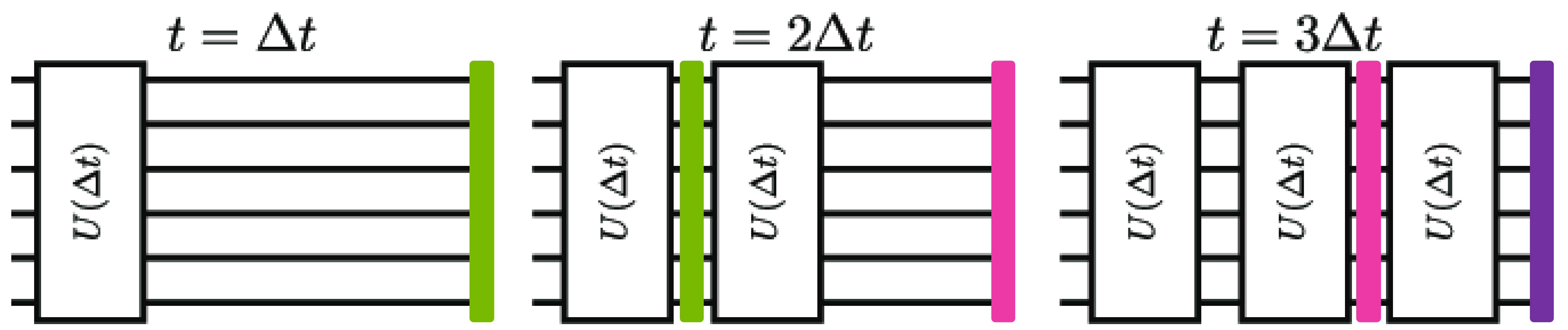
- Quantum state overlap ($\langle \Psi_1 | \Psi_2 \rangle$) calculation (difficult to perform on quantum hardware)
- Dynamical quantum simulation



$\hat{H} = \hat{H}_1 + \hat{H}_2 + \hat{H}_3 + \hat{H}_4$
$\exp\left(-i\hat{H}_2 t\right)$
$\exp\left(-i\hat{H}_4 t\right)$
$\exp\left(-i\hat{H}_1 t\right)$
$\exp\left(-i\hat{H}_3 t\right)$



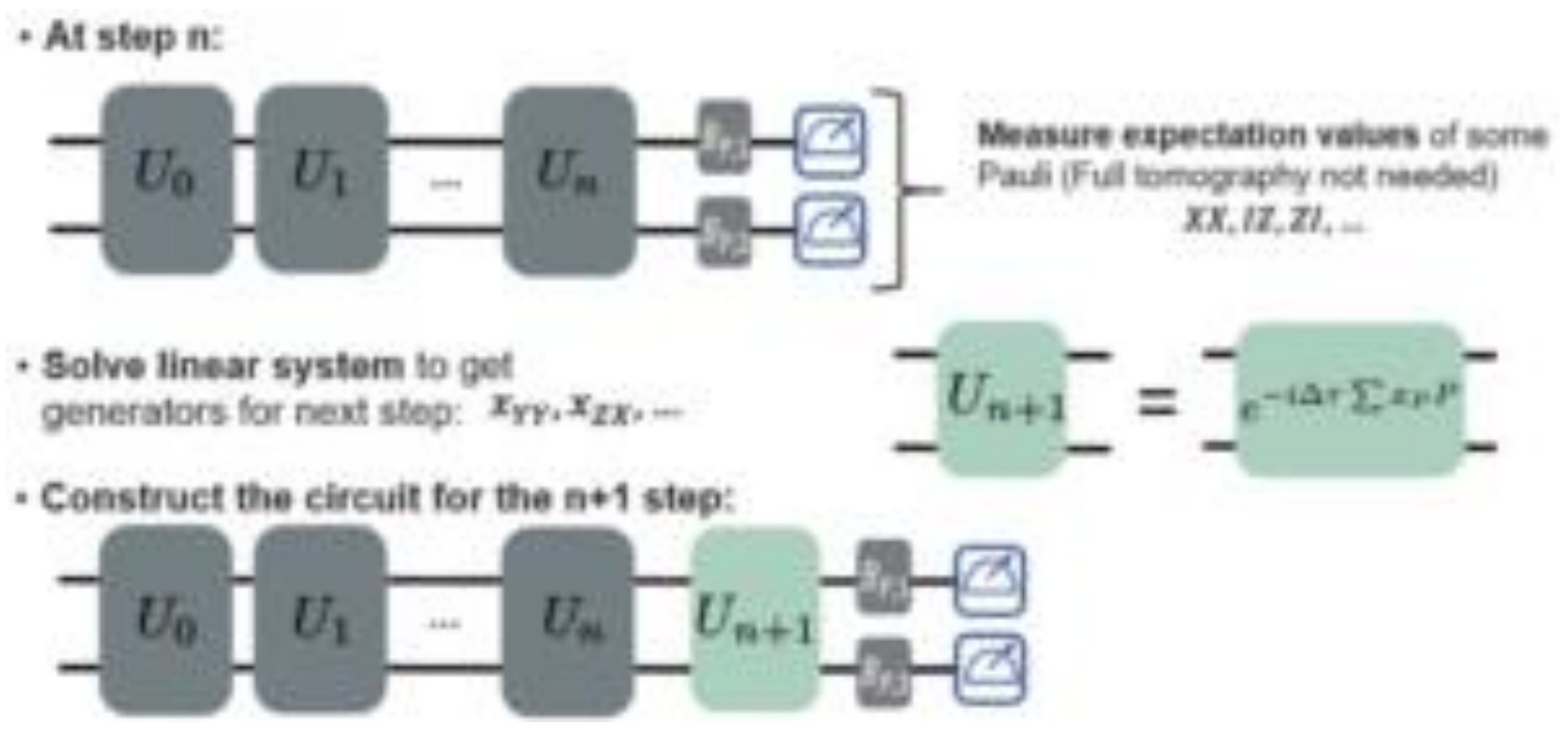




e.g., determining phase transition point via state vector inspection or expectation calculation at each time step.

- Quantum Imaginary Time Evolution (QITE)

(repeating loop of applying a unitary, state tomography, solving linear equations to determine the next unitary block to apply)



• At step n:
U_0
U_1
U_n
Measure expectation values of some Pauli (Full tomography not needed)
$XX, IZ, ZI, \dots$
• Solve linear system to get generators for next step: $x_{YY}, x_{ZX}, \dots$
U_{n+1} = $e^{-i\Delta\tau \sum x_P P}$
• Construct the circuit for the n+1 step:
U_{n+1}

CUDA-Q: State Handling Example

Demo: Trotter Dynamical Simulation

CUDA-Q v0.7

```
struct trotter {
  // Note: this runs Trotter simulation from the initial state
  // (|00..00>) up to a certain time point (specified by num_step).
  void operator()(int num_spins,
                  std::function<cudaq::spin_op(double)> td_ham,
                  double dt, int num_step) __qpu__ {
    cudaq::qvector q(num_spins);
    // Alternating up/down spins
    for (int qId = 0; qId < num_spins; qId += 2)
      x(q[qId]);
    for (int j = 0; j < num_step; ++j) {
      auto ham = td_ham(j * dt);
      ham.for_each_term([&](cudaq::spin_op &term) {
        const auto coeff = term.get_coefficient();
        const auto pauliWord = term.to_string(false);
        const double theta = coeff.real() * dt;
        cudaq::exp_pauli(dt, q, pauliWord.c_str());
      });
    }
  }
};
```

× Qubit vector/register must be allocated in an uninitialized state.

× Starting the simulation from time 0 to time t at every step/data point.

CUDA-Q v0.8

```
struct trotter {
  // Note: This performs a single-step Trotter on top of an
  // initial state, e.g., result state of the previous Trotter
  // step.
  void operator()(cudaq::state initial_state,
                  cudaq::spin_op ham, double dt) __qpu__ {
    cudaq::qvector q(initial_state);
    ham.for_each_term([&](cudaq::spin_op &term) {
      const auto coeff = term.get_coefficient();
      const auto pauliWord = term.to_string(false);
      const double theta = coeff.real() * dt;
      cudaq::exp_pauli(dt, q, pauliWord.c_str());
    });
  }
};
```

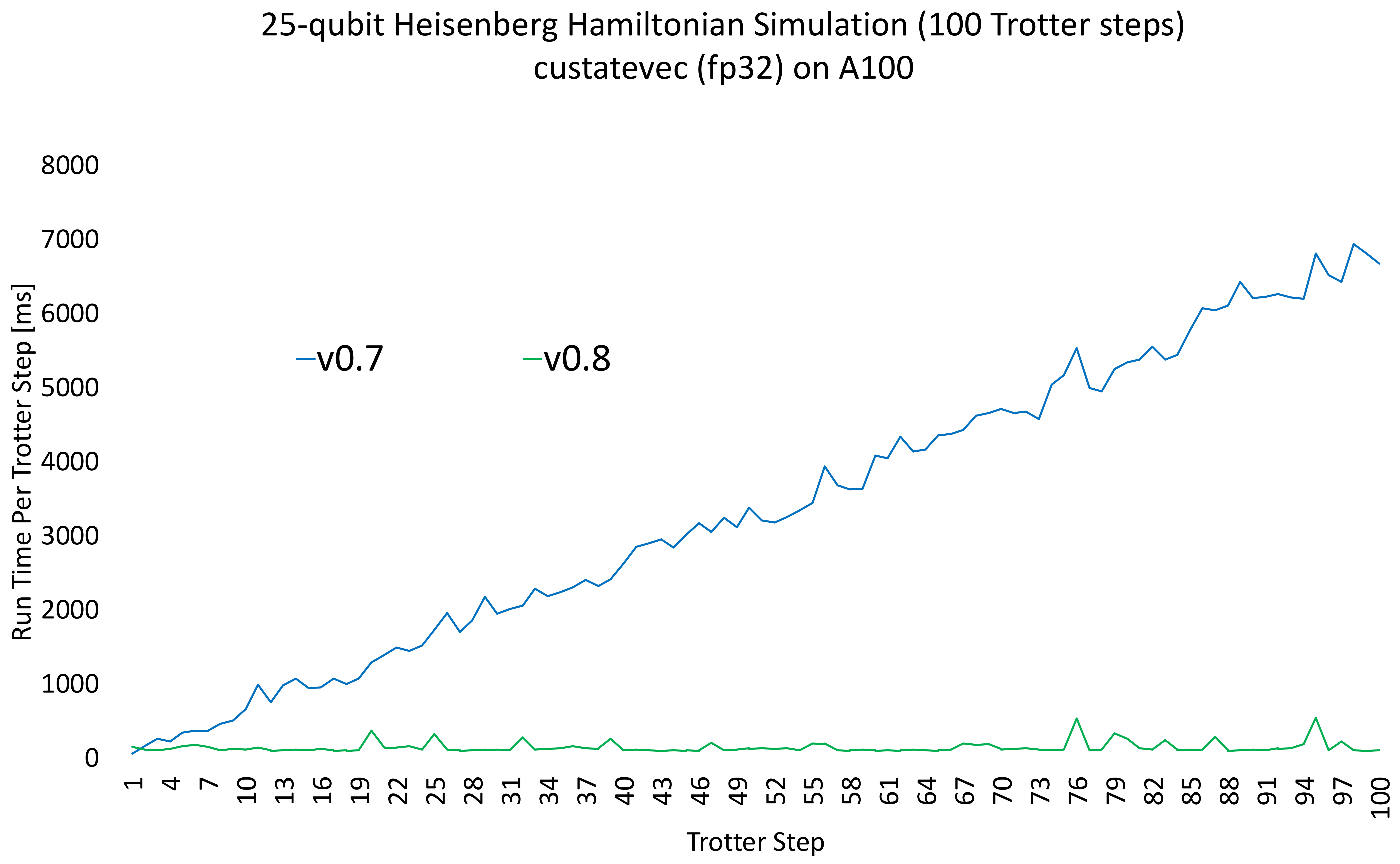
✓ Qubit vector/register can be initialized in a specific state.

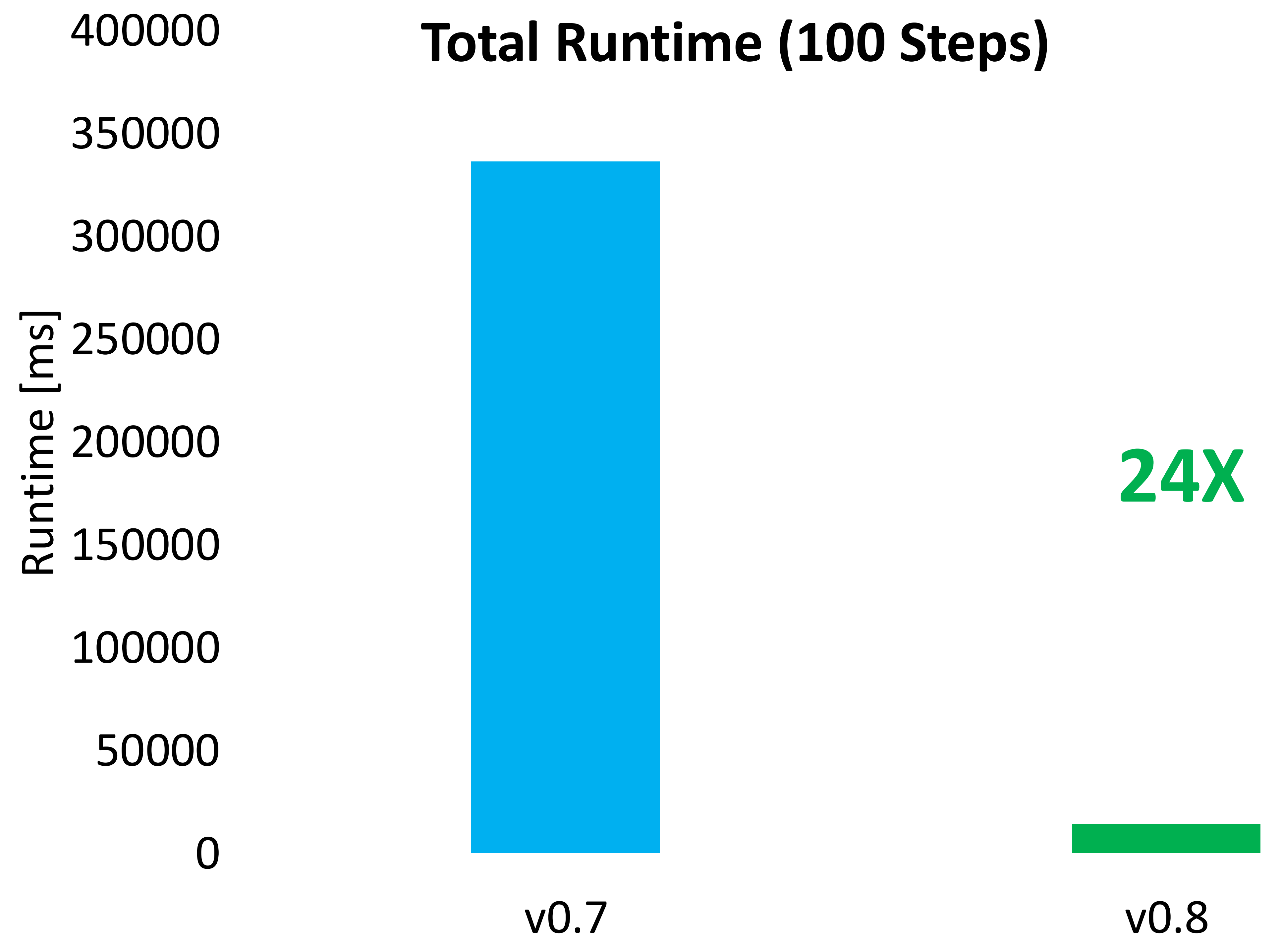
✓ State data can be fed from a prior Trotter step to the next one.

✓ State data is ‘native’ to the current simulator backend, e.g., GPU memory, tensor network form, etc.

CUDA-Q: State Handling Example

Demo: Trotter Dynamical Simulation





Host vs. Device Memory State

CUDA-Q: State Handling Example

Host vs. GPU State

CUDA-Q v0.8

Passing the internal 'state'

```
struct trotter {
  // Note: This performs a single-step Trotter on top of an
  // initial state, e.g., result state of the previous Trotter
  // step.
  void operator()(cudaq::state initial_state,
                  cudaq::spin_op ham, double dt) __qpu__ {
    cudaq::qvector q(initial_state);
    ham.for_each_term([&](cudaq::spin_op &term) {
      const auto coeff = term.get_coefficient();
      const auto pauliWord = term.to_string(false);
      const double theta = coeff.real() * dt;
      cudaq::exp_pauli(dt, q, pauliWord.c_str());
    });
  }
};
```

CUDA-Q v0.8

Passing the host memory vector

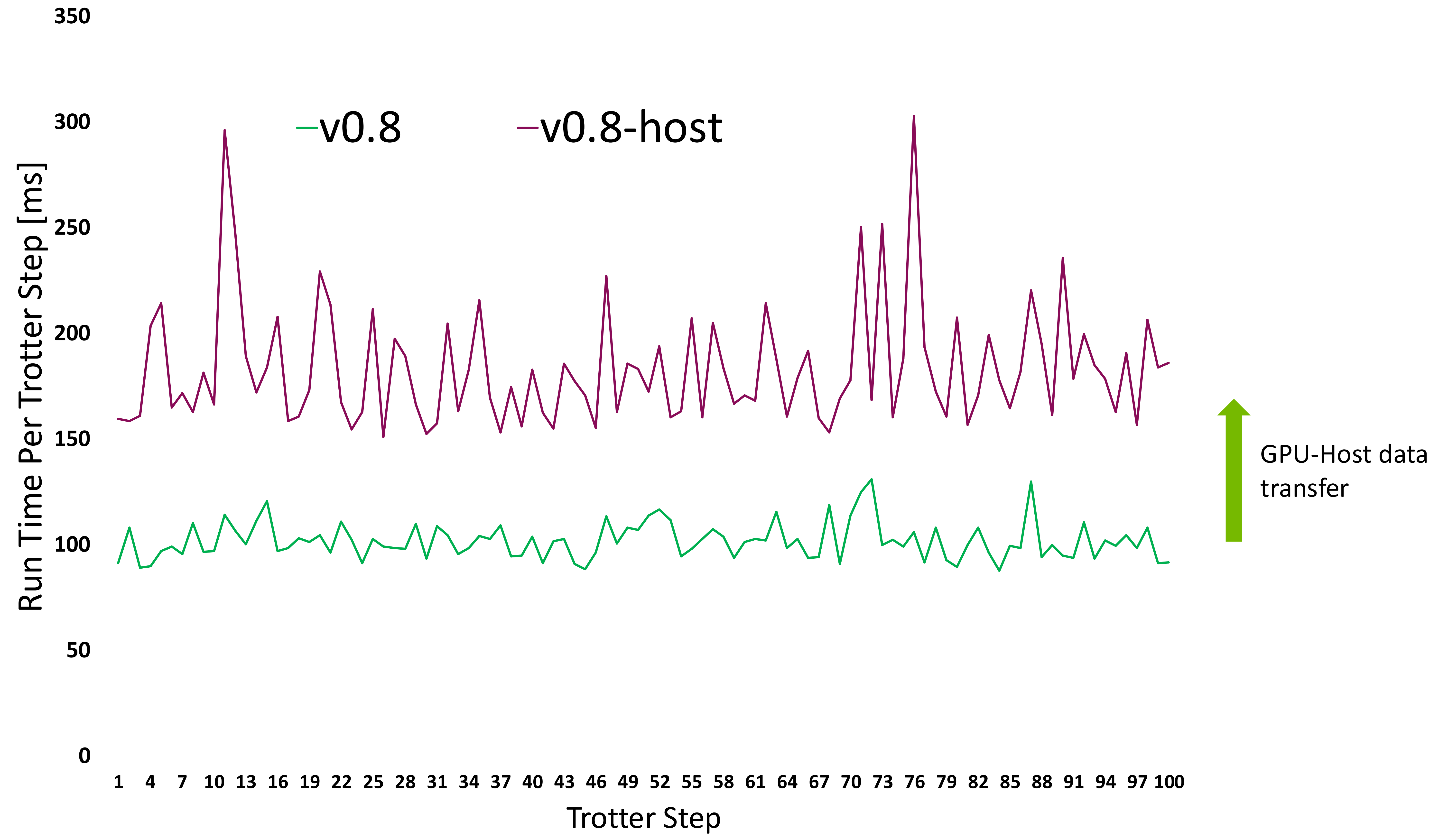
```
struct trotter {
  // Note: This performs a single-step Trotter on top of an initial state
vector
  void operator()(std::vector<cudaq::complex>& initial_state_vector,
cudaq::spin_op ham,
                  double dt) __qpu__ {
    cudaq::qvector q(initial_state_vector);
    ham.for_each_term([&](cudaq::spin_op &term) {
      const auto coeff = term.get_coefficient();
      const auto pauliWord = term.to_string(false);
      const double theta = coeff.real() * dt;
      cudaq::exp_pauli(dt, q, pauliWord.c_str());
    });
  }
};
```

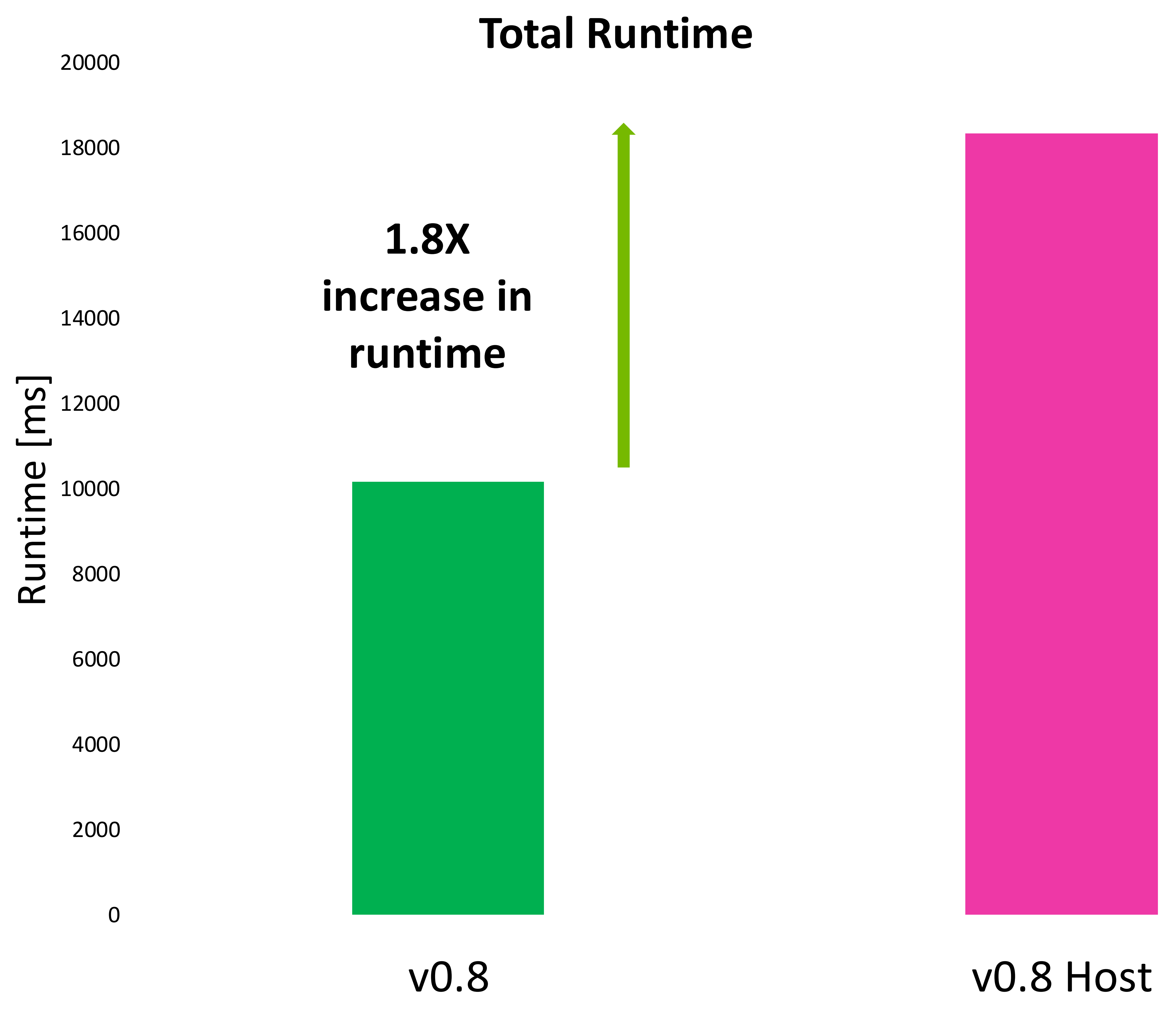
State initialization from host memory is supported (all backends).

Extra overhead of data transfer.

CUDA-Q: State Handling Example

Host vs. GPU State Data





The data transfer overhead (ratio) depends on the number of qubits (data transfer size) vs. the depth of the circuit (GPU runtime).
(Host – GPU BW << GPU memory BW)

Conclusion and Looking Forward

Areas of collaboration

- CUDA-Q is a system-level programming model and compilation platform geared toward enabling quantum acceleration to existing GPU Supercomputing architectures
- NVQ++ orchestrates a collection of modular tools enabling complex quantum-classical compilation workflows
- Explore quantum algorithm simulation scalability on GPU Supercomputing with NVIDIA GPUs and CUDA-Q Multi-QPU virtualization

Resources

Links

- CUDA-Q Repo for issues and contributions: NVIDIA/CUDA-Quantum (github.com)
- CUDA-Q documentation: CUDA-Q — NVIDIA CUDA-Q documentation
- Quantum computing technical blogs: Tag: Quantum Computing | NVIDIA Technical Blog
- CUDA-Q marketing page: CUDA-Q for Hybrid Quantum-Classical Computing | NVIDIA Developer

Documentation Reference

Quick Start

Multi-GPU Workflows

Simulator backends

Hardware backends

Python code examples

C++ code examples

Applications