Introduction to CUDA Quantum

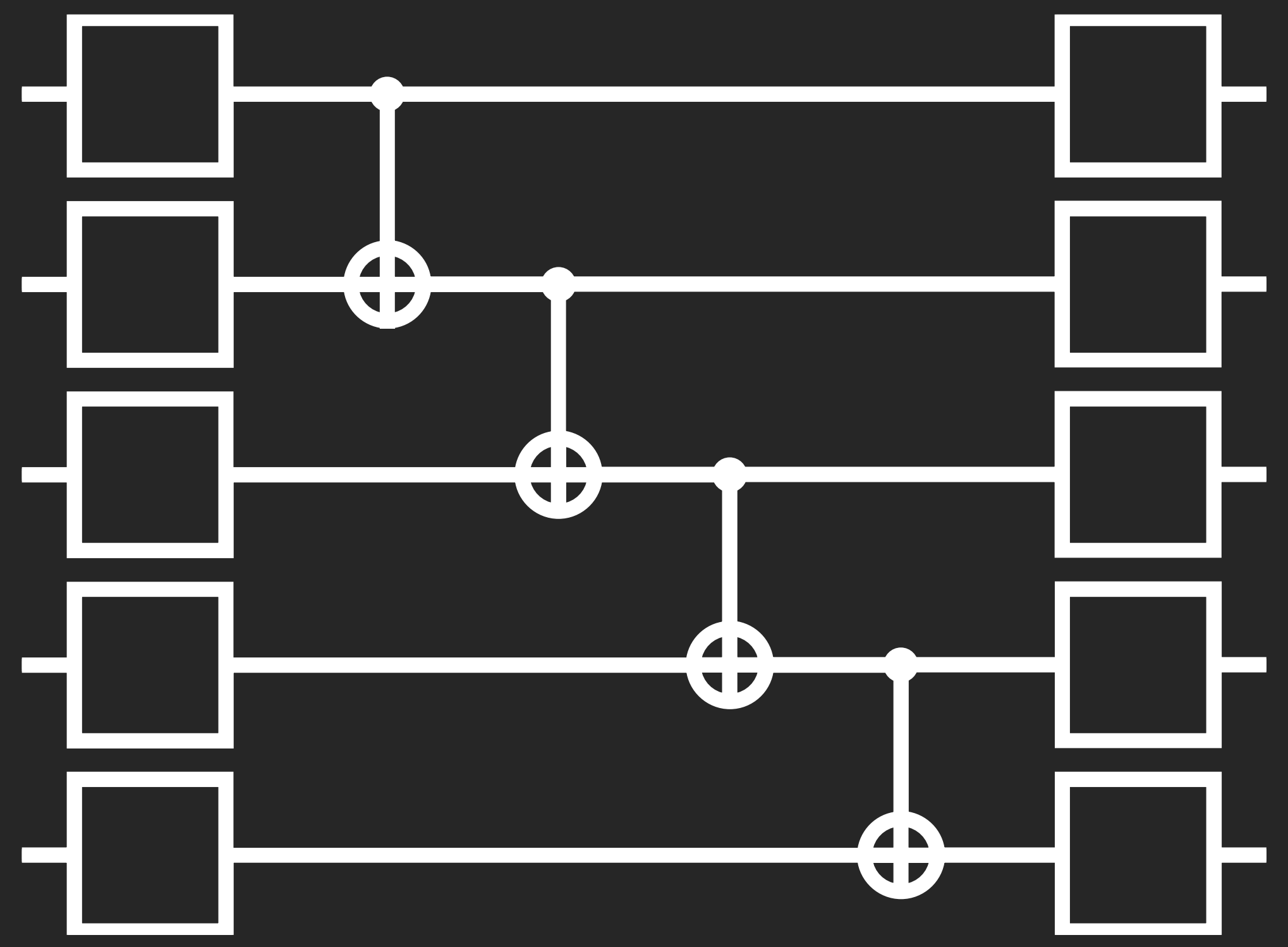
Mark Wolf, Technical Marketing Engineer - Quantum Computing

CU Boulder Hackathon – Feb 2024

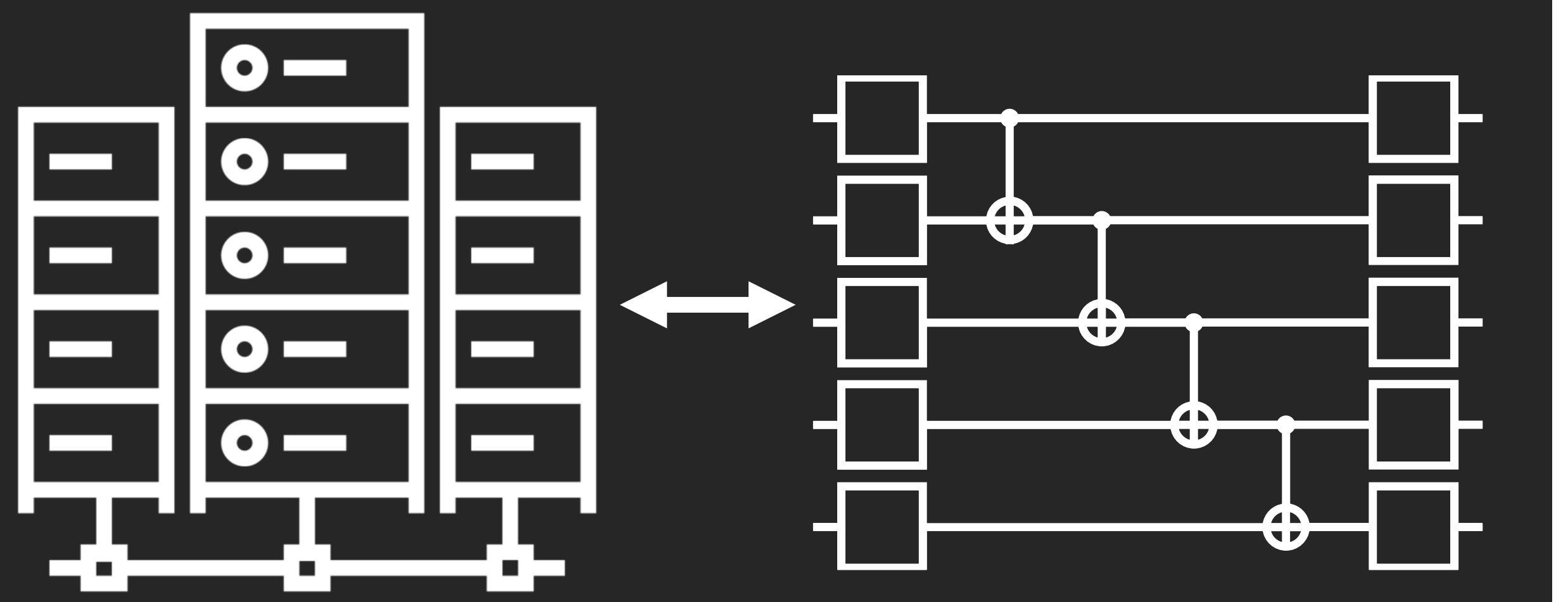
NVIDIA Quantum

Powering Quantum Simulation and Quantum-Integrated Accelerated Computing

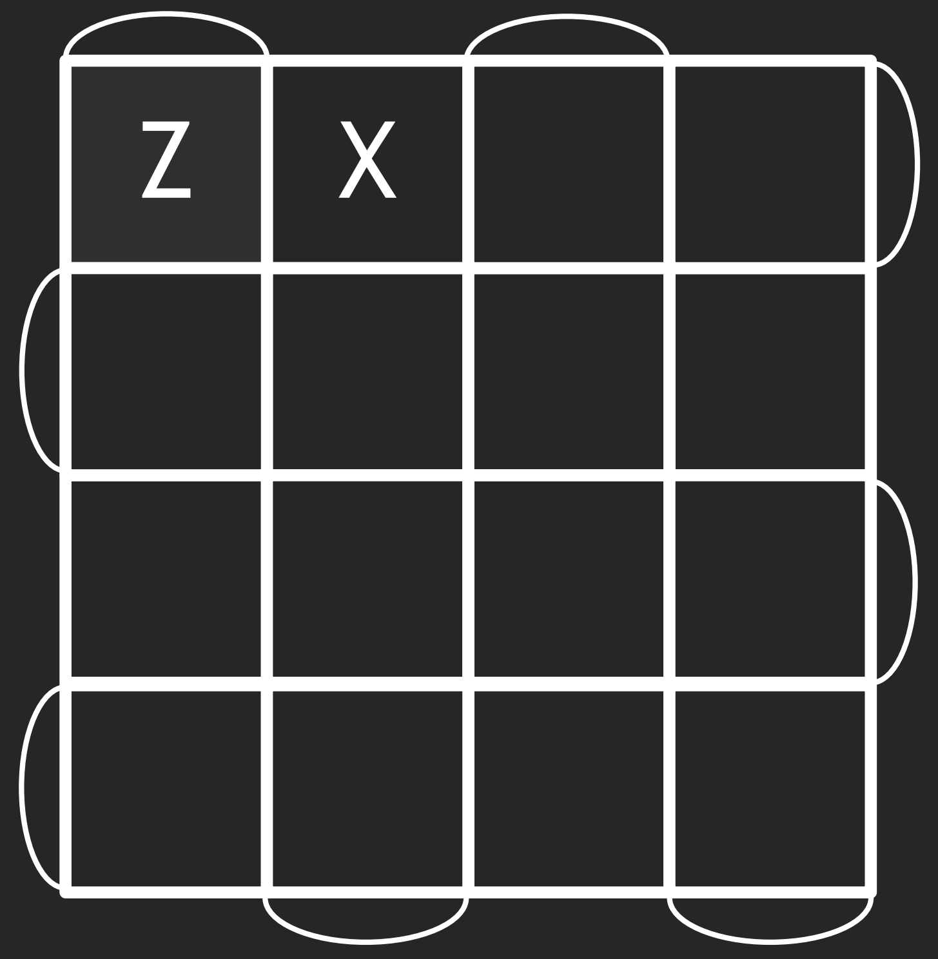
Quantum Algorithms Research



Quantum-Integrated Applications



Error Correction, Calibration, Control



cuQuantum
Accelerated Quantum Simulation

CUDA Quantum
Quantum-Classical Developer Platform

Quantum Integrated GPU Supercomputing
Grace Hopper | DGX | HGX | DGX Quantum

Motivation behind CUDA Quantum

Integrate quantum computers seamlessly with the modern scientific computing ecosystem

We believe quantum programming should be:

Easy to Learn:

- No domain-specific expertise or new language required

- Integrable with today's scientific computing and AI workflows

Fast

- Performant compilation with no quantum-specific bottlenecks

-Integrated state-of-the-art simulation methods

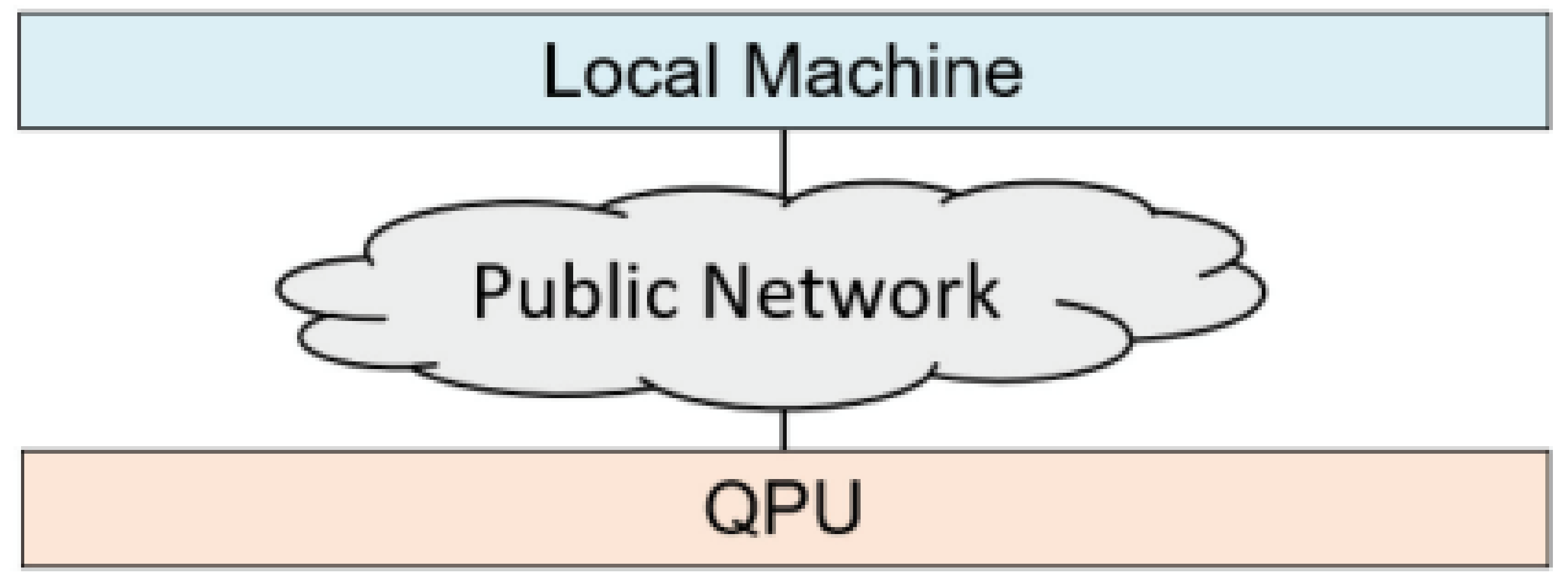
- Straightforward interoperability with classical accelerated computing

Flexible

-Easy porting between classical computers, simulated QPUs, and real QPUs

Scalable

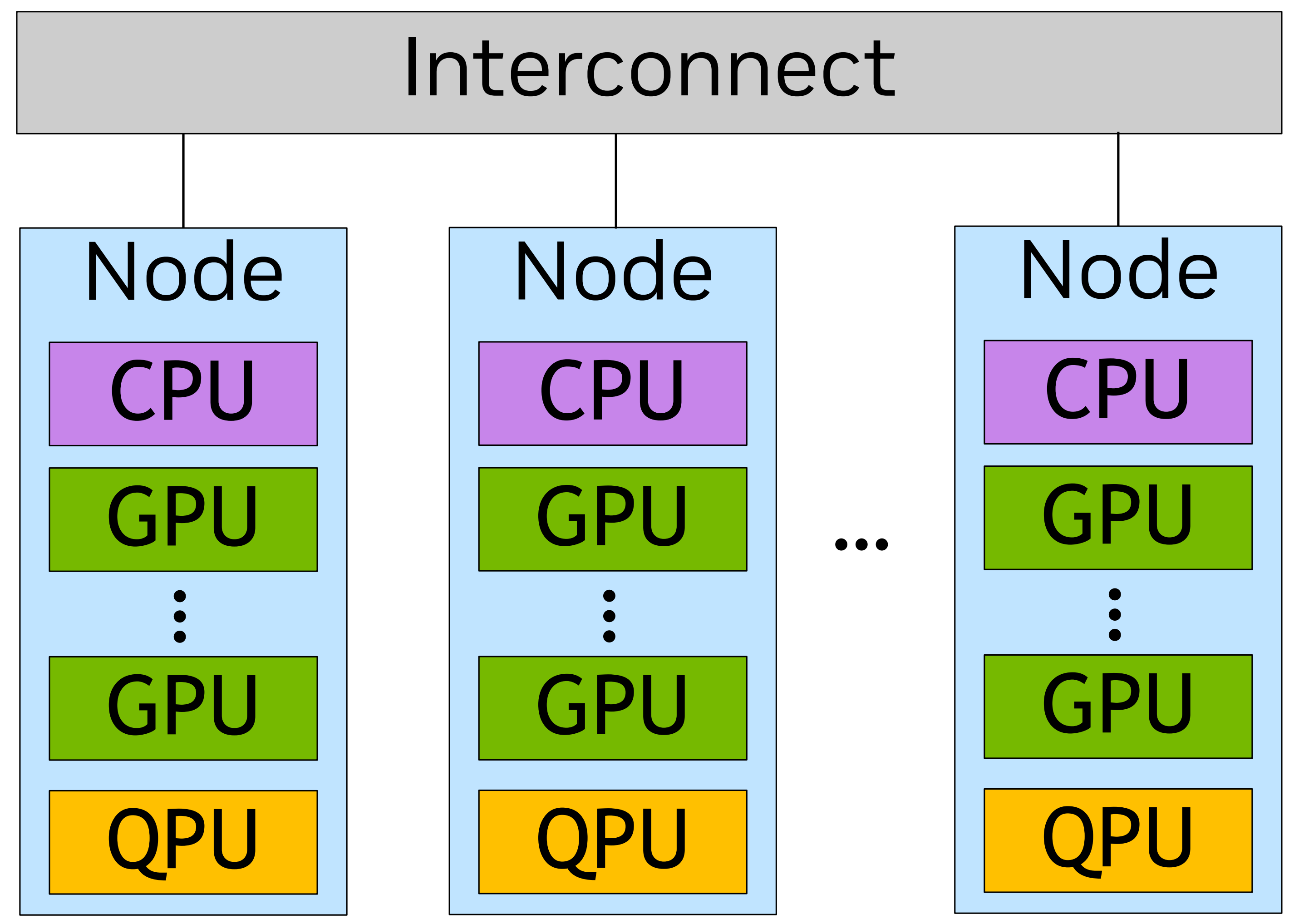
-Bring supercomputers to bear to advance quantum research



Quantum Programming Today

Great for early experimentation.

vs…



Where we need to get…

Application-level Quantum Programming

Hybrid quantum-classical applications at scale.

Figure adapted from:
Quantum Computers for High-Performance Computing.
Humble, McCaskey, Lyakh, Gowrishankar, Frisch, Monz.
IEEE Micro Sept 2021. 10.1109/MM.2021.3099140

Building a basic quantum Kernel

Optional subtitle

```python
# Import the CUDA Quantum module
import cudaq

# We begin by defining the `Kernel` that we will construct our
kernel = cudaq.make_kernel()

# Next, we can allocate qubits to the kernel via `qalloc(qubit_count)`.
# An empty call to `qalloc` will return a single qubit.
qubit = kernel.qalloc()

# Now we can begin adding instructions to apply to this qubit!
# Some single qubit gates that are supported by CUDA Quantum.
kernel.h(qubit)
kernel.x(qubit)
kernel.y(qubit)
kernel.z(qubit)
kernel.t(qubit)
kernel.s(qubit)

# Next, we add a measurement to the kernel so that we can sample
# the measurement results on our simulator!
kernel.mz(qubit)
```

Algorithmic primitives

Optional subtitle

- cudaq.sample()
  - Samples a kernel
- cudaq.spin_op()
  - Defines Pauli spin operators
- cudaq.observe()
  - Determines expectation value given operator and kernel

Build and Sample a Bell State

Optional subtitle

- Samples a kernel and returns a dictionary of measurement outcomes and respective counts
- Samples 1000 shots by default

```
kernel = cudaq.make_kernel()
qubit = kernel.qalloc(2)

kernel.h(qubit[0])
kernel.cx(qubit[0], qubit[1])

kernel.mz(qubit)

sample_result = cudaq.sample(kernel, shots_count=2000)

print(sample_result)

{ 00:1007 11:993 }
```

Extracting data from a sample

Optional subtitle

```
print(f"most probable = {sample_result.most_probable()}")
print(f"count for 11 = {sample_result.count('11')}")
print(f"probability for 11 = {sample_result.probability('11')}")
print(f"Marginal counts for qubit 0 = sample_result.get_marginal_counts([0])}")
```

```
most probable = 00
Count for 11 = 993
Probability for 11 = 0.4965 # of sample
Marginal counts for qubit 0 = { 0:1007 1:993 }
```

```
sample_result.clear()
print(sample_result)
```

```
{ }
```

cudaq.spin_op()

Optional subtitle

- Can be used to compose large, more complex linear combinations of Pauli tensor products

Let's take the Hamitonian H such that, H = $Z_0 \otimes I_1 + I_0 \otimes X_1 + Y_0 \otimes I_1 + Y_0 \otimes Y_1$.

```python
# Importing the spin_op
from cudaq import spin

# the observable
hamiltonian = spin.z(0) + spin.x(1) + spin.y(0) + spin.y(0)*spin.y(1)

# add some more terms
for i in range(2):
  hamiltonian += -2.0*spin.z(i)*spin.z(i+1)

print(hamiltonian)
```

```
[-2+0j] IZZ
[1+0j] ZII
[1+0j] YII
[1+0j] IXI
[1+0j] YYI
[-2+0j] ZZI
```

Some other helpful spin_op() methods

Optional subtitle

- for_each_pauli()
  - Loops over each Pauli element in a term and applies a function
- get_qubit_count()
  - Returns the number of qubits the operator is on
- get_term_count()
  - Return the number of terms in this operator
- to_string()
  - Returns string representation of operator
- distribute_terms(N)
  - Breaks operator into chunks of size N terms

cudaq.observe()

Optional subtitle

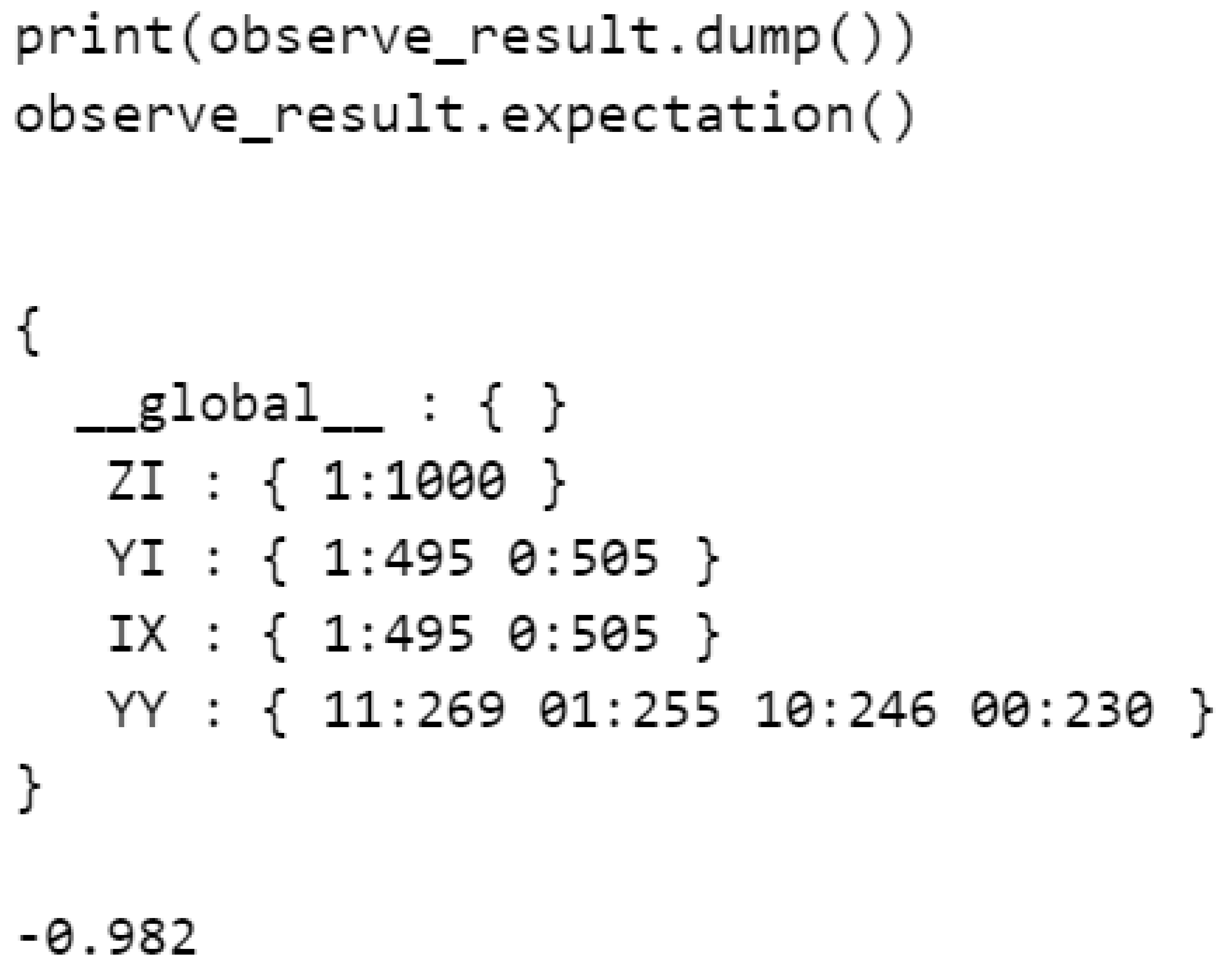
- Computes an expectation value given a kernel and operator.

```
# the observable
hamiltonian = spin.z(0) + spin.x(1) + spin.y(0) + spin.y(0)*spin.y(1)

# the kernel
kernel = cudaq.make_kernel()
qreg = kernel.qalloc(2)
kernel.x(qreg[0])

observe_result = cudaq.observe(kernel, hamiltonian, shots_count=1000)
```

cudaq.observe()

Optional subtitle

- dump() returns the raw data for each term
- expectation() prints the expectation value



```
print(observe_result.dump())
observe_result.expectation()


{
  __global__ : { }
   ZI : { 1:1000 }
   YI : { 1:495 0:505 }
   IX : { 1:495 0:505 }
   YY : { 11:269 01:255 10:246 00:230 }
}

-0.982
```

Parameterized Circuits (multiple parameters)

Optional subtitle

```
# the observable
hamiltonian = 5.907 - 2.1433 * spin.x(0) * spin.x(1) \
            - 2.1433 * spin.y(0) * spin.y(1) + 0.21829 * spin.z(0) \
            - 6.125 * spin.z(1)

# parameterized cudaq kernel, the parameter is of type list
kernel, theta = cudaq.make_kernel(list)
q = kernel.qalloc(2)
kernel.x(q[0])
kernel.ry(theta[0], q[1])
kernel.ry(theta[1], q[1])
kernel.cx(q[1], q[0])

# This time a list of thetas is provided
observe_result = cudaq.observe(kernel, hamiltonian, [.59,.75])
observe_result.expectation()

0.2830166240322214
```

Scaling applications in CUDA Quantum

Optional subtitle

- CUDA Quantum can target a variety of CPU, GPU, and QPU backends
- The default “nvidia” will target a single GPU if available and otherwise fall back to CPU
- Other Important Targets:
  - “nvidia-mgpu” – pools the memory of multiple GPUs for a SV simulation
  - “nvidia-mqpu” – enables programming of multi-QPU programs which execute on GPUs
  - “density-matrix-cpu” – enables noisy simulations via density matrix calculations
  - Physical QPUs – hardware specific targets that correspond to QPU hardware backends

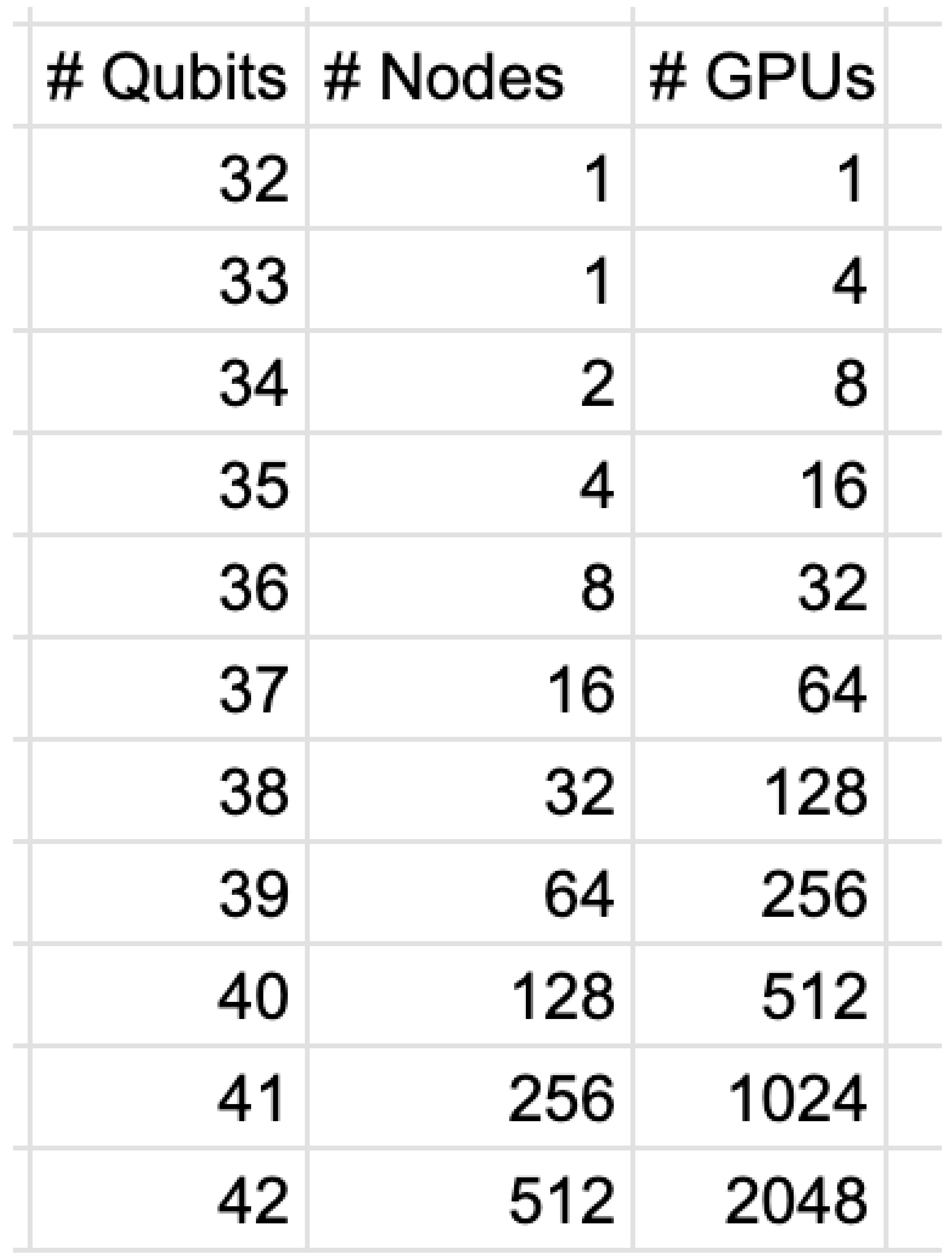
```python
targets = cudaq.get_targets()
for target in targets:
    print(target)
```

Increase the Number of Qubits (nvidia-mgpu)

Optional subtitle

- The exponential scaling of the state vector requires pooling GPU memory to simulate systems a of ~32 or more.

- The example below shows how far we were able to scale a GHZ state prep.



# Qubits	# Nodes	# GPUs
32	1	1
33	1	4
34	2	8
35	4	16
36	8	32
37	16	64
38	32	128
39	64	256
40	128	512
41	256	1024
42	512	2048

Scaling across multiple QPUs (simulated by GPUs “nvidia-mqpu”)

Optional subtitle

- As a rule of thumb, we can parallelize over any of the input parameters of sample and observe.
- Some examples:
  - Asynchronous sampling
  - Hamiltonian batching
  - Parameter batching

Asynchronous sampling

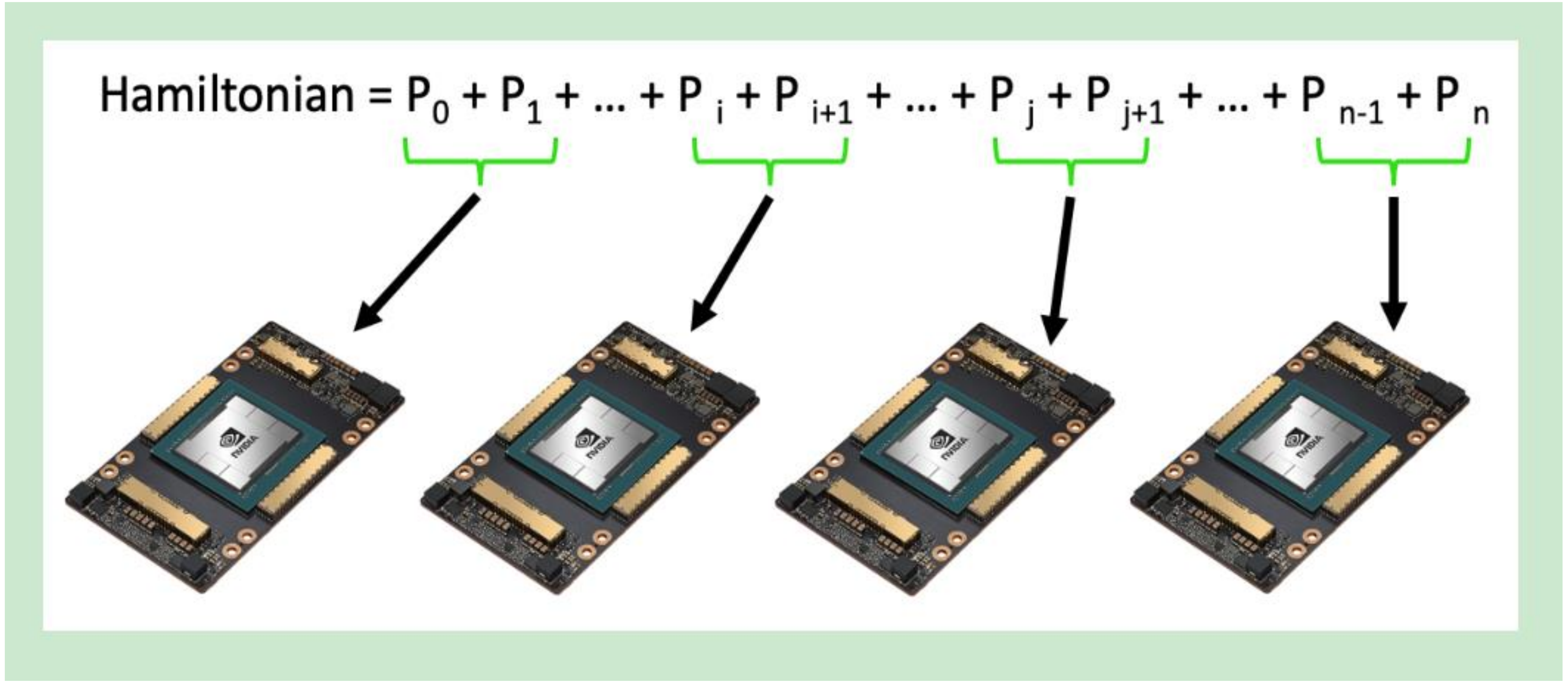
Optional subtitle

```
#Set the target
cudaq.set_target("nvidia-mqpu")
target = cudaq.get_target()

num_qpus = target.num_qpus()
print("Number of QPUs:", num_qpus)

kernel = cudaq.make_kernel()
qubits = kernel.qalloc(2)
kernel.h(qubits[0])
kernel.cx(qubits[0], qubits[1])
kernel.mz(qubits)

futures = []
for i in range(num_qpus):
  futures.append(cudaq.sample_async(kernel, qpu_id=i))

for count in futures:
    print(count.get())
```

```
Number of QPUs: 4
{ 00:470 11:530 }
{ 00:495 11:505 }
{ 00:508 11:492 }
{ 00:504 11:496 }
```

Hamiltonian Batching

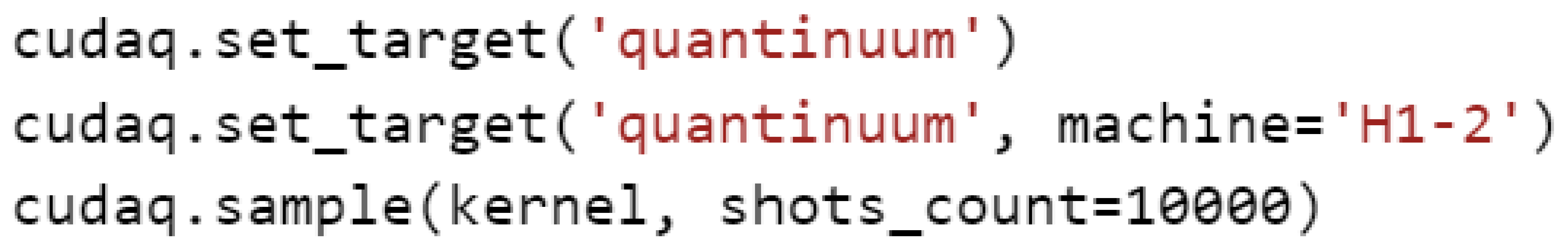
Optional subtitle


Hamiltonian = $P_0 + P_1 + \ldots + P_i + P_{i+1} + \ldots + P_j + P_{j+1} + \ldots + P_{n-1} + P_n$

Target QPUs

Optional subtitle

- Can target QPUs and emulators from:
  - Quantinuum
  - IonQ
  - IQM
  - OQC



```
cudaq.set_target('quantinuum')
cudaq.set_target('quantinuum', machine='H1-2')
cudaq.sample(kernel, shots_count=10000)
```

The VQE Wrapper Makes This Easier

Optional subtitle

```
# Parameterized circuit with theta as the parameter
kernel, theta = cudaq.make_kernel(list)
qreg = kernel.qalloc(2)
kernel.x(qreg[0])
kernel.ry(theta[0], qreg[1])

# Hamiltonian operator
hamiltonian = spin.z(0) + spin.x(1) + spin.y(0)

# Initialize the gradient-free optimizer COBYLA
optimizer = cudaq.optimizers.COBYLA()

# Specify the number of iterations (optional)
optimizer.max_iterations = 50

# Carry out the optimization using VQE wrapper
opt_value, opt_theta = cudaq.vqe(kernel=kernel,
                                 spin_operator=hamiltonian,
                                 optimizer=optimizer,
                                 parameter_count=1)

print(f"\nminimized <H> = {round(opt_value,16)}")
print(f"optimal theta = {round(opt_theta[0],16)}")

minimized <H> = -1.9999997019767757
optimal theta = -1.5707963267948963
```

Some Other Helpful Things

Conditional Measurement

- Execute an operation in a kernel only if control qubit is measured as 1.

```
kernel = cudaq.make_kernel()
qubit = kernel.qalloc()

def then_function():
    kernel.x(qubit)

kernel.x(qubit)

# Measure the qubit.
measurement_ = kernel.mz(qubit)

# applies “then_function” if qubit is a 1 using “c_if”
kernel.c_if(measurement_, then_function)

# Measure the qubit again.
result = cudaq.sample(kernel, shots_count=30)

result.dump()

{
  __global__ : { 0:30 }
   auto_register_0 : { 1:30 }
}
```

Noise Models (Packaged)

Optional subtitle

- Noise models are optional inputs to sample or observe.

```python
# Set the target to our density matrix simulator.
cudaq.set_target('density-matrix-cpu')

# Define an empty noise model
noise = cudaq.NoiseModel()

# Bit flip channel with `1.0` probability of the qubit flipping 180 degrees.
bit_flip = cudaq.BitFlipChannel(1.0)

# apply bit_flip to every X gate on qubit 0
noise.add_channel('x', [0], bit_flip)

# construct a kernel
kernel = cudaq.make_kernel()
qubit = kernel.qalloc()

# Apply an X-gate to the qubit.
# It will remain in the |1> state with a probability of `1 - p = 0.0`.
kernel.x(qubit)
kernel.mz(qubit)
```

Executing the Noise Model

Optional subtitle

```
# noisy simulation
noisy_result = cudaq.sample(kernel, noise_model=noise)
noisy_result.dump()

# noiseless simulation
noiseless_result = cudaq.sample(kernel)
noiseless_result.dump()

Noisy Result { 0:1000 }
Noiseless Result{ 1:1000 }
```

Building a Custom Noise Model

Optional subtitle

- Custom noise models are defined with Kraus operators

```python
# Set the target to our density matrix simulator.
cudaq.set_target('density-matrix-cpu')

# Define noise model
noise = cudaq.NoiseModel()

#Define Kraus operators as functions for ease of control
def kraus_operators(probability):
    kraus_0 = np.array([[1, 0], [0, np.sqrt(1 - probability)]],
                       dtype=np.complex128)
    kraus_1 = np.array([[0, 0], [np.sqrt(probability), 0]],
dtype=np.complex128)
    return [kraus_0, kraus_1]

# Manually defined amplitude damping channel with `1.0` probability
# of the qubit decaying to the ground state.
amplitude_damping = cudaq.KrausChannel(kraus_operators(1.0))

# Apply channel to qubit 0 Hadamard gates
noise.add_channel('h', [0], amplitude_damping)

# construct a simple kernel
kernel = cudaq.make_kernel()
qubit = kernel.qalloc()
kernel.h(qubit)
kernel.mz(qubit)
```

Deploy Custom Noise Model

Optional subtitle

```
# noisy
noisy_result = cudaq.sample(kernel, noise_model=noise)
noisy_result.dump()

# noiseless
noiseless_result = cudaq.sample(kernel)
noiseless_result.dump()
{ 0:1000 }
{ 0:478 1:522 }
```

Summary

Optional subtitle

- CUDA Q is a platform for enabling quantum accelerated supercomputing in a heterogenous (CPU, GPU, QPU) environment.
- Algorithmic primitives facilitate easy and flexible design of quantum programs.
  - Sample
  - Spin_op
  - Observe
  - VQE
  - Noise_channel
- CUDA Q provides tools for targeting multiple backends:
  - Density Matrix Simulator
  - Single GPU
  - Multiple GPUs with pooled to simulate one QPU
  - Multiple GPUs simulating multiple QPUs
  - Physical QPUs
- The documentation can be found here along with:
  - Tutorials
  - Examples

Some useful links

Optional subtitle

Installation: https://nvidia.github.io/cuda-quantum/latest/install.html

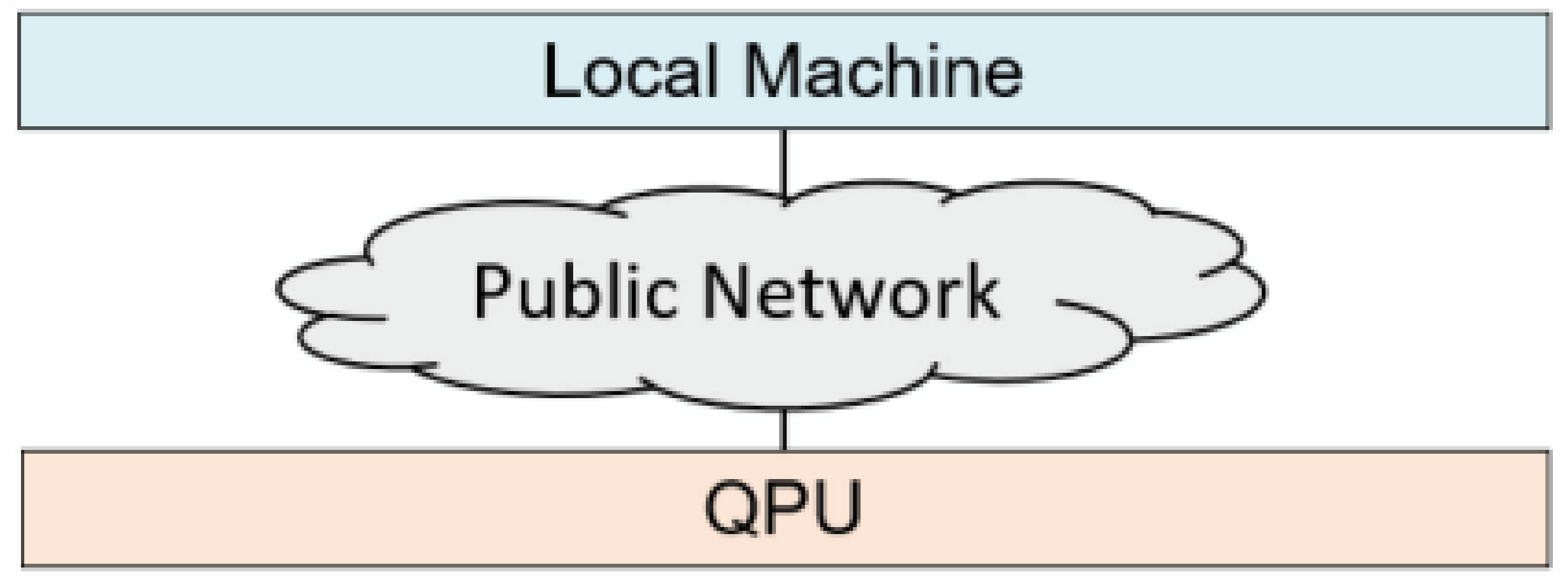
CUDA Quantum Github Repo: `https://github.com/NVIDIA/cuda-quantum`

Tutorials: CUDA Quantum Tutorials — NVIDIA CUDA Quantum documentation

Motivation behind CUDA Quantum

Integrate quantum computers seamlessly with the modern scientific computing ecosystem

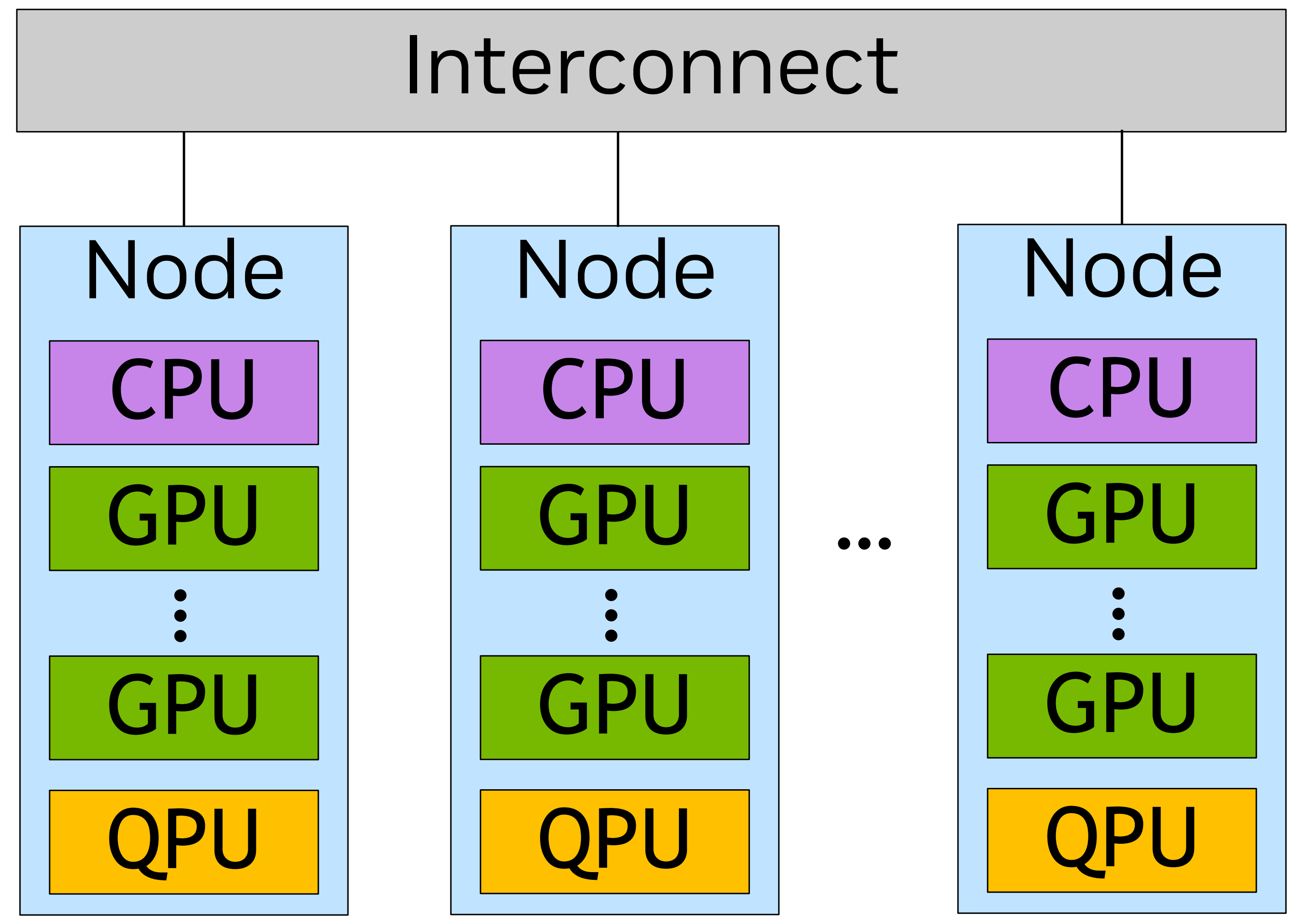
- Research Centers worldwide are focused on integration of quantum computers with classical supercomputers
- Quantum computers will accelerate some of today's most important computational problems and HPC workloads
  - Quantum chemistry, Materials simulation, AI
- We also expect CPUs and GPUs to be able to enhance the performance of QPUs
  - Classical preprocessing (circuit optimization) and postprocessing (error correction)
  - Optimal control and QPU calibration
- Want to enable researchers to seamlessly integrate CPUs, GPUs, and QPUs
  - Develop new hybrid applications and accelerate existing ones
  - Leverage classical GPU computing for control, calibration, error mitigation, and error correction



Quantum Programming Today

Great for early experimentation.

vs...



Where we need to get...

Application-level Quantum Programming

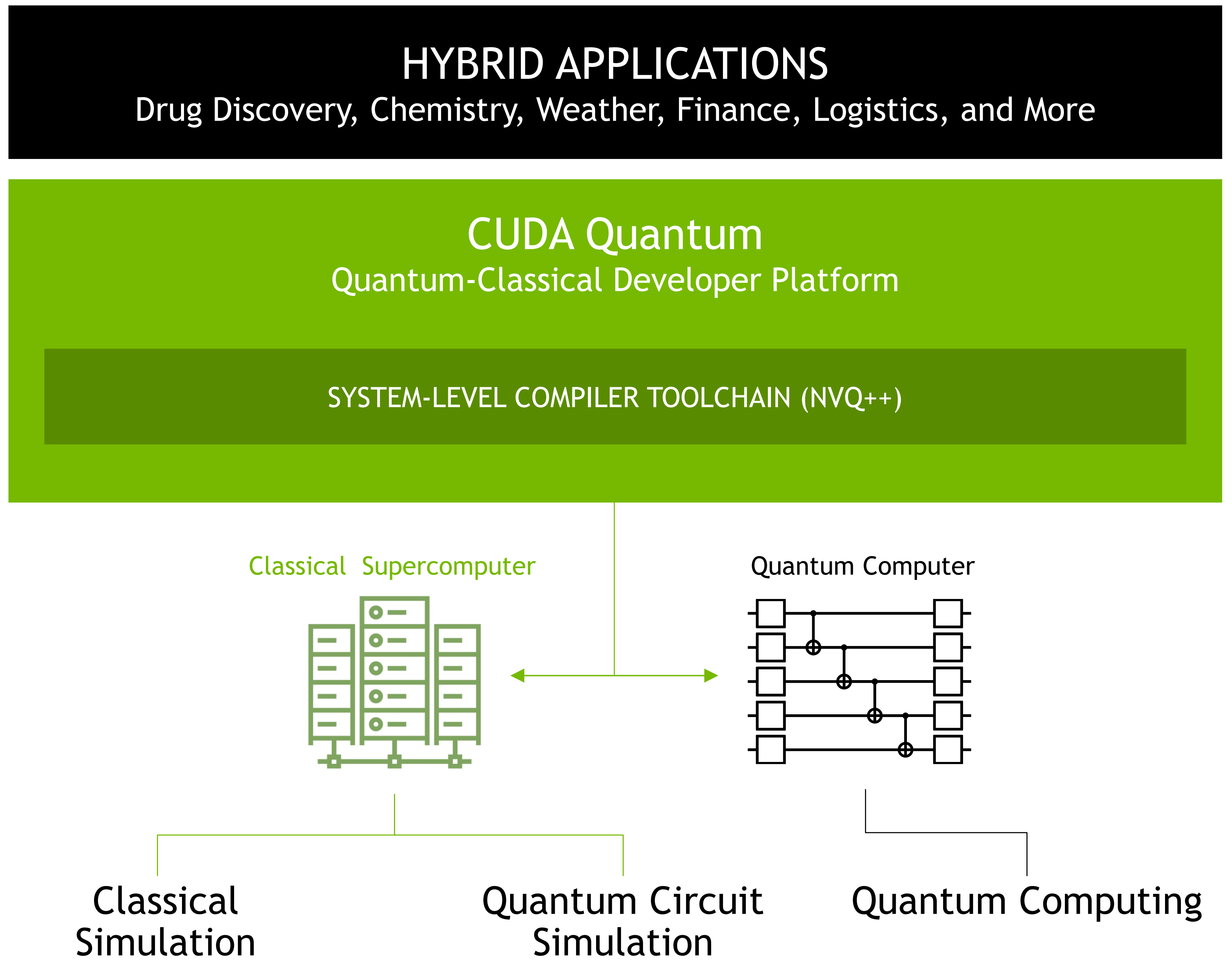
Hybrid quantum-classical applications at scale.

Figure adapted from:
Quantum Computers for High-Performance Computing.
Humble, McCaskey, Lyakh, Gowrishankar, Frisch, Monz.
IEEE Micro Sept 2021. 10.1109/MM.2021.3099140

CUDA Quantum: OSS Platform for Quantum Accelerated Supercomputing

Develop applications for integrated quantum-classical computing

CUDA QUANTUM PLATFORM



HYBRID APPLICATIONS
Drug Discovery, Chemistry, Weather, Finance, Logistics, and More
CUDA Quantum
Quantum-Classical Developer Platform
SYSTEM-LEVEL COMPILER TOOLCHAIN (NVQ++)
Classical Supercomputer
Quantum Computer
Classical Simulation
Quantum Circuit Simulation
Quantum Computing

CUDA QUANTUM FEATURES

Single source C++ and Python programming models

High-performance compiler for hybrid GPU/CPU/QPU systems

QPU Agnostic – Works with any type of QPU, emulated or physical

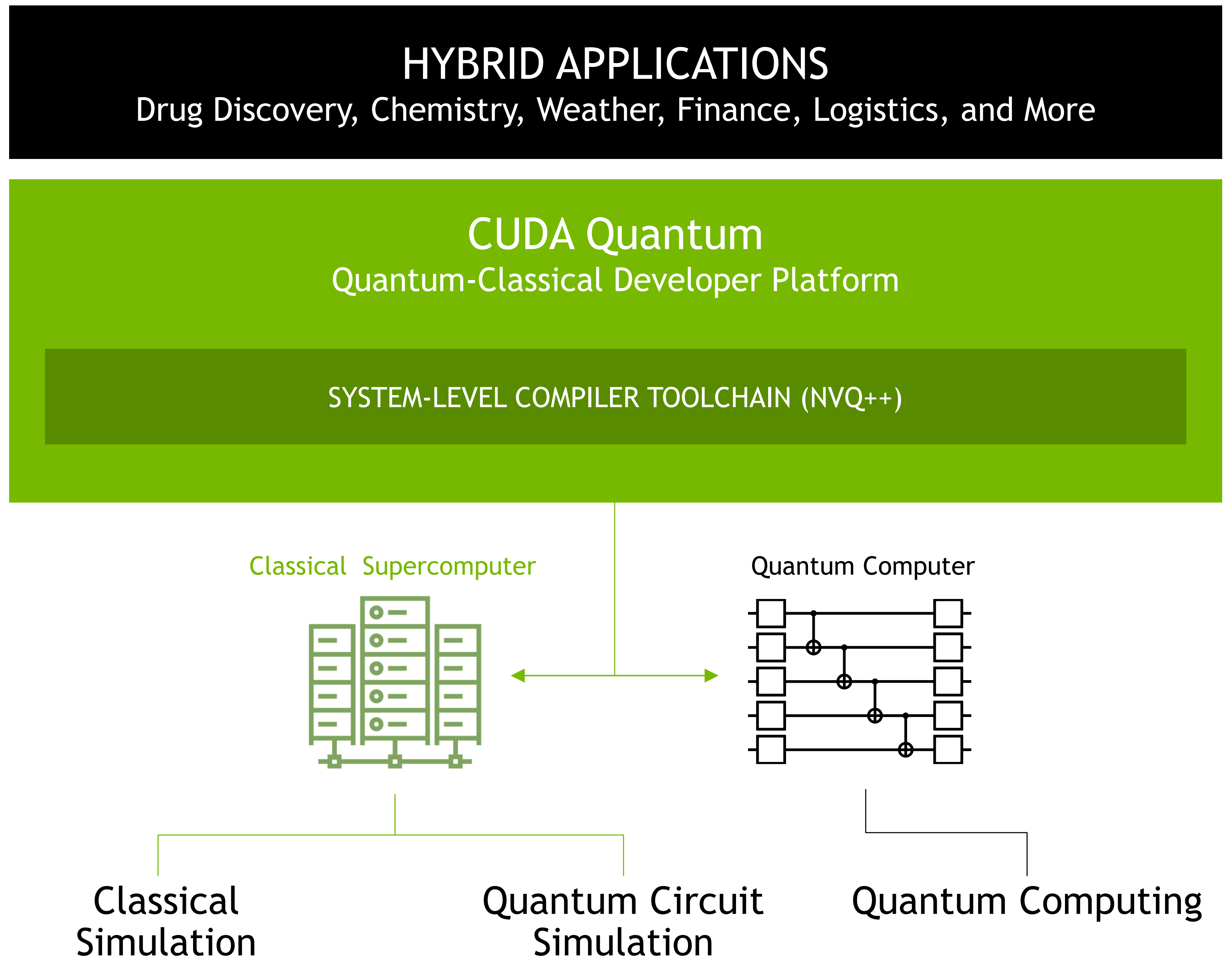
Interoperable with leading scientific computing and AI tools

github.com/nvidia/cuda-quantum | https://catalog.ngc.nvidia.com/orgs/nvidia/containers/cuda-quantum

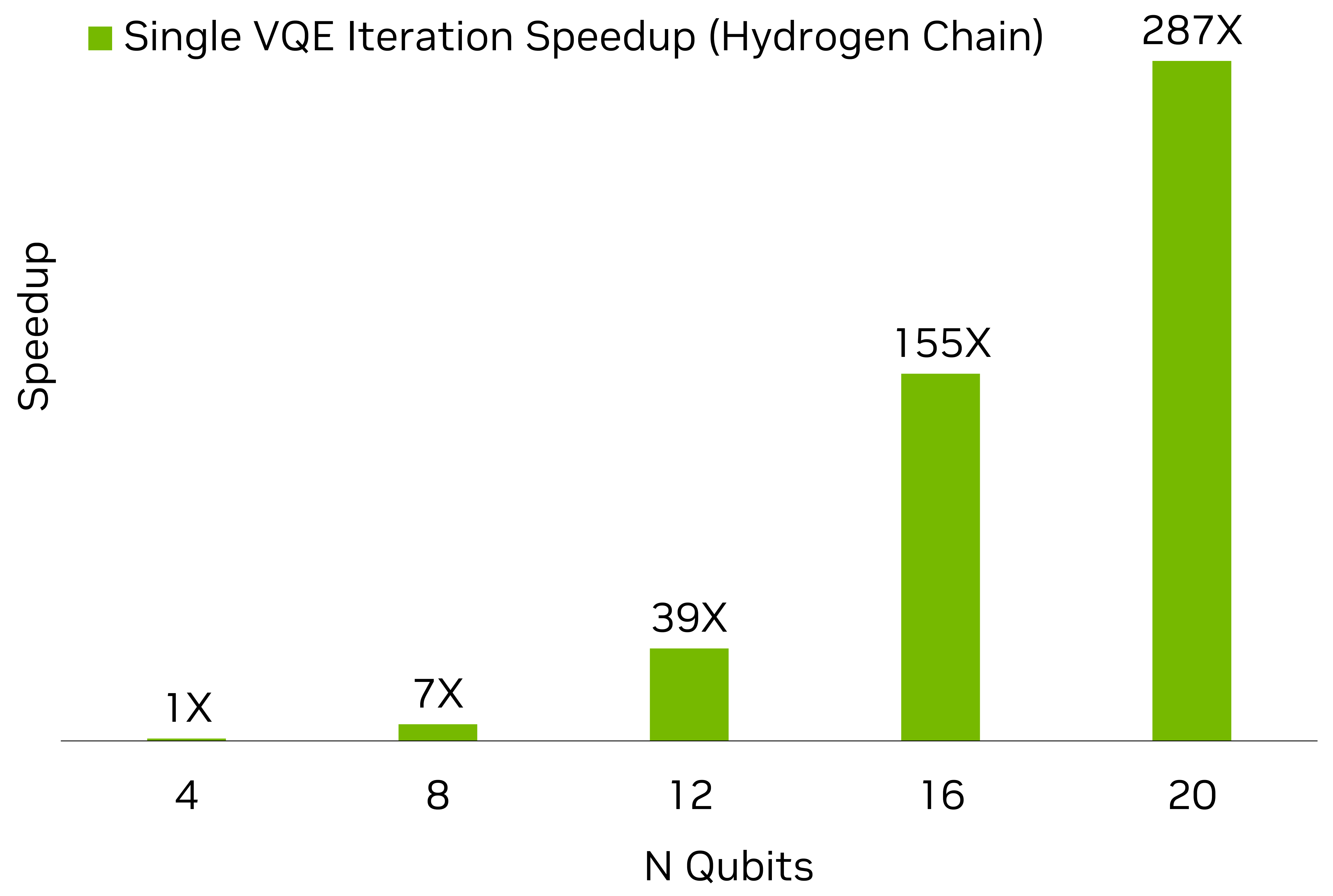
CUDA Quantum: OSS Platform for Quantum Accelerated Supercomputing

Develop applications for integrated quantum-classical computing

CUDA QUANTUM PLATFORM


HYBRID APPLICATIONS
Drug Discovery, Chemistry, Weather, Finance, Logistics, and More
CUDA Quantum
Quantum-Classical Developer Platform
SYSTEM-LEVEL COMPILER TOOLCHAIN (NVQ++)
Classical Supercomputer
Quantum Computer
Classical Simulation
Quantum Circuit Simulation
Quantum Computing

CUDA QUANTUM FEATURES



github.com/nvidia/cuda-quantum | https://catalog.ngc.nvidia.com/orgs/nvidia/containers/cuda-quantum