NVIDIA

Introduction to CUDA-Quantum

Yang Juntao, Solutions Architect, Nvidia Singapore

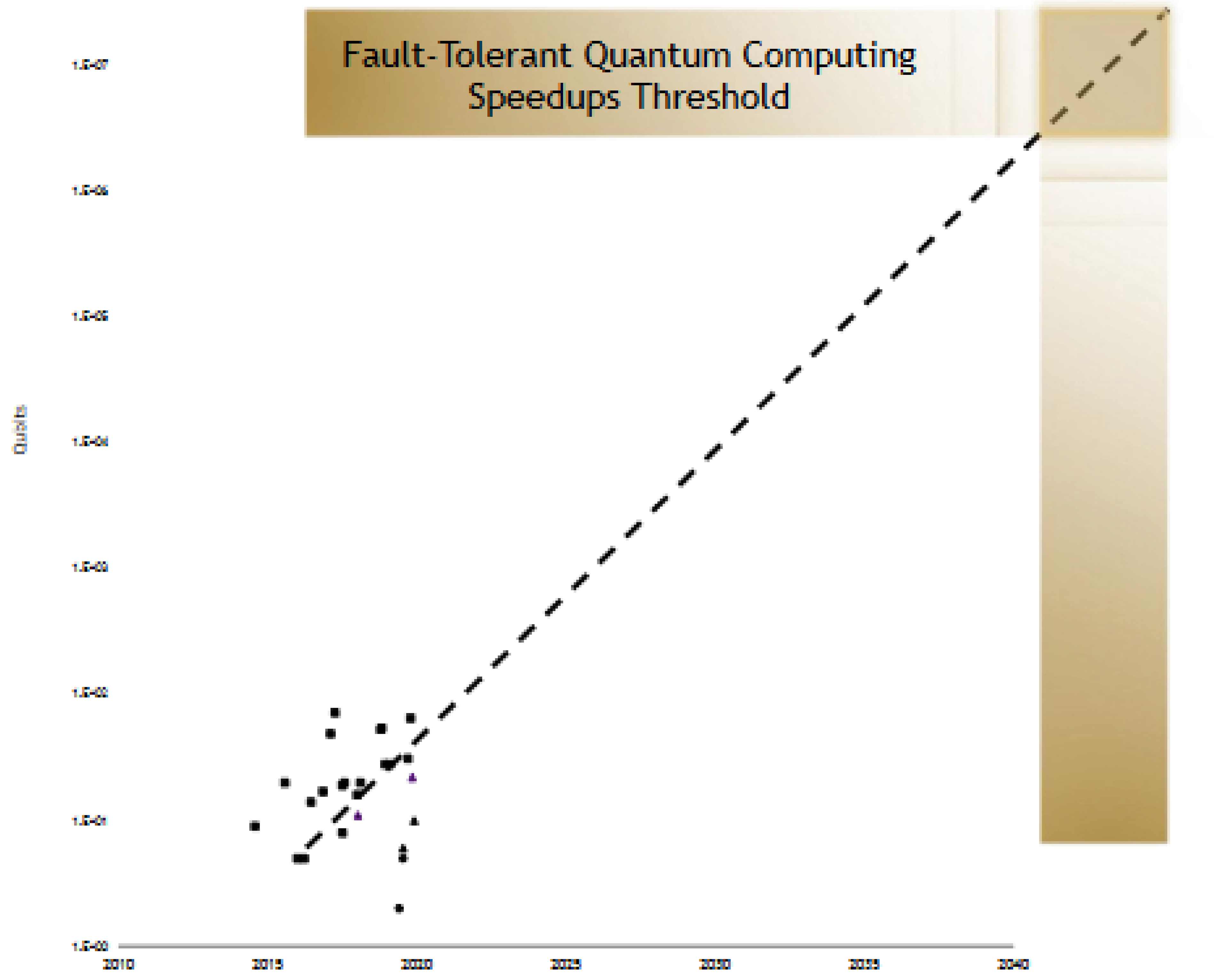
Agenda

- The Motivation behind CUDA Quantum
- A taste of CUDA Quantum via an example
- The CUDA Quantum Language Details
- NVQ++: The C++ Compiler for Quantum Computing
- CUDA Quantum via Examples
- CUDA Quantum Python and Examples

Worldwide Effort Towards a New Computing Model

QUANTUM SYSTEMS SCALING EXPONENTIALLY

Useful, Fault-Tolerant Qubit Scale Could Be Achieved in 15 to 20 Years



GOVERNMENT

22+

National Quantum Initiatives

INDUSTRY

70%

Of companies have quantum Initiatives

HIGHER ED/RESEARCH

2,100+

QC Research Papers

TECHNOLOGY

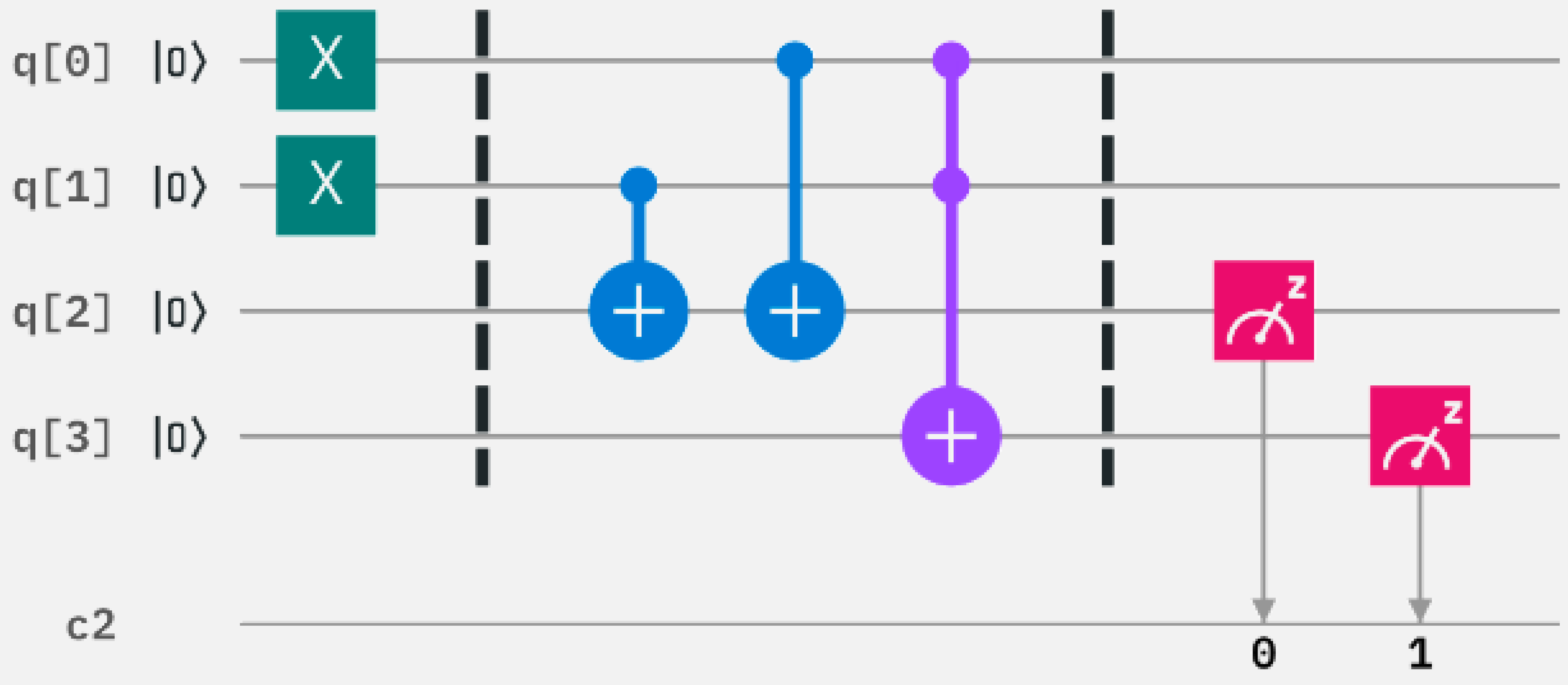
250+

QC Startups

GPU Supercomputing and Quantum

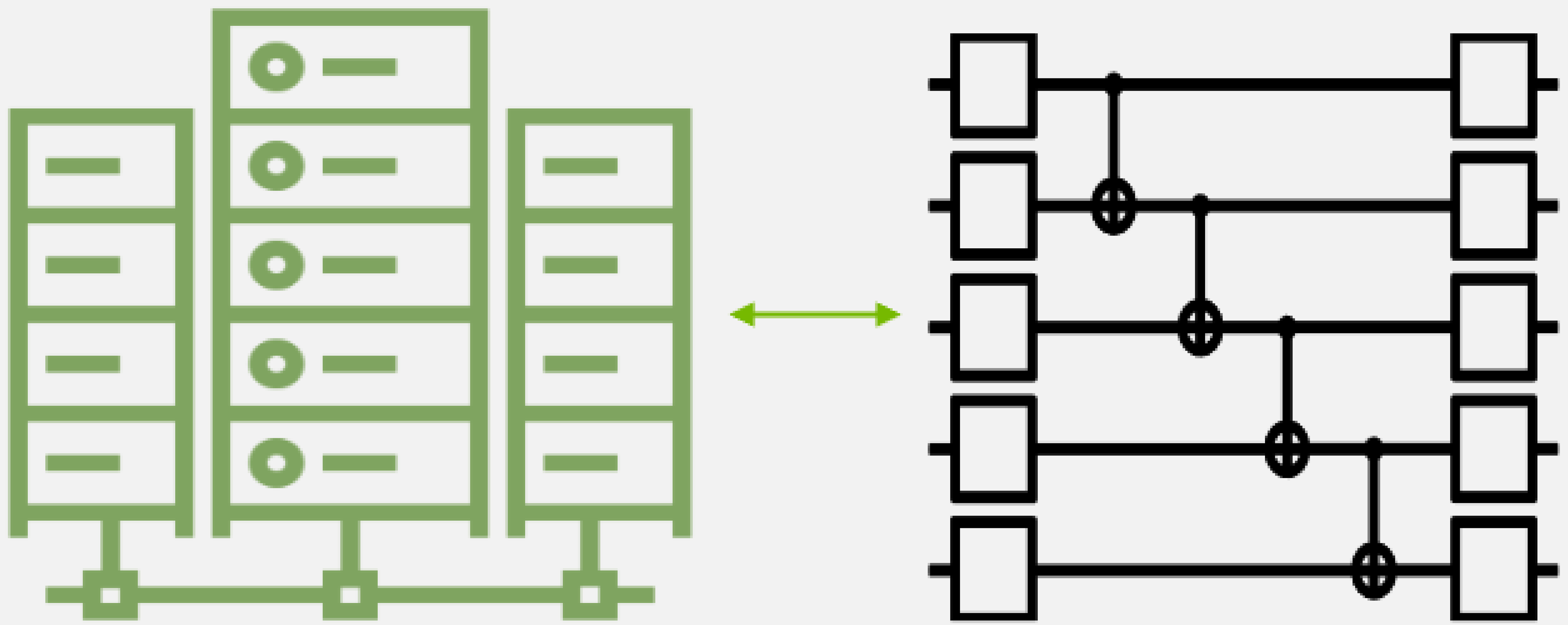
Hybrid Quantum-Classical Computing

QUANTUM SIMULATION



- Develop algorithms at scale of valuable quantum computing
- Discover use cases with quantum advantage
- Design and validate future hardware

HYBRID QUANTUM-CLASSICAL COMPUTING

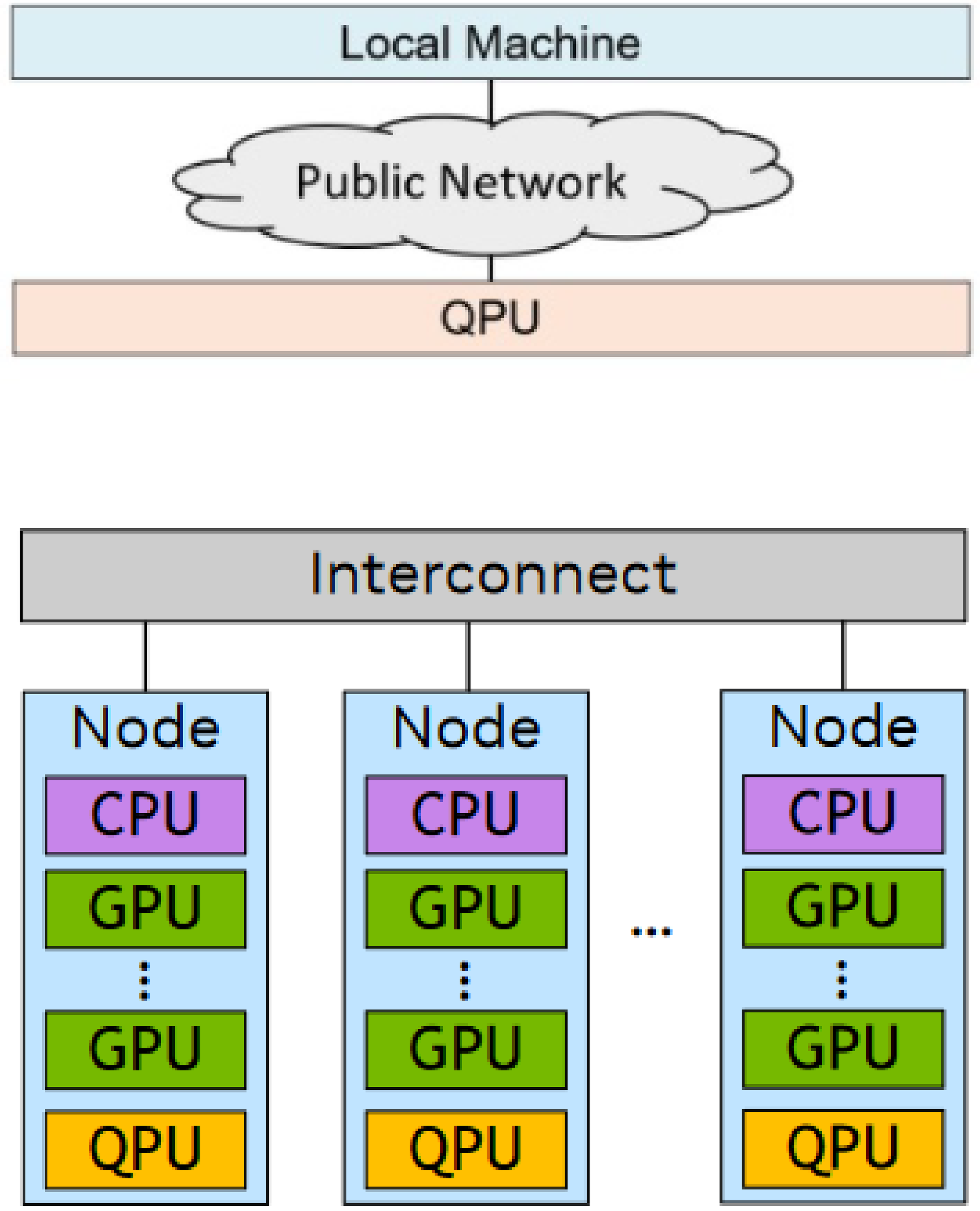


- Integrate quantum into leading accelerated applications
- Unparalleled performance and scientific productivity using the best resource for the task
- GPUs critical for QEC, calibration, hybrid algorithms

Motivation behind CUDA Quantum

Integrate quantum computers seamlessly with the modern scientific computing ecosystem

- HPC centers and many other groups worldwide are focused on the integration of quantum computers with classical computers/supercomputers
- We expect 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.

Where we need to get...

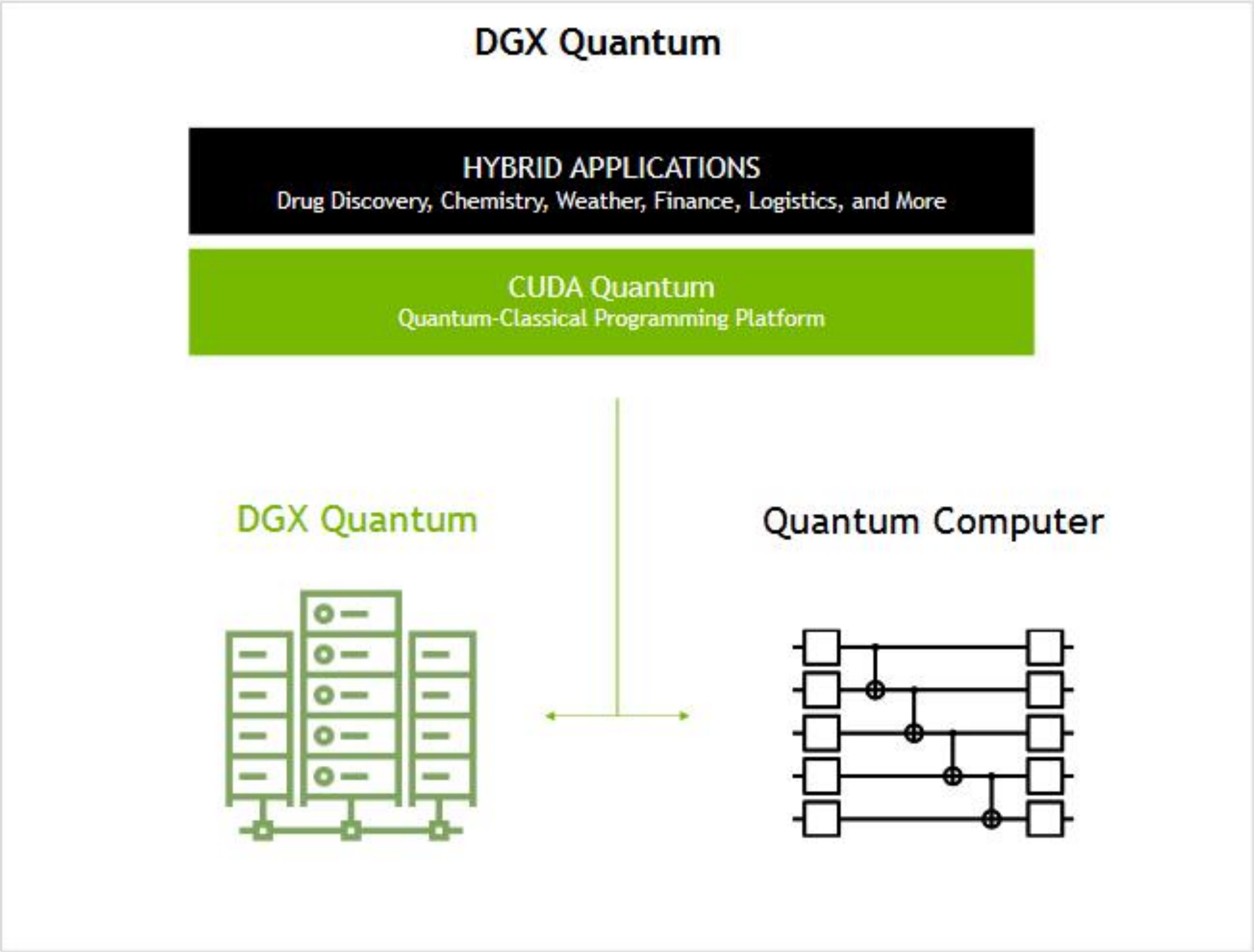
Quantum Programming with NVIDIA

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

DGX Quantum

System for integration of Quantum with GPU Supercomputing



NVIDIA.

QUANTUM MACHINES

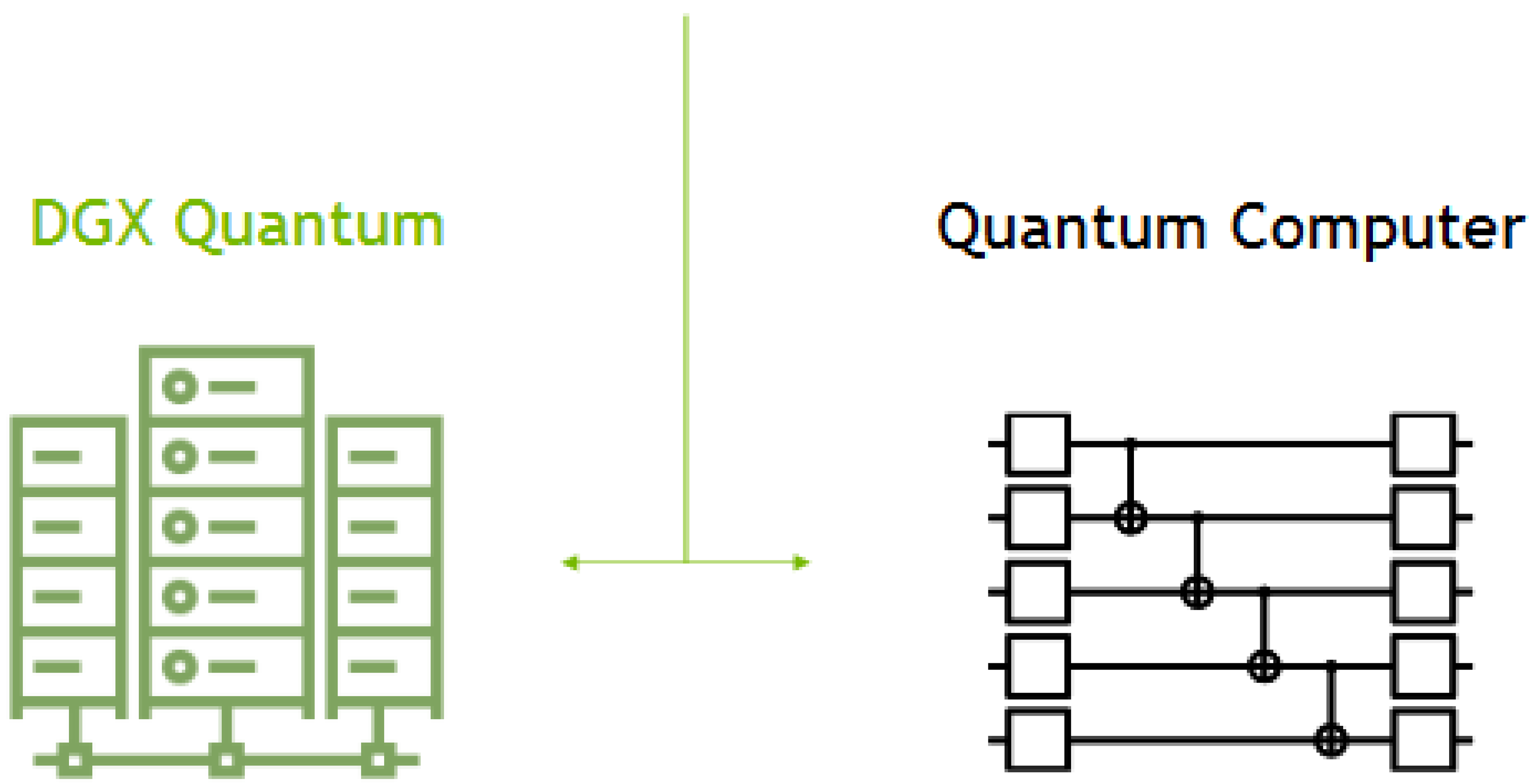
DGX Quantum

System for integration of Quantum with GPU Supercomputing

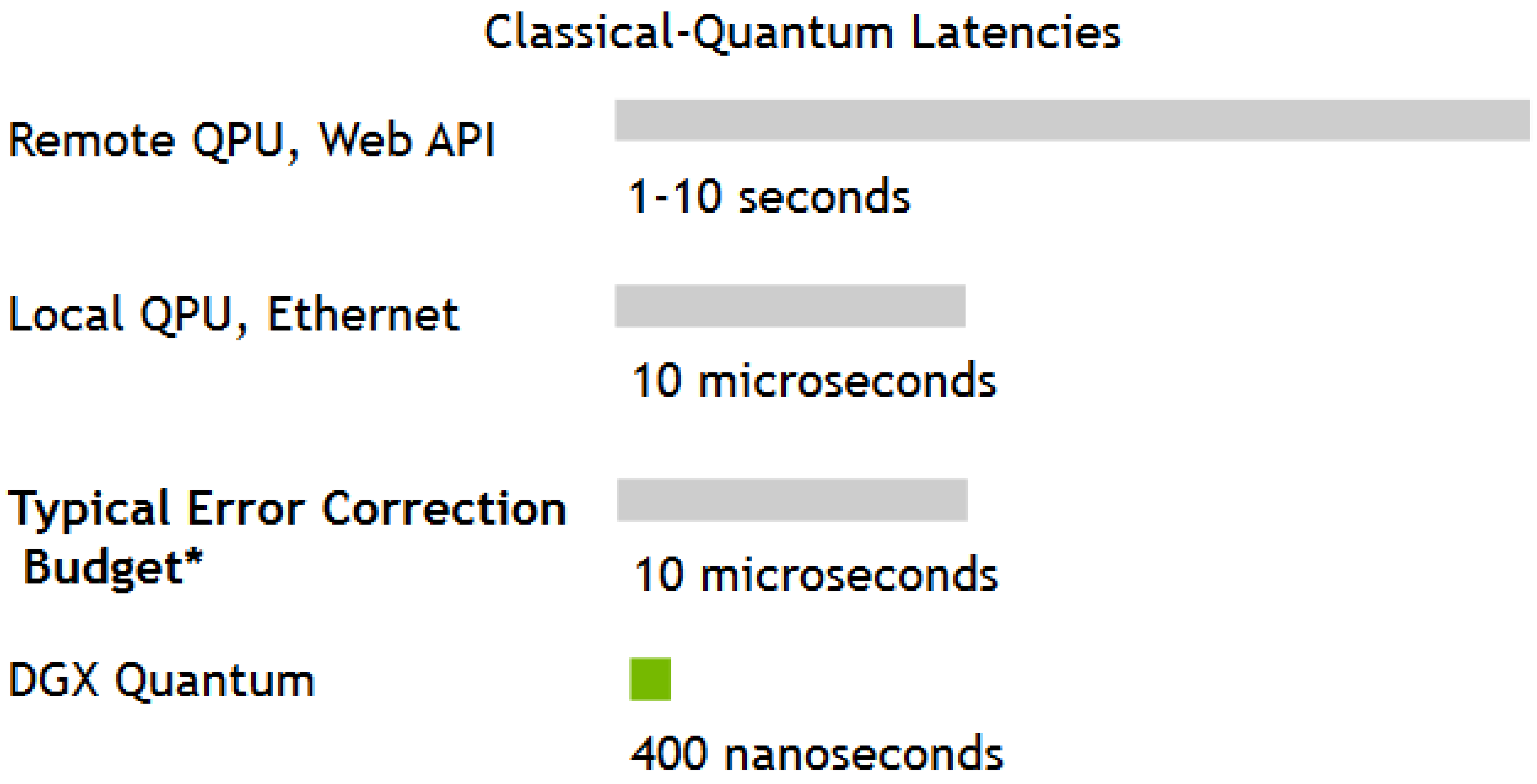
DGX Quantum

HYBRID APPLICATIONS
Drug Discovery, Chemistry, Weather, Finance, Logistics, and More

CUDA Quantum
Quantum-Classical Programming Platform



Tight Integration with Ultra Low Latency

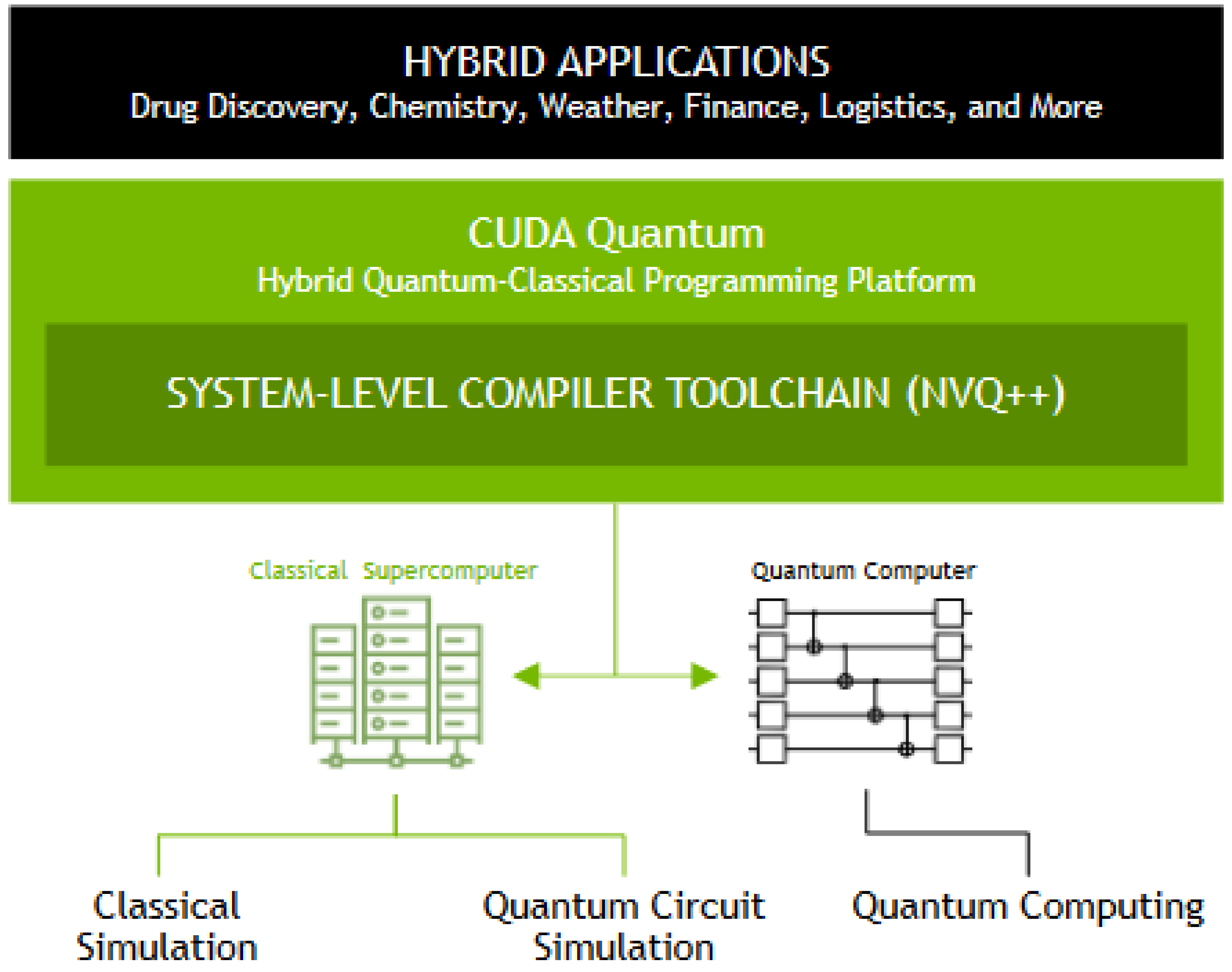


*Includes decoding time

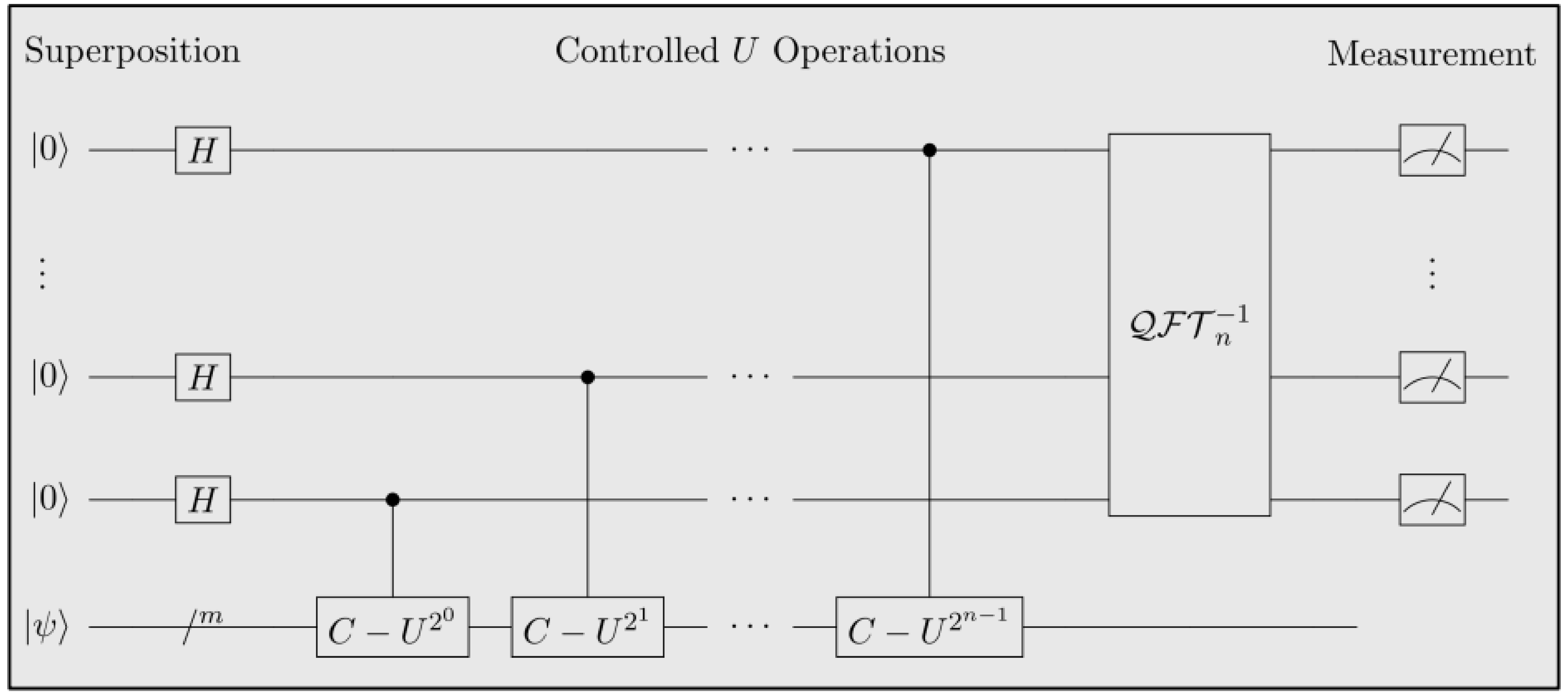
Introducing CUDA Quantum

Platform for unified quantum-classical accelerated computing

- Programming model extending C++ and Python with quantum kernels
- Open programming model, open-source compiler
  - https://github.com/NVIDIA/cuda-quantum
- QPU Agnostic – Partnering broadly including superconducting, trapped ion, neutral atom, photonic, and NV center QPUs
- Interoperable with the modern scientific computing ecosystem
- Seamless transition from simulation to physical QPU



A taste of CUDA Quantum via an example



https://en.wikipedia.org/wiki/Quantum_phase_estimation_algorithm

Example 4: Phase Estimation

- Given unitary U and an eigenvector |ψ>, compute the corresponding eigenvalue.

- Interesting things about this algorithm
  - Distinct sub-registers
  - Parallel gate application, broadcasting
  - Controlled application of a general U
  - Iterative unitary application (classical control flow)
  - Invoke quantum subroutine (IQFT)
  - Measure and post-process

Example 4: Phase Estimation

```
concept takes_qubit = cudaq::signature<void(cudaq::qubit)>;

struct qpe {
// N = top register size, M = eigen state register size
__qpu__ void operator()(int N, int M,
                        takes_qubit auto&& statePrep,
                        takes_qubit auto&& oracle) {
    ... CUDAQ kernel body ...
  }
};

int main() {
  auto statePrep = [](cudaq::qubit& q) __qpu__ {...};
  auto oracle = [](cudaq::qubit& q) __qpu__ {...};

  qpe kernel;
  auto counts = cudaq::sample(kernel, 3, 1,
                              statePrep, oracle);
  ...
}
```

- CUDAQ kernels can take classical data as input
  - Here we take quantum register sizes
- CUDAQ kernels can take other kernels as input
  - Composability of generic algorithms
  - Input CUDA Q kernels can be constrained , i.e. algorithm developers can enforce input kernel signatures.

```
concept takes_qubit = cudaq::signature<void(cudaq::qubit)>;

struct qpe {
// N = top register size, M = eigen state register size
__qpu__ void operator()(int N, int M,
                        takes_qubit auto&& statePrep,
                        takes_qubit auto&& oracle) {
    cudaq::qreg q(N+M);
    auto topRegister = q.front(N);
    auto stateRegister = q.back(M);

    h(topRegister);

    ...
  }
};
```

Example 4: Phase Estimation

- C++ classical control available for use in CUDAQ kernels
  - Here we see a nested for loop
- C++ variable declaration and arithmetic
- General controlled U application

```cpp
concept takes_qubit = cudaq::signature<void(cudaq::qubit)>;

struct qpe {
// N = top register size, M = eigen state register size
__qpu__ void operator()(int N, int M,
                        takes_qubit auto&& statePrep,
                        takes_qubit auto&& oracle) {
    ...

    for (int i = 0; i < N; ++i)
      for (int j = 0; j < (1UL << i); ++j)
        cudaq::control(oracle, topRegister[i],
                       stateRegister);


    ...
  }
};
```

Example 4: Phase Estimation

- C++ classical control available for use in CUDAQ kernels
  - Here we see a nested for loop
- C++ variable declaration and arithmetic
- General controlled U application

```
__qpu__ void iqft(cudaq::qreg<>& q) {
  auto N = q.size();
  for (int i = 0; i < N / 2; i++)
    swap(q[i], q[N-i-1]);

  for (auto i : cudaq::range(N-1)) {
    h(q[i]);
    int j = i + 1;
    for (int y = i; y >= 0; --y) {
      const double theta = -M_PI / std::pow(2., j-y);
      r1<cudaq::ctrl>(theta, q[j], q[y]);
    }
  }

  h(q.back());
}

struct qpe {
__qpu__ void operator()(...) {
    ...
    iqft(topRegister);
    ...
  }
};
```

Example 4: Phase Estimation

- Can invoke in-scope CUDAQ kernels
- This kernel takes quantum memory as input, it is therefore a pure-device kernel. The typed requirement can be relaxed in this case.

Example 4: Phase Estimation

```cpp
struct qpe {
void operator()(const int nCountingQubits, const int nStateQubits,
                cudaq::takes_qubit auto &&state_prep,
                cudaq::takes_qubit auto &&oracle) __qpu__ {

    // Allocate a register of qubits
    cudaq::qreg q(nCountingQubits + nStateQubits);

    // Extract sub-registers, one for the counting qubits
    // another for the eigen state register
    auto counting_qubits = q.front(nCountingQubits);
    auto state_register = q.back(nStateQubits);

    // Prepare the eigenstate
    state_prep(state_register);

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

    // Perform ctrl-U^j
    for (int i = 0; i < nCountingQubits; ++i)
      for (int j = 0; j < (1UL << i); ++j)
        cudaq::control(oracle, {counting_qubits[i]}, state_register);

    // Apply inverse quantum fourier transform
    iqft(counting_qubits);

    // Measure to gather sampling statistics
    mz(counting_qubits);

    return;
  }
};

int main() {
  qpe kernel;
  auto counts = cudaq::sample(kernel, 3, 1,
                [](cudaq::qubit& q) __qpu__ { x(q); },
                [](cudaq::qubit& q) __qpu__ { t(q); });
  counts.dump();
  return 0;
}
```

```cpp
__qpu__ void iqft(cudaq::qspan<> q) {
  auto N = q.size();
  for (int i = 0; i < N / 2; i++)
    swap(q[i], q[N-i-1]);

  for (auto i : cudaq::range(N-1)) {
    h(q[i]);
    int j = i + 1;
    for (int y = i; y >= 0; --y) {
      const double theta = -M_PI / std::pow(2., j-y);
      r1<cudaq::ctrl>(theta, q[j], q[y]);
    }
  }

  h(q.back());
}
```

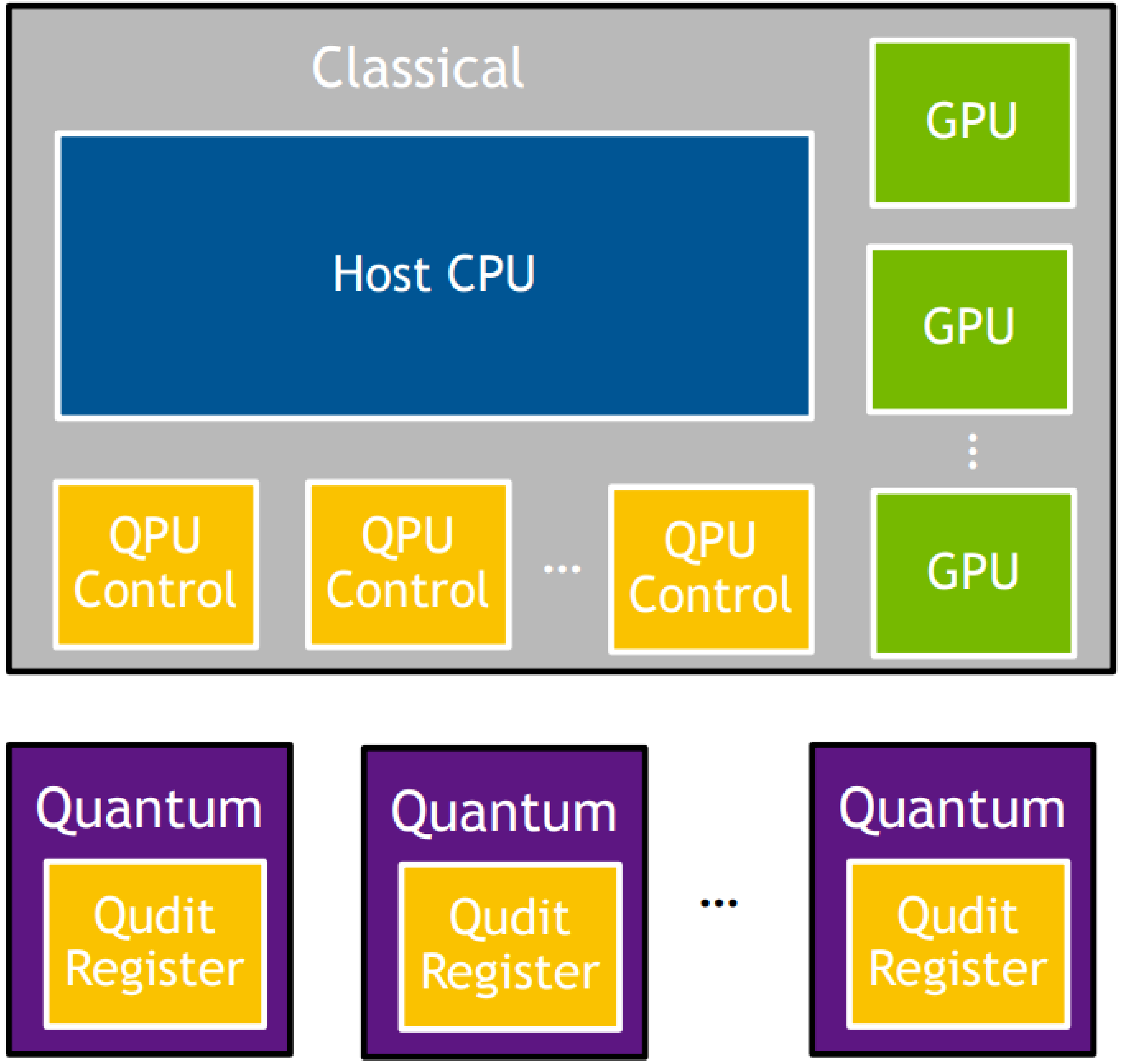
```
$ nvq++ qpe.cpp -qpu cuquantum
$ ./a.out
```

CUDA Quantum Language Details

CUDA Quantum System Architecture Model

Devices available and discrete memory spaces

- The system architecture assumes multiple discrete memory spaces that are either classical or quantum
- Classical
  - Host CPU, Device GPU(s), QPU(s) Control
- Quantum
  - A general qudit register
- Implications
  - Some classical compute available for hybrid quantum-classical code (arithmetic operations, conditional statements on qubit measurements, etc.)
  - GPU compute available for pre- / post-processing, hybrid application workflows
  - QISA on general qudits (e.g. qubits or bosonic modes)
    - **NOTE CUDAQ development focus thus far has been on qudit**
  - Agnostic to the control system architecture (just assume there is some control ISA and our compiler lowers C++ to that representation)



CUDA Quantum Kernels

Entry point into quantum co-processing

- Follow the CUDA kernel function approach
  - Nicely separates host and quantum device code
- Typed callables in C++
  - Free functions are not first-class citizens in C++
  - Structs / Classes with `R operator()(Args...)` overloaded
  - Lambdas (implicitly typed)
  - Enable generic libraries of quantum-classical code
- Annotated to indicate compilation and execution on quantum coprocessor - `__qpu__`

```
struct myEntryPointKernel1 {
  int operator()(int i, int j, float f) __qpu__ {
    ...
  }
};

struct myEntryPointKernel2 {
  void operator()(std::vector<double> x) __qpu__ {
    ...
  }
};

auto pureDeviceLambda = [](cudaq::qubit& q, int i) __qpu__  {
  ...
};

__qpu__ void freeFunctionDeviceKernel(cudaq::qspan<> q) {...}

auto entryPointLambda = [&](double theta, double phi) __qpu__
{
  ... allocate quantum memory q ...
  pureDeviceLambda(q[0], 3);
  freeFunctionDeviceKernel(q);
  ...
};
```

CUDA Quantum Kernels

Entry point into quantum co-processing

- Supported argument / result types:
  - `T` such that `std::is_arithmetic<T> == true`
  - Arithmetic `T`, for `std::vector<T>`
  - Will add more in the future
  - Assume value-semantics for input classical args

- Types of kernels:
  - Entry Point CUDAQ Kernels
    - Called from host code, cannot take quantum input
    - Initiate quantum allocation, deallocation occurs at scope exit
  - Pure Device CUDAQ Kernels
    - Cannot be called from host code, can take quantum input
    - Typed requirement relaxed - can be expressed as free function

```
struct myEntryPointKernel1 {
  int operator()(int i, int j, float f) __qpu__ {
    ...
  }
};


struct myEntryPointKernel2 {
  void operator()(std::vector<double> x) __qpu__ {
    ...
  }
};


auto pureDeviceLambda = [](cudaq::qubit& q, int i) __qpu__ {
  ...
};


__qpu__ void freeFunctionDeviceKernel(cudaq::qspan<> q) {...}


auto entryPointLambda = [&](double theta, double phi) __qpu__
{
  ... allocate quantum memory q ...
  pureDeviceLambda(q[0], 3);
  freeFunctionDeviceKernel(q);
  ...
};
```

CUDA Quantum Memory Types

Allocating quantum memory and associated memory management

- Quantum memory cannot be copied, and we need to be careful about who owns the qudits. Qudits can only be allocated within CUDAQ kernel code (no host-code allocation).
- Starts with the `cudaq::qudit<Levels>`
  - Cannot be copied or moved, must be passed by reference
  - Simply exposes a unique identifier (e.g. qubit 0, qubit 1, etc.)
  - Qubit ids tracked as allocation / deallocation occurs

```
auto l = [](cudaq::qubit& q) {...}; // Good
auto l = [](cudaq::qubit q) { ... }; // Bad
```

- What about Quantum Containers – can we follow patterns from C++?
  - What is a `std::vector<qudit>` or `std::array<qudit>`?
  - What about non-owning views of qudits?

```
... CUDAQ Runtime Code ...
namespace cudaq {
  template<std::size_t Levels>
  class qudit {
    public:
      qudit();
      qudit(const qudit&) = delete;
      qudit(qudit &&) = delete;
      std::size_t id() const;
    };
   using qubit = qudit<2>;
}
... User Code ...
{
  cudaq::qubit q, r;
  assert(q.id() == 0);
  assert(r.id() == 1);
  // scope exit, deallocation
}


cudaq::qubit q;
assert(q.id() == 0); // != 2
```

CUDA Quantum Memory Types

Allocating quantum memory and associated memory management

- `cudaq::qreg` – owning container for qudits.
  - Models either a dynamically allocated container (i.e., vector) or a compile-time-known container (i.e., array).
- `cudaq::qspan` – non-owning container for qudits
  - Can be passed by value / copied.
  - Useful for extracting sub-registers in algorithms

```
// Allocate some qubits
cudaq::qreg<5> q;
cudaq::qreg runtimeDynamicQ(N);


// Get sub-views, returned as a span
auto nonOwningSpan = q.front(3);
```

```
... CUDAQ Runtime Code ...
// Owning, dynamic or compile-time
template <std::size_t N = dyn, std::size_t Levels = 2>
class qreg {
  public:
    using value_type = qudit<Levels>;
  private:
    std::conditional_t<N==dyn, std::vector<value_type>,
    std::array<value_type, N>> qudits;
  public:
    value_type& operator[](const std::size_t idx);
    qspan<dyn, Levels> front(std::size_t count);
    ... Other extraction methods ...
};


// Non-owning, dynamic or compile-time
template <std::size_t N = dyn, std::size_t Levels = 2>
class qspan {
  private:
    std::span<qudit<Levels>, N> quditView;
  public:
    ... Same extraction API as above ...
};
```

Quantum Intrinsic Operations for Qubits

Logical quantum instruction set

- CUDAQ exposes a default gate set for 2-level qudits (qubits).
  - Pauli / Clifford+T
    - x, y, z, h, t, s, swap
  - Arbitrary Rotations
    - r1, rx, ry, rz, phased_rx
  - Universal
    - u2, u3
  - Measurement (mx, my, mz)
- We expose operations that target a single qubit (except swap), and (multi-) control operations achieved via modifiers:
  - `cudaq::ctrl` – prepend qubit argument list with any number of control qubits
  - `cudaq::adj` – reverse / adjoint of instruction
- Kernel-level control and adjoint
  - `cudaq::control(Kernel, ctrlQs..., Args...)`
  - `cudaq::adjoint(Kernel, Args...)`

```
... CUDAQ Runtime Code ...
namespace cudaq {
  // Define modifier types
  struct base; struct ctrl; struct adj;
  // Example One-Qubit Operation
  template<typename mod = base, typename... QubitArgs>
  void x(QubitArgs&... args) { ... }


   template <typename QuantumKernel, typename ControlRegister,
             typename... Args>
    requires(std::ranges::range<ControlRegister>)
  void control(QuantumKernel &&kernel,
            ControlRegister&& controls, Args& ... Args) { ... }
  }
... CUDAQ User Code ...
auto cnot = [](cudaq::qubit& q, cudaq::qubit& r) __qpu__ {
  x<cudaq::ctrl>(q, r); // Can control any one-qubit gate
};


auto toffoli = [&]() __qpu__ {
  cudaq::qreg<3> qr; // Allocate compile-time known register
  x(qr[0], qr[2]); // Initialize the qubits to 101
  cudaq::control(cnot, {qr[0]}, qr[1], qr[2]);
  mz(qr); // measure the whole register
  // mz(q[0], q[1], q[2], ...); // mz can be on multiple values
};
```

CUDA Quantum Spin Operator

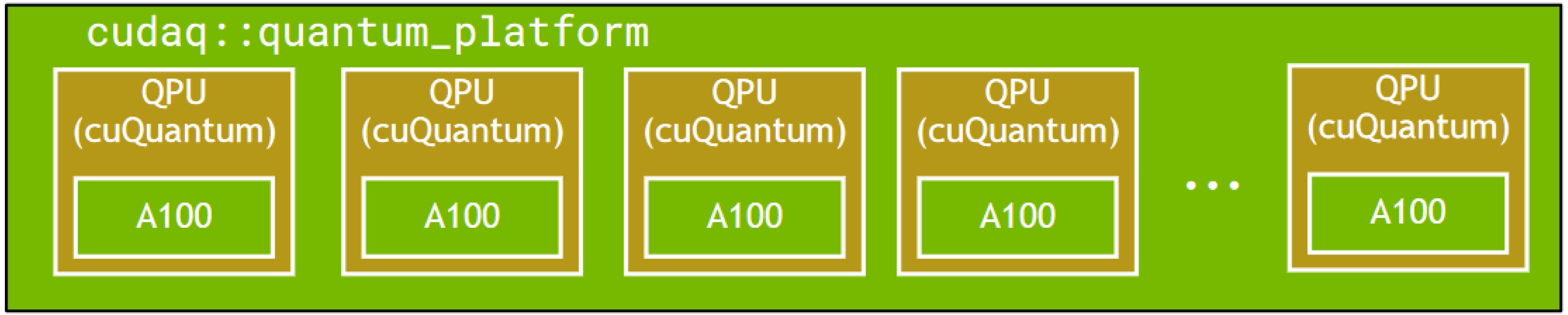
Logical quantum instruction set

- CUDAQ defines a built-in type for describing spin operators
  - `cudaq::spin_op` models sums of Pauli terms in the binary
    symplectic formalism – ([x0 x1 ... xn][z0 z1 ... zn])
  - C++ algebraic operator overloads implemented
  - Convenient API for x, y, z Pauli operators.
- This is used for variational algorithms as well as circuit synthesis.

```
using namespace cudaq::spin;


// Define spin operators with algebraic
// operator overloads


cudaq::spin_op h = 5.907 - 2.1433 * x(0) * x(1) -
                   2.1433 * y(0) * y(1) + .21829 * z(0) -
                   6.125 * z(1);


// Can also use spin operators for circuit synthesis
// Here a first order trotterization exp() operation
struct ansatz {
  void operator()(double theta) __qpu__ {
    cudaq::qreg q(2);
    x(q[0]);
    cudaq::trotter(q, theta, x(0) * y(1) - y(0) * x(1));
  }
};
```

CUDA Quantum Platform and Asynchronous Execution

Expose the underlying system architecture to the programmer



- The system architecture model considers access to multiple quantum accelerators
- CUDAQ provides programmatic access to this configuration via the quantum_platform
- CUDAQ and cuQuantum expose a native platform that models a virtual QPU for every CUDA device.
- Each CUDA device gets a cuQuantum based simulator
- Enable experimentation with distributed quantum computing

```
// 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 = std::reduce(std::execution::par,
cudaq::when_all(subs), 0.0);
```

CUDA Quantum Generic Algorithm Primitives

cudaq namespace functions that are generic on the CUDAQ kernel expressions

- CUDAQ defines a generic function for sampling
  - Provide a CUDAQ kernel and its runtime arguments
  - Return a map of observed bit strings to number of times observed.
- Can perform synchronously or asynchronously
  - if async, can target specific QPU device ID if on a multi-QPU platform
- CUDAQ kernels must return void and specify measurements

```
template <typename QuantumKernel, typename... Args>
sample_result sample(QuantumKernel &&kernel, Args &&...args);
```

```
template <typename QuantumKernel, typename... Args>
async_sample_result sample_async(std::size_t qpu_id,
                                 QuantumKernel &&kernel, Args &&...args);
```

```
#include <cudaq.h>


int main(int argc, char** argv) {
  // Define the CUDAQ Kernel
  auto ghz = [](std::size_t N) __qpu__ {
    cudaq::qreg qr(N);
    h(qr[0]);
    for (auto i : cudaq::range(N-1)) {
      x<cudaq::ctrl>(qr[i], qr[i+1]);
    }
    mz(qr);
  };


  // Synchronously sample the state
  // generated by the kernel
  auto counts = cudaq::sample(ghz, 30);
  counts.dump();


  // Asynchronously sample
  auto future = cudaq::sample_async(ghz, 30);
  // .. Go do other work ..
  counts = future.get();
  counts.dump();
  return 0;
}
```

CUDA Quantum Generic Algorithm Primitives

cudaq namespace functions that are generic on the CUDAQ kernel expressions

- CUDAQ defines a generic function for computing expectation values of spin operators with respect to a parameterized kernel.
  - `<H> = <ψ(Θ) | H | ψ(Θ)>`
  - `<Kernel(Args...) | H | Kernel(Args...)>`
- Takes as input the kernel, the `cudaq::spin_op`, and the concrete runtime parameters for the kernel.
- Returns the expected value as `double`.
- Serves as foundation for many variational algorithms.

```
template <typename QuantumKernel, typename... Args>
observe_result observe(QuantumKernel &&kernel,
                       spin_op& h, Args &&...args);
```

```
template <typename QuantumKernel, typename... Args>
async_observe_result observe_async(
                 QuantumKernel &&kernel, spin_op& h,
                 Args &&...args);
```

```
#include <cudaq.h>
using namespace cudaq::spin;


int main(int argc, char** argv) {
  // Define the ansatz as a CUDAQ lambda
  auto ansatz = [](double theta) __qpu__ {
    cudaq::qreg q(2);
    x(q[0]);
    ry(theta, q[1]);
    x<cudaq::ctrl>(q[1], q[0]);
  };


  // Problem Hamiltonian
  cudaq::spin_op h = 5.907 - 2.1433 * x(0) * x(1) -
                 2.1433 * y(0) * y(1) + .21829 * z(0) -
                 6.125 * z(1);
  for (auto& param : cudaq::linspace(-M_PI,M_PI,20)) {
    double energyAtParam =
                   cudaq::observe(ansatz, h, param);
    printf("<H>(%lf) = %lf\n", param, energyAtParam);
  }


  auto future = cudaq::observe_async(ansatz, h, 0.59);
  double energy = future.get();
  return 0;
}
```

Compiler Implementation

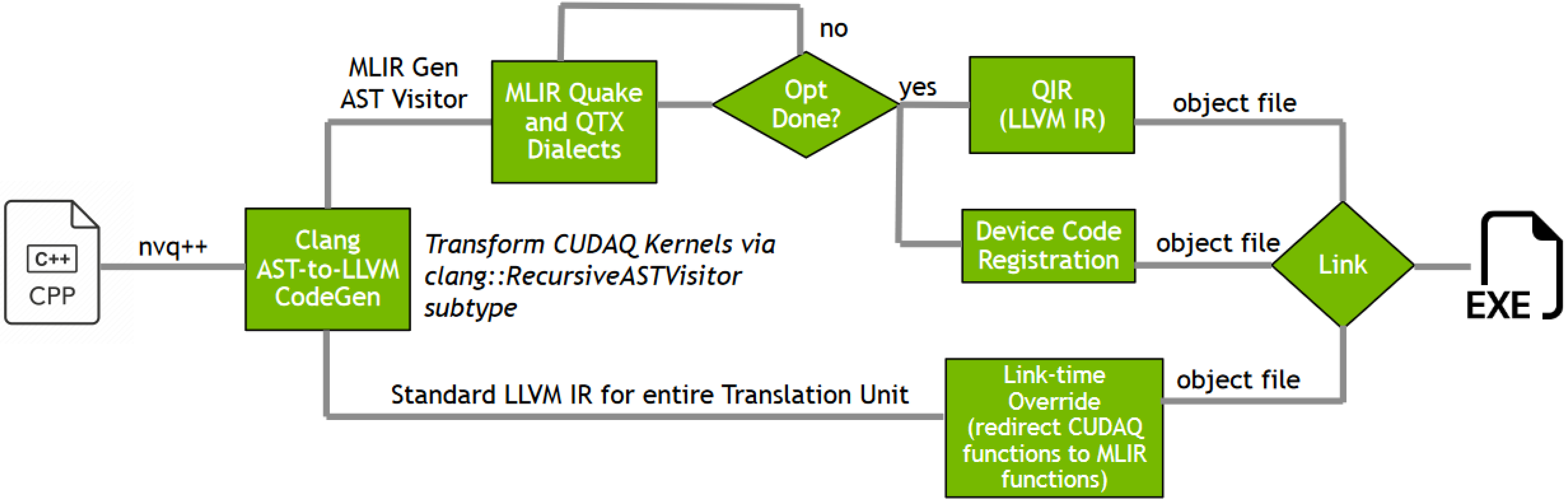
NVQ++

CUDA Quantum Compilation Model

CUDAQ is implemented via the NVQ++ compiler and follows a split compilation model

- Enable ahead-of-time and just-in-time compilation of quantum-classical programs
- Replace CUDAQ kernels with MLIR-generated function symbols

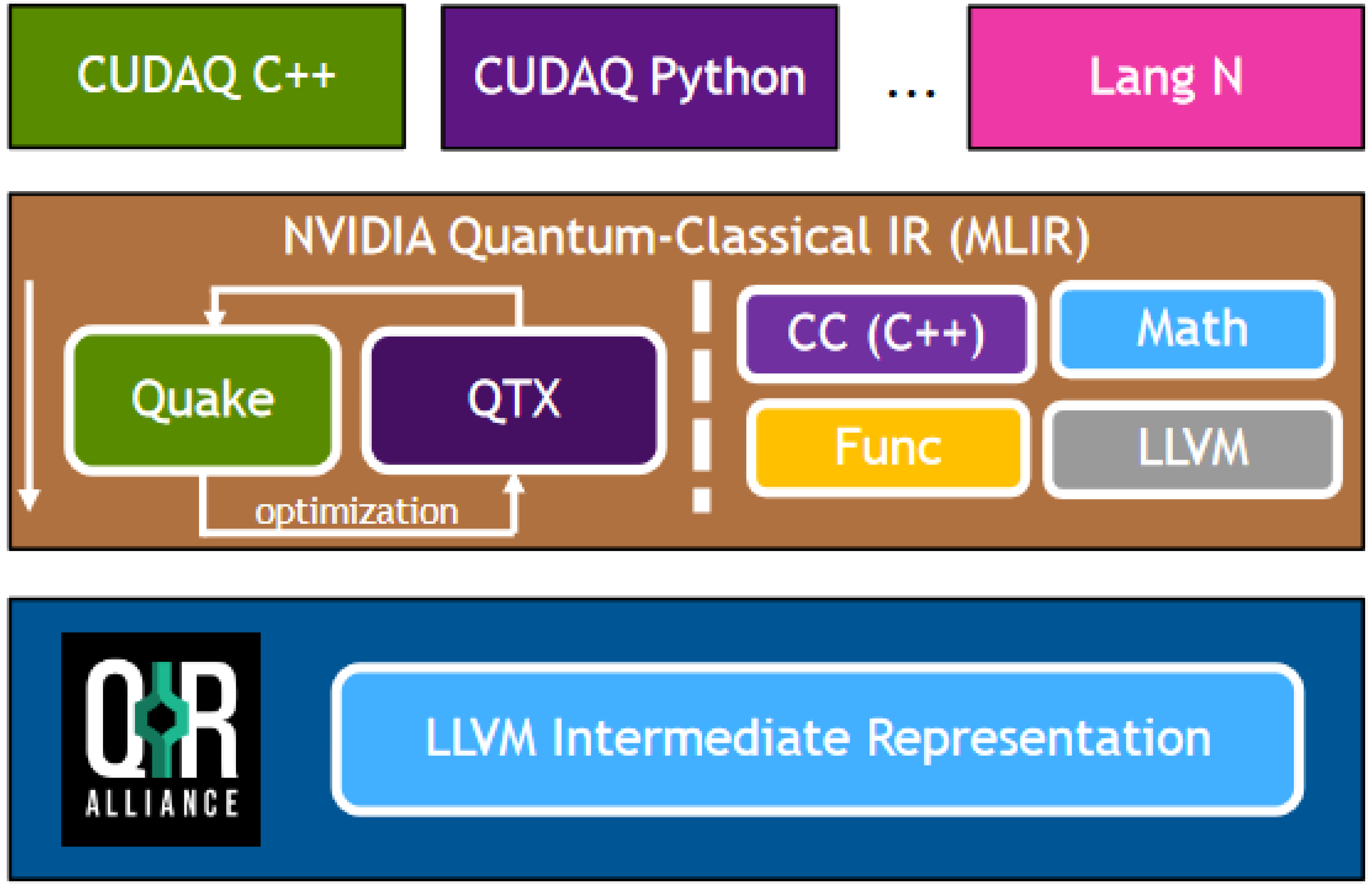
- Entire source file is lowered to LLVM
- MLIR lowered to an IR that can be taken to object code
- QIR is the default target
- MLIR representations registered with the runtime
- Classical LLVM entry-points are overridden at link-time.



CUDA Quantum MLIR Dialects

The NVQ++ IR is composed of Quantum and Classical MLIR Dialects

- We have defined 3 MLIR dialects
  - Quake – language-level, memory semantic model
  - QTX – circuit-level, value semantic model
  - CC – represent pertinent C++ abstractions (stdvec, etc.)
  - Optimizations can benefit from either model
  - Incorporate existing classical dialects from MLIR
  - Hybrid optimization – retain classical optimizations in tandem with the quantum ones
- Lower to executable code via the LLVM IR.
  - Or, can lower to lower MLIR dialects provided by partners.



More CUDA Quantum Examples

Example 1

```cpp
// Compile and run with:
// nvq++ static_kernel.cpp -o ghz.x && ./ghz.x

#include <cudaq.h>

// Define a CUDA Quantum kernel that is fully specified
// at compile time via templates.
template <std::size_t N>
struct ghz {
  auto operator()() __qpu__ {

    // Compile-time, std::array-like qreg.
    cudaq::qreg<N> q;
    h(q[0]);
    for (int i = 0; i < N - 1; i++) {
      x<cudaq::ctrl>(q[i], q[i + 1]);
    }
    mz(q);
  }
};

int main() {

  auto kernel = ghz<10>{};
  auto counts = cudaq::sample(kernel);
  counts.dump();

  // Fine grain access to the bits and counts
  for (auto &[bits, count] : counts) {
    printf("Observed: %s, %lu\n", bits.data(), count);
  }

  return 0;
}
```

Example 1: GHZ State

- CUDA Quantum kernels are any typed callable in the language that is annotated with the __qpu__ attribute.
- Here we see that we can define a custom struct that is templated on a size_t parameter.
- Within the kernel, we are free to apply various quantum operations.
- Controlled operations are **modifications** of single-qubit operations, here we have a controlled-X.
- We leverage the generic cudaq::sample function, which returns a data type encoding the qubit measurement strings and the corresponding number of times that string was observed.

```
nvq++ static_kernel.cpp -o ghz.x
./ghz.x
```

Example 2

```cpp
// Compile and run with:
// nvq++ expectation_values.cpp -o d2.x && ./d2.x

#include <cudaq.h>
#include <cudaq/algorithm.h>

// The example here shows a simple use case for the cudaq::observe()
// function in computing expected values of provided spin_ops.

struct ansatz {
  auto operator()(double theta) __qpu__ {
    cudaq::qreg q(2);
    x(q[0]);
    ry(theta, q[1]);
    x<cudaq::ctrl>(q[1], q[0]);
  }
};

int main() {

  // Build up your spin op algebraically
  using namespace cudaq::spin;
  cudaq::spin_op h = 5.907 - 2.1433 * x(0) * x(1) - 2.1433 * y(0) * y(1) +
                     .21829 * z(0) - 6.125 * z(1);

  // Observe takes the kernel, the spin_op, and the concrete params for the
  // kernel
  double energy = cudaq::observe(ansatz{}, h, .59);
  printf("Energy is %lf\n", energy);
  return 0;
}
```

Example 2: Expectation Values

- Here we define a parameterized CUDA Quantum kernel, a callable type named ansatz that takes as input a single angle theta.
- In host code, we define a Hamiltonian operator we are interested in via the CUDA Quantun spin_op type.
- Quantum provides a generic function cudaq::observe which takes a parameterized kernel.
- The return type of this function is an cudaq::observe_result which contains all the data from the execution, but is trivially convertible to a double, resulting in the expectation value we are interested in.

```
nvq++ expectation_values.cpp -o exp_vals.x
./exp_vals.x
```

Example 3

```cpp
// Compile and run with:
// nvq++ multi_controlled_operations.cpp -o ccnot.x && ./ccnot.x

#include <cudaq.h>
#include <cudaq/algorithm.h>

// Here we demonstrate how one might apply multi-controlled
// operations on a general CUDA Quantum kernel.
struct ApplyX {
  void operator()(cudaq::qubit &q) __qpu__ { x(q); }
};

struct ccnot_test {
  // constrain the signature of the incoming kernel
  void operator()(cudaq::takes_qubit auto &&apply_x) __qpu__ {
    cudaq::qreg qs(3);

    x(qs);
    x(qs[1]);

    // Control U (apply_x) on the first two qubits of
    // the allocated register.
    cudaq::control(apply_x, qs.front(2), qs[2]);

    mz(qs);
  }
};
```

Example 3: Multi-control Synthesis

- For this scenario, our general unitary can be described by another pre-defined CUDA Quantum kernel expression.
- In this example, we show 2 distinct ways for generating a Toffoli operation.
- The first way to generate a Toffoli operation starts with a kernel that takes another kernel as input.

```cpp
int main() {
  // We can achieve the same thing as above via
  // a lambda expression.
  auto ccnot = []() __qpu__ {
    cudaq::qreg q(3);

    x(q);
    x(q[1]);

    x<cudaq::ctrl>(q[0], q[1], q[2]);

    mz(q);
  };

  auto counts = cudaq::sample(ccnot);

  // Fine grain access to the bits and counts
  for (auto &[bits, count] : counts) {
    printf("Observed: %s, %lu\n", bits.data(), count);
  }

  auto counts2 = cudaq::sample(ccnot_test{}, ApplyX{});

  // Fine grain access to the bits and counts
  for (auto &[bits, count] : counts2) {
    printf("Observed: %s, %lu\n", bits.data(), count);
  }
}
```

Example 3: Multi-control Synthesis

- The second one in host code is the definition of a CUDA Quantum lambda that synthesizes a Toffoli via the general multi-control functionality for any single-qubit quantum operation x<cudaq::ctrl>(q[0], q[1], q[2]).

```
nvq++ multi_controlled_operations.cpp -o mcx.x
./mcx.x
```

Example 4

```cpp
#include <cudaq.h>

// Define a quantum kernel with a runtime parameter
struct ghz {
  auto operator()(const int N) __qpu__ {

    // Dynamic, vector-like qreg
    cudaq::qreg q(N);
    h(q[0]);
    for (int i = 0; i < N - 1; i++) {
      x<cudaq::ctrl>(q[i], q[i + 1]);
    }
    mz(q);
  }
};

int main() {
  auto counts = cudaq::sample(ghz{}, 30);
  counts.dump();

  // Fine grain access to the bits and counts
  for (auto &[bits, count] : counts) {
    printf("Observed: %s, %lu\n", bits.data(), count);
  }

  return 0;
}
```

Example 4: Simulation with cuQuantum

- CUDA Quantum provides native support for cuQuantum-accelerated state vector and tensor network simulations.
- Here we generate a GHZ state on 30 qubits.

```
nvq++ --qpu cuquantum cuquantum_backends.cpp -o ghz.x
./ghz.x
```

Exposing CUDA Quantum to Python via Quake JIT

CUDA Quantum Python Bindings

Core C++ compiler platform exposed via builder API
JIT compile Quake representation at runtime

```
// Define the CUDAQ Kernel
auto bell = []() __qpu__ {
  cudaq::qreg qr(2);
  h(qr[0]);
  x<cudaq::ctrl>(qr[0], qr[1]);
  mz(qr);
};
```

```
// Define the CUDAQ Kernel
auto bell = cudaq::make_kernel();
auto qr = bell.qalloc(2);
bell.h(qr[0]);
bell.x<cudaq::ctrl>(qr[0], qr[1]);
bell.mz(qr);
std::cout << bell << "\n";

auto counts = cudaq::sample(bell);
counts.dump()
...
```

- One can program CUDAQ kernel expressions, or build them up dynamically at runtime.
- The runtime approach builds a Quake representation internally and the first invocation JIT compiles the code

```
module {
  func.func @__nvqpp__mlirgen____nvqppBuilderKernel_367535629127() {
    %c2_i64 = arith.constant 2 : i64
    %c1_i32 = arith.constant 1 : i32
    %c0_i32 = arith.constant 0 : i32
    %0 = quake.alloca : !quake.qvec<2>
    %1 = quake.qextract %0[%c0_i32] : !quake.qvec<2>[i32] -> !quake.qref
    quake.h (%1)
    %2 = quake.qextract %0[%c1_i32] : !quake.qvec<2>[i32] -> !quake.qref
    quake.x [%1 : !quake.qref] (%2)
    %3 = llvm.alloca %c2_i64 x i1 : (i64) -> !llvm.ptr<i1>
    %4 = quake.mz(%1 : !quake.qref) : i1
    llvm.store %4, %3 : !llvm.ptr<i1>
    %5 = quake.mz(%2 : !quake.qref) : i1
    %6 = llvm.getelementptr %3[1] : (!llvm.ptr<i1>) -> !llvm.ptr<i1>
    llvm.store %5, %6 : !llvm.ptr<i1>
    return
  }
}
```

CUDA Quantum Python Bindings

Core C++ compiler platform exposed via builder API
JIT compile Quake representation at runtime

- The C++ Quake builder API is what CUDAQ exposes to Python

```python
import cudaq

# Set the Simulator to cuQuantum
cudaq.set_qpu('cuquantum')

# Create a Bell State Kernel
bell = cudaq.make_kernel()
qr = bell.qalloc(2)
bell.h(qr[0])
bell.cx(qr[0], qr[1]);
bell.mz(qr)

# Print the Quake Code
print(bell)

# JIT Compile and Execute
counts = cudaq.sample(bell)
print(counts)
```

Sample

```python
import cudaq

# Set the backend
cudaq.set_qpu('quantinuum')

# Create the kernel function signature
# here void(float)
ansatz, theta = cudaq.make_kernel(float)
q = ansatz.qalloc(2)
ansatz.x(q[0])
ansatz.ry(theta, q[1]);
ansatz.cx(q[0], q[1]);

h = cudaq.SpinOperator(...)

# API mirrors the C++
result = cudaq.observe(ansatz, h, .59)
print('<H> = ', result.expectation_z())
```

Observe

```python
import cudaq

cudaq.set_qpu('dm') # density matrix

# Create a depolarization channel on
# X operations on qubit 0
depol = cudaq.DepolarizationChannel(.1)
noise = cudaq.NoiseModel()
noise.add_channel('x', [0], depol)

# Create your kernel code
bell = cudaq.make_kernel()
...

# Sample in the presence of noise
noisyCounts = cudaq.sample(bell,
                   noise_model=noise)

print(noisyCounts)
```

Noise Modeling

CUDA Quantum in Python

Lab 1: Basics

Lab 2: Hybrid QNN

NVIDIA