Quantum Computing at NVIDIA and Integrating HPC with QC

Justin Gage Lietz, NVIDIA HPC + Quantum Computing Group

Nuclear TALENT Seminar, June 17, 2025

NVIDIA Quantum

NVIDIA Quantum Cloud

CUDA-QX

QEC

Solvers

CUDA-Q

Compiler

Programming Model

Tools

Infrastructure

Quantum

Classical

cuQuantum

Libraries for simulating applications and hardware

QPU

CUDA-Q supports range of QPU vendor backends

OQC

QUANTINUUM

ORCA Computing

ALICE & BOB

QUANTUM BRILLIANCE

PASQAL

QuEra Computing Inc.

seeqc

rigetti

DGX Quantum

Reference architecture for low-latency QPU/DGX integration

DGX SuperPOD

NVIDIA

NVIDIA Quantum

NVIDIA Quantum Cloud

CUDA-QX

QEC

Solvers

CUDA-Q

Compiler

Programming Model

Tools

Infrastructure

Quantum

Classical

cuQuantum

Libraries for simulating applications and hardware

QPU

CUDA-Q supports range of QPU vendor backends

OQC

QUANTINUUM

ORCA Computing

ALICE & BOB

QUANTUM BRILLIANCE

PASQAL

QuEra Computing Inc.

seeqc

rigetti

DGX Quantum

Reference architecture for low-latency QPU/DGX integration

DGX SuperPOD

Jobs in Quantum Tech at NVIDIA

Highly interdisciplinary workplace

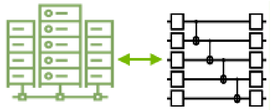
- My role: Quantum Software Architect
  - Integrating Quantum Computers with Supercomputers (HPC)
  - Focus on quantum error correction (QEC)
  - Research into GPU decoders
- Algorithm Engineer
- Software Engineer
  - Compilers
  - Simulators
  - Libraries
  - GPU acceleration
- Research Scientist
- Applied Research Scientist

Many backgrounds from across STEM fields: EE, CS, Math, Chemistry, Physics!

- My background: PhD in Computational Physics with Morten at Michigan State University (2019)
- I attended a TALENT school at the ECT* in 2014 for density functional theory!
- Oak Ridge National Laboratory Postdoc (2019-2021) Staff (2021-2023)

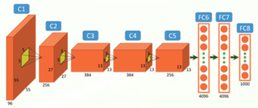
NVIDIA and Accelerated Computing

Quantum Computing



Generative AI

Deep Learning



Scientific Computing

Early NVIDIA investment has historically lead to drastic improvements in computational scalability.

Can we take what we've built thus far and enable a future quantum-accelerated platform?

Quantum-Accelerated Supercomputing

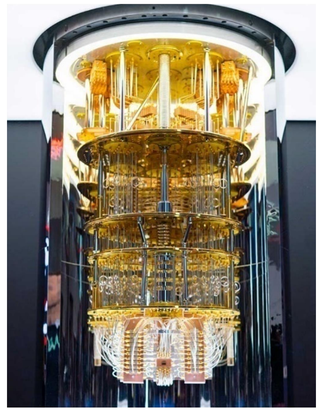
GPU Supercomputers are the foundation of Quantum R&D

Simulation

- Quantum computers are small and error-prone
  - simulation is an essential tool
- **Today:** Powerful simulators enable algorithm and application R&D – new approaches:
  - tensor networks, state vector, density matrix, Clifford simulators
- **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

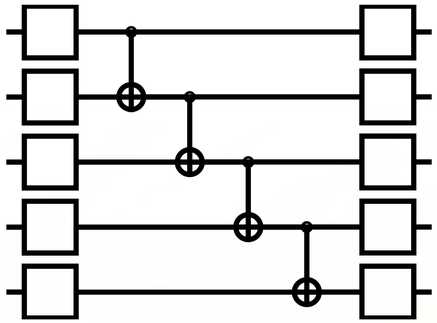


AI for Quantum

- 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

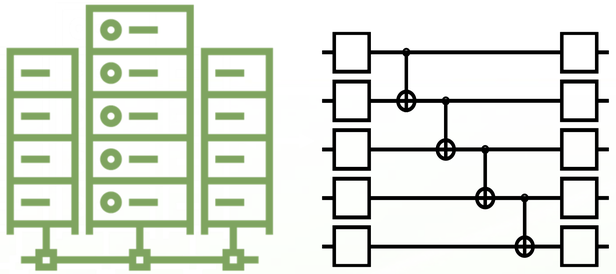
NVIDIA Quantum

Powering the Global Quantum Computing Community

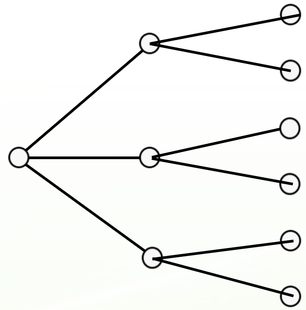


Simulation

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

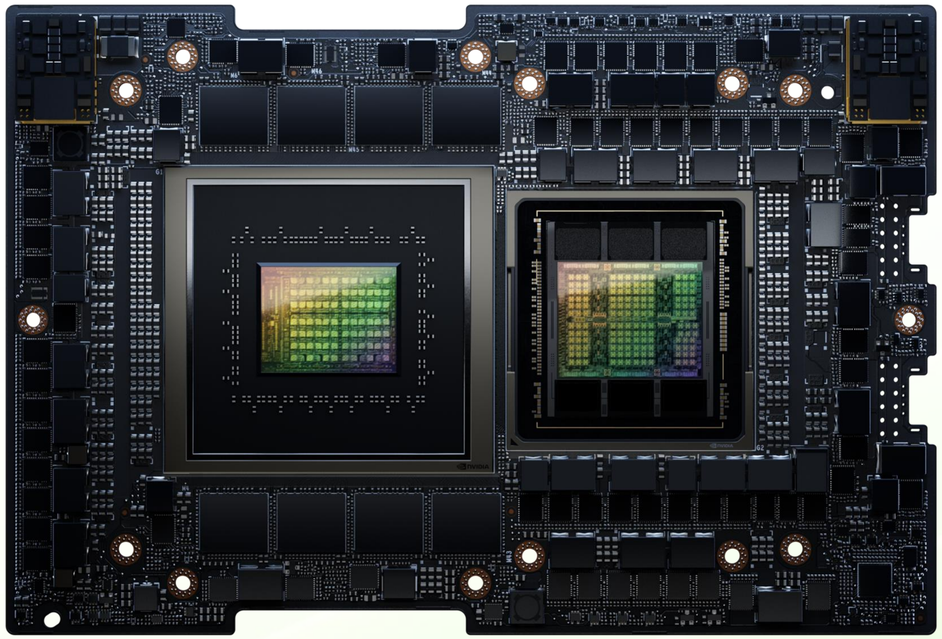
DGX-Q

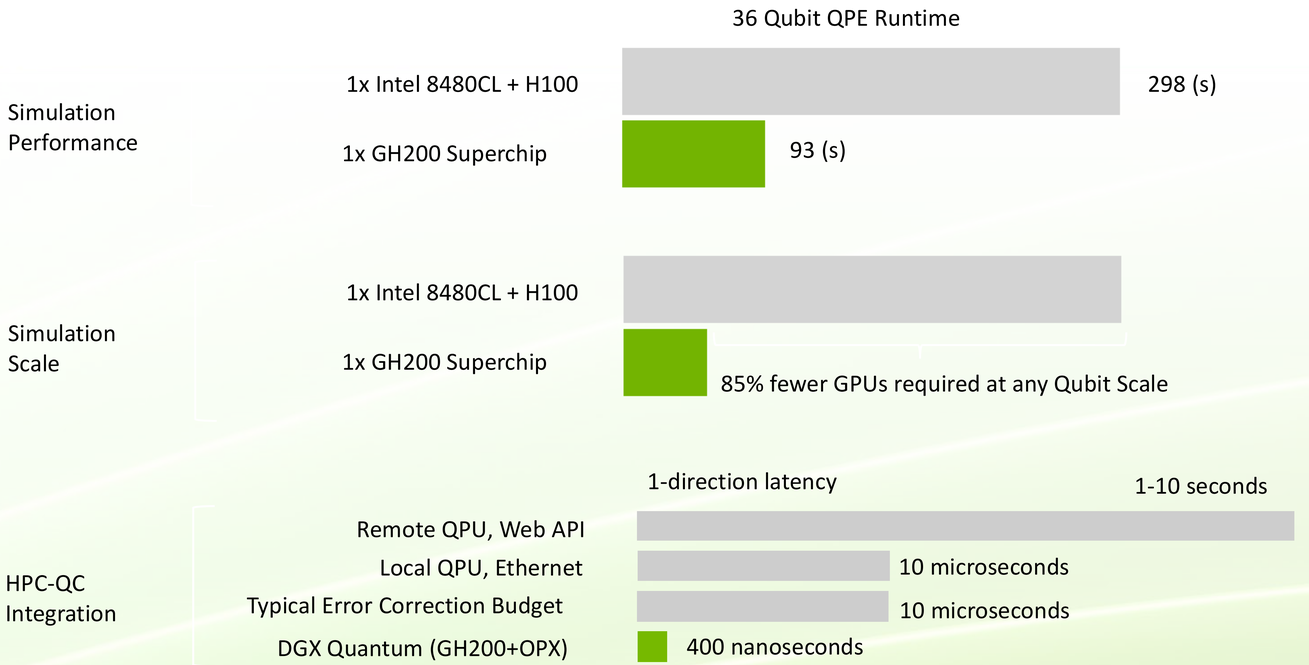
Quantum Integrated Computing

GPU Supercomputing

Grace Hopper: The Engine for Quantum Research

New Grace Hopper Deployments for Quantum Research





CUDA-Q

Enabling Classical and Quantum Accelerated Supercomputing

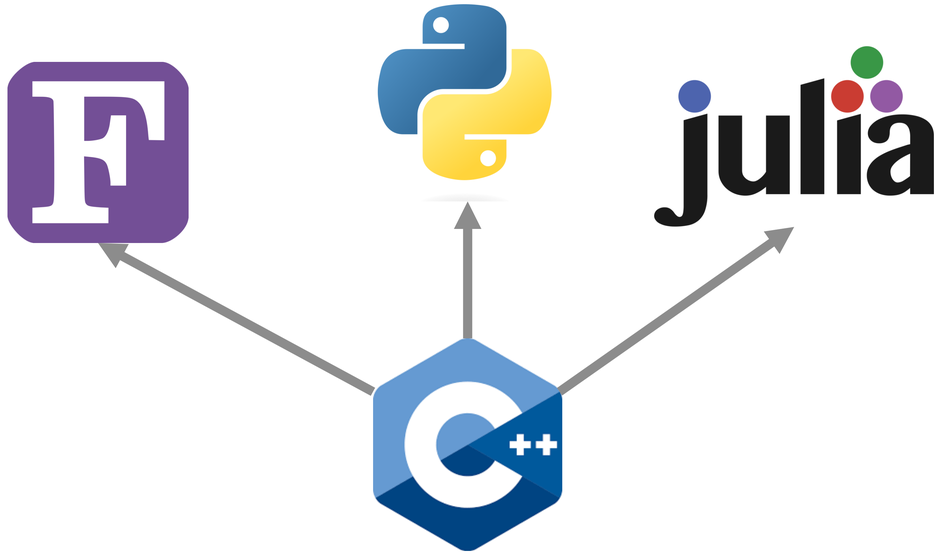
A library-based C++ language extension that compiles directly to the MLIR

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)

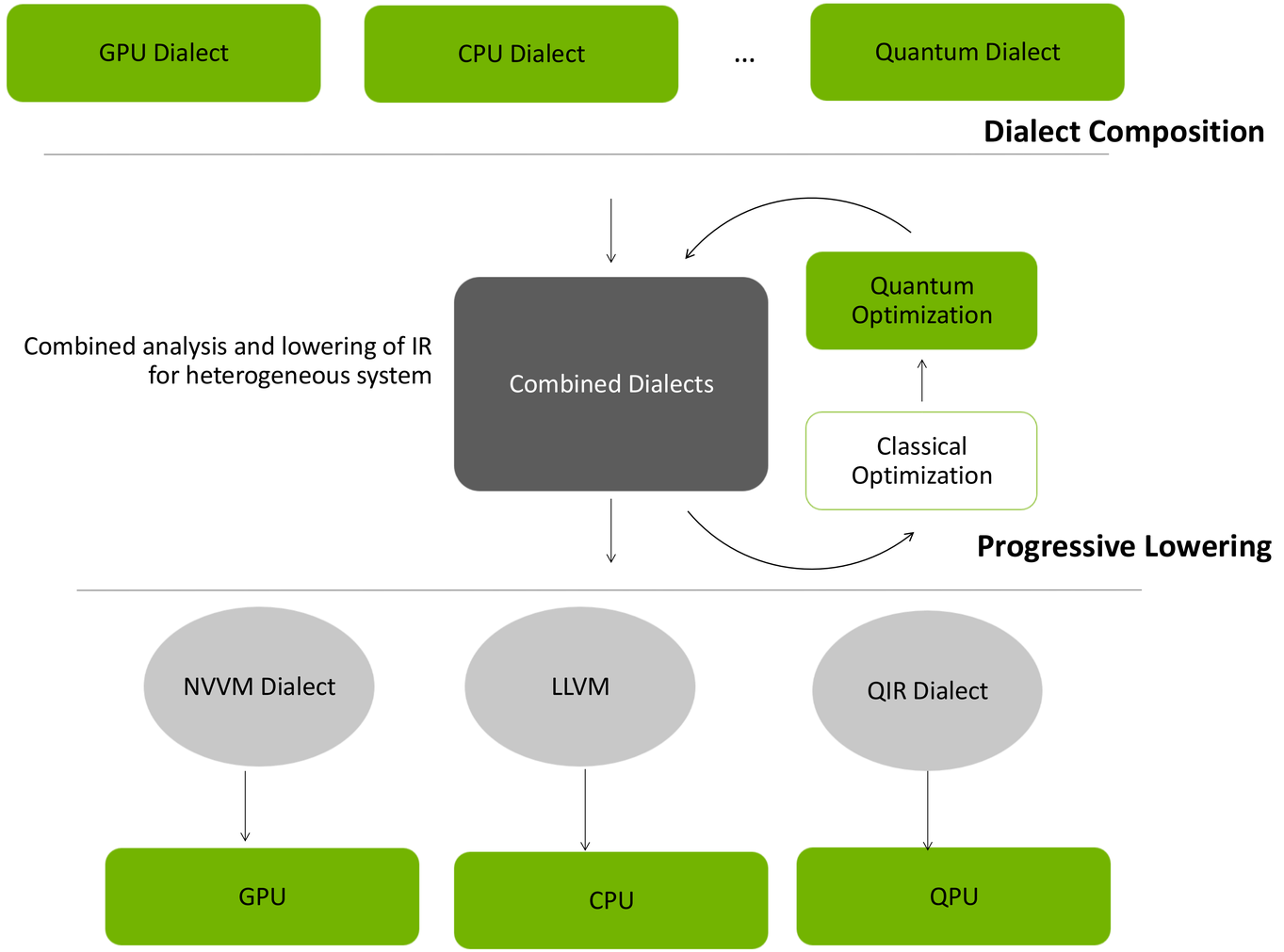



```
// Kernel functions enable clean separation of
// host and device code

__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

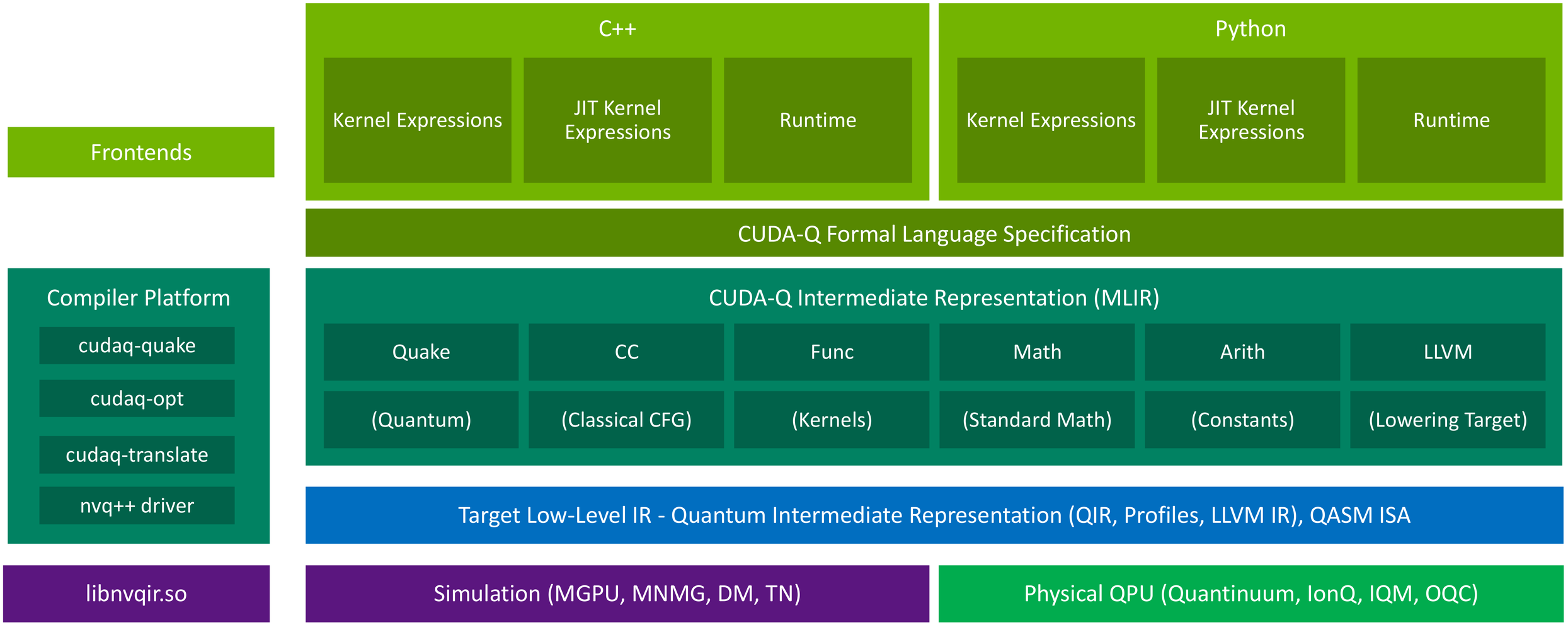
Leverage existing state-of-the-art and enable tight quantum-classical integration at IR level

- ***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

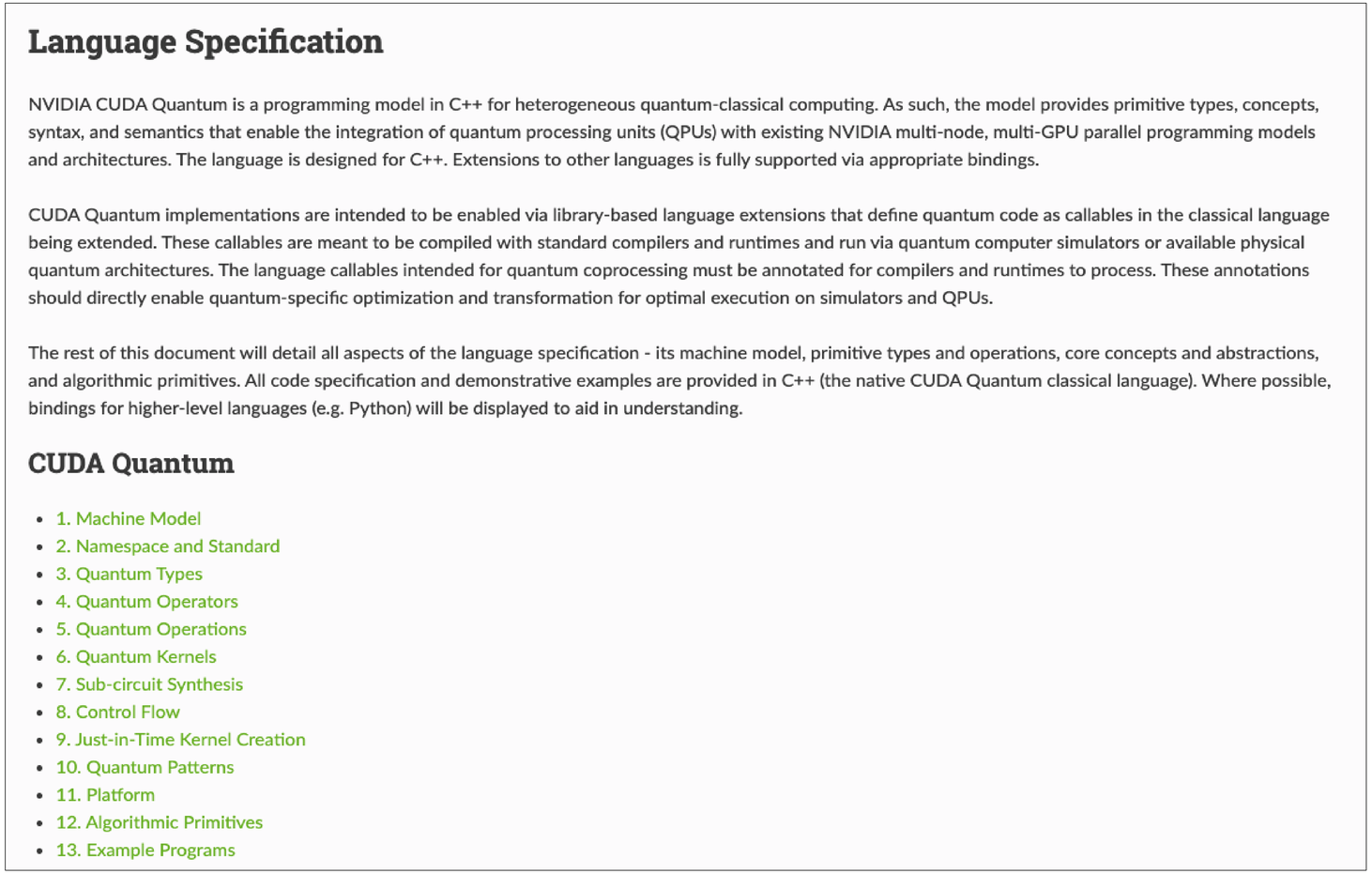
Platform for unified quantum-classical accelerated computing



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…

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

```
__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;
}
```

```
@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(...)`

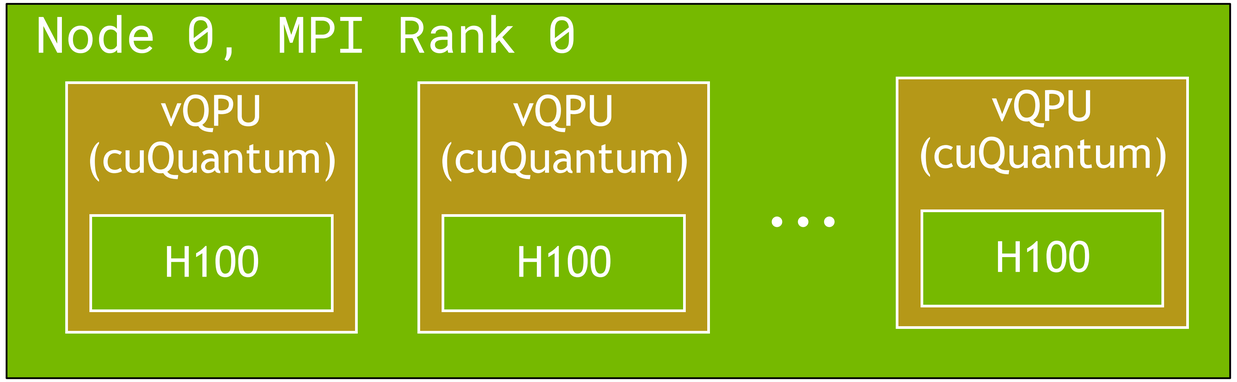
```
@cudaq.kernel
def cnotKernel(q : cudaq.qubit, r : cudaq.qubit):
    x.ctrl(q, r)

@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]);
```

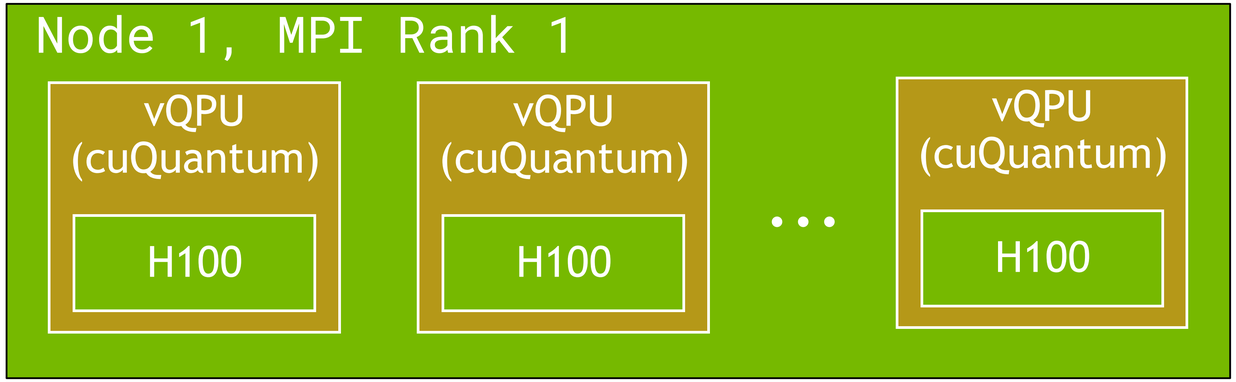
CUDA-Q Platform and Asynchronous Execution

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).

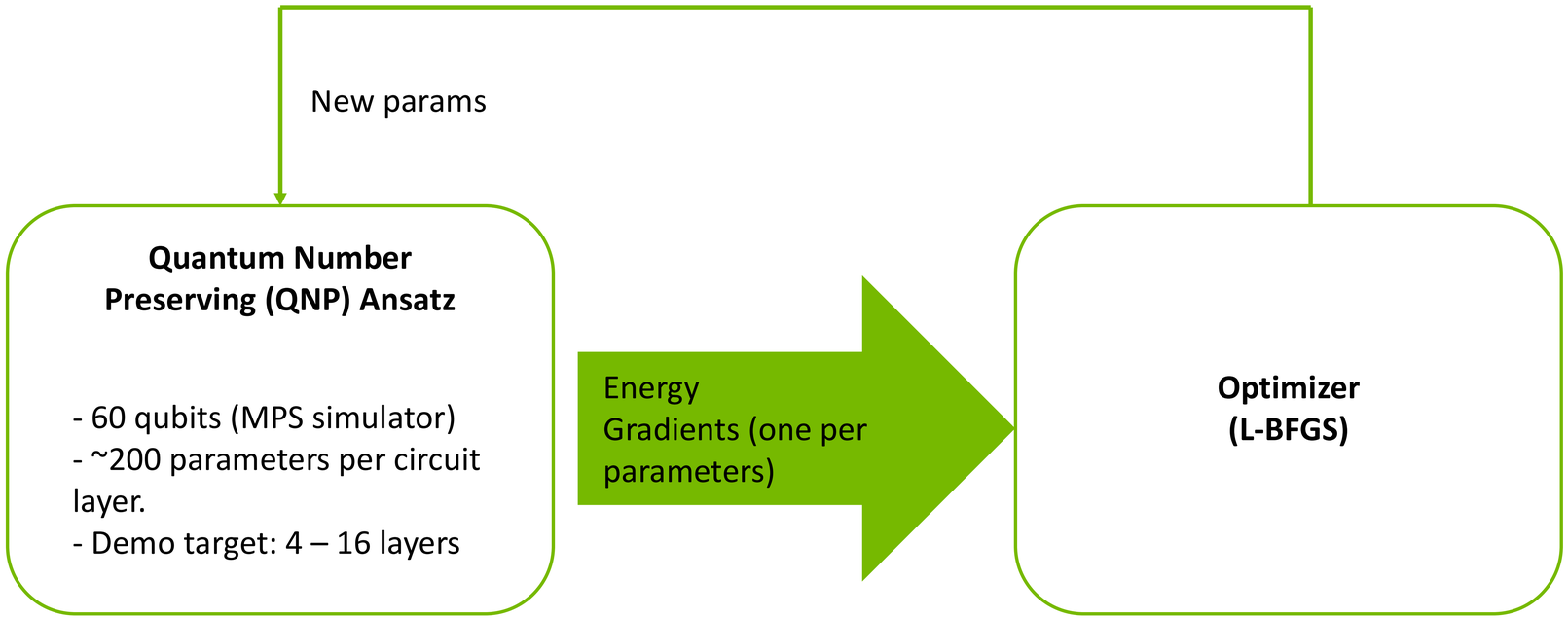
```
// Programmer can query info about the platform
auto& platform = cudaq::get_platform();

// 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);
```





CUDA-Q Platform and Asynchronous Execution

Expose the underlying system architecture to the programmer



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;
};
```

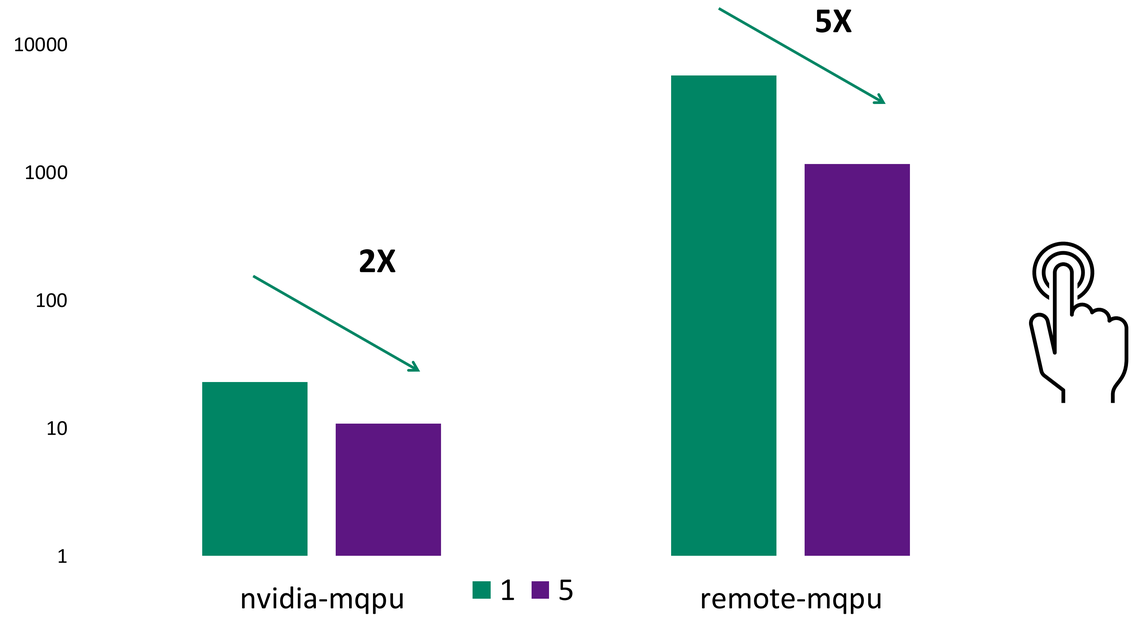
Task List

Async. Dispatch to vQPUs

Gather Data and Compute Gradients

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

```
> n
```



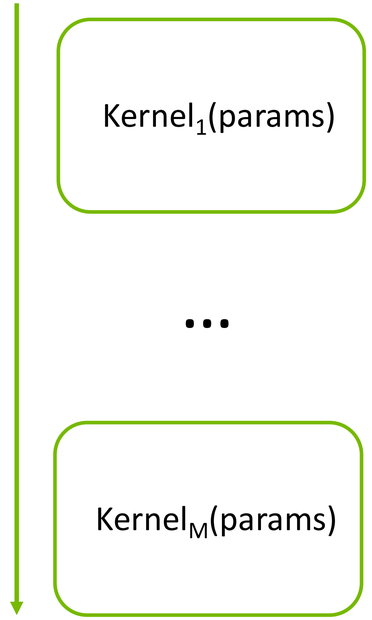
NVIDIA

CUDA-Q Platform and Asynchronous Execution

Expose the underlying system architecture to the programmer

Quantum Machine Learning use case

M Quantum Kernels (data samples)



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

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)

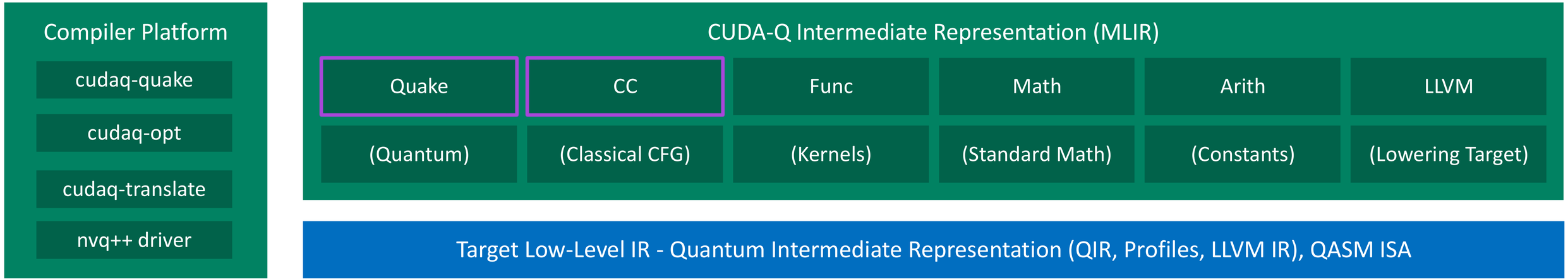
```python
# gather the results from the different ranks
results = comm.gather(states_split, root = 0)

# 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



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

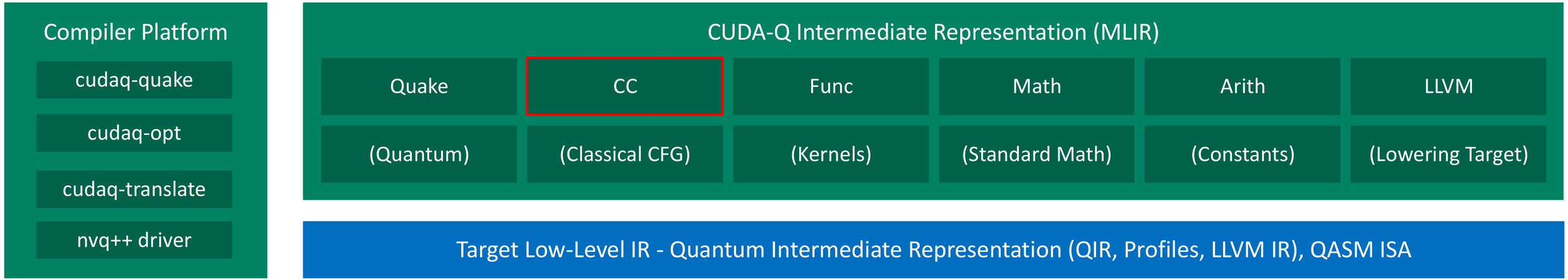
The nvq++ driver orchestrates this collection of these tools to produce hybrid quantum-classical executables and library code.

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



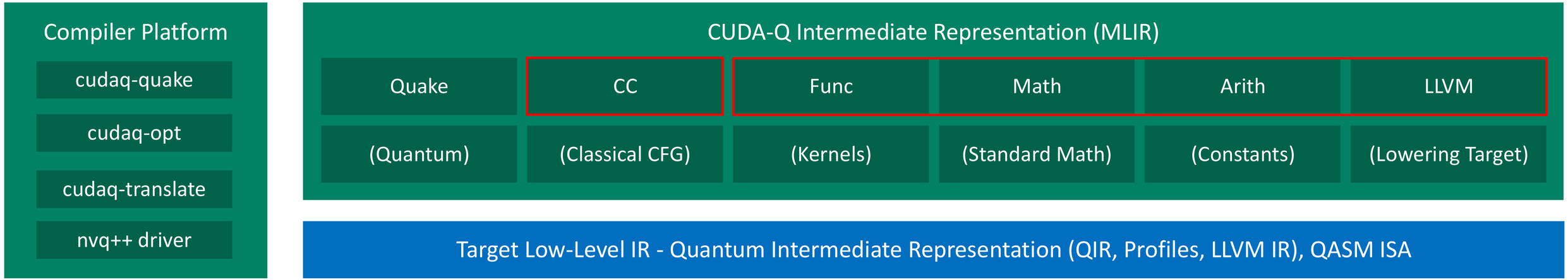

```
__qpu__ double kernel(int N,
        std::vector<double> params) {
   int another = 5;
   return params[0] + params[1];
}
```

```
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




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

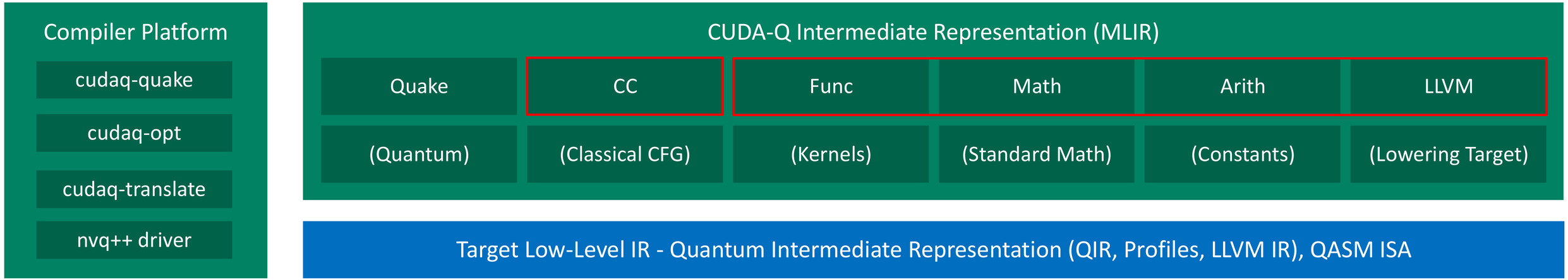
```
module {
 func.func @kernel(%arg0: i32, %arg1: !cc.stdvec<f64>) -> f64 {
  %c5_i32 = arith.constant 5 : i32
  %0 = cc.alloca i32
  cc.store %arg0, %0 : !cc.ptr<i32>
  %1 = cc.alloca i32
  cc.store %c5_i32, %1 : !cc.ptr<i32>
  %2 = cc.stdvec_data %arg1 : (!cc.stdvec<f64>) -> !cc.ptr<f64>
  %3 = cc.compute_ptr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

Reuse other dialects

- 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




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

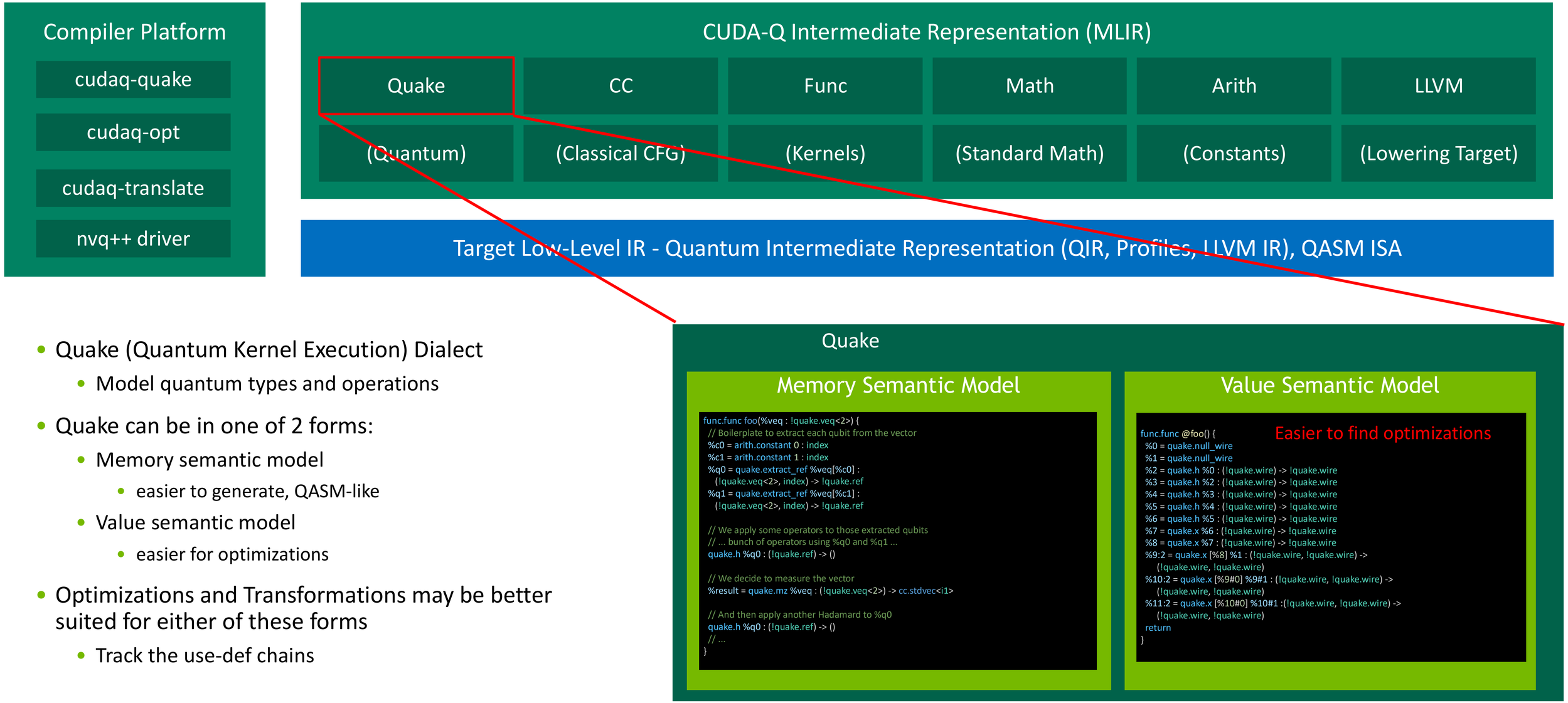
```
llvm.func @kernel(%arg0: i32, %arg1: !llvm.struct<(ptr, i64)>) -> f64 {
  %0 = llvm.mlir.constant(5 : i32) : i32
  %1 = llvm.mlir.constant(1 : i32) : i32
  %2 = llvm.alloca %1 x i32 : (i32) -> !llvm.ptr
  llvm.store %arg0, %2 : i32, !llvm.ptr
  %3 = llvm.alloca %1 x i32 : (i32) -> !llvm.ptr
  llvm.store %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

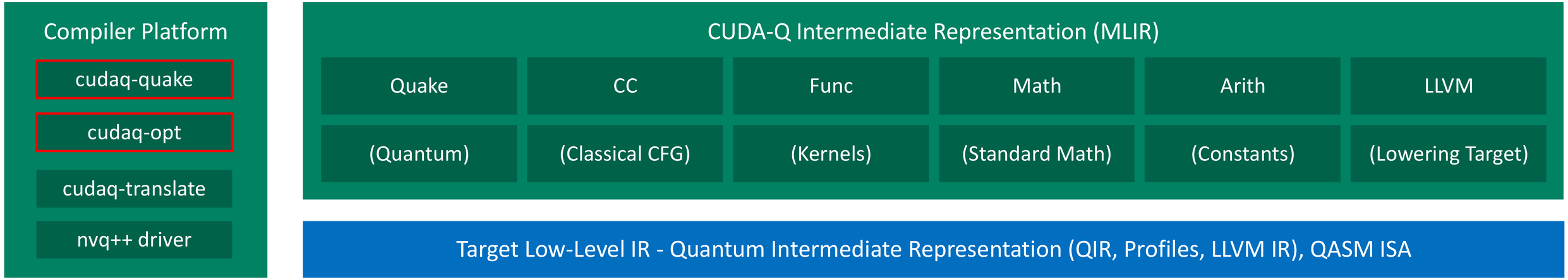
Platform for unified quantum-classical accelerated computing



- 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

The CUDA-Q Compiler Stack

Platform for unified quantum-classical accelerated computing



- 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

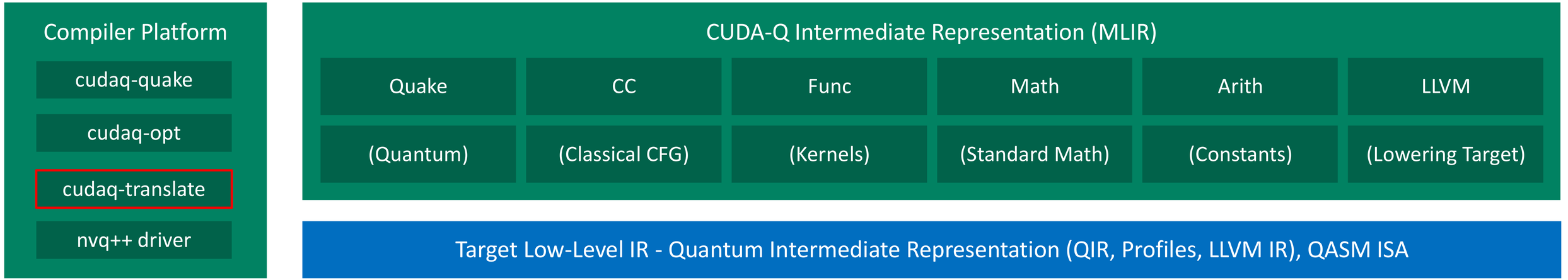
```
__qpu__ void ghzForLoop(int N) {
  cudaq::qreg q(N);
  h(q[0]);
  for (std::size_t i = 0; i < N - 1; i++)
    x<cudaq::ctrl>(q[i], q[i + 1]);
  mz(q);
}
```

```
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



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

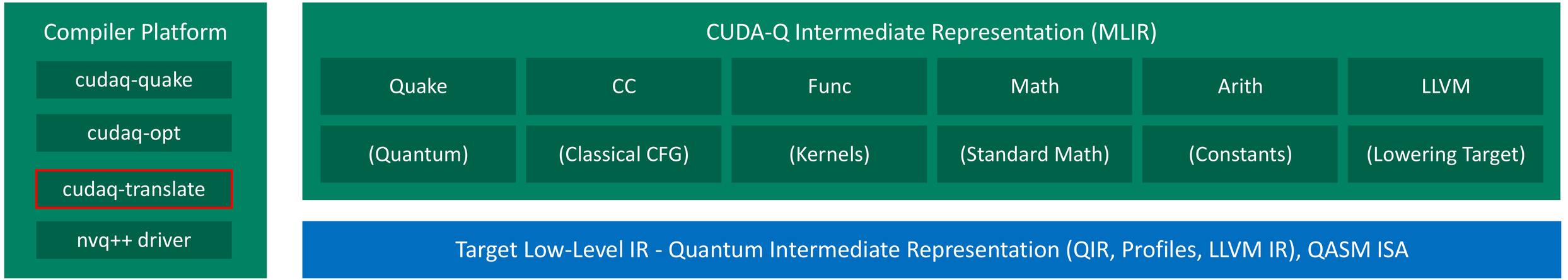
```
__qpu__ void simple() {
  cudaq::qubit q, r;
  h(r);
  x(q);
}
```

```
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



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

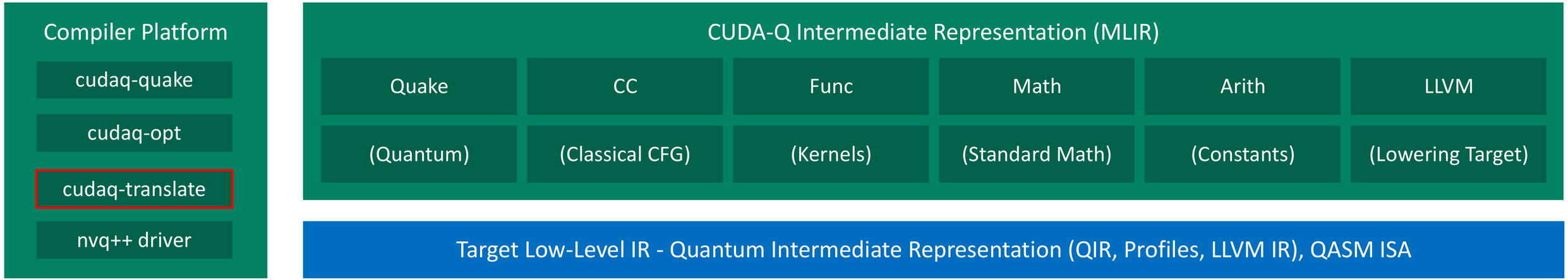
```
__qpu__ void simple() {
  cudaq::qubit q, r;
  h(r);
  x(q);
}
```

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



- 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

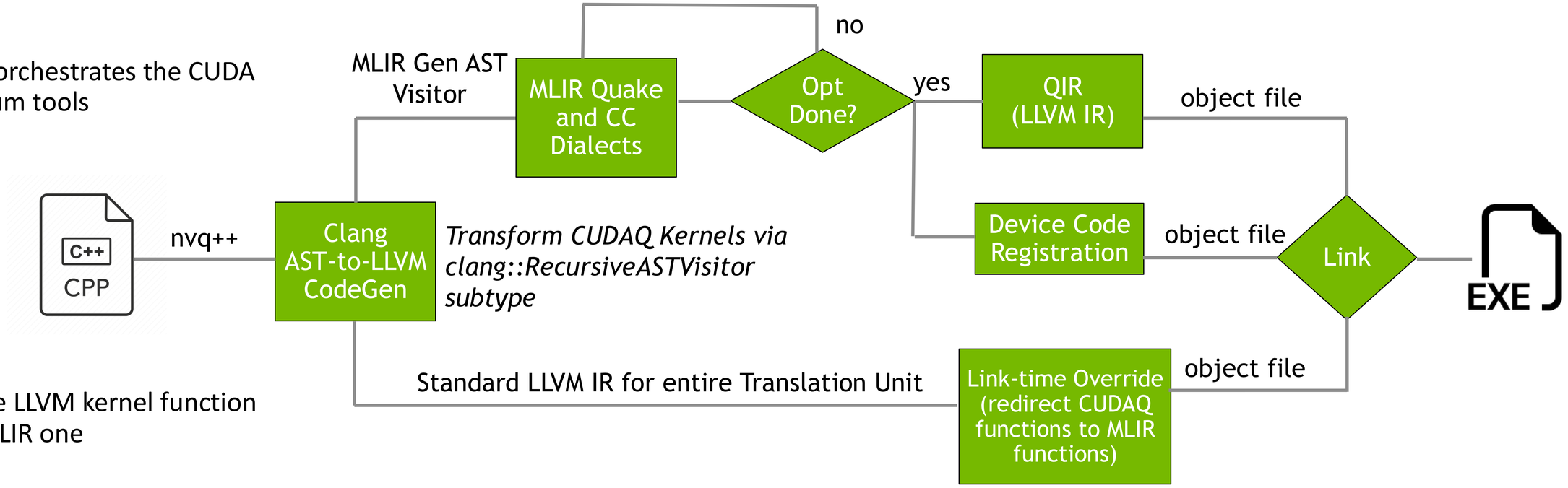
- cudaq-quake
- cudaq-opt
- cudaq-translate
- nvq++ driver

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



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

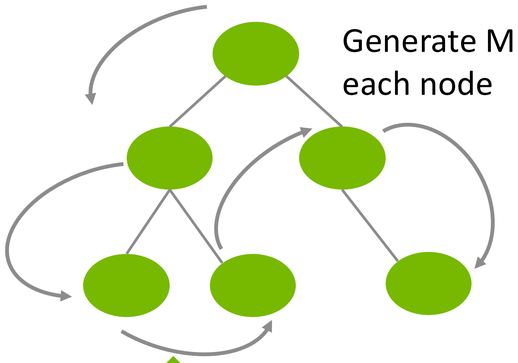
```python
@cudaq.kernel
def bell():
    q = cudaq.qvector(2)
    h(q[0])
    x.ctrl(q[0], q[1])
```

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

JIT compile, extract function pointer


```
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
}
```

CUDA-Q in Action

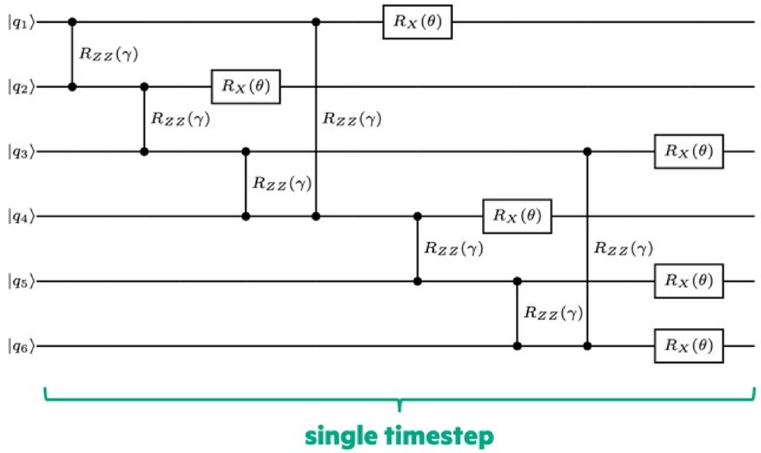
- Simulation Acceleration
- Hybrid QPU-GPU applications

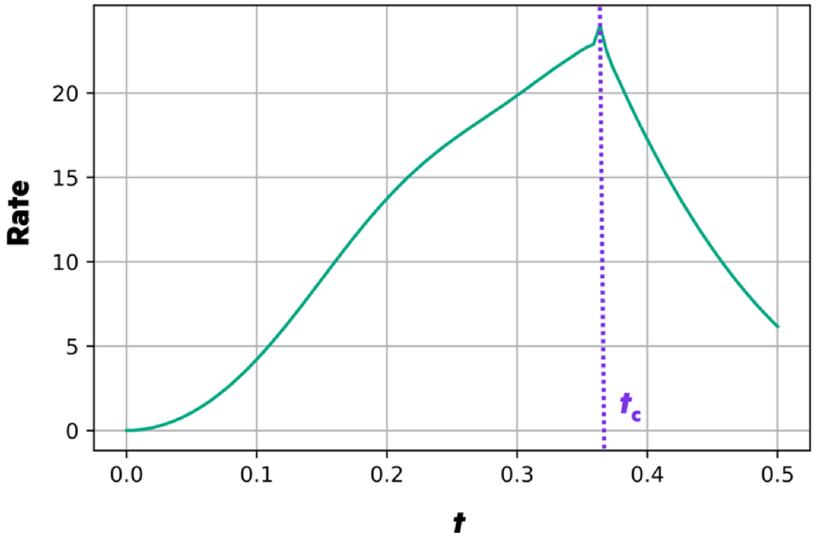
CUDA-Q in Action

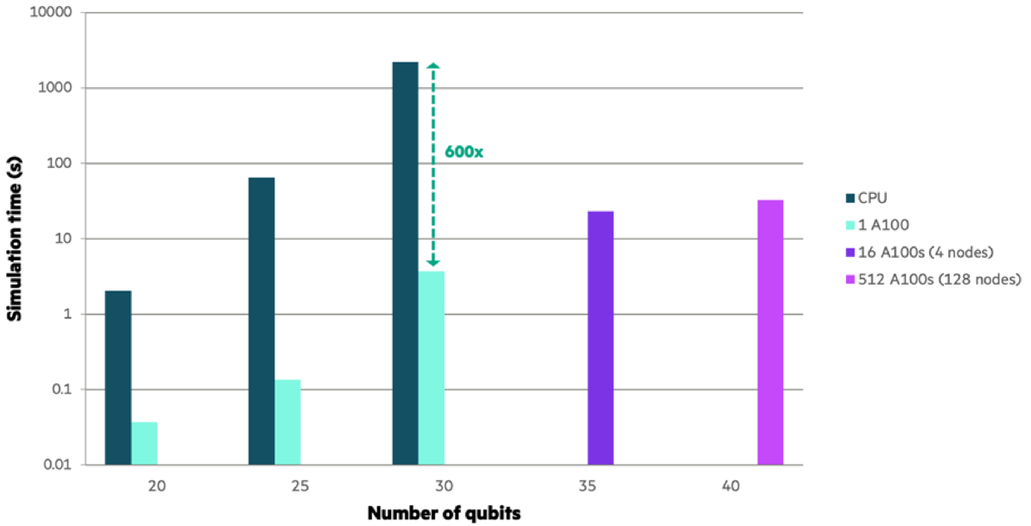
Speed-ups for time-evolution of the transverse field Ising model (TFIM)

- 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))
```





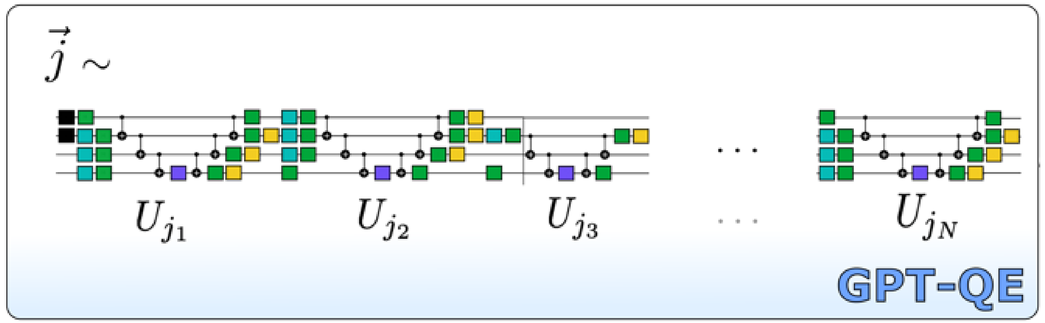


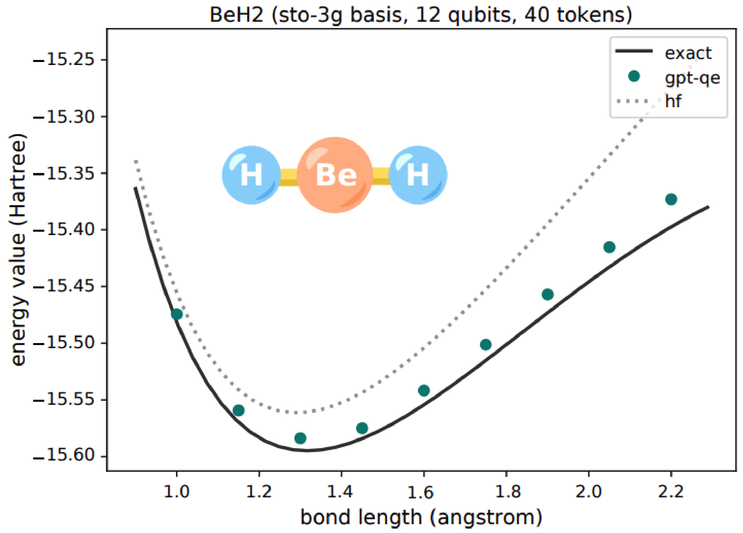
NVIDIA

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





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

CUDA-Q performance updates

Improved Gate Fusion

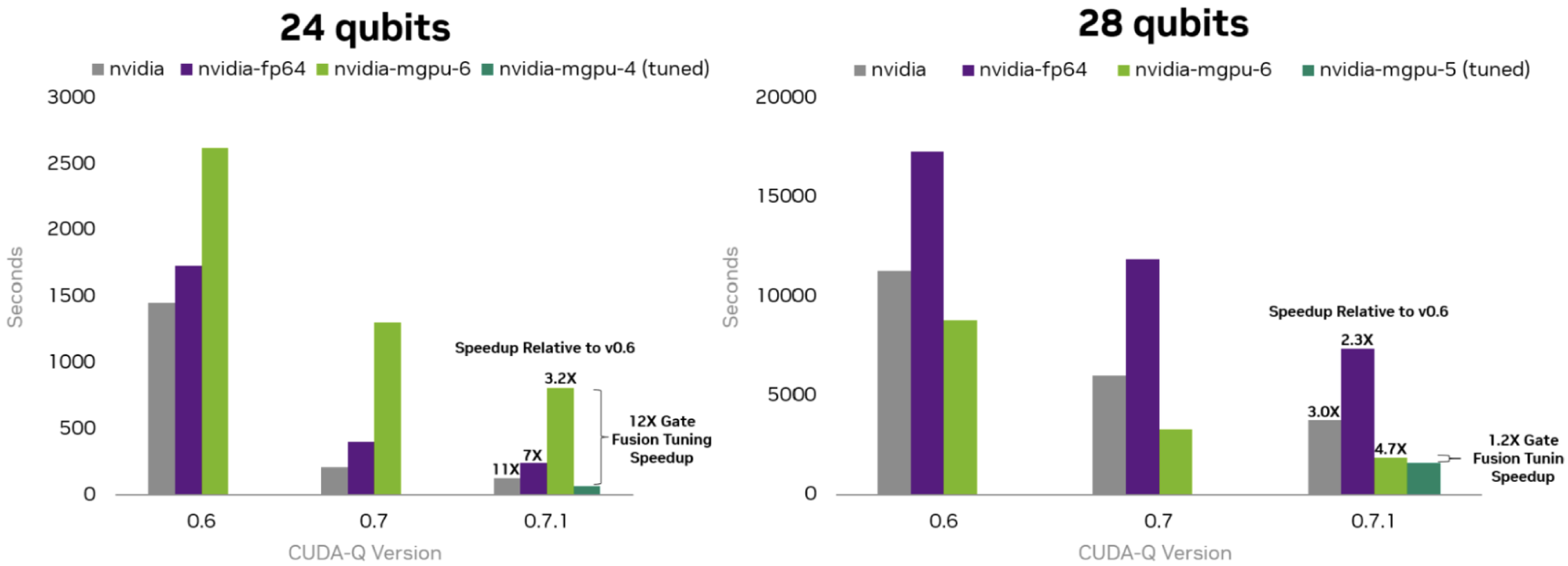


Figure 1. Execution times for 10 `observe` calls in 24 and 28 qubit UCCSD-VQE experiments

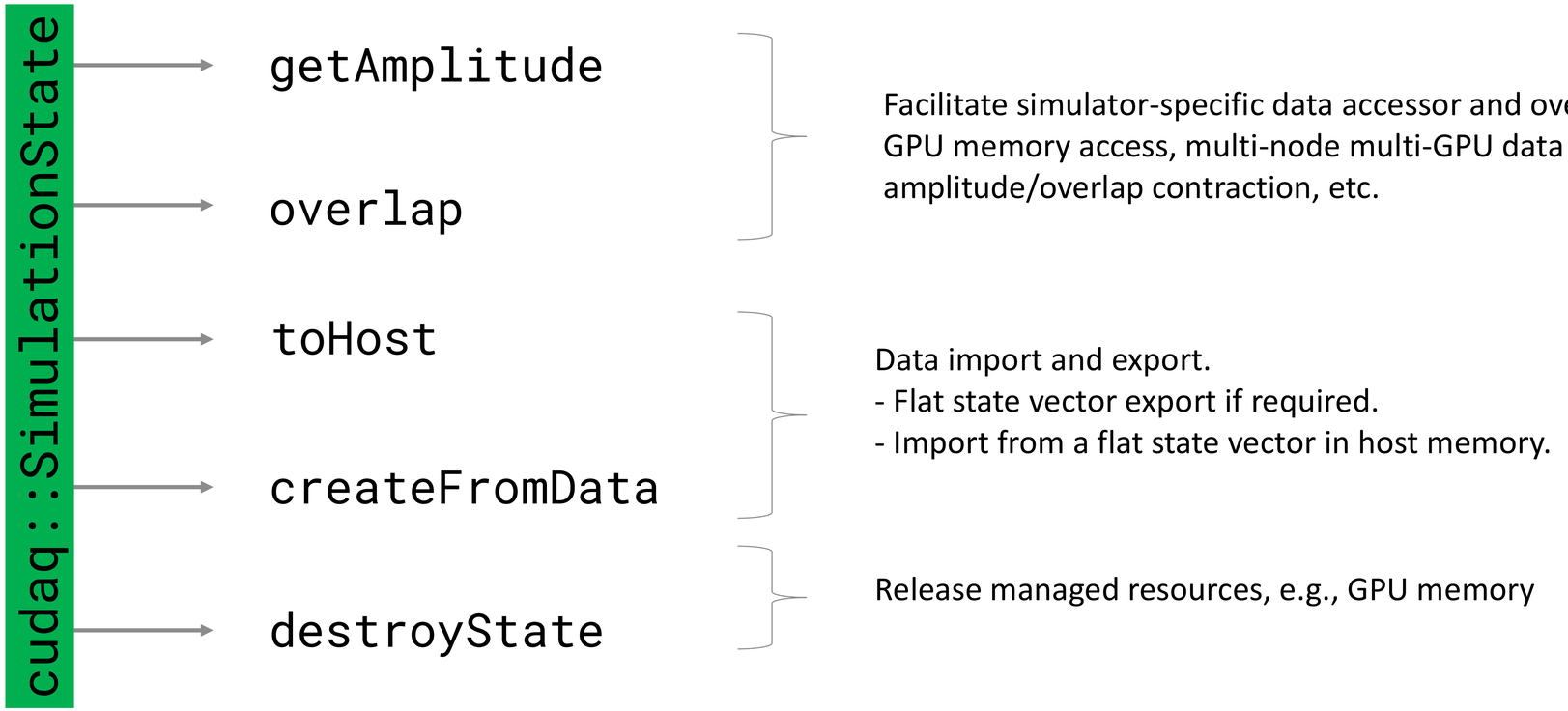
https://developer.nvidia.com/blog/new-nvidia-cuda-q-features-boost-quantum-application-performance/

CUDA-Q: 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.)

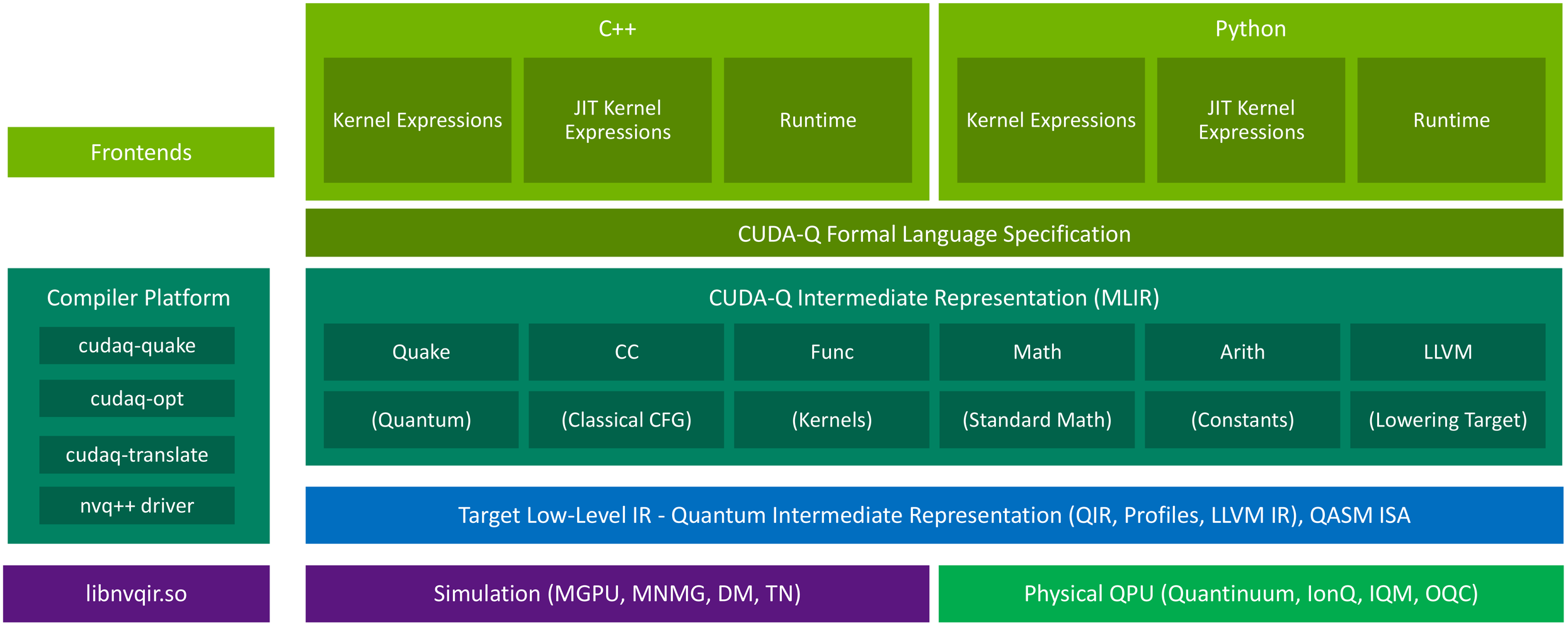


- `cudaq::state` = user-facing class with simplified life-cycle management (RAII)

CUDAQ-QEC

The CUDA-Q Stack

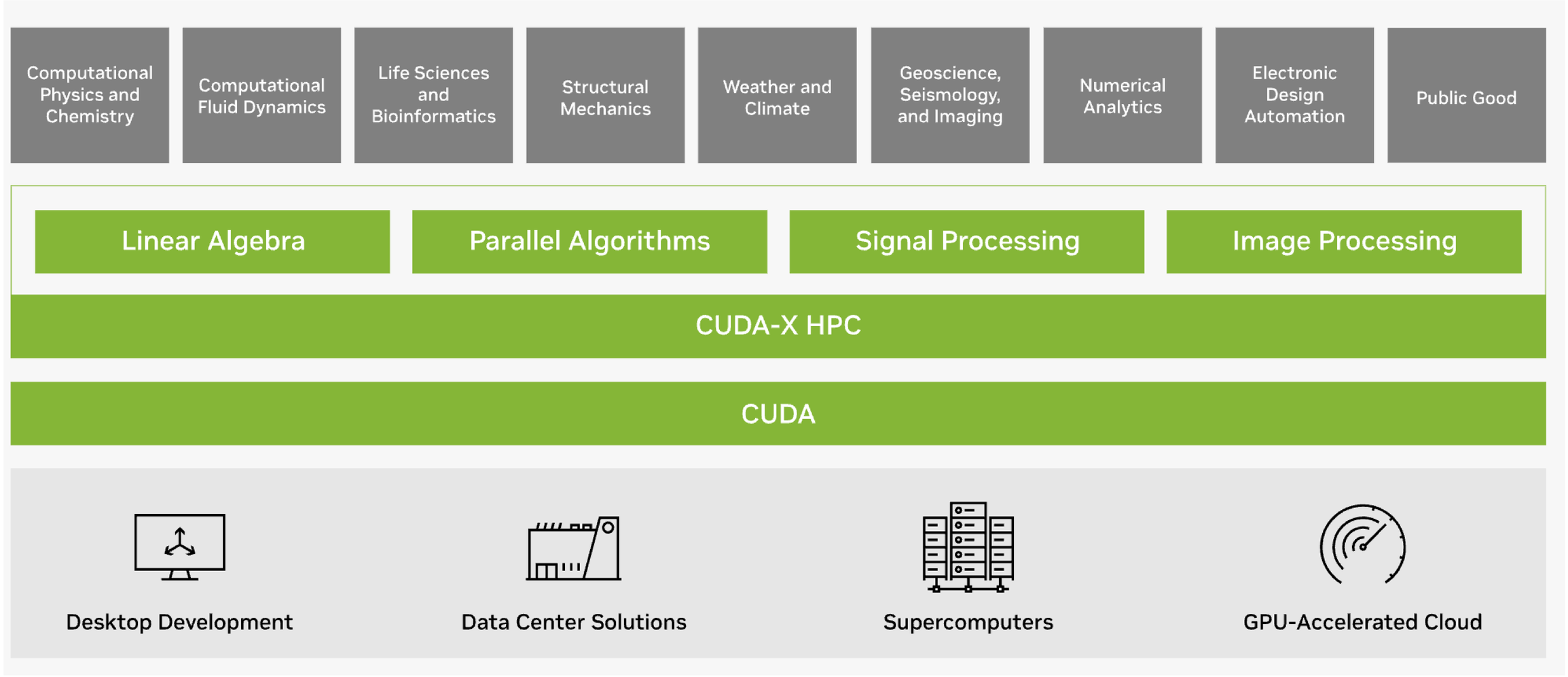
Platform for unified quantum-classical accelerated computing



Our goal is to build upon this foundation and provide building blocks for application development at higher levels of abstraction

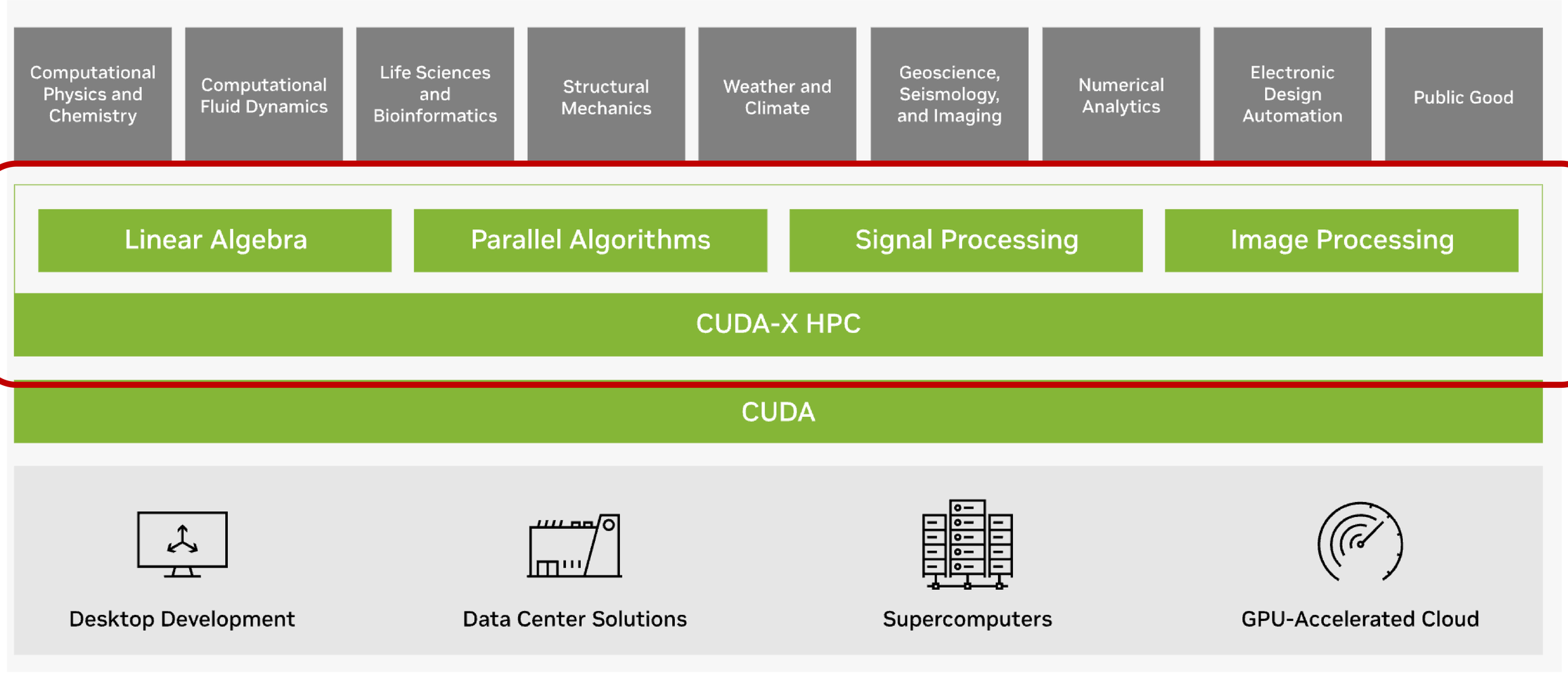
CUDA-X Libraries

Libraries built on top of CUDA

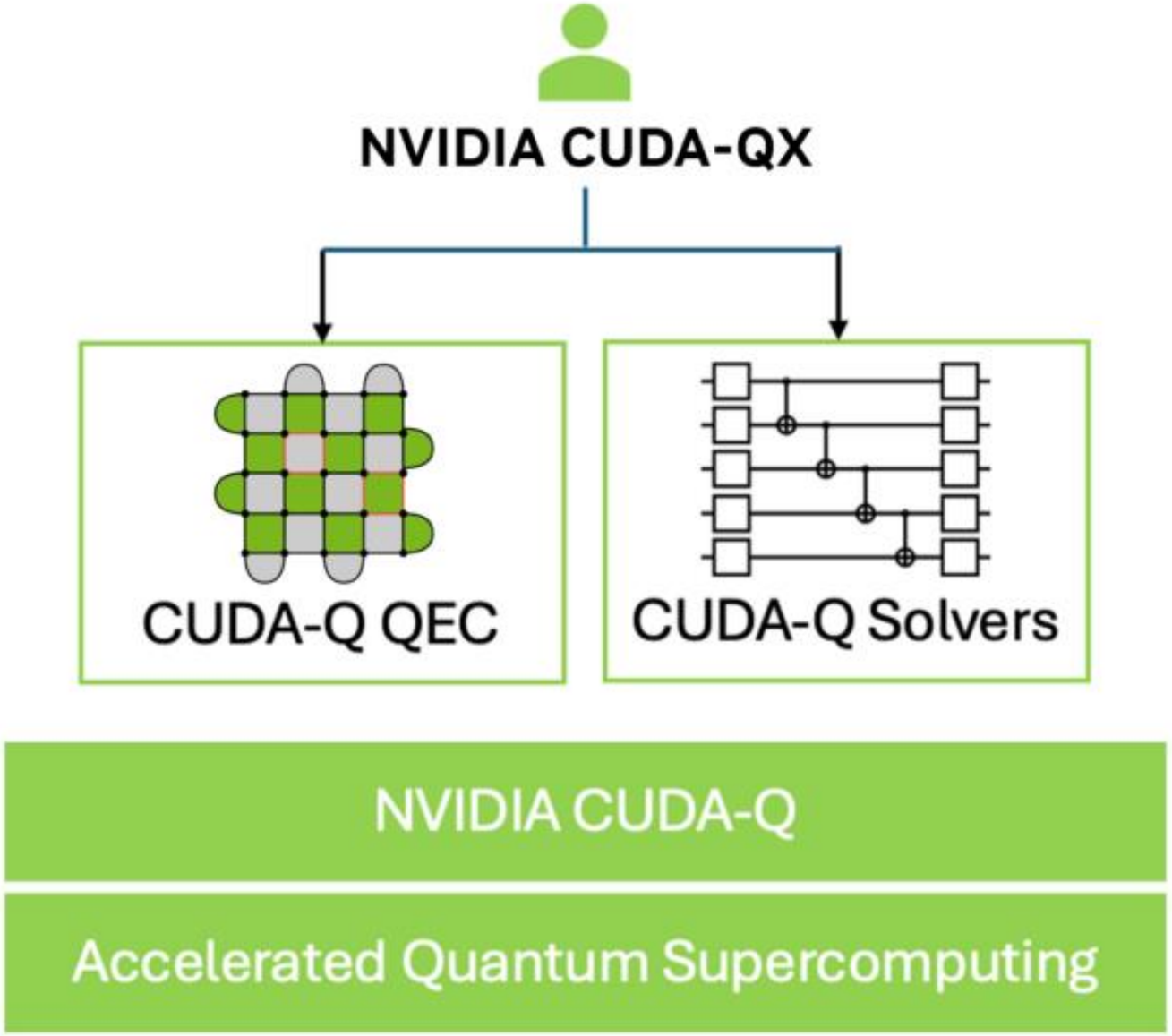


CUDA-X Libraries

Libraries built on top of CUDA



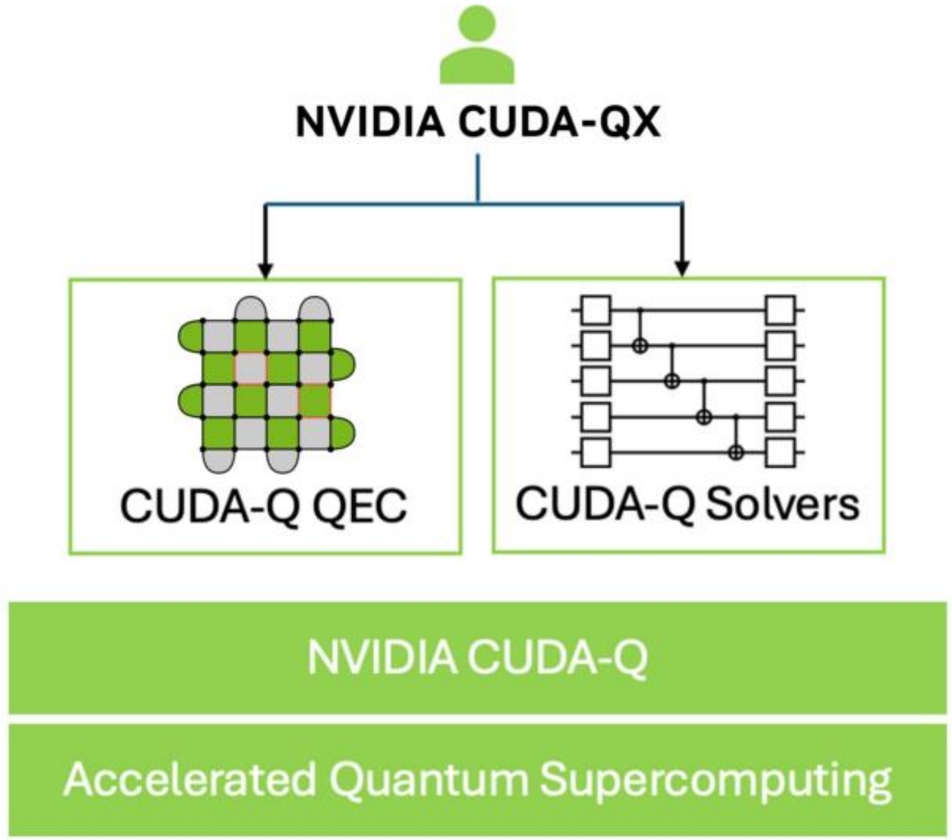
The CUDA-QX Libraries



CUDA-QX

Libraries built on top of CUDA-Q

- QEC: Quantum Error Correction
  - Encoding logical qubits
  - Decoding
  - Numerical Experiments
- Solvers:
  - Implementations of quantum-classical algorithms
  - Integration with quantum chemistry tools
  - Quantum and classical optimizers
- More to come!



CUDAQ-QEC

Quantum Error Correction

- Quantum computers are very noisy today
  - Research community still can’t find practical uses for noisy machines
  - Long road ahead to fully fault-tolerant quantum computers
- QEC aims to allow quantum computation to be fault-tolerant
  - Active field of research into how this can be done at scale
  - Initial promising experiments this past year
    - https://www.nature.com/articles/s41567-024-02479-z
    - https://www.nature.com/articles/s41586-024-08449-y
    - https://arxiv.org/abs/2404.02280
  - Rapidly growing field

Quick Overview of (classical) Error Correcting Codes (ECC)

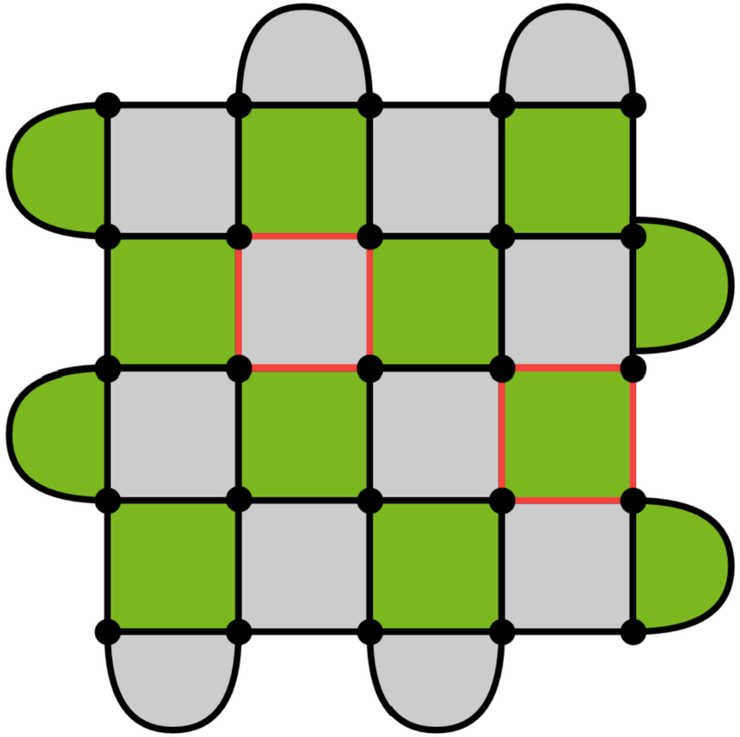
- Three main components:
  - Encode a “logical” bit into many bits
    - 0_L --> 00000
    - 1_L --> 11111
  - Procedure to detect errors
    - Receive anything other than {00000, 11111} --> error!
  - Procedure to correct errors
    - Majority vote: 00010 --> 00000 (works up to 2 errors, redundancy == protection)
- Most codes can be cast into the language of parity check equations.
  - 4 data bits, 1 parity bit: [d0, d1, d2, d3, p0]
    - p0 = d0 + d1 + d2 + d3 (mod 2)
    - Can detect 1 error (correct 0 errors)
  - 1 data bit, 4 parity bits: [d0, p0, p1, p2, p3]
    - p0 = d0
    - p1 = d0
    - p2 = d0
    - p3 = d0
    - equivalent to the repetition code above (can correct 2 errors)

Quick Overview of (classical) Error Correcting Codes (ECC)

- Modern ECCs are much more advanced, but principles are the same
  - Encode
  - Decode
  - Correct
- Many codes with their various trade-offs
  - Block codes, Convolutional codes, LDPC, Turbo,…
- Increasing redundancy gives better error correction, but you have a worse data ”rate”.
- Modern EC codes with good trade-offs can have complicated structures to decode.
  - Part of the Aerial 5g project here at NVIDIA is implementing performant GPU decoders

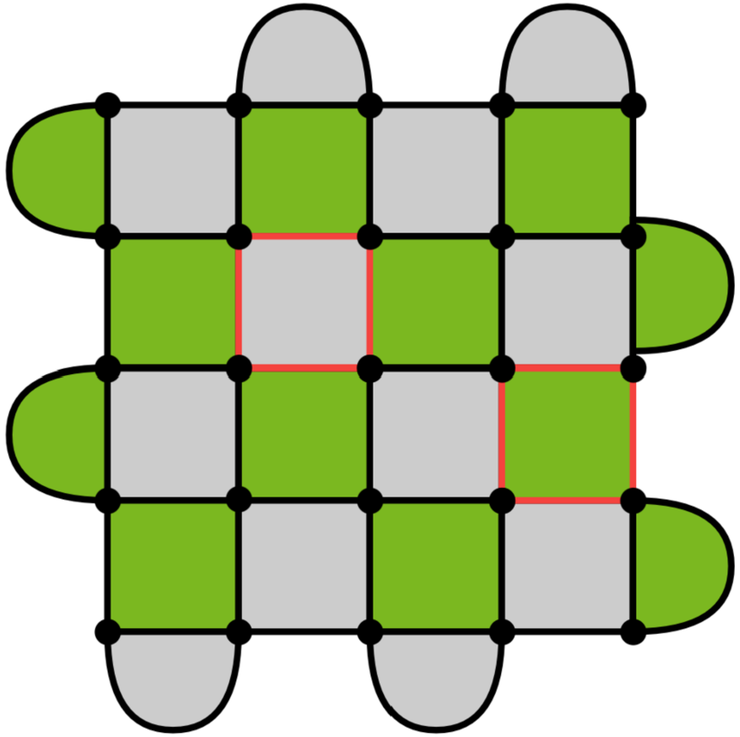
Quantum Error Correction

- Significantly more challenging than classical, but same ideas
- Encode “logical” qubit into many qubits
  - $|0\rangle_L = |00000\rangle$
  - $|1\rangle_L = |11111\rangle$
  - $|\psi\rangle_L = \alpha|0\rangle_L + \beta|1\rangle_L$
- Decode
- Correct
- Many codes with their various trade-offs
  - Shor, Steane, Surface code, LDPC, …



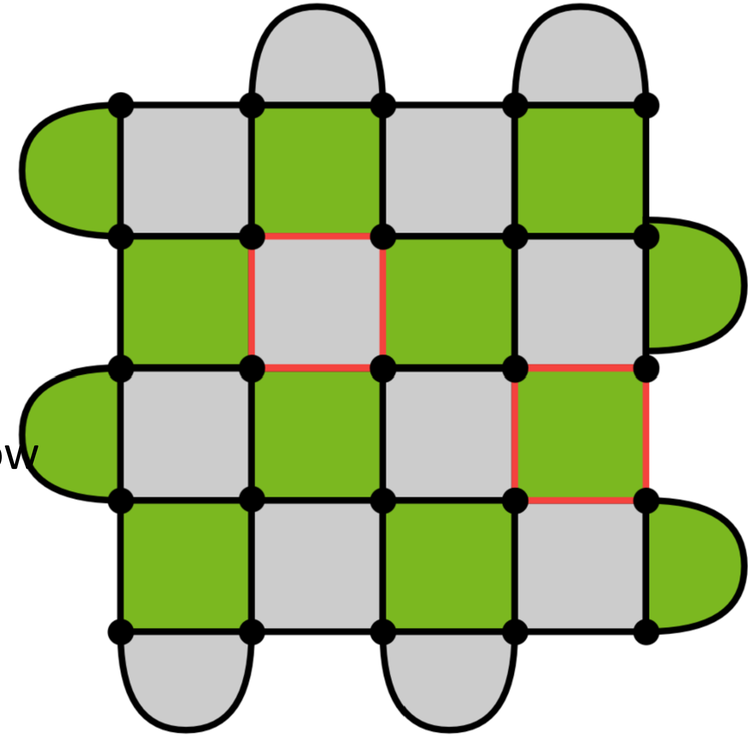
Intro Quantum Error Correction

- Challenges for QEC:
  - Bit --> Qubit: 2 states -> 2D vector space of states
  - No cloning theorem means we can’t copy data
  - Can’t even look at our qubits without collapse
  - Only unitary/reversible operations allowed
- Too much to go into now
  - But these are all resolvable
  - Actually better than classical analog error correction!
  - QEC course for CUDA-Q academic in progress!



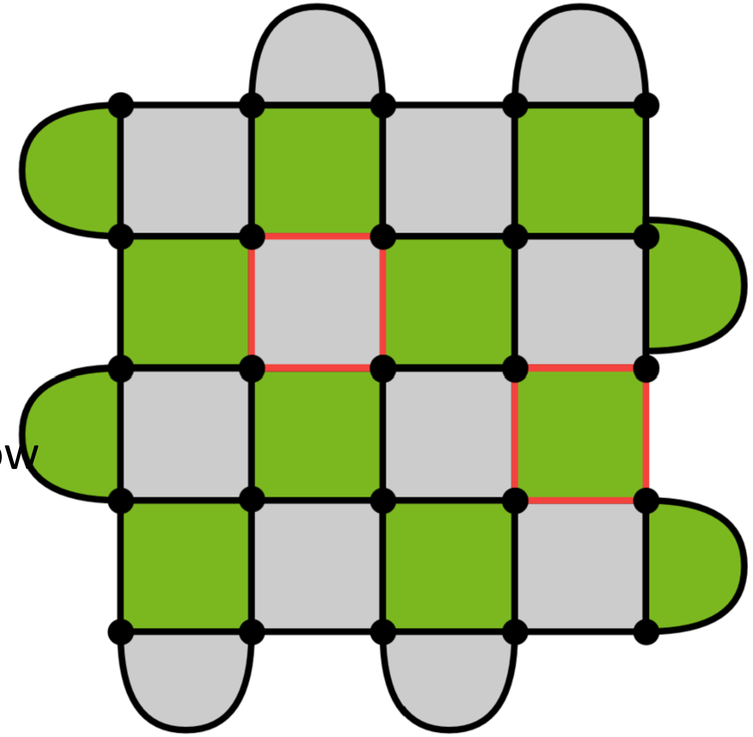
Intro Quantum Error Correction

- Small intuition how we resolve this:
  - Parity checks on our data qubits
  - By checking the parity of collections of qubits, we're not projecting out any quantum information
- Stabilizers: +1 eigenvalue subspace
  - $Z_0 Z_3 |\psi> = +1 |\psi>$
  - Bitflip error! X_0
  - $Z_0 Z_3 X_0 |\psi> = -X_0 Z_0 Z_3 |\psi> = -X_0 |\psi>$
- Perform these stabilizer measurements, and any -1 readout now tells us something about an error.
- Decode these and correct the error



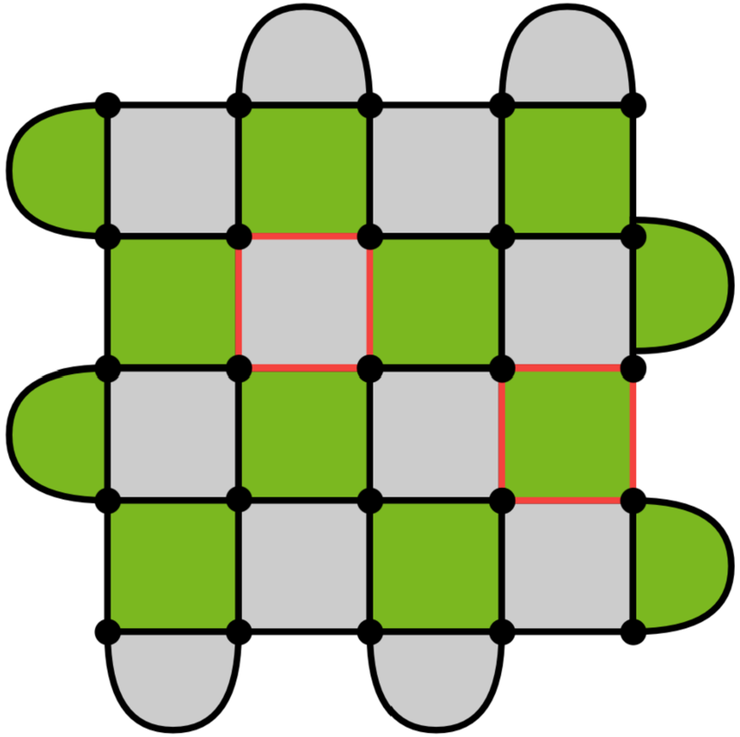
Intro Quantum Error Correction

- Small intuition how we resolve this:
  - Parity checks on our data qubits
  - By checking the parity of collections of qubits, we're not projecting out any quantum information
- Stabilizers: +1 eigenvalue subspace
  - $Z_0 Z_3 |\psi> = +1 |\psi>$
  - Bitflip error! X_0
  - $Z_0 Z_3 X_0 |\psi> = -X_0 Z_0 Z_3 |\psi> = -X_0 |\psi>$
- Perform these stabilizer measurements, and any -1 readout now tells us something about an error.
- Decode these and correct the error



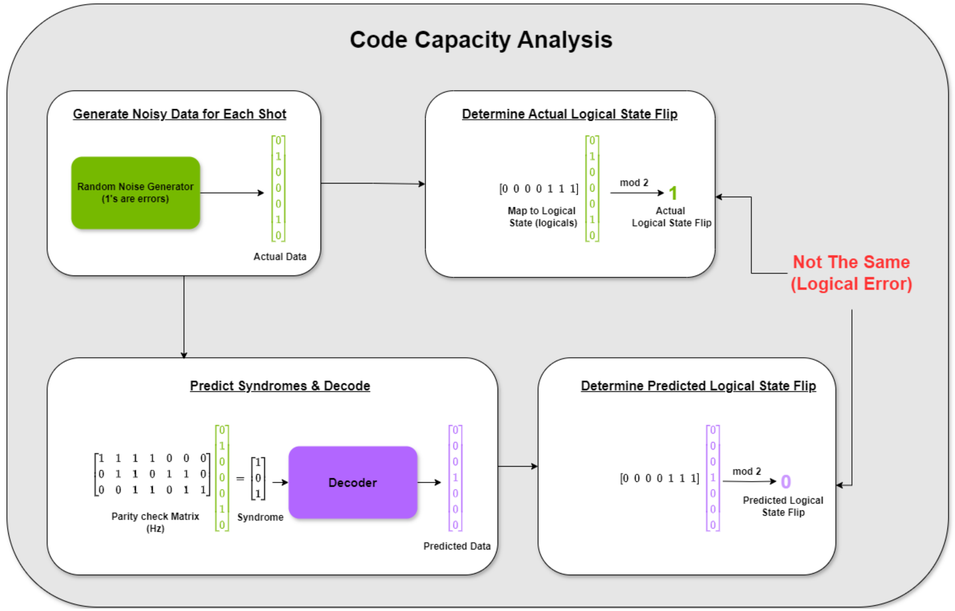
Quantum Error Correction Decoding

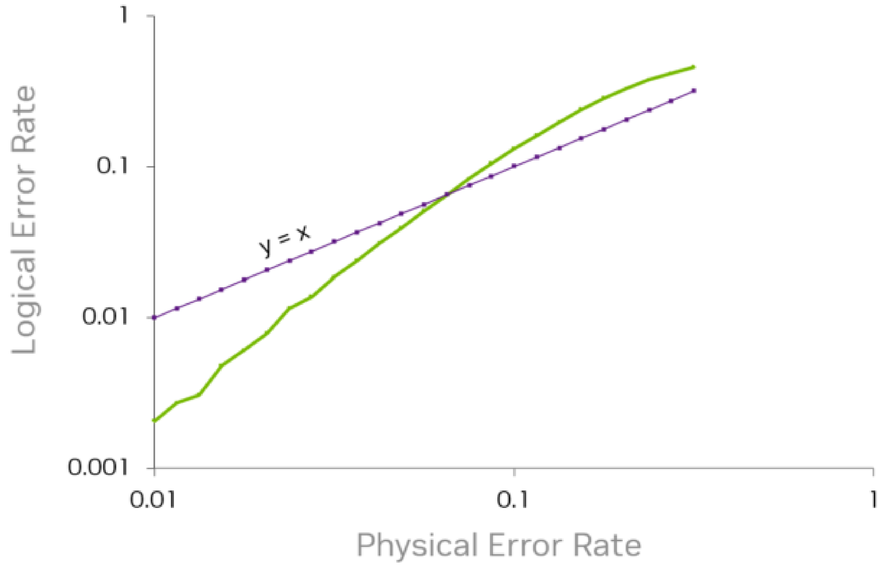
- Decoding:
  - Given a set of parity check measurements, deduce the error.
  - Find the most probable one
- Degenerate Quantum Maximum Likelihood Decoding
  - #P-Complete (ouch)
- Quantum Maximum Likelihood Decoding
  - NP-Complete (ouch)
- Graph-like decoding
  - Minimum weight perfect matching (MWPM)
  - Polynomial!
- Belief-propagation decoding
  - Extra steps needed for convergence
  - Polynomial!



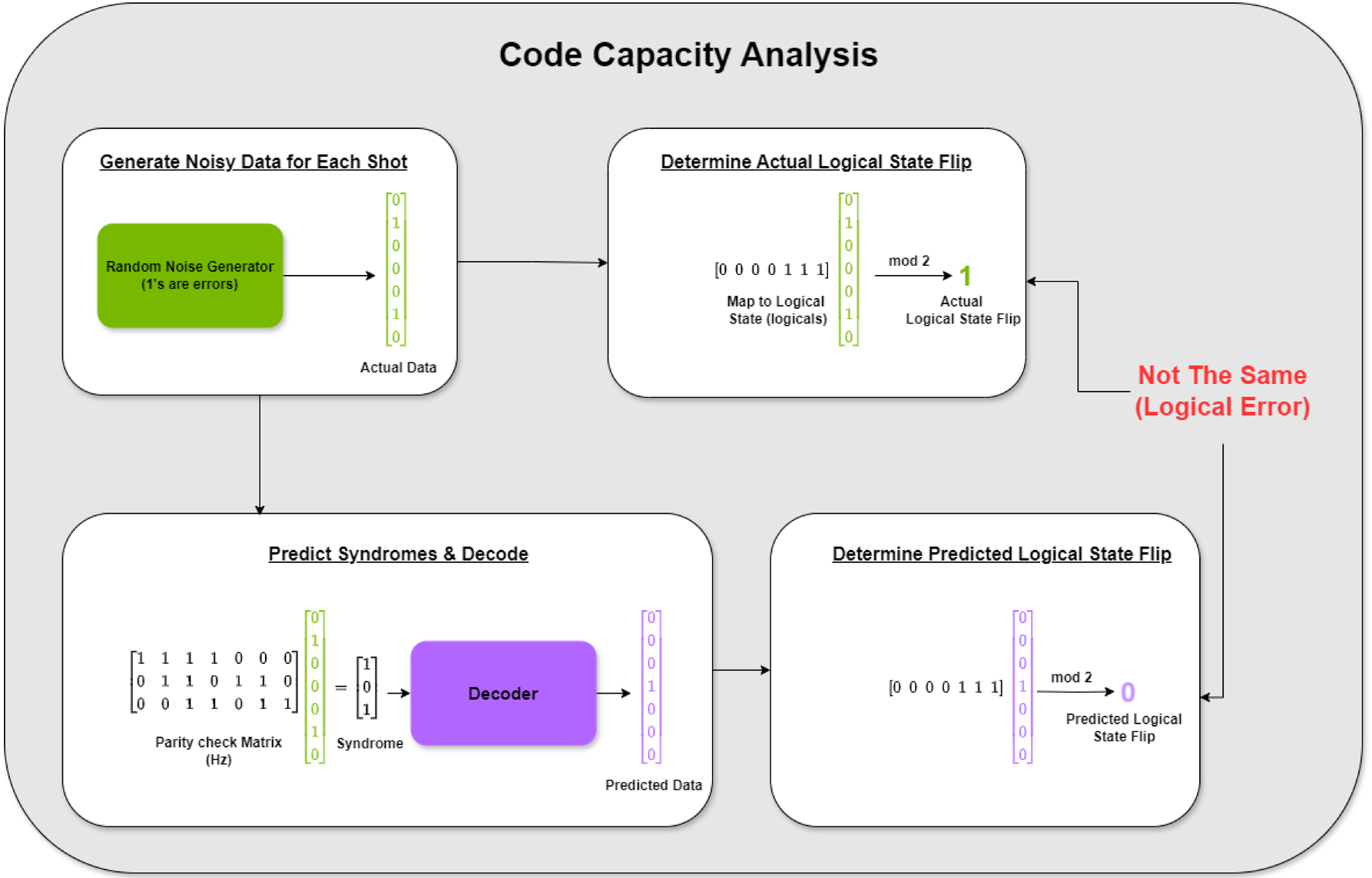
CUDAQ-QEC

- Encode: cudaq::qec::code
  - Provide subtypes, plus extension point
- Decode: cudaq::qec:decoder
  - Provide subtypes, plus extension point
- Correct: cudaq::qec::sample_memory_circuit
  - Run simulations with CUDA-Q





CUDAQ-QEC Workflow



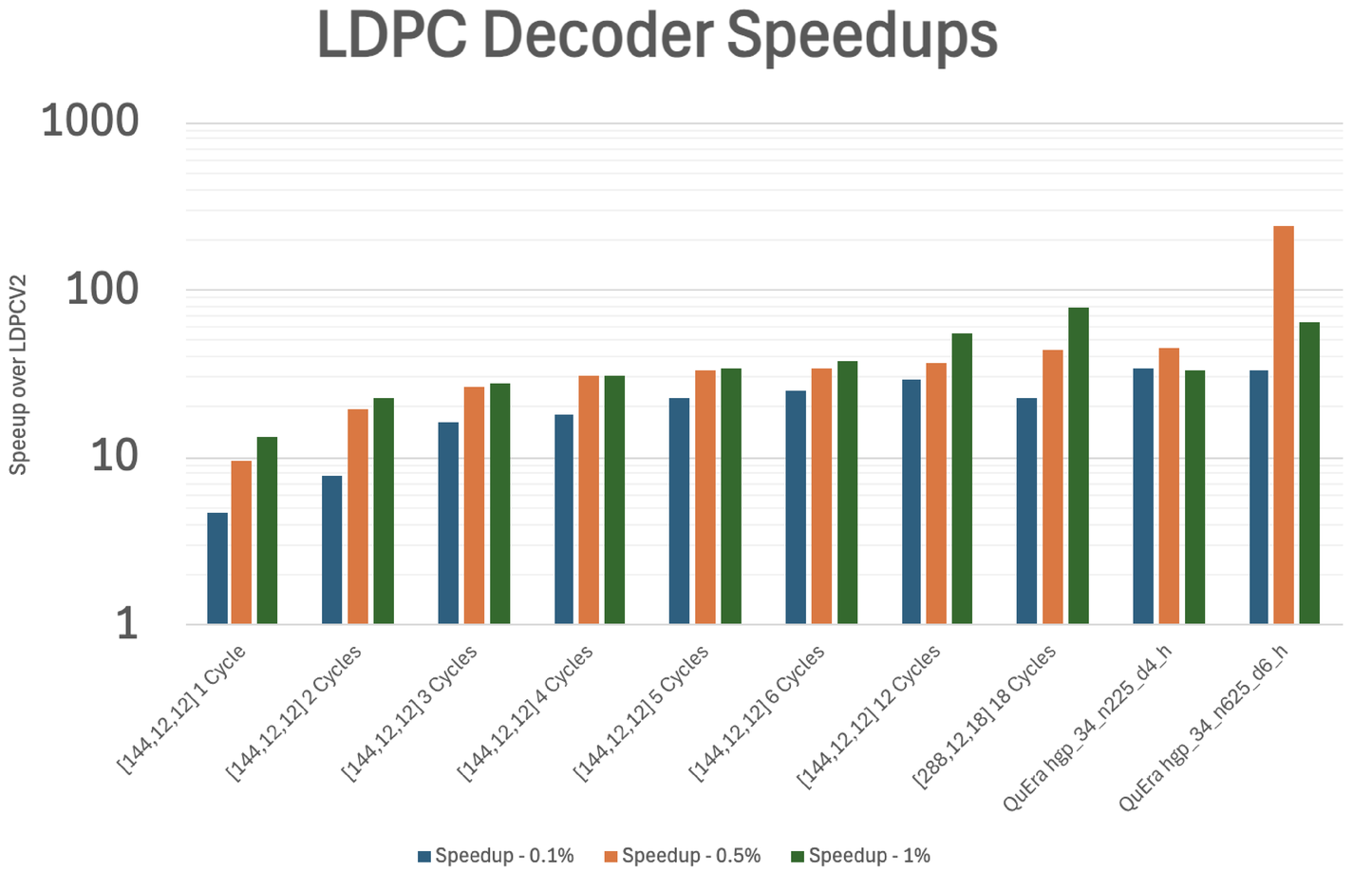
cudaq::qec::code

```python
@cudaq.kernel
def prep0(logicalQubit: patch):
    h(logicalQubit.data[0], logicalQubit.data[4], logicalQubit.data[6])
    x.ctrl(logicalQubit.data[0], logicalQubit.data[1])
    x.ctrl(logicalQubit.data[4], logicalQubit.data[5])
    # ... additional initialization gates ...

@cudaq.kernel
def stabilizer(logicalQubit: patch,
               x_stabilizers: list[int],
               z_stabilizers: list[int]) -> list[bool]:
    # Measure X stabilizers
    h(logicalQubit.ancx)
    for xi in range(len(logicalQubit.ancx)):
        for di in range(len(logicalQubit.data)):
            if x_stabilizers[xi * len(logicalQubit.data) + di] == 1:
                x.ctrl(logicalQubit.ancx[xi], logicalQubit.data[di])
    h(logicalQubit.ancx)

    # Measure Z stabilizers
    for zi in range(len(logicalQubit.ancx)):
        for di in range(len(logicalQubit.data)):
            if z_stabilizers[zi * len(logicalQubit.data) + di] == 1:
                x.ctrl(logicalQubit.data[di], logicalQubit.ancz[zi])

    # Get and reset ancillas
    results = mz(logicalQubit.ancz, logicalQubit.ancx)
    reset(logicalQubit.ancx)
    reset(logicalQubit.ancz)
    return results
```

```cpp
enum class operation {
    x,      // Logical X gate
    y,      // Logical Y gate
    z,      // Logical Z gate
    h,      // Logical Hadamard gate
    s,      // Logical S gate
    cx,     // Logical CNOT gate
    cy,     // Logical CY gate
    cz,     // Logical CZ gate
    stabilizer_round,  // Stabilizer measurement round
    prep0, // Prepare |0⟩ state
    prep1, // Prepare |1⟩ state
    prepp, // Prepare |+⟩ state

my_code::my_code(const heterogeneous_map& options) : code() {
    // Register operations
    operation_encodings.insert(
      std::make_pair(operation::x, x));
    operation_encodings.insert(
      std::make_pair(operation::stabilizer_round, stabilizer));

    // Define stabilizer generators
    m_stabilizers = qec::stabilizers({"XXXX", "ZZZZ"});
}
```

cudaq::qec::decoder

- Simple Look-up-table
- Bring your own
- LDPC decoder:
  - Belief Propagation
  - Additional convergence methods
  - Hybrid GPU + CPU approach
- In progress:
  - Machine learning decoders



CUDAQ-QEC Decoders

- High performance decoders are a priority
  - GPU acceleration potential especially as quantum computers grow
  - ML decoders can outperform SOTA
  - First NVIDIA quantum decoder released in March
- Performance is critical both in quality, and speed
  - Quality means fewer errors
  - Characteristic time scale of quantum computer sets speed of solution
  - ~1ms for atoms/ions
  - ~1us for superconductors
- Performance needed outside of real-time as well
  - QEC researchers need fast decoders for simulation workflows
  - Purely bandwidth driven, not latency

Questions?

Feel free to email me at:
jlietz@nvidia.com