HPC+QC Tight Integration with CUDA-Q

1st ORNL Quantum Systems & Software Workshop
July 25, 2025

Justin Gage Lietz- Quantum Software Architecture Group
Alex McCaskey- Quantum Software Architecture Group

Background and Motivation

Where is HPC+QC Integration?

- Time Hierarchy:
  - Days: Sitting in the Queue
  - Seconds: Shot submission via reservation
  - Milliseconds: QEC Cycle time for Atoms/Ions
  - Microseconds: QEC Cycle time for Superconducting Qubits
  - Nanoseconds: Quantum gates, CPU/GPU instructions

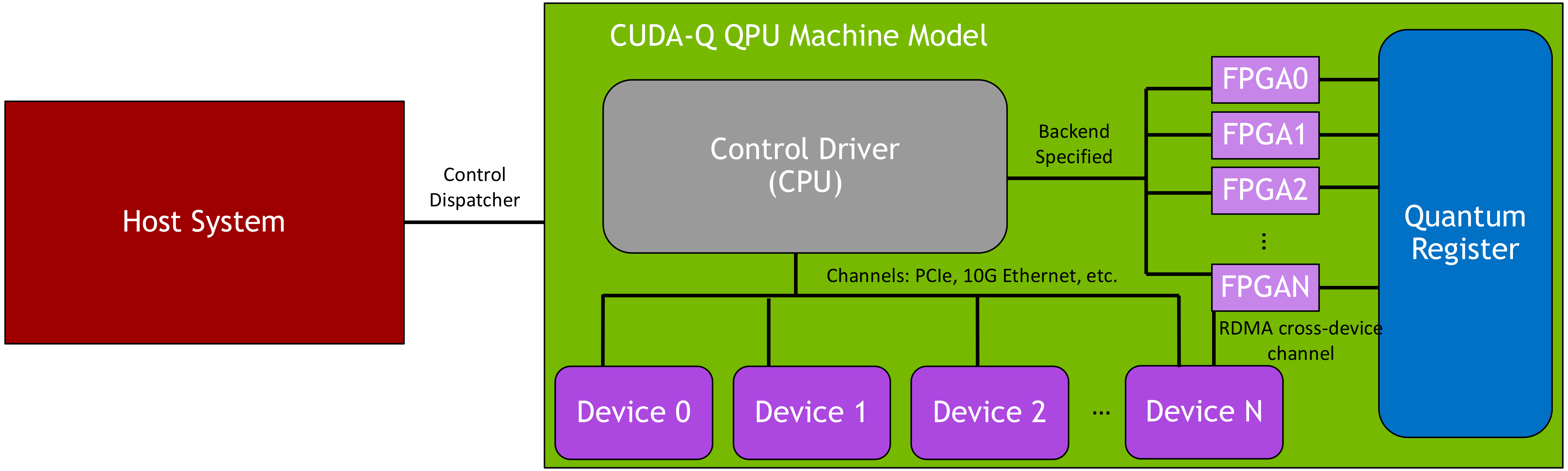
Background and Motivation

The heterogeneous QPU machine model

- CUDA-Q has always considered the machine model to be tightly integrated and heterogeneous (CPU, GPU, QPU)
  - **1.2. Each QPU is composed of a classical quantum control system (distributed FPGAs, GPUs, etc.) and a register of quantum bits (qubits) whose state evolves via signals from the classical control system.**
- To date, we've built the foundation (Language, MLIR, Runtime) and are now able to work on how we fully expose the true QPU heterogeneity to programmers
- The primary concepts not yet exposed to programmers are as follows:
  - Intra-QPU device function calling
  - Automated data marshalling across devices (e.g. fast PCIe data transfer)
  - These are concepts that the reference DGX-Q started to tackle, goal now is to generalize and formalize the software infrastructure
- First use cases for these concepts:
  - Real-time decoding
  - Calibration / control
  - RL / AI use cases
- Goal – enable infrastructure for heterogeneous quantum kernel libraries (quantum + low latency classical callbacks)
  - Small hardware agnostic QEC primitives for sending data and decoding
  - Can opt-in to access hardware specific expressivity when desired

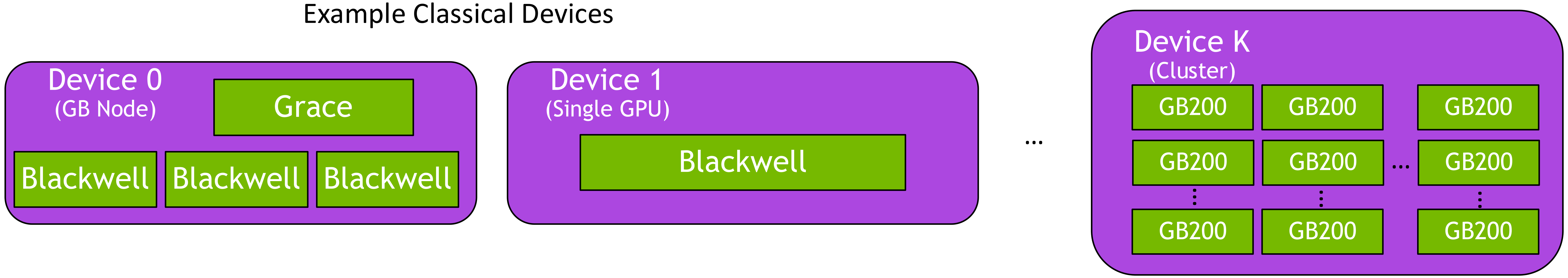
The CUDA-Q QPU Machine Model

Scaling quantum acceleration will require classical co-processing at low latencies



- Data transfer mechanisms (channels)
  - GPU RDMA – DOCA GPUNETIO
  - 10G Ethernet, TCP/UDP
  - Direct PCIe (CUDA)
  - NVLink, Infiniband
- Classical devices within a QPU assigned an ID
- Devices can be single GPU, CPU-GPU Superchips, Clusters

Example Classical Devices



Programmers require "in-kernel" (within quantum coherence times) function calling and data marshaling

The question then becomes – How do you program this?

Programming Model – Device Functions and Invocation

Programmers express intra-device function calling with generic CUDA-Q API and classical device kernels

- Each device is assigned a unique ID
- Programmers define ***classical device kernels***
  - Classical functions running on device
  - CPU or CUDA Device Functions
  - `__qpu__` kernels can invoke device kernels within the same logical QPU. Controller drives invocation.
  - Enables the development of intra-QPU device libraries
- Programmers are explicit about intra-device function calling via `cudaq::device_call`
  - Generic template function
  - Compilers can lower this IR node to data transfer / invocation sub-system
- Simple examples
  - Real-time decoding
  - Data generation from GPU feeding into quantum computation
  - Real-time calibration
  - Reinforcement-Learning or AI co-processing

```
void process_measurement(double result) {
  // Classical processing code
}

void process_error_syndrome(std::vector<bool> syndrome) {
  // library-based syndrome processing on device
}

__global__ void fastCUDAKernelForProcessing(int * in, int* out) {
  ...
}
```

```
// Target device with device_id. No Block or Grid Size, this is a CPU call
template <typename ExternalDeviceCall, typename... Args>
auto device_call(std::size_t device_id, ExternalDeviceCall&& callee, Args&&... args)
    -> decltype(callee(std::forward<Args>(args)...));

// Target default (0th) device.No Block or Grid Size, this is a CPU call
template <typename ExternalDeviceCall, typename... Args>
auto device_call(ExternalDeviceCall&& callee, Args&&... args)
    -> decltype(callee(std::forward<Args>(args)...));

// Target direct CUDA kernel invocation on device 0.
template <std::size_t blockSize, std::size_t gridSize,
          typename ExternalDeviceCall, typename... Args>
auto device_call(ExternalDeviceCall &&callee, Args &&...args)
    -> decltype(callee(std::forward<Args>(args)...));

// Target direct CUDA kernel invocation on specified device.
template <std::size_t blockSize, std::size_t gridSize,
          typename ExternalDeviceCall, typename... Args>
auto device_call(std::size_t device_id, ExternalDeviceCall &&callee, Args &&...args)
    -> decltype(callee(std::forward<Args>(args)...));

```

Programming Model – Device Functions and Invocation

Measurement data can feed into processing on distributed classical devices

Hello World Usage

- Device call kicks off data-movement across communication channel. Data allocated / sent to device.
- Callback is invoked on ”remote” data pointer.
- Result is returned over communication channel and fed into the rest of the computation.

```
__qpu__ void quantum_error_correction(cudaq::qvector<>& qubits) {
  // Measure stabilizers
  auto results = mz(qubits);

  // Process syndrome on GPU 0
  auto correction = cudaq::device_call(0, process_syndrome, results);

  // Apply corrections based on syndrome
  apply_corrections(qubits, correction);
}
```

Programming Model – Device Functions and Invocation

Real-time decoding workflows enabled via intra-device function calling

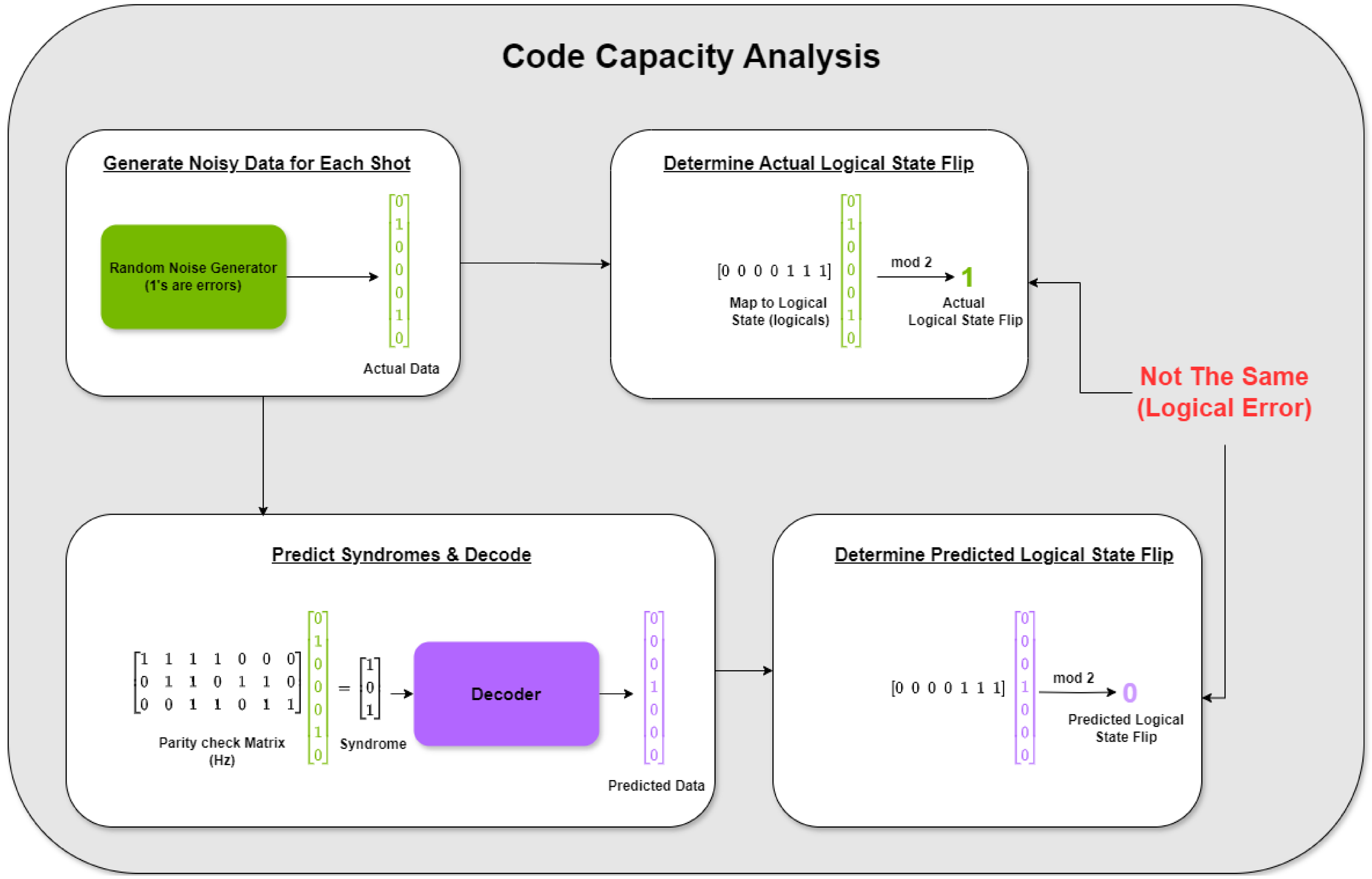
More Advanced Example

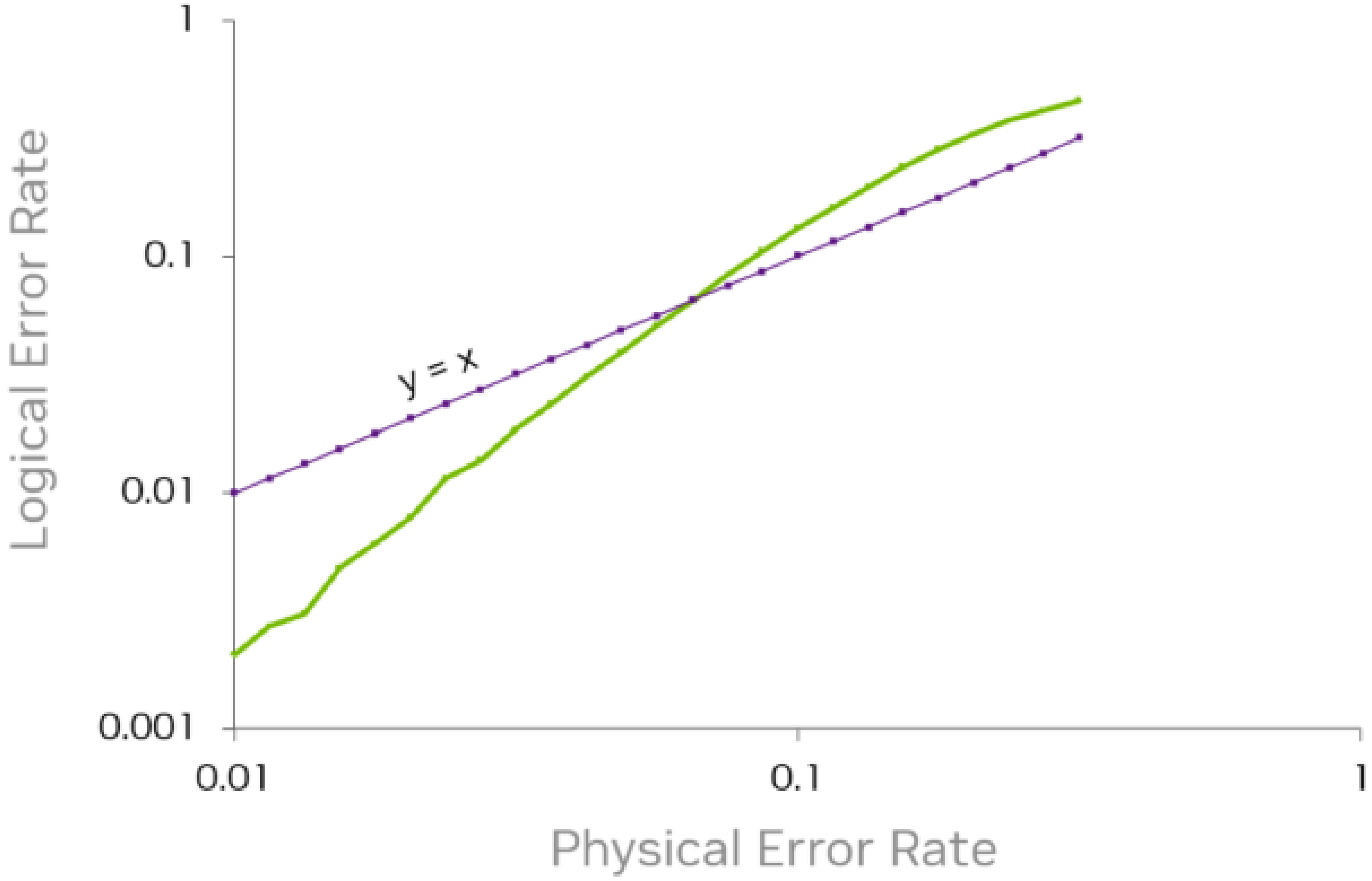
- Real-time decoding, memory circuit experiment
- Device function for enqueuing syndrome data per round
- Device function for running GPU-accelerated decoding and returning the corrections

```
__qpu__ void memory_circuit_and_decode_mz(
    const qec::code::stabilizer_round &stabilizer_round,
    const qec::code::one_qubit_encoding &statePrep, std::size_t numData,
    std::size_t numAncx, std::size_t numAncz, std::size_t numRounds,a
    const std::vector<std::size_t> &x_stabilizers,
    const std::vector<std::size_t> &z_stabilizers) {
  // Allocate a logical patch of qubits
  qec::patch logical(numData, numAncx, numAncz);

  // Prepare the desired state, provided as pure-device kernel
  statePrep(logical);

  for (std::size_t round = 0; round < numRounds; round++) {
    // Run the stabilizer round, generate the syndrome measurements
    auto syndrome = stabilizer_round(logical, x_stabilizers, z_stabilizers);
    // Tell device 0 to enqueue a new syndrome as an encoded integer
    cudaq::device_call(qec::enqueue_syndrome, cudaq::to_integer(syndrome));
  }

  // Decode on device 0
  auto corrections_int = cudaq::device_call(qec::decode_i64);

  // Apply corrections
  for (std::size_t i = 0; i < numAncx; i++)
    if ((corrections_int & 1 << i) != 0) // ith bit is set
      x(logical.data[i]);
  for (std::size_t i = numAncx; i < numAncz; i++)
    if ((corrections_int & 1 << i) != 0) // ith bit is set
      z(logical.data[i]);

  return;
}
```

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





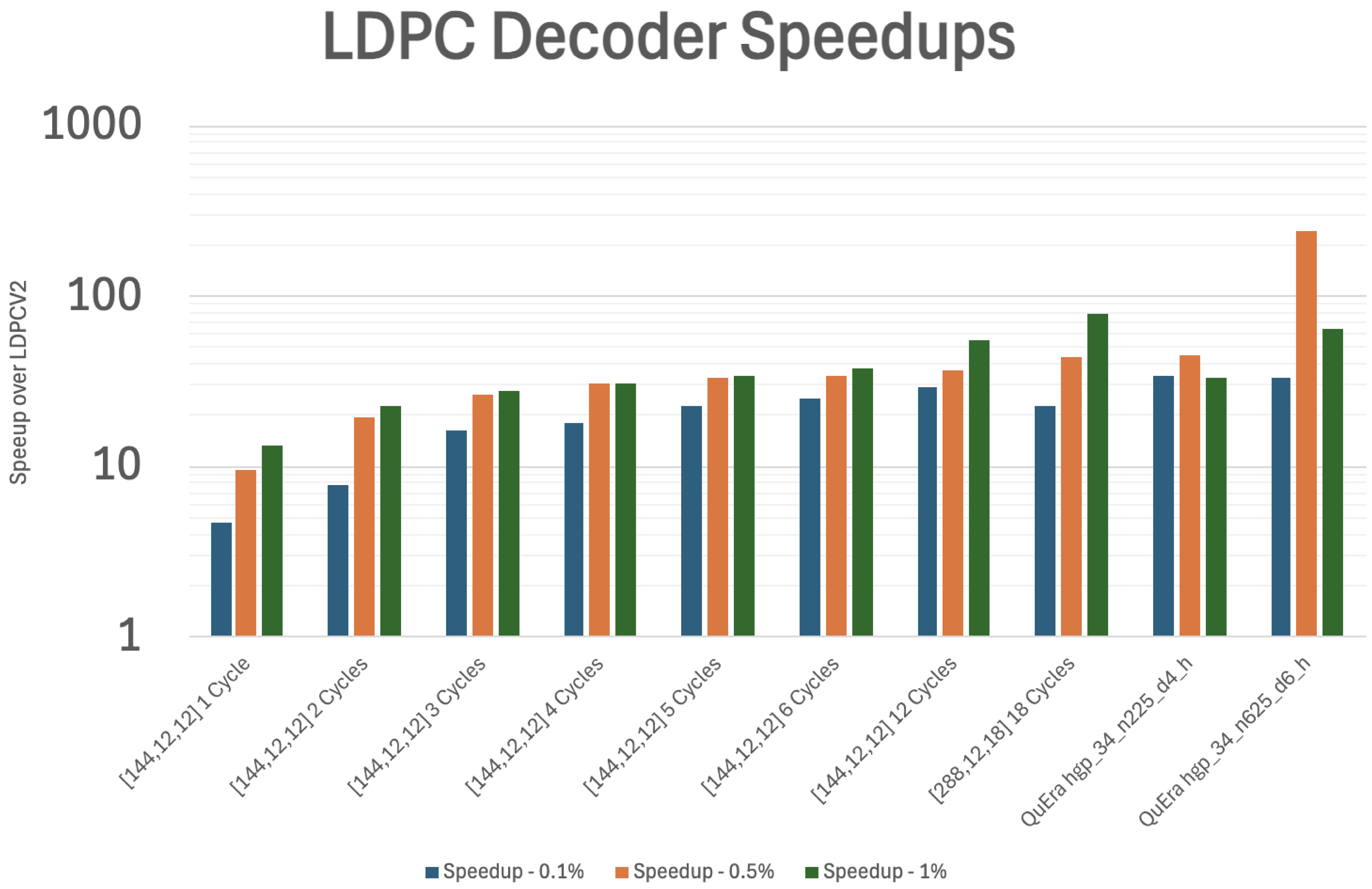
cudaq::qec::code

```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



Thanks!

Justin Gage Lietz- jlietz@nvidia.com