Programming Future Heterogenous Quantum-Classical Supercomputing Architectures

Alex McCaskey, Manager Quantum Architecture and Algorithms

BNL HPC Seminar, January 31, 2024

Agenda

- NVIDIA, HPC, and Quantum Computing
- Why Start at C++? Why build on MLIR / LLVM?
- CUDA Quantum in Action
- CUDA Quantum Language Deep Dive
- CUDA Quantum Compiler Stack

NVIDIA, HPC, and Quantum Computing

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 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
  - Hybrid workflows
- 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

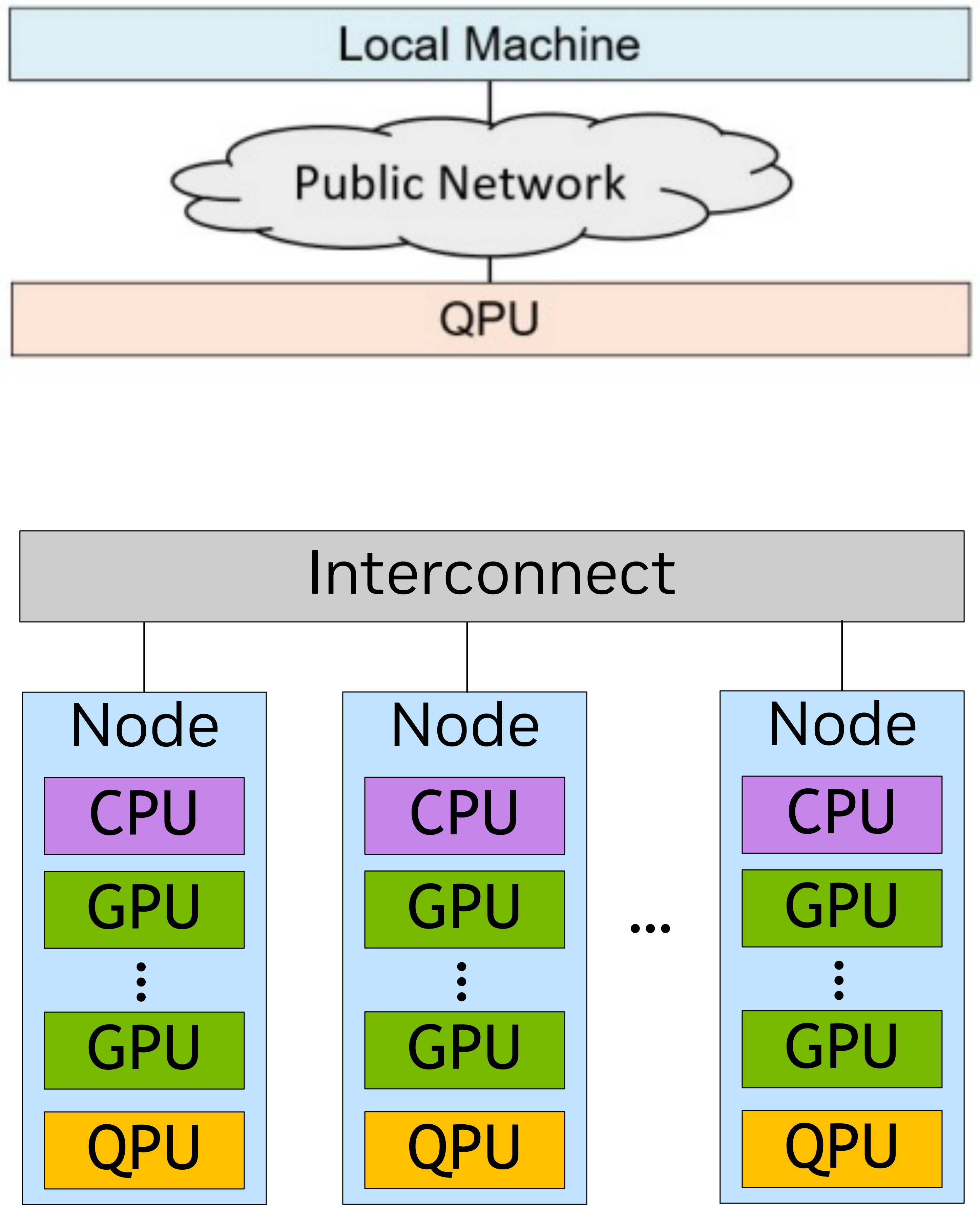
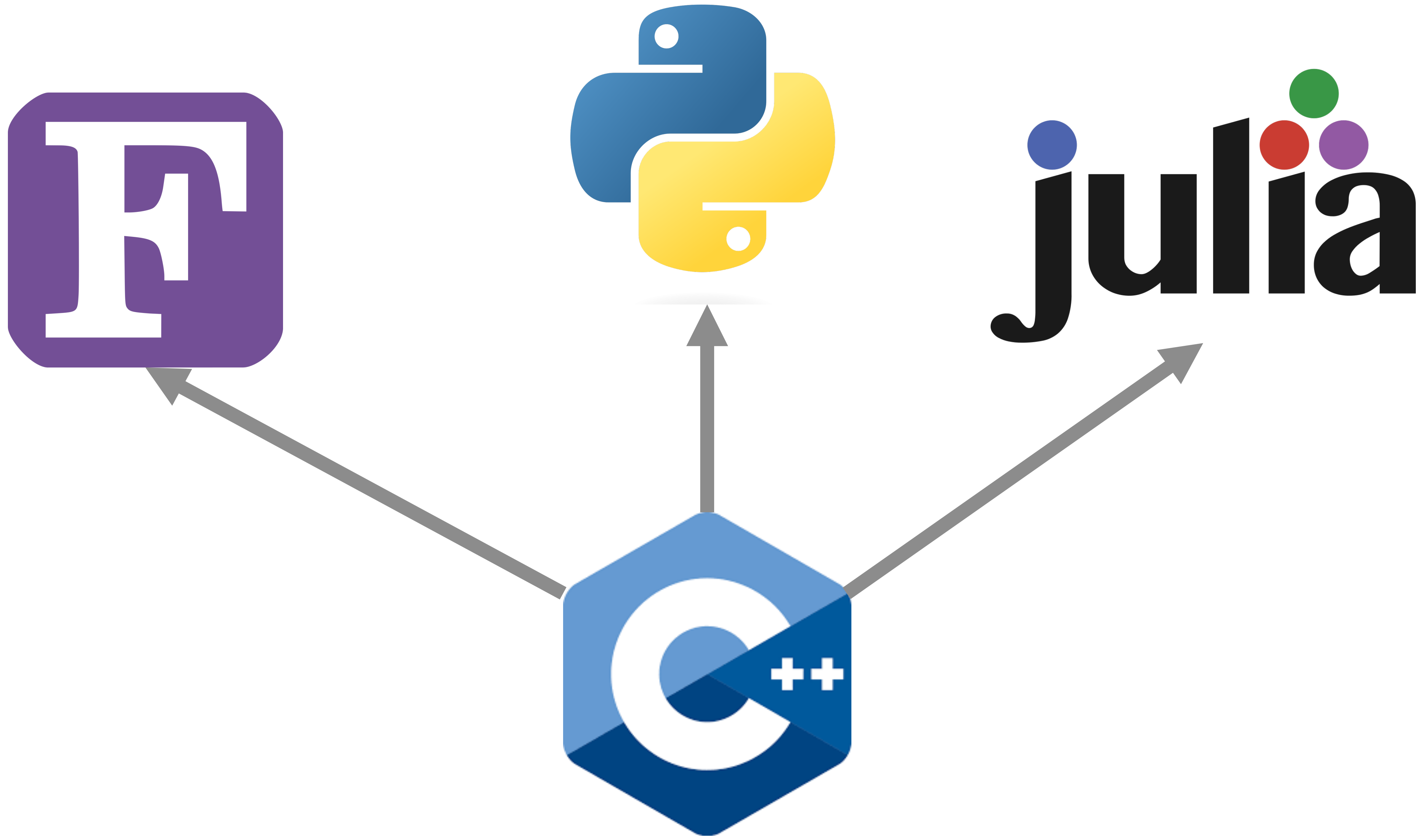


Figure adapted from:
Quantum Computers for High-Performance Computing.
Humble, McCaskey, Lyakh, Gowrishankar, Frisch, Monz.
IEEE Micro Sept 2021. 10.1109/MM.2021.3099140

Requirements for Programming the Hybrid Quantum-Classical Node

What can we learn from experience in the purely classical programming space?

- 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 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)



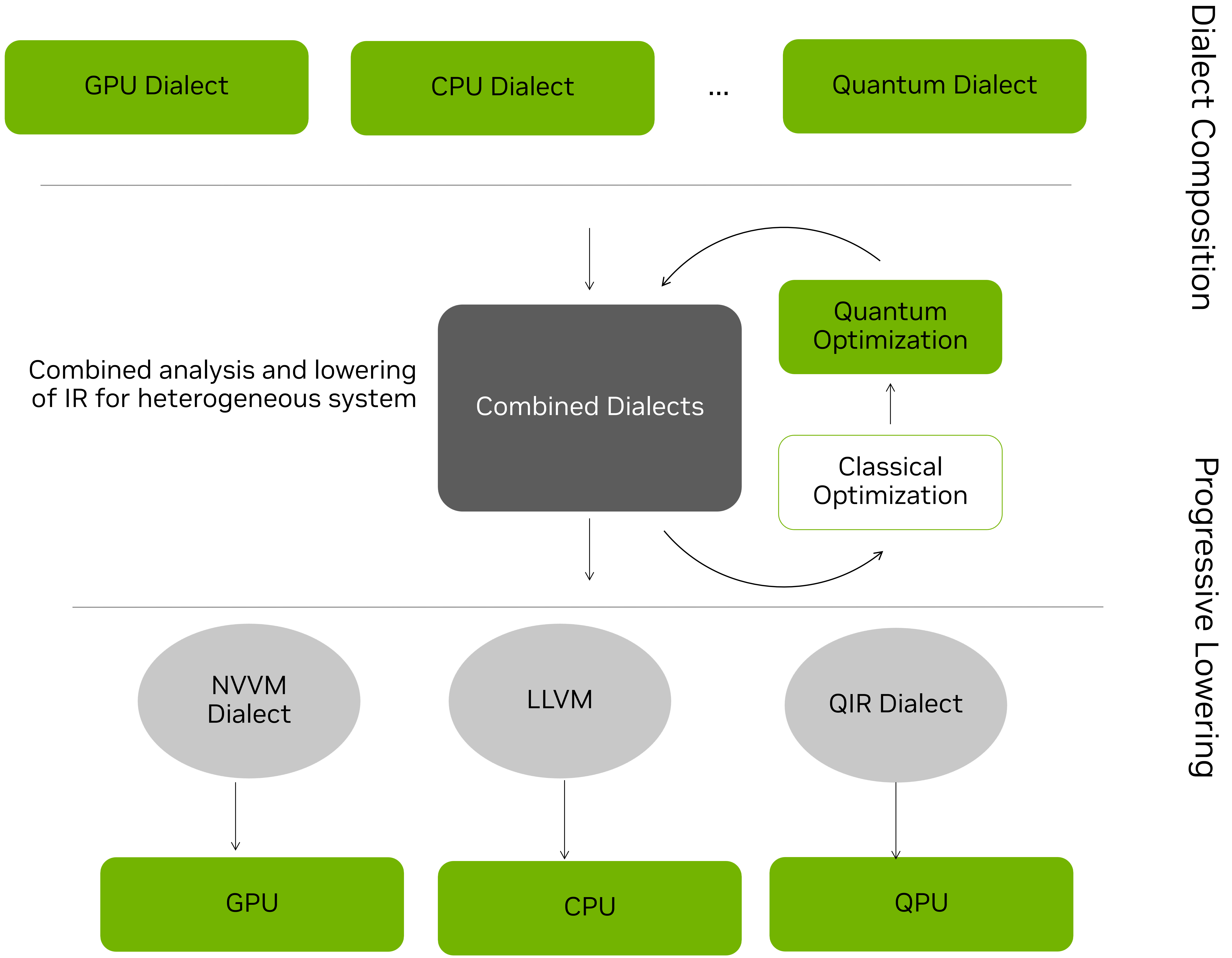
```
// Kernel functions enable clean separation of
// host and device code

__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



GPU Dialect

CPU Dialect

…

Quantum Dialect

Dialect Composition

Combined analysis and lowering of IR for heterogeneous system

Combined Dialects

Quantum Optimization

Classical Optimization

Progressive Lowering

NVVM Dialect

LLVM

QIR Dialect

GPU

CPU

QPU

CUDA Quantum

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

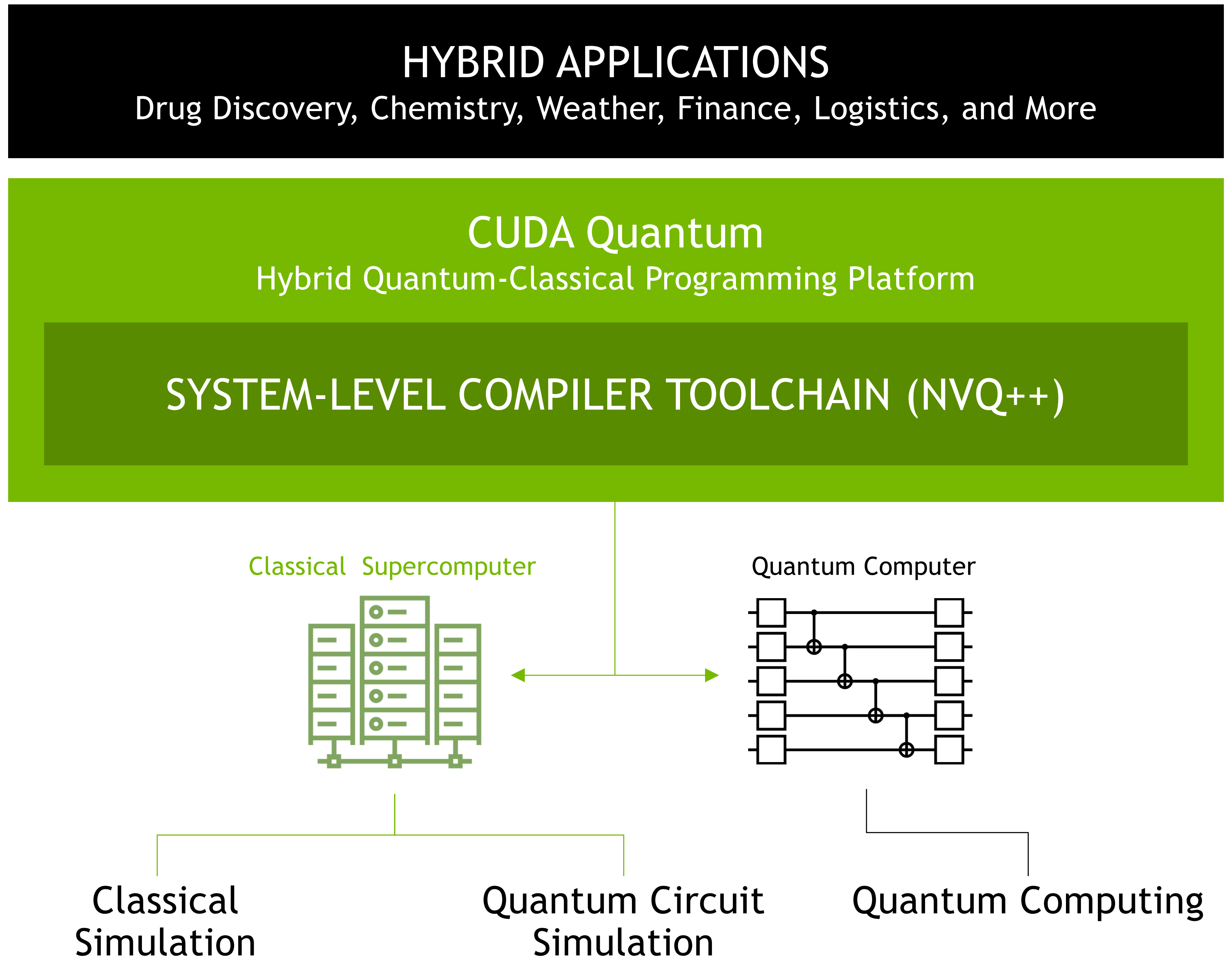
Introducing CUDA Quantum

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
- ***Retargetable*** - seamless transition from simulation to physical QPU

```
auto ansatz = [](std::vector<double> thetas) __qpu__ {
  cudaq::qreg<3> q;
  x(q[0]);
  ry(thetas[0], q[1]);
  ry(thetas[1], q[2]);
  x<cudaq::ctrl>(q[2], q[0]);
  x<cudaq::ctrl>(q[0], q[1]);
  ry(-thetas[0], q[1]);
  x<cudaq::ctrl>(q[0], q[1]);
  x<cudaq::ctrl>(q[1], q[0]);
};

cudaq::spin_op H = ...;
double energy = cudaq::observe(ansatz, H, {M_PI, M_PI_2});
```



HYBRID APPLICATIONS

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

CUDA Quantum

Hybrid Quantum-Classical Programming Platform

SYSTEM-LEVEL COMPILER TOOLCHAIN (NVQ++)

Classical Supercomputer

Quantum Computer

Classical Simulation

Quantum Circuit Simulation

Quantum Computing

IQM
atom computing
PASQAL
XANADU
QUANTINUUM
rigetti
ZAPATA
Microsoft
QUANTUM BRILLIANCE
OQC
QuEra Computing Inc.
agnostiq
QMware
ANYON

The CUDA Quantum Stack

Platform for unified quantum-classical accelerated computing

Frontends

C++

Kernel Expressions

JIT Kernel Expressions

Runtime

Python

(future) Kernel Expressions

JIT Kernel Expressions

Runtime

Compiler Platform

cudaq-quake

cudaq-opt

cudaq-translate

nvq++ driver

CUDA Quantum Intermediate Representation (MLIR)

Quake

(Quantum)

CC

(Classical CFG)

Func

(Kernels)

Math

(Standard Math)

Arith

(Constants)

LLVM

(Lowering Target)

Quantum Intermediate Representation (QIR, Profiles, LLVM IR)

libnvqir.so

Simulation (MGPU, MNMG, DM, TN)

Physical QPU (Quantinuum, IonQ, IQM, OQC)

CUDA Quantum in Action

- Dramatic performance improvements
- Hybrid QPU-GPU applications

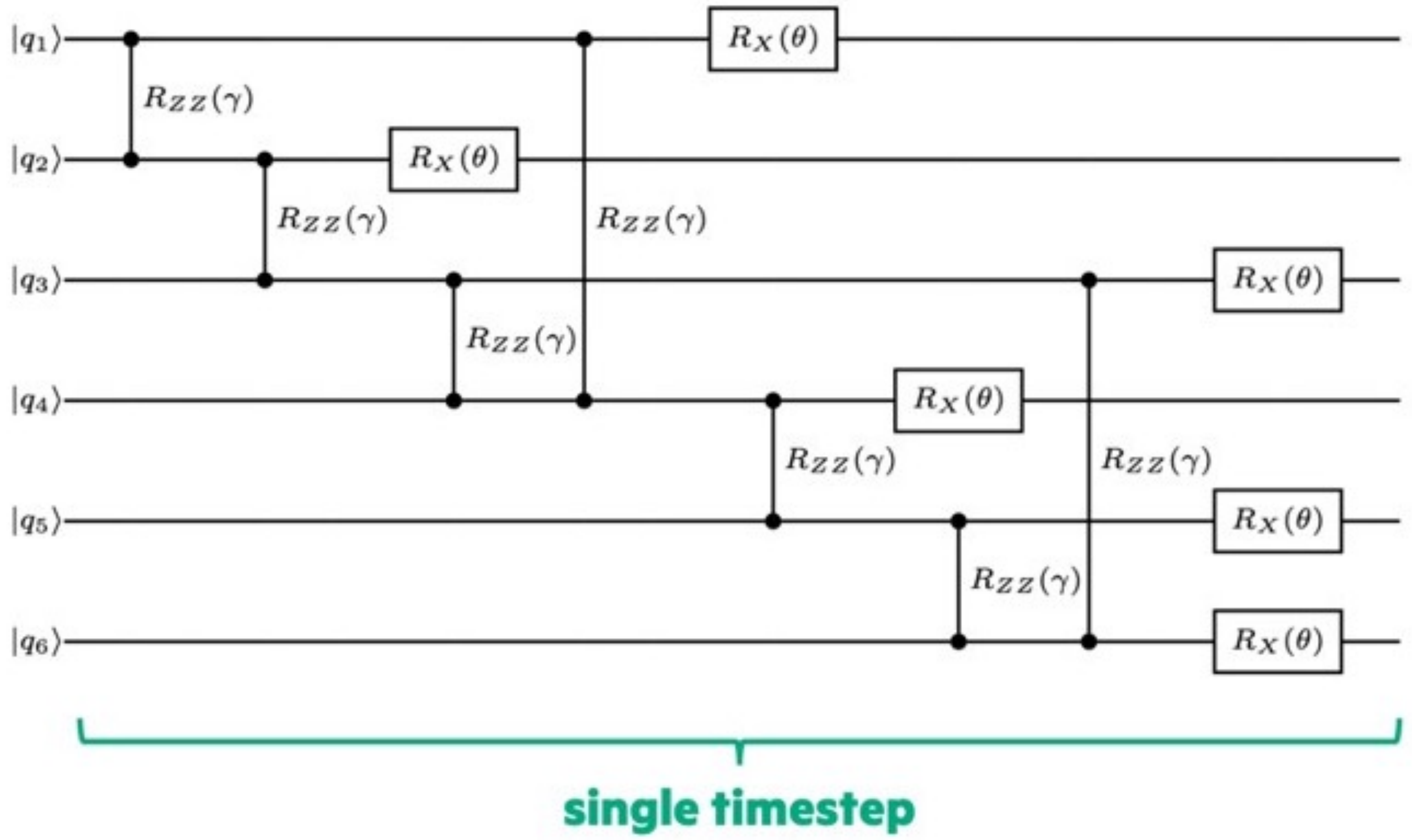
CUDA Quantum in Action

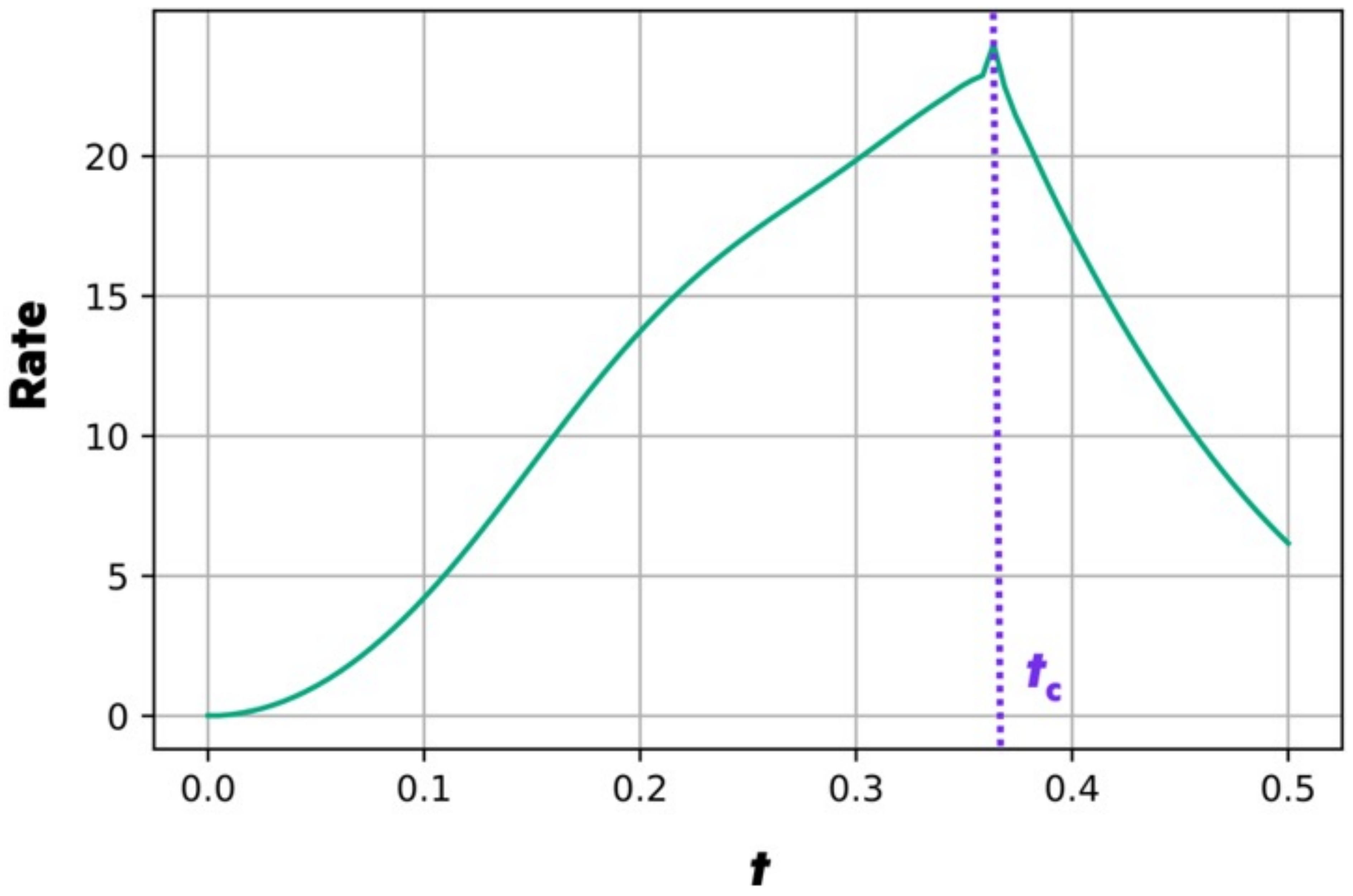
Speed-ups for time-evolution of the transverse field Ising model (TFIM)

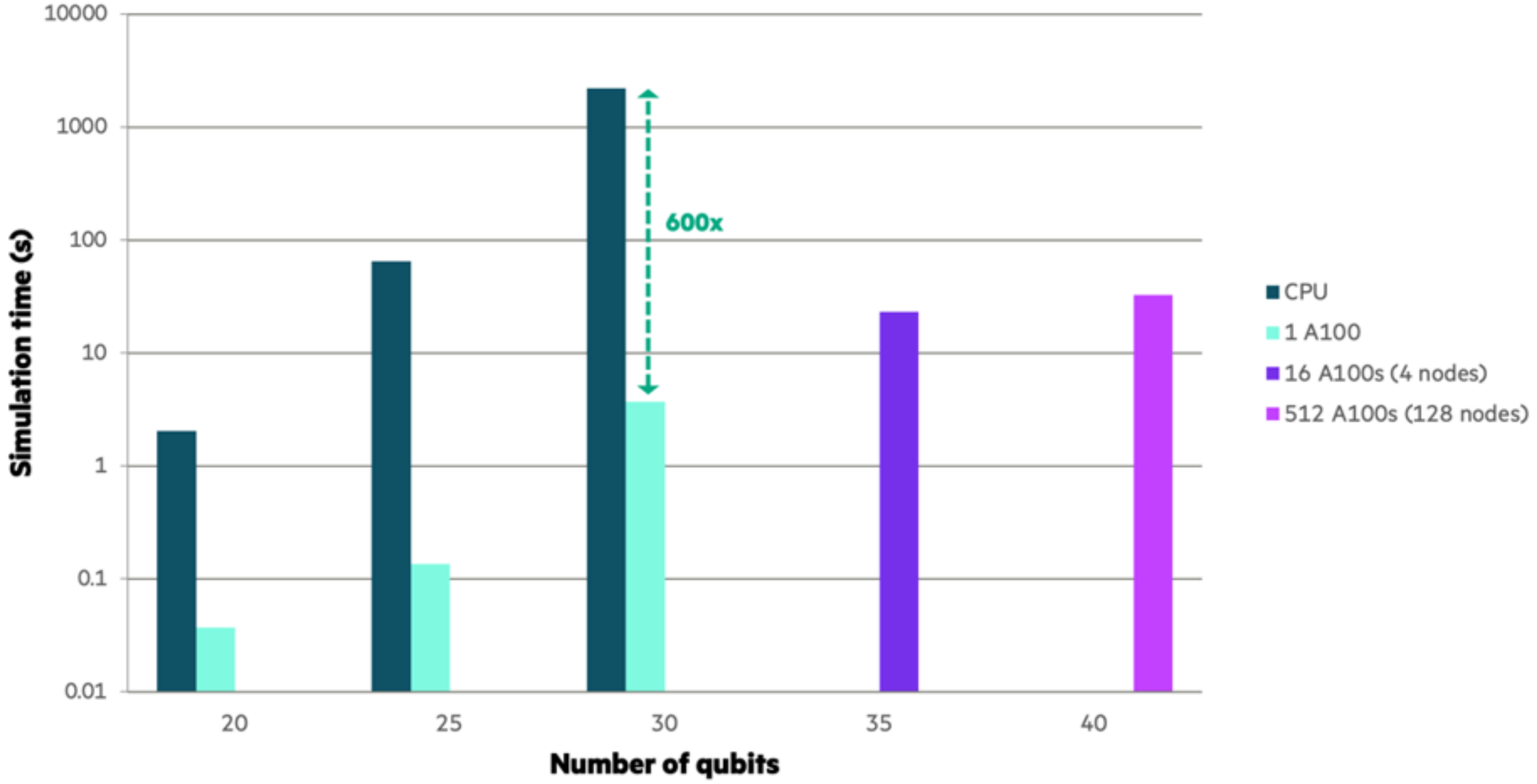
- Collaboration with Hewlett Packard Labs
- Study dynamical quantum phase transitions
  - Requires computation of overlap of initial state with time evolved state
- Leverage NVIDIA multi-node, multi-GPU simulation backend.
  - Distributed state-vector simulator
- 600x performance increase over multi-threaded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

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

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



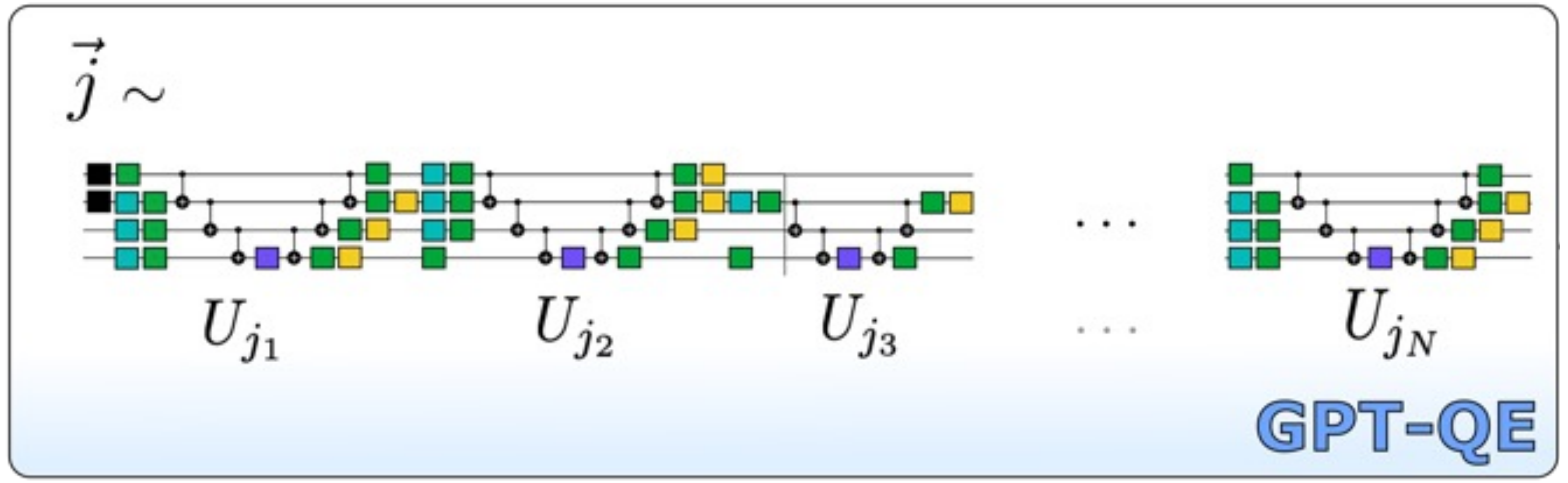


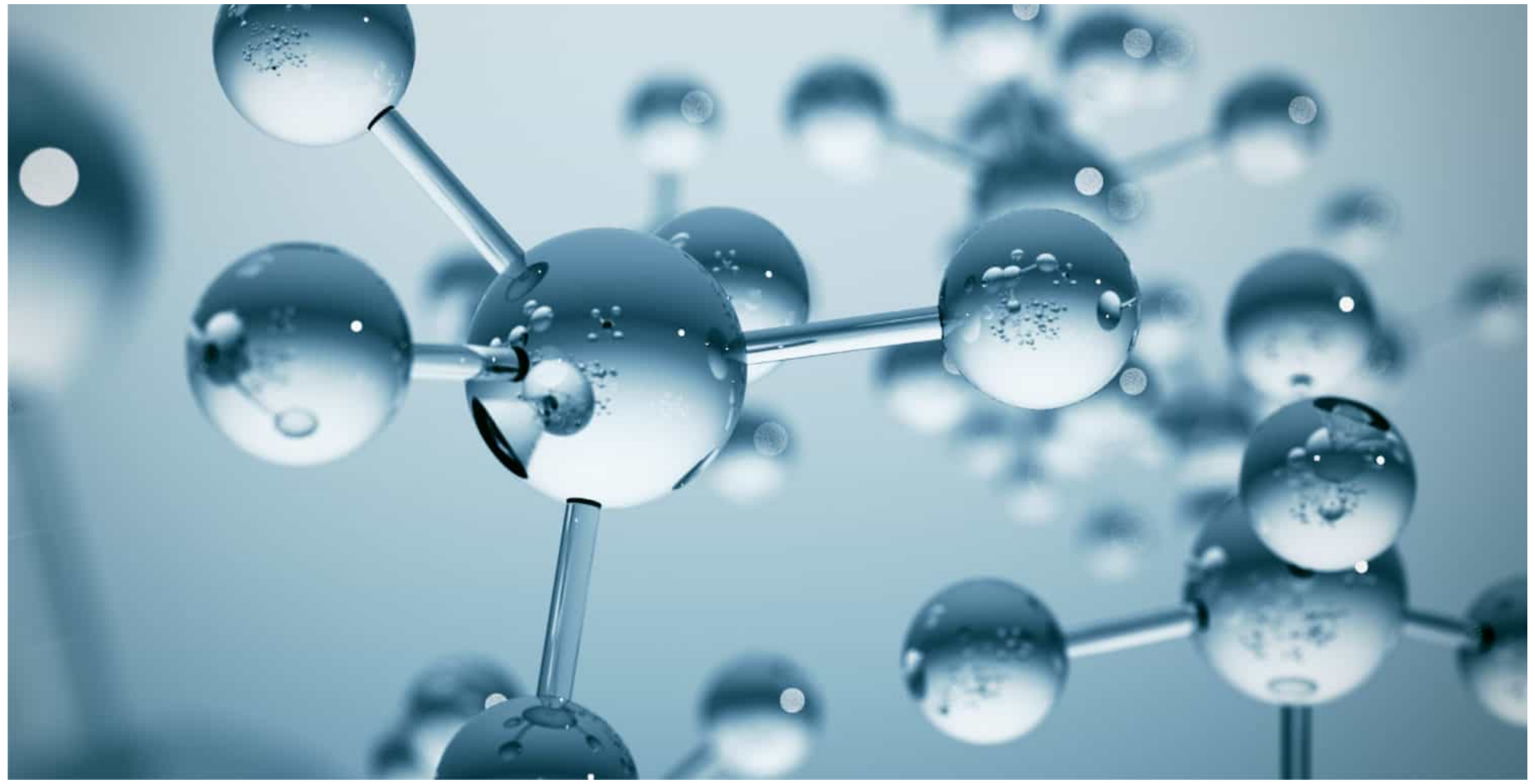


CUDA Quantum in Action

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

- 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 Quantum 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

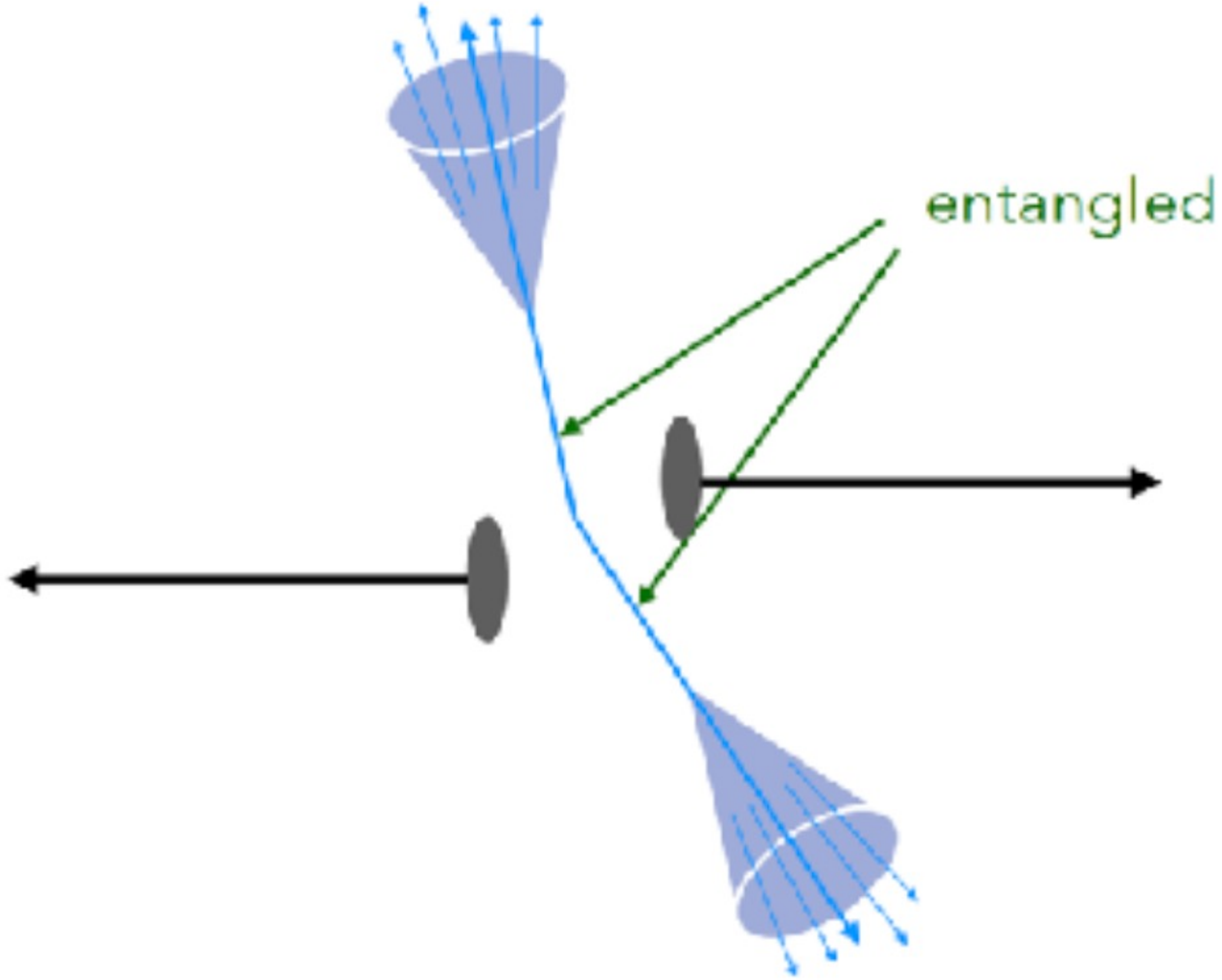
UNIVERSITY OF
TORONTO

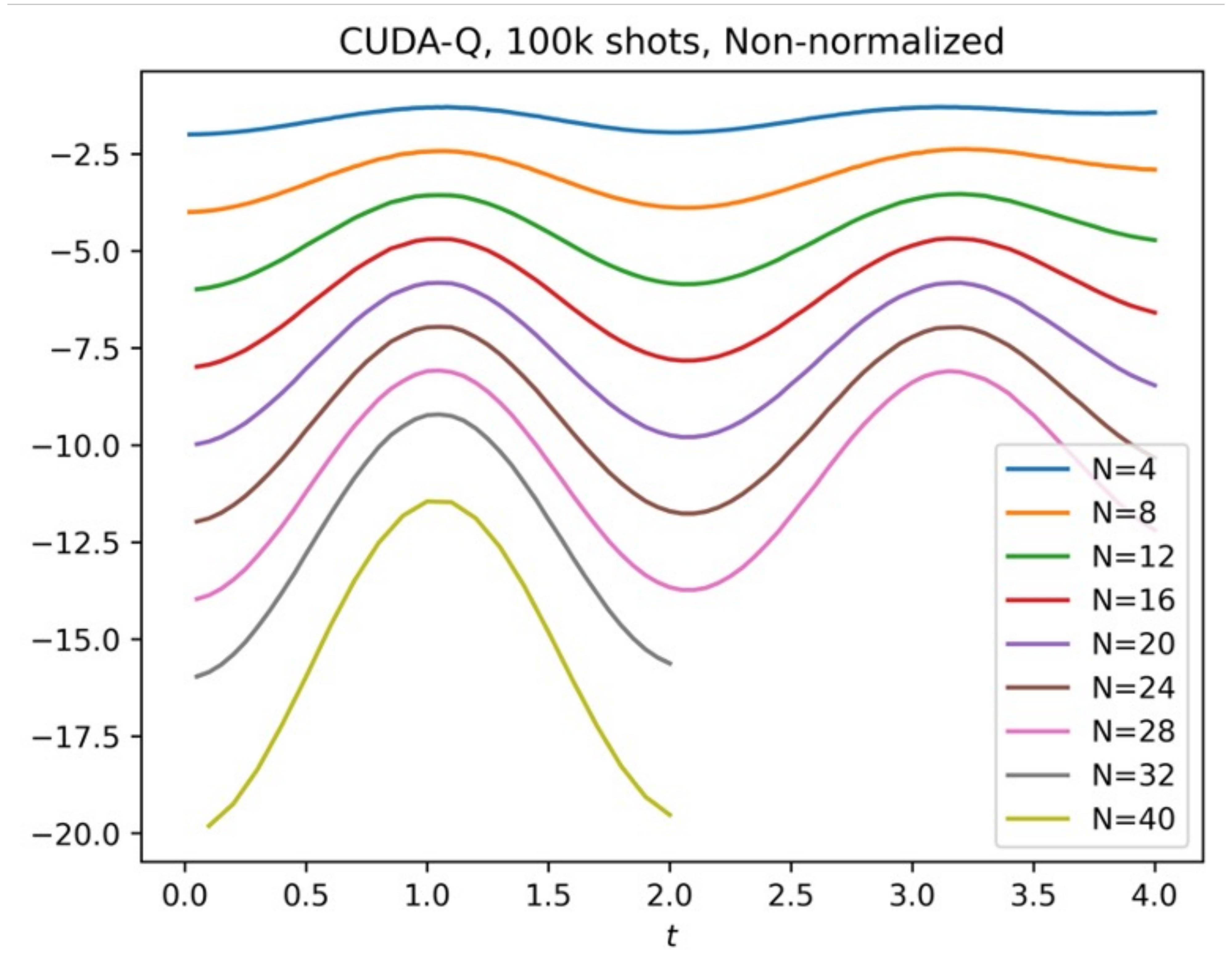
St. Jude Children’s®
Research Hospital
Finding cures. Saving children.

CUDA Quantum in Action

Speed-ups for quantum simulation in high-energy physics

- Quantum simulation of Schwinger Model with Jet
- Joint work with SBU and BNL
- 40 qubit run on 128 nodes (512 NVIDIA A100 GPUs) using NERSC Perlmutter
- Even for lower qubits, such as 20-qubits, 200x faster than other approaches
- Crucial for furthering of physics and material science research





Higher qubit runs enabled by CUDA Quantum help in improving accuracy in determining peak values

C++ Frontend
CUDA Quantum Kernel Expressions

Kernels, quantum memory, operations, and control flow

What is a CUDA Quantum Kernel?

Cleanly separate host code from quantum device code

- Any ***callable*** in the language – free function, typed callable, lambda
  - Must be annotated for compiler, `__qpu__`
- Types of kernels
  - Entry point and Pure-device
- Can take classical input
  - Pass by value
  - T and `std::vector<T>` such that `std::is_arithmetic_v<T> == true`

```
// Pure Device Kernel can take quantum input
// Cannot be called from host code
__qpu__ void pureDevice(cudaq::qubit& q, double angle) {
  x(q);
  ry(angle, q);
}

// Entry Point Kernel can be called from host
// code, can take classical input
__qpu__ void simpleEntry(double angle) {
  cudaq::qubit q;
  pureDevice(q, angle);
};

int main() { simpleEntry(M_PI_2); }
```

```
// Pure Device Kernel can take quantum input
// Cannot be called from host code
struct pureDevice {
  void operator()(cudaq::qubit& q, double angle) __qpu__ {
    x(q);
    ry(angle, q);
  }
};


// Entry Point Kernel can be called from host
// code, can only take classical input
__qpu__ void simpleEntry(double angle) {
  cudaq::qubit q;
  pureDevice{}(q, angle);
};

int main() { simpleEntry(M_PI_2); }
```

```
// Pure Device Kernel can take quantum input
// Cannot be called from host code
__qpu__ void pureDevice(cudaq::qubit& q, double angle) {
  x(q);
  ry(angle, q);
}

int main() {

  // Entry Point Kernel can be called from host
  // code, can take classical input
  auto simpleEntry = [](double angle) __qpu__ {
    cudaq::qubit q;
    pureDevice(angle, q);
  };

  simpleEntry(M_PI_2);
}
```

What is a CUDA Quantum Kernel?

Cleanly separate host code from quantum device code

- ***Composable*** – call in scope kernels, pass them as arguments
- Leverage C++ template metaprogramming
- Powerful mechanism for building ***generic application libraries***
- Used for algorithmic primitives (`sample`, `observe`)
- Define library code parameterized on oracles / state preparation steps specified by programmer

```
// Define a kernel that takes another kernel as input
// rely on C++ template deduction
struct simpleEntry {
  bool operator()(auto&& statePrep, double p) __qpu__ {
    cudaq::qubit q;
    statePrep(q, p);
    return mz(q);
  }
};

int main() {

  // Create a pure-device kernel lambda
  auto pureDevice =
    [](cudaq::qubit& q, double theta) __qpu__ {
      x(q);
      ry(theta, q);
    };

  // Pass the lambda as the statePrep argument
  auto bitResult = simpleEntry{}(pureDevice, M_PI_2);

}
```

Quantum Memory Allocation and Deallocation

Instantiate quantum registers and deallocate at scope exit

- Quantum memory allocation via standard C++ semantics
  - Owned memory vs non-owned
  - Dynamic allocation vs static allocation
- `qreg` and `qspan` (Specification change: qvector, qarray, qview)
  - `qreg<N>`, `qreg(N)`
- No-cloning enforced at compile-time
  - Copy and move constructors deleted
- General qudits are supported

```
// Define a kernel that programs on qutrits
struct kernelOnQudits {
  int operator()(double theta) __qpu__ {
    cudaq::qudit<3> q, r;
    plus_gate(q);
    plus_gate(r);
    beam_splitter(q, r, theta);
    return mz(r);
  }
};

// NOTE:
// using qubit = qudit<2>;
```

```
__qpu__ void kernelBad(cudaq::qubit q, double p) {
  ...
}

__qpu__ void kernelGood(cudaq::qubit& q, double p) {
  ...
}

__qpu__ void kernel(std::size_t N) {
  {
    cudaq::qubit q;
    // kernelBad(q, 2.2); // Compiler Error!
    kernelGood(q, 2.2);
    // qubit deallocated at scope exit
  }
  {
    cudaq::qreg<5> q; // array-like semantics
    // all qubits deallocated at scope exit
  }
  {
    cudaq::qreg q(N); // vector-like semantics
  }
  {
    cudaq::qreg q(4); // can extract sub-registers
    auto subReg = q.front(2); // is a qspan
  }
};

template<typename N>
struct staticCircuit { // Compile-time-known register size
  void operator()() __qpu__ {
    cudaq::qreg<N> q;
    ...
  }
};
```

Quantum Operations and Controlled / Adjoint Modifiers

Single qubit operations and compiler-synthesized controlled and adjoint operations

- Quantum operations are unique functions that take a qubit reference and optional floating point parameters.
- Operations can be modified with `cudaq::ctrl` or `cudaq::adj`
  - Multi-qubit operations are synthesized by the compiler
  - Control qubits are first `N-1` qubit arguments
  - Control qubits can be 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 kernel(std::size_t N) {
  {
    cudaq::qreg<3> q;
    // Toffoli
    x<cudaq::ctrl>(q[0], q[1], q[2]);
  }
  {
    cudaq::qreg q(N);
    // Toffoli
    cudaq::control(cnotKernel, {q[0]}, q[1], q[2]);
  }
  {
    cudaq::qreg q(2);
    h<cudaq::ctrl>(!q[0], q[1]); // negated control
  }
  {
    cudaq::qubit q, r;
    t<cudaq::adj>(q); // adjoint modifier
    // compiler synthesizes the adjoint
    cudaq::adjoint(toBeAdjointed, q, r);
  }
};
```

Control Flow Inherited from the Language

Control flow should feel natural

- Inherit control flow from the language being extended.
  - All loop constructs, continue, break, etc.
  - Conditional statements and branching
- Move away from runtime-level abstractions for control flow
  - Circuit builder patterns that expose `.c_if()` functions
- We've done this for C++, Python coming in March

```
__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);
}
```

```
__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;
}
```

```
# Where we're headed, not there yet ...

@cudaq.kernel
def kernelControlFlow():
  qubits = cudaq.qreg(4)
  for q in qubits:
    x(q)
  if mz(qubits[0]):
    ... Do something ...
```

CUDA Quantum Compiler Platform

- MLIR Dialects
- Tools

The CUDA Quantum Compiler Stack

Platform for unified quantum-classical accelerated computing

Compiler Platform

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

CUDA Quantum Intermediate Representation (MLIR)

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

Quantum Intermediate Representation (QIR, Profiles, LLVM IR)

CUDA Quantum provides a collection of tools that enables the compilation of Kernel 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.

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

$ nvq++ -o gpuAccelerated.x example.cpp --target nvidia
$ ./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 quantinuumH1.x example.cpp --target quantinuum
$ ./quantinuumH1.x
```

The CUDA Quantum Compiler Stack

Platform for unified quantum-classical accelerated computing

Compiler Platform

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

CUDA Quantum Intermediate Representation (MLIR)

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

Quantum Intermediate Representation (QIR, Profiles, LLVM IR)

- CC Dialect (Classical Computing)
  - Model C++ types and behavior
  - Control flow
  - `std::vector<T>`, callables
  - 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++

Type
cc.ptr<T>
cc.struct<T...>
cc.array<T>
cc.callable
cc.stdvec<T>

Operation
cc.scope
cc.loop
cc.if
cc.condition
cc.return, cc.break, cc.continue
cc.load, cc.store

The CUDA Quantum Compiler Stack

Platform for unified quantum-classical accelerated computing

Compiler Platform

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

CUDA Quantum Intermediate Representation (MLIR)

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

Quantum Intermediate Representation (QIR, Profiles, LLVM IR)

- CC Dialect (Classical Computing)
  - Model C++ types and behavior
  - Control flow
  - `std::vector<T>`, callables
  - 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++

```
%0 = cc.alloca i32
cc.store %c0_i32, %0 : !cc.ptr<i32>
%1 = cc.alloca i32
cc.store %c0_i32, %1 : !cc.ptr<i32>
cc.loop while {
  %2 = cc.load %1 : !cc.ptr<i32>
  %3 = arith.cmpi slt, %2, %c5_i32 : i32
  cc.condition %3
} do {
  %2 = cc.load %1 : !cc.ptr<i32>
  %3 = cc.load %0 : !cc.ptr<i32>
  %4 = arith.addi %3, %2 : i32
  cc.store %4, %0 : !cc.ptr<i32>
  cc.continue
} step {
  %2 = cc.load %1 : !cc.ptr<i32>
  %3 = arith.addi %2, %c1_i32 : i32
  cc.store %3, %1 : !cc.ptr<i32>
}
```

```
%false = arith.constant false
%true = arith.constant true
%0 = cc.alloca i1
cc.store %true, %0 : !cc.ptr<i1>
%1 = cc.load %0 : !cc.ptr<i1>
cc.if(%1) {
  cc.store %false, %0 : !cc.ptr<i1>
}
```

```
func.func @foo(%arg0: !cc.stdvec<i32>) {
  %0 = cc.stdvec_size %arg0 :
              (!cc.stdvec<i32>) -> i64
  ...
}
```

The CUDA Quantum Compiler Stack

Platform for unified quantum-classical accelerated computing

Compiler Platform

cudaq-quake

cudaq-opt

cudaq-translate

nvq++ driver

CUDA Quantum Intermediate Representation (MLIR)

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

Quantum Intermediate Representation (QIR, Profiles, LLVM IR)

```
func.func foo() attributes {"cudaq-entrypoint", "cudaq-kernel"} {
    %0 = quake.alloca !quake.ref
    quake.x %0 : (!quake.ref) -> ()
    return
  }
```

```
llvm.func foo() attributes {"cudaq-entrypoint", "cudaq-kernel"} {
    %0 = llvm.call @__quantum__rt__qubit_allocate() : () -> !llvm.ptr<struct<"Qubit", opaque>>
    llvm.call @__quantum__qis__x(%0) : (!llvm.ptr<struct<"Qubit", opaque>>) -> ()
    llvm.return
  }
```

```
define void @__nvqpp__mlirgen__function_test._Z4testv() local_unnamed_addr {
  %1 = tail call %Array* @__quantum__rt__qubit_allocate_array(i64 1)
  %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
  tail call void @__quantum__qis__x(%Qubit* %4)
  tail call void @__quantum__rt__qubit_release_array(%Array* %1)
  ret void
}
```

- 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 Quantum Compiler Stack

Platform for unified quantum-classical accelerated computing

Compiler Platform

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

CUDA Quantum Intermediate Representation (MLIR)

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

Quantum Intermediate Representation (QIR, Profiles, LLVM IR)

- Quake (Quantum Kernel Execution) Dialect
  - Model quantum types and operations
- Quake can be in one of 2 forms:
  - Memory semantic model
  - Value semantic model
  - MemToReg Pass transforms Memory to Value
  - RegToMem Pass transforms Value to Memory
- Optimizations and Transformations may be better suited for either of these forms

Type
quake.wire
quake.ref
quake.veq<N>

Operation
quake.alloca
quake.extract_ref
quake.apply
quake.{mx,my,mz}
quake.{h,x,y,z,rx,ry,rz,t,s,...}
quake.dealloc

The CUDA Quantum Compiler Stack

Platform for unified quantum-classical accelerated computing

Compiler Platform

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

CUDA Quantum Intermediate Representation (MLIR)

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

Quantum Intermediate Representation (QIR, Profiles, LLVM IR)

Quake Memory Semantic Model

```
func.func @__nvqpp__mlirgen__GHZ() {
  %0 = quake.alloca !quake.veq<4>
  %1 = quake.extract_ref %0[0] : (!quake.veq<4>) -> !quake.ref
  quake.h %1 : (!quake.ref) -> ()
  %2 = quake.extract_ref %0[0] : (!quake.veq<4>) -> !quake.ref
  %3 = quake.extract_ref %0[1] : (!quake.veq<4>) -> !quake.ref
  quake.x [%2] %3 : (!quake.ref, !quake.ref) -> ()
  %4 = quake.extract_ref %0[1] : (!quake.veq<4>) -> !quake.ref
  %5 = quake.extract_ref %0[2] : (!quake.veq<4>) -> !quake.ref
  quake.x [%4] %5 : (!quake.ref, !quake.ref) -> ()
  %6 = quake.extract_ref %0[2] : (!quake.veq<4>) -> !quake.ref
  %7 = quake.extract_ref %0[3] : (!quake.veq<4>) -> !quake.ref
  quake.x [%6] %7 : (!quake.ref, !quake.ref) -> ()
  return
}
```

Quake Value Semantic Model

```
cudaq-opt --canonicalize --memtoreg
```

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

The CUDA Quantum Compiler Stack

Platform for unified quantum-classical accelerated computing

Compiler Platform

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

CUDA Quantum Intermediate Representation (MLIR)

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

Quantum Intermediate Representation (QIR, Profiles, LLVM IR)

- 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
```

```
module {
  func.func @__nvqpp__mlirgen__function_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 Quantum Compiler Stack

Platform for unified quantum-classical accelerated computing

Compiler Platform

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

CUDA Quantum Intermediate Representation (MLIR)

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

Quantum Intermediate Representation (QIR, Profiles, LLVM IR)

- 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 Quantum Compiler Stack

Platform for unified quantum-classical accelerated computing

Compiler Platform

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

CUDA Quantum Intermediate Representation (MLIR)

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

Quantum Intermediate Representation (QIR, Profiles, LLVM IR)

- 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
```

```
source_filename = "LLVMDialectModule"
target datalayout = "e-m:e-p270:32:32-p271:32:32-p272:64:64-i64:64-f80:128-n8:16:32:64"
target triple = "x86_64-unknown-linux-gnu"

%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 Quantum Compiler Stack

Platform for unified quantum-classical accelerated computing

Compiler Platform

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

CUDA Quantum Intermediate Representation (MLIR)

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

Quantum Intermediate Representation (QIR, Profiles, LLVM IR)

- 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
```

```
// Code generated by NVIDIA's nvq++ compiler
OPENQASM 2.0;

include "qelib1.inc";


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

The CUDA Quantum Compiler Stack

Platform for unified quantum-classical accelerated computing

Compiler Platform

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

CUDA Quantum Intermediate Representation (MLIR)

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

Quantum Intermediate Representation (QIR, Profiles, LLVM IR)

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

Transform CUDAQ Kernels via clang::RecursiveASTVisitor subtype

CUDA Quantum Runtime

- JIT Compiler / Python
- The Platform and Asynchronicity
- Algorithmic Primitives

JIT Compilation and Execution and its utility to Python

Expose the compiler to the programmer and ultimately to Python

- With all the tooling just presented, we should enable one to build up the MLIR representation programmatically and at runtime.
- Enable first exposure of CUDA Quantum to Python
  - Ultimately, we want the language embedded in Python
  - Easy first pass to bind a ***builder pattern***
  - Drawbacks – conciseness and code bloat
- **`kernel_builder<Args...>`** type enables one to build Quake programmatically
  - Is a callable object, thereby fitting the CUDA Quantum kernel definition
  - Leverage MLIR **`ExecutionEngine`** for JIT compilation, extraction of a function pointer to invoke
  - Can define kernels with specific signature

```python
import cudaq

# Build your CUDA Quantum kernel
kernel = cudaq.make_kernel()
qubits = kernel.qalloc(2)
kernel.h(qubits[0])
kernel.cx(qubits[0], qubits[1])

# Can see the Quake code
print(kernel)

# Can execute it
kernel()
```

```cpp
__qpu__ std::vector<bool> bellAheadOfTime() {
  cudaq::qreg q(2);
  h(q[0]);
  x<cudaq::ctrl>(q[0], q[1]);
  return mz(q);
}
// See the Quake code for bellAheadOfTime with
// cudaq-quake thisFile.cpp

int main() {

  // We could have instead, built this programmatically
  {
    auto bellRuntime = cudaq::make_kernel();
    auto qubits = bellRuntime.qalloc(2);
    bellRuntime.h(qubits[0]);
    bellRuntime.x<cudaq::ctrl>(qubits[0], qubits[1]);
    bellRuntime.mz(qubits);
    std::cout << “Quake Code:\n” << kernel << “\n”;
    // The kernel you build is callable
    bellRuntime();
  }
  {
    // Kernels can be parameterized
    auto [kernelWithArg, arg] = cudaq::make_kernel<double>();
    auto qubits = kernelWithArg.qalloc(2);
    kernelWithArg.x(qubits[0]);
    kernelWithArg.ry(arg, qubits[1]);
    kernelWithArg.x<cudaq::ctrl>(qubits[1], qubits[0]);

    // The kernel you build is callable
    kerneWithArg(M_PI_2);
  }
}
```

CUDA Quantum Platform and Asynchronous Execution

Expose the underlying system architecture to the programmer

- The system architecture model considers access to multiple quantum accelerators
- CUDA Quantum provides programmatic access to this configuration via the **`quantum_platform`**
- CUDA Quantum and cuQuantum expose 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

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

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

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

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

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

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

Node 0, MPI Rank 0

vQPU (cuQuantum)

A100

vQPU (cuQuantum)

A100

…

vQPU (cuQuantum)

A100

Node 1, MPI Rank 1

vQPU (cuQuantum)

A100

vQPU (cuQuantum)

A100

…

vQPU (cuQuantum)

A100

CUDA Quantum Generic Algorithmic Primitives

cudaq namespace functions that are generic on the CUDAQ kernel expressions

- CUDA Quantum defines a generic function for sampling
  - Provide a CUDA Quantum 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
- CUDA Quantum 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 CUDA Quantum 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 Algorithmic Primitives

cudaq namespace functions that are generic on the CUDAQ kernel expression

- CUDA Quantum 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;
}
```

Conclusion and Looking Forward

Areas of collaboration

- CUDA Quantum is a system-level programming model and compilation platform geared toward enabling quantum acceleration to existing GPU Supercomputing architectures
- NVQ++ orchestrates a collection of modular tools enabling complex quantum-classical compilation workflows

NVIDIA