Building an LLVM-based Compiler Toolchain for Distributed Quantum Computing

Vyacheslav Levytskyy

Outline

- About quantum computing
- Know your hardware
- Distributed quantum computing
  - why the bigger picture matters for a compiler toolchain
- Design principles
  - compiler for quantum computing of spin-photon modality
- Photonic View on the Compiler Stack
  - LLVM/MLIR in quantum computing
  - virtual ISA
  - technical challenges
- The language of pulses
- Non-LLVM frontends: Q#, Python DSL
  - QIR
- Maintenance

The race of quantum modalities

Quantum computer is a heterogeneous system (distributed – when scaled into quantum networks or multi-QPU setups)

- to perform quantum computations, we need to translate a quantum algorithm to instructions for hardware tools that control, measure and, if possible, move a qubit in the device qubits

Historical parallels: the mission to build a universal computer

- the early 20th century to the mid-20th century: mechanical and electro-mechanical modalities, analog computing
- the rise of electrical digital computing: vacuum tubes, transistors, integrated circuits
- niche solutions: optical and sound-related modalities (propagation of sound waves to store information)
    - compare with quantum computing news (August, 2025): a team of Caltech scientists is “using sound to remember quantum information”

Variety of quantum modalities

- Superconducting (IBM, Google, Rigetti, OQC) and Microsoft’s topological qubits
- Trapped-Ion (IonQ, Quantinuum), Neutral-Atom Qubits (QuEra Computing, Pasqal), Photonic Qubits (Xanadu, PsiQuantum), and so forth

Superconducting Qubits

Created using superconducting circuits where quantum states are encoded in electrical currents or voltages

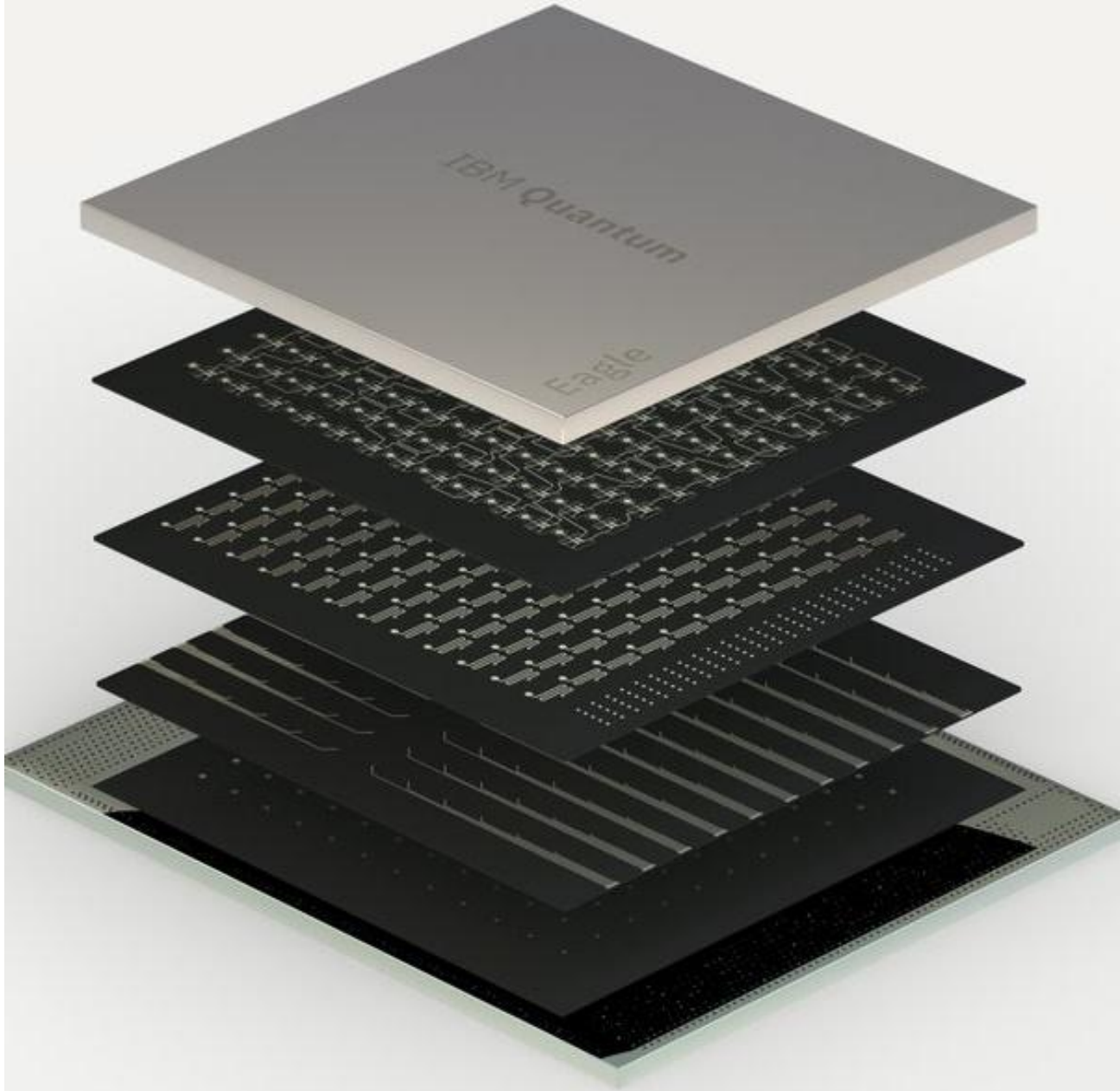
- Most widely available devices
- Fast gate operations
- Large physical footprint
- Massive cooling requirements
- Limited connectivity
- Horizontal scaling limitations
- Expensive and technically challenging to scale

Google AI Quantum

ALICE & BOB

rigetti

IBM Quantum

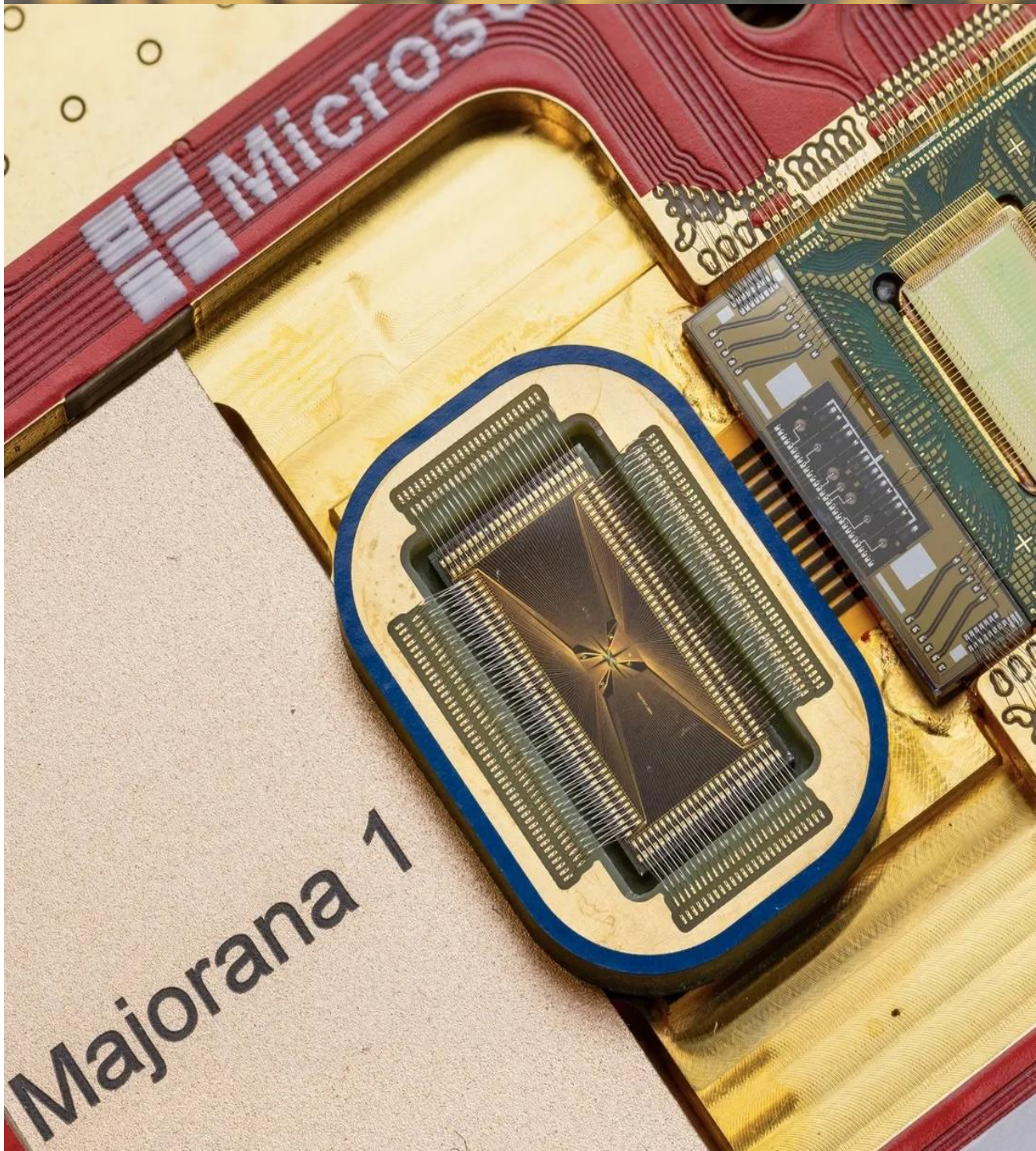


Majorana Topological Qubits

Based upon exotic half-abelian anyons at superconductor/semiconductor interfaces

- Protected from local noise and decoherence
- Long coherence times
- Challenging to control
- Inefficient interconnect
- Concerns scaling controls

Microsoft

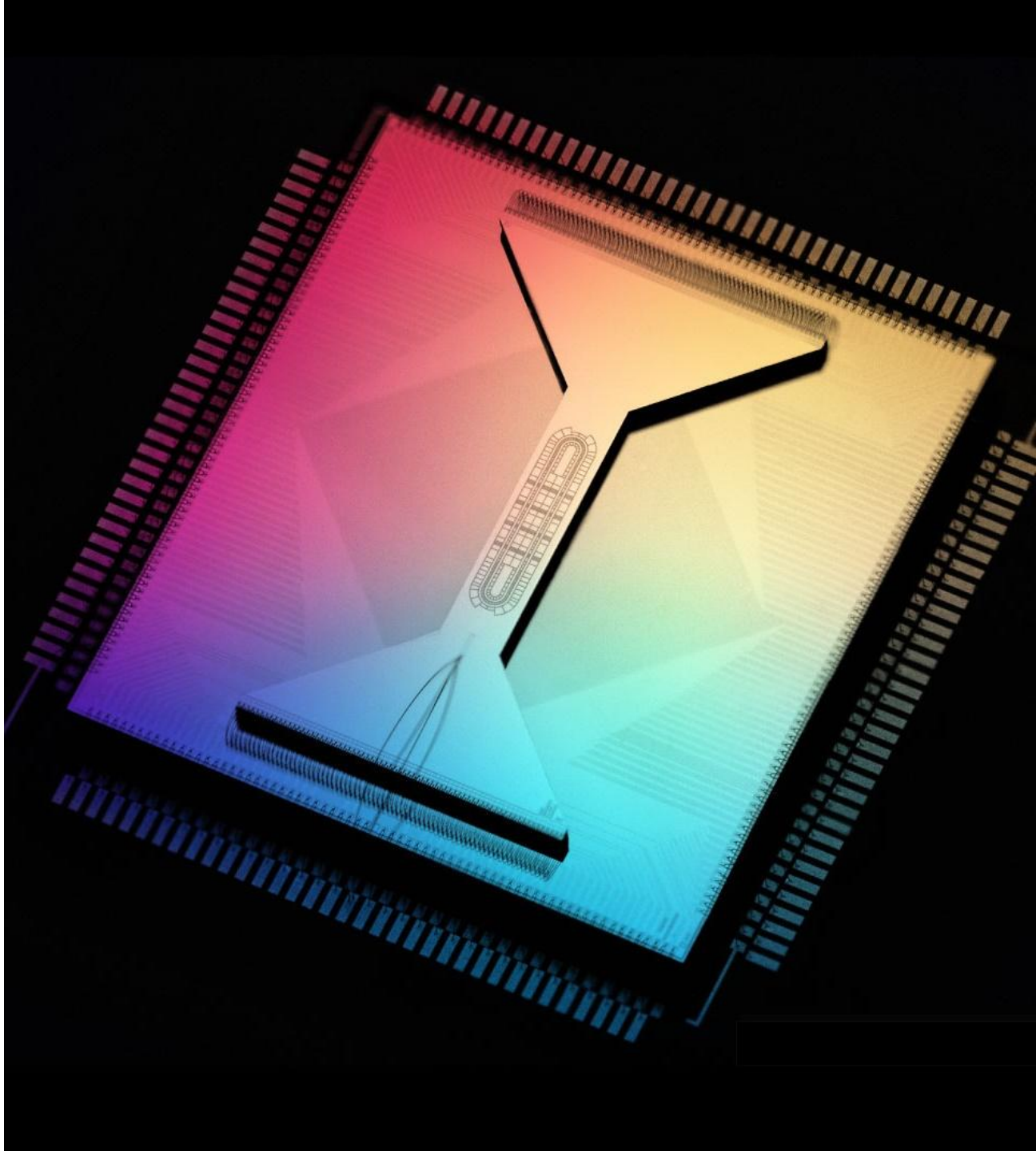


Trapped Ions

Based on the electronic and nuclear spin states of individual ions that are trapped using electromagnetic fields and manipulated using lasers

- High-fidelity gates
- Long coherence times
- All-to-all connectivity
- Slow trapping & constant re-cooling
- Slow shuttling & slow gates
- Vertical & horizontal scaling limitations

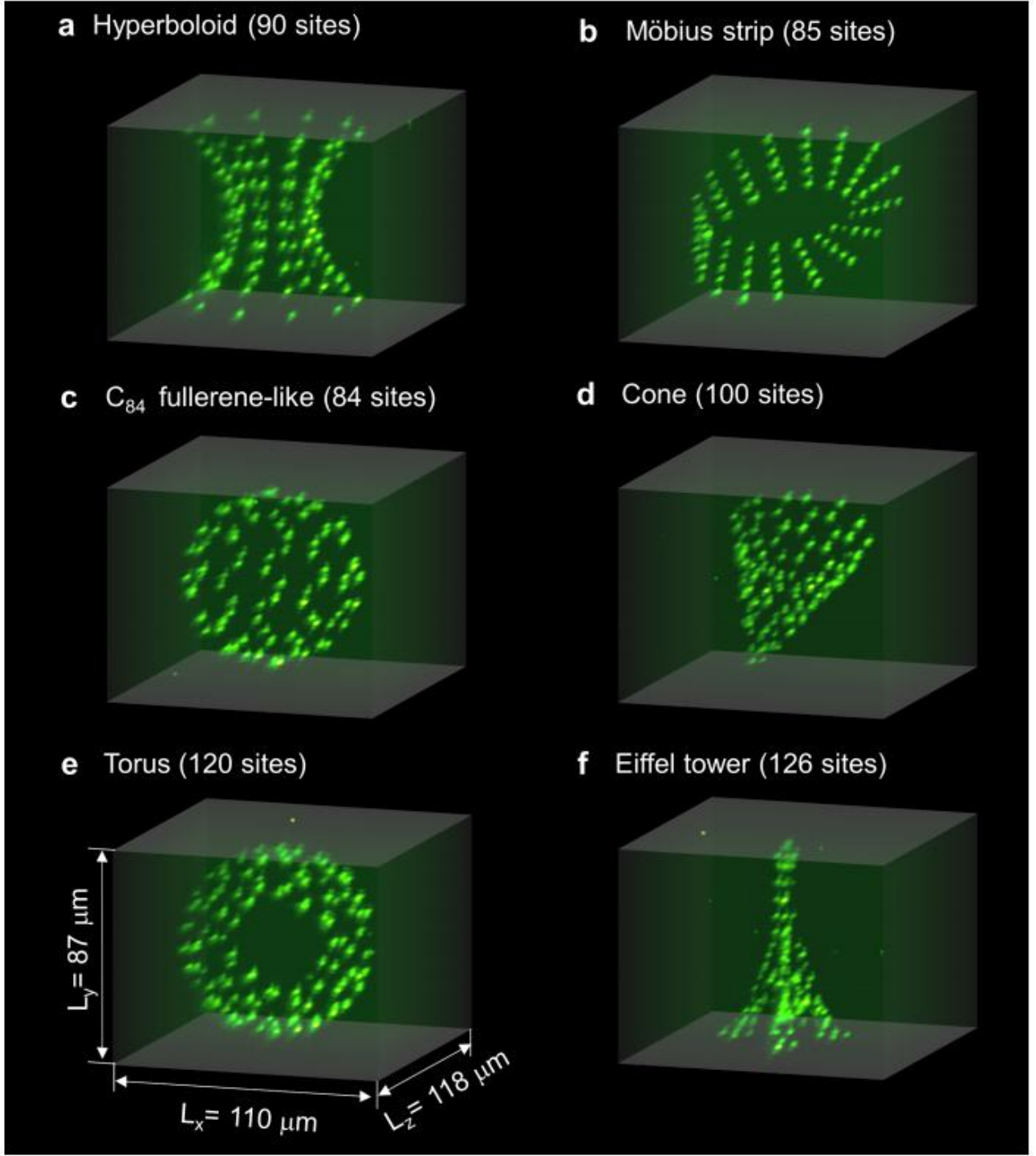
QUANTINUUM IONQ oxford ionics



Neutral Atoms

Realized by manipulating the quantum states of individual neutral atoms using laser beams

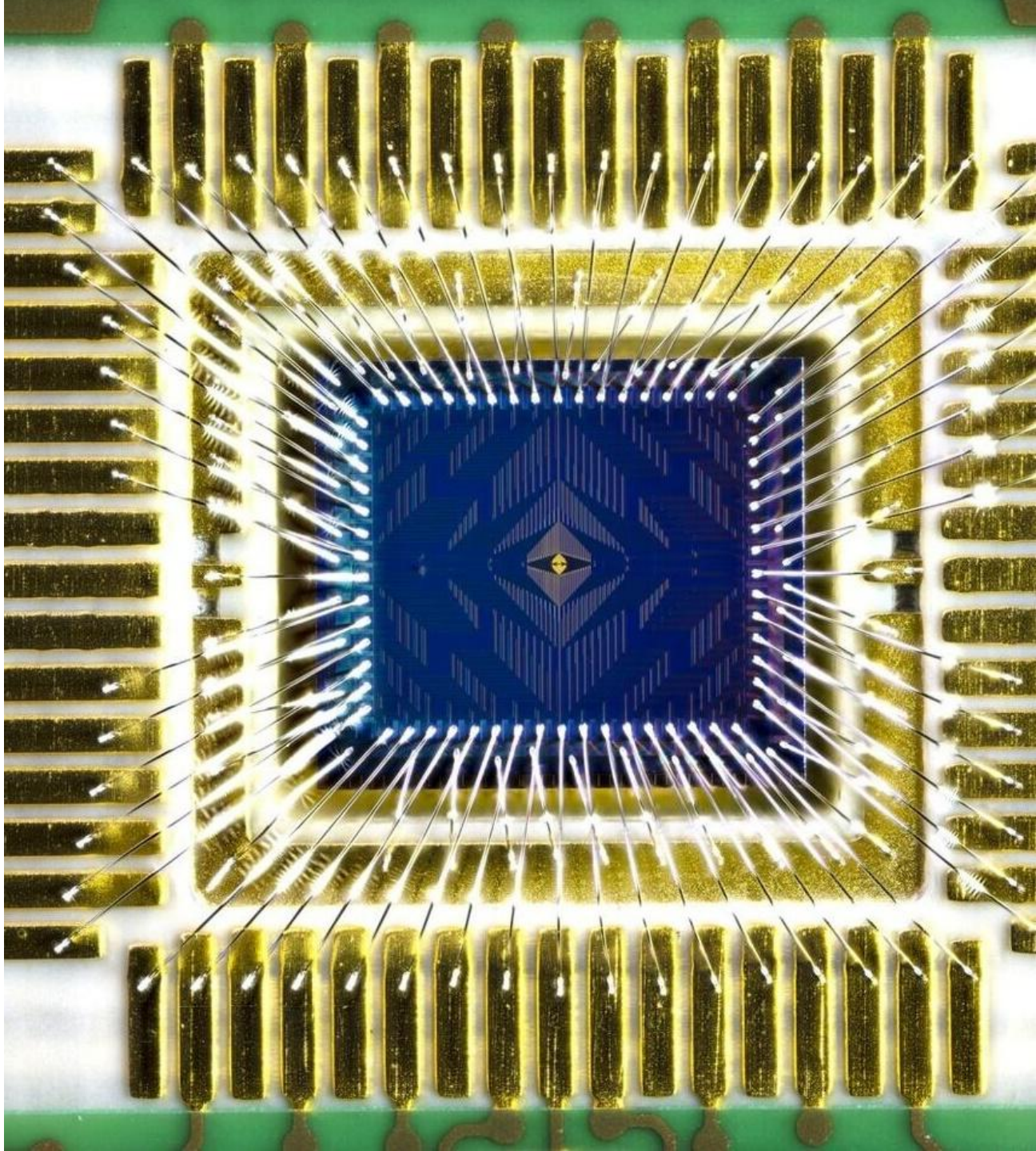
- Large qubit count
- Error correction progress
- Room temperature operations
- Slow probabilistic loading & constant re-cooling
- Slow shuttling & slow readout
- Vertical & horizontal scaling limitations



Pure Silicon Spin Qubits

Formed by the quantum spin states of particles, such as electrons or nuclei or quantum dots, often controlled using magnetic or electric fields

- Cost effective manufacturing
- Record coherence times
- Limited connectivity
- Horizontal scaling limitations
- Expensive to scale



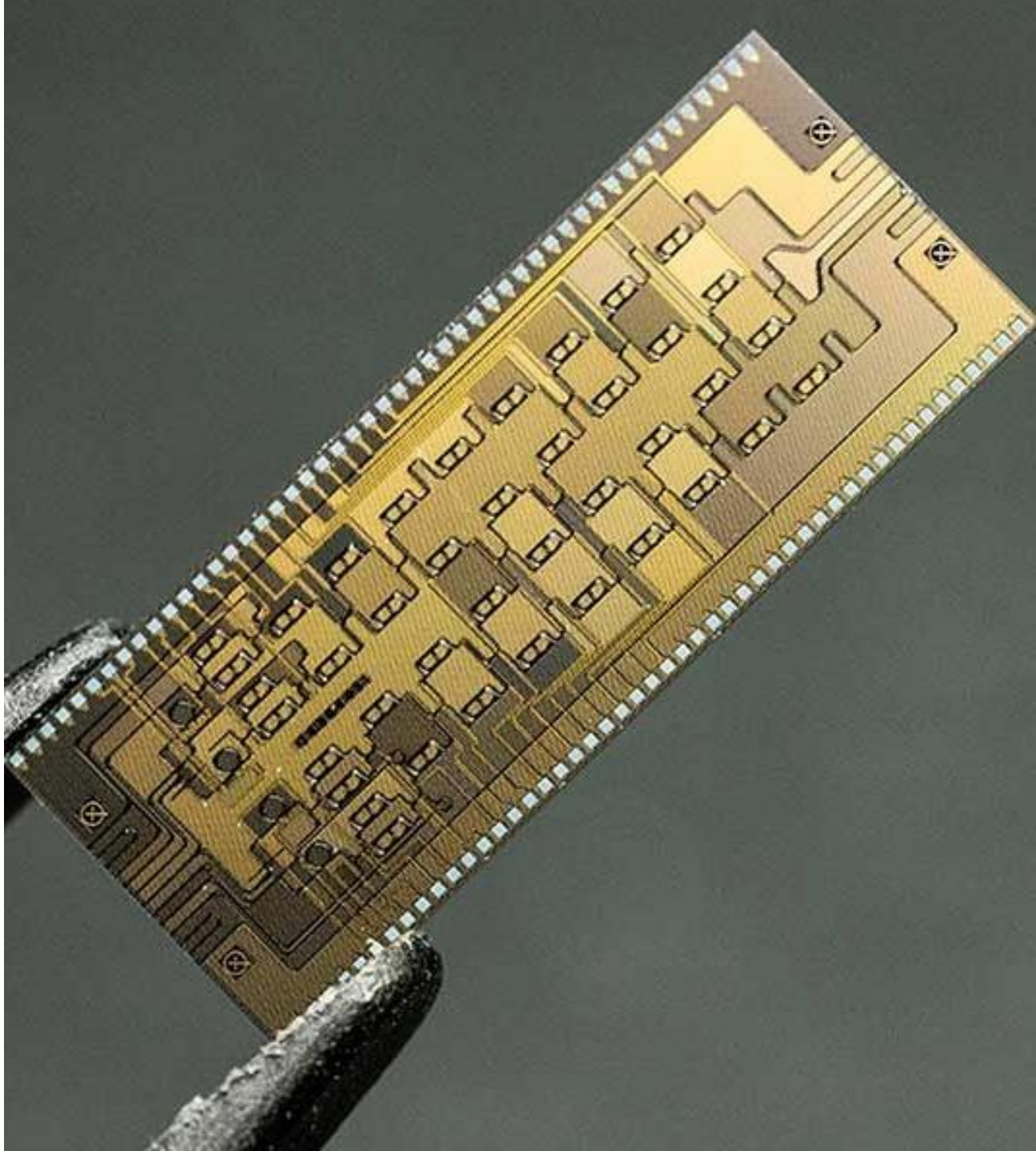
Pure Photon Based Qubits

Based on the quantum properties of light, such as polarization and phase, with manipulations via optical components

- Room temperature operations
- High connectivity = easy scale out
- Challenged by photon loss
- High footprint and energy requirements (expensive)
- Inefficient error correction

PsiQuantum

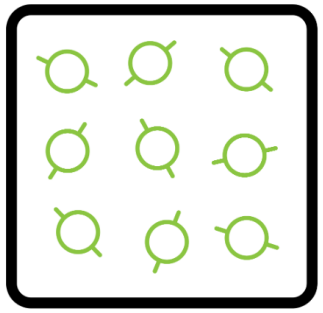
XANADU



Design Principles: Scaling Up vs. Scaling Out

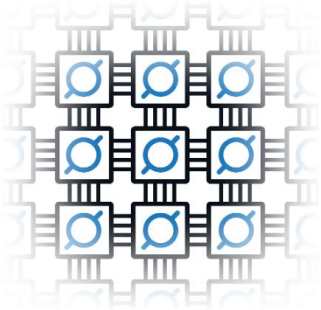
Even if quantum advantage is years away for most breathtaking use cases, architecture-centric decision-making defines future winners in quantum race

- quantum systems modularity matters: networking individual modules together
- highly parallelized entanglement distribution as a foundational aspect of the horizontally scalable architecture
- an important requirement that guides development of the compiler toolchain



Noisy Intermediate-Scale Quantum (NISQ)
- No known commercial applications

Monolithic, Small-Scale Logical
- No known commercial applications
- QaaS market limited to training & upskilling



Networked Logical Systems, Utility Scale
- Commercial applications
  - Drug discovery & development
  - Materials development
  - Catalyst development

Examples of MLIR in Compilers for Quantum Computing

IBM, qe-compiler: an MLIR-based compiler for OpenQASM 3

- QUIR (QUantum Intermediate Representation, circuit-level definitions), OQ3 (OpenQASM 3, classical and quantum abstractions), Pulse (OpenPulse, a language of "pulses programming"), QCS (Quantum Computing System, execution and synchronizing over the system of multiple controllers)

NVIDIA CUDA-Q: heterogeneous quantum-classical workflows

- Quake (circuit-level definitions, "not tied to a particular quantum hardware/machine"), CC (classical compute for the programming model)

Xanadu Quantum Technologies Inc., Catalyst in PennyLane, hybrid quantum programs

- Quantum (qubit management, gate operations, measurements and more), QEC (support for quantum error correction schemes)

Munich Quantum Toolkit, software tools for quantum computing

- mqtopt (MQTOpt) – value-semantics / running optimization dialect
- mqtref (MQTRef) – reference semantics / compatibility dialect (translations to and from Qiskit, OpenQASM, QIR)

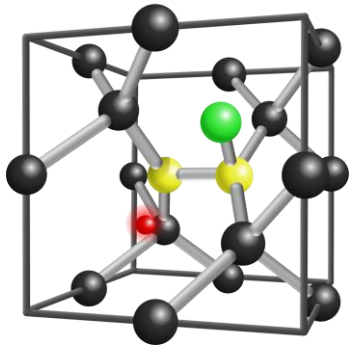
Photonic: a hybrid modality for distributed quantum computing

Spin-photon quantum modality: optically linked silicon spin qubits

- a dual-purpose physical component serves as both a quantum “processor” and a quantum “network node”
- technology is essentially different from the single-chip, local-gate model
- a distributed and modular system that uses photons to link silicon spin qubits
- the computation model is not based on direct physical gates but on the distribution and consumption of entanglement resources

How the quantum modality shapes design principles for the compiler toolchain

- classical computation and data transfer that happens during quantum execution must fit within the coherence time of the quantum state
- quantum gates comprise more complicated series of control and measurement steps
- probabilistic and network-aware process with retries
- control flow in the low-level language
  - hard real-time feedback loop at the hardware controllers: deterministic and latency-critical
  - mid-circuit measurements and interactions with low latency



From Hardware to Compiler: Defining Requirements

Distributed quantum computing within a hybrid (spin-photon) modality introduces specific compiler requirements

- Photonic's quantum control system consists of a network of interconnected qubit controllers
- Distributed embedded systems compiler
    - translates a global control model into synchronized code for a network of controllers with timing and communication constraints and control flow at the pulse level

Modular architecture

- a system is a hierarchy with independent, interchangeable components on each layer
- entanglement distribution and consumption is a first-class citizen of the architecture
    - well-defined interface between modules
    - base of better error correction codes due to all-to-all connectivity

Efficiency of the quantum computer design:

- maximize computing and reduce decision-making overhead (ref: Todd Austin's LEAN),
- fundamental need and primary constraint of physical qubits properties: nanoscale control is necessary

LLVM Takes the Stage

A solid engineering foundation

- no need to reimplement basic infrastructure

MLIR provides a compiler infrastructure and a tool to reason about transformations, data flow and stable IRs

- to leverage MLIR upstream dialects to express domain specific types, operations, control flow and use MLIR modeling and expressive capabilities (e.g., tensor, complex, and linalg to work with the quantum state, matrices and vectors operations)
- a convenient way to define a Virtual ISA

LLVM IR enables further code transformations and unlocks required backends (RISC-V)

- qubit controllers execute Photonic's ISA, which is a superset of RISC-V

LLVM helps, but backing is limited

- quantum compilation vs. pulses-level compilation: more help closer to the hardware
- MLIR: no upstream dialect for lower levels of distributed quantum computing (modeling time, synchronization, distributed environment , real-time constraints): time semantics is an opaque effect for analysis

From Quantum Architecture to Compiler Stack

Overcoming the challenge of entanglement distribution: to support logical compilation and quantum execution on scale

- treat entangled qubits as a resource that requires a dedicated subsystem managing the lifecycle (allocation, routing, scheduling)
- balance local and distributed capabilities to support both distributed computing and communication

Hardware and Software co-design:

- a hardware tool is controlled by a dedicated processing unit
  - straightforward execution of the instruction stream to work with a qubit, no speculative execution
  - decisions how to perform control are made at compile time if possible
- the compiler deals with the specific physical properties of the qubits (relaxation time T1, decoherence time T2)
  - coherence times of silicon spin (longer) and photons used for networking (shorter) anyway mean nanoscale precision and close-loop logic whenever feasible
  - error detection and recovery means constant interaction with QEC subsystem: less of an isolated component like in classic software environments, a library/tool to support quantum execution adaptive w.r.t the qubits state

A Compiler Toolchain for Fault-tolerant Quantum Computing

Quantum Error Correction (QEC) and non-deterministic quantum protocols (e.g., Magic State Distillation)

Quantum (specialized) compilation vs. low-level (general) compilation

- both translate high-level instructions into a set of instructions
- different meaning of “instructions” (target machine code vs. native quantum gates)
- both heavily rely on optimization
- focus on different resources (speed, memory, … vs. circuit depths, gate count, …)
- hardware limits of ISA and xPU architecture vs. physical properties on the underlying qubits technology

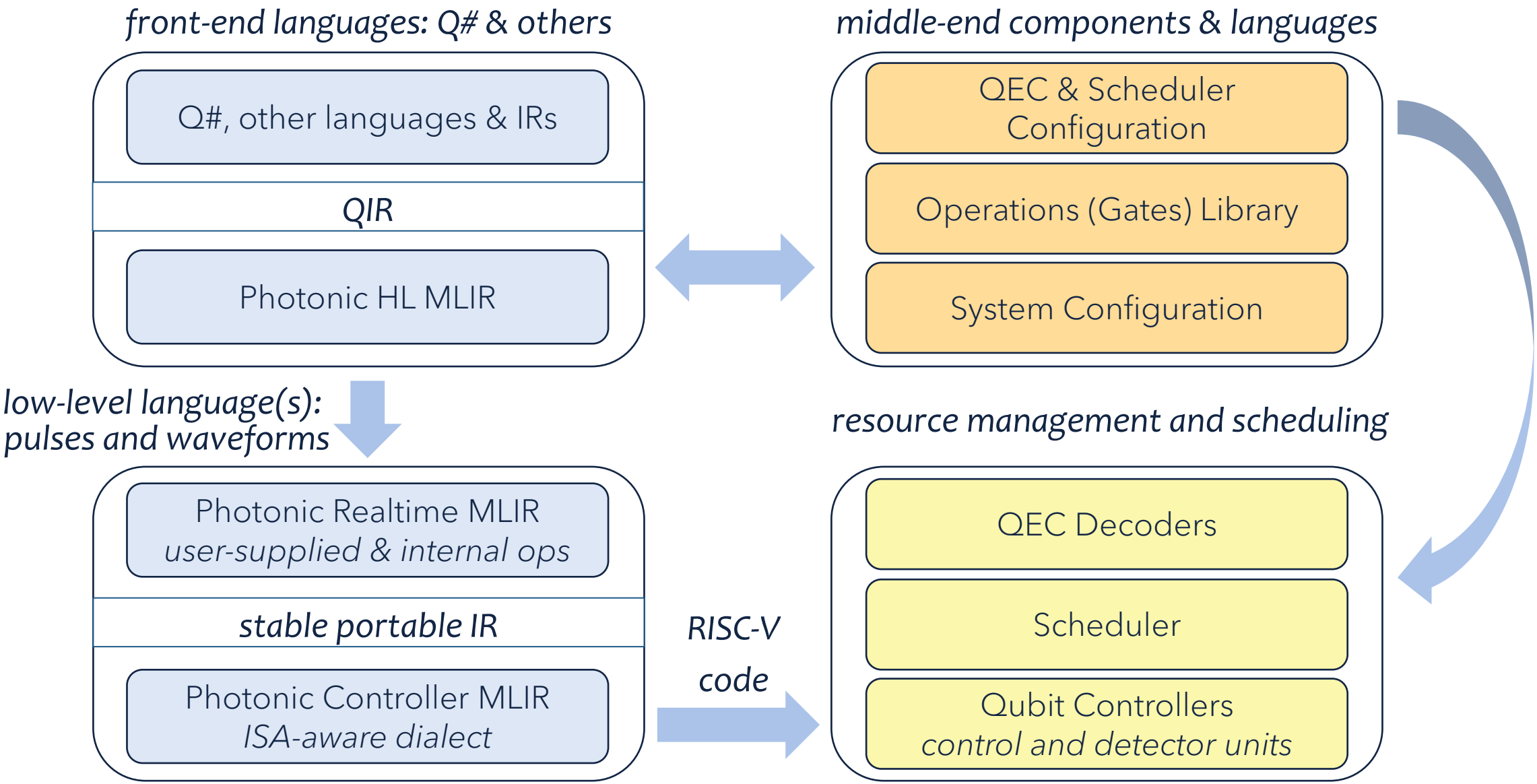
The compiler software stack may be naturally divided into several integrated but rather dissimilar architectural layers

- diverse scope and goals means different users, input languages, requirements and constraints

Compiler Layers: From Language to Control

Logical circuit	• Ideal logical qubits and gates • Logical entities must be decomposed, and circuit must be altered to support fault-tolerance
Conditional Corrections on the outcomes of syndrome extraction rounds	• QEC and non-deterministic quantum protocols require repeated measurements and conditional corrections • Auxiliary computational qubits for new operations
Computational Gates to Native Physical	• Decompose ideal gates into hardware-native basis • E.g., teleport CNOT for the spin-photon Photonic architecture
Gates Compilation	• Gates are compiled according to the operations/gates library (an analogy is standard library) • The program is represented as a schedule of pulses
Produce Controller-Level Code	• Use system configuration, transform the schedule according to devices calibration (an analogy is intrinsics) • Precise real-time schedule of pulses and waveforms

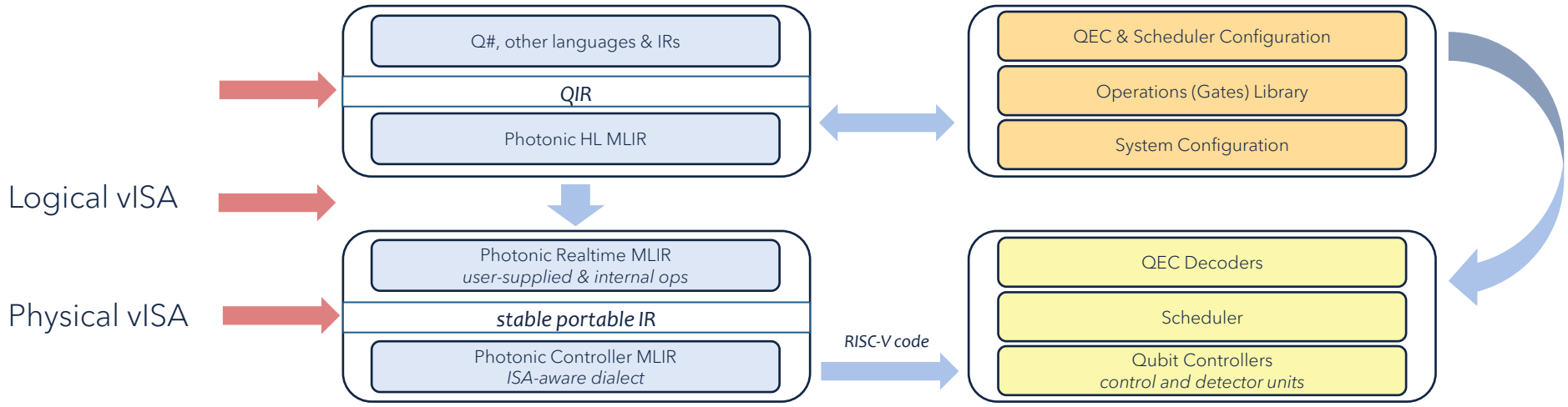
Photonic's Multi-Layer Compiler Stack



Virtual ISA

One of the core goals is to provide a unified representation for programmable manipulations on qubits:

- to comprise intermediate representation(s) of programs execution in the Photonic system architecture along with the underlying programming model
- to serve as an anchor and origin of stabilization for points after quantum compilation and before fully hardware aware lowering



- to span multiple hardware designs and generations of Photonic computing devices
- to isolate concerns, providing abstractions to bridge run-time with hardware control instruments and multiple front-ends with lowering and optimization

Examples of Technical Challenges in the Compiler

Integrate non-LLVM languages and instruments with the LLVM-based toolchain

- QIR extends LLVM IR with quantum-specific operations
  - representation for qubits, measurements, and runtime interactions using LLVM types and calls for different tools/backends to interoperate
  - needs to be integrated with Photonic MLIR concepts
- Users of middle- and low-level languages expect to work with Python or at least something Python-ish
  - LLVM and MLIR are technologies at the core of our compiler infrastructure
  - Python(-ish) language must be translated into Photonic MLIR dialects

Generation of distributed code for real-time software/hardware cooperation

- A programming model to support distributed computations, control flow and messaging at the nanoseconds scale

Code generation that helps to support compatibility across multiple hardware designs

- A stable representation across diverse hardware designs and generations of Photonic computing devices

Language of Pulses and Waveforms

The program is represented as a schedule of pulses

- the level of physical qubits and pulses/waveforms used to control and measure quantum states
- requires alignment and messaging between control and measurement devices behind physical qubits

Translation process

- outputs a precise real-time nanoscale schedule of pulses and waveforms in a form of RISC-V instructions and annotations
- use system configuration, transform the schedule according to devices calibration
- hardware controller-level code, proprietary micro-operations

Operation examples

- send/receive
- hardware-specific series of measurements coupled between devices
- perform a pulse or a waveform
- delay a pulse, align pulses across a set of devices

Language of Pulses and Waveforms Expressed in MLIR

Dialects

- Physical Virtual ISA and “a source of truth” w.r.t. the execution model

```
qx.channel "CtrlGroup" ["OPTA", "OPTB"]
qx.channel "DetGroup" [“DETA", “DETB"]
qx.kernel @Init_e() {
    scf.for %i = %c0 to %c10 step %c1  : i32 {
        qx.pulse 500
        qx.delay 500
    }
    qx.barrier "CtrlGroup"
}
qx.kernel @rt_x() {
    qx.waveform "rtx_e"
}
```

```
qx.device "DETA" {
    ...
    qx.barrier "DetGroup"
    %d0 = qx.detect 100
    %d1 = qx.receive "DETB"
    ...
    qx.send %heralded
```

- low level ISA-aware IR: MLIR dialects (cf, arith, llvm) and precise real-time schedule; proprietary; lowered to RISC-V

```
llvm.inline_asm has_side_effects asm_dialect = att "\09#COM x31 = GETTS", ""  : () -> ()
%14 = llvm.inline_asm has_side_effects asm_dialect = att "\09.word 0x06007f8b", "={x31}“ : () -> i32
```

Language of Pulses and Waveforms: Infrastructure

C++ core on LLVM/MLIR: performance, extensibility, upstream passes and standard pass management

Compiler as a library: Python package interface via MLIR bindings

- API above C++ passes; pybind/nanobind
- introspection exposed through the API: mid-translation analyses; reconfigurable pipeline
- integration with lab software and system configs
  - hardware topology
  - calibration profiles
  - pipeline options

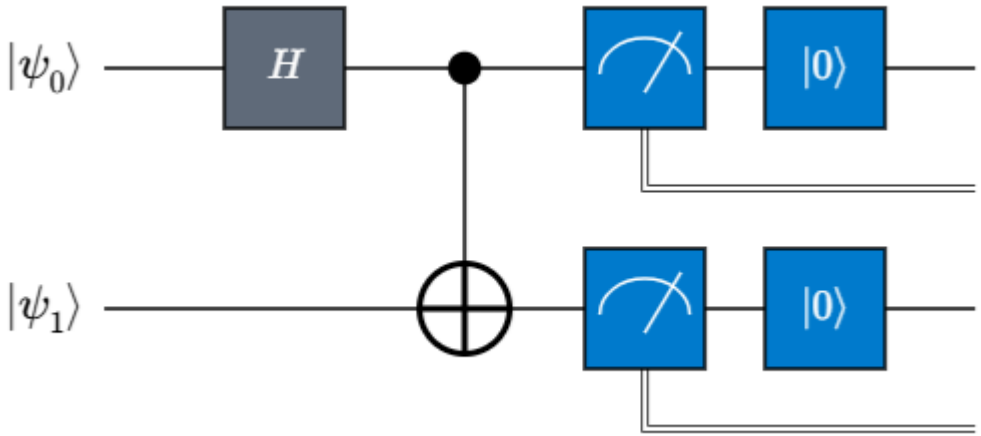
Python ergonomics for notebooks, automation, orchestration and testing

Bridging non-LLVM frontends with the MLIR based toolchain: Q#

Microsoft Q# compiler stack

- The public microsoft/qsharp repository: Rust, non-LLVM toolchain
- QIR is based on LLVM IR

```
namespace Test {
    @EntryPoint()
    operation Main() : Result[] {
        use q1 = Qubit();
        use q2 = Qubit();
        H(q1);
        CNOT(q1, q2);
        MResetEachZ([q1, q2])
    }
}
```

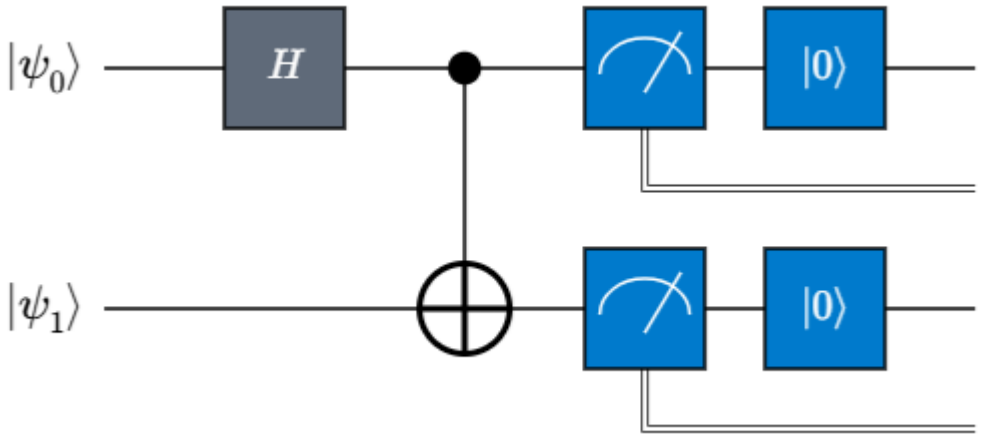


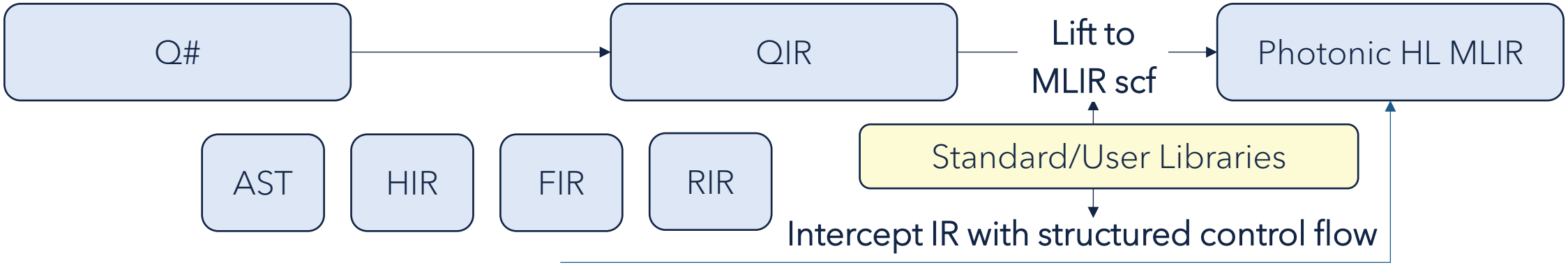
Bridging non-LLVM frontends with the MLIR based toolchain: Q#

Microsoft Q# compiler stack

- The public microsoft/qsharp repository: Rust, non-LLVM toolchain
- QIR is based on LLVM IR

```
namespace Test {
    @EntryPoint()
    operation Main() : Result[] {
        use q1 = Qubit();
        use q2 = Qubit();
        H(q1);
        CNOT(q1, q2);
        MResetEachZ([q1, q2])
    }
}
```





photonic

Bridging non-LLVM frontends with the MLIR based toolchain: Q# and QIR

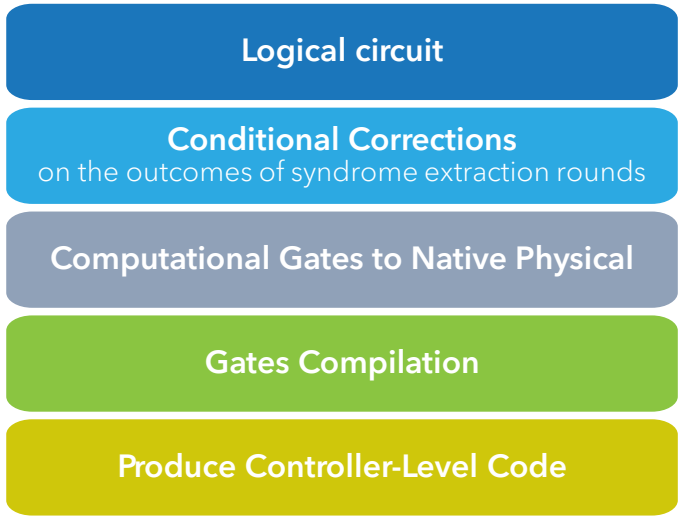
QIR extends LLVM IR with quantum-specific operations and it's architecture-neutral, but

- memory-semantics and intrinsics, representing gates as side effects on qubits
  - missing value-semantics mode, where changes in quantum states are represented in a functional style
- integration with compiler toolchains misses fine-grained control over lowering (e.g., qubits-gates level of annotation)
  - attaching metadata via attributes could be used by quantum compilation as hints or directives
- reasoning on a low level of (quantum concepts) abstraction: no support of fault-tolerant quantum computing
  - too big a gap between logical circuit level and QIR operations

Two ways to address the problem

- add custom MLIR dialects inspired by QIR with explicit support of quantum compilation concepts
- update QIR to state-of-the-art
  - introduce support for quantum compilation
  - become MLIR friendly

Quantum Computing DSL and Standard Library



The representation gap between compiler layers is too big:

- Logical circuit to Conditional Corrections
- Computational Gates to Native Physical
- Gates Compilation

Some middle-end layers can be parameterized via a “standard library”:

- the task can be offloaded to non-compiler engineers using high-level languages/Python DSLs: wrappers above MLIR dialects for different levels of abstraction

Use cases for Python DSLs: apply when MLIR real-time pulses dialects are too low level to express pages of business logics

- QEC and related non-deterministic quantum protocols-aware extensions to the Photonic HL MLIR dialect: to describe various repeat-until-success control flow chains and semantics of the interaction with the compiler pipeline and run-time (e.g., details of the syndrome extraction, magic-state injection, configurable transformations)
- Operations/Gate library: to define logics of mapping more complicated (teleporting) quantum gates
- wrap MLIR-encoded pulses and waveforms language(s) for native gates compilation: benchmarking, prototyping, calibration, experimentation (used independently from higher levels in the compiler stack)

Compilation (compile-time) vs. Execution (run-time)

Compile-time

Domain specific languages
(e.g., a subset of Python)

Photonic HLQ MLIR

Compiler for pulse languages

Photonic Pulse-level MLIR
user-supplied & internal ops

Photonic Controller-level MLIR
ISA-aware dialect

classical control flow

RISC-V: ops offloading

Run-time

Qubit Controllers
control and detector units

Classical components

System Configuration

Quantum runtime system

- Mapping between logical, computational qubits and hardware controllers
- Tracing computational qubits
- Schedule execution
- Manage resources

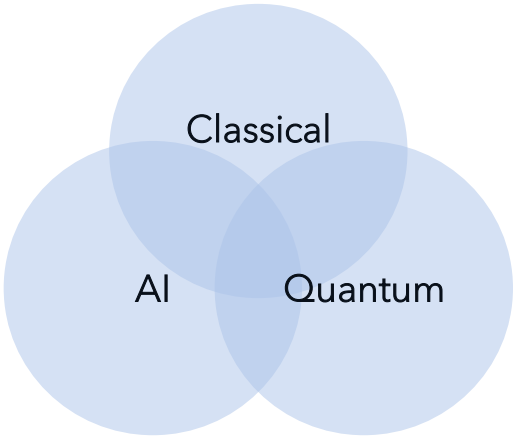
Python DSLs: a need for a native frontend for MLIR

Many Python features are not a quick and easy fit for MLIR: requires substantial analysis and runtime support

- MLIR: SSA, explicit types and dialects, structured control flow dialect restrictions
- Python: dynamic types, object model, and reflection; generators, exceptions and dynamic control flow

There are intersections between Mojo ideas and design goals and needs of quantum computing

- ambitions to rearrange the domain of heterogenous computing where quantum also belongs (in a way of thinking about hybrid computing environments)
- a Python-like syntax that functions as "syntax sugar for MLIR": direct programming support for MLIR concepts
- abstracting away the complexities of manual IR construction and making MLIR accessible to a broader audience
- match a high-level language concepts: e.g., early exits with region terminators (rcf.yield , rcf.break, rcf.continue), see the 2024 EuroLLVM talk and the RFC Region-based control-flow with early exits in MLIR



Can this lead eventually to a more general form of "a native frontend for MLIR"?

- supporting custom dialects' need for a Python-like syntax with less boilerplate

Controlling maintenance burden

What LLVM version to target?

- QIR and opaque pointers; the industry overall targets almost anything in between LLVM 14 and 21

What should be owned/customized?

- LLVM is a huge code base with fast pacing changes
- staying up-to-date requires more maintenance efforts
- example: the language of pulses and waveforms utilizes the RISC-V backend target
  - domain specific MLIR is lowered to the LLVM dialect, LLVM IR, and passed into the upstream backend
  - owning and customization would help to address certain engineering challenges of nanoscale control and monitoring, giving more fine-grained control instead of workarounds

We rely on stable releases and even then, upgrade the version conservatively

- a tradeoff between the latest features and higher risks: research/experiments pose specific requirements
- balance components of the stack: stability of the backend is more important than usability of middle/front-end

Quantum Computing: a Quest for Talents from Classical Software Science

More historical parallels: metaphorically speaking, are we in the 20th or the 17th century?

- John Wilkins: the Real Character, Gottfried Leibniz: The Calculus Ratiocinator and the Characteristica Universalis
- after 300 years with the advent of LLMs and generative AI, we are witnessing the advanced realization of this dream
    - open questions about the nature of "understanding," "consciousness," and "reason" in the context of LLMs

Quantum Computing: Substantial uncertainty and technical challenges vs. risks to miss the opportunity

- CMOS transistors vs. Qubits: extremely reliable (0.99999999…) vs. ~0.995-0.999 (two or three nines) for 2-qubit fidelity
- the risk to miss advent of practically useful quantum computing is too high – given its high promises, think about scientific revolution of the 17th century – to ignore the domain even when the technical difficulty level is so high

Growing industry demand for talents

- big funding rounds, large valuations and government support
- practical quantum systems: Hybrid systems, Compilers & runtimes, Algorithms & simulation, Tooling & platforms

Thank You