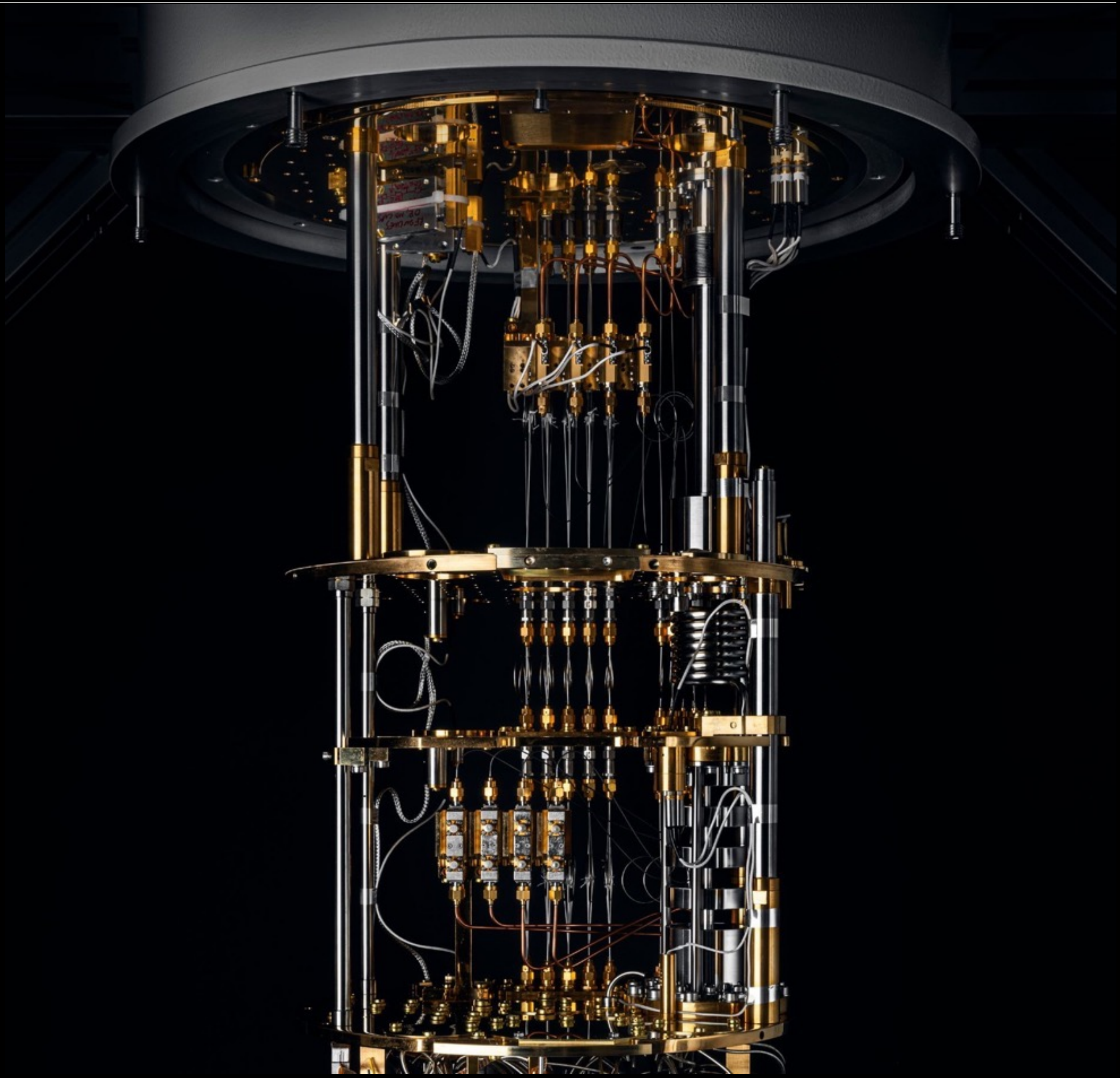
WE BUILD QUANTUM COMPUTERS

Integrating CUDA Quantum with Quantum Computers [S63123]



Vladimir Kukushkin
Lead Software Engineer

www.meetiqm.com

Hybrid quantum-classical programs

- Classical program – a program that runs on a CPU
- Quantum program – a program (called quantum circuit), will run on a Quantum Processing Unit (QPU)
  - A quantum circuit consists of quantum gates operating over qubits, like classical gates operating over bits
- Hybrid classical program with quantum subprograms
  - Any classical program that prepares a circuit, visualizes the results or saves results to a file
  - Special case to make it more clear – variational algorithms
- Similar to “classical” CPU/GPU programs

Writing quantum programs

- Python frameworks to define and execute circuits, IQM Qiskit adapter as an example to the right
- Jupyter notebooks to organize them into logical blocks
- Isolated reusable Python scripts
- Quantum ASM as an alternative approach, but lacks tooling
- Main users – quantum physicists and algorithms researchers optimizing programs for a specific quantum hardware
- Target hardware – small Noisy Intermediate Scale Quantum devices (tens to hundreds of noisy qubits)

```
import os

from qiskit import QuantumCircuit, execute, transpile

from iqm.qiskit_iqm import IQMProvider
from iqm.qiskit_iqm.iqm_transpilation import optimize_single_qubit_gates

iqm_server_url = os.environ.get("IQM_SERVER_URL")

circuit = QuantumCircuit(1)
circuit.h(0)
circuit.measure_all()

provider = IQMProvider(iqm_server_url)
backend = provider.get_backend()
qc_transpiled = transpile(circuit, backend=backend, layout_method="sabre", optimization_level=3)
qc_optimized = optimize_single_qubit_gates(qc_transpiled)
job = execute(circuit, backend, shots=10000)

counts = job.result().get_counts()
```

Pros and cons of common writing quantum programs with Python frameworks

- Pros:
  - Fast feedback cycle
  - Tailored for the quantum domain, e.g. allows operating on individual specific qubits
  - Easy to start, ideal for non software engineers
- Cons:
  - Python environments should be prepared in advance
  - Python dependencies are loaded on every run
  - Gates decomposition and routing is done on every execution, even if running against the same QPU
  - Reproducibility is challenging
  - Pure Python code is generally slow, which is especially noticeable with NISQ devices which do not yet show Quantum Advantage on a scale
  - Tricky to distribute Python programs

CUDA Quantum

- An open-source project maintained by NVIDIA
- `nvq++` compiler for C++-like hybrid programs, example program to the right
- Based on the state-of-the-art compiler technology – LLVM+MLIR
- Quantum Intermediate Representation for quantum subprograms representation
- Leverages LLVM compiler passes for **quantum specific logic**, like routing and gates decomposition
- Runtime support for execution against various quantum computer hardware
- Operates with “quantum memory”, not individual named qubits
- A more convenient tool for the future larger and QPUs

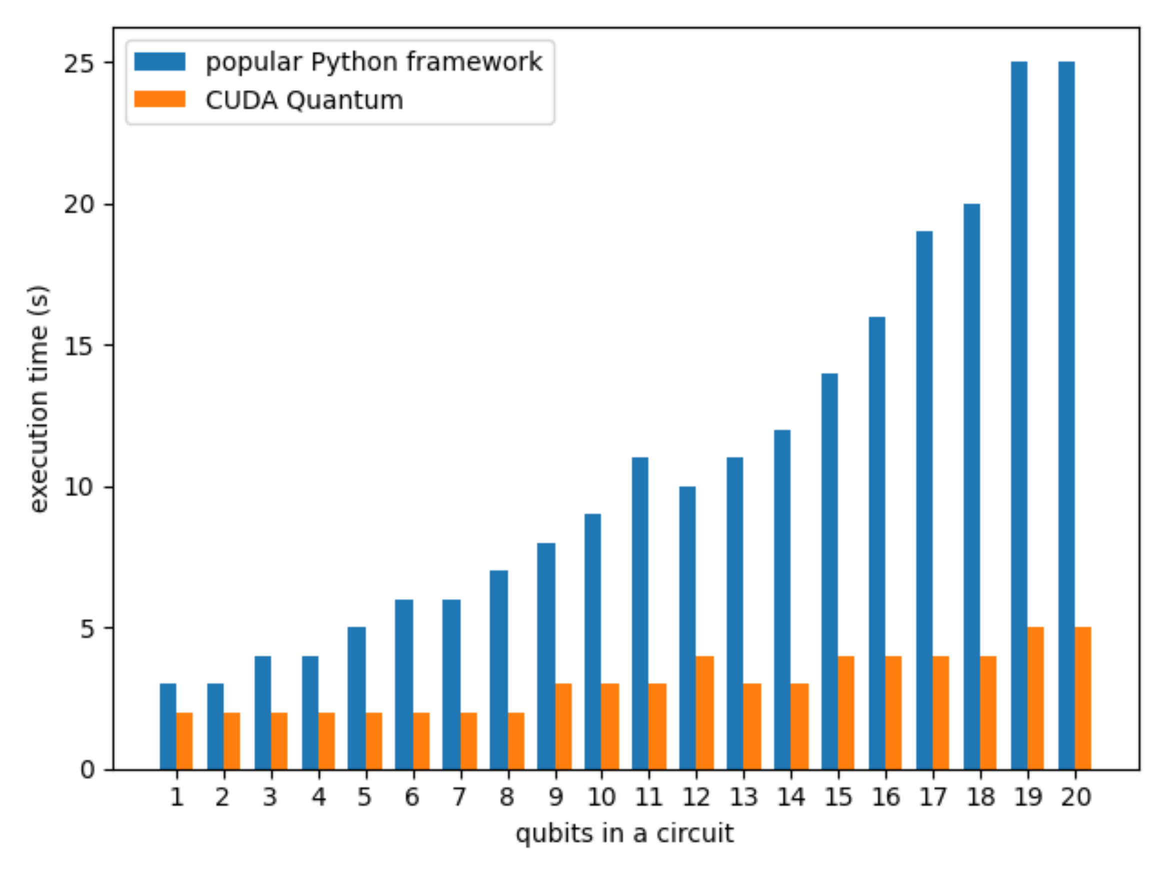
```cpp
#include <cudaq.h>
#include <iostream>

struct circuit {
  auto operator()() __qpu__ {
    cudaq::qreg q(1);
    h(q[0]);
    auto result = mz(q);
  }
};

int main() {
  cudaq::sample(10000, circuit{});
}
```

How different is CUDA Quantum?

- Compilation is separated in space and time from running (this is not obvious to engineers only familiar with Python!)
- Compiled binaries run faster in general
- Requires software engineering expertise to work with
- Easier distribution of compiled binaries, imagine an algorithms store
- Brings clear benefit (see the plot on the right) for multiple runs of programs compiled for a specific QPU architecture, like benchmarks or ready-made algorithms



Integration with IQM quantum computers

- Matches perfectly IQM quantum computers:
  - Multiple stations with the same QPU architecture available for customers
  - Internal QC hardware and software R&D with multiple stations, QPUs and software versions
  - Automated testing

```
nvq++  -target iqm --iqm-machine Adonis -v iqm.cpp -o iqm_example

IQM_SERVER_URL=https://demo.qc.iqm.fi/cocos ./iqm_example
```

Super-computer integration

- Predefined toolchain for handling user programs
- One of the steps for unifying integration with various QC hardware and software vendors
- Connects hardware infrastructure for hybrid CPU/GPU/QPU programs

QC HTTP API integration practicalities

- Make a new `nvq++` target, see `runtime/coda/platform/default/rest/helpers`
- Your remote API should support (examples at `runtime/codas/platform/default/rest`):
  - Single circuit submission for execution, QIR or any proprietary format (generate your definition from the QIR)
  - Jobs execution management, i.e. checking status and retrieving **counts** results
- Your API can also support authentication, but CUDA Quantum needs to know how to use it, e.g. getting an API Key from environment
- Keep in mind two steps while implementing: compilation vs execution
  - Example: IQM adapter allows compiling against a specific QPU architecture and then run against any QC with a runtime provided control software URL
- Add tests and CI integration, consider providing an integration endpoint for the CUDA Quantum CI
- Submit a pull request to the CUDA Quantum project

How can you try it?

- IQM Cloud (available later 2024)
- IQM Spark – on-premise 5 qubits QC
- Check CUDA Quantum project for more hardware integrations

Thanks!

Vladimir Kukushkin
Lead Software Engineer
Vladimir.Kukushkin@meetiqm.com

IQM

www.meetiqm.com