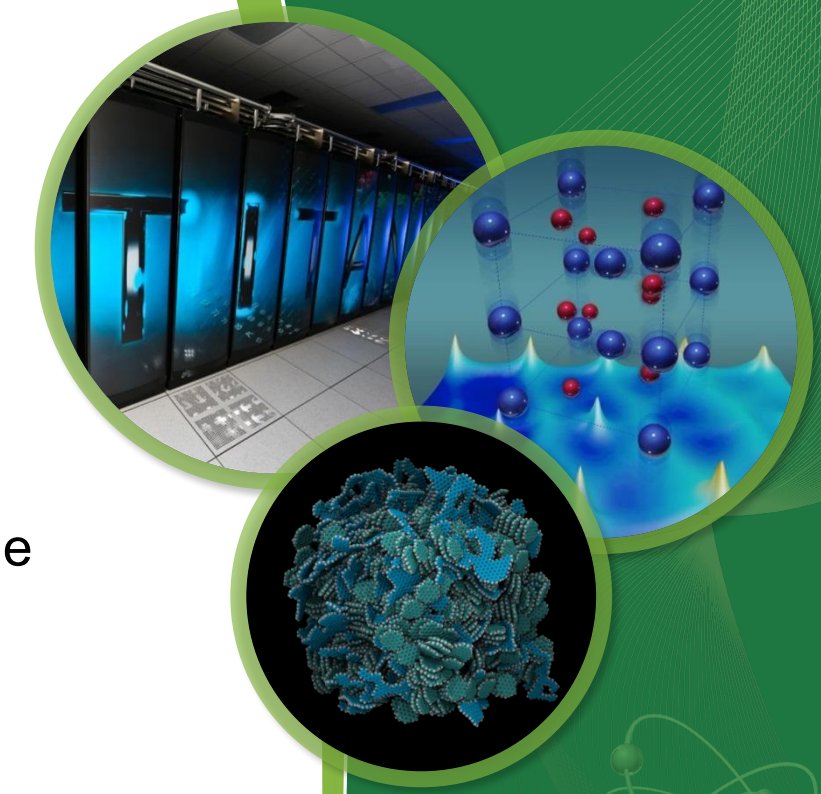
Quantum Computer Programming, Compilation, and Execution with XACC

Alex McCaskey
Software Lead, Quantum Computing Institute
Oak Ridge National Laboratory
mccaskeyaj@ornl.gov

with Eugene Dumitrescu, Dmitry Liakh, and Travis Humble

D-Wave Qubits Conference, September 2018

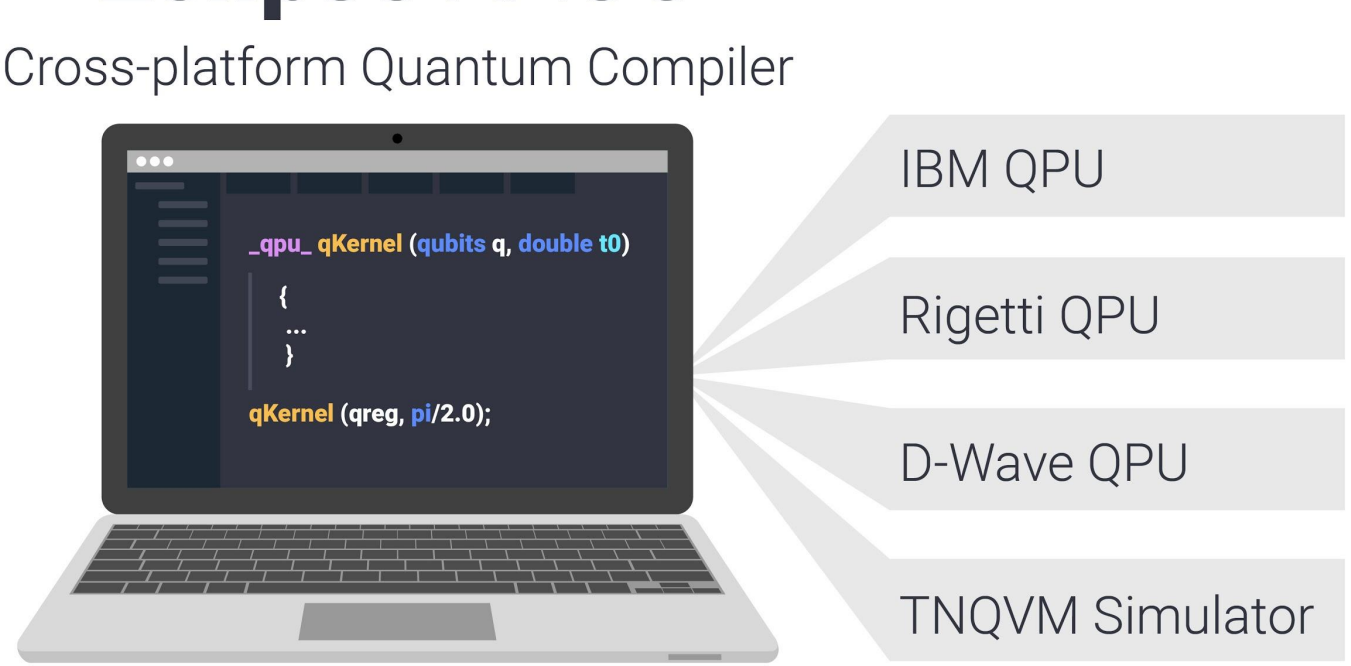


This work is supported by the DOE ASCR Early Career Research Program, Quantum Algorithms Teams, Quantum Testbed Pathfinder and the ORNL LDRD fund.

26 September 2018

Outline

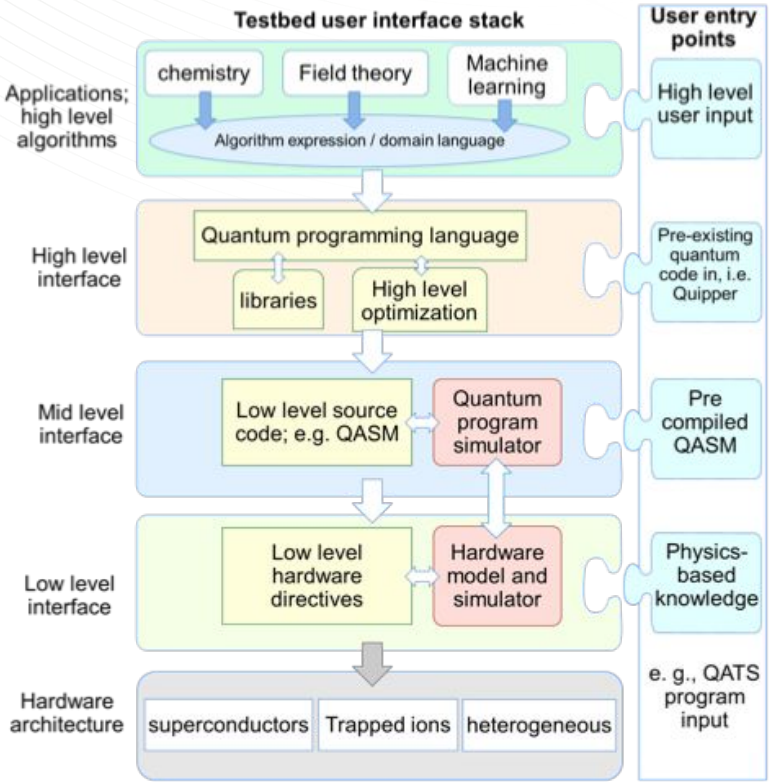
- Motivation, ORNL Requirements
- XACC
  - Birds-eye view and software stack
- Architecture
  - Platform and Memory model
  - Programming model
- Python JIT



Quantum Software Efforts at ORNL

- What is driving ORNL quantum programming efforts?

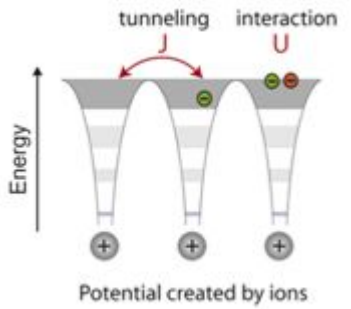
DOE Testbed Project - Materials and Interfaces for Quantum Acceleration of Science Applications (MIQASA)

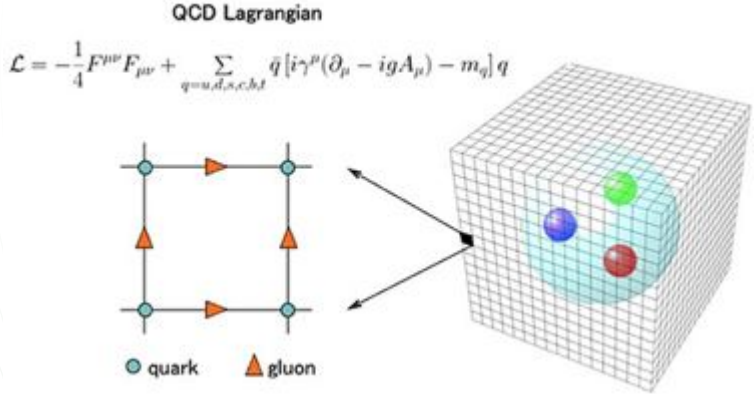


ORNL Programming Requirements

- Anticipation of ORNL Post-Exascale Computing
  - treat QPUs as accelerators
  - leverage wealth of classical co-processor R&D
- MIQASA benchmarking across QPU hardware types
  - enable hardware-independent programming
- HDAQDS execution across QPU types and high-level program expression
  - extensibility in compilation techniques

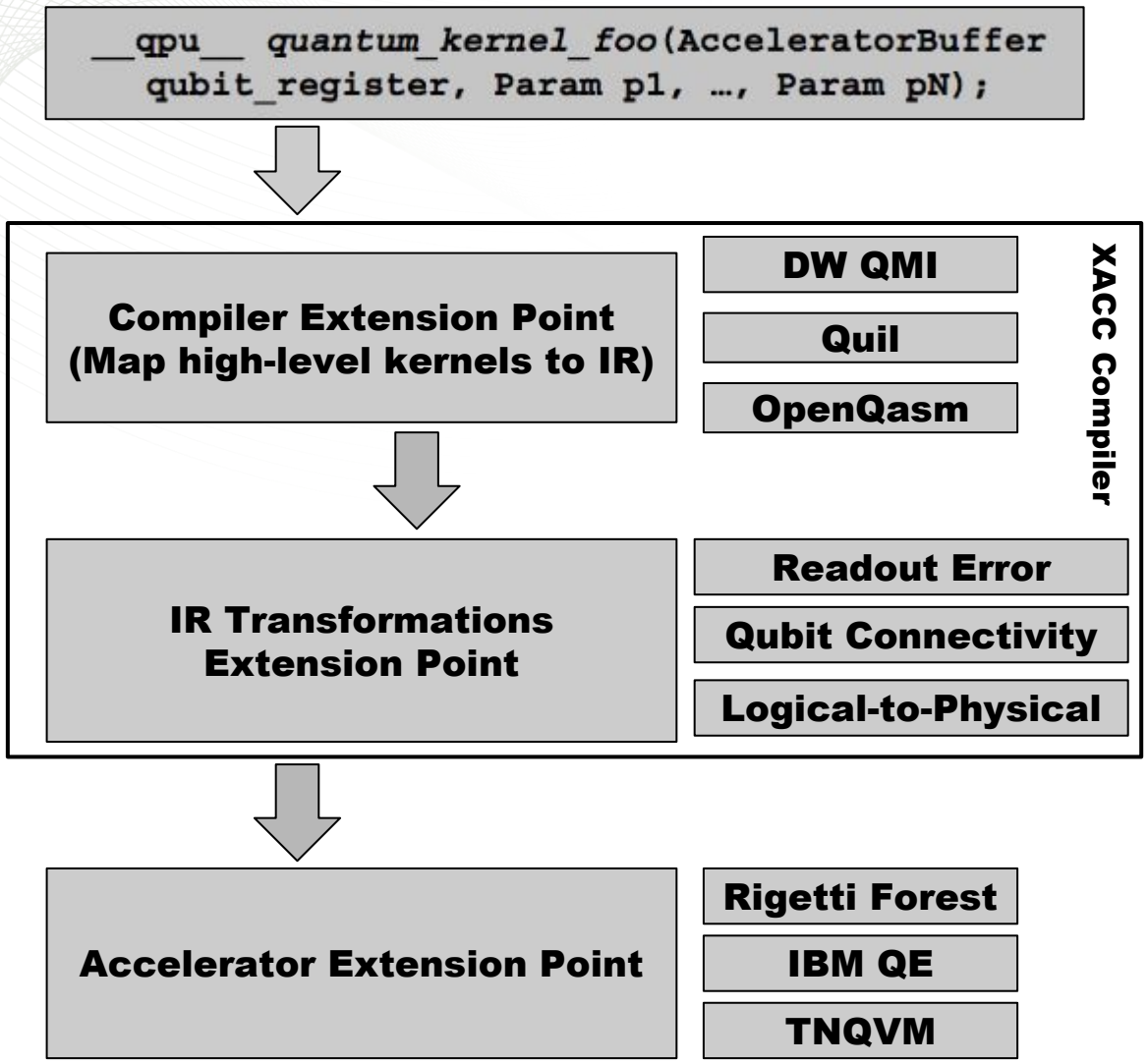
DOE QAT - Heterogeneous Digital-Analog Quantum Dynamics Simulation (HDAQDS)





XACC - Compiler Framework for Quantum Computing

Users define Quantum Kernels (Kernel in the GPU sense, a C-like Function)



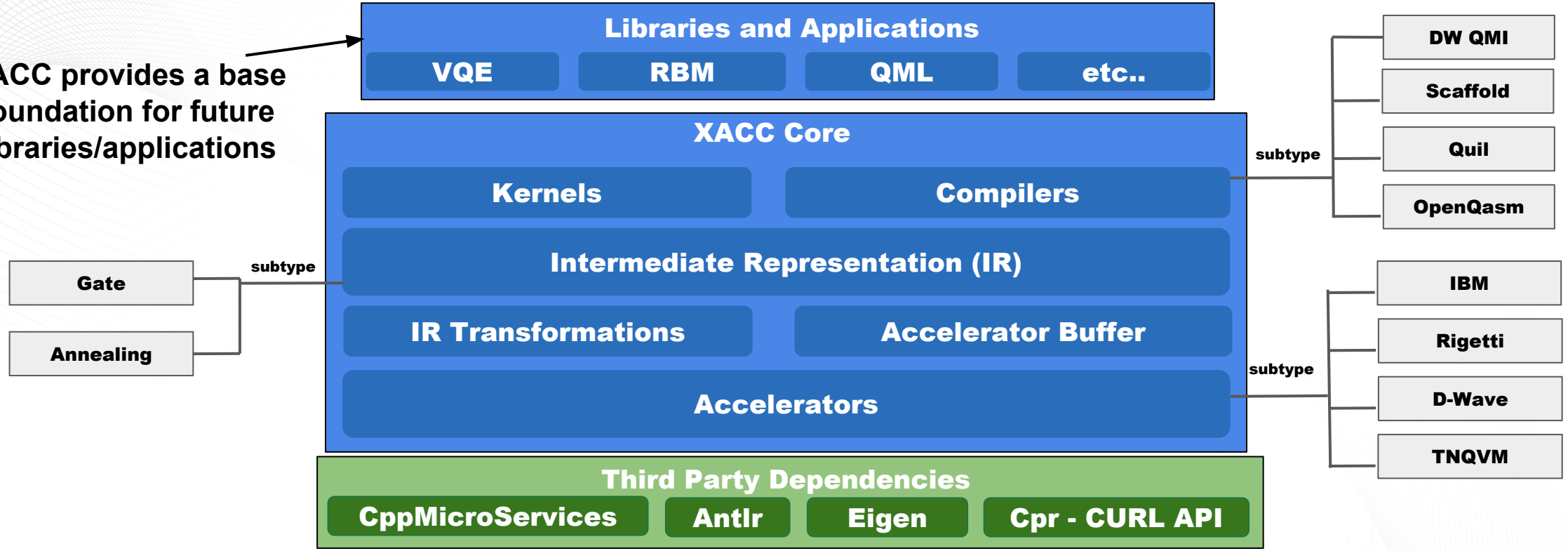
- Open-source at https://github.com/eclipse/xacc
- Familiar API and model
  - OpenCL-like, LLVM-like, hardware and language independent
- Key Abstractions
  - Kernels and Compilers (Frontend)
  - Intermediate Representation (Middle-end)
  - Accelerators (Backend)
- OSGI C++ *modular, service-oriented architecture*
  - Framework Extension Points

```
// Get reference to the QPU, and allocate a buffer of qubits
auto qpu = xacc::getAccelerator("ibm");
auto buffer = qpu->createBuffer("qreg", 2);

// Create and compile the Program from the kernel
// source code. Get executable Kernel source code
auto kernelSourceCode = "__qpu__ foo(double theta) {....}";
xacc::Program program(qpu, kernelSourceCode);
program.build();
auto kernel = program.getKernel<double>("foo");

// Execute over theta range
for (auto theta : thetas) kernel(buffer, theta);
```

XACC Software Stack

Key architectural design: service-oriented, modular, extensible

XACC provides a base foundation for future libraries/applications



- **CppMicroServices**
  - https://github.com/CppMicroservices/CppMicroServices
  - Native C++ OSGI implementation
  - Framework, Bundle, BundleContext abstractions
  - XACC plugins are exposed as Framework Bundles provided at runtime as shared libraries
  - CppMicroServices abstracts away dlload,dlopen,dlsym commands

- **Antlr**
  - Automated Parser Generation for user-defined languages
  - Define EBNF Grammar, generate Parser code
  - Our compilers use Antlr Parsers to generate quantum code AST, then we map AST to XACC IR
- **Cpr**
  - Curl for People :), spiritual port of Python Requests

XACC Architecture

Platform, Memory, and Programming Model

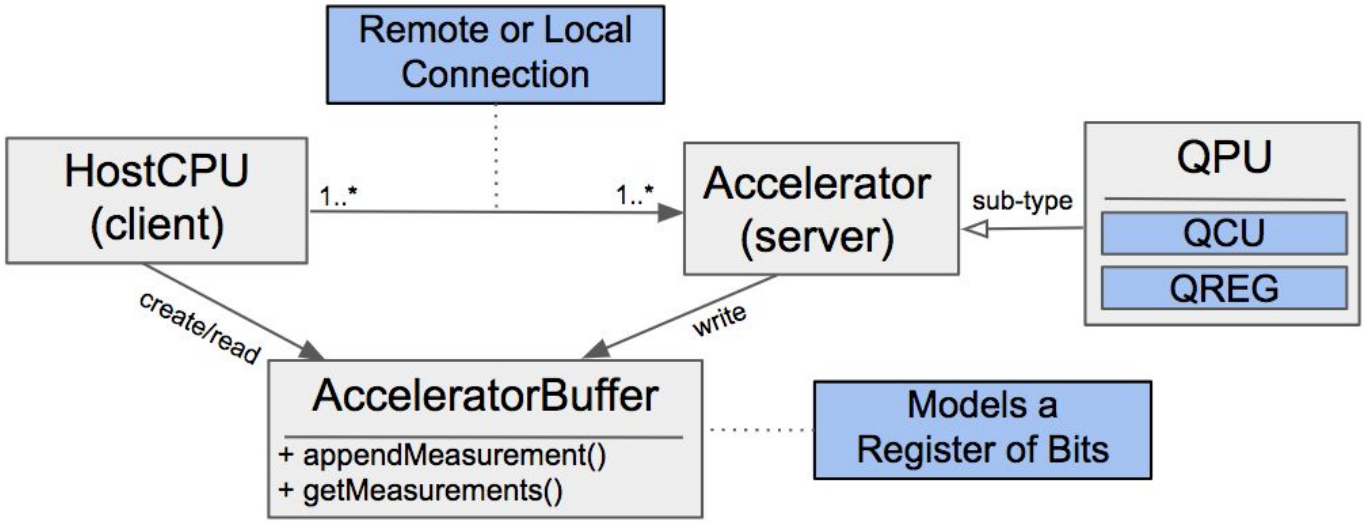
What makes this API possible?

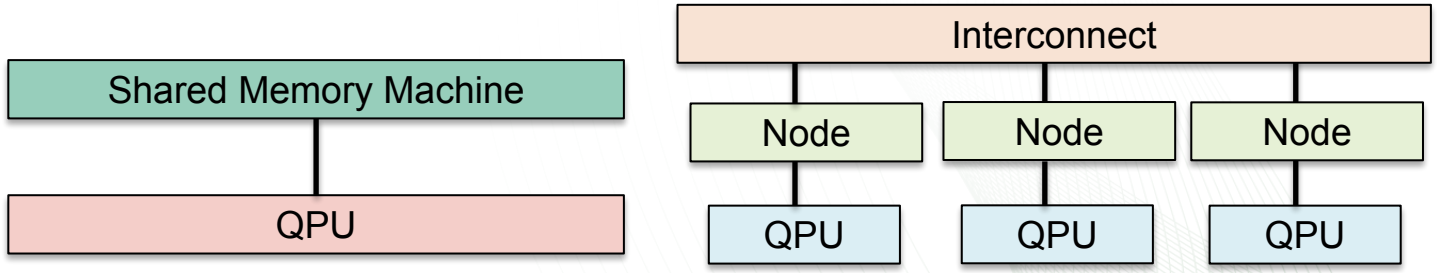
```python
# Get reference to the QPU, allocate a buffer of qubits
qpu = xacc.getAccelerator('ibm')
buffer = qpu.createBuffer('q',2)

# Create and compile the Program from the kernel
# source code. Get executable kernel
kernelSourceCode = '__qpu__ foo(double theta) {...}'
program = xacc.Program(qpu, kernelSourceCode)
program.build()
kernel = program.getKernel('foo')

# Execute over theta range
[kernel.execute(buffer, [theta]) for theta in thetas]
```

```cpp
// Get reference to the QPU, and allocate a buffer of qubits
auto qpu = xacc::getAccelerator("ibm");
auto buffer = qpu->createBuffer("qreg", 2);

// Create and compile the Program from the kernel
// source code. Get executable Kernel source code
auto kernelSourceCode = "__qpu__ foo(double theta) {....}";
xacc::Program program(qpu, kernelSourceCode);
program.build();
auto kernel = program.getKernel<double>("foo");

// Execute over theta range
for (auto theta : thetas) kernel(buffer, theta);
```

XACC Platform and Memory Model

- Platform model, system context
  - 3 primary actors, host CPU, Accelerator, and AcceleratorBuffer
  - client-server model
  - Subtypes of the Accelerator system
- Enables serial and parallel execution contexts
- Enables a number of co-processor integration strategies
  - loose and tight coupling, remote or localhost
- Measurement results shared through AcceleratorBuffer





Programming Model - Kernels

- XACC Kernel definitions are GPU-like (OpenCL, CUDA)
  - familiar mechanism for programming accelerator
- Kernels can be parameterized
  - **`InstructionParameter`** variant data structure (**`float,int,double,string,complex`**)
- Annotated with **`__qpu__`** function attribute for future static, ahead-of-time compiler based on Clang

`__qpu__ quantum_kernel_foo(AcceleratorBuffer qubit_register, Param p1, …, Param pN);`	
XACC Kernel Requirement	**Description**
Annotation	All kernels must be annotated with the __qpu__ function attribute to enable static, ahead-of-time compilation
Kernel Name	All kernels must be given a unique name
Accelerator Buffer Argument	All kernels must take as a first function argument the Accelerator Buffer this kernel acts on.
Runtime Parameters	All kernels can take any number of runtime arguments.

Example XACC Kernels

```
__qpu__ teleport (qbit& qreg) {
    X(qreg[0]);
    H(qreg[1]);
    CNOT(qreg[1],qreg[2]);
    CNOT(qreg[0],qreg[1]);
    H(qreg[0]);
    cbit c1 = MeasZ(qreg[0]);
    cbit c2 = MeasZ(qreg[1]);
    if(c1 == 1) Z(qreg[2]);
    if(c2 == 1) X(qreg[2]);
}
```

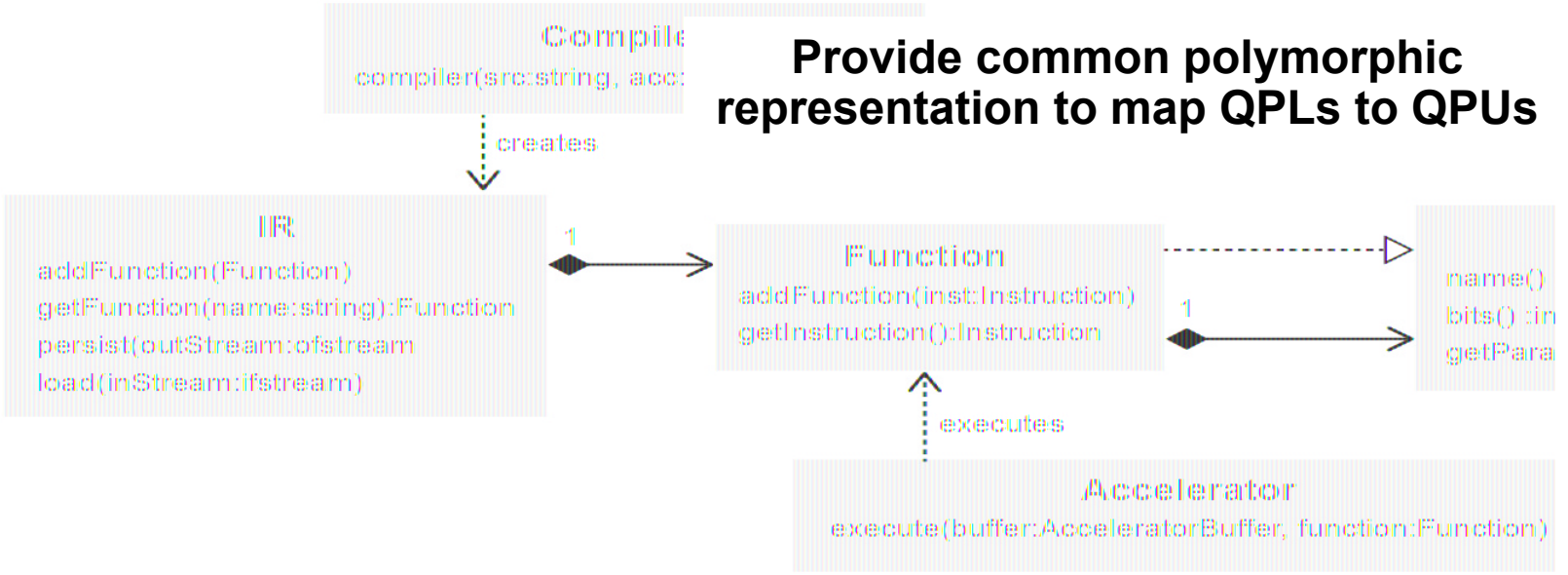
```
__qpu__ variable(AcceleratorBuffer ab, double ta,
                 double tp, double tq,
                 double h, double J) {
    anneal ta tp tq;
    0 0 h;
    1 1 h;
    0 1 J;
}
```

```
__qpu__ ansatz(AcceleratorBuffer b, double t0) {
RX(3.1415926) 0
RY(1.57079) 1
RX(7.85397) 0
CNOT 1 0
RZ(t0) 0
CNOT 1 0
RY(7.8539752) 1
RX(1.57079) 0
}
__qpu__ term0(AcceleratorBuffer b, double t0) {
ansatz(b, t0)
MEASURE 0 [0]
}
__qpu__ term1(AcceleratorBuffer b, double t0) {
ansatz(b, t0)
MEASURE 1 [0]
}
```

```
__qpu__ term2(AcceleratorBuffer b, double t0) {
ansatz(b, t0)
MEASURE 0 [0]
MEASURE 1 [1]
}
__qpu__ term3(AcceleratorBuffer b, double t0) {
ansatz(b, t0)
H 0
H 1
MEASURE 0 [0]
MEASURE 1 [1]
}
__qpu__ term4(AcceleratorBuffer b, double t0) {
ansatz(b, t0)
RX(1.57079) 0
RX(1.57079) 1
MEASURE 0 [0]
MEASURE 1 [1]
}
```

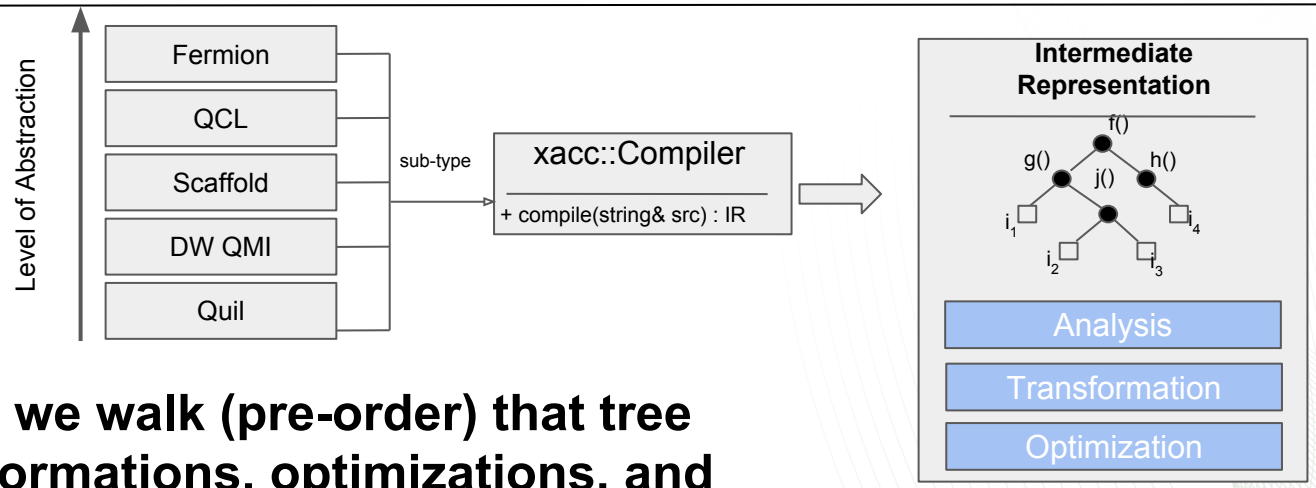
XACC Intermediate Representation

- **Instructions**
  - operate on qubits, unique names, parameterized
- **Functions**
  - Are Instructions but also contain further Instructions - Composite Pattern
  - Compiled representation of Kernel functions
- **IR**
  - Container of Functions, forest of trees
  - IR Transformations and IR Preprocessors



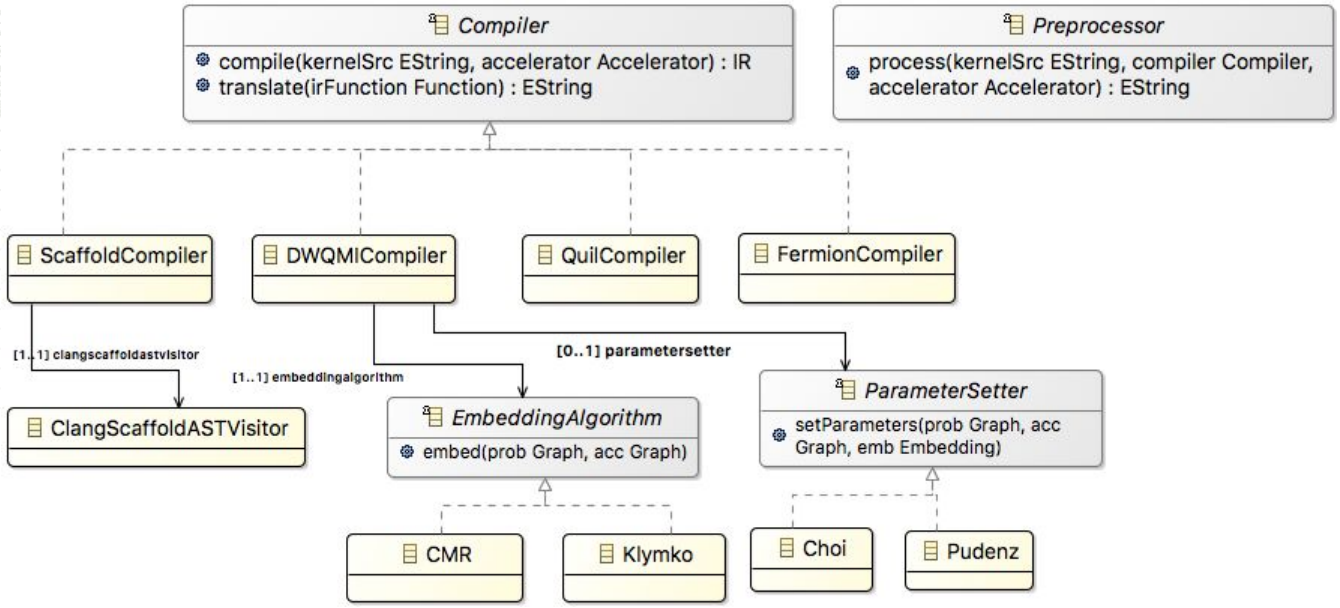
Provide common polymorphic representation to map QPLs to QPUs

```
__qpu__ j(AcceleratorBuffer b) { i2 1; i3 0;}
__qpu__ g(AcceleratorBuffer b) { i1 0; j();}
__qpu__ h(AcceleratorBuffer b) { i4 2;}
__qpu__ f(AcceleratorBuffer b) { g(); h();}
```



XACC IR models an *n-ary tree*, and we walk (pre-order) that tree to perform program analysis, transformations, optimizations, and executions

Programming Model - Compilers and Transpilers

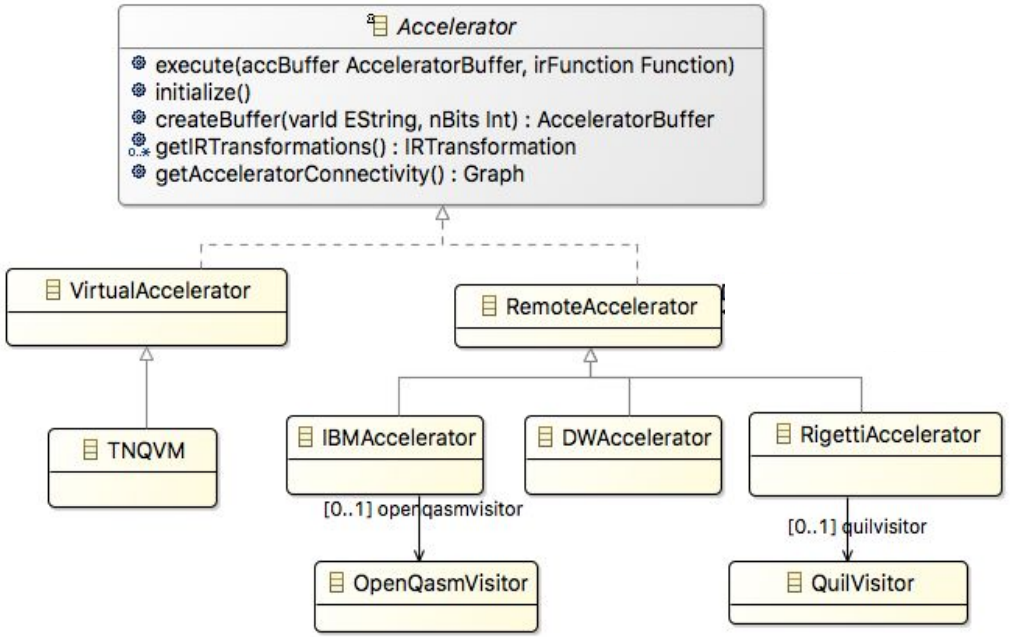


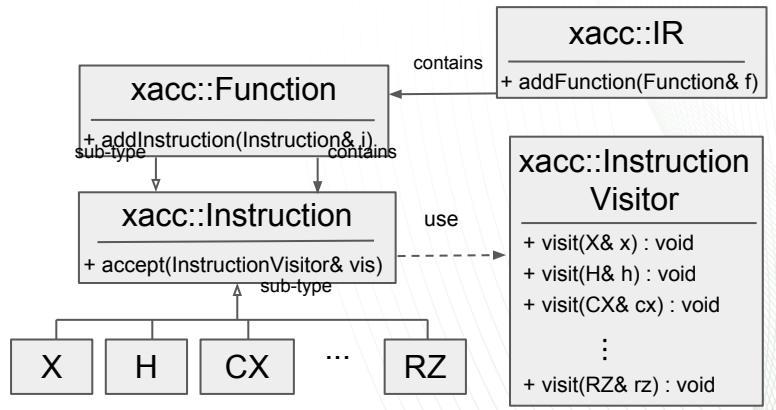
- **D-Wave Compiler** delegates to Antlr for source code parsing
- Provides extension point for minor graph embedding and parameter setting
- Enables **`anneal`** instruction to specify anneal schedule with pause,quench

- Compilers take source code and Accelerator, produce XACC IR
  - Hardware-specific information available at compile time
- Compilers provide source-to-source translation capabilities
  - transpile one hardware instruction set to another
- Extensible Preprocessor mechanism
  - Macro-like functionality

Programming Model - Accelerators

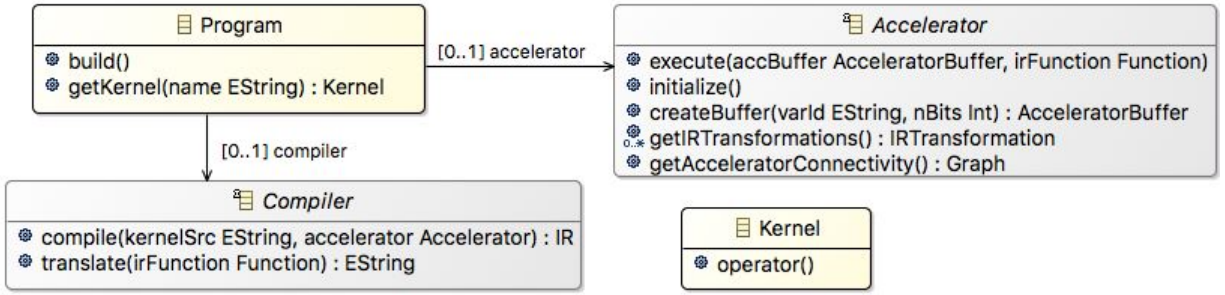
- Accelerators execute XACC IR
- Can be simulators, physical QPUs, remote or local
- Provides **`initialize()`** for costly startup
- Provide hardware-dependent IR Transformations
- Remote Accelerators delegate to remote execution server with CPR CURL API
  - If QPU has REST API, we can target it
- Visitors walk IR, produce native code
  - or apply gates in case of simulation (as done with TNQVM)



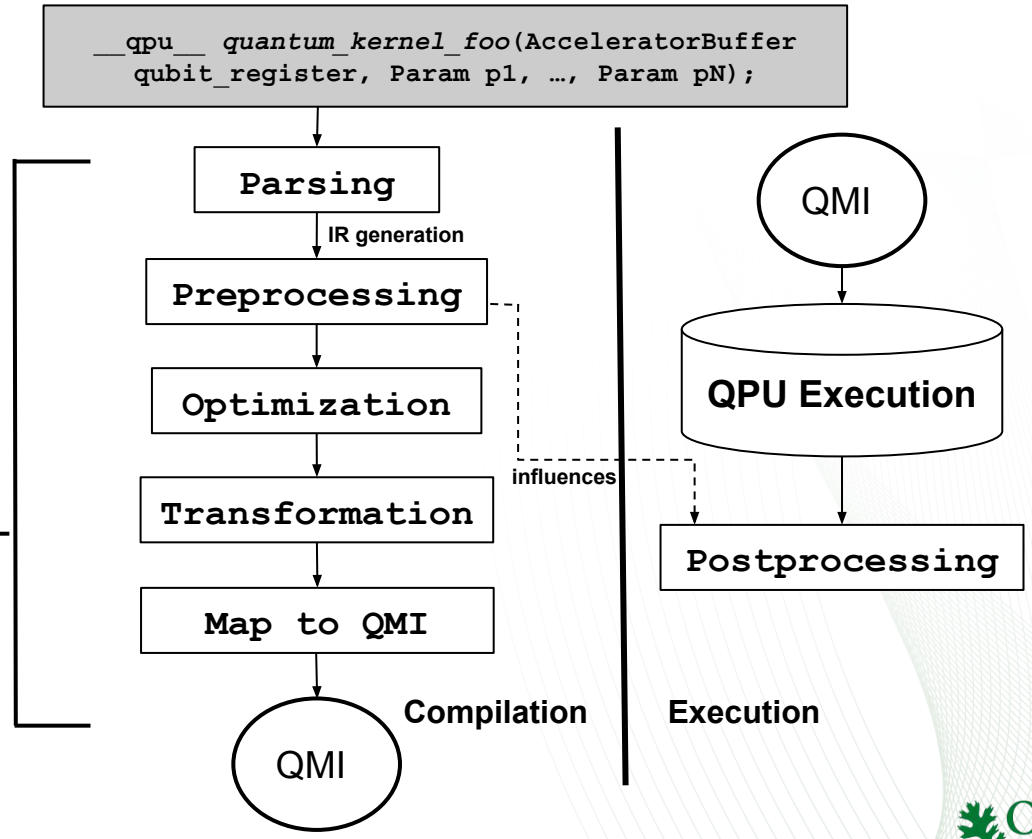


Programming Model - Program

- Programs orchestrate the XACC compilation workflow
  - Execute correct Compiler for the given kernel language
  - Execute IR Preprocessors, Transformations, Optimizations
  - Map to hardware-native QMI
  - Store data post-processors
- Provide an executable lambda (potentially parameterized)




```
// Get reference to the QPU, and allocate a buffer of qubits
auto qpu = xacc::getAccelerator("ibm");
auto buffer = qpu->createBuffer("qreg", 2);

// Create and compile the Program from the kernel
// source code. Get executable Kernel source code
auto kernelSourceCode = "__qpu__ foo(double theta) {....}";
xacc::Program program(qpu, kernelSourceCode);
program.build();
auto kernel = program.getKernel<double>("foo");

// Execute over theta range
for (auto theta : thetas) kernel(buffer, theta);
```



XACC Extensions

XACC Extension Points
Compiler
Preprocessor
Accelerator
Instruction
EmbeddingAlgorithm
ParameterSetter
IRTransformation
IRPreprocessor
AcceleratorBufferPostprocessor
IRGenerator

Extension	Provides
xacc-rigetti	QuilCompiler (leverages ANTLR), RigettiAccelerator
xacc-ibm	OpenQasmCompiler (leverages ANTLR), IBMAccelerator
xacc-dwave	DWQMICompiler (leverages ANTLR and Embeding Algorithm extension point), DWAccelerator
xacc-projectq	ProjectQCompiler (low-level ProjectQ-Qasm transpiler)
xacc-vqe	VQETask, VQEMinimizeBackend, VQEProgram, FermionCompiler, JW and BK IRTransformations, UCCSD IRGenerator
xacc-vqe-fcidump	FCIDumpPreprocessor (map FCIDump format to FermionCompiler)
xacc-vqe-bayesopt	VQEMinimizeBackend implementing bayesian optimization
xacc-cmr	CMR Embedding Algorithm delegating to D-Wave MinorMiner
tnqvm	TNQVM Accelerator (Tensor Network Quantum Virtual Machine), TNQVM Visitor Backend
xacc-atos	ATOSVisitor (mapping IR to ATOS Circuit), ATOSAccelerator

XACC Architecture

What's wrong with the API below?

```python
# Get reference to the QPU, allocate a buffer of qubits
qpu = xacc.getAccelerator('ibm')
buffer = qpu.createBuffer('q',2)

# Create and compile the Program from the kernel
# source code. Get executable kernel
kernelSourceCode = '__qpu__ foo(double theta) {...}'
program = xacc.Program(qpu, kernelSourceCode)
program.build()
kernel = program.getKernel('foo')

# Execute over theta range
[kernel.execute(buffer, [theta]) for theta in thetas]
```

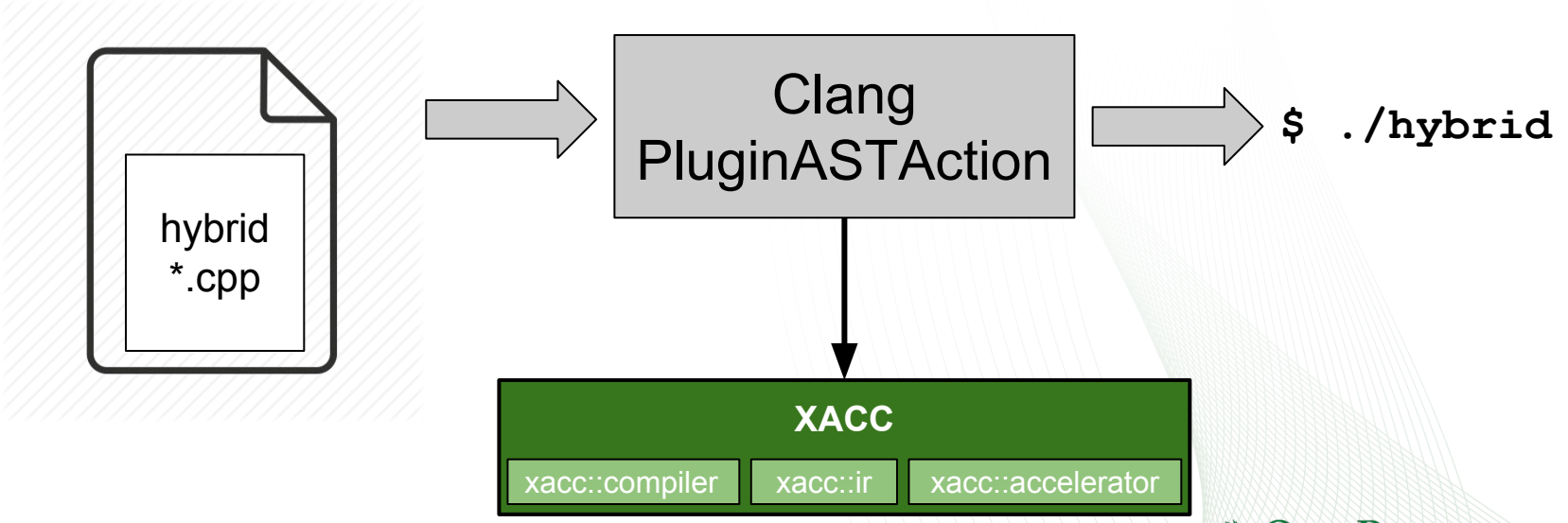
```cpp
// Get reference to the QPU, and allocate a buffer of qubits
auto qpu = xacc::getAccelerator("ibm");
auto buffer = qpu->createBuffer("qreg", 2);

// Create and compile the Program from the kernel
// source code. Get executable Kernel source code
auto kernelSourceCode = "__qpu__ foo(double theta) {...}";
xacc::Program program(qpu, kernelSourceCode);
program.build();
auto kernel = program.getKernel<double>("foo");

// Execute over theta range
for (auto theta : thetas) kernel(buffer, theta);
```

Primarily, kernels provided as strings and too much boilerplate code

Hybrid Quantum-Classical C++ Compiler

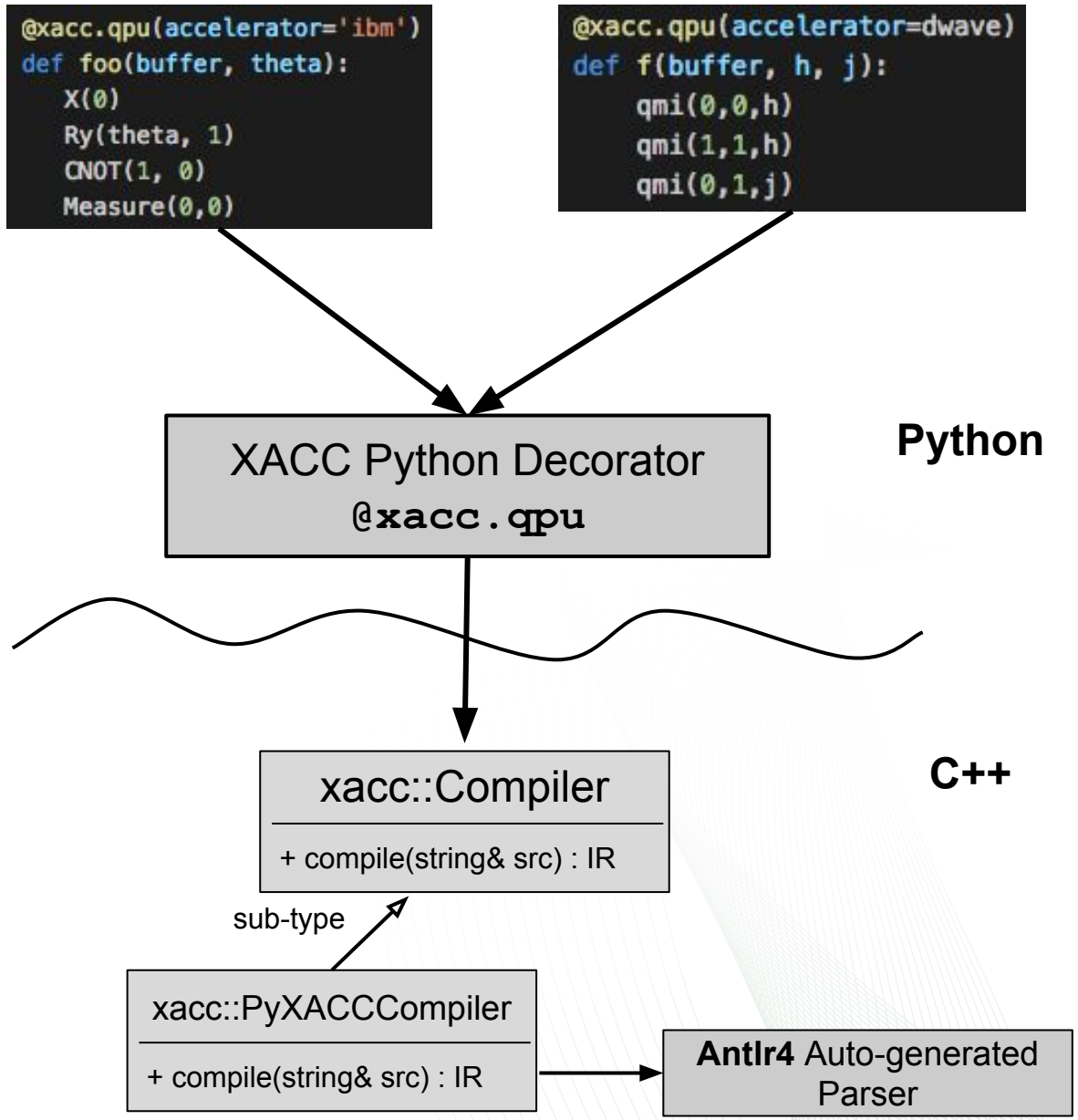
- Run **`xacc`** just like you would **`g++/clang++`**
- Make use of kernel **`__qpu__`** function attribute
- Leverage Clang Plugins or LibTooling
  - Walk clang AST searching for XACC kernels
  - Refactor kernel code to leverage low-level API
- This is something we are currently working on, not available yet.

```
$ cat hybrid_code.cpp
__qpu__ f(AcceleratorBuffer b, double t) {...}
int main() {
    ...
    f(qubits, xacc::pi/2.);
    ...
}
$ xacc hybrid_code.cpp -a ibm -o hybrid
```



XACC Python JIT Compiler

- Goals
  - single-source, so kernel strings
  - unified programming model for available QPUs
  - Single gate+annealing model language
  - IRGenerator instructions
- Requirements
  - language must be Pythonic (to pass interpreter)
  - Users must annotate python functions to indicate intention for QPU execution
  - typical xacc kernel requirements (buffer is first argument)
- How's it done?
  - Python Inspect Module (get source string at runtime)
  - Decorator compiles with XACC PyXACCCompiler, runs usual XACC execution API



XACC Python JIT Compiler and the IRGenerator

- Goals
  - Provide a mechanism for programming complex circuits / QUBOs parameterized by high-level user input
- How's it done?
  - PyXACCCompiler interprets custom **`xacc()`** instruction
    - first argument is the name of the IRGenerator
    - following arguments are the list of user input

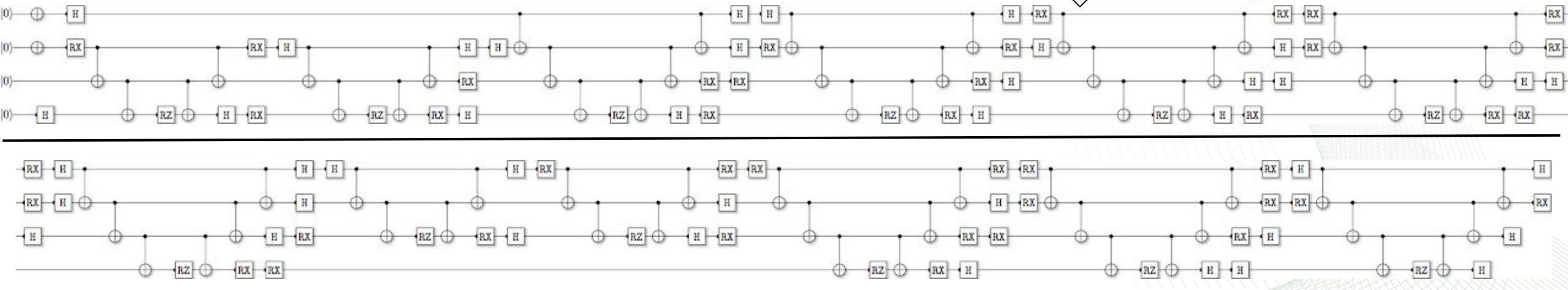
```
# Define the function we'd like to
# off-load to the QPU, here
# we're using a parameterized Factoring
# IR Generator
@xacc.qpu(accelerator='dwave')
def factor15(buffer):
    xacc(dwave-factoring, n=15)
```

produces

```
0 0 164.25;
1 1 168.25;
2 2 4;
3 3 -256.5;
0 1 100.25;
0 2 -14;
0 3 -224.5;
1 2 4;
1 3 -224.5;
2 3 32;
```

```
@xacc.qpu(accelerator=qpu) # or ibm, rigetti, etc...
def foo(buffer, t0, t1):
   xacc(uccsd, n_qubits=4, n_electrons=2)
   Measure(0,0)
```

produces



Python JIT Examples for D-Wave

```python
import xacc

xacc.Initialize()

# Get access to D-Wave QPU and
# allocate some qubits
dwave = xacc.getAccelerator('dwave')
qubits = dwave.createBuffer('q')

# Define the function we'd like to
# off-load to the QPU, here
# we're using a the QMI low-level language
@xacc.qpu(accelerator=dwave)
def f(buffer, h, j):
    qmi(0,0,h)
    qmi(1,1,h)
    qmi(0,1,j)

# Execute on D-Wave
f(qubits, 1., 2.)

# Print the buffer, this displays
# solutions and energies
print(qubits)

xacc.Finalize()
```

Get reference to the D-Wave Accelerator

Define the code you want to run, annotate it

Execute and post-process the results

```python
import xacc

xacc.Initialize()

# Get access to D-Wave QPU and
# allocate some qubits
dwave = xacc.getAccelerator('dwave')
buffer = dwave.createBuffer('q')

# Define the function we'd like to
# off-load to the QPU, here
# we're using a parameterized Factoring
# IR Generator
@xacc.qpu(accelerator='dwave')
def factor15(buffer):
    xacc(dwave-factoring, n=15)

# Factor 15 on the D-Wave
factor15(buffer)

# We have solutions as 0s and 1s
# decode that into our factors
xacc.analyzeBuffer(buffer)

# Print the factors
factors = buffer.getInformation('analysis-results')
xacc.info('Factors = ' + str(factors))

xacc.Finalize()
```

Thanks! Check out the project at:

https://github.com/eclipse/xacc

https://xacc.readthedocs.io

https://arxiv.org/abs/1710.01794

https://hub.docker.com/r/xacc

Email: xacc-dev@eclipse.org, mccaskeyaj@ornl.gov

Funded by ORNL LDRD, DOE Testbed Pathfinder, DOE Quantum Algorithms Teams, DOE Early Career Research Program

Looking ahead...

Quantum Parallel Processing

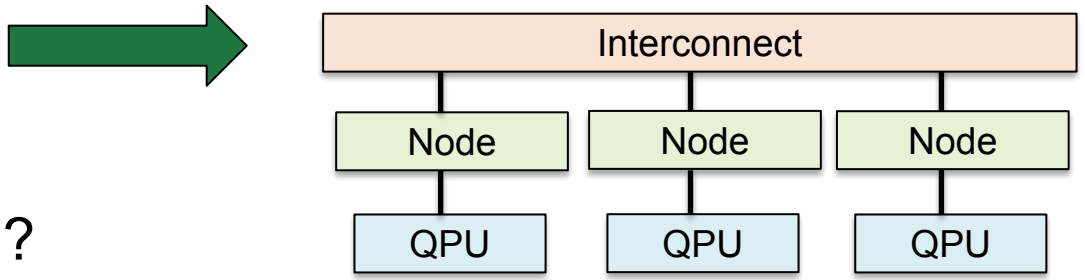
Hybrid ahead-of-time C++ compilation

Improved Python Integration

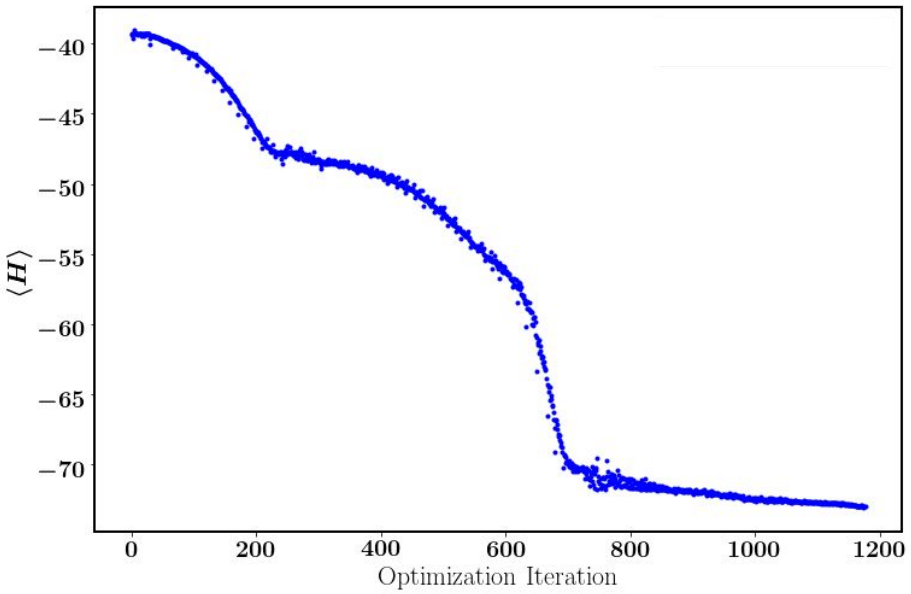
An Overall Integration Framework for Quantum Computing

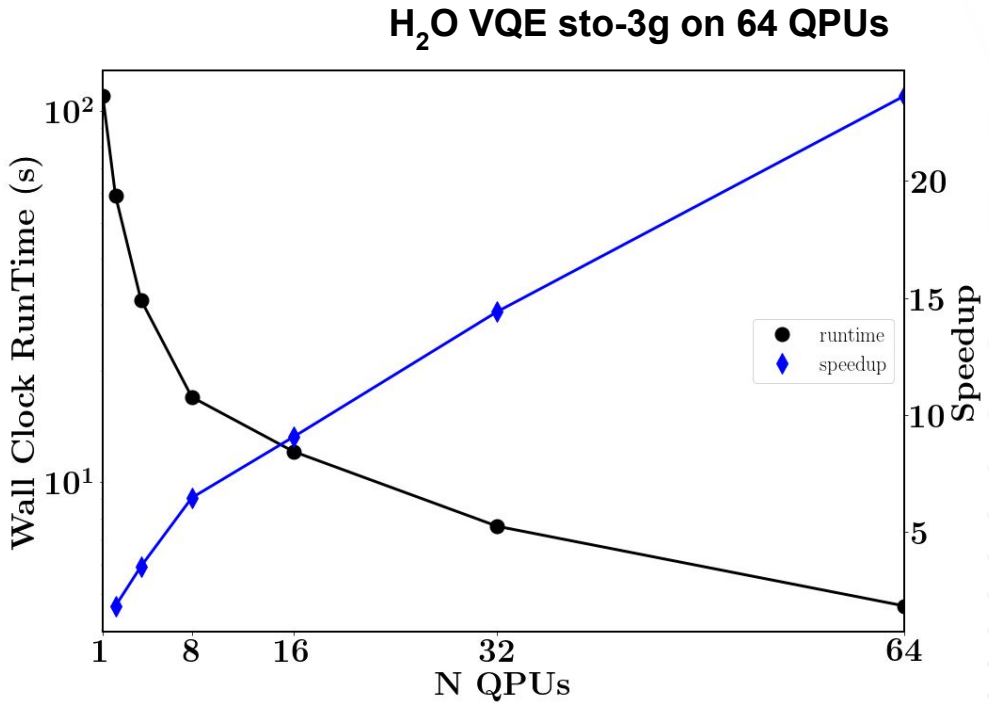
Future Quantum Parallel Processing with XACC

What if we had a compute system like this?



- How would you program it?
  - Popular MPI+X (X=OpenMP, CUDA, etc)
  - How about **MPI+XACC** programming Model?
  - **Shared vs Distributed** Memory QPP
- We've demonstrated this via an embarrassingly parallel distributed-memory VQE implementation





- **1086** Terms, 14 qubits
- **64 virtual QPUs**
- **28 VQE Parameters** - ~1200 Nelder Mead optimization steps
- **MPI All_Reduce** call to sum energies across cores
- Speedup and wall clock run-time based on simulation wall-clock time, rescale for QPU gate execution times

OAK RIDGE National Laboratory

Improved Python Integration

- Learn from CUDA Numba
- https://github.com/eclipse/xacc/issues/37
- Move away from string based kernel definitions
- Leverage
  - Python Decorators
  - Inspect Module
  - Custom XACC IR Antlr Grammar and Compiler
- Python users just annotate functions wrapping QPU code with `@qpu` and indicate target accelerator

framework code

```
import inspect

class qpu(object):
    def __init__(self, *args, **kwargs):
        self.args = args
        self.kwargs = kwargs
        return
    def __call__(self, f):
        def wrapped_f(*args, **kwargs):
            src = inspect.getsource(f)
            qpu = xacc.getAccelerator(self.kwargs['accelerator'])
            buf = qpu.createBuffer('q')
            program = xacc.Program(qpu, src)
            program.build()
            kernel = program.getKernels()[0]
            kernel.execute(buf, *args)
            return buf
        return wrapped_f
```

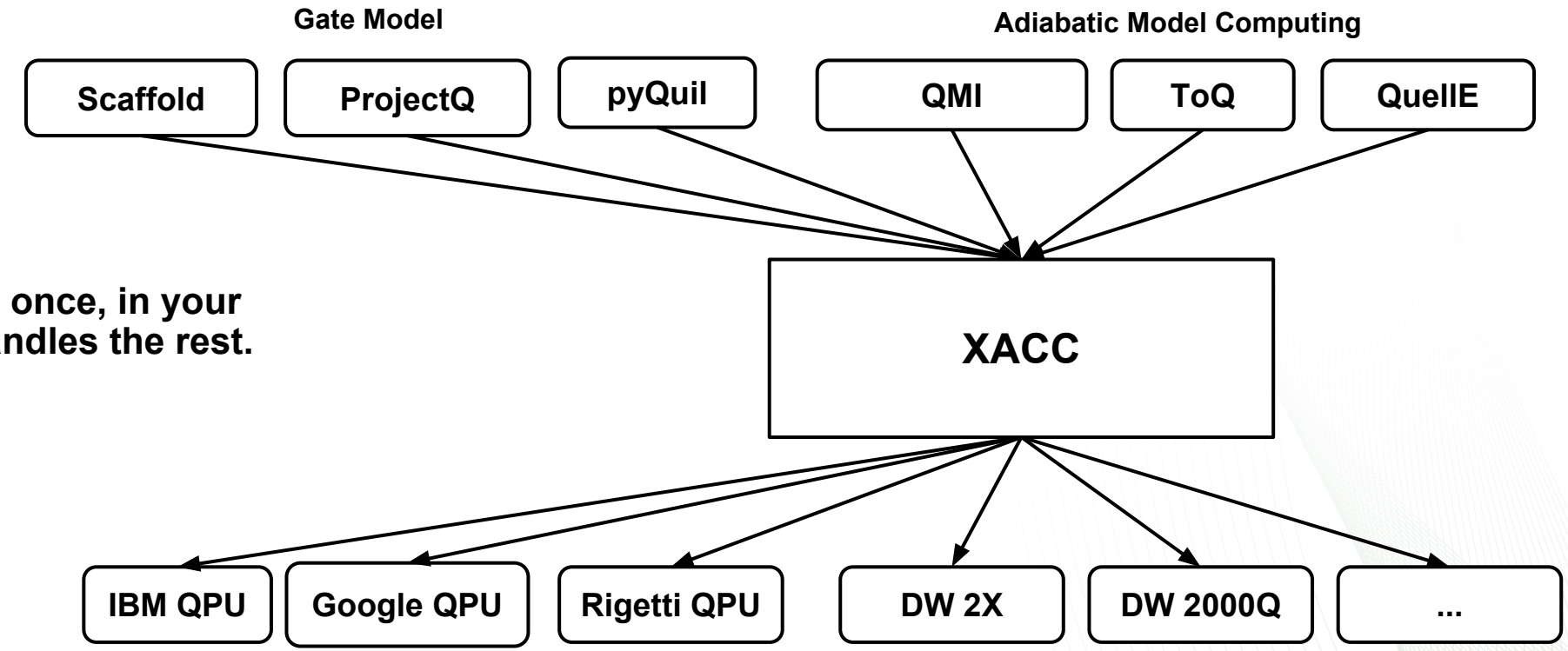
user code

```
@qpu(accelerator='tnqvm') # or ibm, rigetti, etc...
def foo(theta):
   X(0)
   RY(theta, 1)
   CNOT(1, 0)
   Measure(0,0)
   return

results = foo(np.pi/3.)
print(results.getMeasurementBitStrings())
```

XACC Enabling Integration and Interoperability

- Language and Hardware interoperability
  - *"I prototyped this in X but want to run on Y"* - I've heard this a lot
  - Benchmarking will suffer without better software integration

Program quantum code once, in your language, and XACC handles the rest.

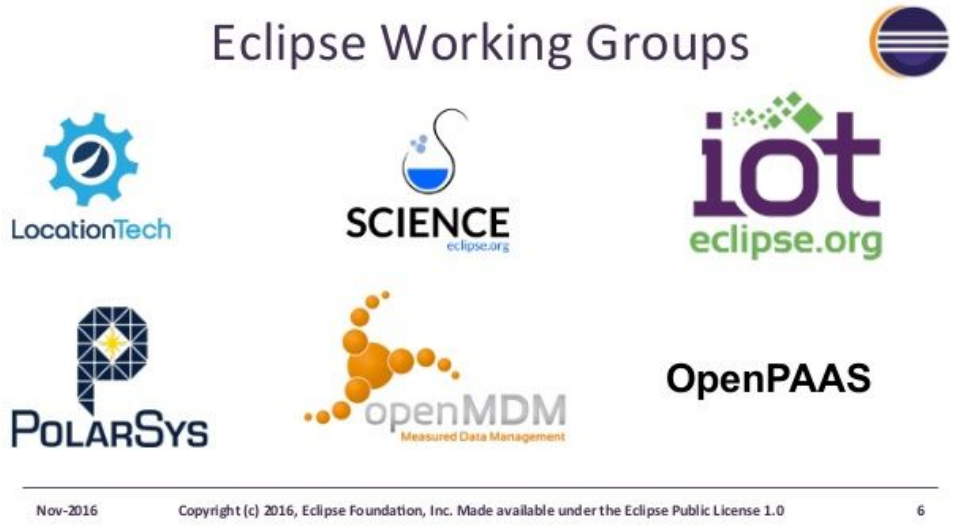


Eclipse XACC

- **Eclipse Foundation**
  - International organization
  - Eclipse IDE, 350 other open source projects
  - Working Group governance structure
    - Science, Location, IoT, etc...
  - IP tracking and management
  - IT Infrastructure
    - CI, GitHub, websites, email lists, etc…
  - ORNL is a member
- **XACC joined the Eclipse Foundation in 2017**
  - First C++ and quantum computing software project for Eclipse
  - Goal is research…
- The beginnings of a **Quantum Computing Software Working Group**
  - Quarterly virtual meetings (when possible)
  - Email mccaskeyaj@ornl.gov if interested in participating

OAK RIDGE National Laboratory

eclipse



Libraries and Applications built on XACC - XACC-VQE

```
$ xacc-vqe -f observable.hpp -t vqe -a ansatz.hpp
```

- XACC designed to serve as foundation for libraries and applications.
- Application built on XACC for VQE
  - https://github.com/ornl-qci/xacc-vqe
- Library of compiler routines for VQE problems
  - **`pragma`** for observable coefficients
- Provides **`VQEProgram`** and **`FermionCompiler`** extensions
- **`xacc-vqe`** command-line executable
- Task-based architecture
  - compute-energy, vqe, diagonalize, etc.

```
#pragma vqe-coefficient 5.906709445
__qpu__ idTerm(AcceleratorBuffer b) {
}
#pragma vqe-coefficient -2.1433
__qpu__ x0x1(AcceleratorBuffer b) {
H 0
H 1
MEASURE 0 [0]
MEASURE 1 [1]
}
#pragma vqe-coefficient -2.1433
__qpu__ y0y1(AcceleratorBuffer b) {
RX(1.57078) 0
RX(1.57078) 1
MEASURE 0 [0]
MEASURE 1 [1]
}
#pragma vqe-coefficient .21829
__qpu__ z0(AcceleratorBuffer b) {
MEASURE 0 [0]
}
```

```
__qpu__ ansatz(AcceleratorBuffer b,
               double t0) {
X 0
RY(t0) 1
CNOT 1 0
}
```

Higher-level programming and error mitigation

Deuteron binding energy via standard XACC Python API

```
import xaccvqe as vqe
from xaccvqe import qpu, PauliOperator
import xacc
import numpy as np

xacc.Initialize()
tnqvm = xacc.getAccelerator('tnqvm')
buffer = tnqvm.createBuffer('q',2)

ham = PauliOperator(5.906709445) + \
    PauliOperator({0:'X',1:'X'}, -2.1433) + \
    PauliOperator({0:'Y',1:'Y'}, -2.1433) + \
    PauliOperator({0:'Z'}, .21829) + \
    PauliOperator({1:'Z'}, -6.125)

@qpu.vqe(accelerator=tnqvm, observable=ham, optimizer='vqe-bayesopt', opt_params={'tol':1e-2})
def ansatz(buffer, initialTheta):
   X(0)
   Ry(initialTheta, 1)
   CNOT(1, 0)

ansatz(buffer, .5)

print(buffer.getInformation('vqe-energy'), buffer.getInformation('vqe-angles'))

xacc.Finalize()
```

Of note with this code sample

- Unified VQE API
- Problem programming at high-level, i.e. define Hamiltonian
- Custom ansatz generation using Python JIT
- Specify target Accelerator - IBM, Rigetti, TNQVM
- Built-in error mitigation (readout, arxiv:1612.02058, extrap. arxiv:1611.09301)
- Specify mapping to physical qubits