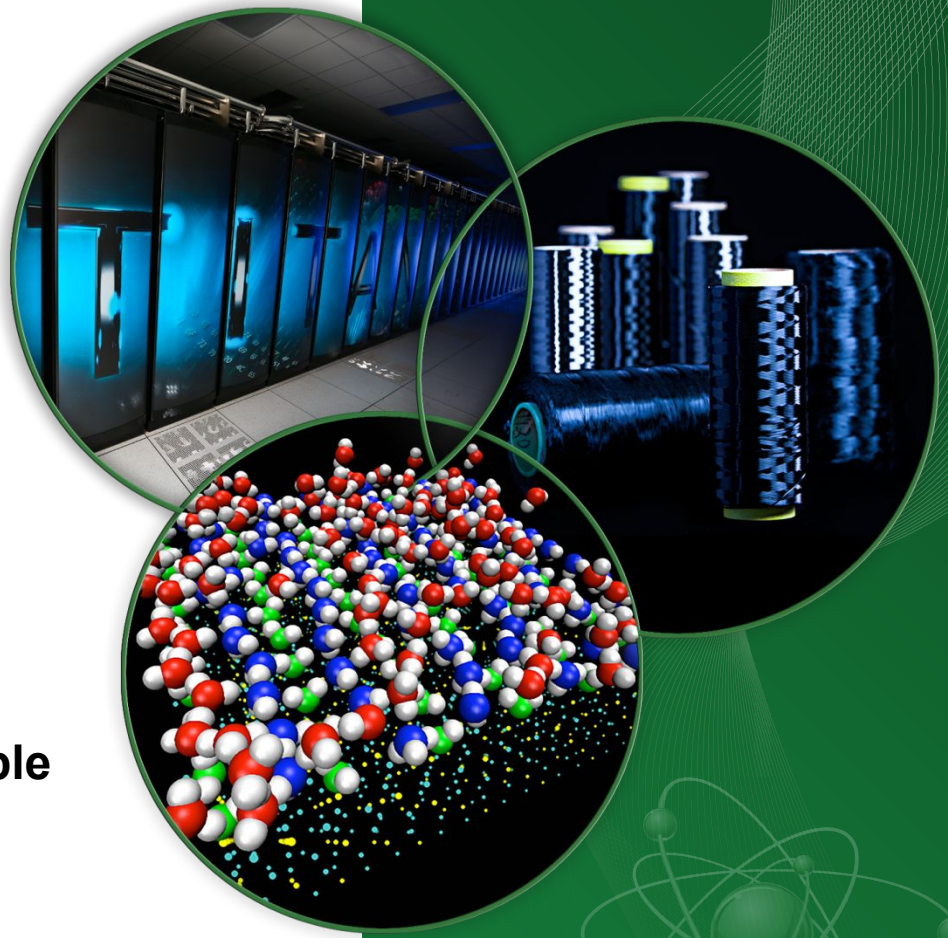
XACC: Enabling Quantum Acceleration in Scientific High-Performance Computing



Alex McCaskey, Eugene Dumitrescu,
Dmitry Liakh, Keith Britt, Travis Humble

D-Wave Qubits Conference

Sep 2017

ORNL is managed by UT-Battelle
for the US Department of Energy

OAK RIDGE National Laboratory

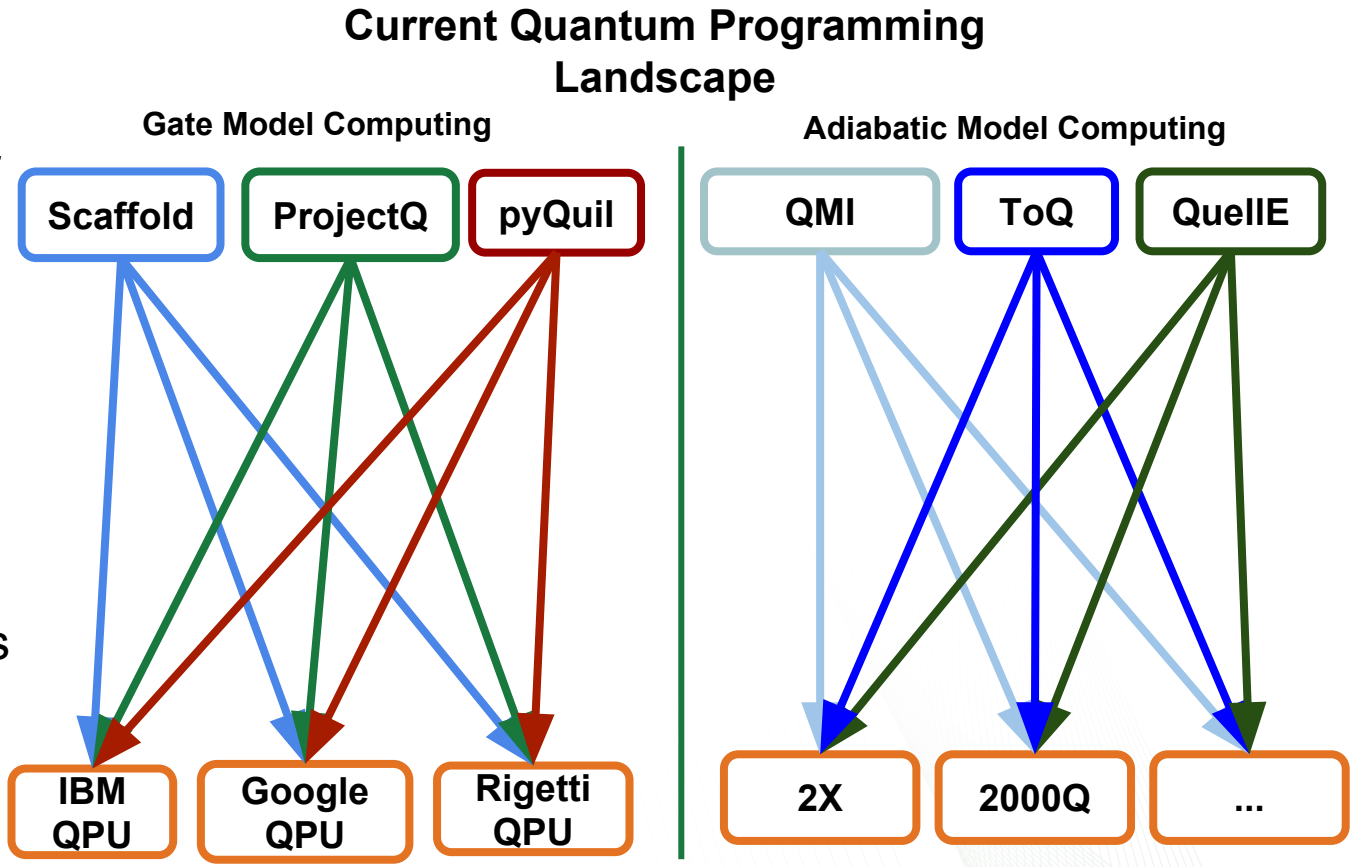
Outline of this talk...

- Purpose of XACC
- The Programming Model
  - Kernels
  - IR
  - Compilers
  - Accelerators
- Examples
  - Factoring 15=3x5
  - General Factoring App
- Where are we going? A look at 5 years from now

Purpose/Goals of XACC

How do we use near term quantum computing? How do we program it?

- Current Problems:
  - Many QPLs and many QPUs
  - Massive amount of work to map QPL to QPU
- DOE Test bed may have multiple QPUs attached to classical HPC cluster
- Nothing familiar here to current domain computational scientists
- None targets HPC environments



Our solution - XACC Programming Model

Treat near-term QPUs as accelerators within a larger HPC environment.

Familiar API and Programming model

- OpenCL-like
- LLVM-like

Current Quantum Programming Landscape

Gate Model Computing

Scaffold | **ProjectQ** | **pyQuil**

Adiabatic Model Computing

QMI | **ToQ** | **QuelIE**

XACC - Heterogeneous CPU-QPU Programming Model

IBM QPU | **Google QPU** | **Rigetti QPU** | **2X** | **2000Q** | **...**

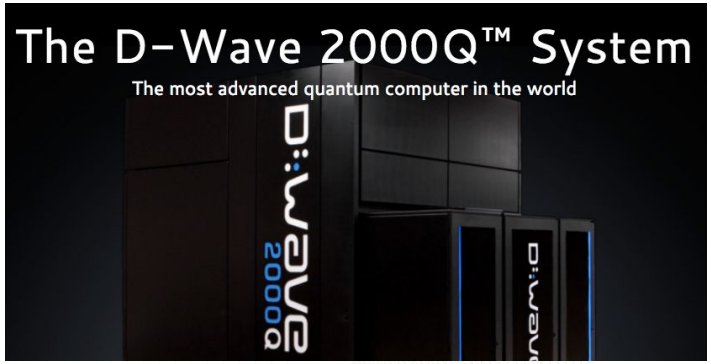
Program quantum code once, in your language, and XACC handles the rest.

XACC At a Glance

eclipse

- Open source, up on Github at https://github.com/ORNL-QCI/xacc (soon to be github.com/eclipse/xacc)
- Just joined the Eclipse Foundation
- Primarily written in C++14
- *Plugin infrastructure* for easy extensibility
- Integration with Rigetti QVM and 2 qubit QPU, Rigetti Quil Compiler
- Scaffold Gate Model QC Compiler integration
- Just added D-Wave QPU and Compiler integration - focus of this talk!

rigetti



OAK RIDGE National Laboratory

XACC Plugins

Framework enables language and hardware agnostic quantum programming

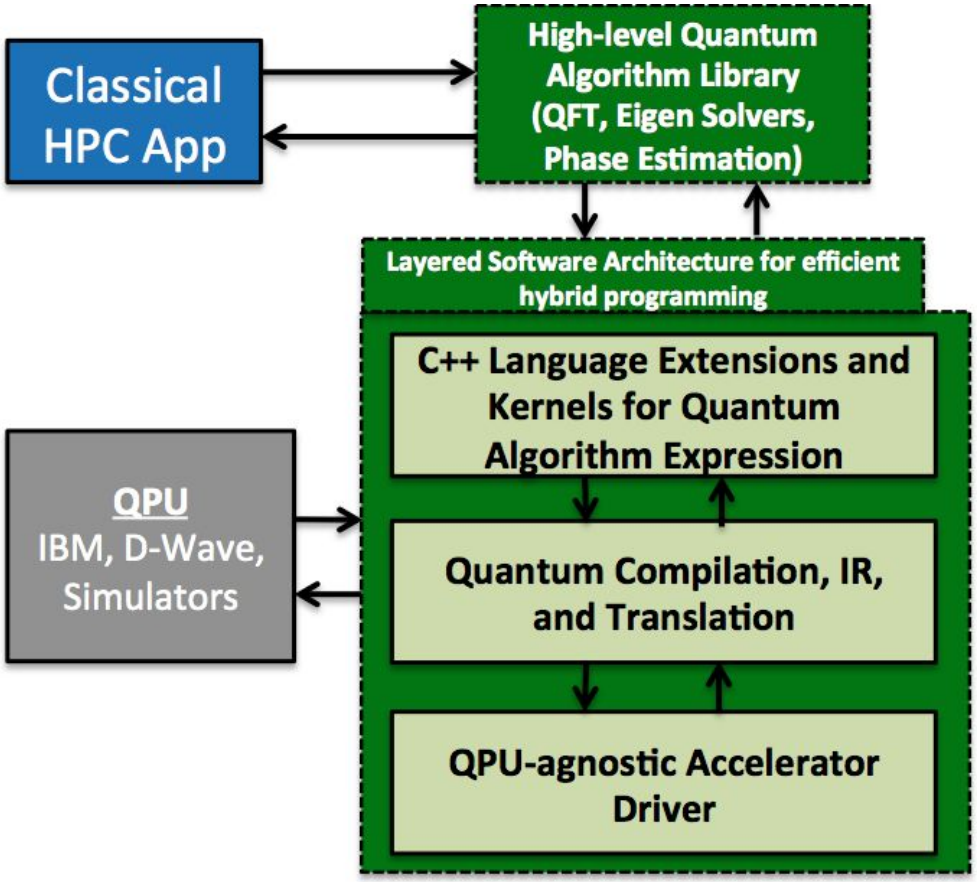
Plugin	Provides	Repository
xacc-dwave	DWAccelerator, DWQMICompiler	https://github.com/ornl-qci/xacc-dwave
xacc-rigetti	RigettiAccelerator, QuilCompiler	https://github.com/ornl-qci/xacc-rigetti
xacc-python	XACC Python Bindings	https://github.com/ornl-qci/xacc-python
xacc-ibm	IBMAccelerator	https://github.com/ornl-qci/xacc-ibm
xacc-projectq	ProjectQCompiler	https://github.com/ornl-qci/xacc-projectq
tqnvm	TNQVMAccelerator (tensor network simulator)	https://github.com/ornl-qci/tqnvm
xacc-vqe	FermionCompiler, general Variational Quantum Eigensolver	https://github.com/ornl-qci/xacc-vqe

Installing new plugins is easy:

```
$ xacc-install-plugins.py -p xacc-dwave
```

XACC Concepts and Layered Architecture

XACC - Heterogeneous CPU-QPU Programming Model



Key Concepts:

1. Quantum Kernel
2. Quantum Compiler
3. QPU Intermediate Representation
4. Quantum Accelerator and Accelerator Buffer

XACC Model

Quantum Kernels

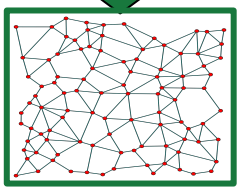
XACC - Heterogeneous CPU-QPU Programming Model

```
__qpu__ quantum_kernel_foo(AcceleratorBuffer qubit_register,
                           Param p1, …, Param pN);
```

Circuit

$$U_{kernel}|\psi_{register}\rangle$$

QUBO



Example D-Wave QMI Kernel

```
__qpu__ factor15() {
        0 0 20;
        1 1 50;
        2 2 60;
        4 4 50;
        5 5 60;
        6 6 -160;
        1 4 -1000;
        2 5 -1000;
        0 4 -14;
        0 5 -12;
        0 6 32;
        1 5 68;
        1 6 -128;
        2 6 -128;
}
```

`__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.

XACC Model

Quantum Kernels

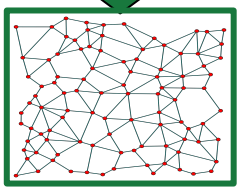
XACC - Heterogeneous CPU-QPU Programming Model

```
__qpu__ quantum_kernel_foo(AcceleratorBuffer qubit_register,
                           Param p1, …, Param pN);
```

Circuit

$$U_{kernel}|\psi_{register}\rangle$$

QUBO



Example Gate Model Kernel written in Scaffold

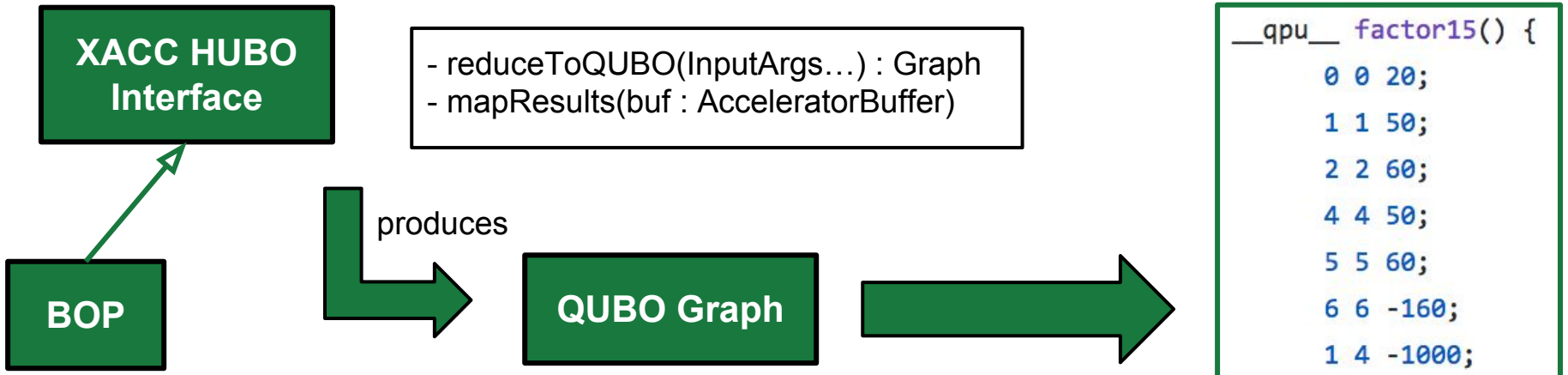
```
__qpu__ teleport (qbit& qreg) {
    X(qreg[0]);
    H(qreg[1]);
    CNOT(qreg[1],qreg[2]);
    CNOT(qreg[0],qreg[1]);
    H(qreg[0]);
    cbit c1 = MeasZ(qreg[0]);
    cbit c2 = MeasZ(qreg[1]);
    if(c1 == 1) Z(qreg[2]);
    if(c2 == 1) X(qreg[2]);
}
```

`__qpu__ 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.

XACC Model

XACC - Heterogeneous CPU-QPU Programming Model

Extensible way to generate D-Wave Kernels - HUBO, high-order unconstrained binary optimization



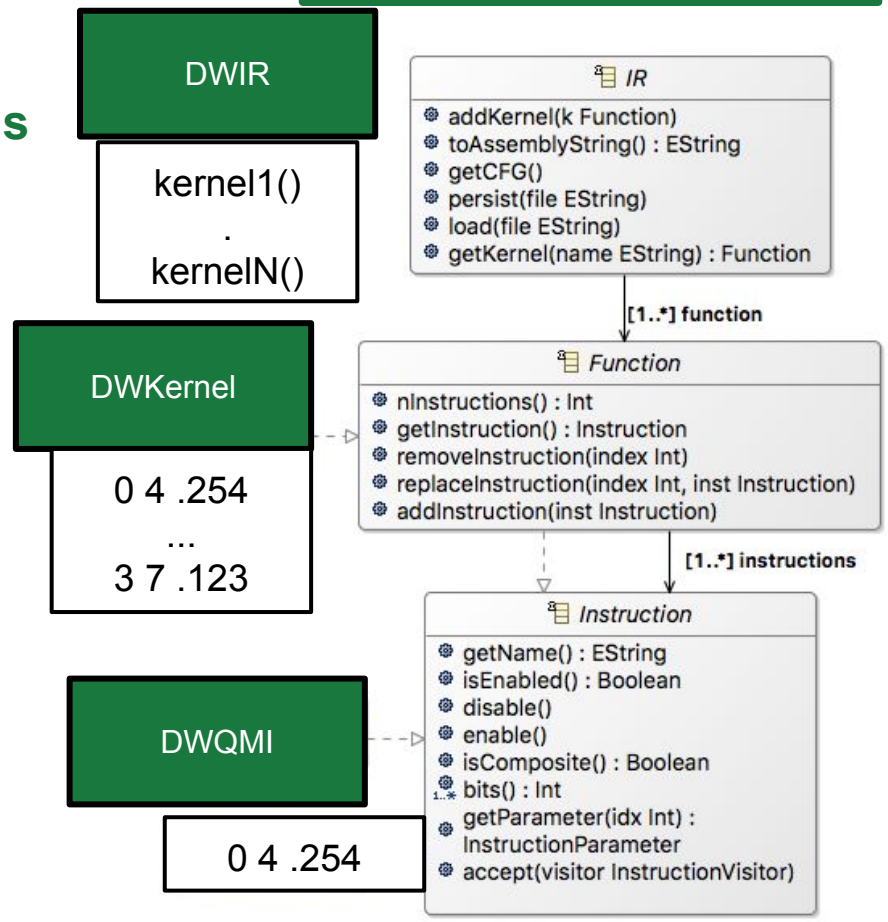

```
__qpu__ factor15() {
    0 0 20;
    1 1 50;
    2 2 60;
    4 4 50;
    5 5 60;
    6 6 -160;
    1 4 -1000;
    2 5 -1000;
    0 4 -14;
    0 5 -12;
    0 6 32;
    1 5 68;
    1 6 -128;
    2 6 -128;
}
```

Output kernel of FactoringHUBO for 15 = 3*5

XACC Model

Quantum Intermediate Representation (QIR) Specification

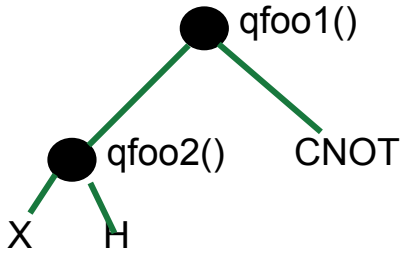
XACC - Heterogeneous CPU-QPU Programming Model



Key insight: Provide common representation to map N QPLs to N QPUs

- Lowest level of IR is the Instruction interface
- Functions are composed of Instructions - Composite Pattern, n-ary tree
- IR composed of Functions
- 4-fold IR characteristics
- Instructions can be parameterized with Boost variant type.
- Instructions can be visited
- IR Transformation and Optimization infrastructure

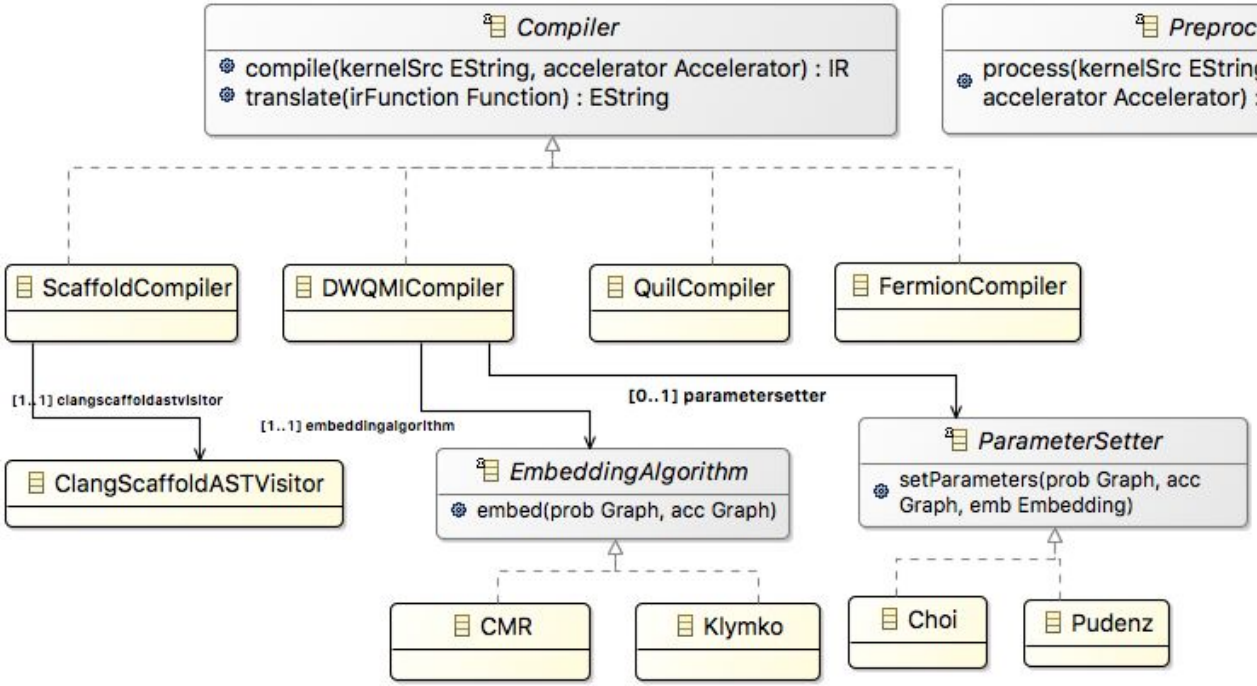
IR models a tree, and we walk that tree to perform translations, optimizations, and executions!



XACC Model

Quantum Compiler Specification

XACC - Heterogeneous CPU-QPU Programming Model



Compilers take source code and Accelerator, produce XACC IR

Compilers provide source-to-source translation capabilities

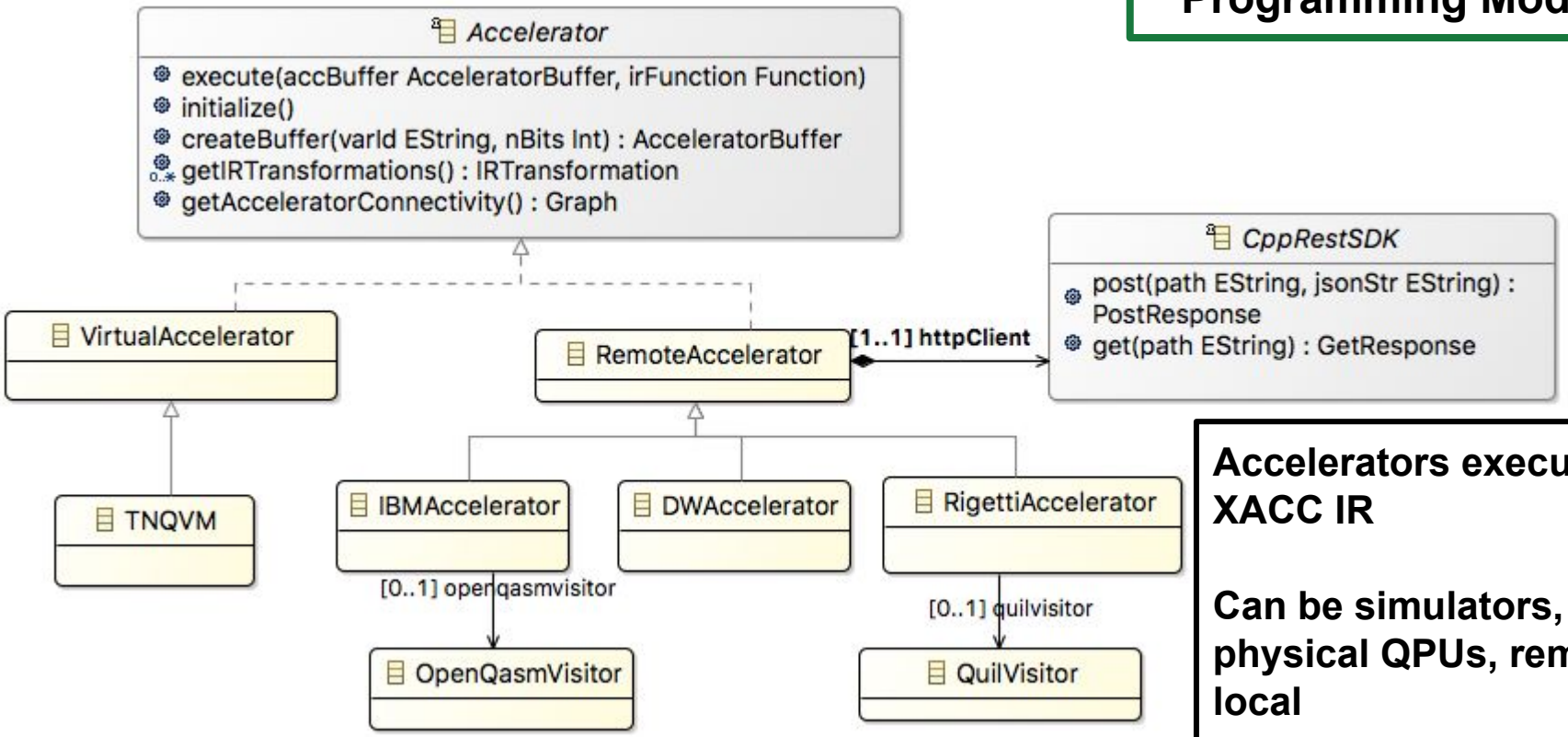
Extensible Preprocessor mechanism

DWQMICompiler is extensible for Embedding Algorithms and Parameter Setting

XACC Model

Quantum Accelerators

XACC - Heterogeneous CPU-QPU Programming Model



Accelerators provide qubit resources - AcceleratorBuffers

Accelerators can provide IR Transformations to make code amenable for execution on device

Accelerators execute XACC IR

Can be simulators, physical QPUs, remote or local

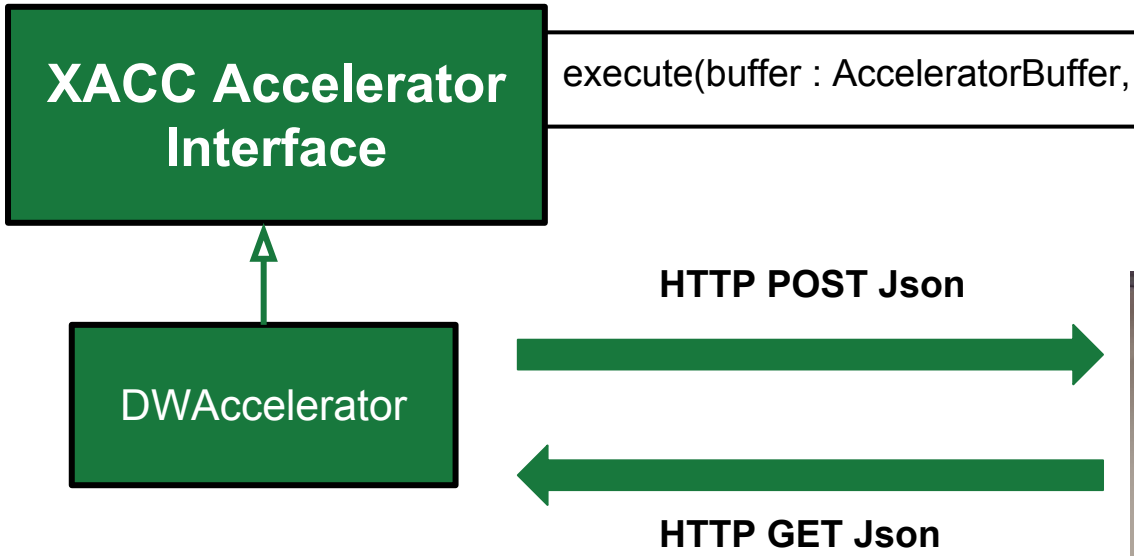
Remote Accelerators operate with HTTP Rest Client

Visitors walk IR, produce Accelerator-specific code

XACC Model

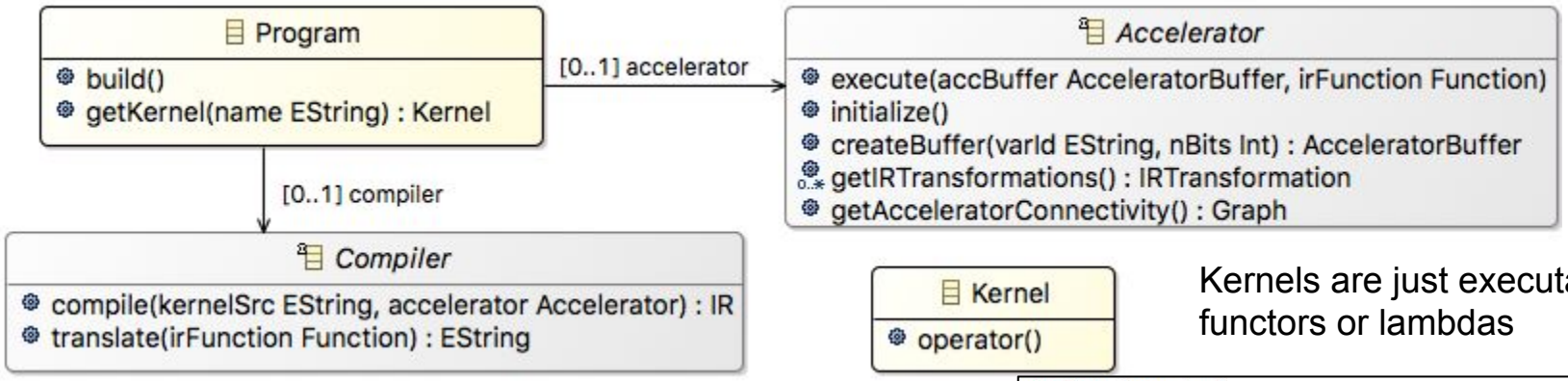
Quantum Accelerator - DWAccelerator

XACC - Heterogeneous CPU-QPU Programming Model



1. Get h_range and j_range for solver
2. Take QMI from IR Function and normalize based on ranges
3. Build up Json post string
4. HTTP Post to http://qubist.dwavesys.com/sapi/problems
5. HTTP Get Results as Json
6. Postprocess and update AcceleratorBuffer with results

Programs, Execution Workflow, and API



Kernels are just executable functors or lambdas

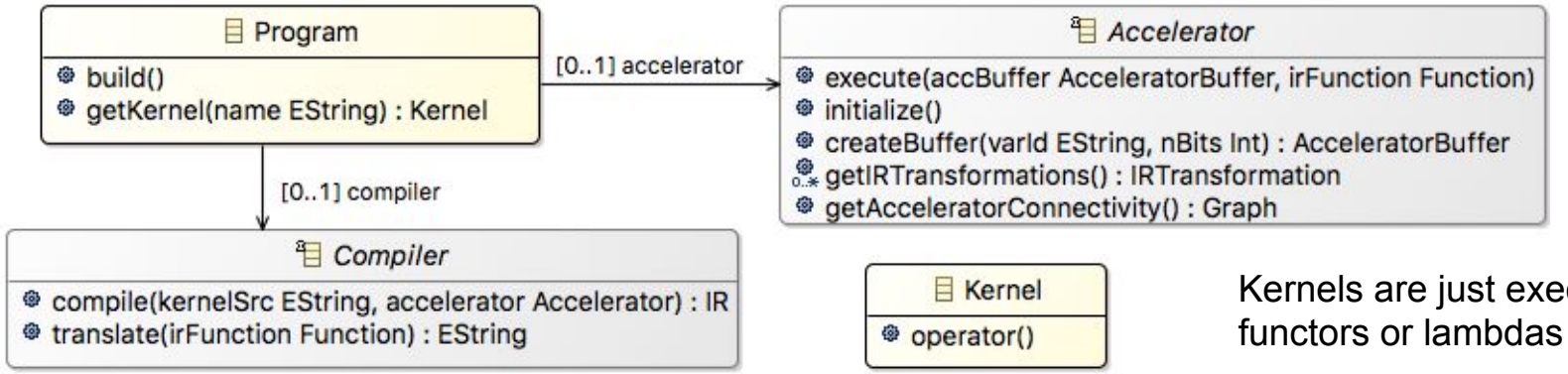
```
// Quantum Kernel executing teleportation of
// qubit state to another.
__qpu__ teleport (qbit& qreg) {
    X(qreg[0]);
    H(qreg[1]);
    CNOT(qreg[1],qreg[2]);
    CNOT(qreg[0],qreg[1]);
    H(qreg[0]);
    cbit c1 = MeasZ(qreg[0]);
    cbit c2 = MeasZ(qreg[1]);
    if(c1 == 1) Z(qreg[2]);
    if(c2 == 1) X(qreg[2]);
}
```

teleport.hpp

```
#include "XACC.hpp"
#include "teleport.hpp"

int main (int argc, char** argv) {
        // Initialize the XACC Framework
        xacc::Initialize(argc, argv);

        // Create a reference to the Rigetti
        // QPU at api.rigetti.com/qvm
        auto qpu = xacc::getAccelerator("rigetti");

        // Allocate a register of 3 qubits
        auto qubitReg = qpu->createBuffer("qreg", 3);

        // Create a Program
        xacc::Program program(qpu, src);

        // Request the quantum kernel representing
        // the above source code
        auto teleport = program.getKernel("teleport");

        // Execute!
        teleport(qubitReg);
}
```

teleport.cpp

Programs, Execution Workflow, and API



Kernels are just executable functors or lambdas

```
__qpu__ factor15() {
      0 0 20;
      1 1 50;
      2 2 60;
      4 4 50;
      5 5 60;
      6 6 -160;
      1 4 -1000;
      2 5 -1000;
      0 4 -14;
      0 5 -12;
      0 6 32;
      1 5 68;
      1 6 -128;
      2 6 -128;
}
```

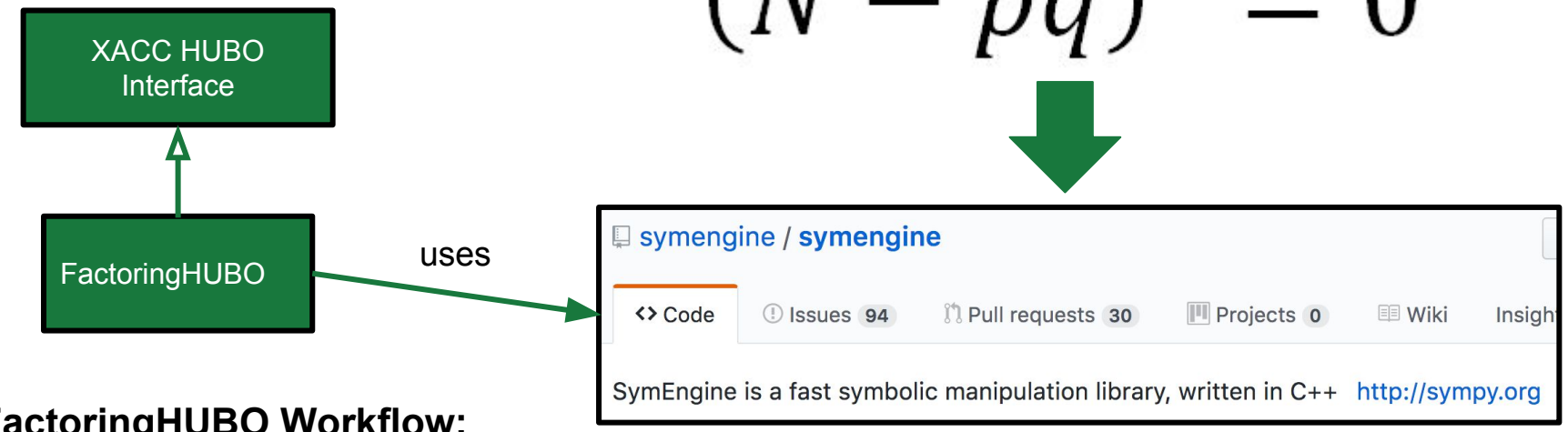
factor15.hpp

```
#include "XACC.hpp"
#include "factor15.hpp"

int main (int argc, char** argv) {
        // Initialize the XACC Framework
        xacc::Initialize(argc, argv);

        // Create a reference to the D-Wave
        // QPU at Qubist Server URL
        auto qpu = xacc::getAccelerator("dwave");

        // Create AcceleratorBuffer representing all
        // D-Wave qubits
        auto qubitReg = qpu->createBuffer("qreg");

        // Create a Program
        xacc::Program program(qpu, src);

        // Request the quantum kernel representing
        // the above source code
        auto factoring15 = program.getKernel("factoring15");

        // Execute!
        factoring15(qubitReg);

        // Finalize
        xacc::Finalize();
}
```

factor15.cpp

General Factoring Application

$$(N - pq)^2 = 0$$



FactoringHUBO Workflow:

1. **Construct symbolic algebra for 0=(N-pq)**2 with SymEngine**
2. **Manipulate expression and reduce to quadratic form**
3. **Convert to Ising form**
4. **Map expression to Qubo xacc::DWGraph**

We also implemented an Embedding Algorithm extension for this work - a wrapper for the DW Sapi findEmbedding

Factoring Application

How it looks in code - the XACC API, leveraging D-Wave Accelerator, HUBO, and Compiler implementations

```
int main() {
        xacc::Initialize(argc, argv);
        FactoringHUBO factoring;
        auto factoringQubo = factoring.reduceToQubo(
                        std::vector<xacc::InstructionParameter> {
                                        xacc::InstructionParameter(N) });
        auto xaccKernelSrcStr = factoringQubo->toKernelSource(kernelName);
        auto qpu = xacc::getAccelerator("dwave");
        auto buffer = qpu->createBuffer("qubits");
        xacc::Program program(qpu, xaccKernelSrcStr);
        auto factoringKernel = program.getKernel(kernelName);
        factoringKernel(buffer);
        auto factors = factoring.mapResults(buffer);
        XACCInfo("Factors were " + std::to_string(factors.first) + " and " + std::to_string(factors.second));
        xacc::Finalize();
}
```

Where are we going with XACC? The next 5 years…

- **Primary AQC programming bottleneck - compilation**
  - Need to streamline minor graph embedding workflow
  - Compile once, run parameterized executable
- **In the next year, XACC will provide a static, ahead-of-time compiler.**
  - Search AST for __qpu__ annotations, compile kernel with appropriate Compiler

```
$ ls
   factoring15.hpp factoring15.cpp
$
$ xacc -I. factoring15.cpp -o factoring15
$ ./factoring15
   [xacc] Factors were 3 and 5
```

The End

Questions?

Special thanks to Eugene Dumitrescu, Dmitry Liakh, Mengsu Chen, and Travis Humble.