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.
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)

- 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

ORNL Programming Requirements

DOE QAT - Heterogeneous Digital-Analog Quantum Dynamics Simulation (HDAQQDS)
XACC - Compiler Framework for Quantum Computing

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

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

- **Open-source at** [https://github.com/eclipse/xacc](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**

```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) {...}
```

- **Compiler Extension Point** (Map high-level kernels to IR)
- **IR Transformations Extension Point**
- **Accelerator Extension Point**

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

Key architectural design: service-oriented, modular, extensible

XACC Core
- Kernels
- Compilers
- Intermediate Representation (IR)
- IR Transformations
- Accelerator Buffer
- Accelerators

Libraries and Applications
- VQE
- RBM
- QML
- etc..

Third Party Dependencies
- CppMicroServices
- Antlr
- Eigen
- Cpr - CURL API

XACC provides a base foundation for future libraries/applications

- CppMicroServices
  - [GitHub](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?

```c++
# 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
for (auto theta : thetas) kernel(buffer, theta)
```

```c++
// 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
auto kernelSourceCode = "__qpu__ foo(double theta) {...}";
xacc::Program program(qpu, kernelSourceCode);
program.build();
auto kernel = program.getKernel<
type>("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

### 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 tb, double tc, double h, double j) {
    anmeal 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__ term2(AcceleratorBuffer b, double t0) {
    __qpu__ term3(AcceleratorBuffer b, double t0) {
        ansatz(b, t0)
        MEASURE 0 [0]
        MEASURE 1 [1]
    }
    __qpu__ term4(AcceleratorBuffer b, double t0) {
        ansatz(b, t0)
        MEASURE 0 [0]
        MEASURE 1 [1]
    }
    __qpu__ term1(AcceleratorBuffer b, double t0) {
        ansatz(b, t0)
        MEASURE 1 [0]
    }
```

---

<table>
<thead>
<tr>
<th>XACC Kernel Requirement</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Annotation</td>
<td>All kernels must be annotated with the <code>__qpu__</code> function attribute to enable static, ahead-of-time compilation</td>
</tr>
<tr>
<td>Kernel Name</td>
<td>All kernels must be given a unique name</td>
</tr>
<tr>
<td>Accelerator Buffer Argument</td>
<td>All kernels must take as a first function argument the Accelerator Buffer this kernel acts on.</td>
</tr>
<tr>
<td>Runtime Parameters</td>
<td>All kernels can take any number of runtime arguments.</td>
</tr>
</tbody>
</table>
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

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

Provide common polymorphic representation to map QPLs to QPUs
Programming Model - Compilers and Transpilers

- **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

- **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
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)

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

xacc::Program program(qpu, kernelSourceCode);
program.build();
auto kernel = program.getKernel<
    for (auto theta : thetas) kernel(buffer, theta);
```
## XACC Extensions

<table>
<thead>
<tr>
<th>XACC Extension Points</th>
<th>Extension</th>
<th>Provides</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compiler</td>
<td>xacc-rigetti</td>
<td>QuilCompiler (leverages ANTLR), RigettiAccelerator</td>
</tr>
<tr>
<td>Preprocessor</td>
<td>xacc-ibm</td>
<td>OpenQasmCompiler (leverages ANTLR), IBMAccelerator</td>
</tr>
<tr>
<td>Accelerator</td>
<td>xacc-dwave</td>
<td>DWQMICompiler (leverages ANTLR and Embedding Algorithm extension point), DWAccelerator</td>
</tr>
<tr>
<td>Instruction</td>
<td>xacc-projectq</td>
<td>ProjectQCompiler (low-level ProjectQ-Qasm transpiler)</td>
</tr>
<tr>
<td>EmbeddingAlgorithm</td>
<td>xacc-vqe</td>
<td>VQETask, VQEMinimizeBackend, VQEPogram, FermionCompiler, JW and BK IRTransformations, UCCSD IRGenerator</td>
</tr>
<tr>
<td>ParameterSetter</td>
<td>xacc-vqe-fcidump</td>
<td>FCIDumpPreprocessor (map FCIDump format to FermionCompiler)</td>
</tr>
<tr>
<td>IRTransformation</td>
<td>xacc-vqe-bayesopt</td>
<td>VQEMinimizeBackend implementing bayesian optimization</td>
</tr>
<tr>
<td>IRPreprocessor</td>
<td>xacc-cmr</td>
<td>CMR Embedding Algorithm delegating to D-Wave MinorMinor</td>
</tr>
<tr>
<td>AcceleratorBufferPostprocessor</td>
<td>tnqvm</td>
<td>TNQVM Accelerator (Tensor Network Quantum Virtual Machine), TNQVM Visitor Backend</td>
</tr>
<tr>
<td>IRGenerator</td>
<td>xacc-atos</td>
<td>ATOSVisitor (mapping IR to ATOS Circuit), ATOSAccelerator</td>
</tr>
</tbody>
</table>
What’s wrong with the API below?

```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
auto kernelSourceCode = "__qpu__ foo(double theta) {...}";
auto program = xacc::Program(qpu, kernelSourceCode);
program.build();
auto kernel = program.getKernel("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.

```cpp
$ 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
```

```bash
$ ./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?**
  - PyXACC Compiler interprets custom `xacc()` instruction
    - first argument is the name of the IR Generator
    - following arguments are the list of user input
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):
    qni(0,0,h)
    qni(1,1,h)
    qni(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

Get reference to the D-Wave Accelerator

Define the code you want to run, annotate it

Execute and post-process the results
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
Improved Python Integration

- Learn from CUDA Numba
- [https://github.com/eclipse/xacc/issues/37](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
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
• XACC designed to serve as foundation for libraries and applications.
• Application built on XACC for VQE – [https://github.com/ornl-qci/xacc-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.
Higher-level programming and error mitigation

Deuteron binding energy via standard XACC Python API

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