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
Outline of this talk...

- Purpose of XACC
- The Programming Model
  - Kernels
  - IR
  - Compilers
  - Accelerators
- Examples
  - Factoring $15 = 3 \times 5$
  - 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

Current Quantum Programming Landscape

Gate Model Computing
- Scaffold
- ProjectQ
- pyQuil

Adiabatic Model Computing
- QMI
- ToQ
- Quelle

IBMQPU
Google QPU
Rigetti QPU

2X
2000Q
...
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
  - Quelle

XACC - Heterogeneous CPU-QPU Programming Model

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

• 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!
# XACC Plugins

Framework enables language and hardware agnostic quantum programming

<table>
<thead>
<tr>
<th>Plugin</th>
<th>Provides</th>
<th>Repository</th>
</tr>
</thead>
<tbody>
<tr>
<td>xacc-dwave</td>
<td>DWAccelerator, DWQMICompiler</td>
<td><a href="https://github.com/ornl-qci/xacc-dwave">https://github.com/ornl-qci/xacc-dwave</a></td>
</tr>
<tr>
<td>xacc-rigetti</td>
<td>RigettiAccelerator, QuilCompiler</td>
<td><a href="https://github.com/ornl-qci/xacc-rigetti">https://github.com/ornl-qci/xacc-rigetti</a></td>
</tr>
<tr>
<td>xacc-python</td>
<td>XACC Python Bindings</td>
<td><a href="https://github.com/ornl-qci/xacc-python">https://github.com/ornl-qci/xacc-python</a></td>
</tr>
<tr>
<td>xacc-ibm</td>
<td>IBMAccelerator</td>
<td><a href="https://github.com/ornl-qci/xacc-ibm">https://github.com/ornl-qci/xacc-ibm</a></td>
</tr>
<tr>
<td>xacc-projectq</td>
<td>ProjectQCompiler</td>
<td><a href="https://github.com/ornl-qci/xacc-projectq">https://github.com/ornl-qci/xacc-projectq</a></td>
</tr>
<tr>
<td>tqnvm</td>
<td>TNQVMAccelerator (tensor network simulator)</td>
<td><a href="https://github.com/ornl-qci/tqnvm">https://github.com/ornl-qci/tqnvm</a></td>
</tr>
<tr>
<td>xacc-vqe</td>
<td>FermionCompiler, general Variational Quantum Eigensolver</td>
<td><a href="https://github.com/ornl-qci/xacc-vqe">https://github.com/ornl-qci/xacc-vqe</a></td>
</tr>
</tbody>
</table>

## Installing new plugins is easy:

```
$ xacc-install-plugins.py -p xacc-dwave
```
XACC Concepts and Layered Architecture

Key Concepts:
1. Quantum Kernel
2. Quantum Compiler
3. QPU Intermediate Representation
4. Quantum Accelerator and Accelerator Buffer
XACC Model

Quantum Kernels

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

Example D-Wave QMI Kernel

```c
__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;
}
```
XACC Model

Quantum Kernels

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

Circuit

QUBO

Example Gate Model Kernel written in Scaffold

```c
__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]);
}
```
XACC Model

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

XACC HUBO Interface

- reduceToQUBO(InputArgs…) : Graph
- mapResults(buf : AcceleratorBuffer)

BOP

produces

QUBO Graph

Output kernel of FactoringHUBO for 15 = 3*5

```cpp
__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;
}
```
XACC Model

Quantum Intermediate Representation (QIR) Specification

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

- **Compiler**
  - compile(kernelSrc EString, accelerator Accelerator) : IR
  - translate(irFunction Function) : EString

- **Preprocessor**
  - process(kernelSrc EString, compiler Compiler, accelerator Accelerator) : EString

- **ScaffoldCompiler**
  - DWQMICompiler
  - QuilCompiler
  - FermionCompiler

- **ClangScaffoldASTVisitor**
  - [1..1] clangscaffoldastvisitor

- **EmbeddingAlgorithm**
  - embed(prob Graph, acc Graph)

- **ParameterSetter**
  - setParameters(prob Graph, acc Graph, emb Embedding)

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

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

Accelerators provide qubit resources - AcceleratorBuffers

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

Quantum Accelerator - DWAccelerator

XACC Accelerator Interface

execute(buffer : AcceleratorBuffer, ir : Function)

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
Kernels are just executable functors or lambdas.
# Programs, Execution Workflow, and API

Kernels are just executable functors or lambdas

```cpp
#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();
}
```
FactoringHUBO Workflow:
1. Construct symbolic algebra for \(0 = (N - pq)^2\)
2. Manipulate expression and reduce to quadratic form
3. Convert to Ising form
4. Map expression to Qubo

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

https://github.com/ORNL-QCI/xacc-dwsapi-embedding
Factoring Application

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

```cpp
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

```bash
$ 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.