# XACC Documentation

***Release 0.0.1***

**Alex McCaskey**

**Sep 07, 2017**

# CONTENTS:

CHAPTER

ONE

# GETTING STARTED WITH XACC

## 1.1 Overview

XACC is a programming framework for extreme-scale accelerator architectures that integrates alongside existing conventional classical applications. It is a pluggable framework for programming languages and accelerators developed for next-gen computing hardware architectures like quantum and neuromorphic computing. It lets computational scientists efficiently off-load classically intractable work to attached accelerators through a user-friendly kernel API. XACC makes post-exascale hybrid programming approachable for domain computational scientists.

The XACC programming model, and associated open-source reference implementation, follows the traditional co-processor model, akin to OpenCL or CUDA for GPUs, but takes into account the subtleties and complexities inherent to the interplay between classical and quantum hardware. XACC provides a high-level API that enables classical applications to offload computationally intractable work (represented as quantum kernels) to an attached quantum accelerator in a manner that is agnostic to the quantum programming language and the quantum hardware. This enables one to write quantum code once, and perform benchmarking, verification and validation, and performance studies for a set of virtual (simulators) or physical hardware.

To achieve this interoperability, XACC defines four primary abstractions or concepts: quantum kernels, intermediate representation, compilers, and accelerators. Quantum kernels are C-like functions that contain code intended for execution on the QPU. These kernels are compiled to the XACC intermediate representation (IR), an object model that is key for promoting the integration of a diverse set of languages and hardware. The IR provides four main forms for use by algorithm programmers: (1) an in-memory representation and API, (2) an on-disk persisted representation, (3) human-readable quantum assembly representation, and (4) a control flow graph or quantum circuit representation. This IR is produced by realizations of the XACC compiler concept, which delegates to the kernel language's appropriate parser, compiler, and optimizer. Finally, XACC IR instances (and therefore programmed kernels) are executed by realizations of the Accelerator concept, which defines an interface for injecting physical or virtual quantum accelerators. Accelerators take this IR as input and delegate execution to vendor-supplied APIs for the QPU (or API for a simulator). The orchestration of these concepts enable an expressive API for quantum acceleration of classical applications.

XACC has support for a number of languages and physical and virtual hardware instances. XACC provides a Compiler realization that enables quantum kernel programming in the Scaffold programming language - an effort that came out of the IARPA QCS program. This compiler leverages the Clang/LLVM library extensions developed under that project that extend the LLVM IR with quantum gate operations. XACC extends this compiler with support for new constructs, like custom quantum functions and source-to-source translations (mapping Scaffold to other languages). XACC provides an Accelerator realization that enables execution of quantum kernels in any available language for both the Rigetti Quantum Virtual Machine (QVM, Forest API) and the physical two qubit (pyquillow) Rigetti QPU. These Accelerators map the XACC IR to Quil (the Rigetti low-level assembly language) and leverage an HTTP Rest client to post compiled quantum kernel code to the Rigetti QVM/QPU driver servers. XACC also has support for the D-Wave QPU, which demonstrates the wide applicability of this heterogeneous hybrid programming model across quantum computing models. XACC has Compiler and Accelerator realizations that enable minor graph embedding of binary optimization problems and execution on the D-Wave Qubist QPU driver server, respectively.

## 1.2 XACC Plugin Infrastructure

XACC relies on a project called CppMicroServices - a native C++ implementation of the OSGi specification that enables an extensible plugin infrastructure for compilers and accelerators. As such, installation of XACC provides the core infrastructure for describing Programs, Compilers, Accelerators, and IR. To enable support for various compilers and accelerators (like the Scaffold or Quil compilers, or the IBM or Rigetti QPUs) you must install the appropriate plugin (see XACC Plugins).

## 1.3 XACC Development Team

XACC is developed and maintained by:

- Alex McCaskey
- Travis Humble
- Eugene Dumitrescu
- Dmitry Liakh
- Mengsu Chen

## 1.4 Questions, Bug Reporting, and Issue Tracking

Questions, bug reporting and issue tracking are provided by GitHub. Please report all bugs by creating a new issue. You can ask questions by creating a new issue with the question tag.

## 1.5 Publications and Presentations

CHAPTER

TWO

# INSTALLATION

This section provide guidelines for installing XACC and its TPLs.

## 2.1 Install third-party libraries

The following third party libraries (TPLs) are used by XACC:

| Packages | Dependency | Version |
|---|---|---|
| C++14 Compiler | Required | See below |
| Boost | Required | 1.59.0+ |
| MPI | Optional | N/A |

Note that you must have a C++14 compliant compiler. For GCC, this means version 6.1+, for Clang, this means 3.4+.

These dependencies are relatively easy to install on popular operating systems. Any of the following commands will work (showing with and without MPI):

```
$ (macosx) brew install boost
$ (macosx) brew install boost-mpi
$ (fedora) dnf install boost-devel
$ (fedora) dnf install boost-mpich-devel
$ (fedora) dnf install boost-openmpi-devel
$ (ubuntu) apt-get install libboost-all-dev # will install openmpi
```

## 2.2 Build XACC

Clone the XACC repository:

```
$ git clone https://github.com/ornl-qci/xacc
```

XACC requires CMake 3.2+ to build. Run the following to configure and build XACC:

```
$ cd xacc && mkdir build && cd build
$ cmake ..
$ make install # can pass -jN for N = number of threads to use
```

This will install XACC to /usr/local/xacc (Pass -DCMAKE_INSTALL_PREFIX=$YOURINSTALLPATH to install it somewhere else).

Set your PATH variable to include the XACC bin directory:

```
$ export PATH=/usr/local/xacc/bin:$PATH
```

Additionally, you could add this to your home directory's .basrhc file (or equivalent).

## 2.3 Installing XACC Plugins

If you have successfully built XACC (see above) then you can now run

```
$ xacc-install-plugins.py --help
```

This is a convenience python script to help download, build, and install all currently supported XACC plugins. The execution syntax is as follows:

```
$ xacc-install-plugins.py -p PLUGIN-NAME
```

You can also run this script with multiple plugin names.

Let's see how to use this script to install the Rigetti, IBM, TNQVM, Scaffold, D-Wave, and Python plugins.

---

**Note:** If you want support for the IBM, D-Wave, and Rigetti Accelerators, you must install CppRestSDK and OpenSSL. This is required for these Accelerators to make remote HTTP Rest calls to their respective server APIs. Here's how to install these as binaries on various popular platforms:

```
$ (macosx) brew install cpprestsdk
$ (fedora) dnf install cpprest-devel openssl-devel
$ (ubuntu) apt-get install libcpprest-dev libssl-dev
```

---

### 2.3.1 Rigetti Support

The Rigetti Plugin provides support to XACC for compiling kernels writting in Quil, and executing programs on the Rigetti QVM via a Rigetti Accelerator.

To install this plugin, run the following

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

You have now installed the Rigetti plugin. It is located in $XACC_ROOT/lib/plugins/accelerator and $XACC_ROOT/lib/plugins/compilers, where XACC_ROOT is your XACC install prefix.

### 2.3.2 IBM Support

The IBM Plugin provides support to XACC for executing programs on the IBM Quantum Experience via the IBM Accelerator.

To install this plugin, run the following

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

You have now installed the IBM plugin. It is located in $XACC_ROOT/lib/plugins/accelerator, where XACC_ROOT is your XACC install prefix.

### 2.3.3 TNQVM

The TNQVM Plugin provides support to XACC for executing programs on the ORNL tensor network quantum virtual machine.

---

**Note:** This Accelerator requires BLAS/LAPACK libraries to be installed. Here's how to install these as binaries on various popular platforms:

```
$ (macosx) should already be there in Accelerate Framework, if not
$ (macosx) brew install openblas lapack
$ (fedora) dnf install blas-devel lapack-devel
$ (ubuntu) apt-get install libblas-dev liblapack-dev
```

---

To install this plugin, run the following

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

You have now installed the TNQVM plugin. It is located in $XACC_ROOT/lib/plugins/accelerator, where XACC_ROOT is your XACC install prefix.

### 2.3.4 Scaffold Support

---

**Note:** Due to issues getting ScaffCC to link correctly with RTTI on Mac OS X, we do not have a binary package installer for Mac OS X. We are open to PRs on this if you can help.

---

To use the Scaffold Plugin you must have our fork of Scaffold installed as a binary package. We have builds for Fedora 25/26 and Ubuntu 16.04/17.04. To install

```
$ (fedora) dnf install https://github.com/ORNL-QCI/ScaffCC/releases/download/v2.0/
↪scaffold-2.0-1.fc25.x86_64.rpm (REPLACE 25 with 26 if on FC26)
$ (ubuntu) wget https://github.com/ORNL-QCI/ScaffCC/releases/download/v2.0/scaffold_2.
↪0_amd64.deb
$ (ubuntu) apt-get install -y $(dpkg --info scaffold_2.0_amd64.deb | grep Depends |␣
↪sed "s/.*ends:\ //" | sed 's/,//g')
$ (ubuntu) dpkg -i scaffold_2.0_amd64.deb
```

To install this plugin, run the following

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

You have now installed the Scaffold plugin. It is located in $XACC_ROOT/lib/plugins/compilers, where XACC_ROOT is your XACC install prefix.

### 2.3.5 D-Wave Support

The D-Wave Plugin provides support to XACC for executing programs on the D-Wave QPU via the D-Wave Accelerator.

To install this plugin, run the following

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

You have now installed the D-Wave plugin. It is located in $XACC_ROOT/lib/plugins/accelerator, where XACC_ROOT is your XACC install prefix.

Furthermore, XACC has extensibility built in for minor graph embedding algorithms. We currently have one supported embedding algorithm, a wrapper around the D-Wave SAPI Cai, Macready, Roi algorithm. In order to install this as a plugin, run the following

---

**Note:** The following embedding algorithm needs to leverage the proprietary D-Wave SAPI header file and associated shared library: dwave_sapi.h and libdwave_sapi.so. In order for the installation below to work, place dwave_sapi.h in /usr/local/include/ and libdwave_sapi.so in /usr/local/lib/

---

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

You have now installed the D-Wave plugin. It is located in $XACC_ROOT/lib/plugins/accelerator and $XACC_ROOT/lib/plugins/compilers, where XACC_ROOT is your XACC install prefix.

### 2.3.6 Python Bindings

The Python Plugin provides Python language bindings to XACC through the pybind11 project.

---

**Note:** This plugin requires Python 2.7+ development headers/library. Here's how to install these on various popular platforms:

```
$ (macosx) brew install python
$ (fedora) dnf install python-devel
$ (ubuntu) apt-get install python-dev
```

---

To install this plugin, run the following

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

You have now installed the Python plugin. It is located in $XACC_ROOT/lib/python, where XACC_ROOT is your XACC install prefix.

In order to use this installation, you must update your PYTHONPATH environment variable

```
$ export PYTHONPATH=$XACC_ROOT/lib/python:$PYTHONPATH
```

We recommend placing this command in your home directory's .bashrc file (or equivalent).

## 2.4 XACC and Spack

You can build XACC and its dependencies with the Spack package manager.

To configure your available system compilers run

```
$ spack compilers
```

**Note:** If you run 'spack config get compilers' and your desired compiler has fc and f77 set to Null or None, then the install will not work if you are including MPI support. If this is the case, it usually works to run 'spack config edit compilers' and manually replace Null with /path/to/your/gfortran

We will rely on the environment-modules package to load/unload installed Spack modules. If you don't have this installed (you can check by running 'module avail') install with

```
$ spack install environment-modules
```

Add the following to your ~/.bashrc (or equivalent)

```
. $SPACK_ROOT/share/spack/setup-env.sh
source $(spack location -i environment-modules)/Modules/init/bash
```

If you do not have a C++14 compliant compiler, you can install one with Spack, for example

```
$ spack install gcc@7.2.0 # this will take awhile...
$ spack load gcc
$ spack compiler find
```

Now install the dependencies with your specified C++14 compiler (mine will be gcc 7.2.0)

```
$ (with MPI support) spack install boost+mpi+graph ^mpich %gcc@7.2.0
$ (without MPI support) spack install boost+graph %gcc@7.2.0
```

XACC has not yet been accepted into the Spack (we will soon issue a PR to get it shipped as part of Spack). So in order to install it with Spack we have to download our custom package recipe from the XACC repository:

```
$ cd $SPACK_ROOT/var/spack/repos/builtin/packages/ && mkdir xacc
$ cd xacc && wget https://github.com/ORNL-QCI/xacc/raw/master/cmake/spack/xacc/
↪package.py .
```

Now we can run

```
$ spack install xacc %gcc@7.2.0
```

Once all these are installed, load them as environment modules so they are available for the XACC build:

```
$ spack load boost
```

CHAPTER

THREE

# XACC API

CHAPTER

FOUR

# XACC PLUGINS

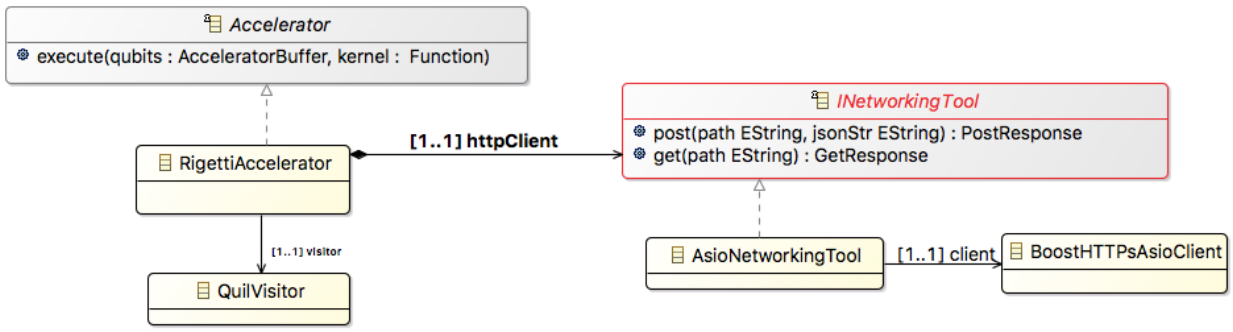
## 4.1 Rigetti

Rigetti Computing, Inc (rigetti.com) is a recently founded startup that is focused on developing quantum computing hardware and software and bring it to market. They are working to build a cloud quantum computing platform for AI and computational chemistry. They currently have a QVM simulation server that can be accessed via a REST API with a private API key. Rigetti has also done great work as of late in providing open source programming tools for interfacing with their QVM - specifically, the PyQuil python framework PyQuil.

Recently, the ORNL QCI (quantum.ornl.gov), the XACC project, and the Software and Applications Team from Rigetti have begun collaborating in an effort to expose the Rigetti QVM server and programming tools to XACC and its user community. This article describes the results of that work - specifically, a new XACC Accelerator implementation that executes quantum kernels on the Rigetti QVM server. For a more hands-on tutorial on how to use XACC and the Rigetti Accelerator, check out Rigetti Tutorial).

### 4.1.1 RigettiAccelerator

The RigettiAccelerator is an implementation or realization of the pluggable XACC Accelerator interface. The RigettAccelerator class architecture diagram is shown in Figure 1. The RigettiAccelerator's implementation of the Accelerator::execute() method is charged with two primary tasks: (1) the translation of the XACC IR to an equivalent Quil string, and (2) constructing and executing an appropriate HTTPS Post on the Rigetti QVM server. The only remaining thing to do once those two tasks are complete is to processes the resultant response from the server.




### 4.1.2 Mapping XACC IR to Quil

Basically, at its core, the XACC IR provides a tree-like, in-memory representation and API for a compiled quantum kernel. The leaves of this tree are XACC Instructions and the nodes of the tree are XACC Functions, which are composed of further child Instructions. The XACC Quantum IR implementation provides a number of standard gate Instruction implementations (Hadamard, CNOT, rotations, etc…) These serve as the leaves of the IR tree. These

instruction implementations know nothing of the Quil intermediate language and it would be tedious and a poor design decision to to update the entire XACC Quantum IR package (we would have to do the same for any and all current and future low-level languages). So XACC employs a common software engineering design pattern to enable this XACC IR to Quil mapping: the visitor pattern, which provides a mechansim for adding new operations to an existing object without modifying the design of that object (Visitor Pattern). For each derived gate Instruction, a Visitor class implements a corresponding `visit` method (`visit(Hadamard& h)`, etc...). All gate instructions have the ability to accept an incoming Visitor, and upon doing so, invoke the `visit` method that corresponds to their type, thus giving the Visitor type information for the Gate Instruction. Therefore, mapping to Quil simply involves walking the IR tree, and telling each Instruction to accept the visitor:

```
auto visitor = std::make_shared<QuilVisitor>();
InstructionIterator it(kernel);
while (it.hasNext()) {
   // Get the next node in the tree
   auto nextInst = it.next();
   if (nextInst->isEnabled()) nextInst->accept(visitor);
}
auto quilStr = visitor->getQuilString();
```

The visitor implementation is known as the QuilVisitor, and its visit methods look like this (Hadamard for example):

```
void visit(Hadamard& h) {
   quilStr += "H " + std::to_string(h.bits()[0]) + "\n";
}
```

or for a more complicated gate Instruction:

```
void visit(ConditionalFunction& c) {
   auto visitor = std::make_shared<QuilVisitor>();
   auto classicalBitIdx = qubitToClassicalBitIndex[c.getConditionalQubit()]; //␣
↪populated in visit(Measure)
   quilStr += "JUMP-UNLESS @" + c.getName() + " [" + std::to_string(classicalBitIdx)␣
↪+ "]\n";
   for (auto inst : c.getInstructions()) {
       inst->accept(visitor);
   }
  quilStr += visitor->getQuilString();
  quilStr += "LABEL @" + c.getName() + "\n";
}
```

After walking the IR tree, the Quil representation is produced with a call to getQuilString().

### 4.1.3 Executing Quil code on Rigetti QVM

With the XACC IR mapped to Quil, the RigettiAccelerator is ready to execute on the Rigetti QVM. The main task here is to construct the proper JSON payload string that contains information about the type of the execution, the classical memory address indices, and the Quil instructions string. The types of execution that the QVM allows are multishot, multishot-measure, wavefunction, and expectation. In this work, we have primarily focused on the multishot method. If the execution type is multishot, then we can provide a further JSON key that is an integer that gives the number of executions of the Quil code to run.

## 4.2 IBM

## 4.3 D-Wave

## 4.4 Scaffold

## 4.5 TNQVM

Python —— -

CHAPTER

FIVE

# XACC TUTORIALS

First off, make sure you have successfully built XACC (see XACC Install).

## 5.1 Rigetti QVM Tutorial

Create a new directory called test-xacc-rigetti and cd into it. Let's now create a test-xacc-rigetti.cpp file and get it started with the following boilerplate code:

```
#include "XACC.hpp"

int main(int argc, char** argv) {

   // Initialize XACC - find all available
   // compilers and accelerators, parse command line.
   xacc::Initialize(argc, argv);

   // ... Code to come ...

   // Finalize the framework.
   xacc::Finalize();
}
```

Building this code is straightforward with CMake. Create a CMakeLists.txt file in the same directory as the test-xacc-rigetti.cpp file, and add the following to it:

```
# Start a CMake project
project(test-xacc-rigetti CXX)

# Set the minimum version to 3.2
cmake_minimum_required(VERSION 3.2)

# Find XACC
find_package(XACC REQUIRED)

# Find Boost
find_package(Boost COMPONENTS system program_options filesystem chrono thread␣
↪REQUIRED)

# Include all XACC Include Directories
include_directories(${XACC_INCLUDE_DIRS})

# Link to the XACC Library Directory,
link_directories(${XACC_LIBRARY_DIR})
```

```
# Create the executabe
add_executable(test-xacc-rigetti test-xacc-rigetti.cpp)

# Like the necessary libaries
target_link_libraries(test-xacc-rigetti ${XACC_LIBRARIES} dl pthread)
```

Now from within the test-xacc-rigetti directory, run the following:

```
$ mkdir build
$ cd build
$ cmake ..
$ make
```

This will build test-xacc-rigetti.cpp and provide you with a test-xacc-rigetti executable. Run that executable to ensure that your build worked (you should see the following output): .. code:

```
$ make
$ ./test-xacc-rigetti
[2017-06-20 16:14:07.076] [xacc-console] [info] [xacc] Initializing XACC Framework
[2017-06-20 16:14:07.091] [xacc-console] [info] [xacc::compiler] XACC has 1 Compilers␣
↪available.
[2017-06-20 16:14:07.091] [xacc-console] [info] [xacc::accelerator] XACC has 1␣
↪Accelerators available.
[2017-06-20 16:14:07.091] [xacc-console] [info]
[xacc] XACC Finalizing
[xacc::compiler] Cleaning up Compiler Registry.
[xacc::accelerator] Cleaning up Accelerator Registry.
```

Now that we have our build and initial boilerplate code setup, let's actually write some quantum code, specifically teleporting the state of one qubit to another. Following the XACC.hpp include statement at the top of the file, add the following quantum kernel declaration:

```
const std::string src("__qpu__ teleport (qbit qreg) {\n"
"   cbit creg[3];\n"
"   // Init qubit 0 to 1\n"
"   X(qreg[0]);\n"
"   // Now teleport...\n"
"   H(qreg[1]);\n"
"   CNOT(qreg[1],qreg[2]);\n"
"   CNOT(qreg[0],qreg[1]);\n"
"   H(qreg[0]);\n"
"   creg[0] = MeasZ(qreg[0]);\n"
"   creg[1] = MeasZ(qreg[1]);\n"
"   if (creg[0] == 1) Z(qreg[2]);\n"
"   if (creg[1] == 1) X(qreg[2]);\n"
"   // Check that 3rd qubit is a 1\n"
"   creg[2] = MeasZ(qreg[2]);\n"
"}\n");
```

Now we are ready to build and execute this kernel using the XACC Runtime API. After the call to xacc::Initialize, add the following:

```
// 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);
```

The code above starts by getting a reference to the RigettiAccelerator. With that reference, we then allocate a register of qubits to operate the teleport kernel on. Next, we instantiate an XACC Program instance, which keeps track of the desired Accelerator and the source code to be compiled. The Program instance orchestrates the compilation of the quantum kernel to produce the XACC intermediate representation, and then handles the creation of an executable classical kernel function that offloads the compiled quantum code to the specified Accelerator. Finally, the user requests a reference to the executable kernel functor, and executes it on the provided register of qubits.

The total test-xacc-rigetti.cpp file should look like this: .. code:

```
#include "XACC.hpp"

// Quantum Kernel executing teleportation of
// qubit state to another.
const std::string src("__qpu__ teleport (qbit qreg) {\n"
"   cbit creg[3];\n"
"   // Init qubit 0 to 1\n"
"   X(qreg[0]);\n"
"   // Now teleport...\n"
"   H(qreg[1]);\n"
"   CNOT(qreg[1],qreg[2]);\n"
"   CNOT(qreg[0],qreg[1]);\n"
"   H(qreg[0]);\n"
"   creg[0] = MeasZ(qreg[0]);\n"
"   creg[1] = MeasZ(qreg[1]);\n"
"   if (creg[0] == 1) Z(qreg[2]);\n"
"   if (creg[1] == 1) X(qreg[2]);\n"
"   // Check that 3rd qubit is a 1\n"
"   creg[2] = MeasZ(qreg[2]);\n"
"}\n");

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);

   // Finalize the XACC Framework
   xacc::Finalize();

   return 0;
}
```

Now, to build simple run: .. code:

```
$ cd test-xacc-rigetti/build
$ make
```

To execute this code on the Rigetti QVM, you must provide your API key. You can do this the same way you do with PyQuil (in your $HOME/.pyquil_config file, or in the $PYQUIL_CONFIG environment variable). You can also pass your API key to the XACC executable through the –rigetti-api-key command line argument:

```
$ ./test-xacc-rigetti --rigetti-api-key KEY
[2017-06-20 17:43:38.744] [xacc-console] [info] [xacc] Initializing XACC Framework
[2017-06-20 17:43:38.760] [xacc-console] [info] [xacc::compiler] XACC has 3 Compilers␣
↪available.
[2017-06-20 17:43:38.760] [xacc-console] [info] [xacc::accelerator] XACC has 2␣
↪Accelerators available.
[2017-06-20 17:43:38.766] [xacc-console] [info] Executing Scaffold compiler.
[2017-06-20 17:43:38.770] [xacc-console] [info] Rigetti Json Payload = { "type" :
↪"multishot", "addresses" : [0, 1, 2], "quil-instructions" : "X 0\nH 1\nCNOT 1␣
↪2\nCNOT 0 1\nH 0\nMEASURE 0 [0]\nMEASURE 1 [1]\nJUMP-UNLESS @conditional_0 [0]\nZ␣
↪2\nLABEL @conditional_0\nJUMP-UNLESS @conditional_1 [1]\nX 2\nLABEL @conditional_
↪1\nMEASURE 2 [2]\n", "trials" : 10 }
[2017-06-20 17:43:40.439] [xacc-console] [info] Successful HTTP Post to Rigetti.
[2017-06-20 17:43:40.439] [xacc-console] [info] Rigetti QVM Response:
[[0,1,1],[1,1,1],[1,1,1],[0,1,1],[1,0,1],[1,1,1],[0,1,1],[0,1,1],[0,0,1],[0,0,1]]
[2017-06-20 17:43:40.439] [xacc-console] [info]
[xacc] XACC Finalizing
[xacc::compiler] Cleaning up Compiler Registry.
[xacc::accelerator] Cleaning up Accelerator Registry.
```

You should see the console text printed above.

You can also tailor the number of executions that occur for the multishot execution type:

```
$ ./test-xacc-rigetti --rigetti-trials 1000
[2017-06-20 17:50:57.285] [xacc-console] [info] [xacc] Initializing XACC Framework
[2017-06-20 17:50:57.301] [xacc-console] [info] [xacc::compiler] XACC has 3 Compilers␣
↪available.
[2017-06-20 17:50:57.301] [xacc-console] [info] [xacc::accelerator] XACC has 2␣
↪Accelerators available.
[2017-06-20 17:50:57.307] [xacc-console] [info] Executing Scaffold compiler.
[2017-06-20 17:50:57.310] [xacc-console] [info] Rigetti Json Payload = { "type" :
↪"multishot", "addresses" : [0, 1, 2], "quil-instructions" : "X 0\nH 1\nCNOT 1␣
↪2\nCNOT 0 1\nH 0\nMEASURE 0 [0]\nMEASURE 1 [1]\nJUMP-UNLESS @conditional_0 [0]\nZ␣
↪2\nLABEL @conditional_0\nJUMP-UNLESS @conditional_1 [1]\nX 2\nLABEL @conditional_
↪1\nMEASURE 2 [2]\n", "trials" : 100 }
[2017-06-20 17:50:57.909] [xacc-console] [info] Successful HTTP Post to Rigetti.
```

```
[2017-06-20 17:50:57.909] [xacc-console] [info] Rigetti QVM Response:
[[1,0,1],[0,0,1],[1,1,1],[0,1,1],[1,0,1],[0,1,1],[0,0,1],[1,1,1],[1,0,1],[1,0,1],[0,0,
↪1],[1,0,1],[1,1,1],[0,1,1],[0,0,1],[1,1,1],[1,0,1],[1,1,1],[0,0,1],[1,1,1],[1,0,1],
↪[0,0,1],[0,0,1],[1,0,1],[0,1,1],[0,0,1],[1,1,1],[0,0,1],[0,1,1],[1,1,1],[1,0,1],[1,
↪0,1],[0,1,1],[0,1,1],[1,1,1],[1,1,1],[1,1,1],[0,1,1],[1,1,1],[1,0,1],[1,0,1],[1,1,
↪1],[1,1,1],[0,0,1],[1,1,1],[0,0,1],[1,0,1],[1,1,1],[1,0,1],[1,1,1],[0,1,1],[0,1,1],
↪[1,0,1],[0,0,1],[1,1,1],[0,1,1],[0,1,1],[1,1,1],[1,0,1],[1,1,1],[0,0,1],[0,0,1],[1,
↪0,1],[0,1,1],[0,0,1],[0,1,1],[1,0,1],[0,1,1],[1,0,1],[0,0,1],[1,0,1],[1,1,1],[1,0,
↪1],[1,1,1],[0,0,1],[0,1,1],[1,0,1],[1,1,1],[1,1,1],[0,1,1],[1,0,1],[1,1,1],[0,1,1],
↪[1,0,1],[1,0,1],[0,0,1],[1,0,1],[0,0,1],[0,0,1],[1,0,1],[1,1,1],[0,1,1],[0,1,1],[0,
↪1,1],[1,0,1],[1,1,1],[1,1,1],[0,1,1],[0,1,1],[0,1,1]]
[2017-06-20 17:50:57.910] [xacc-console] [info]
[xacc] XACC Finalizing
[xacc::compiler] Cleaning up Compiler Registry.
[xacc::accelerator] Cleaning up Accelerator Registry.
```

Note above we let XACC find the API Key in the standard .pyquil_config file.

## 5.2 XACC Python Bindings Tutorial

## 5.3 PyQuil-XACC Integration

## 5.4 XACC D-Wave Markowitz Financial Modeling

CHAPTER

SIX

# INDICES AND TABLES

- genindex
- modindex
- search