qrack Documentation

qrack

Mar 10, 2021

Contents

CHAPTER 1

Build Status

- Qrack:
- VM6502Q:
- CC65:
- Examples:

CHAPTER 2

Introduction

Qrack is a C++ quantum bit and gate simulator, with the ability to support arbitrary numbers of entangled qubits - up to system limitations. Suitable for embedding in other projects, the `Qrack::QInterface` contains a full and performant collection of standard quantum gates, as well as variations suitable for register operations and arbitrary rotations.

The developers of Qrack maintain a fork of the ProjectQ quantum computer compiler which can use Qrack as the simulator, generally. This stack is also compatible with the SimulaQron quantum network simulator. Further, we maintain a QrackProvider for Qiskit. Both ProjectQ and Qiskit integrations for Qrack support the PennyLane stack. (For Qiskit, a fork of the Qiskit plugin provides support for a "QrackDevice".) Qrack's developers are not directly affiliated with any of these projects, but we thank them for their contribution to the open source quantum computing community!

As a demonstration of the `Qrack::QInterface` implementation, a MOS-6502 microprocessor [MOS-6502] virtual machine has been modified with a set of new opcodes (*MOS-6502Q Opcodes*) supporting quantum operations. The vm6502q virtual machine exposes new integrated quantum opcodes such as Hadamard transforms and an X-indexed LDA, with the X register in superposition, across a page of memory. An assembly example of a Grover's search with a simple oracle function is demonstrated in the examples repository.

CHAPTER 3

Copyright

Daniel Strano would like to specifically note that Benn Bollay is almost entirely responsible for the original implementation of QUnit and tooling, including unit tests, in addition to large amounts of work on the documentation and many other various contributions in intensive reviews. Also, thank you to Marek Karcz for supplying an awesome base classical 6502 emulator for proof-of-concept.

3.1 Getting Started

3.1.1 Prerequisites

Qrack compiles with a C++11 compiler, such as g++ or clang++, with any required compilation flags to enable the C++11 standard.

You also need CMake to build. CMake installation instructions can be found here.

Optional GPU support is provided by OpenCL development libraries. See *Installing OpenCL* for further instructions.

3.1.2 Checking Out

Clone the repository with git:

```
  / $ mkdir qc
  / $ cd qc
qc/ $ git clone https://github.com/vm6502q/qrack.git
```

3.1.3 Compiling

The `qrack` project supports two primary implementations: OpenCL-optimized and software-only. See *Installing OpenCL* for details on installing OpenCL on some platforms, or your appropriate OS documentation. If you do not

have OpenCL or do not wish to use it, supply the ENABLE_OPENCL=OFF environment to cmake when building qrack the first time.

```
qc/ $ cd qrack/build && cmake [-DENABLE_OPENCL=OFF] [-DENABLE_COMPLEX8=ON] [-DENABLE_
↪COMPLEX_X2=OFF] [-DENABLE_RDRAND=ON] [-DQBCAPPOW=5-31] ..
```

Then make all or (sudo) make install to compile, (with -j8 for 8 parallel build cores, or as appropriate).

Qrack compiles with either double (ENABLE_COMPLEX8=OFF) or single (ENABLE_COMPLEX8=ON) accuracy complex numbers. Single float accuracy is used by default. Single float accuracy uses almost exactly half as much RAM, allowing one additional qubit. Single accuracy may also be faster or the only compatible option for certain OpenCL devices.

Vectorization (ENABLE_COMPLEX_X2=ON) of doubles uses AVX, while single accuracy vectorization uses SSE 1.0. Turning vectorization off at compile time removes all SIMD vectorization.

QBCAPPOW takes an integer "n" between 5 and 31, such that maximum addressable qubits in a QInterface instance is 2^n. n=5 would be 32 qubits per QInterface instance, n=6 is the defaullt at 64 qubits per, n=7 addresses up to 128 qubits per, and so on up to n=31. "Addressable" qubits does not mean that the qubits can necessarily by allocated on the particular system. However, QUnit Schmidt decomposition optimizations and/or sparse state vector optimizations do render certain very high-qubit-width circuits tractable, when they stay well below the limit of total arbitrary entanglement. (Reducing representational entanglement happens almost entirely "under-the-hood," in QUnit.)

Many OpenCL devices that don't support double accuracy floating point operations still support 64-bit integer types. If a device doesn't support 64-bit integer types, QBCAPPOW=5 (or equivalently ENABLE_PURE32=OFF) will disable all 64-bit types in OpenCL kernels, as well as SIMD. This theoretically supports the OpenCL standard on a device such as a Raspberry Pi 3.

3.1.4 Running Tests

To run unit tests, run the following in the build directory:

```
./unittest [-?] [--layer-...] [--proc-...] [specific_test_name]
```

See the -? help instructions for option details, and Qrack "layer" and "processor type" choices.

The benchmarks respect the same parameters:

```
./benchmarks [-?] [--max-qubits=-1] [--layer-...] [--proc-...] [specific_test_name]
```

--max-qubits will automatically size with -1 as given argument, or otherwise up to the number of qubits specified for this parameter.

3.1.5 Using the API

Qrack API methods operate on "QEngine" and "QUnit" objects. ("QUnit" objects are a specific optional optimization on "QEngine" objects, with the same API interface.) These objects are organized as 1-dimensional arrays of coherent qubits which can be arbitrarily entangled within the QEngine or QUnit. These object have methods that act like quantum gates, for a specified qubit index in the 1-dimensional array, as well as any analog parameters needed for the gate (like for variable angle rotation gates). Many fundamental gate methods have variants that are optimized to act on a contiguous length of qubits in the array at once. For OpenCL QEngineOCL objects, the preferred OpenCL device can be specified in the constructor. For multiprocessor QEngineOCLMulti engines, you can specify distribution of equal-sized sub-engines between available OpenCL devices. See the API reference for more details.

To create a QEngine or QUnit object, you can use the factory provided in include/qfactory.hpp:

```
QInterfacePtr qftReg = CreateQuantumInterface(QINTERFACE_QUNIT, QINTERFACE_STABILIZER_
↪HYBRID, qubitCount, intPerm, rng);
QInterfacePtr qftReg2 = CreateQuantumInterface(QINTERFACE_OPENCL, QINTERFACE_
↪STABILIZER_HYBRID, qubitCount, intPerm, rng);
```

By default, the `Qrack::OCLEngine` singleton attempts to compile kernels and initialize supporting OpenCL objects for all devices on a system. You can strike devices from the list to free their OpenCL resources, usually before initializing OpenCL QEngine objects:

```
// Initialize the singleton and get the list of devices
std::vector<Qrack::OCLDeviceContext> devices = OCLEngine::Instance()->
↪GetDeviceContextPtrVector();
std::vector<Qrack::OCLDeviceContext> filteredDevices;

// Iterate through the list with cl::Device::getInfo to check devices for desirability
std::string devCheck("HD");
for (int i = 0; i < devices.size(); i++) {
    // From the OpenCL C++ API headers:
    string devName = std::string(devices[i].getInfo<CL_DEVICE_NAME>());
    // Check properties...
    if (devName.find(devCheck) != string::npos) {
        // Take or remove devices selectively
        filteredDevices.push_back(devices[i]);
    }
}

// Replace the original list with the filtered one, and (with an optional argument)␣
↪specify the default device.
OCLEngine::Instance()->SetDeviceContextPtrVector(filteredDevices, filteredDevices[0]);
```

With or without this kind of filtering, the device or devices used by OpenCL-based engines can be specified explicitly in their constructors:

```
// "deviceID" is the (int) index of the desired device in the OCLEngine list:
int deviceID = 0;
QEngineOCL qEngine = QEngineOCL(qBitCount, initPermutation, random_generator_pointer,␣
↪deviceID);
```

3.1.6 Optimal CreateQuantumInterface Factory Options

Qrack's most specifically optimized "layer" stack is also its best general use case simulator, (at this time):

```
QInterfacePtr qftReg = CreateQuantumInterface(QINTERFACE_QUNIT, QINTERFACE_STABILIZER_
↪HYBRID, qubitCount, intPerm, rng[, ...]);
```

`QUnit` is Qrack's "novel optimization layer." `QStabilizerHybrid` is a "QUnit shard" that combines Gottesman-Knill stabilizer simulation with Dirac "ket" simulation. The "ket" simulation further "hybridizes" between asynchronous GPU and CPU workloads as is efficient for workloads. When `QUnit` can determine that levels of entanglement are low, it will maintain Schmidt decomposed representations of subunit (or sub-register) state, in an attempt to increase efficiency.

3.1.7 Embedding Qrack

The `qrack` project produces a `libqrack.a` archive, suitable for being linked into a larger binary. See the *`Qrack::QInterface`* documentation for API references, as well as the examples present in the unit tests.

3.1.8 Performance

See the extensive *performance analysis and graphs* section.

3.1.9 Contributing

Pull requests and issues are happily welcome!

Please make sure `make format` (depends on clang-format-5) has been executed against any PRs before being published.

3.1.10 Community

Qrack and VM6502Q have a development community on the Advanced Computing Topics discord server on channel #qrack. Come join us!

3.2 Theory

3.2.1 Foundational Material

A certain amount of prerequisite knowledge is necessary to utilize and understand quantum computational algorithms and processes. Someday this material may be substantially diminished by intelligently chosen abstractions, but today quantum systems are still heavily dependent on an understanding of the underlying mathematical principles.

It is outside the scope of this document to cover that material. However, a set of references have been collected here. These materials provide a sufficient foundation for onboarding a new engineer or scientist.

Quantum Computational Basics

Grover Search Algorithm

3.2.2 Quantum Bit Simulation

Quantum bits are simulated by recording the complex number amplitude of a wave function solution to Schrödinger's equation. All this means is, we record one complex number with legnth between 0 and 1 correspendonding to each possible permutation of bits in the "coherent" set of quantum bits. The sum of the norms of all permutation amplitudes must sum to 1. The norm is given by the complex number times its "complex conjugate." The complex conjugate of a number is given by flipping the plus/minus sign on its imaginary component. Any possible state of a three qubit system can be expressed as follows:

$$|\psi\rangle = x_0|000\rangle + x_1|001\rangle + x_2|010\rangle + x_3|011\rangle + x_4|100\rangle + x_5|101\rangle + x_6|110\rangle + x_7|111\rangle \tag{3.1}$$

Each of the x_n are complex numbers. These are the amplitudes of the quantum system, multiplied times the "eigenstates." If all bits are measured simultaneously to check if they are 0 or 1, the norm of an amplitude gives its probability on measurement, resulting in a particular bit pattern based on the amplitudes and a randomly generated number. While a real quantum mechanical system would produce probabilistic measurements based on these amplitudes, a simulation like this is deterministic, but pseudo-random.

Note: Probability vs Amplitude It is a common misconception that the defining characteristic of a quantum computer, compared to a classical computer, is that a quantum computer is probabilistic. Except, an x_n eigenstate has both

probability and a *phase.* It is not a bit with just a dimension of probability, but rather a bit with two dimensions, one of probablity and one of phase, an "amplitude." The root of this misconception lies in the measurement operation, which can have a probabilistic outcome. But the x_n coefficients are not probabilistic values - rather, they are the amplitude of complex number wave function. If we both represent and measure the state as a permutation of 0 and 1 bits, the value of the wave function for any state is the square root of its probability times a phase factor.

By collecting x_n into a complex number array (called `Qrack::QInterface::stateVec`), the full quantum representation of the system can be recorded using 2^N *complex number* variables for N quantum bits:

```
std::unique_ptr<Complex16[]> sv(new Complex16[1 << qBitCount]);
```

Given a standard X gate matrix,

$$\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \tag{3.2}$$

one might ask, how can this 2×2 matrix be applied against the $1xN$ vector for N arbitrary entangled qubits, where the vector is the x_n amplitudes from (3.1)?

$$\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} ??? \begin{bmatrix} x_{000} \\ x_{001} \\ x_{010} \\ x_{011} \\ x_{100} \\ x_{101} \\ x_{110} \\ x_{111} \end{bmatrix}$$

To do so, we apply a Kronecker product to the gate matrix. This expands the matrix out to the appropriate number of dimensions - in this case we would need to perform two Kronecker products for each of the two bits whose values are irrelevant to the result:

$$(X \otimes I \otimes I) \times M \tag{3.3}$$

$$\left(\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \otimes \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \otimes \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \right) \times \begin{bmatrix} x_{000} \\ x_{001} \\ x_{010} \\ x_{011} \\ x_{100} \\ x_{101} \\ x_{110} \\ x_{111} \end{bmatrix} \tag{3.4}$$

$$\begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix} \times \begin{bmatrix} x_{000} \\ x_{001} \\ x_{010} \\ x_{011} \\ x_{100} \\ x_{101} \\ x_{110} \\ x_{111} \end{bmatrix} \tag{3.5}$$

$$(X \otimes I \otimes I) \times \begin{bmatrix} x_{000} \\ x_{001} \\ x_{010} \\ x_{011} \\ x_{100} \\ x_{101} \\ x_{110} \\ x_{111} \end{bmatrix} = \begin{bmatrix} x_{001} \\ x_{000} \\ x_{011} \\ x_{010} \\ x_{101} \\ x_{100} \\ x_{111} \\ x_{110} \end{bmatrix} \tag{3.6}$$

The operation in (3.3) swaps the amplitudes of 0 and 1 for the first bit out of three, but leaves the second and third bits alone. Using the identity matrix I preserves the amplitudes of the x_{0nn} and x_{1nn} positions. The expanded matrix in (3.5) now has the proper dimensionality to be multiplied directly against the amplitude vector.

Note: It's important to remember here that, unlike a classical NOT which directly inverts a bit, the X gate swaps the *amplitudes* for the states where the qubit is 1 with the amplitudes where the qubit is 0. If the value of $M[0]$ is $|100\rangle$, then a subsequent $X[0]$ gate would exchange x_{100} and x_{000} and therefore leave the state as $|000\rangle$. See Quantum Logic Gates for more information.

Implementing this naively would require matrices sized at 2^{2N} complex numbers for N bits (as illustrated above in (3.5)). This rapidly grows prohibitive in memory usage, and this is the primary limitation for simulating quantum systems using classical components. Fortunately, these types of matrix operations are easily optimized for both memory usage and parallelization.

There are two immediate optimizations that can be performed. The first is an optimization on the matrix size: by performing the math with only a 2×2 matrix, the amount of memory allocated is substantially reduced. The `Qrack::QInterface::Apply2x2()` method utilizes this optimization.

In shorthand for clarity, an optimized X gate is calculated using the following linear algebra:

$$\begin{bmatrix} \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \times \begin{bmatrix} x_{000} \\ x_{001} \end{bmatrix} \\ \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \times \begin{bmatrix} x_{010} \\ x_{011} \end{bmatrix} \\ \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \times \begin{bmatrix} x_{100} \\ x_{101} \end{bmatrix} \\ \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \times \begin{bmatrix} x_{110} \\ x_{111} \end{bmatrix} \end{bmatrix} = \begin{bmatrix} \begin{bmatrix} x_{001} \\ x_{000} \end{bmatrix} \\ \begin{bmatrix} x_{011} \\ x_{010} \end{bmatrix} \\ \begin{bmatrix} x_{101} \\ x_{100} \end{bmatrix} \\ \begin{bmatrix} x_{111} \\ x_{110} \end{bmatrix} \end{bmatrix} \tag{3.7}$$

And, fully decomposing (3.7):

$$\begin{bmatrix} \begin{bmatrix} 0 & 1 \end{bmatrix} \times \begin{bmatrix} x_{000} \\ x_{001} \end{bmatrix} \\ \begin{bmatrix} 1 & 0 \end{bmatrix} \times \begin{bmatrix} x_{000} \\ x_{001} \end{bmatrix} \\ \begin{bmatrix} 0 & 1 \end{bmatrix} \times \begin{bmatrix} x_{010} \\ x_{011} \end{bmatrix} \\ \begin{bmatrix} 1 & 0 \end{bmatrix} \times \begin{bmatrix} x_{010} \\ x_{011} \end{bmatrix} \\ \begin{bmatrix} 0 & 1 \end{bmatrix} \times \begin{bmatrix} x_{100} \\ x_{101} \end{bmatrix} \\ \begin{bmatrix} 1 & 0 \end{bmatrix} \times \begin{bmatrix} x_{100} \\ x_{101} \end{bmatrix} \\ \begin{bmatrix} 0 & 1 \end{bmatrix} \times \begin{bmatrix} x_{110} \\ x_{111} \end{bmatrix} \\ \begin{bmatrix} 1 & 0 \end{bmatrix} \times \begin{bmatrix} x_{110} \\ x_{111} \end{bmatrix} \end{bmatrix} = \begin{bmatrix} x_{001} \\ x_{000} \\ x_{011} \\ x_{010} \\ x_{101} \\ x_{100} \\ x_{111} \\ x_{110} \end{bmatrix}$$

It's worth pointing out that the operation detailed in (3.7) is heavily parallelize-able, yielding substantial benefits when working with gates spanning more than just one register (e.g. $CNOT$ and $CCNOT$ gates). In C++, this would be implemented like so:

```
// Create a three qubit register.
Qrack::QInterface qReg(3);

// X-gate the bit at index 0
qReg->X(0);
```

The second optimization is to maintain separability of state vectors between bits where entanglement is not necessary. See IBM's article and related publication for details on how to optimize these operations in more detail. The `Qrack::QUnit` and *`Qrack::QInterface`* register-wide operations (e.g. *`Qrack::QInterface::X()`*) leverage these types of optimizations, with parallelization provided through threading and OpenCL, as supported.

LDA,X Unitary Matrix

Note that the VM6502Q X-addressed LDA, ADC, and SBC operations can load, add, or subtract with a superposed X register. If the permutation states of the classical memory addressed by the X register are treated as quantum degrees of freedom, these operations are unitary. A simplified example of the unitary matrix or operator for 2 qubits and a "lookup table" of two independent bits is given below. The least significant bit is the index (or X register), the second least significant bit is the value (or accumulator), and the third and fourth bits are the 0 and 1 indexed classical bits in the "lookup table," treated as quantum degrees of freedom. The rows and columns of the matrix proceed in bit

signifance permutation order from $|0000\rangle$ to $|1111\rangle$.

$$\begin{bmatrix}
1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & & & & \\
0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & & & & \\
0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & & & & \\
0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & & & & \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & & & & \\
0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & & & & \\
0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & & & & \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & & & & \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & & & & \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 & 0 & & & & \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & 0 & 0 & 0 & 0 & 0 & & & & \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & & & & \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 & & & & \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & & & & \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 & & & & \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & 0 & & & &
\end{bmatrix}$$

This allows search of a "real unstructured database" or unstructured lookup table, per [Broda2016]. That paper also proposes a model for the memory of the lookup table.

3.2.3 6502 Reference Documents

For details on the added opcodes supported by vm6502q, see *MOS-6502Q Opcodes*.

3.3 Installing OpenCL

OpenCL development libraries are required to enable GPU support for Qrack. OpenCL library installation instructions vary widely depending on hardware vendor and operating system. See the instructions from your hardware vendor (NVIDIA, Intel, AMD, etc.) for your operating system, for installing OpenCL development libraries. The OpenCL C++ bindings header is also required, though it might be included with your vendor's development libraries installation.

3.3.1 VMWare

1. Download the AMD APP SDK
2. Install it.
3. Add symlinks for `/opt/AMDAPPSDK-3.0/lib/x86_64/sdk/libOpenCL.so.1` to `/usr/lib`
4. Add symlinks for `/opt/AMDAPPSDK-3.0/lib/x86_64/sdk/libamdocl64.so` to `/usr/lib`
5. Make sure `clinfo` reports back that there is a valid backend to use (anything other than an error should be fine).
6. Install OpenGL headers: `$ sudo apt install mesa-common-dev`
7. Adjust the `Makefile` to have the appropriate search paths, if they are not already correct.

3.3.2 Installing OpenCL on Mac

While the OpenCL framework is available by default on most modern Macs, the C++ header "cl.hpp" or "cl2.hpp" is usually not. One option for building for OpenCL is to download this header file and include it in include/OpenCL (as "cl.hpp"). The OpenCL C++ header can be found at the Khronos OpenCL registry:

https://www.khronos.org/registry/OpenCL/

3.4 Examples

3.4.1 Qrack Examples

Check the Qrack repository `examples` directory for compiled single file source examples of Qrack usage. (These build with `make all` or `make [example-name]`, with the name of the source file). This is an overview of their content:

grovers - Runs a canonical Grover's search

grovers_lookup - Runs a variant of Grover's search that searches a lookup table for a target value

ordered_list_search - Runs Grover's search base case iterations on an ordered list, potentially recovering a theoretical "big-O" complexity reduction of x2 over the canonical binary search.

pearson32 - Hashes a value according to the Pearson 32 bit hashing algorithm, and searches a superposed set of hashes for a target quality, (like "proof-of-work")

qneuron_classification - Runs a simple quantum neuron classification model task

quantum_associative_memory - Creates an example of an associative memory via a simple quantum neuron model

quantum_perceptron - Demonstrates the single neuron "atom" of Qrack's quantum neural net examples

shors_factoring - Carries out a simulation of Shor's factoring algorithm integers

teleport - Demonstrates quantum teleportation

3.4.2 VM6502Q Examples

The quantum enabled cc65 compiler provides a mechanism to both compile the examples as well as develop new programs to execute on the *vm6502q* virtual machine. These changes live on the *6502q* branch.

Start by compiling the *cc65* repository and the *vm6502q* virtual machine:

```
   cc65/ $ git checkout 6502q
   cc65/ $ make
         ...
vm6502q/ $ make
```

Then, make the various examples:

```
examples/ $ cd hello_c && make
          # OR if to directly execute within the emulator
examples/ $ cd hello_c && make run
          ...
```

```
hello world
^C
Interrupted at e002

Emulation performance stats is OFF.

*-------------*----------------------*----------*----------*
|  PC: $e002  |  Acc: $0e (00001110)  |  X: $55  |  Y: $0c  |
*-------------*----------------------*----------*----------*
|  NVQBDIZC   | :
|  00000100   | :
*-------------*

Stack: $f7
       [03 04 03 04 e2 00 fe 01 ]

I/O status: enabled,  at: $e000,  local echo: OFF.
Graphics status: disabled,  at: $e002
ROM: disabled. Range: $d000 - $dfff.
Op-code execute history: disabled.
-------------------------------------+----------------------------------------
   C - continue,  S - step           |    A - set address for next step
   G - go/cont. from new address     |    N - go number of steps, P - IRQ
   I - toggle char I/O emulation     |    X - execute from new address
   T - show I/O console              |    B - blank (clear) screen
   E - toggle I/O local echo         |    F - toggle registers animation
   J - set animation delay           |    M - dump memory, W - write memory
   K - toggle ROM emulation          |    R - show registers, Y - snapshot
   L - load memory image             |    O - display op-code exec. history
   D - disassemble code in memory    |    Q - quit, 0 - reset, H - help
   V - toggle graphics emulation     |    U - enable/disable exec. history
   Z - enable/disable debug traces   |    1 - enable/disable perf. stats
   2 - display debug traces          |    ? - show this menu
-------------------------------------+----------------------------------------
> q
Thank you for using VM65.
```

Use *Ctrl-C* to bring up the in-VM menu, and *q* to exit.

Creating a new example

- Copy the `prototype/` directory to your example name, renaming the `.cfg` file to match the source file.
- Change `prototype` in `Makefile` to be the basename of your cfg and source file.

- Adjust the `project.cfg` file as necessary for memory sizing.

3.5 Implementation

3.5.1 QEngine

A *Qrack::QEngine* stores a set of permutation basis complex number coefficients and operates on them with bit gates and register-like methods.

The state vector indicates the probability and phase of all possible pure bit permutations, numbered from 0 to $2^N - 1$, by simple binary counting. All operations except measurement should be "unitary," except measurement. They should be representable as a unitary matrix acting on the state vector. Measurement, and methods that involve measurement, should be the only operations that break unitarity. As a rule-of-thumb, this means an operation that doesn't rely on measurement should be "reversible." That is, if a unitary operation is applied to the state, their must be a unitary operation to map back from the output to the exact input. In practice, this means that most gate and register operations entail simply direct exchange of state vector coefficients in a one-to-one manner. (Sometimes, operations involve both a one-to-one exchange and a measurement, like the *QInterface::SetBit* method, or the logical comparison methods.)

A single bit gate essentially acts as a 2×2 matrix between the 0 and 1 states of a single bits. This can be acted independently on all pairs of permutation basis state vector components where all bits are held fixed while 0 and 1 states are paired for the bit being acted on. This is "embarassingly parallel."

To determine how state vector coefficients should be exchanged in register-wise operations, essentially, we form bitmasks that are applied to every underlying possible permutation state in the state vector, and act an appropriate bitwise transformation on them. The result of the bitwise transformation tells us which input permutation coefficients should be mapped to each output permutation coefficient. Acting a bitwise transformation on the input index in the state vector array, we return the array index for the output, and we move the double precision complex number at the input index to the output index. The transformation of the array indexes is basically the classical computational bit transformation implied by the operation. In general, this is again "embarrassingly parallel" over fixed bit values for bits that are not directly involved in the operation. To ease the process of exchanging coefficients, we allocate a new duplicate permutation state array vector, which we output values into and replace the original state vector with at the end.

The act of measurement draws a random double against the probability of a bit or string of bits being in the 1 state. To determine the probability of a bit being in the 1 state, sum the probabilities of all permutation states where the bit is equal to 1. The probablity of a state is equal to the complex norm of its coefficient in the state vector. When the bit is determined to be 1 by drawing a random number against the bit probability, all permutation coefficients for which the bit would be equal to 0 are set to zero. The original probabilities of all states in which the bit is 1 are added together, and every coefficient in the state vector is then divided by this total to "normalize" the probablity back to 1 (or 100%).

In the ideal, acting on the state vector with only unitary matrices would preserve the overall norm of the permutation state vector, such that it would always exactly equal 1, such that on. In practice, floating point error could "creep up" over many operations. To correct we this, Qrack can optionally normalize its state vector, depending on constructor arguments. Specifically, normalization is enabled in tandem with floating point error mitigation that floors very small probability amplitudes to exactly 0, below the estimated level of typical systematic float error for a gate like "H." In fact, to save computational overhead, since most operations entail iterating over the entire permutation state vector once, we can calculate the norm on the fly on one operation, finish with the overall normalization constant in hand, and apply the normalization constant on the next operation, thereby avoiding having to loop twice in every operation.

Qrack has been implemented with `float` precision complex numbers by default. Optional use of `double` precision costs us basically one additional qubit, entailing twice as many potential bit permutations, on the same system. However, double precision complex numbers naturally align to the width of SIMD intrinsics. It is up to the developer, whether precision and alignment with SIMD or else one additional qubit on a system is more important.

3.5.2 QUnit

Qrack::QUnit is a "fundamentally" optimized layer on top of *Qrack::QEngine* types. *QUnit* optimizations include a broadly developed, practical realization of "Schmidt decomposition," (see [Pednault2017],) per-qubit basis transformation with gate commutation, 2-qubit controlled gate buffering and "fusion," optimizing out global phase effects that have no effect on physical "observables," (i.e. the expectation values of Hermitian operators,) a classically efficient SWAP still equivalent to the quantum operation, and many "syngeristic" and incidental points of optimization on top of these general approaches. Publication of an academic report on Qrack and its performance is planned soon, but the *Qrack::QUnit* source code is freely publicly available to inspect.

3.5.3 VM6502Q Opcodes

This extension of the MOS 6502 instruction set honors all legal (as well as undocumented) opcodes of the original chip. See [6502ASM] for the classical opcodes.

The accumulator and X register are replaced with qubits. The Y register is left as a classical bit register. A new "quantum mode" and number of new opcodes have been implemented to facilitate quantum computation, documented in *MOS-6502Q Opcodes*.

The quantum mode flag takes the place of the `unused` flag bit in the original 6502 status flag register. When quantum mode is off, the virtual chip should function exactly like the original MOS-6502, so long as the new opcodes are not used. When the quantum mode flag is turned on, the operation of the other status flags changes. An operation that would reset the "zero," "negative," or "overflow" flags to 0 does nothing. An operation that would set these flags to 1 instead flips the phase of the quantum registers if the flags are already on. In quantum mode, these flags can all be manually set or reset with supplementary opcodes, to engage and disengage the conditional phase flip behavior. The "carry" flag functions in addition and subtraction as it does in the original 6502, though it can exist in a state of superposition. A "CoMPare" operation overloads the function of the carry flag in the original 6502. For a "CMP" instruction in the quantum 6502 extension, the carry flag analogously flips quantum phase when set, if the classical "CMP" instruction would usually set the carry flag. The intent of this flag behavior, setting and resetting them to enable conditional phase flips, is meant to enable quantum "amplitude amplification" algorithms based on the usual status flag capabilities of the original chip.

When an operation happens that would necessarily collapse all superposition in a register or a flag, the emulator keeps track of this, so it can know when its emulation is genuinely quantum as opposed to when it is simply an emulation of a quantum computer emulating a 6502. When quantum emulation is redundant overhead on classical emulation, the emulator is aware, and it performs only the necessary classical emulation. When an operation happens that could lead to superposition, the emulator switches back over to full quantum emulation, until another operation which is guaranteed to collapse a register's state occurs.

3.6 Qrack Performance

3.6.1 Abstract

The Qrack quantum simulator is an open-source C++ high performance, general purpose simulation supporting arbitrary numbers of entangled qubits. While there are a variety of other quantum simulators such as [QSharp], [QHiPSTER], and others listed on [Quantiki], Qrack represents a unique offering suitable for applications across the field.

A selection of performance tests are identified for creating comparisons between various quantum simulators. These metrics are implemented and analyzed for Qrack. These experimentally derived results compare favorably against theoretical boundaries, and out-perform naive implementations for many scenarios.

3.6.2 Introduction

There are a growing number of quantum simulators available for research and industry use. Many of them perform quite well for smaller number of qubits, and are suitable for non-rigorous experimental explorations. Fewer projects are suitable as "high performance" candidates in the >32 qubit range. Many rely on the common approach often described as the "Schrödinger method," doubling RAM usage by a factor of 2 per fully interoperable qubit, or else Feynman path integrals, which can become intractible at arbitrary circuit depth. Attempting to build on the work of IBM's *Breaking the 49-Qubit Barrier in the Simulation of Quantum Circuits* [Pednault2017] paper, with more recent attention to potential improvements inspired by Gottesman-Knill stabilizer simulators, Qrack can execute surprisingly general circuits past 32 qubits in width on modest single nodes.

Qrack is an open-source quantum computer simulator option, implemented in C++, supporting integration into other popular compilers and interfaces, suitable for utilization in a wide variety of projects. As such, it is an ideal test-bed for establishing a set of benchmarks useful for comparing performance between various quantum simulators.

Qrack provides a "QEngineCPU" and a "QEngineOCL" that represent non-OpenCL and OpenCL base implementations for Schrödinger method simulation. "QHybrid" switches off between these two types internally for best performance at low qubit widths. "QStabilizerHybrid" switches off internally between Gottesman-Knill "stabilizer" simulation and Schrödinger method. For general use cases, the "QUnit" layer provides explicit Schmidt decomposition on top of another engine type (per [Pednault2017]). "QPager" segments a Schrödinger method simulation into equally sized "pages" that can be run on multiple OpenCL devices or multiple maximum allocation segments of a single device, increasing greatest maximally entangled width. A "QEngine" type is always the base layer, and QUnit, QStabilizerHybrid, and QPager types may be layered over these, and over each other.

This version of the Qrack benchmarks contains comparisons against other publicly available simulators, specifically QCGPU, and Qiskit (each with its default simulator, if multiple were available). Qrack has been incorporated as an optional back end for ProjectQ and plugin for Qiskit, in repositories maintained by the developers of Qrack, and benchmarks for their performance will follow.

Reader Guidance

This document is largely targeted towards readers looking for a quantum simulator that desire to establish the expected bounds for various use-cases prior to implementation.

Disclaimers

- Your Mileage May Vary - Any performance metrics here are the result of experiments executed with selected compilation and execution parameters on a system with a degree of variability; execute the supplied benchmarks on the desired target system for accurate performance assessments.
- Benchmarking is Hard - While we've attempted to perform clean and accurate results, bugs and mistakes do occur. If flaws in process are identified, please let us know!

3.6.3 Method

This performance document is meant to be a simple, to-the-point, and preliminary digest of these results. We plan to submit a formal academic report for peer review of these results, in full detail, as soon as we collect sufficient feedback on the preprint. (The originally planned date of submission was in February of 2020, but it seems that COVID-19 has hindered our ability to seek preliminary feedback.) These results were prepared with the generous financial support of the Unitary Fund. However, we offer that our benchmark code is public, largely self-explanatory, and easily reproducible, while we prepare that report. Hence, we release these partial preliminary results now.

100 timed trials of single and parallel gates were run for each qubit count between 4 and 28 qubits. Three tests were performed: the quantum Fourier transform, ("QFT"), random circuits constructed from a universal gate set, and

an idealized approximation of Google's Sycamore chip benchmark, as per [Sycamore]. The benchmarking code is available at https://github.com/vm6502q/simulator-benchmarks. Default build and runtime options were used for all candidates. **Notably, this means Qrack ran at single floating point accuracy whereas QCGPU and Qiskit ran at double floating point accuracy.**

Among AWS virtual machine instances, we sought to find those systems with the lowest possible cost to run the benchmarks for their respective execution times, at or below for the 28 qubit mark. An AWS g4dn.2xlarge running Ubuntu Server 20.04LTS was selected for GPU benchmarks. Benchmarks were collected from March 4, 2021 through March 7, 2021. These results were combined with single gate, N-width gate benchmarks for Qrack, collected overnight from December 19th, 2018 into the morning of December 20th. (The potential difference since December 2018 in these particular Qrack tests reused from then should be insignificant. We took care to try to report fair tests, within cost limitations, but please let us know if you find anything that appears misrepresentative.)

Comparative benchmarks included QCGPU, the Qiskit-Aer GPU simulator, and Qrack's default typically optimal "stack" of a "QUnit" layer on top of "QStabilizerHybrid," on top of "QPager," on top "QHybrid." All of these candidates are GPU-based. CPU-based Cirq was considered for presentation here, but a nonexhaustive experiment on AWS CPU instances advertised as low cost-for-performance failed to return Cirq results on rough order of cost-for-performance of any GPU candidate. However, the author does not feel comfortable concluding on this basis that CPU-based simulation cost could not be made competitve with GPU-based simulation, hence we omit Cirq from our graphs to avoid potential misrepresentation.

QFT benchmarks could be implemented in a straightforward manner on all simulators, and were run as such. Qrack appears to be the only candidate considered for which inputs into the QFT can (drastically) affect its execution time, with permutation basis states being run in much shorter time, for example, hence only Qrack required a more general random input, whereas all other simulators were started in the |0> state. For a sufficiently representatively general test, Qrack instead used registers of single separable qubits intialized with uniformly randomly distributed probability between |0> and |1>, and uniformly randomly distributed phase offset between those states.

Random universal circuits carried out layers of single qubit gates on every qubit in the width of the test, followed by layers randomly selected couplings of (2-qubit) CNOT, CZ, and SWAP, or (3-qubit) CCNOT, eliminating each selected bit for the layer. 20 layers of 1-qubit-plus-multi-qubit iterations were carried out, for each qubit width, for the benchmarks presented here.

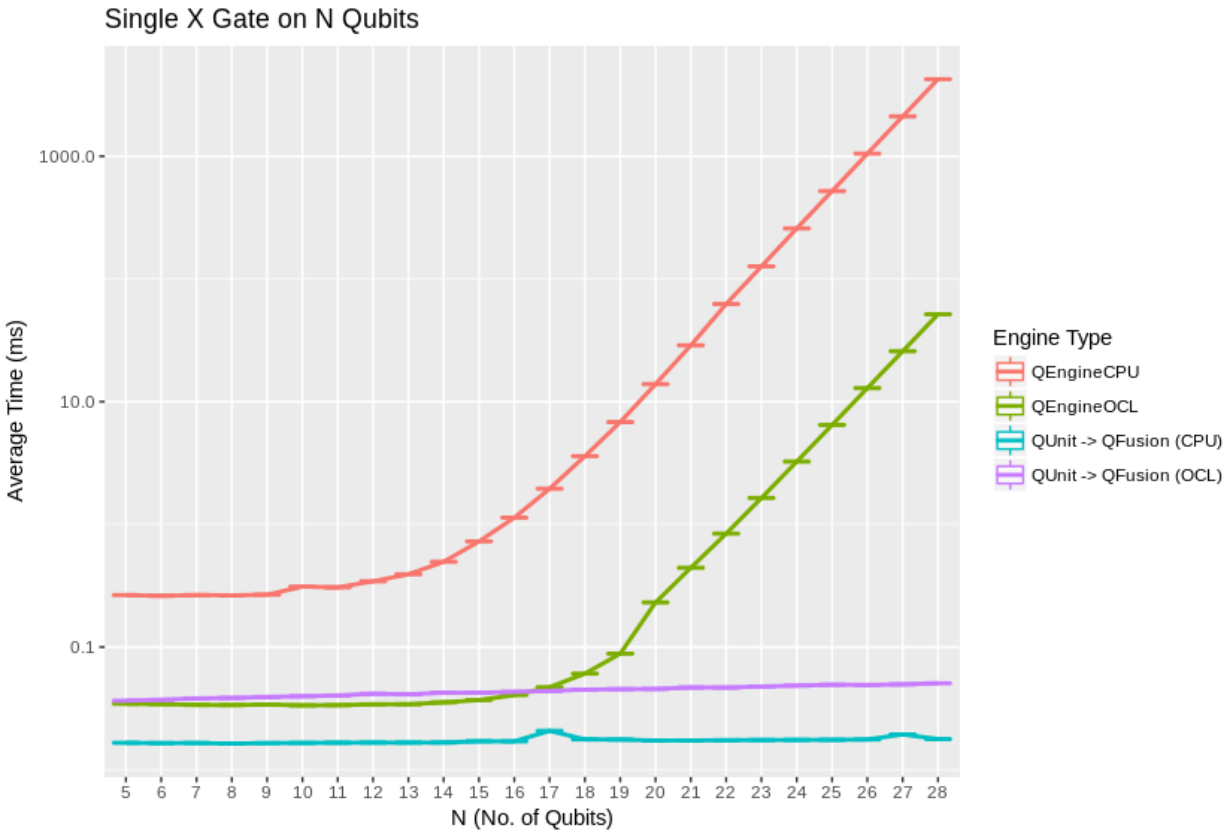
Sycamore circuits were carried out similarly to random universal circuits and the method of the [Sycamore] paper, interleaving 1-qubit followed by 2-qubit layers, to depth of 20 layers each. Whereas as that original source appears to have randomly fixed its target circuit ahead of any trials, and then carried the same pre-selected circuit out repeatedly for the required number of trials, all benchmarks in the case of this report generated their circuits per-iteration on-the-fly, per the selection criteria as read from the text of [Sycamore]. Qrack easily implemented the original Sycamore circuit exactly. By nature of the Schrödinger method simulation used in each other candidate, atomic "convenience method" 1-qubit and 2-qubit gate definitions could potentially easily be added to other candidates for this test, hence **we thought it most representative to make largely performance-irrelevant substitutions of "SWAP" for "iSWAP" for those candidates which did not already define sufficient API convenience methods for "Sycamore" circuits,** without nonrepresentatively complicated gate decompositions. We strongly encourage the reader to inspect and independently execute the simple benchmarking code which was already linked in the beginning of this "Method" section, for total specific detail.

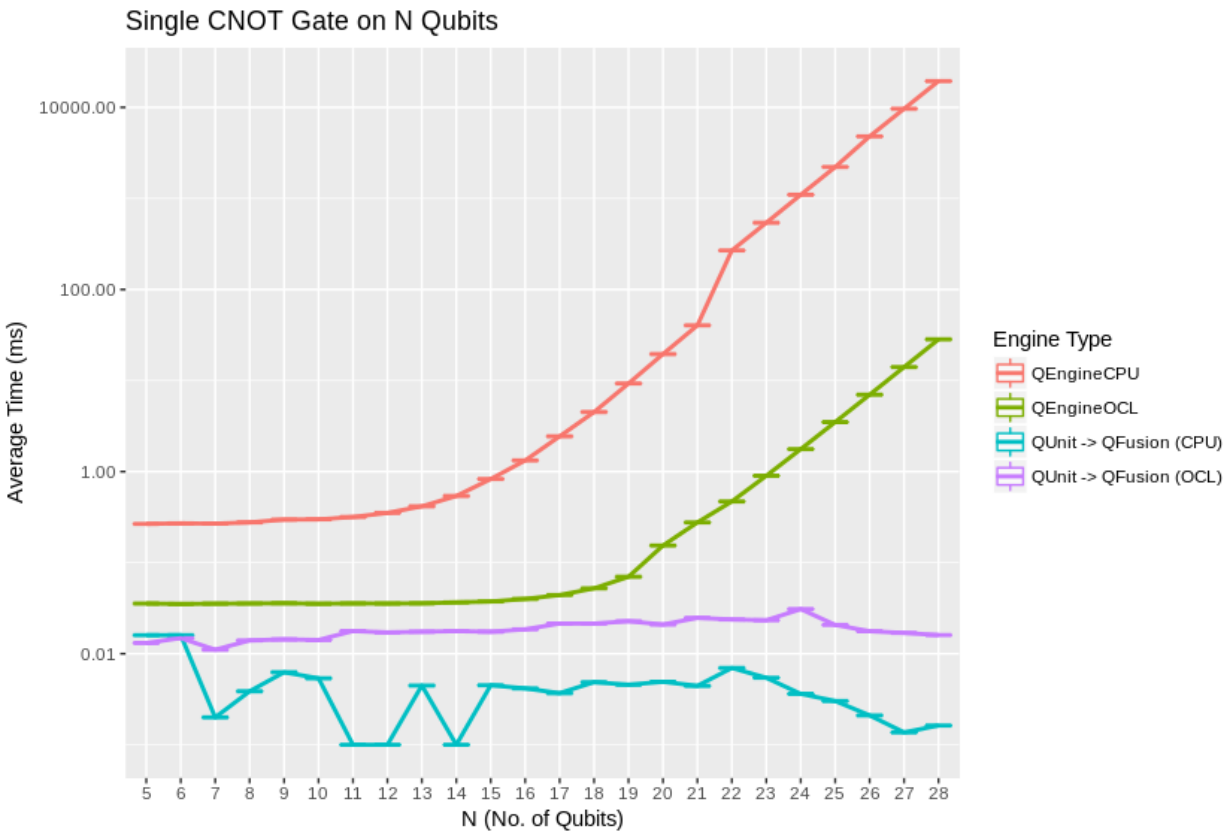
Qrack QEngine type heap usage was established as very closely matching theoretical expections, in earlier benchmarks, and this has not fundamentally changed. QUnit type heap usage varies greatly dependent on use case, though not in significant excess of QEngine types. No representative RAM benchmarks have been established for QUnit types, yet. QEngine Heap profiling was carried out with Valgrind Massif. Heap sampling was limited but ultimately sufficient to show statistical confidence.

3.6.4 Results

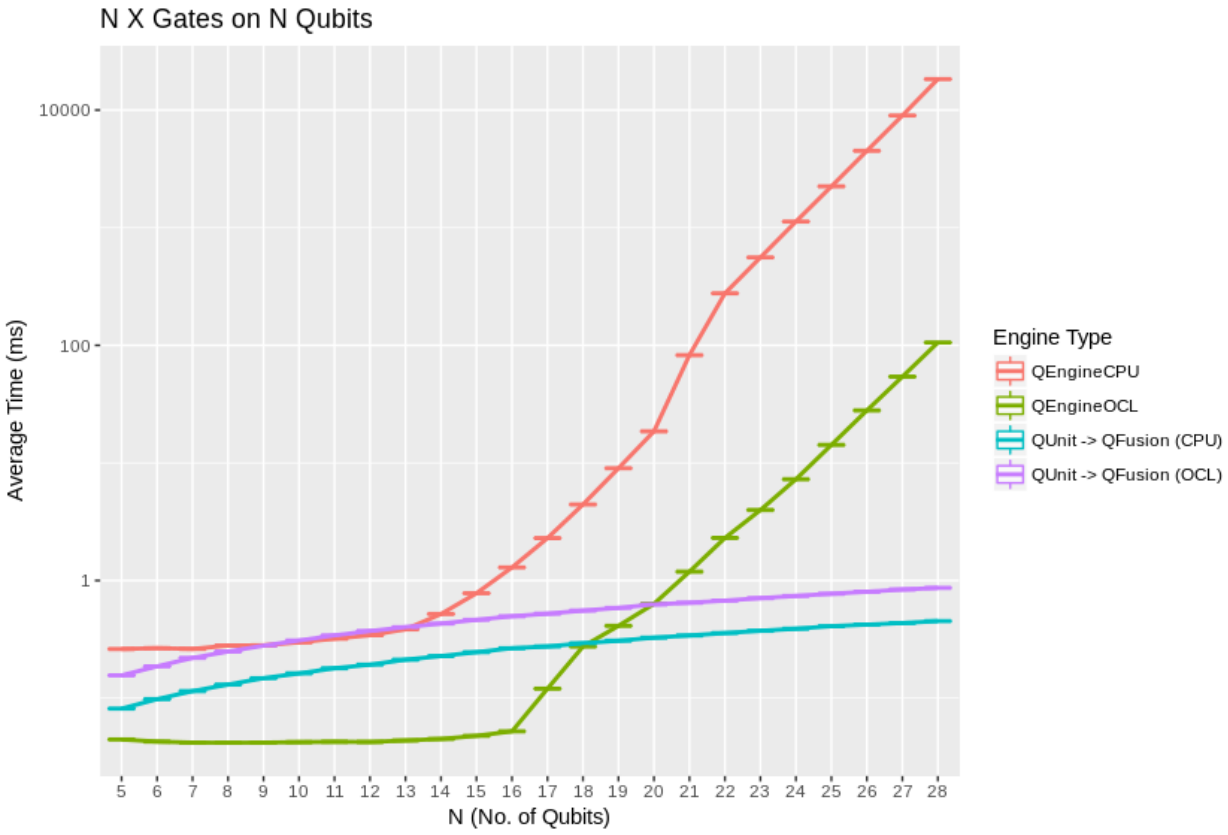
We observed extremely close correspondence with Schrödinger method theoretical complexity and RAM usage considerations for the behavior of QEngine types. QEngineCPU and QEngineOCL require exponential time for a single

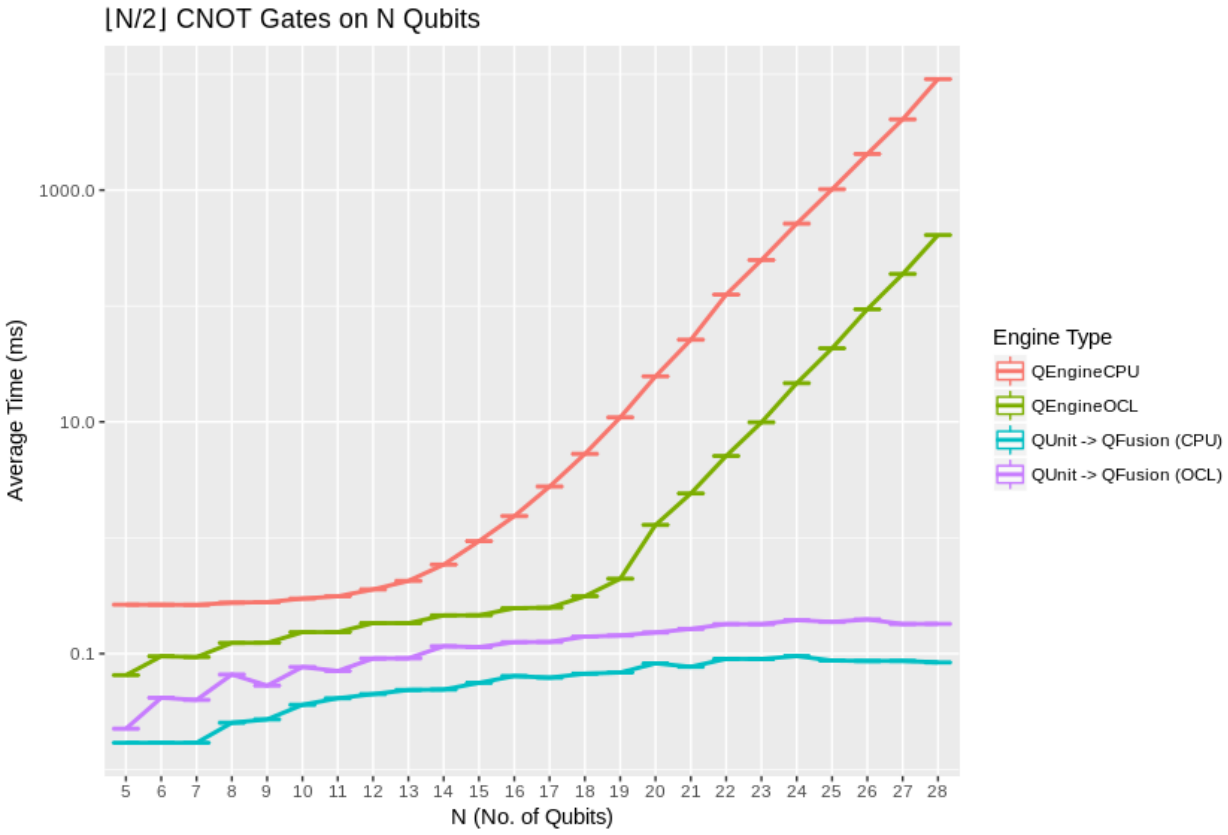
gate on a coherent unit of N qubits. QUnit types with explicitly separated subsystems as per [Pednault2017] show constant time requirements for the same single gate.



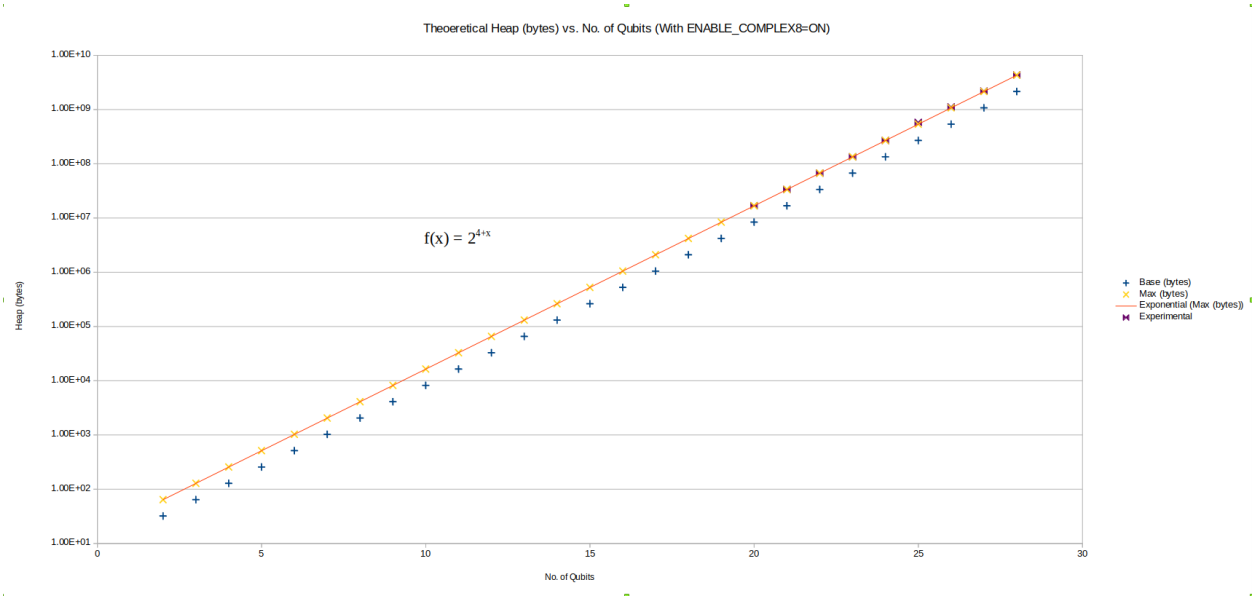


QEngineCPU and QEngineOCL can perform many identical gates in parallel across entangled subsystems for an approximately constant costs, when total qubits in the engine are held fixed as breadth of the parallel gate application is varied. To test this, we can apply parallel gates at once across the full width of a coherent array of qubits. (CNOT is a two bit gate, so $(N-1)/2$ gates are applied to odd numbers of qubits.) Notice in these next graphs how QEngineCPU and QEngineOCL have similar scaling cost as the single gate graphs above, while QUnit types show a linear trend (appearing logarithmic on an exponential axis scale):


N X Gates on N Qubits
Average Time (ms)
10000
100
1
5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28
N (No. of Qubits)
Engine Type
QEngineCPU
QEngineOCL
QUnit -> QFusion (CPU)
QUnit -> QFusion (OCL)

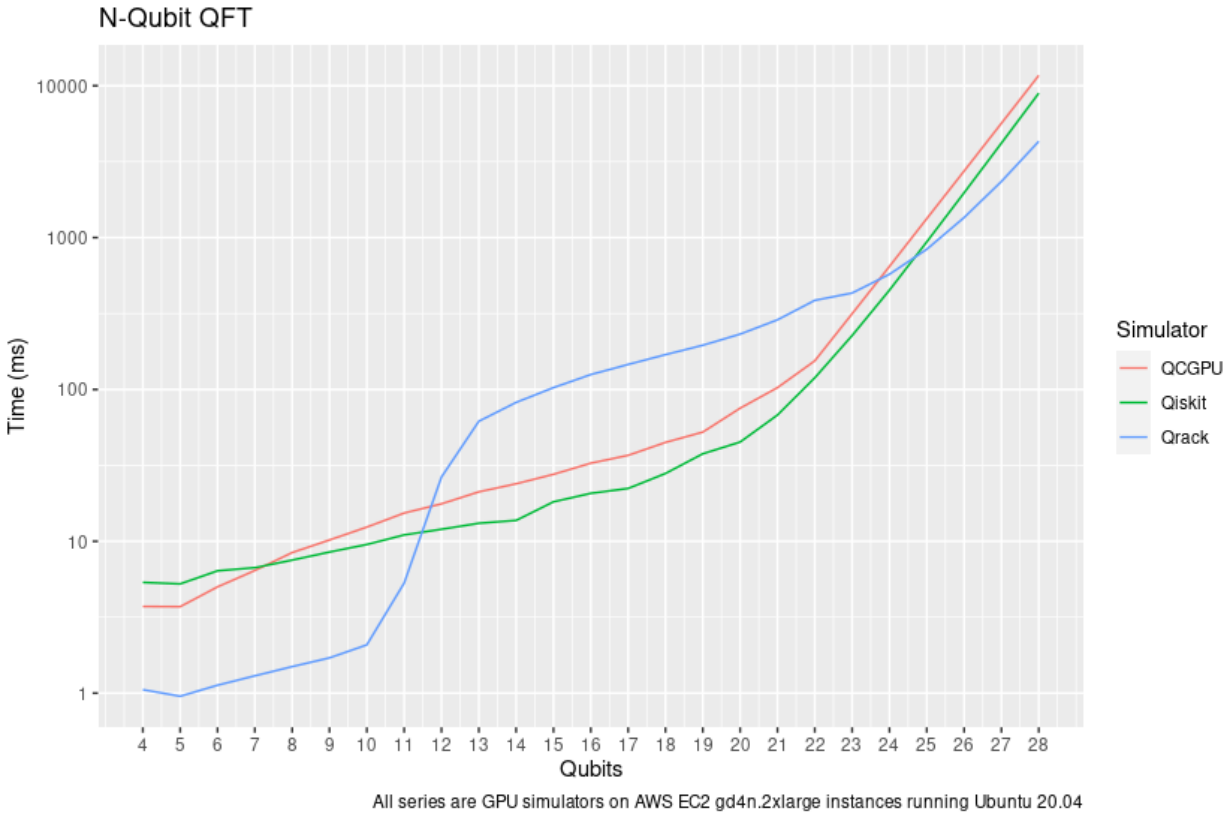

⌊N/2⌋ CNOT Gates on N Qubits
Average Time (ms)
1000.0
10.0
0.1
5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28
N (No. of Qubits)
Engine Type
QEngineCPU
QEngineOCL
QUnit -> QFusion (CPU)
QUnit -> QFusion (OCL)

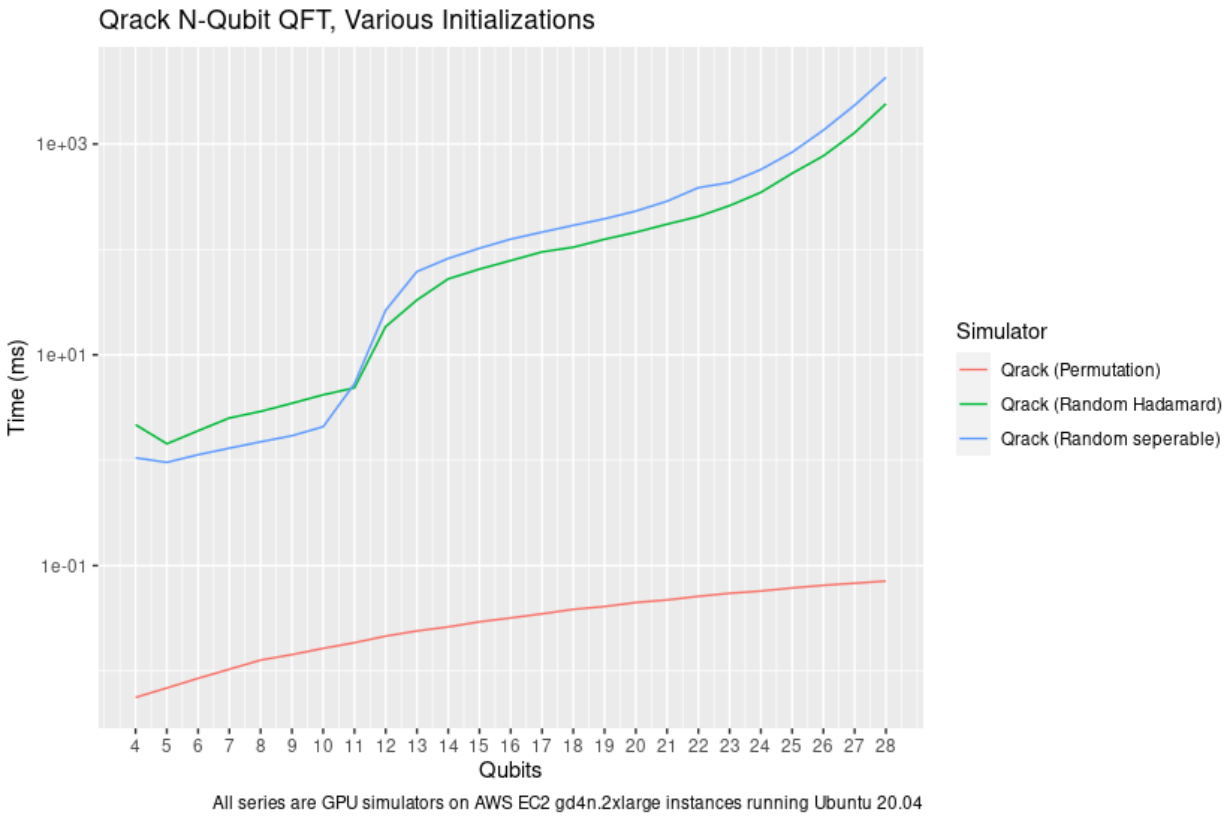
Heap sampling supports theoretical expecations to high confidence. Complex numbers are represented as 2 single (32-bit) or 2 double (64-bit) accuracy floating point types, for real and imaginary components. The use of double or single precision is controlled by a compilation flag. There is one complex number per permutation in a separable subsystem of qubits. QUnit explicitly separates subsystems, while QEngine maintains complex amplitudes for all 2^N permutations of N qubits. QEngines duplicate their state vectors once during many gates, like arithmetic gates, for speed and simplicity where it eases implementation.



QUnit explicitly separates its representation of the quantum state and may operate with much less RAM, but QEngine's RAM usage represents approximately the worst case for QUnit, of maximal entanglement. OpenCL engine types attempt to use memory on the accelerator device instead of general heap when a QEngineOCL instance can fit a single copy of its state vector in a single allocation on the device. On many modern devices, state vectors up to about 1GB in size can be allocated directly on the accelerator device instead of using general heap. "Paging" with QPager allows multiple such maximum allocation segments to be used for the same single simulation. If the normalization option is on, an auxiliary buffer is allocated for normalization that is half the size of the state vector.

The "quantum" (or "discrete") Fourier transform (QFT/DFT) is a realistic and important test case for its direct application in day-to-day industrial computing applications, as well as for being a common processing step in many quantum algorithms.

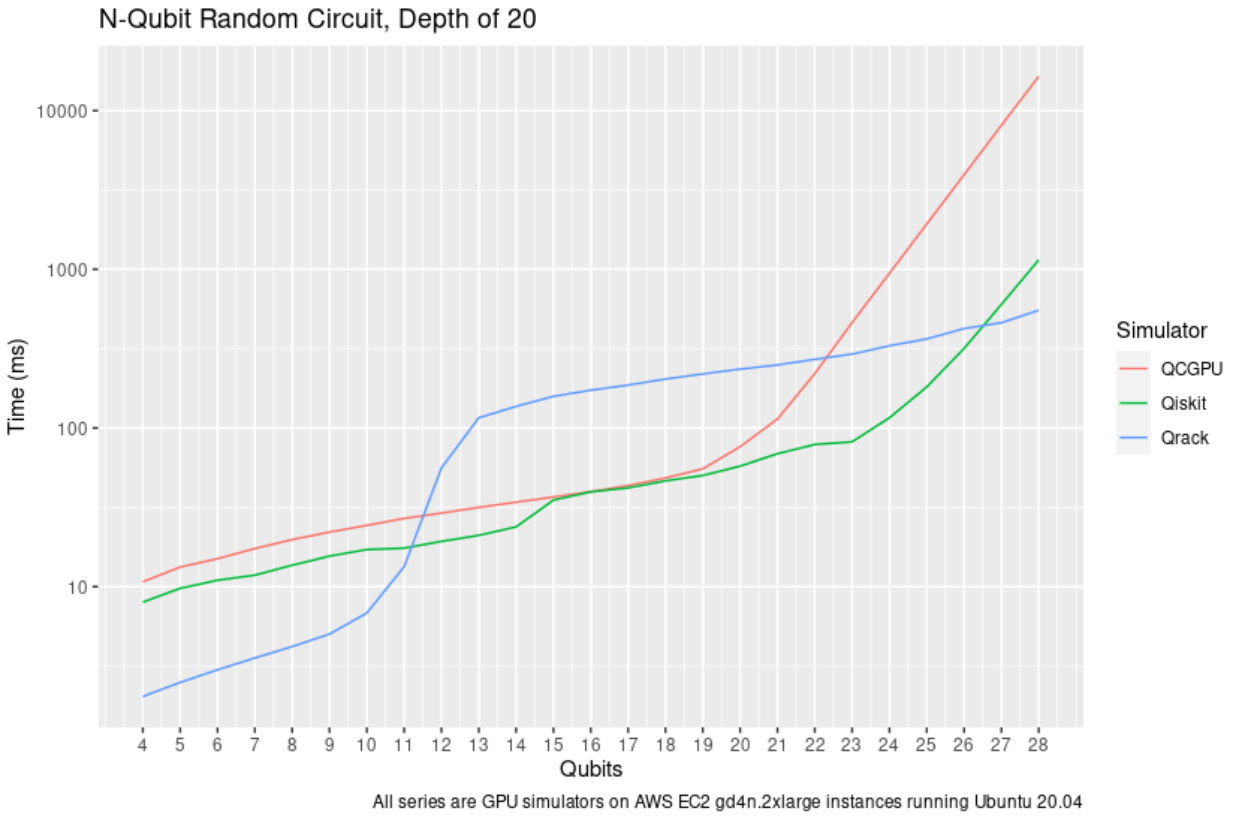




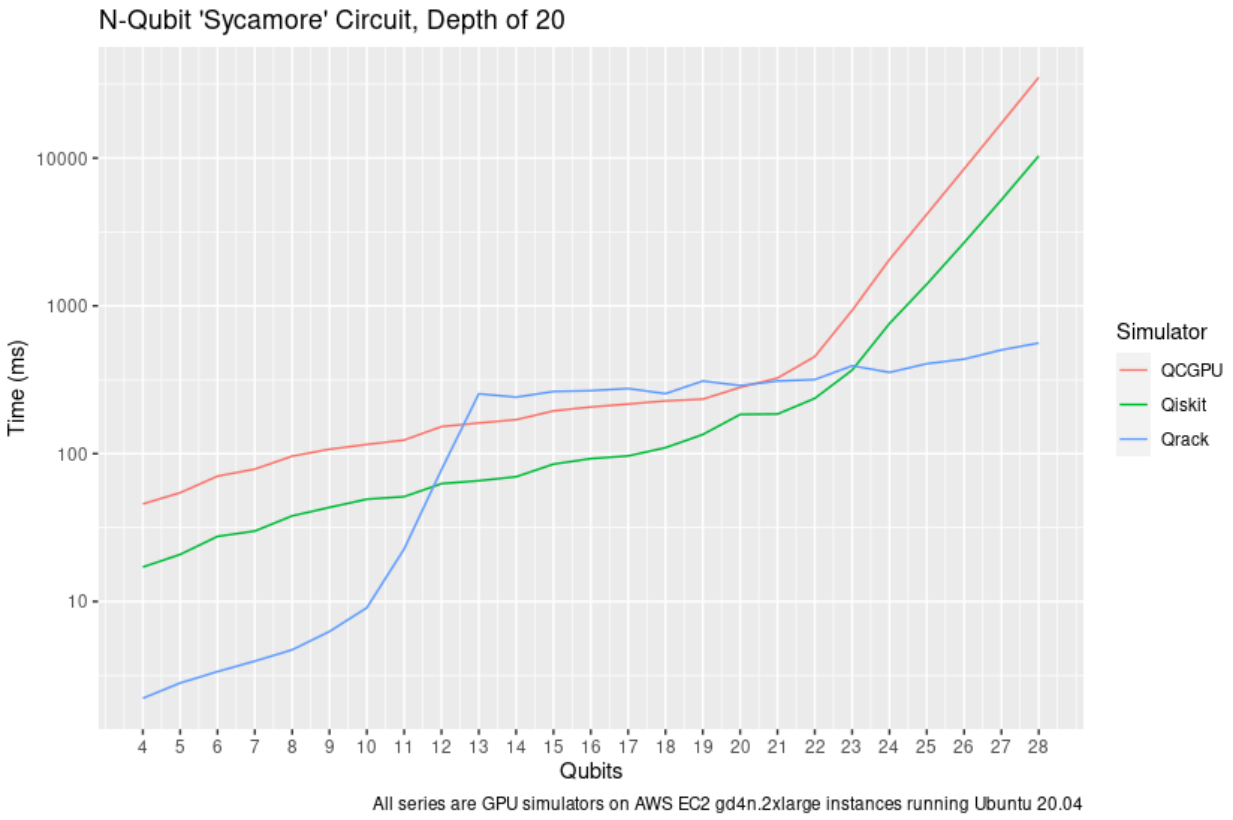
By the 28 qubit level, and at very low qubit widths, Qrack out-performs QCGPU and Qiskit. (Recall that Qrack uses a

representatively "hard" initialization on this test, as described above, whereas permutation basis eigenstate inputs, for example, are much more quickly executed.) Qrack is the only candidate tested which exhibits special case performance on the QFT, as for random permutation basis eigenstate initialization, or initialization via permutation basis eigenstates with random "H" gates applied, before QFT.

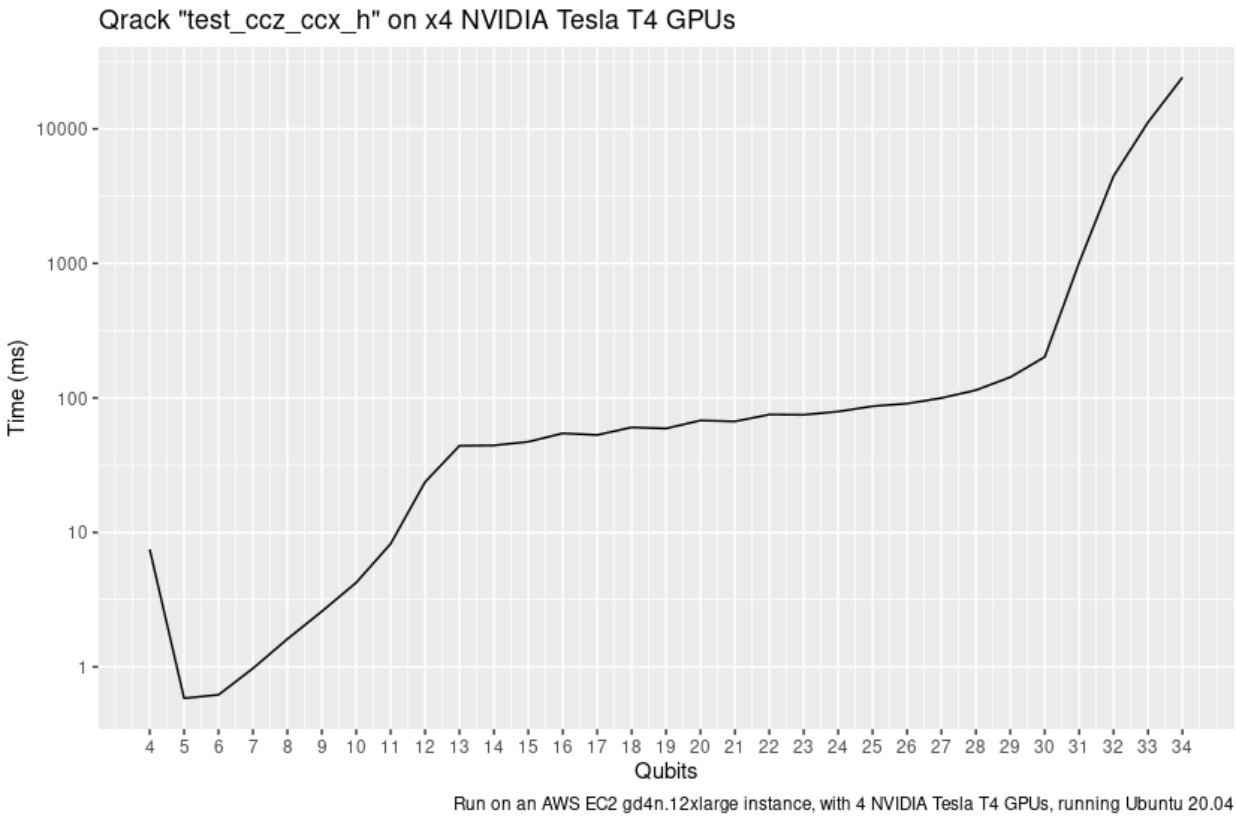
Similarly, on random universal circuits, defined above and in the benchmark repository, Qrack leads over all other candidates at the high qubit width end.



For "Sycamore" circuits, argued by other authors to establish "quantum supremacy" of native quantum hardware, all simulators tested maintain their general performance trends, as above.



To test new capabilities of the "QPager" layer, a slightly different random universal circuit provided in the Qrack benchmark suite was run on a g4dn.12xlarge with 4 NVIDIA Tesla T4 GPUs, to the maximum qubit width possible, which was 34 qubits. The random gate set selected from is {CCZ, CCNOT, CZ, CNOT} and {H, X, Z} for multi- and single qubit gates.



With the recently improved QPager layer, it is often possible to achieve a 2 qubit greater maximum width on the same GPU hardware as a result of using all 4 maximum allocation segments typical of NVIDIA GPUs. QPager combines "pages" of maximum allocation segment on an OpenCL device, which are typically of a much smaller size than the overall RAM of the GPU. Proceeding to higher factors of 2 times page count, it becomes possible to use general RAM heap without exceeding maximum allocation according to the OpenCL standard, as is demonstrated in the graph above. The threshold to cross from single GPU into multi-GPU is 31 qubits, using 2 GPUs at that level, and the threshold for general heap usage is likely crossed at 33 qubits, using the maximum VRAM of 4 NVIDIA T4 GPUs at 32 qubits.

3.6.5 Discussion

Qrack::QUnit succeeds as a novel and fundamentally improved quantum simulation algorithm, over the naive Schrödinger algorithm in special cases. Primarily, QUnit does this by representing its state vector in terms of decomposed subsystems, as well as buffering and commuting Pauli X and Y basis transformations and singly-controlled gates. On user and internal probability checks, QUnit will attempt to separate the representations of independent subsystems by Schmidt decomposition. Further, Qrack will avoid applying phase effects that make no difference to the expectation values of any Hermitian operators, (no difference to "physical observables"). For each bit whose representation is separated this way, we recover a factor of close to or exactly 1/2 the subsystem RAM and gate execution time.

Qrack::QPager, recently, gives several major advantages with or without a Qrack::QUnit layer on top. It usually allows 2 greater maximum qubit width allocation on the same 4-segment GPU RAM store, and it performs surprisingly well for execution speed at high qubit widths. It can also utilize larger system general RAM heap stores than what is available just as GPU RAM.

Qrack has seemingly poor mid-range qubit width performance on the selected g4dn.2xlarge instance, (or, alternatively, good performance at very narrow and very wide ends of the scale, which is not maintained in middle range). As the g4dn.2xlarge only provides 8 "vCPU" units, which is far smaller than a typical PC CPU, mid-range performance might

be alleviated somewhat by a more powerful CPU alongside GPU resources. Further, the use of the QPager layer under QUnit might incur a performance penalty at widths too wide for QHybrid optimization with CPU simulation, but too narrow to see returns from the complexity of QPager. While it might be disappointing that the default "layer stack" for Qrack does not perform best across all qubit widths on the selected AWS EC2 instance, good performance at the very wide and very narrow ends of the scale likely still motivates the adoption of Qrack for HPC and PC simulation.

3.6.6 Further Work

A formal report of the above and additional benchmark results, in much greater detail and specificity, is planned to be submitted for publication as soon as sufficient preliminary peer opinion can be collected on the preprint, in early to mid 2021, thanks to the generous support of the Unitary Fund.

We will maintain systematic comparisons to published benchmarks of quantum computer simulation standard libraries, as they arise.

3.6.7 Conclusion

Per [Pednault2017], and many other attendant and synergistic optimizations engineered specifically in Qrack's QUnit, explicitly separated subsystems of qubits in QUnit have a significant RAM and speed edge in many cases over the Schrödinger algorithm of most popular quantum computer simulators. With QPager, it is possible to achieve even higher qubit widths and execution speeds. Qrack gives very efficient performance on a single node past 32 qubits, up to the limit of maximal entanglement.

3.6.8 Citations

3.7 QInterface

Defined in qinterface.hpp.

This provides a basic interface with a wide-ranging set of functionality

class `Qrack::QInterface`
A "Qrack::QInterface" is an abstract interface exposing qubit permutation state vector with methods to operate on it as by gates and register-like instructions.

See README.md for an overview of the algorithms Qrack employs.

Subclassed by Qrack::QEngine, Qrack::QPager, Qrack::QStabilizerHybrid, Qrack::QUnit

3.7.1 Creating a QInterface

There are five primary implementations of a `QInterface`:

enum `Qrack::QInterfaceEngine`
Enumerated list of supported engines.

Use QINTERFACE_OPTIMAL for the best supported engine.

Values:

QINTERFACE_CPU = 0
Create a QEngineCPU leveraging only local CPU and memory resources.

QINTERFACE_OPENCL
Create a QEngineOCL, leveraging OpenCL hardware to increase the speed of certain calculations.

QINTERFACE_HYBRID
Create a QHybrid, switching between QEngineCPU and QEngineOCL as efficient.

QINTERFACE_STABILIZER_HYBRID
Create a QStabilizerHybrid, switching between a QStabilizer and a QHybrid as efficient.

QINTERFACE_QPAGER
Create a QPager, which breaks up the work of a QEngine into equally sized "pages.".

QINTERFACE_QUNIT
Create a QUnit, which utilizes other *QInterface* classes to minimize the amount of work that's needed for any given operation based on the entanglement of the bits involved.

This, combined with QINTERFACE_OPTIMAL, is the recommended object to use as a library consumer.

QINTERFACE_QUNIT_MULTI
Create a QUnitMulti, which distributes the explicitly separated "shards" of a QUnit across available OpenCL devices.

QINTERFACE_FIRST = QINTERFACE_CPU

QINTERFACE_OPTIMAL_SCHROEDINGER = QINTERFACE_CPU

QINTERFACE_OPTIMAL_SINGLE_PAGE = QINTERFACE_CPU

QINTERFACE_OPTIMAL_G0_CHILD = QINTERFACE_STABILIZER_HYBRID

QINTERFACE_OPTIMAL_G1_CHILD = QINTERFACE_CPU

QINTERFACE_OPTIMAL_G2_CHILD = QINTERFACE_CPU

QINTERFACE_OPTIMAL = QINTERFACE_QUNIT

QINTERFACE_OPTIMAL_MULTI = QINTERFACE_QUNIT_MULTI

QINTERFACE_MAX

These enums can be passed to an allocator to create a `QInterface` of that specified implementation type:
template <typename... *Ts*>
QInterfacePtr `Qrack::`**CreateQuantumInterface**(*QInterfaceEngine engine*, *QInterfaceEngine subengine1*, *QInterfaceEngine subengine2*, Ts... *args*)
Factory method to create specific engine implementations.

3.7.2 Constructors

Warning: doxygenfunction: Unable to resolve multiple matches for function "Qrack::QInterface::QInterface" with arguments (bitLenInt, qrack_rand_gen_ptr, bool, bool, bool, real1) in doxygen xml output for project "qrack" from directory: /tmp/qrack/doc/xml. Potential matches:

```
- Qrack::QInterface::QInterface()
- Qrack::QInterface::QInterface(bitLenInt, qrack_rand_gen_ptr, bool, bool, bool,␣
↪real1_f)
```

3.7.3 Members

StateVectorPtr `Qrack::QEngineCPU::`**stateVec**

3.7.4 Configuration Methods

bitLenInt Qrack::*QInterface*::**GetQubitCount**()
Get the count of bits in this register.

bitCapInt Qrack::*QInterface*::**GetMaxQPower**()
Get the maximum number of basis states, namely n^2 for n qubits.

3.7.5 State Manipulation Methods

virtual void Qrack::*QInterface*::**SetPermutation**(bitCapInt *perm*, complex *phaseFac* = CMPLX_DEFAULT_ARG) = 0
Set to a specific permutation.

virtual void Qrack::*QInterface*::**SetQuantumState**(**const** complex **inputState*) = 0
Set an arbitrary pure quantum state representation.

Warning PSEUDO-QUANTUM

virtual bitLenInt Qrack::*QInterface*::**Compose**(QInterfacePtr *toCopy*)
Combine another *QInterface* with this one, after the last bit index of this one.

"Compose" combines the quantum description of state of two independent *QInterface* objects into one object, containing the full permutation basis of the full object. The "inputState" bits are added after the last qubit index of the *QInterface* to which we "Compose." Informally, "Compose" is equivalent to "just setting another group of qubits down next to the first" without interacting them. Schroedinger's equation can form a description of state for two independent subsystems at once or "separable quantum subsystems" without interacting them. Once the description of state of the independent systems is combined, we can interact them, and we can describe their entanglements to each other, in which case they are no longer independent. A full entangled description of quantum state is not possible for two independent quantum subsystems until we "Compose" them.

"Compose" multiplies the probabilities of the indepedent permutation states of the two subsystems to find the probabilites of the entire set of combined permutations, by simple combinatorial reasoning. If the probablity of the "left-hand" subsystem being in |00> is 1/4, and the probablity of the "right-hand" subsystem being in |101> is 1/8, than the probability of the combined |00101> permutation state is 1/32, and so on for all permutations of the new combined state.

If the programmer doesn't want to "cheat" quantum mechanically, then the original copy of the state which is duplicated into the larger *QInterface* should be "thrown away" to satisfy "no clone theorem." This is not semantically enforced in Qrack, because optimization of an emulator might be acheived by "cloning" "under-the-hood" while only exposing a quantum mechanically consistent API or instruction set.

Returns the quantum bit offset that the *QInterface* was appended at, such that bit 5 in toCopy is equal to offset+5 in this object.

virtual bitLenInt Qrack::*QInterface*::**Compose**(QInterfacePtr *toCopy*, bitLenInt *start*) = 0

virtual void Qrack::*QInterface*::**Decompose**(bitLenInt *start*, QInterfacePtr *dest*) = 0
Minimally decompose a set of contiguous bits from the separably composed unit, into "destination".

Minimally decompose a set of contigious bits from the separably composed unit. The length of this separable unit is reduced by the length of bits decomposed, and the bits removed are output in the destination *QInterface* pointer. The destination object must be initialized to the correct number of bits, in 0 permutation state. For quantum mechanical accuracy, the bit set removed and the bit set left behind should be quantum mechanically "separable."

Like how "Compose" is like "just setting another group of qubits down next to the first," then "Decompose" is like "just moving a few qubits away from the rest." Schroedinger's equation does not require bits to be explicitly

interacted in order to describe their permutation basis, and the descriptions of state of **separable** subsystems, those which are not entangled with other subsystems, are just as easily removed from the description of state. (This is equivalent to a "Schmidt decomposition.")

If we have for example 5 qubits, and we wish to separate into "left" and "right" subsystems of 3 and 2 qubits, we sum probabilities of one permutation of the "left" three over ALL permutations of the "right" two, for all permutations, and vice versa, like so:

$P(|1000> |xy>) = P(|100000>) + P(|100010>) + P(|100001>) + P(|100011>)$.

If the subsystems are not "separable," i.e. if they are entangled, this operation is not well-motivated, and its output is not necessarily defined. (The summing of probabilities over permutations of subsytems will be performed as described above, but this is not quantum mechanically meaningful.) To ensure that the subsystem is "separable," i.e. that it has no entanglements to other subsystems in the *QInterface*, it can be measured with *M()*, or else all qubits *other than* the subsystem can be measured.

virtual void Qrack::*QInterface*::**Dispose**(bitLenInt *start*, bitLenInt *length*) = 0

Minimally decompose a set of contiguous bits from the separably composed unit, and discard the separable bits from index "start" for "length.".

Minimally decompose a set of contigious bits from the separably composed unit. The length of this separable unit is reduced by the length of bits decomposed, and the bits removed are output in the destination *QInterface* pointer. The destination object must be initialized to the correct number of bits, in 0 permutation state. For quantum mechanical accuracy, the bit set removed and the bit set left behind should be quantum mechanically "separable."

Like how "Compose" is like "just setting another group of qubits down next to the first," then "Decompose" is like "just moving a few qubits away from the rest." Schroedinger's equation does not require bits to be explicitly interacted in order to describe their permutation basis, and the descriptions of state of **separable** subsystems, those which are not entangled with other subsystems, are just as easily removed from the description of state. (This is equivalent to a "Schmidt decomposition.")

If we have for example 5 qubits, and we wish to separate into "left" and "right" subsystems of 3 and 2 qubits, we sum probabilities of one permutation of the "left" three over ALL permutations of the "right" two, for all permutations, and vice versa, like so:

$P(|1000> |xy>) = P(|100000>) + P(|100010>) + P(|100001>) + P(|100011>)$.

If the subsystems are not "separable," i.e. if they are entangled, this operation is not well-motivated, and its output is not necessarily defined. (The summing of probabilities over permutations of subsytems will be performed as described above, but this is not quantum mechanically meaningful.) To ensure that the subsystem is "separable," i.e. that it has no entanglements to other subsystems in the *QInterface*, it can be measured with *M()*, or else all qubits *other than* the subsystem can be measured.

virtual void Qrack::*QInterface*::**Dispose**(bitLenInt *start*, bitLenInt *length*, bitCapInt *disposedPerm*) = 0

Dispose a a contiguous set of qubits that are already in a permutation eigenstate.

virtual real1_f Qrack::*QInterface*::**Prob**(bitLenInt *qubitIndex*) = 0

Direct measure of bit probability to be in |1> state.

Warning PSEUDO-QUANTUM

virtual real1_f Qrack::*QInterface*::**ProbAll**(bitCapInt *fullRegister*) = 0

Direct measure of full permutation probability.

Warning PSEUDO-QUANTUM

real1_f Qrack::*QInterface*::**ProbReg**(**const** bitLenInt &*start*, **const** bitLenInt &*length*, **const** bitCapInt &*permutation*)
Direct measure of register permutation probability.

Returns probability of permutation of the register.

Warning PSEUDO-QUANTUM

real1_f Qrack::*QInterface*::**ProbMask**(**const** bitCapInt &*mask*, **const** bitCapInt &*permutation*)
Direct measure of masked permutation probability.

Returns probability of permutation of the mask.

"mask" masks the bits to check the probability of. "permutation" sets the 0 or 1 value for each bit in the mask. Bits which are set in the mask can be set to 0 or 1 in the permutation, while reset bits in the mask should be 0 in the permutation.

Warning PSEUDO-QUANTUM

virtual void Qrack::*QInterface*::**GetProbs**(real1 **outputProbs*) = 0
Get the pure quantum state representation.

Warning PSEUDO-QUANTUM

virtual void Qrack::*QInterface*::**Swap**(bitLenInt *qubitIndex1*, bitLenInt *qubitIndex2*) = 0
Swap values of two bits in register.

virtual void Qrack::*QInterface*::**Swap**(bitLenInt *start1*, bitLenInt *start2*, bitLenInt *length*)
Bitwise swap.

virtual void Qrack::*QInterface*::**ISwap**(bitLenInt *qubitIndex1*, bitLenInt *qubitIndex2*) = 0
Swap values of two bits in register, and apply phase factor of i if bits are different.

virtual void Qrack::*QInterface*::**ISwap**(bitLenInt *start1*, bitLenInt *start2*, bitLenInt *length*)
Bitwise swap.

virtual void Qrack::*QInterface*::**SqrtSwap**(bitLenInt *qubitIndex1*, bitLenInt *qubitIndex2*) = 0
Square root of Swap gate.

virtual void Qrack::*QInterface*::**SqrtSwap**(bitLenInt *start1*, bitLenInt *start2*, bitLenInt *length*)
Bitwise square root of swap.

virtual void Qrack::*QInterface*::**CSwap**(**const** bitLenInt **controls*, **const** bitLenInt &*controlLen*, **const** bitLenInt &*qubit1*, **const** bitLenInt &*qubit2*) = 0
Apply a swap with arbitrary control bits.

virtual void Qrack::*QInterface*::**AntiCSwap**(**const** bitLenInt **controls*, **const** bitLenInt &*controlLen*, **const** bitLenInt &*qubit1*, **const** bitLenInt &*qubit2*) = 0
Apply a swap with arbitrary (anti) control bits.

virtual void Qrack::*QInterface*::**CSqrtSwap**(**const** bitLenInt **controls*, **const** bitLenInt &*controlLen*, **const** bitLenInt &*qubit1*, **const** bitLenInt &*qubit2*) = 0
Apply a square root of swap with arbitrary control bits.

virtual void Qrack::*QInterface*::**AntiCSqrtSwap**(**const** bitLenInt **controls*, **const** bitLenInt &*controlLen*, **const** bitLenInt &*qubit1*, **const** bitLenInt &*qubit2*) = 0
Apply a square root of swap with arbitrary (anti) control bits.

Warning: doxygenfunction: Unable to resolve multiple matches for function "Qrack::QInterface::FSim" with arguments (real1, real1, bitLenInt, bitLenInt) in doxygen xml output for project "qrack" from directory: /tmp/qrack/doc/xml. Potential matches:

```
- virtual void Qrack::QInterface::FSim(real1_f, real1_f, bitLenInt, bitLenInt) = 0
- void Qrack::QInterface::FSim(real1_f, real1_f, bitLenInt, bitLenInt, bitLenInt)
```

Warning: doxygenfunction: Unable to resolve multiple matches for function "Qrack::QInterface::FSim" with arguments (real1, real1, bitLenInt, bitLenInt, bitLenInt) in doxygen xml output for project "qrack" from directory: /tmp/qrack/doc/xml. Potential matches:

```
- virtual void Qrack::QInterface::FSim(real1_f, real1_f, bitLenInt, bitLenInt) = 0
- void Qrack::QInterface::FSim(real1_f, real1_f, bitLenInt, bitLenInt, bitLenInt)
```

virtual void Qrack::*QInterface*::**Reverse**(bitLenInt *first*, bitLenInt *last*)

Reverse all of the bits in a sequence.

virtual bool Qrack::*QInterface*::**TrySeparate**(bitLenInt *start*, bitLenInt *length* = 1, real1_f *error_tol* = REAL1_EPSILON)

Qrack::QUnit types maintain explicit separation of representations of qubits, which reduces memory usage and increases gate speed.

This method is used to manually attempt internal separation of a QUnit subsytem. We attempt a *Decompose()* operation, on a state which might not be separable. If the state is not separable, we abort and return false. Otherwise, we complete the operation, add the separated subsystem back in place into the QUnit "shards," and return true.

This should never change the logical/physical state of the *QInterface*, only possibly its internal representation, for simulation optimization purposes. This is not a truly quantum computational operation, but it also does not lead to nonphysical effects.

Warning PSEUDO-QUANTUM

std::map<bitCapInt, int> Qrack::*QInterface*::**MultiShotMeasureMask**(**const** bitCapInt **qPowers*, **const** bitLenInt *qPowerCount*, **const** unsigned int *shots*)

Statistical measure of masked permutation probability.

"qPowers" contains powers of 2^n, each representing *QInterface* bit "n." The order of these values defines a mask for the result bitCapInt, of 2^0 ~ qPowers[0] to 2^(qPowerCount - 1) ~ qPowers[qPowerCount - 1], in contiguous ascending order. "shots" specifies the number of samples to take as if totally re-preparing the pre-measurement state. This method returns a dictionary with keys, which are the (masked-order) measurement results, and values, which are the number of "shots" that produced that particular measurement result. This method does not "collapse" the state of this *QInterface*. (The idea is to efficiently simulate a potentially statistically random sample of multiple re-preparations of the state right before measurement, and to collect random measurement resutls, without forcing the user to re-prepare or "clone" the state.)

Warning PSEUDO-QUANTUM

3.7.6 Quantum Gates

Note: Most gates offer both a single-bit version taking just the index to the qubit, as well as a register-spanning variant for convienence and performance that performs the gate across a sequence of bits.

Single Gates

virtual void Qrack::*QInterface*::**ApplySingleBit**(**const** complex **mtrx*, bitLenInt *qubitIndex*) = 0

Apply an arbitrary single bit unitary transformation.

virtual void Qrack::*QInterface*::**ApplyControlledSingleBit**(**const** bitLenInt **controls*, **const** bitLenInt *&controlLen*, **const** bitLenInt *&target*, **const** complex **mtrx*) = 0

Apply an arbitrary single bit unitary transformation, with arbitrary control bits.

void Qrack::*QInterface*::**AND**(bitLenInt *inputBit1*, bitLenInt *inputBit2*, bitLenInt *outputBit*)

Quantum analog of classical "AND" gate.

(Assumes the outputBit is in the 0 state)

void Qrack::*QInterface*::**CLAND**(bitLenInt *inputQBit*, bool *inputClassicalBit*, bitLenInt *outputBit*)

Quantum analog of classical "AND" gate.

Takes one qubit input and one classical bit input. (Assumes the outputBit is in the 0 state)

void Qrack::*QInterface*::**OR**(bitLenInt *inputBit1*, bitLenInt *inputBit2*, bitLenInt *outputBit*)

Quantum analog of classical "OR" gate.

(Assumes the outputBit is in the 0 state)

void Qrack::*QInterface*::**CLOR**(bitLenInt *inputQBit*, bool *inputClassicalBit*, bitLenInt *outputBit*)

Quantum analog of classical "OR" gate.

Takes one qubit input and one classical bit input. (Assumes the outputBit is in the 0 state)

void Qrack::*QInterface*::**XOR**(bitLenInt *inputBit1*, bitLenInt *inputBit2*, bitLenInt *outputBit*)

Quantum analog of classical "XOR" gate.

(Assumes the outputBit is in the 0 state)

void Qrack::*QInterface*::**CLXOR**(bitLenInt *inputQBit*, bool *inputClassicalBit*, bitLenInt *outputBit*)

Quantum analog of classical "XOR" gate.

Takes one qubit input and one classical bit input. (Assumes the outputBit is in the 0 state)

virtual bool Qrack::*QInterface*::**M**(bitLenInt *qubitIndex*)

Measurement gate.

Measures the qubit at "qubitIndex" and returns either "true" or "false." (This "gate" breaks unitarity.)

All physical evolution of a quantum state should be "unitary," except measurement. Measurement of a qubit "collapses" the quantum state into either only permutation states consistent with a |0> state for the bit, or else only permutation states consistent with a |1> state for the bit. Measurement also effectively multiplies the overall quantum state vector of the system by a random phase factor, equiprobable over all possible phase angles.

Effectively, when a bit measurement is emulated, Qrack calculates the norm of all permutation state components, to find their respective probabilities. The probabilities of all states in which the measured bit is "0" can be summed to give the probability of the bit being "0," and separately the probabilities of all states in which the measured bit is "1" can be summed to give the probability of the bit being "1." To simulate measurement, a

random float between 0 and 1 is compared to the sum of the probability of all permutation states in which the bit is equal to "1". Depending on whether the random float is higher or lower than the probability, the qubit is determined to be either |0> or |1>, (up to phase). If the bit is determined to be |1>, then all permutation eigenstates in which the bit would be equal to |0> have their probability set to zero, and vice versa if the bit is determined to be |0>. Then, all remaining permutation states with nonzero probability are linearly rescaled so that the total probability of all permutation states is again "normalized" to exactly 100% or 1, (within double precision rounding error). Physically, the act of measurement should introduce an overall random phase factor on the state vector, which is emulated by generating another constantly distributed random float to select a phase angle between 0 and 2 * Pi.

Measurement breaks unitary evolution of state. All quantum gates except measurement should generally act as a unitary matrix on a permutation state vector. (Note that Boolean comparison convenience methods in Qrack such as "AND," "OR," and "XOR" employ the measurement operation in the act of first clearing output bits before filling them with the result of comparison, and these convenience methods therefore break unitary evolution of state, but in a physically realistic way. Comparable unitary operations would be performed with a combination of X and CCNOT gates, also called "Toffoli" gates, but the output bits would have to be assumed to be in a known fixed state, like all |0>, ahead of time to produce unitary logical comparison operations.)

virtual bool Qrack::*QInterface*::**ForceM**(bitLenInt *qubit*, bool *result*, bool *doForce* = true, bool *doApply* = true) = 0

Act as if is a measurement was applied, except force the (usually random) result.

Warning PSEUDO-QUANTUM

virtual void Qrack::*QInterface*::**H**(bitLenInt *qubitIndex*)

Hadamard gate.

Applies a Hadamard gate on qubit at "qubitIndex."

virtual void Qrack::*QInterface*::**X**(bitLenInt *qubitIndex*)

X gate.

Applies the Pauli "X" operator to the qubit at "qubitIndex." The Pauli "X" operator is equivalent to a logical "NOT."

virtual void Qrack::*QInterface*::**Y**(bitLenInt *qubitIndex*)

Y gate.

Applies the Pauli "Y" operator to the qubit at "qubitIndex." The Pauli "Y" operator is similar to a logical "NOT" with permutation phase effects.

virtual void Qrack::*QInterface*::**Z**(bitLenInt *qubitIndex*)

Z gate.

Applies the Pauli "Z" operator to the qubit at "qubitIndex." The Pauli "Z" operator reverses the phase of |1> and leaves |0> unchanged.

void Qrack::*QInterface*::**S**(bitLenInt *qubitIndex*)

S gate.

Apply 1/4 phase rotation.

Applies a 1/4 phase rotation to the qubit at "qubitIndex."

void Qrack::*QInterface*::**IS**(bitLenInt *qubitIndex*)

Inverse S gate.

Apply inverse 1/4 phase rotation.

Applies an inverse 1/4 phase rotation to the qubit at "qubitIndex."

void Qrack::*QInterface*::**T**(bitLenInt *qubitIndex*)
T gate.

Apply 1/8 phase rotation.

Applies a 1/8 phase rotation to the qubit at "qubitIndex."

void Qrack::*QInterface*::**IT**(bitLenInt *qubitIndex*)
Inverse T gate.

Apply inverse 1/8 phase rotation.

Applies an inverse 1/8 phase rotation to the qubit at "qubitIndex."

virtual void Qrack::*QInterface*::**SqrtX**(bitLenInt *qubitIndex*)
Square root of X gate.

Applies the square root of the Pauli "X" operator to the qubit at "qubitIndex." The Pauli "X" operator is equivalent to a logical "NOT."

virtual void Qrack::*QInterface*::**ISqrtX**(bitLenInt *qubitIndex*)
Inverse square root of X gate.

Applies the (by convention) inverse square root of the Pauli "X" operator to the qubit at "qubitIndex." The Pauli "X" operator is equivalent to a logical "NOT."

virtual void Qrack::*QInterface*::**SqrtY**(bitLenInt *qubitIndex*)
Square root of Y gate.

Applies the square root of the Pauli "Y" operator to the qubit at "qubitIndex." The Pauli "Y" operator is similar to a logical "NOT" with permutation phase effects.

virtual void Qrack::*QInterface*::**ISqrtY**(bitLenInt *qubitIndex*)
Square root of Y gate.

Applies the (by convention) inverse square root of the Pauli "Y" operator to the qubit at "qubitIndex." The Pauli "Y" operator is similar to a logical "NOT" with permutation phase effects.

virtual void Qrack::*QInterface*::**SqrtH**(bitLenInt *qubitIndex*)
Square root of Hadamard gate.

Applies the square root of the Hadamard gate on qubit at "qubitIndex."

virtual void Qrack::*QInterface*::**SqrtXConjT**(bitLenInt *qubitIndex*)
Phased square root of X gate.

Applies T.SqrtX.IT to the qubit at "qubitIndex."

virtual void Qrack::*QInterface*::**ISqrtXConjT**(bitLenInt *qubitIndex*)
Inverse phased square root of X gate.

Applies IT.ISqrtX.T to the qubit at "qubitIndex."

void Qrack::*QInterface*::**CNOT**(bitLenInt *control*, bitLenInt *target*)
Controlled NOT gate.

Controlled not.

If the control is set to 1, the target bit is NOT-ed or X-ed.

void Qrack::*QInterface*::**AntiCNOT**(bitLenInt *control*, bitLenInt *target*)
Anti controlled NOT gate.

"Anti-controlled not" - Apply "not" if control bit is zero, do not apply if control bit is one.

If the control is set to 0, the target bit is NOT-ed or X-ed.

void Qrack::*QInterface*::**CCNOT**(bitLenInt *control1*, bitLenInt *control2*, bitLenInt *target*)
Doubly-controlled NOT gate.

Doubly-controlled not.

If both controls are set to 1, the target bit is NOT-ed or X-ed.

void Qrack::*QInterface*::**AntiCCNOT**(bitLenInt *control1*, bitLenInt *control2*, bitLenInt *target*)
Anti doubly-controlled NOT gate.

"Anti-doubly-controlled not" - Apply "not" if control bits are both zero, do not apply if either control bit is one.

If both controls are set to 0, the target bit is NOT-ed or X-ed.

void Qrack::*QInterface*::**CY**(bitLenInt *control*, bitLenInt *target*)
Controlled Y gate.

Apply controlled Pauli Y matrix to bit.

If the "control" bit is set to 1, then the Pauli "Y" operator is applied to "target."

void Qrack::*QInterface*::**CZ**(bitLenInt *control*, bitLenInt *target*)
Controlled Z gate.

Apply controlled Pauli Z matrix to bit.

If the "control" bit is set to 1, then the Pauli "Z" operator is applied to "target."

Warning: doxygenfunction: Unable to resolve multiple matches for function "Qrack::QInterface::RT" with arguments (real1, bitLenInt) in doxygen xml output for project "qrack" from directory: /tmp/qrack/doc/xml. Potential matches:

```
- virtual void Qrack::QInterface::RT(real1_f, bitLenInt, bitLenInt)
- void Qrack::QInterface::RT(real1_f, bitLenInt)
```

void Qrack::*QInterface*::**RTDyad**(int *numerator*, int *denomPower*, bitLenInt *qubitIndex*)
Dyadic fraction phase shift gate.

Dyadic fraction "phase shift gate" - Rotates as e^(i*(M_PI * numerator) / 2^denomPower) around |1> state.

Rotates as $\exp\left(i * \pi * numerator / 2^{denomPower}\right)$ around |1> state.

Warning: doxygenfunction: Unable to resolve multiple matches for function "Qrack::QInterface::CRT" with arguments (real1, bitLenInt, bitLenInt) in doxygen xml output for project "qrack" from directory: /tmp/qrack/doc/xml. Potential matches:

```
- virtual void Qrack::QInterface::CRT(real1_f, bitLenInt, bitLenInt, bitLenInt)
- void Qrack::QInterface::CRT(real1_f, bitLenInt, bitLenInt)
```

void Qrack::*QInterface*::**CRTDyad**(int *numerator*, int *denomPower*, bitLenInt *control*, bitLenInt *target*)
Controlled dyadic fraction "phase shift gate".

Controlled dyadic "phase shift gate" - if control bit is true, rotates target bit as e^(i*(M_PI * numerator) / 2^denomPower) around |1> state.

If control bit is set to 1, rotates target bit as $\exp\left(i * \pi * numerator / 2^{denomPower}\right)$ around |1> state.

Warning: doxygenfunction: Unable to resolve multiple matches for function "Qrack::QInterface::RX" with arguments (real1, bitLenInt) in doxygen xml output for project "qrack" from directory: /tmp/qrack/doc/xml. Potential matches:

```
- virtual void Qrack::QInterface::RX(real1_f, bitLenInt, bitLenInt)
- void Qrack::QInterface::RX(real1_f, bitLenInt)
```

void `Qrack::QInterface::RXDyad`(int *numerator*, int *denomPower*, bitLenInt *qubitIndex*)

Dyadic fraction X axis rotation gate.

Dyadic fraction x axis rotation gate - Rotates around Pauli x axis.

Rotates exp $\left(i * \pi * numerator / 2^{denomPower}\right)$ on Pauli x axis.

Warning: doxygenfunction: Unable to resolve multiple matches for function "Qrack::QInterface::CRX" with arguments (real1, bitLenInt, bitLenInt) in doxygen xml output for project "qrack" from directory: /tmp/qrack/doc/xml. Potential matches:

```
- virtual void Qrack::QInterface::CRX(real1_f, bitLenInt, bitLenInt, bitLenInt)
- void Qrack::QInterface::CRX(real1_f, bitLenInt, bitLenInt)
```

void `Qrack::QInterface::CRXDyad`(int *numerator*, int *denomPower*, bitLenInt *control*, bitLenInt *target*)

Controlled dyadic fraction X axis rotation gate.

Controlled dyadic fraction x axis rotation gate - Rotates around Pauli x axis.

If "control" is 1, rotates as exp $\left(i * \pi * numerator / 2^{denomPower}\right)$ around Pauli x axis.

Warning: doxygenfunction: Unable to resolve multiple matches for function "Qrack::QInterface::RY" with arguments (real1, bitLenInt) in doxygen xml output for project "qrack" from directory: /tmp/qrack/doc/xml. Potential matches:

```
- virtual void Qrack::QInterface::RY(real1_f, bitLenInt, bitLenInt)
- void Qrack::QInterface::RY(real1_f, bitLenInt)
```

void `Qrack::QInterface::RYDyad`(int *numerator*, int *denomPower*, bitLenInt *qubitIndex*)

Dyadic fraction Y axis rotation gate.

Dyadic fraction y axis rotation gate - Rotates around Pauli y axis.

Rotates as exp $\left(i * \pi * numerator / 2^{denomPower}\right)$ around Pauli Y axis.

Warning: doxygenfunction: Unable to resolve multiple matches for function "Qrack::QInterface::CRY" with arguments (real1, bitLenInt, bitLenInt) in doxygen xml output for project "qrack" from directory: /tmp/qrack/doc/xml. Potential matches:

```
- virtual void Qrack::QInterface::CRY(real1_f, bitLenInt, bitLenInt, bitLenInt)
- void Qrack::QInterface::CRY(real1_f, bitLenInt, bitLenInt)
```

void `Qrack::QInterface::CRYDyad`(int *numerator*, int *denomPower*, bitLenInt *control*, bitLenInt *target*)

Controlled dyadic fraction y axis rotation gate.

Controlled dyadic fraction y axis rotation gate - Rotates around Pauli y axis.

If "control" is set to 1, rotates as $\exp\left(i * \pi * numerator / 2^{denomPower}\right)$ around Pauli Y axis.

Warning: doxygenfunction: Unable to resolve multiple matches for function "Qrack::QInterface::RZ" with arguments (real1, bitLenInt) in doxygen xml output for project "qrack" from directory: /tmp/qrack/doc/xml. Potential matches:

```
- virtual void Qrack::QInterface::RZ(real1_f, bitLenInt, bitLenInt)
- void Qrack::QInterface::RZ(real1_f, bitLenInt)
```

void Qrack::*QInterface*::**RZDyad**(int *numerator*, int *denomPower*, bitLenInt *qubitIndex*)

Dyadic fraction Z axis rotation gate.

Dyadic fraction y axis rotation gate - Rotates around Pauli y axis.

Rotates as $\exp\left(i * \pi * numerator / 2^{denomPower}\right)$ around Pauli Z axis.

Warning: doxygenfunction: Unable to resolve multiple matches for function "Qrack::QInterface::CRZ" with arguments (real1, bitLenInt, bitLenInt) in doxygen xml output for project "qrack" from directory: /tmp/qrack/doc/xml. Potential matches:

```
- virtual void Qrack::QInterface::CRZ(real1_f, bitLenInt, bitLenInt, bitLenInt)
- void Qrack::QInterface::CRZ(real1_f, bitLenInt, bitLenInt)
```

void Qrack::*QInterface*::**CRZDyad**(int *numerator*, int *denomPower*, bitLenInt *control*, bitLenInt *target*)

Controlled dyadic fraction Z axis rotation gate.

Controlled dyadic fraction z axis rotation gate - Rotates around Pauli z axis.

If "control" is set to 1, rotates as $\exp\left(i * \pi * numerator / 2^{denomPower}\right)$ around Pauli Z axis.

Warning: doxygenfunction: Unable to resolve multiple matches for function "Qrack::QInterface::Exp" with arguments (real1, bitLenInt) in doxygen xml output for project "qrack" from directory: /tmp/qrack/doc/xml. Potential matches:

```
- virtual void Qrack::QInterface::Exp(real1_f, bitLenInt, bitLenInt)
- void Qrack::QInterface::Exp(bitLenInt *, bitLenInt, bitLenInt, complex *, bool)
- void Qrack::QInterface::Exp(real1_f, bitLenInt)
```

void Qrack::*QInterface*::**ExpDyad**(int *numerator*, int *denomPower*, bitLenInt *qubitIndex*)

Dyadic fraction (identity) exponentiation gate.

Dyadic fraction (identity) exponentiation gate - Applies exponentiation of the identity operator.

Applies $\exp\left(-i * \pi * numerator * I / 2^{denomPower}\right)$, exponentiation of the identity operator

Warning: doxygenfunction: Unable to resolve multiple matches for function "Qrack::QInterface::ExpX" with arguments (real1, bitLenInt) in doxygen xml output for project "qrack" from directory: /tmp/qrack/doc/xml. Potential matches:

```
- virtual void Qrack::QInterface::ExpX(real1_f, bitLenInt, bitLenInt)
- void Qrack::QInterface::ExpX(real1_f, bitLenInt)
```

void Qrack::*QInterface*::**ExpXDyad**(int *numerator*, int *denomPower*, bitLenInt *qubitIndex*)

Dyadic fraction Pauli X exponentiation gate.

Dyadic fraction Pauli X exponentiation gate - Applies exponentiation of the Pauli X operator.

Applies exp $\left(-i * \pi * numerator * \sigma_x / 2^{denomPower}\right)$, exponentiation of the Pauli X operator

Warning: doxygenfunction: Unable to resolve multiple matches for function "Qrack::QInterface::ExpY" with arguments (real1, bitLenInt) in doxygen xml output for project "qrack" from directory: /tmp/qrack/doc/xml. Potential matches:

```
- virtual void Qrack::QInterface::ExpY(real1_f, bitLenInt, bitLenInt)
- void Qrack::QInterface::ExpY(real1_f, bitLenInt)
```

void Qrack::*QInterface*::**ExpYDyad**(int *numerator*, int *denomPower*, bitLenInt *qubitIndex*)

Dyadic fraction Pauli Y exponentiation gate.

Dyadic fraction Pauli Y exponentiation gate - Applies exponentiation of the Pauli Y operator.

Applies exp $\left(-i * \pi * numerator * \sigma_y / 2^{denomPower}\right)$, exponentiation of the Pauli Y operator

Warning: doxygenfunction: Unable to resolve multiple matches for function "Qrack::QInterface::ExpZ" with arguments (real1, bitLenInt) in doxygen xml output for project "qrack" from directory: /tmp/qrack/doc/xml. Potential matches:

```
- virtual void Qrack::QInterface::ExpZ(real1_f, bitLenInt, bitLenInt)
- void Qrack::QInterface::ExpZ(real1_f, bitLenInt)
```

void Qrack::*QInterface*::**ExpZDyad**(int *numerator*, int *denomPower*, bitLenInt *qubitIndex*)

Dyadic fraction Pauli Z exponentiation gate.

Dyadic fraction Pauli Z exponentiation gate - Applies exponentiation of the Pauli Z operator.

Applies exp $\left(-i * \pi * numerator * \sigma_z / 2^{denomPower}\right)$, exponentiation of the Pauli Z operator

void Qrack::*QInterface*::**Exp**(bitLenInt **controls*, bitLenInt *controlLen*, bitLenInt *qubit*, complex **matrix2x2*, bool *antiCtrled* = false)

Imaginary exponentiation of arbitrary 2x2 gate.

Imaginary exponentiate of arbitrary single bit gate.

Applies e^{-i*Op}, where "Op" is a 2x2 matrix, (with controls on the application of the gate).

virtual void Qrack::*QInterface*::**UniformlyControlledSingleBit**(**const** bitLenInt **controls*, **const** bitLenInt *&controlLen*, bitLenInt *qubitIndex*, **const** complex **mtrxs*)

Apply a "uniformly controlled" arbitrary single bit unitary transformation.

(See https://arxiv.org/abs/quant-ph/0312218)

A different unitary 2x2 complex matrix is associated with each permutation of the control bits. The first control bit index in the "controls" array is the least significant bit of the permutation, proceeding to the most significant bit. "mtrxs" is a flat (1-dimensional) array where each subsequent set of 4 components is an arbitrary 2x2 single bit gate associated with the next permutation of the control bits, starting from 0. All combinations of control bits apply one of the 4 component (flat 2x2) matrices. For k control bits, there are therefore 4 * 2^k

complex components in "mtrxs," representing 2^k complex matrices of 2x2 components. (The component ordering in each matrix is the same as all other gates with an arbitrary 2x2 applied to a single bit, such as Qrack::ApplySingleBit.)

void Qrack::*QInterface*::**UniformlyControlledRY**(**const** bitLenInt **controls*, **const** bitLenInt &*controlLen*, bitLenInt *qubitIndex*, **const** real1 **angles*)

Apply a "uniformly controlled" rotation of a bit around the Pauli Y axis.

Uniformly controlled y axis rotation gate - Rotates as e^(-i*/2) around Pauli y axis for each permutation "k" of the control bits.

(See https://arxiv.org/abs/quant-ph/0312218)

A different rotation angle is associated with each permutation of the control bits. The first control bit index in the "controls" array is the least significant bit of the permutation, proceeding to the most significant bit. "angles" is an array where each subsequent component is rotation angle associated with the next permutation of the control bits, starting from 0. All combinations of control bits apply one of rotation angles. For k control bits, there are therefore 2^k real components in "angles."

void Qrack::*QInterface*::**UniformlyControlledRZ**(**const** bitLenInt **controls*, **const** bitLenInt &*controlLen*, bitLenInt *qubitIndex*, **const** real1 **angles*)

Apply a "uniformly controlled" rotation of a bit around the Pauli Z axis.

Uniformly controlled z axis rotation gate - Rotates as e^(-i*/2) around Pauli z axis for each permutation "k" of the control bits.

(See https://arxiv.org/abs/quant-ph/0312218)

A different rotation angle is associated with each permutation of the control bits. The first control bit index in the "controls" array is the least significant bit of the permutation, proceeding to the most significant bit. "angles" is an array where each subsequent component is rotation angle associated with the next permutation of the control bits, starting from 0. All combinations of control bits apply one of rotation angles. For k control bits, there are therefore 2^k real components in "angles."

Register-wide Gates

virtual void Qrack::*QInterface*::**AND**(bitLenInt *inputStart1*, bitLenInt *inputStart2*, bitLenInt *outputStart*, bitLenInt *length*)

Bitwise "AND".

"AND" registers at "inputStart1" and "inputStart2," of "length" bits, placing the result in "outputStart".

virtual void Qrack::*QInterface*::**CLAND**(bitLenInt *qInputStart*, bitCapInt *classicalInput*, bitLenInt *outputStart*, bitLenInt *length*)

Classical bitwise "AND".

"AND" registers at "inputStart1" and the classic bits of "classicalInput," of "length" bits, placing the result in "outputStart".

virtual void Qrack::*QInterface*::**OR**(bitLenInt *inputStart1*, bitLenInt *inputStart2*, bitLenInt *outputStart*, bitLenInt *length*)

Bitwise "OR".

virtual void Qrack::*QInterface*::**CLOR**(bitLenInt *qInputStart*, bitCapInt *classicalInput*, bitLenInt *outputStart*, bitLenInt *length*)

Classical bitwise "OR".

virtual void Qrack::*QInterface*::**XOR**(bitLenInt *inputStart1*, bitLenInt *inputStart2*, bitLenInt *outputStart*, bitLenInt *length*)

Bitwise "XOR".

virtual void Qrack::*QInterface*::**CLXOR**(bitLenInt *qInputStart*, bitCapInt *classicalInput*, bitLenInt *outputStart*, bitLenInt *length*)
Classical bitwise "XOR".

virtual bitCapInt Qrack::*QInterface*::**MReg**(bitLenInt *start*, bitLenInt *length*)
Measure permutation state of a register.

virtual void Qrack::*QInterface*::**H**(bitLenInt *start*, bitLenInt *length*)
Bitwise Hadamard.

virtual void Qrack::*QInterface*::**X**(bitLenInt *start*, bitLenInt *length*)
Bitwise Pauli X (or logical "NOT") operator.

virtual void Qrack::*QInterface*::**Y**(bitLenInt *start*, bitLenInt *length*)
Bitwise Pauli Y operator.

virtual void Qrack::*QInterface*::**Z**(bitLenInt *start*, bitLenInt *length*)
Bitwise Pauli Z operator.

virtual void Qrack::*QInterface*::**S**(bitLenInt *start*, bitLenInt *length*)
Bitwise S operator (1/4 phase rotation)

virtual void Qrack::*QInterface*::**IS**(bitLenInt *start*, bitLenInt *length*)
Bitwise inverse S operator (1/4 phase rotation)

virtual void Qrack::*QInterface*::**T**(bitLenInt *start*, bitLenInt *length*)
Bitwise T operator (1/8 phase rotation)

virtual void Qrack::*QInterface*::**IT**(bitLenInt *start*, bitLenInt *length*)
Bitwise inverse T operator (1/8 phase rotation)

virtual void Qrack::*QInterface*::**SqrtX**(bitLenInt *start*, bitLenInt *length*)
Bitwise square root of Pauli X operator.

virtual void Qrack::*QInterface*::**ISqrtX**(bitLenInt *start*, bitLenInt *length*)
Bitwise inverse square root of Pauli X operator.

virtual void Qrack::*QInterface*::**SqrtY**(bitLenInt *start*, bitLenInt *length*)
Bitwise square root of Pauli Y operator.

virtual void Qrack::*QInterface*::**ISqrtY**(bitLenInt *start*, bitLenInt *length*)
Bitwise inverse square root of Pauli Y operator.

virtual void Qrack::*QInterface*::**SqrtH**(bitLenInt *start*, bitLenInt *length*)
Bitwise square root of Hadamard.

virtual void Qrack::*QInterface*::**SqrtXConjT**(bitLenInt *start*, bitLenInt *length*)
Bitwise phased square root of Pauli X operator.

virtual void Qrack::*QInterface*::**ISqrtXConjT**(bitLenInt *start*, bitLenInt *length*)
Bitwise inverse phased square root of Pauli X operator.

virtual void Qrack::*QInterface*::**CNOT**(bitLenInt *inputBits*, bitLenInt *targetBits*, bitLenInt *length*)
Bitwise controlled-not.

virtual void Qrack::*QInterface*::**AntiCNOT**(bitLenInt *inputBits*, bitLenInt *targetBits*, bitLenInt *length*)
Bitwise "anti-"controlled-not.

virtual void Qrack::*QInterface*::**CCNOT**(bitLenInt *control1*, bitLenInt *control2*, bitLenInt *target*, bitLenInt *length*)
Bitwise doubly controlled-not.

virtual void Qrack::*QInterface*::**AntiCCNOT**(bitLenInt *control1*, bitLenInt *control2*, bitLenInt *target*, bitLenInt *length*)

Bitwise doubly "anti-"controlled-not.

virtual void Qrack::*QInterface*::**CY**(bitLenInt *control*, bitLenInt *target*, bitLenInt *length*)

Bitwise controlled Y gate.

If the "control" bit is set to 1, then the Pauli "Y" operator is applied to "target."

virtual void Qrack::*QInterface*::**CZ**(bitLenInt *control*, bitLenInt *target*, bitLenInt *length*)

Bitwise controlled Z gate.

If the "control" bit is set to 1, then the Pauli "Z" operator is applied to "target."

Warning: doxygenfunction: Unable to resolve multiple matches for function "Qrack::QInterface::RT" with arguments (real1, bitLenInt, bitLenInt) in doxygen xml output for project "qrack" from directory: /tmp/qrack/doc/xml. Potential matches:

```
- virtual void Qrack::QInterface::RT(real1_f, bitLenInt, bitLenInt)
- void Qrack::QInterface::RT(real1_f, bitLenInt)
```

virtual void Qrack::*QInterface*::**RTDyad**(int *numerator*, int *denomPower*, bitLenInt *start*, bitLenInt *length*)

Bitwise dyadic fraction phase shift gate.

Rotates as $\exp\left(i * \pi * numerator / 2^{denomPower}\right)$ around |1> state.

Warning: doxygenfunction: Unable to resolve multiple matches for function "Qrack::QInterface::RX" with arguments (real1, bitLenInt, bitLenInt) in doxygen xml output for project "qrack" from directory: /tmp/qrack/doc/xml. Potential matches:

```
- virtual void Qrack::QInterface::RX(real1_f, bitLenInt, bitLenInt)
- void Qrack::QInterface::RX(real1_f, bitLenInt)
```

virtual void Qrack::*QInterface*::**RXDyad**(int *numerator*, int *denomPower*, bitLenInt *start*, bitLenInt *length*)

Bitwise dyadic fraction X axis rotation gate.

Rotates $\exp\left(i * \pi * numerator / 2^{denomPower}\right)$ on Pauli x axis.

Warning: doxygenfunction: Unable to resolve multiple matches for function "Qrack::QInterface::CRX" with arguments (real1, bitLenInt, bitLenInt, bitLenInt) in doxygen xml output for project "qrack" from directory: /tmp/qrack/doc/xml. Potential matches:

```
- virtual void Qrack::QInterface::CRX(real1_f, bitLenInt, bitLenInt, bitLenInt)
- void Qrack::QInterface::CRX(real1_f, bitLenInt, bitLenInt)
```

virtual void Qrack::*QInterface*::**CRXDyad**(int *numerator*, int *denomPower*, bitLenInt *control*, bitLenInt *target*, bitLenInt *length*)

Bitwise controlled dyadic fraction X axis rotation gate.

If "control" is 1, rotates as $\exp\left(i * \pi * numerator / 2^{denomPower}\right)$ around Pauli x axis.

Warning: doxygenfunction: Unable to resolve multiple matches for function "Qrack::QInterface::RY" with arguments (real1, bitLenInt, bitLenInt) in doxygen xml output for project "qrack" from directory: /tmp/qrack/doc/xml. Potential matches:

```
- virtual void Qrack::QInterface::RY(real1_f, bitLenInt, bitLenInt)
- void Qrack::QInterface::RY(real1_f, bitLenInt)
```

virtual void Qrack::*QInterface*::**RYDyad**(int *numerator*, int *denomPower*, bitLenInt *start*, bitLenInt *length*)

Bitwise dyadic fraction Y axis rotation gate.

Rotates as $\exp\left(i * \pi * numerator / 2^{denomPower}\right)$ around Pauli Y axis.

Warning: doxygenfunction: Unable to resolve multiple matches for function "Qrack::QInterface::CRY" with arguments (real1, bitLenInt, bitLenInt, bitLenInt) in doxygen xml output for project "qrack" from directory: /tmp/qrack/doc/xml. Potential matches:

```
- virtual void Qrack::QInterface::CRY(real1_f, bitLenInt, bitLenInt, bitLenInt)
- void Qrack::QInterface::CRY(real1_f, bitLenInt, bitLenInt)
```

virtual void Qrack::*QInterface*::**CRYDyad**(int *numerator*, int *denomPower*, bitLenInt *control*, bitLenInt *target*, bitLenInt *length*)

Bitwise controlled dyadic fraction y axis rotation gate.

If "control" is set to 1, rotates as $\exp\left(i * \pi * numerator / 2^{denomPower}\right)$ around Pauli Y axis.

Warning: doxygenfunction: Unable to resolve multiple matches for function "Qrack::QInterface::RZ" with arguments (real1, bitLenInt, bitLenInt) in doxygen xml output for project "qrack" from directory: /tmp/qrack/doc/xml. Potential matches:

```
- virtual void Qrack::QInterface::RZ(real1_f, bitLenInt, bitLenInt)
- void Qrack::QInterface::RZ(real1_f, bitLenInt)
```

virtual void Qrack::*QInterface*::**RZDyad**(int *numerator*, int *denomPower*, bitLenInt *start*, bitLenInt *length*)

Bitwise dyadic fraction Z axis rotation gate.

Rotates as $\exp\left(i * \pi * numerator / 2^{denomPower}\right)$ around Pauli Z axis.

Warning: doxygenfunction: Unable to resolve multiple matches for function "Qrack::QInterface::CRZ" with arguments (real1, bitLenInt, bitLenInt, bitLenInt) in doxygen xml output for project "qrack" from directory: /tmp/qrack/doc/xml. Potential matches:

```
- virtual void Qrack::QInterface::CRZ(real1_f, bitLenInt, bitLenInt, bitLenInt)
- void Qrack::QInterface::CRZ(real1_f, bitLenInt, bitLenInt)
```

virtual void Qrack::*QInterface*::**CRZDyad**(int *numerator*, int *denomPower*, bitLenInt *control*, bitLenInt *target*, bitLenInt *length*)

Bitwise controlled dyadic fraction Z axis rotation gate.

If "control" is set to 1, rotates as $\exp\left(i * \pi * numerator / 2^{denomPower}\right)$ around Pauli Z axis.

Warning: doxygenfunction: Unable to resolve multiple matches for function "Qrack::QInterface::Exp" with arguments (real1, bitLenInt, bitLenInt) in doxygen xml output for project "qrack" from directory: /tmp/qrack/doc/xml. Potential matches:

```
- virtual void Qrack::QInterface::Exp(real1_f, bitLenInt, bitLenInt)
- void Qrack::QInterface::Exp(bitLenInt *, bitLenInt, bitLenInt, complex *, bool)
- void Qrack::QInterface::Exp(real1_f, bitLenInt)
```

virtual void Qrack::*QInterface*::**ExpDyad**(int *numerator*, int *denomPower*, bitLenInt *start*, bitLenInt *length*)

Bitwise Dyadic fraction (identity) exponentiation gate.

Applies exp $\left(-i * \pi * numerator * I/2^{denomPower}\right)$, exponentiation of the identity operator

Warning: doxygenfunction: Unable to resolve multiple matches for function "Qrack::QInterface::ExpX" with arguments (real1, bitLenInt, bitLenInt) in doxygen xml output for project "qrack" from directory: /tmp/qrack/doc/xml. Potential matches:

```
- virtual void Qrack::QInterface::ExpX(real1_f, bitLenInt, bitLenInt)
- void Qrack::QInterface::ExpX(real1_f, bitLenInt)
```

virtual void Qrack::*QInterface*::**ExpXDyad**(int *numerator*, int *denomPower*, bitLenInt *start*, bitLenInt *length*)

Bitwise Dyadic fraction Pauli X exponentiation gate.

Applies exp $\left(-i * \pi * numerator * \sigma_x/2^{denomPower}\right)$, exponentiation of the Pauli X operator

Warning: doxygenfunction: Unable to resolve multiple matches for function "Qrack::QInterface::ExpY" with arguments (real1, bitLenInt, bitLenInt) in doxygen xml output for project "qrack" from directory: /tmp/qrack/doc/xml. Potential matches:

```
- virtual void Qrack::QInterface::ExpY(real1_f, bitLenInt, bitLenInt)
- void Qrack::QInterface::ExpY(real1_f, bitLenInt)
```

virtual void Qrack::*QInterface*::**ExpYDyad**(int *numerator*, int *denomPower*, bitLenInt *start*, bitLenInt *length*)

Bitwise Dyadic fraction Pauli Y exponentiation gate.

Applies exp $\left(-i * \pi * numerator * \sigma_y/2^{denomPower}\right)$, exponentiation of the Pauli Y operator

Warning: doxygenfunction: Unable to resolve multiple matches for function "Qrack::QInterface::ExpZ" with arguments (real1, bitLenInt, bitLenInt) in doxygen xml output for project "qrack" from directory: /tmp/qrack/doc/xml. Potential matches:

```
- virtual void Qrack::QInterface::ExpZ(real1_f, bitLenInt, bitLenInt)
- void Qrack::QInterface::ExpZ(real1_f, bitLenInt)
```

virtual void Qrack::*QInterface*::**ExpZDyad**(int *numerator*, int *denomPower*, bitLenInt *start*, bitLenInt *length*)

Bitwise Dyadic fraction Pauli Z exponentiation gate.

Applies exp $\left(-i * \pi * numerator * \sigma_z/2^{denomPower}\right)$, exponentiation of the Pauli Z operator

3.7.7 Algorithmic Implementations

void Qrack::*QInterface*::**QFT**(bitLenInt *start*, bitLenInt *length*, bool *trySeparate* = false)
Quantum Fourier Transform - Apply the quantum Fourier transform to the register.

Quantum Fourier Transform - Optimized for going from |0>/|1> to |+>/|-> basis.

"trySeparate" is an optional hit-or-miss optimization, specifically for QUnit types. Our suggestion is, turn it on for speed and memory effciency if you expect the result of the QFT to be in a permutation basis eigenstate. Otherwise, turning it on will probably take longer.

void Qrack::*QInterface*::**IQFT**(bitLenInt *start*, bitLenInt *length*, bool *trySeparate* = false)
Inverse Quantum Fourier Transform - Apply the inverse quantum Fourier transform to the register.

Inverse Quantum Fourier Transform - Quantum Fourier transform optimized for going from |+>/|-> to |0>/|1> basis.

"trySeparate" is an optional hit-or-miss optimization, specifically for QUnit types. Our suggestion is, turn it on for speed and memory effciency if you expect the result of the QFT to be in a permutation basis eigenstate. Otherwise, turning it on will probably take longer.

virtual bitCapInt Qrack::*QInterface*::**IndexedLDA**(bitLenInt *indexStart*, bitLenInt *indexLength*, bitLenInt *valueStart*, bitLenInt *valueLength*, unsigned char **values*, bool *resetValue* = true) = 0
Set 8 bit register bits by a superposed index-offset-based read from classical memory.

"inputStart" is the start index of 8 qubits that act as an index into the 256 byte "values" array. The "outputStart" bits are first cleared, then the separable |input, 00000000> permutation state is mapped to |input, values[input]>, with "values[input]" placed in the "outputStart" register. FOR BEST EFFICIENCY, the "values" array should be allocated aligned to a 64-byte boundary. (See the unit tests suite code for an example of how to align the allocation.)

While a *QInterface* represents an interacting set of qubit-based registers, or a virtual quantum chip, the registers need to interact in some way with (classical or quantum) RAM. IndexedLDA is a RAM access method similar to the X addressing mode of the MOS 6502 chip, if the X register can be in a state of coherent superposition when it loads from RAM.

The physical motivation for this addressing mode can be explained as follows: say that we have a superconducting quantum interface device (SQUID) based chip. SQUIDs have already been demonstrated passing coherently superposed electrical currents. In a sufficiently quantum-mechanically isolated qubit chip with a classical cache, with both classical RAM and registers likely cryogenically isolated from the environment, SQUIDs could (hopefully) pass coherently superposed electrical currents into the classical RAM cache to load values into a qubit register. The state loaded would be a superposition of the values of all RAM to which coherently superposed electrical currents were passed.

In qubit system similar to the MOS 6502, say we have qubit-based "accumulator" and "X index" registers, and say that we start with a superposed X index register. In (classical) X addressing mode, the X index register value acts an offset into RAM from a specified starting address. The X addressing mode of a LoaD Accumulator (LDA) instruction, by the physical mechanism described above, should load the accumulator in quantum parallel with the values of every different address of RAM pointed to in superposition by the X index register. The superposed values in the accumulator are entangled with those in the X index register, by way of whatever values the classical RAM pointed to by X held at the time of the load. (If the RAM at index "36" held an unsigned char value of "27," then the value "36" in the X index register becomes entangled with the value "27" in the accumulator, and so on in quantum parallel for all superposed values of the X index register, at once.) If the X index register or accumulator are then measured, the two registers will both always collapse into a random but valid key-value pair of X index offset and value at that classical RAM address.

Note that a "superposed store operation in classical RAM" is not possible by analagous reasoning. Classical RAM would become entangled with both the accumulator and the X register. When the state of the registers

was collapsed, we would find that only one "store" operation to a single memory address had actually been carried out, consistent with the address offset in the collapsed X register and the byte value in the collapsed accumulator. It would not be possible by this model to write in quantum parallel to more than one address of classical memory at a time.

virtual bitCapInt Qrack::*QInterface*::**IndexedADC**(bitLenInt *indexStart*, bitLenInt *indexLength*, bitLenInt *valueStart*, bitLenInt *valueLength*, bitLenInt *carryIndex*, unsigned char **values*) = 0

Add to entangled 8 bit register state with a superposed index-offset-based read from classical memory.

"inputStart" is the start index of 8 qubits that act as an index into the 256 byte "values" array. The "outputStart" bits would usually already be entangled with the "inputStart" bits via a *IndexedLDA()* operation. With the "inputStart" bits being a "key" and the "outputStart" bits being a value, the permutation state |key, value> is mapped to |key, value + values[key]>. This is similar to classical parallel addition of two arrays. However, when either of the registers are measured, both registers will collapse into one random VALID key-value pair, with any addition or subtraction done to the "value." See *IndexedLDA()* for context.

FOR BEST EFFICIENCY, the "values" array should be allocated aligned to a 64-byte boundary. (See the unit tests suite code for an example of how to align the allocation.)

While a *QInterface* represents an interacting set of qubit-based registers, or a virtual quantum chip, the registers need to interact in some way with (classical or quantum) RAM. IndexedLDA is a RAM access method similar to the X addressing mode of the MOS 6502 chip, if the X register can be in a state of coherent superposition when it loads from RAM. "IndexedADC" and "IndexedSBC" perform add and subtract (with carry) operations on a state usually initially prepared with *IndexedLDA()*.

virtual bitCapInt Qrack::*QInterface*::**IndexedSBC**(bitLenInt *indexStart*, bitLenInt *indexLength*, bitLenInt *valueStart*, bitLenInt *valueLength*, bitLenInt *carryIndex*, unsigned char **values*) = 0

Subtract from an entangled 8 bit register state with a superposed index-offset-based read from classical memory.

"inputStart" is the start index of 8 qubits that act as an index into the 256 byte "values" array. The "outputStart" bits would usually already be entangled with the "inputStart" bits via a *IndexedLDA()* operation. With the "inputStart" bits being a "key" and the "outputStart" bits being a value, the permutation state |key, value> is mapped to |key, value - values[key]>. This is similar to classical parallel addition of two arrays. However, when either of the registers are measured, both registers will collapse into one random VALID key-value pair, with any addition or subtraction done to the "value." See *QInterface::IndexedLDA* for context.

FOR BEST EFFICIENCY, the "values" array should be allocated aligned to a 64-byte boundary. (See the unit tests suite code for an example of how to align the allocation.)

While a *QInterface* represents an interacting set of qubit-based registers, or a virtual quantum chip, the registers need to interact in some way with (classical or quantum) RAM. IndexedLDA is a RAM access method similar to the X addressing mode of the MOS 6502 chip, if the X register can be in a state of coherent superposition when it loads from RAM. "IndexedADC" and "IndexedSBC" perform add and subtract (with carry) operations on a state usually initially prepared with *IndexedLDA()*.

virtual void Qrack::*QInterface*::**Hash**(bitLenInt *start*, bitLenInt *length*, unsigned char **values*) = 0

Transform a length of qubit register via lookup through a hash table.

The hash table must be a one-to-one function, otherwise the behavior of this method is undefined. The value array definition convention is the same as *IndexedLDA()*. Essentially, this is an *IndexedLDA()* operation that replaces the index register with the value register, but the lookup table must therefore be one-to-one, for this operation to be unitary, as required.

void Qrack::*QInterface*::**TimeEvolve**(Hamiltonian *h*, real1_f *timeDiff*)

To define a Hamiltonian, give a vector of controlled single bit gates ("HamiltonianOp" instances) that are applied

by left-multiplication in low-to-high vector index order on the state vector.

As a general point of linear algebra, where A and B are linear operators,

$$e^{i(A+B)t} = e^{iAt}e^{iBt} \quad (3.8)$$

might NOT hold, if the operators A and B do not commute. As a rule of thumb, A will commute with B at least in the case that A and B act on entirely different sets of qubits. However, for defining the intended Hamiltonian, the programmer can be guaranteed that the exponential factors will be applied right-to-left, by left multiplication, in the order

$$e^{-iH_{N-1}t}e^{-iH_{N-2}t}\dots e^{-iH_0t}|\psi\rangle \quad .(3.9)$$

(For example, if A and B are single bit gates acting on the same bit, form their composition into one gate by the intended right-to-left fusion and apply them as a single HamiltonianOp.)

Warning Hamiltonian components might not commute.

3.8 OCLEngine

Defined in common/oclengine.hpp.

This provides a basic interface with a wide-ranging set of functionality.

class Qrack::**OCLEngine**
"Qrack::OCLEngine" manages the single OpenCL context.

3.8.1 Creating an OCLEngine

OCLEngine is a singleton class that manages all OpenCL devices and supported objects, for use in `QEngineOCL` and `QEngineOCLMulti`.

OCLEngine *Qrack::*OCLEngine*::**Instance**()
Get a pointer to the Instance of the singleton. (The instance will be instantiated, if it does not exist yet.)

3.8.2 Configuration Methods

DeviceContextPtr Qrack::*OCLEngine*::**GetDeviceContextPtr**(**const** int *&dev* = -1)
Get a pointer one of the available OpenCL contexts, by its index in the list of all contexts.

"Qrack::OCLEngine" manages the single OpenCL context

std::vector<DeviceContextPtr> Qrack::*OCLEngine*::**GetDeviceContextPtrVector**()
Get the list of all available devices (and their supporting objects).

void Qrack::*OCLEngine*::**SetDeviceContextPtrVector**(std::vector<DeviceContextPtr> *vec*, DeviceContextPtr *dcp* = nullptr)

Set the list of DeviceContextPtr object available for use.

If one takes the result of *GetDeviceContextPtrVector()*, trims items from it, and sets it with this method, (at initialization, before any QEngine objects depend on them,) all resources associated with the removed items are freed.

int Qrack::*OCLEngine*::**GetDeviceCount**()

Get the count of devices in the current list.

void Qrack::*OCLEngine*::**SetDefaultDeviceContext**(DeviceContextPtr *dcp*)

Pick a default device, for QEngineOCL instances that don't specify a preferred device.

3.9 QEngineCPU

Defined in qengine_cpu.hpp.

The API is provided by Qrack::QInterface. This is a general purpose implementation of Qrack::QInterface, without OpenCL.

3.10 QEngineOCL

Defined in qengine_opencl.hpp.

The API is provided by Qrack::QInterface. However, QEngineOCL has a custom constructor:

Qrack::QEngineOCL::**QEngineOCL**(bitLenInt *qBitCount*, bitCapInt *initState*, qrack_rand_gen_ptr *rgp* = nullptr, complex *phaseFac* = CMPLX_DEFAULT_ARG, bool *doNorm* = false, bool *randomGlobalPhase* = true, bool *useHostMem* = false, int *devID* = -1, bool *useHardwareRNG* = true, bool *ignored* = false, real1_f *norm_thresh* = REAL1_EPSILON, std::vector<int> *ignored2* = {}, bitLenInt *ignored3* = 0)

Initialize a Qrack::QEngineOCL object.

Specify the number of qubits and an initial permutation state. Additionally, optionally specify a pointer to a random generator engine object, a device ID from the list of devices in the *OCLEngine* singleton, and a boolean that is set to "true" to initialize the state vector of the object to zero norm.

"devID" is the index of an OpenCL device in the *OCLEngine* singleton, to select the device to run this engine on. If "useHostMem" is set false, as by default, the QEngineOCL will attempt to allocate the state vector object only on device memory. If "useHostMem" is set true, general host RAM will be used for the state vector buffers. If the state vector is too large to allocate only on device memory, the QEngineOCL will attempt to fall back to allocating it in general host RAM.

Warning "useHostMem" is not conscious of allocation by other QEngineOCL instances on the same device. Attempting to allocate too much device memory across too many QEngineOCL instances, for which each instance would have sufficient device resources on its own, will probably cause the program to crash (and may lead to general system instability). For safety, "useHostMem" can be turned on.

3.11 QHybrid

Defined in qhybrid.hpp.

Qrack::QHybrid switches between QEngineCPU and QEngineOCL as optimal. It may be used as sub-engine type with Qrack::QUnit. It supports the standard Qrack::QInterface API.

The parameter "qubitThreshold" is the number of qubits at which QHybrid will automatically switch to GPU operation. A value of "0" will automatically pick this threshold based on best estimates of efficiency.

`Qrack::QHybrid::`**`QHybrid`**(bitLenInt *qBitCount*, bitCapInt *initState* = 0, qrack_rand_gen_ptr *rgp* = nullptr, complex *phaseFac* = CMPLX_DEFAULT_ARG, bool *doNorm* = false, bool *randomGlobalPhase* = true, bool *useHostMem* = false, int *deviceId* = -1, bool *useHardwareRNG* = true, bool *useSparseStateVec* = false, real1_f *norm_thresh* = REAL1_EPSILON, std::vector<int> *ignored* = {}, bitLenInt *qubitThreshold* = 0)

virtual void `Qrack::QHybrid::`**`SwitchModes`**(bool *useGpu*)
Switches between CPU and GPU used modes.

(This will not incur a performance penalty, if the chosen mode matches the current mode.) Mode switching happens automatically when qubit counts change, but Compose() and Decompose() might leave their destination *QInterface* parameters in the opposite mode.

3.12 QUnit

Defined in qunit.hpp.

Qrack::QUnit maintains explicit separation of representation between separable subsystems, when possible and efficient, greatly reducing memory and execution time overhead.

Qrack::QInterface::TrySeparate() is primarily intended for use with Qrack::QUnit.

virtual bool `Qrack::`*`QInterface`*`::`**`TrySeparate`**(bitLenInt *start*, bitLenInt *length* = 1, real1_f *error_tol* = REAL1_EPSILON)
Qrack::QUnit types maintain explicit separation of representations of qubits, which reduces memory usage and increases gate speed.

This method is used to manually attempt internal separation of a QUnit subsytem. We attempt a *Decompose()* operation, on a state which might not be separable. If the state is not separable, we abort and return false. Otherwise, we complete the operation, add the separated subsystem back in place into the QUnit "shards," and return true.

This should never change the logical/physical state of the *QInterface*, only possibly its internal representation, for simulation optimization purposes. This is not a truly quantum computational operation, but it also does not lead to nonphysical effects.

Warning PSEUDO-QUANTUM

3.13 MOS-6502Q Opcodes

Bellow is a list of new and modified opcodes with their binary and function. If an opcode description is not here to specifically state that the opcode collapses register or flag superposition, it can be assumed that it does not. However, if a (non X register indexed instruction would overwrite the value of a register or flag, then superposition would be expected to be overwritten. If an instruction is X register indexed, then in quantum mode, it will operate according to the superposition of the X register.

Table 1: 6502Q New Opcodes

OP	Byte	Mode	Description
HAA	0x02	Implied	Bitwise Hadamard on the Accumulator
HAX	0x03	Implied	Bitwise Hadamard on the X Register
HAY	0x04	Implied	Bitwise Hadamard on the Y Register
SEN	0x0F	Implied	SEt the Negative flag
PXA	0x12	Implied	Apply a bitwise Pauli X on the Accumulator
PXX	0x13	Implied	Apply a bitwise Pauli X on the X Register
PXY	0x0C	Implied	Apply a bitwise Pauli X on the Y Register
HAC	0x17	Implied	Apply a Hadamard gate on the carry flag
PYA	0x1A	Implied	Apply a bitwise Pauli Y on the Accumulator
PYX	0x1B	Implied	Apply a bitwise Pauli Y on the X Register
PYY	0x1C	Implied	Apply a bitwise Pauli Y on the Y Register
CLQ	0x1F	Implied	CLear Quantum mode flag
SEV	0x27	Implied	SEt the oVerflow flag
SEZ	0x2B	Implied	SEt the Zero flag
CLN	0x2F	Implied	CLear the Negative flag
PZA	0x32	Implied	Apply a bitwise Pauli Z on Accumulator
PZX	0x33	Implied	Apply a bitwise Pauli Z on the X Register
PZY	0x34	Implied	Apply a bitwise Pauli Z on the Y Register
RTA	0x3A	Implied	Bitwise quarter rotation on $\|1\rangle$ axis for Accumulator
RTX	0x3B	Implied	Bitwise quarter rotation on $\|1\rangle$ axis for the X Register
RTY	0x3C	Implied	Bitwise quarter rotation on $\|1\rangle$ axis for the Y Register
SEQ	0x1F	Implied	SEt the Quantum mode flag
RXA	0x42	Implied	Bitwise quarter rotation on X axis for Accumulator
RXX	0x43	Implied	Bitwise quarter rotation on X axis for the X Register
RXY	0x44	Implied	Bitwise quarter rotation on X axis for the Y Register
CLZ	0x47	Implied	CLear the Zero flag
RZA	0x5A	Implied	Bitwise quarter rotation on Z axis for Accumulator
RZX	0x5B	Implied	Bitwise quarter rotation on Z axis for the X Register
RZY	0x5C	Implied	Bitwise quarter rotation on Z axis for the Y Register
FTA	0x62	Implied	Quantum Fourier Transform on Accumulator
FTX	0x63	Implied	Quantum Fourier Transform on the X register
FTY	0x64	Implied	Quantum Fourier Transform on the Y register
ADC	0x75	Zero page X addressing	ADd with Carry, Zero Page indexed, will add in superposition if the X register is superposed. Results in the Accumulator and carry flag become entangled with the X register, such that the result of the addition is entangled with the address loaded from in the X register. (Addressing past the zero page loops to the start.)
ADC	0x7D	Absolute X addressing	ADd with Carry, Zero Page indexed, will add in superposition if the X register is superposed. Results in the Accumulator and carry flag become entangled with the X register, such that the result of the addition is entangled with the address loaded from in the X register.
ADC	0x71	Implied Y addressing	ADd with Carry, Implied Y indexed, will add in superposition if the Y register is superposed. Results in the Accumulator and carry flag become entangled with the Y register, such that the result of the addition is entangled with the address loaded from in the Y register.
ADC	0x79	Absolute Y addressing	ADd with Carry, Zero Page indexed, will add in superposition if the Y register is superposed. Results in the Accumulator and carry flag become entangled with the Y register, such that the result of the addition is entangled with the address loaded from in the Y register.

Continued on next page

Table 1 – continued from previous page

OP	Byte	Mode	Description
TXA	0x8A	Implied	Transfer X register to Accumulator, will maintain superposition of the X register, entangling it to be the same as the Accumulator when measured
TXS	0x9A	Implied	Transfer X register to Stack pointer, will also collapse superposition of the X register
TAY	0xA8	Implied	Transfer Accumulator Y register, will maintain superposition of the Accumulator, entangling it to be the same as the Y register when measured
TAX	0x8A	Implied	Transfer Accumulator to X register, will maintain superposition of the Accumulator, entangling it to be the same as the X register when measured
LDA	0xB5	Zero page X addressing	LoaD Accumulator, Zero Page indexed, will load in superposition if the X register is superposed. Results loaded in the Accumulator become entangled with the X register, such that the result of the load is entangled with the address loaded from in the X register. (Addressing past the zero page loops to the start.
LDA	0xBD	Absolute X addressing	LoaD Accumulator, Zero Page indexed, will load in superposition if the X register is superposed. Results loaded in the Accumulator become entangled with the X register, such that the result of the load is entangled with the address loaded from in the X register.
SBC	0xF5	Zero page X addressing	SuBtract with Carry, Zero Page indexed, will subtract in superposition if the X register is superposed. Results in the Accumulator and carry flag become entangled with the X register, such that the result of the addition is entangled with the address loaded from in the X register. (Addressing past the zero page loops to the start.
SBC	0xFD	Absolute X addressing	SuBtract with Carry, Zero Page indexed, will subtract in superposition if the X register is superposed. Results in the Accumulator and carry flag become entangled with the X register, such that the result of the addition is entangled with the address loaded from in the X register.
ADC	0xF1	Implied Y addressing	SuBtract with Carry, Implied Y indexed, will subtract in superposition if the Y register is superposed. Results in the Accumulator and carry flag become entangled with the Y register, such that the result of the subtraction is entangled with the address loaded from in the Y register.
ADC	0xF9	Absolute Y addressing	ADd with Carry, Zero Page indexed, will subtract in superposition if the Y register is superposed. Results in the Accumulator and carry flag become entangled with the Y register, such that the result of the subtraction is entangled with the address loaded from in the Y register.
QZZ	0xF7	Implied	Apply Pauli Z operator to zero flag
QZS	0xFA	Implied	Apply Pauli Z operator to negative flag
QZC	0xFB	Implied	Apply Pauli Z operator to carry flag

Table 2: 6502Q Modified Opcodes

OP	Description
AND	Bitwise AND with the Accumulator, will also collapse the quantum state of the Accumulator
ASL	Arithmetic Shift Left, will also collapse superposition of the carry flag
BIT	The 6502's test BITs opcodes, will also collapse the superposition of the Accumulator
CMP	CoMPare accumulator. If quantum mode is off, this opcode functions as in the original 6502. If quantum mode is on, and if a flag would be set to 1 in the original system, and if this flag is already on, then this instead flips the phase of the quantum registers, for each such flag.
CPX	CoMPare X register. If quantum mode is off, this opcode functions as in the original 6502. If quantum mode is on, and if a flag would be set to 1 in the original system, and if this flag is already on, then this instead flips the phase of the quantum registers, for each such flag.
CPY	CoMPare Y register. If quantum mode is off, this opcode functions as in the original 6502. If quantum mode is on, and if a flag would be set to 1 in the original system, and if this flag is already on, then this instead flips the phase of the quantum registers, for each such flag.
EOR	Bitwise EOR with the Accumulator, will also collapse the quantum state of the Accumulator
LSR	Logical Shift Right, will also collapse superposition of the carry flag
ORA	Bitwise OR with the Accumulator, will also collapse the quantum state of the Accumulator
ROL	ROtate Left, will also collapse superposition of the carry flag
STA	STore Accumulator, will also collapse superposition of the Accumulator
STX	STore X register, will also collapse superposition of the X register
STY	STore Y register, will also collapse superposition of the Y register

Bibliography

[Susskind] Modern Physics: Quantum Mechanics, by Dr. Leonard Susskind

[WiredSummary] Wired's Overview of the Industry

[AlgoZoo] Quantum Algorithm Zoo

[QC10th] Quantum Computing 10th Edition - Nielson and Chuang

[QAVLA] Quantum Algorithms via Linear Algebra: A Primer - Lipton and Regan

[Grover] Grover Search Algorithm

[GroverSummary] Introduction to Implementing Grover's Search Algorithm

[GroverVisual] Visualization of Grover's Search Algorithm

[Broda2016] Broda, Bogusław. "Quantum search of a real unstructured database." The European Physical Journal Plus 131.2 (2016): 38.

[MOS-6502] The 6502 CPU - https://en.wikipedia.org/wiki/MOS_Technology_6502

[6502ASM] 6502 Assembly Reference - http://www.6502.org/tutorials/6502opcodes.html

[Pednault2017] Pednault, Edwin, et al. "Breaking the 49-qubit barrier in the simulation of quantum circuits." arXiv preprint arXiv:1710.05867 (2017).

[Broda2016] Broda, Bogusław. "Quantum search of a real unstructured database." The European Physical Journal Plus 131.2 (2016): 38.

[Pednault2017] Pednault, Edwin, et al. "Breaking the 49-qubit barrier in the simulation of quantum circuits." arXiv preprint arXiv:1710.05867 (2017).

[QSharp] Q#

[QHiPSTER] QHipster

[Quantiki] Quantiki: List of QC simulators

[Sycamore] Arute, Frank, et al. "Quantum supremacy using a programmable superconducting processor"

Index

Q