qrack Documentation

qrack

Aug 04, 2018

Contents

CHAPTER 1

Build Status

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

CHAPTER 2

Introduction

Qrack is a C++ quantum bit 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.

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.

Finally, a 6502 toolchain - based on CC65 - has been modified and enhanced to support both the new opcodes - for the assembler - as well as *C Syntax Enhancements*. This is performed primarily as sandbox/exploratory work to help clarify what quantum computational software engineering might look like as the hardware reaches commoditization.

CHAPTER 3

Copyright

Daniel Strano would like to specifically note that Benn Bollay is almost entirely responsible for the 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

Check out each of the major repositories into a project branch:

```
  / $ mkdir qc
  / $ cd qc
qc/ $ git clone https://github.com/vm6502q/qrack.git
qc/ $ git clone https://github.com/vm6502q/vm6502q.git
qc/ $ git clone https://github.com/vm6502q/examples.git
    # Note: the cc65 repository changes live in the 6502q branch
qc/ $ git clone https://github.com/vm6502q/cc65.git -b 6502q

    # Add a necessary symlink connecting the vm6502q project with qrack
```

(continues on next page)

(continued from previous page)

```
qc/ $ cd vm6502q && ln -s ../qrack

    # vm6502q expects the qrack buildfiles to exist in qrack/build
qc/ $ mkdir qrack/build
qc/ $ cd qrack/build && cmake ..
    # OR if no OpenCL support is enabled
qc/ $ cd qrack/build && cmake -DUSE_OPENCL=OFF ..
```

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 `USE_OPENCL=OFF` environment to `cmake` when building qrack the first time.

Qrack compiles with either double or single accuracy complex numbers. Doubles are 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, such as accelerators. Double vectorization uses AVX, while single vectorization uses SSE 1.0.

To enable float accuracy as opposed to double, run CMake with the appropriate flag:

```
qc/ $ cd qrack/build && cmake -DENABLE_COMPLEX8=ON ..
```

3.1.4 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_OPENCL,␣
↪qubitCount, intPerm, rng);
QInterfacePtr qftReg2 = CreateQuantumInterface(QINTERFACE_OPENCL, QINTERFACE_OPENCL,␣
↪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");
```

(continues on next page)

(continued from previous page)

```
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);

// "deviceIDs" is a std::vector<int> of indices of the desired device in the␣
↪OCLEngine list.
// This also might be used for a simple form of load-balancing, like so:
std::vector<int> deviceIDs(4);
// Three equally-sized sub-engines go to device index 1, and a fourth sub-engine goes␣
↪to device index 0.
// The number of sub-engines must be a power of two.
deviceIDs[0] = 1;
deviceIDs[0] = 1;
deviceIDs[0] = 1;
deviceIDs[0] = 0;
QEngineOCL qEngineMulti = QEngineOCLMulti(qBitCount, initPermutation, deviceIDs,␣
↪random_generator_pointer);
```

3.1.5 Testing

The qrack project has an extensive set of unittests for the various `Qrack::QInterface` gates and simulator methods. This can be executed through running the test suite in the `qrack` project:

```
qrack/build/ $ make test
```

This may take a few minutes to complete, depending on the strength of the system executing the tests.

Note: The unittests, by default, run against all supported engines. If only a specific engine type is desired, the `--disable-opencl` or `--disable-software` command line parameters may be supplied to the `unittest` binary.

3.1.6 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.

3.1.7 Performance

See the extensive *performance analysis and graphs* section.

3.1.8 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.

3.1.9 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

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   | :
*-------------*
```

(continues on next page)

(continued from previous page)

```
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.

3.4.1 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 QInterface

A *Qrack::QInterface* 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, we normalize at least immediately before (and immediately after) measurement operations. Many operations imply only measurements by either 1 or 0 and will therefore not introduce floating point error, but in cases where we multiply by say $1/\sqrt{2}$, we can normalize proactively. 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 `double` precision complex numbers. Use of single precision `float` could get us basically one additional qubit, twice as many 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 implementing a quantum emulator, whether precision and alignment with SIMD or else one additional qubit on a system is more important.

3.5.2 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.

3.5.3 CC65

An assembler for the vm6502q project has been implemented by extending the instruction set of the MOS-6502. To implement the assembler, one can duplicate an assembler implementation for the 6502 and add the new instruction symbols and binary values to the table of implemented instructions.

C Syntax Enhancements

New higher level syntax extensions are under development using the CC65 C compiler for the 6502. These syntax extensions will leverage the quantum parallel LoaD Accumulator ("LDA") instruction, quantum paralell ADd with Carry ("ADC") instruction, and quantum parallel SuBtract with Carry ("SBC") instruction, as well as the amplitude amplification capabilities of vm6502q, using the modified behavior of status flags in "quantum mode." More is to follow soon.

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 for the growing mid-tier range of experimentation in the 20-30 qubit range.

Despite the availability of a selection of implementations, very little has been established when comparing the performance between different simulators. Broadly, the substantial bottlenecks around memory and IO utilization have largely preempted analysis into CPU efficiencies and algorithmic optimizations. There are some exceptions, such as IBM's *Breaking the 49-Qubit Barrier in the Simulation of Quantum Circuits [Pednault2017]* paper.

Qrack provides high performance in the 20-30 qubit range, as well as an open-source implementation in C++ 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.

Future publications will compare the performance of Qrack against other publicly available simulators, as rigorous implementations can be implemented.

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

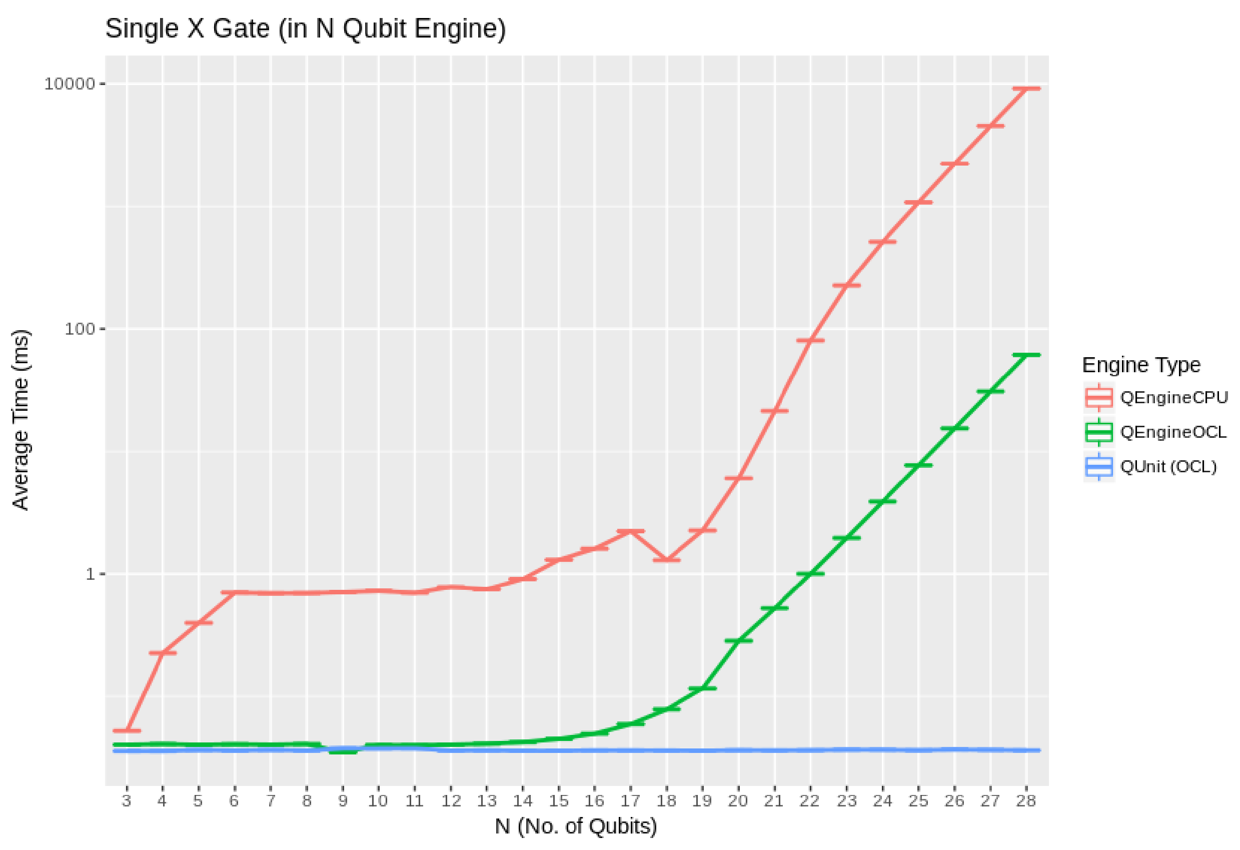
100 timed trials of each method were run for each qubit count between 3 and 28 qubits, on an AWS p3.8xlarge running Ubuntu Server 16.04LTS. The average and quartile boundary values of each set of 100 were recorded and graphed. Grover's search to invert a black box subroutine, or "oracle," was similarly implemented for trials between 3 and 16 qubits. Grover's algorithm was iterated an optimal number of times, vs. qubit count, to maximize probability on a half cycle of the algorithm's period, being $floor\left[\frac{\pi}{4asin\left(1/\sqrt{2^N}\right)}\right]$ iterations for N qubits.

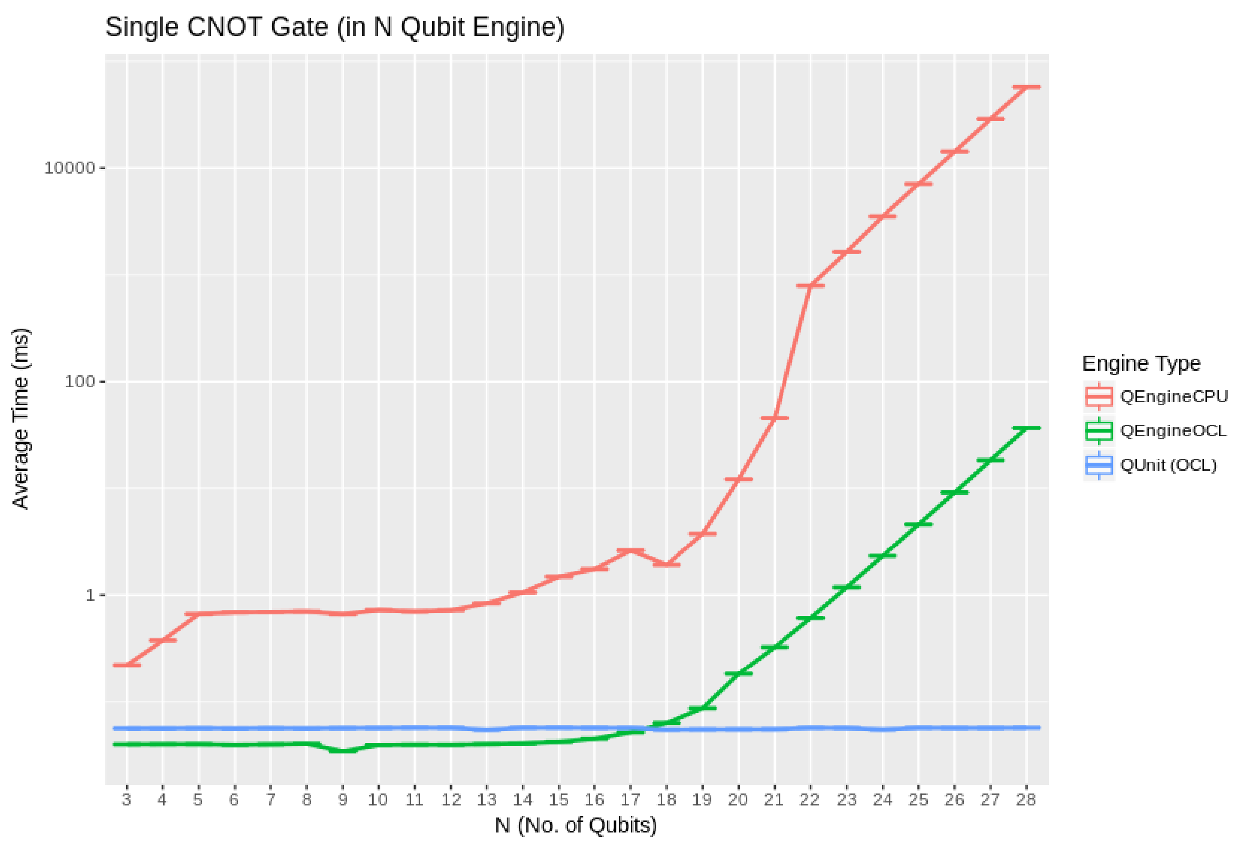
The 16 qubit data point for Grover's search was not included for QEngineCPU, which is not hardware accelerated, because the monetary cost of exhaustive benchmarking is prohibitive, and QEngineCPU as currently implemented now primarily represents a legacy engine that would usually not be used for real application of Qrack. QEngineOCLMulti is experimental and generally performed worse or no better than QEngineOCL, despite running on multiple processors at once. It would also not yet be used for practical applications, so we omit its results, noting that it performs worse than the single device engine. (We will elaborate in the discussion section.)

Heap profiling was carried out with Valgrind Massif. Heap sampling was limited but ultimately sufficient to show statistical confidence.

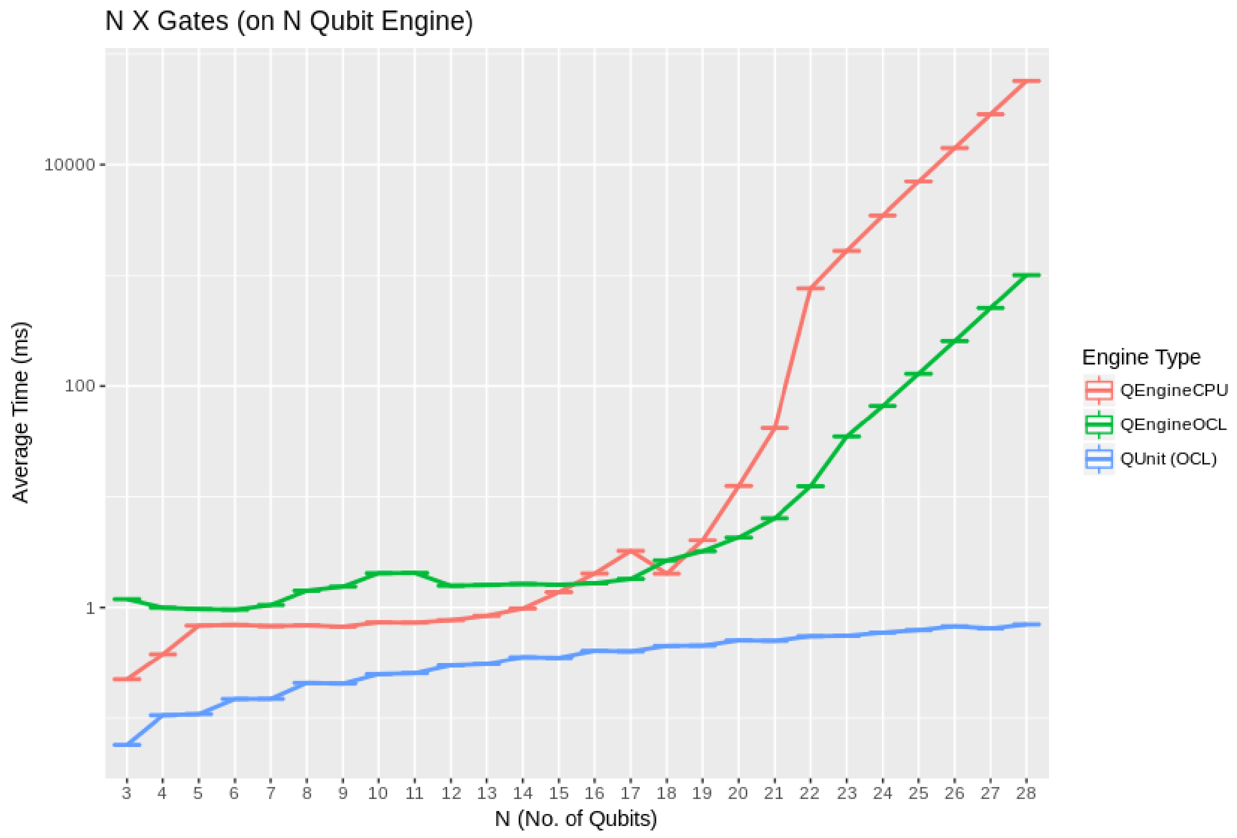
3.6.4 Results

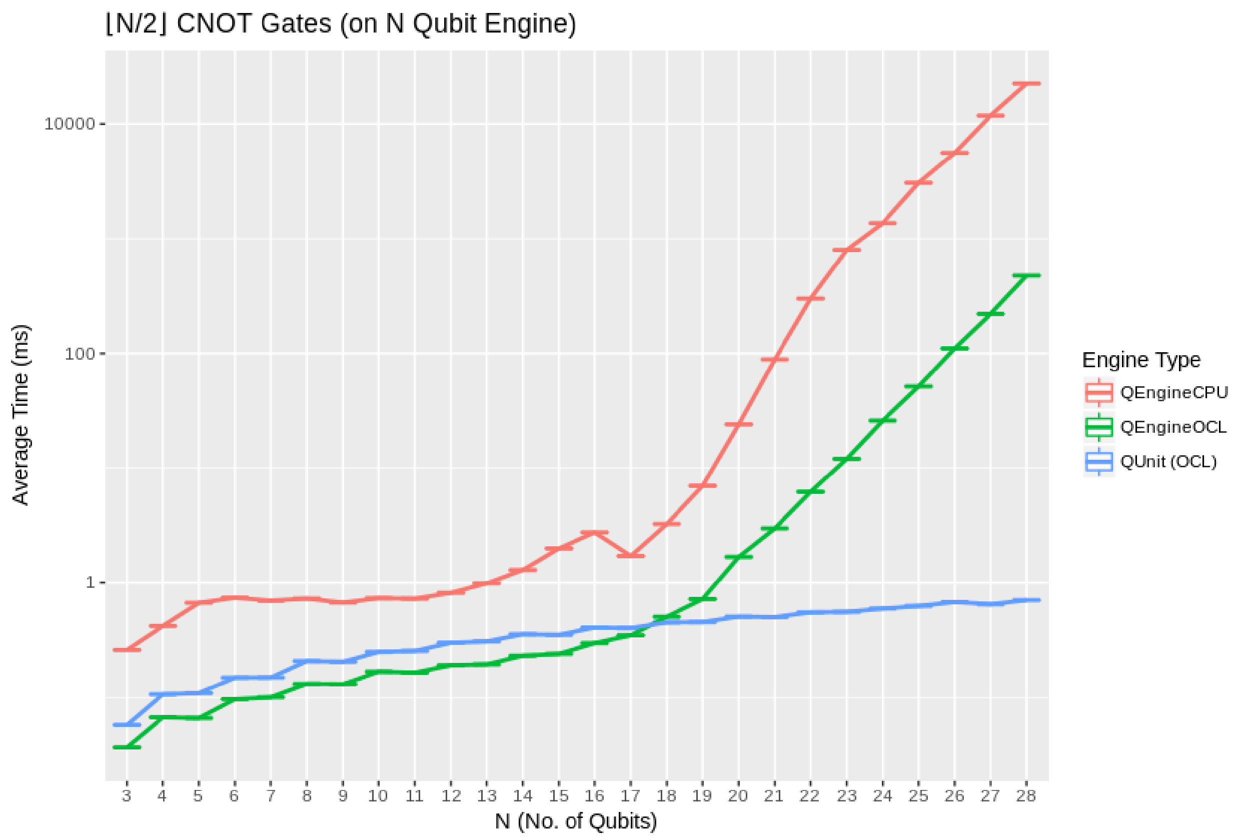
We observed extremely close correspondence with theoretical complexity and RAM usage considerations for the behavior of all engine types. QEngineCPU and QEngineOCL require exponential time for a single gate on a coherent unit of N qubits. QUnit types with explicitly separated subsystems as per *[Pednault2017]* show constant time requirements for the same single gate.


Single X Gate (in N Qubit Engine)
Average Time (ms)
10000
100
1
3 4 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 (OCL)

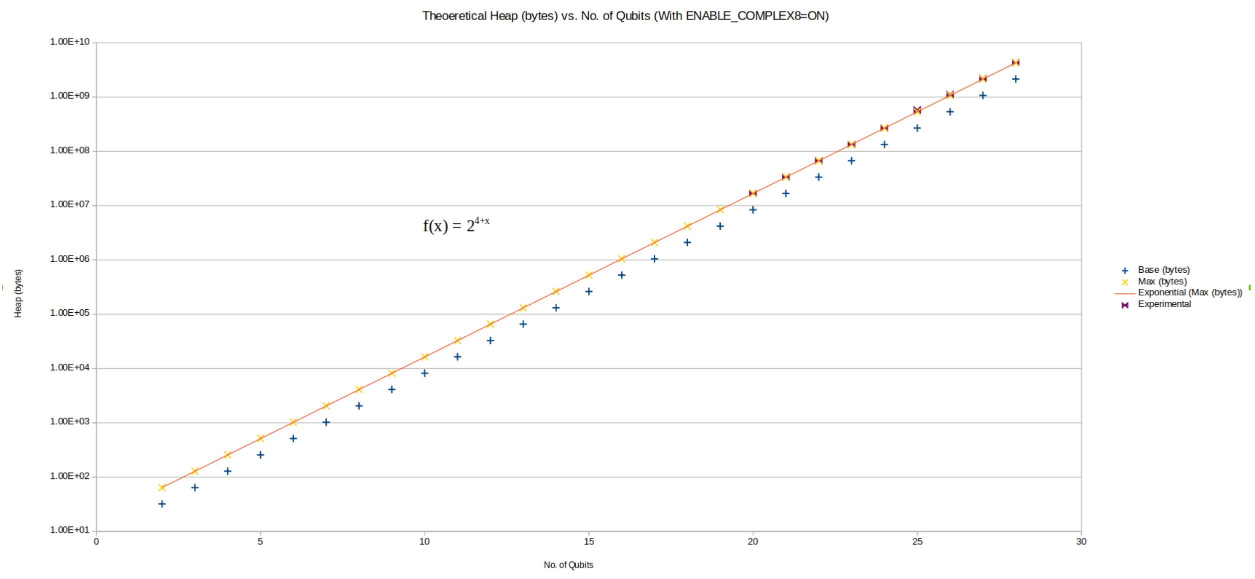

Single CNOT Gate (in N Qubit Engine)
Average Time (ms)
10000
100
1
3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28
N (No. of Qubits)
Engine Type
QEngineCPU
QEngineOCL
QUnit (OCL)

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):

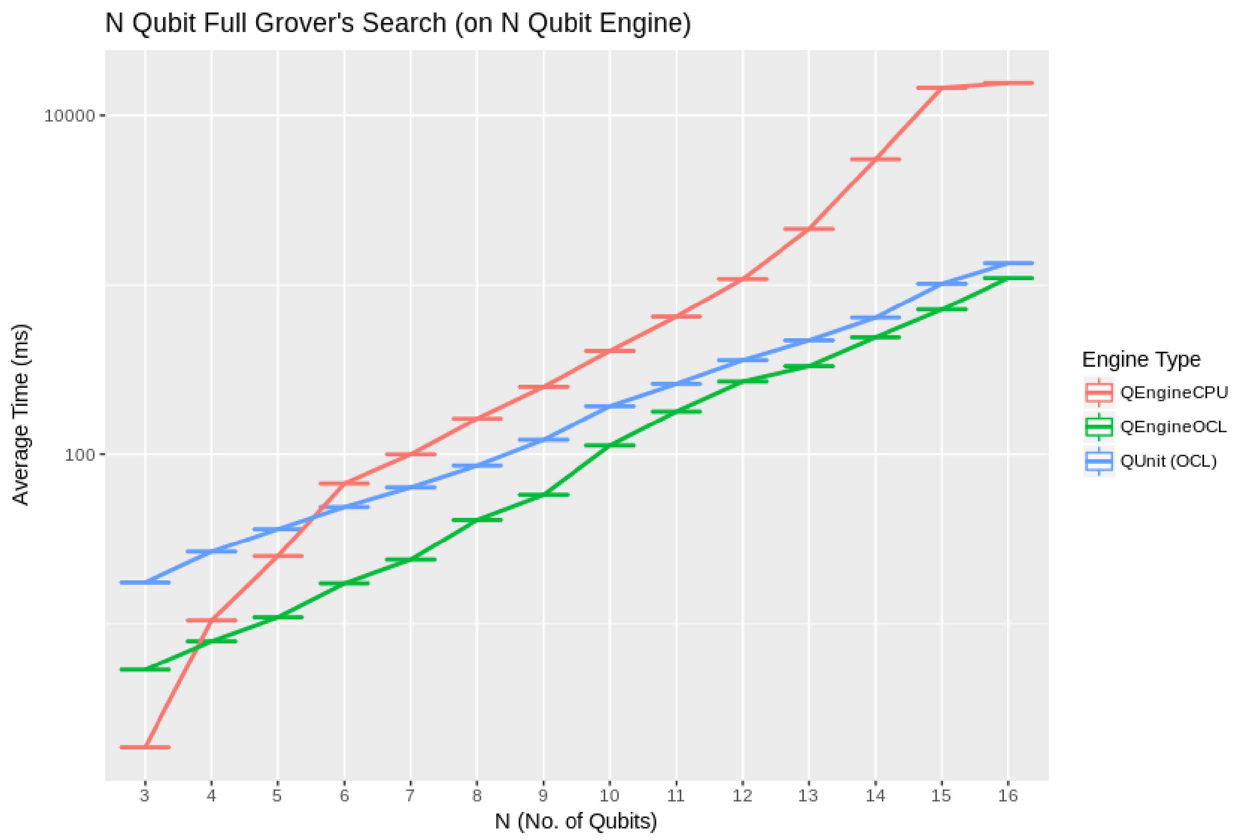




Heap sampling showed high confidence adherence to theoretical expecations. Complex numbers are represented as 2 double (64-bit) or 2 single (32-bit) accuracy floating point types, for real and imaginary components. The use of double or single precision is controlled by a compilation flag. There is one 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 most gates for speed and simplicity where it eases implementation.



Grover's algorithm is a relatively ideal test case, in that it allows a modicum of abstraction in implementation while representing an ostensibly practical and common task for truly quantum computational hardware. For 1 expected correct function inversion result, there is a well-defined highest likelihood search iteration count on half a period of the algorithm for a given number of oracle input permutations to search. This graphs shows average time against qubit count for an optimal half period search:



[Broda2016] discusses how Grover's might be adapted in practicality to actually "search an unstructured database," or search an unstructured lookup table, and Qrack is also capable of applying Grover's search to a lookup table with its IndexedLDA, IndexedADC, and IndexedSBC methods. Benchmarks are not given for this arguably more practical application of the algorithm, because few other quantum computer simulator libraries implement it, yet.

3.6.5 Discussion

Up to a consistent deviation at low qubit counts, speed and RAM usage is well predicted by theoretical complexity considerations of the gates, up to a factor of 2 on heap usage for duplication of the state vector.

We might speculate that, at high qubit counts, the calculations operate almost entirely on heap, while system call and cache hit efficiency consistently alter the trend up until a persistent and detectable "bump" at around roughly 8 qubits for the software implementation, and another "bump" at around 17 qubits for the hardware-accelerated engine, on the P3 test machine. For "software" simulation, this would be roughly consistent with a 4MB cache. For the hardware acceleration, this implies a preferred faster RAM bank of about 2GB.

QEngineOCLMulti, an OpenCL-based multiprocessor engine based on the algorithms developed in Intel's *[QHiPSTER]*, fails to outperform the single processor QEngineOCL. We include it in the current release to help the open source community realize a practical multiprocessor implementation in the context of Qrack. A branch of the multiprocessor engine has already been implemented with dynamic load balancing between processors, but it produced no significant improvement. By deduction, likely, we guess that the explicit reliance on general "host" RAM banks for storing the substate vectors creates a bottleneck, when multiple processors need to communicate with general RAM at the same time on the same bus. To alleviate this, a new QEngineOCLMulti implementation might assume that each processor device has a large personal store of memory, at least 4 or 8 GB, in which case OpenCL could be told explicitly to allocate substate vectors in device memory and not use "host" RAM at all for storing the state vector.

3.6.6 Further Work

Qrack has been successfully run on multiple processors at once, and even on clusters, but not with practical performance for real application; a good next step is to redesign the multiprocessor engine to actually outperform the single device engine. Also, CPU "software" implementation parallelism relies on certain potentially expensive standard library functionality, like lambda expressions, and might still be micro-optimized. The API offers many optimized bitwise parallel operations over contiguous bit strings, but similar methods for discontiguous bit sets should be feasible with bit masks, if there is a reasonable demand for them. Further, there is still opportunity for better constant bitwise parallelism cost coverage and better explicit qubit subsystem separation in QUnit.

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

3.6.7 Conclusion

Per *[Pednault2017]*, explicitly separated subsystems of qubits in QUnit have a significant RAM and speed edge in many cases over the "Schrödinger algorithm" of QEngineCPU and QEngineOCL. One of Qrack's greatest new optimizations to either general algorithm is constant complexity or "free" scaling of bitwise parallelism in entangled subsystems, compared to linear complexity scaling without this optimization. Qrack gives very efficient performance on a single node up to at least about 30 qubits, in 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::QEngineCPU, Qrack::QEngineOCL, Qrack::QEngineOCLMulti, Qrack::QUnit

3.7.1 Creating a QInterface

There are four 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, derived from QEngineCPU, leveraging OpenCL hardware to increase the speed of certain calculations.

QINTERFACE_OPENCL_MULTI
Create a QEngineOCLMUlti, composed from multiple QEngineOCLs, using OpenCL in parallel across 2^N devices, for N an integer >= 0.

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_FIRST = QINTERFACE_CPU

QINTERFACE_OPTIMAL = QINTERFACE_CPU

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 subengine*, Ts... *args*)
Factory method to create specific engine implementations.

3.7.2 Constructors

`Qrack::`*QInterface*`::`**QInterface**(bitLenInt *n*, std::shared_ptr<std::default_random_engine> *rgp* = nullptr, bool *doNorm* = true)

`Qrack::`*QInterface*`::`**QInterface**(bitLenInt *n*, std::shared_ptr<std::default_random_engine> *rgp* = nullptr, bool *doNorm* = true)

`Qrack::`*QInterface*`::`**QInterface**(bitLenInt *n*, std::shared_ptr<std::default_random_engine> *rgp* = nullptr, bool *doNorm* = true)

3.7.3 Members

complex *`Qrack::QEngineCPU::`**stateVec**

3.7.4 Configuration Methods

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

int `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*) = 0
Set to a specific permutation.

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

virtual bitLenInt `Qrack::`*`QInterface`*`::`**`Cohere`**(QInterfacePtr *toCopy*) = 0

Combine another *QInterface* with this one, after the last bit index of this one.

"Cohere" 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 "Cohere." Informally, "Cohere" 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 "Cohere" them.

"Cohere" 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.

std::map<QInterfacePtr, bitLenInt> `Qrack::`*`QInterface`*`::`**`Cohere`**(std::vector<QInterfacePtr> *toCopy*)

virtual void `Qrack::`*`QInterface`*`::`**`Decohere`**(bitLenInt *start*, bitLenInt *length*, QInterfacePtr *dest*) = 0

Minimally decohere a set of contiguous bits from the full coherent unit, into "destination.".

Minimally decohere a set of contigious bits from the full coherent unit. The length of this coherent unit is reduced by the length of bits decohered, 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 "Cohere" is like "just setting another group of qubits down next to the first," then "Decohere" 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.

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:

$prob(|(left)1000>) = prob(|100000>) + prob(|100010>) + prob(|100001>) + prob(|100011>)$.

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 decohere a set of contigious bits from the full coherent unit, throwing these qubits away.

Minimally decohere a set of contigious bits from the full coherent unit, discarding these bits. The length of this coherent unit is reduced by the length of bits decohered. For quantum mechanical accuracy, the bit set removed and the bit set left behind should be quantum mechanically "separable."

Like how "Cohere" is like "just setting another group of qubits down next to the first," then "Dispose" is like "just moving a few qubits away from the rest, and throwing them in the trash." 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.

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:

$prob(|(left)1000 >) = prob(|100000 >) + prob(|100010 >) + prob(|100001 >) + prob(|100011 >).$

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 real1 `Qrack::QInterface::`**Prob** (bitLenInt *qubitIndex*) = 0
Direct measure of bit probability to be in |1> state.

Warning PSEUDO-QUANTUM

virtual real1 `Qrack::QInterface::`**ProbAll** (bitCapInt *fullRegister*) = 0
Direct measure of full register probability to be in permutation state.

Warning PSEUDO-QUANTUM

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

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

virtual void `Qrack::QInterface::`**Reverse** (bitLenInt *first*, bitLenInt *last*)
Reverse all of the bits in a sequence.

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 Register Gates

void `Qrack::QInterface::`**AND** (bitLenInt *inputBit1*, bitLenInt *inputBit2*, bitLenInt *outputBit*)
Quantum analog of classical "AND" gate.

Measures the outputBit, then overwrites it with result.

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. Measures the outputBit, then overwrites it with result.

void `Qrack::QInterface::`**`OR`**`(`bitLenInt *inputBit1*, bitLenInt *inputBit2*, bitLenInt *outputBit*`)`
Quantum analog of classical "OR" gate.

Measures the outputBit, then overwrites it with result.

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. Measures the outputBit, then overwrites it with result.

void `Qrack::QInterface::`**`XOR`**`(`bitLenInt *inputBit1*, bitLenInt *inputBit2*, bitLenInt *outputBit*`)`
Quantum analog of classical "XOR" gate.

Measures the outputBit, then overwrites it with result.

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. Measures the outputBit, then overwrites it with result.

void `Qrack::QInterface::`**`H`**`(`bitLenInt *qubitIndex*`)`
Hadamard gate.

Applies a Hadamard gate on qubit at "qubitIndex."

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.)

void `Qrack::QInterface::`**`X`**`(`bitLenInt *qubitIndex*`)`
X gate.

NOT gate, which is also Pauli x matrix.

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

void `Qrack::QInterface::`**`Y`**(bitLenInt *qubitIndex*)
Y gate.

Apply Pauli Y matrix to bit.

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

void `Qrack::QInterface::`**`Z`**(bitLenInt *qubitIndex*)
Z gate.

Apply Pauli Z matrix to bit.

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::`**`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."

void `Qrack::QInterface::`**`RT`**(real1 *radians*, bitLenInt *qubitIndex*)
Phase shift gate.

"Phase shift gate" - Rotates as e^(-i*/2) around |1> state

Rotates as $e^{-i*\theta/2}$ around |1> state

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 $e^{i*\pi*numerator/2^denomPower}$ around |1> state.

NOTE THAT * DYADIC OPERATION ANGLE SIGN IS REVERSED FROM RADIAN ROTATION OPERATORS AND LACKS DIVISION BY A FACTOR OF TWO.

void `Qrack::QInterface::`**`CRT`**(real1 *radians*, bitLenInt *control*, bitLenInt *target*)
Controlled "phase shift gate".

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

If control bit is set to 1, rotates target bit as $e^{-i*\theta/2}$ around |1> state.

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 $e^{i*\pi*numerator/2^denomPower}$ around |1> state.

void `Qrack::`*`QInterface`*`::`**`RX`**(real1 *radians*, bitLenInt *qubitIndex*)

X axis rotation gate.

x axis rotation gate - Rotates as e^(-i*/2) around Pauli x axis

Rotates as $e^{-i*\theta/2}$ around Pauli X axis

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

Dyadic fraction X axis rotation gate.

Dyadic fraction x axis rotation gate - Rotates as e^(i*(M_PI * numerator) / 2^denomPower) around Pauli x axis.

Rotates $e^{i*\pi*numerator/2^denomPower}$ on Pauli x axis.

NOTE THAT DYADIC OPERATION ANGLE SIGN IS REVERSED FROM RADIAN ROTATION OPERATORS AND LACKS DIVISION BY A FACTOR OF TWO.

void `Qrack::`*`QInterface`*`::`**`CRX`**(real1 *radians*, bitLenInt *control*, bitLenInt *target*)

Controlled X axis rotation gate.

Controlled x axis rotation - if control bit is true, rotates as e^(-i*/2) around Pauli x axis.

If "control" is 1, rotates as $e^{-i*\theta/2}$ on Pauli x axis.

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 as e^(i*(M_PI * numerator) / 2^denomPower) around Pauli x axis.

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

NOTE THAT DYADIC OPERATION ANGLE SIGN IS REVERSED FROM RADIAN ROTATION OPERATORS.

void `Qrack::`*`QInterface`*`::`**`RY`**(real1 *radians*, bitLenInt *qubitIndex*)

Y axis rotation gate.

y axis rotation gate - Rotates as e^(-i*/2) around Pauli y axis

Rotates as $e^{-i*\theta/2}$ around Pauli y axis.

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

Dyadic fraction Y axis rotation gate.

Dyadic fraction y axis rotation gate - Rotates as e^(i*(M_PI * numerator) / 2^denomPower) around Pauli y axis.

Rotates as $e^{i*\pi*numerator/2^denomPower}$ around Pauli Y axis.

NOTE THAT DYADIC OPERATION ANGLE SIGN IS REVERSED FROM RADIAN ROTATION OPERATORS AND LACKS DIVISION BY A FACTOR OF TWO.

void `Qrack::`*`QInterface`*`::`**`CRY`**(real1 *radians*, bitLenInt *control*, bitLenInt *target*)

Controlled Y axis rotation gate.

Controlled y axis rotation - if control bit is true, rotates as e^(-i*) around Pauli y axis.

If "control" is set to 1, rotates as $e^{-i*\theta/2}$ around Pauli Y axis.

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 as e^(i*(M_PI * numerator) / 2^denomPower) around Pauli y axis.

If “control” is set to 1, rotates as $e^{i*\pi*numerator/2^denomPower}$ around Pauli Y axis.

NOTE THAT DYADIC OPERATION ANGLE SIGN IS REVERSED FROM RADIAN ROTATION OPERATORS.

void `Qrack::QInterface::`**`RZ`**(real1 *radians*, bitLenInt *qubitIndex*)

Z axis rotation gate.

z axis rotation gate - Rotates as e^(-i*/2) around Pauli z axis

Rotates as $e^{-i*\theta/2}$ around Pauli Z axis.

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

Dyadic fraction Z axis rotation gate.

Dyadic fraction y axis rotation gate - Rotates as e^(i*(M_PI * numerator) / 2^denomPower) around Pauli y axis.

Rotates as $e^{i*\pi*numerator/2^denomPower}$ around Pauli Z axis.

NOTE THAT DYADIC OPERATION ANGLE SIGN IS REVERSED FROM RADIAN ROTATION OPERATORS AND LACKS DIVISION BY A FACTOR OF TWO.

void `Qrack::QInterface::`**`CRZ`**(real1 *radians*, bitLenInt *control*, bitLenInt *target*)

Controlled Z axis rotation gate.

Controlled z axis rotation - if control bit is true, rotates as e^(-i*) around Pauli z axis.

If “control” is set to 1, rotates as $e^{-i*\theta/2}$ around Pauli Zaxis.

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 as e^(i*(M_PI * numerator) / 2^denomPower) around Pauli z axis.

If “control” is set to 1, rotates as $e^{i*\pi*numerator/2^denomPower}$ around Pauli Z axis.

NOTE THAT DYADIC OPERATION ANGLE SIGN IS REVERSED FROM RADIAN ROTATION OPERATORS.

void `Qrack::QInterface::`**`Exp`**(real1 *radians*, bitLenInt *qubitIndex*)

(Identity) Exponentiation gate

Exponentiate identity operator.

Applies $e^{-i*\theta*I}$, exponentiation of the identity operator

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

Dyadic fraction (identity) exponentiation gate.

Dyadic fraction (identity) exponentiation gate - Applies $e^{-i*\pi*numerator*I/2^denomPower}$, exponentiation of the identity operator.

Applies $e^{-i*\pi*numerator*I/2^denomPower}$, exponentiation of the identity operator

NOTE THAT DYADIC OPERATION ANGLE SIGN IS REVERSED FROM RADIAN ROTATION OPERATORS AND LACKS DIVISION BY A FACTOR OF TWO.

void `Qrack::QInterface::`**`ExpX`**(real1 *radians*, bitLenInt *qubitIndex*)

Pauli X exponentiation gate.

Exponentiate Pauli X operator.

Applies $e^{-i*\theta*\sigma_x}$, exponentiation of the Pauli X operator void `Qrack::QInterface::`**`ExpXDyad`**(int *numerator*, int *denomPower*, bitLenInt *qubitIndex*)

Dyadic fraction Pauli X exponentiation gate.

Dyadic fraction Pauli X exponentiation gate - Applies $e^{-i*\pi*numerator*\sigma_x/2^denomPower}$, exponentiation of the Pauli X operator.

Applies $e^{-i*\pi*numerator*\sigma_x/2^denomPower}$, exponentiation of the Pauli X operator

NOTE THAT DYADIC OPERATION ANGLE SIGN IS REVERSED FROM RADIAN ROTATION OPERATORS AND LACKS DIVISION BY A FACTOR OF TWO.

void `Qrack::QInterface::`**`ExpY`**(real1 *radians*, bitLenInt *qubitIndex*)

Pauli Y exponentiation gate.

Exponentiate Pauli Y operator.

Applies $e^{-i*\theta*\sigma_y}$, exponentiation of the Pauli Y operator

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

Dyadic fraction Pauli Y exponentiation gate.

Dyadic fraction Pauli Y exponentiation gate - Applies $e^{-i*\pi*numerator*\sigma_y/2^denomPower}$, exponentiation of the Pauli Y operator.

Applies $e^{-i*\pi*numerator*\sigma_y/2^denomPower}$, exponentiation of the Pauli Y operator

NOTE THAT DYADIC OPERATION ANGLE SIGN IS REVERSED FROM RADIAN ROTATION OPERATORS AND LACKS DIVISION BY A FACTOR OF TWO.

void `Qrack::QInterface::`**`ExpZ`**(real1 *radians*, bitLenInt *qubitIndex*)

Pauli Z exponentiation gate.

Exponentiate Pauli Z operator.

Applies $e^{-i*\theta*\sigma_z}$, exponentiation of the Pauli Z operator

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

Dyadic fraction Pauli Z exponentiation gate.

Dyadic fraction Pauli Z exponentiation gate - Applies $e^{-i*\pi*numerator*\sigma_z/2^denomPower}$, exponentiation of the Pauli Z operator.

Applies $e^{-i*\pi*numerator*\sigma_z/2^denomPower}$, exponentiation of the Pauli Z operator

NOTE THAT DYADIC OPERATION ANGLE SIGN IS REVERSED FROM RADIAN ROTATION OPERATORS AND LACKS DIVISION BY A FACTOR OF TWO.

Register-wide Gates

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

Bitwise “AND”.

“AND” compare two bits in *QInterface*, and store result in outputBit

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

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

Classical bitwise “AND”.

“AND” compare a bit range in *QInterface* with a classical unsigned integer, and store result in range starting at output

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

void `Qrack::`*`QInterface`*`::`**`OR`**(bitLenInt *inputStart1*, bitLenInt *inputStart2*, bitLenInt *outputStart*, bitLenInt *length*)
Bitwise "OR".

"OR" compare two bits in *QInterface*, and store result in outputBit

void `Qrack::`*`QInterface`*`::`**`CLOR`**(bitLenInt *qInputStart*, bitCapInt *classicalInput*, bitLenInt *outputStart*, bitLenInt *length*)
Classical bitwise "OR".

"OR" compare a bit range in *QInterface* with a classical unsigned integer, and store result in range starting at output

void `Qrack::`*`QInterface`*`::`**`XOR`**(bitLenInt *inputStart1*, bitLenInt *inputStart2*, bitLenInt *outputStart*, bitLenInt *length*)
Bitwise "XOR".

"XOR" compare two bits in *QInterface*, and store result in outputBit

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

"XOR" compare a bit range in *QInterface* with a classical unsigned integer, and store result in range starting at output

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

void `Qrack::`*`QInterface`*`::`**`AntiCCNOT`**(bitLenInt *control1*, bitLenInt *control2*, bitLenInt *target*, bitLenInt *length*)
Bitwise doubly "anti-"controlled-not.

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

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 *inputBits*, bitLenInt *targetBits*, bitLenInt *length*)
Bitwise "anti-"controlled-not.

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

Apply Hadamard gate to each bit in "length," starting from bit index "start".

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

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

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

Apply Pauli Y matrix to each bit.

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

Apply Pauli Z matrix to each bit.

void `Qrack::QInterface::`**`CY`**(bitLenInt *control*, bitLenInt *target*, bitLenInt *length*)
Bitwise controlled Y gate.

Apply controlled Pauli Y matrix to each 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*, bitLenInt *length*)
Bitwise controlled Z gate.

Apply controlled Pauli Z matrix to each bit.

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

void `Qrack::QInterface::`**`RT`**(real1 *radians*, bitLenInt *start*, bitLenInt *length*)
Bitwise phase shift gate.

“Phase shift gate” - Rotates each bit as e^(-i*/2) around |1> state

Rotates as $e^{-i*\theta/2}$ around |1> state

void `Qrack::QInterface::`**`RTDyad`**(int *numerator*, int *denomPower*, bitLenInt *start*, bitLenInt *length*)
Bitwise dyadic fraction phase shift gate.

Dyadic fraction “phase shift gate” - Rotates each bit as e^(i*(M_PI * numerator) / denominator) around |1> state.

Rotates as $e^{i*\pi*numerator/2^denomPower}$ around |1> state.

NOTE THAT DYADIC OPERATION ANGLE SIGN IS REVERSED FROM RADIAN ROTATION OPERATORS AND LACKS DIVISION BY A FACTOR OF TWO.

void `Qrack::QInterface::`**`RX`**(real1 *radians*, bitLenInt *start*, bitLenInt *length*)
Bitwise X axis rotation gate.

x axis rotation gate - Rotates each bit as e^(-i*/2) around Pauli x axis

Rotates as $e^{-i*\theta/2}$ around Pauli X axis

void `Qrack::QInterface::`**`RXDyad`**(int *numerator*, int *denomPower*, bitLenInt *start*, bitLenInt *length*)
Bitwise dyadic fraction X axis rotation gate.

Dyadic fraction x axis rotation gate - Rotates each bit as e^(i*(M_PI * numerator) / denominator) around Pauli x axis.

Rotates $e^{i*\pi*numerator/2^denomPower}$ on Pauli x axis.

NOTE THAT DYADIC OPERATION ANGLE SIGN IS REVERSED FROM RADIAN ROTATION OPERATORS AND LACKS DIVISION BY A FACTOR OF TWO.

void `Qrack::QInterface::`**`CRX`**(real1 *radians*, bitLenInt *control*, bitLenInt *target*, bitLenInt *length*)
Bitwise controlled X axis rotation gate.

Controlled x axis rotation.

If “control” is 1, rotates as $e^{-i*\theta/2}$ on Pauli x axis.

void `Qrack::QInterface::`**`CRXDyad`**(int *numerator*, int *denomPower*, bitLenInt *control*, bitLenInt *target*, bitLenInt *length*)
Bitwise controlled dyadic fraction X axis rotation gate.

Controlled dyadic fraction x axis rotation gate - for each bit, if control bit is true, rotates target bit as as e^(i*(M_PI * numerator) / denominator) around Pauli x axis.

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

void `Qrack::QInterface::`**`RY`**(real1 *radians*, bitLenInt *start*, bitLenInt *length*)

Bitwise Y axis rotation gate.

y axis rotation gate - Rotates each bit as e^(-i*/2) around Pauli y axis

Rotates as $e^{-i*\theta/2}$ around Pauli y axis.

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

Bitwise dyadic fraction Y axis rotation gate.

Dyadic fraction y axis rotation gate - Rotates each bit as e^(i*(M_PI * numerator) / denominator) around Pauli y axis.

Rotates as $e^{i*\pi*numerator/2^denomPower}$ around Pauli Y axis.

NOTE THAT DYADIC OPERATION ANGLE SIGN IS REVERSED FROM RADIAN ROTATION OPERATORS AND LACKS DIVISION BY A FACTOR OF TWO.

void `Qrack::QInterface::`**`CRY`**(real1 *radians*, bitLenInt *control*, bitLenInt *target*, bitLenInt *length*)

Bitwise controlled Y axis rotation gate.

Controlled y axis rotation.

If "control" is set to 1, rotates as $e^{-i*\theta/2}$ around Pauli Y axis.

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

Bitwise controlled dyadic fraction y axis rotation gate.

Controlled dyadic fraction y axis rotation gate - for each bit, if control bit is true, rotates target bit as e^(i*(M_PI * numerator) / denominator) around Pauli y axis.

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

void `Qrack::QInterface::`**`RZ`**(real1 *radians*, bitLenInt *start*, bitLenInt *length*)

Bitwise Z axis rotation gate.

z axis rotation gate - Rotates each bit as e^(-i*/2) around Pauli z axis

Rotates as $e^{-i*\theta/2}$ around Pauli Z axis.

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

Bitwise dyadic fraction Z axis rotation gate.

Dyadic fraction z axis rotation gate - Rotates each bit as e^(i*(M_PI * numerator) / denominator) around Pauli y axis.

Rotates as $e^{i*\pi*numerator/2^denomPower}$ around Pauli Z axis.

NOTE THAT DYADIC OPERATION ANGLE SIGN IS REVERSED FROM RADIAN ROTATION OPERATORS AND LACKS DIVISION BY A FACTOR OF TWO.

void `Qrack::QInterface::`**`CRZ`**(real1 *radians*, bitLenInt *control*, bitLenInt *target*, bitLenInt *length*)

Bitwise controlled Z axis rotation gate.

Controlled z axis rotation.

If "control" is set to 1, rotates as $e^{-i*\theta/2}$ around Pauli Zaxis.

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

Bitwise controlled dyadic fraction Z axis rotation gate.

Controlled dyadic fraction z axis rotation gate - for each bit, if control bit is true, rotates target bit as e^(i*(M_PI * numerator) / denominator) around Pauli z axis.

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

void `Qrack::QInterface::`**`Exp`**(real1 *radians*, bitLenInt *start*, bitLenInt *length*)

Bitwise (identity) exponentiation gate.

Bitwise (identity) exponentiation gate - Applies $e^{-i*\theta*I}$, exponentiation of the identity operator.

Applies $e^{-i*\theta*I}$, exponentiation of the identity operator

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

Bitwise Dyadic fraction (identity) exponentiation gate.

Dyadic fraction (identity) exponentiation gate - Applies $e^{-i*\pi*numerator*I/2^denomPower}$, exponentiation of the identity operator.

Applies $e^{-i*\pi*numerator*I/2^denomPower}$, exponentiation of the identity operator

NOTE THAT DYADIC OPERATION ANGLE SIGN IS REVERSED FROM RADIAN ROTATION OPERATORS AND LACKS DIVISION BY A FACTOR OF TWO.

void `Qrack::QInterface::`**`ExpX`**(real1 *radians*, bitLenInt *start*, bitLenInt *length*)

Bitwise Pauli X exponentiation gate.

Bitwise Pauli X exponentiation gate - Applies $e^{-i*\theta*\sigma_x}$, exponentiation of the Pauli X operator.

Applies $e^{-i*\theta*\sigma_x}$, exponentiation of the Pauli X operator

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

Bitwise Dyadic fraction Pauli X exponentiation gate.

Dyadic fraction Pauli X exponentiation gate - Applies $e^{-i*\pi*numerator*\sigma_x/2^denomPower}$, exponentiation of the Pauli X operator.

Applies $e^{-i*\pi*numerator*\sigma_x/2^denomPower}$, exponentiation of the Pauli X operator

NOTE THAT DYADIC OPERATION ANGLE SIGN IS REVERSED FROM RADIAN ROTATION OPERATORS AND LACKS DIVISION BY A FACTOR OF TWO.

void `Qrack::QInterface::`**`ExpY`**(real1 *radians*, bitLenInt *start*, bitLenInt *length*)

Bitwise Pauli Y exponentiation gate.

Bitwise Pauli Y exponentiation gate - Applies $e^{-i*\theta*\sigma_y}$, exponentiation of the Pauli Y operator.

Applies $e^{-i*\theta*\sigma_y}$, exponentiation of the Pauli Y operator

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

Bitwise Dyadic fraction Pauli Y exponentiation gate.

Dyadic fraction Pauli Y exponentiation gate - Applies $e^{-i*\pi*numerator*\sigma_y/2^denomPower}$, exponentiation of the Pauli Y operator.

Applies $e^{-i*\pi*numerator*\sigma_y/2^denomPower}$, exponentiation of the Pauli Y operator

NOTE THAT DYADIC OPERATION ANGLE SIGN IS REVERSED FROM RADIAN ROTATION OPERATORS AND LACKS DIVISION BY A FACTOR OF TWO.

void `Qrack::QInterface::`**`ExpZ`**(real1 *radians*, bitLenInt *start*, bitLenInt *length*)

Bitwise Pauli Z exponentiation gate.

Bitwise Pauli Z exponentiation gate - Applies $e^{-i*\theta*\sigma_z}$, exponentiation of the Pauli Z operator.

Applies $e^{-i*\theta*\sigma_z}$, exponentiation of the Pauli Z operator

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

Bitwise Dyadic fraction Pauli Z exponentiation gate.

Dyadic fraction Pauli Z exponentiation gate - Applies $e^{-i*\pi*numerator*\sigma_z/2^denomPower}$, exponentiation of the Pauli Z operator.

Applies $e^{-i*\pi*numerator*\sigma_z/2^denomPower}$, exponentiation of the Pauli Z operator

NOTE THAT DYADIC OPERATION ANGLE SIGN IS REVERSED FROM RADIAN ROTATION OPERATORS AND LACKS DIVISION BY A FACTOR OF TWO.

3.7.7 Algorithmic Implementations

void `Qrack::QInterface::`**`QFT`**(bitLenInt *start*, bitLenInt *length*)

Quantum Fourier Transform - Apply the quantum Fourier transform to the register.

`virtual` bitCapInt `Qrack::QInterface::`**`IndexedLDA`**(bitLenInt *indexStart*, bitLenInt *indexLength*, bitLenInt *valueStart*, bitLenInt *valueLength*, unsigned char **values*) = 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()*.

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 QEngineOCL

Defined in qengine_opencl.hpp.

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

Warning: doxygenfunction: Unable to resolve multiple matches for function "Qrack::QEngineOCL::QEngineOCL" with arguments (bitLenInt, bitCapInt, std::shared_ptr<std::default_random_engine>, int, bool) in doxygen xml output for project "qrack" from directory: /tmp/qrack/doc/xml. Potential matches:

```
- Qrack::QEngineOCL::QEngineOCL(QEngineOCLPtr)
- Qrack::QEngineOCL::QEngineOCL(bitLenInt, bitCapInt, std::shared_ptr<std::default_
↪random_engine>, int, bool, complex)
```

3.10 QEngineOCLMulti

Defined in qengine_opencl_multi.hpp.

The API is provided by Qrack::QInterface. However, QEngineOCLMulti has two custom constructors:

`Qrack::QEngineOCLMulti::`**`QEngineOCLMulti`**(bitLenInt *qBitCount*, bitCapInt *initState*, std::shared_ptr<std::default_random_engine> *rgp* = nullptr, int *deviceCount* = -1)

Initialize a Qrack::QEngineOCLMulti object.

Specify the number of qubits and an initial permutation state. Additionally, optionally specify a pointer to a random generator engine object and a number of sub-engines, (usually one per device, though this can be over-allocated,) to break the object into. The "deviceCount" should be a power of 2, but it will be floored to a power of two if the parameter is not already a power of two. The QEngineOCL can not use more than 1 power of 2 devices per qubit. (2^N devices for N qubits.) Powers of 2 in excess of the qubit count will only be used if this engine acquires additional qubits.

`Qrack::QEngineOCLMulti::`**`QEngineOCLMulti`**(bitLenInt *qBitCount*, bitCapInt *initState*, std::vector<int> *devIDs*, std::shared_ptr<std::default_random_engine> *rgp* = nullptr)

Initialize a Qrack::QEngineOCLMulti object.

Specify the number of qubits and an initial permutation state. Additionally, optionally specify a list of device IDs for sub-engines and a pointer to a random generator engine object.

"devIDs" is a list of integers that represent the index of OpenCL devices in the *OCLEngine* singleton, to select how equal sized sub-engines are distributed between devices in this engine. The QEngineOCLMulti will only have a power of 2 count of subengines at a time, and not more than 1 power of 2 devices per qubit. (2^N devices for N qubits.) Devices in excess of the highest power of two in the list count will essentially be ignored. Powers of 2 in excess of the qubit count will only be used if this engine acquires additional qubits. It might be possible to load balance this way, for example, by allocating 3 sub-engines on one device index and one sub-engine on a second device index. (Whether this is an efficient load balancing mechanism will depend on the particulars of the system architecture and instance initialization.)

3.11 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
SEN	0x0F	Implied	SEt the Negative flag
PXA	0x12	Implied	Apply a bitwise Pauli X on the Accumulator
PXA	0x13	Implied	Apply a bitwise Pauli X on the X Register
HAC	0x17	Implied	Apply a Hadamard gate on the carry flag
PYA	0x1A	Implied	Apply a bitwise Pauli Y on the Accumulator

Continued on next page

Table 1 – continued from previous page

OP	Byte	Mode	Description
PYA	0x1B	Implied	Apply a bitwise Pauli Y on the X 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
PZA	0x33	Implied	Apply a bitwise Pauli Z on the X 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
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
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
RZX	0x5B	Implied	Bitwise quarter rotation on Z axis for the X Register
FTA	0x62	Implied	Quantum Fourier Transform on Accumulator
FTX	0x63	Implied	Quantum Fourier Transform on the X 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.
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 also collapse superposition of the Accumulator
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.

Continued on next page

Table 1 – continued from previous page

OP	Byte	Mode	Description
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
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.

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.
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

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

[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

Index