qrack Documentation

qrack

Mar 12, 2022

Contents

CHAPTER 1

Introduction

An introductory talk to QRack can be found here Intro to Qrack: a framework for fast quantum simulation by Daniel Strano | Quantum Software Talks.

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

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

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

Qrack with tools designed to control, extend and visualize data emanating to and from Quantum circuits utilising Docker can be found here ThereminQ. QRack integerated with Unity game creation framework and quantum physics engine can be found here OpenRelativity.

CHAPTER 2

Copyright

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

2.1 Getting Started

2.1.1 Accelerate Qiskit workloads in 3 steps! (PyQrack)

We encourage all users to build the Qrack framework from source as primary installation method, for maximum configurability, system compatibility, and application suitability. However, you don't have to!

Many users can benefit from an installation method designed to replace a Qiskit simulator with a Qrack simulator one-for-one, without little or no Qiskit user code modification. We provide such an installation method, typically in 3 simple steps. We assume that you've already installed a Qiskit environment and have a script that you'd like to try running with Qrack instead of Aer, with little or no modification.

1. Install the qiskit-qrack-provider Python package, which will also install PyQrack as a dependency:

```
pip install qiskit-qrack-provider`
```

2. In the Qiskit script you'd like to run, replace all instances of "Aer" and "AerProvider" with "Qrack" and "QrackProvider" from a subset of classes among the following import statements:

```
from qiskit.providers.qrack import Qrack, QrackProvider
from qiskit.providers.qrack.backends import QasmSimulator
```

3. If you must, adapt your script to use the "qasm_simulator" backend API. (You might not need to modify anything at all, for this step!) Qrack currently only provides a "qasm_simulator" backend API, because Qrack's novel simulation method optimizations don't necessarily internally track a traditional state vector representation at

all, except when expedient. Hence, Qrack benefits from an API formulated in terms of gate primitives and measurement distributions that closely resemble a native quantum hardware interface.

That's it! Enjoy GPU-acceleration and Qrack's "transparent" novel optimization layers! (For example, try writing Clifford group stabilizer programs to run with the default simulator backend configuration, and watch Qrack automatically detect that the program should execute as stabilizer simulation!)

The rest of the document explains the "advanced" (but honestly also not difficult) C++ build requirements and process.

2.1.2 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.

2.1.3 Checking Out

Clone the repository with git:

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

2.1.4 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 `-DENABLE_OPENCL=OFF` build option to `cmake` when building qrack the first time.

Then `make all` to compile (with `-j8` for 8 parallel build cores, or as appropriate) or (`sudo`) `make install` to compile and install into a shared location in your system. `make install` would be needed if you're going to use this version of the qrack library with pyqrack.

Qrack compiles with either double (`FPPOW=6`) or single (`FPPOW=5`) accuracy complex numbers, for most OpenCL-enabled builds. 16 bit half accurracy (`FPPOW=4`) is available for CPU-only builds and for a limited selection of NVIDIA GPUs. Single float accuracy is used by default. Single float accuracy uses almost exactly half as much RAM as double accuracy, allowing one additional qubit. Single accuracy may also be faster or the only compatible option for certain OpenCL devices.

Note: If your GPU or accelerator supports 16-bit floating point, but 16-bit OpenCL kernels do not compile or run, then a Qrack build for your device likely needs simply a different `#pragma` line, which is a tiny change to the OpenCL code. We suggest you open an issue on the Qrack GitHub repository to direct the developers' attention to the likely very small change that will support your device, and permanently fix the issue for other users. However, it's difficult to anticipate these cases, particularly without your help to actually test the modified build for a device to which the developers don't have access.

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

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

Many OpenCL devices that don't support double accuracy floating point operations still support 64-bit integer types. If a device doesn't support 64-bit integer types, UINTPOW=5 with ENABLE_COMPLEX_X2=OFF will disable all 64-bit types in OpenCL kernels, as well as SIMD. This theoretically supports the OpenCL standard on a device such as a Raspberry Pi 3.

2.1.5 Running Tests

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

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

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

The benchmarks respect the same parameters:

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

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

2.1.6 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. The easiest way to choose an optimal "layer stack" is to use QINTERFACE_OPTIMAL for a single OpenCL device simulator, and use QINTERFACE_OPTIMAL_MULTI for a multi-device simulator:

```
QInterfacePtr qftReg = CreateQuantumInterface(QINTERFACE_OPTIMAL, qubitCount, intPerm,
↪ rng);
QInterfacePtr qftReg2 = CreateQuantumInterface(QINTERFACE_OPTIMAL_MULTI, 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;
```

(continues on next page)

(continued from previous page)

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

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

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

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

2.1.7 Optimal CreateQuantumInterface Factory Options

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

```
QInterfacePtr qftReg = CreateQuantumInterface(QINTERFACE_QUNIT, QINTERFACE_STABILIZER_HYBRID, QINTERFACE_QPAGER, QINTERFACE_MASK_FUSION, QINTERFACE_HYBRID, qubitCount, intPerm, rng[, ...]);
```

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

2.1.8 Embedding Qrack

For static linkage, the build process 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.

For dynamic linkage, use `libqrack_pinvoke.so`. (This is the shared object library upon which such wrapper projects as PyQrack are based and linked from.)

2.1.9 Performance

See the extensive *performance analysis and graphs* section.

2.1.10 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.

2.1.11 Community

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

2.2 Theory

2.2.1 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

2.2.2 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:

$$|\psi\rangle = x_0|000\rangle + x_1|001\rangle + x_2|010\rangle + x_3|011\rangle + x_4|100\rangle + x_5|101\rangle + x_6|110\rangle + x_7|111\rangle \tag{2.1}$$

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,

$$\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \tag{2.2}$$

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 (2.1)?

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

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:

$$(X \otimes I \otimes I) \times M \tag{2.3}$$

$$\left(\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \otimes \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \otimes \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \right) \times \begin{bmatrix} x_{000} \\ x_{001} \\ x_{010} \\ x_{011} \\ x_{100} \\ x_{101} \\ x_{110} \\ x_{111} \end{bmatrix} \tag{2.4}$$

$$\begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix} \times \begin{bmatrix} x_{000} \\ x_{001} \\ x_{010} \\ x_{011} \\ x_{100} \\ x_{101} \\ x_{110} \\ x_{111} \end{bmatrix} \tag{2.5}$$

$$(X \otimes I \otimes I) \times \begin{bmatrix} x_{000} \\ x_{001} \\ x_{010} \\ x_{011} \\ x_{100} \\ x_{101} \\ x_{110} \\ x_{111} \end{bmatrix} = \begin{bmatrix} x_{001} \\ x_{000} \\ x_{011} \\ x_{010} \\ x_{101} \\ x_{100} \\ x_{111} \\ x_{110} \end{bmatrix} \tag{2.6}$$

The operation in (2.3) 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 (2.5) 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 (2.5)). 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:

$$\begin{bmatrix} \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \times \begin{bmatrix} x_{000} \\ x_{001} \end{bmatrix} \\ \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \times \begin{bmatrix} x_{010} \\ x_{011} \end{bmatrix} \\ \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \times \begin{bmatrix} x_{100} \\ x_{101} \end{bmatrix} \\ \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \times \begin{bmatrix} x_{110} \\ x_{111} \end{bmatrix} \end{bmatrix} = \begin{bmatrix} \begin{bmatrix} x_{001} \\ x_{000} \end{bmatrix} \\ \begin{bmatrix} x_{011} \\ x_{010} \end{bmatrix} \\ \begin{bmatrix} x_{101} \\ x_{100} \end{bmatrix} \\ \begin{bmatrix} x_{111} \\ x_{110} \end{bmatrix} \end{bmatrix} \tag{2.7}$$

And, fully decomposing (2.7):

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

It's worth pointing out that the operation detailed in (2.7) 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);
```

(continues on next page)

(continued from previous page)

```
// 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.

2.2.3 6502 Reference Documents

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

2.3 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.

2.3.1 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.

2.3.2 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/

2.4 Examples

2.4.1 Qrack Examples

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

grovers - Runs a canonical Grover's search

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

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

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

qneuron_classification - Runs a simple quantum neuron classification model task

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

quantum_perceptron - Demonstrates the single neuron “atom” of Qrack’s quantum neural net examples

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

teleport - Demonstrates quantum teleportation

2.4.2 VM6502Q Examples

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

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

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

Then, make the various examples:

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

```
hello world
^C
Interrupted at e002

Emulation performance stats is OFF.

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

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

I/O status: enabled,  at: $e000,  local echo: OFF.
Graphics status: disabled,  at: $e002
ROM: disabled. Range: $d000 - $dfff.
Op-code execute history: disabled.
-----------------------------------+----------------------------------------
   C - continue,  S - step         |    A - set address for next step
   G - go/cont. from new address   |    N - go number of steps, P - IRQ
   I - toggle char I/O emulation   |    X - execute from new address
   T - show I/O console            |    B - blank (clear) screen
   E - toggle I/O local echo       |    F - toggle registers animation
   J - set animation delay         |    M - dump memory, W - write memory
   K - toggle ROM emulation        |    R - show registers, Y - snapshot
   L - load memory image           |    O - display op-code exec. history
```

(continues on next page)

(continued from previous page)

```
   D - disassemble code in memory   |    Q - quit, 0 - reset, H - help
   V - toggle graphics emulation    |    U - enable/disable exec. history
   Z - enable/disable debug traces  |    1 - enable/disable perf. stats
   2 - display debug traces         |    ? - show this menu
------------------------------------+-----------------------------------------
> q
Thank you for using VM65.
```

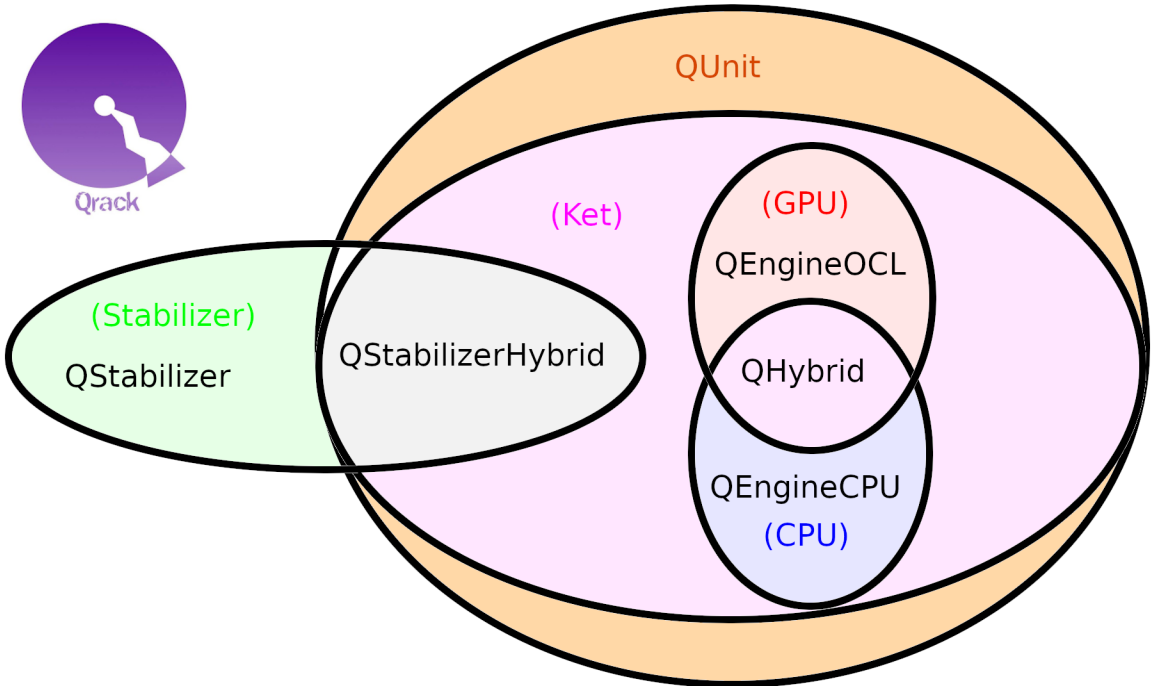
Use *Ctrl-C* to bring up the in-VM menu, and *q* to exit.

Creating a new example

- Copy the `prototype/` directory to your example name, renaming the `.cfg` file to match the source file.
- Change `prototype` in `Makefile` to be the basename of your cfg and source file.
- Adjust the `project.cfg` file as necessary for memory sizing.

2.5 Implementation

2.5.1 Organizational Chart



In overview, Qrack freely mixes simulator representations, between "Schrödinger method"/"ket," stabilizer tableau, and Schmidt decomposed representations of both ket and stabilizer. *Qrack::QStabilizerHybrid* will attempt to maintain stabilizer representation for as deep as possible into a circuit, then switch over internally to ket representation when it encounters an operation it cannot carry out with Clifford and Pauli gates. For ket simulation, we have both CPU and GPU based implementations, which will be automatically switched between as opportune by *QHybrid*. *QUnit* creates a collection of partially Schmidt decomposed peer simulator types, like *QStabilizerHybrid* representation over

QHybrid (or *QPager*, for Intel-QS-like "paged" simulation across different maximum allocation segments on one or more GPU devices).

2.5.2 QEngine

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

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

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

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

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

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

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

2.5.3 QUnit

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

2.5.4 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.

2.6 Qrack Performance

2.6.1 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, QCGPU, and Qiskit Aer GPU. Qrack's experimentally derived results compare favorably against theoretical boundaries, and out-perform naive implementations for many scenarios.

2.6.2 Introduction

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

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

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

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

Reader Guidance

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

Disclaimers

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

2.6.3 Method

This performance document is meant to be a simple, to-the-point, and preliminary digest of these results. These results were prepared with the generous financial support of the Unitary Fund. Our benchmark code is public, largely self-explanatory, and easily reproducible.

100 timed trials of single and parallel gates were run for each qubit count between 4 and 28 qubits. Three tests were performed: the quantum Fourier transform, ("QFT"), random circuits constructed from a universal gate set, and an idealized approximation of Google's Sycamore chip benchmark, as per [Sycamore]. Additionally, parallel single

qubits and random input CCX gates to 20 layer depth were chosen to highlight use cases Qrack is particularly well-suited for. The benchmarking code is available at https://github.com/vm6502q/simulator-benchmarks. Default build and runtime options were used for all candidates. **Notably, this means Qrack ran at single floating point accuracy whereas QCGPU and Qiskit ran at double floating point accuracy.** Also, all optional PyQrack "layers" except CPU/GPU "hybridization" were disabled for Sycamore circuits, as this led to a drastic improvement in performance.

The same Alienware 17 laptop device was used for all benchmarks, (BIOS version 1.8.0, Intel(R) Core(TM) i9-10980HK CPU @ 2.40GHz, NVIDIA GeForce RTX 2070 Super). Benchmarks were collected from the week of October 3, 2021 through October 16, 2021.

Comparative benchmarks included QCGPU, the Qiskit-Aer GPU simulator, and Qrack's default typically optimal "stack" of a "QUnit" layer on top of "QStabilizerHybrid," on top of "QPager," on top of a new Pauli gate fusion layer, on top of "QHybrid." All of these candidates are GPU-based, though Qrack "hybridizes" with CPU based simulation as appropriate to improve performance.

QFT benchmarks could be implemented in a straightforward manner on all simulators, and were run as such. Qrack appears to be the only candidate considered for which inputs into the QFT can (drastically) affect its execution time, with permutation basis states being run in much shorter time, for example, hence only Qrack required a more general random input, whereas all other simulators were started in the |0> state. For a sufficiently representatively general test, Qrack instead used registers of single separable qubits intialized with uniformly randomly distributed probability between |0> and |1>, and uniformly randomly distributed phase offset between those states. Random permutation basis eigenstate initialization is also presented for Qrack, to demonstrate the quantitative difference, though we do not think this a representative test in itself.

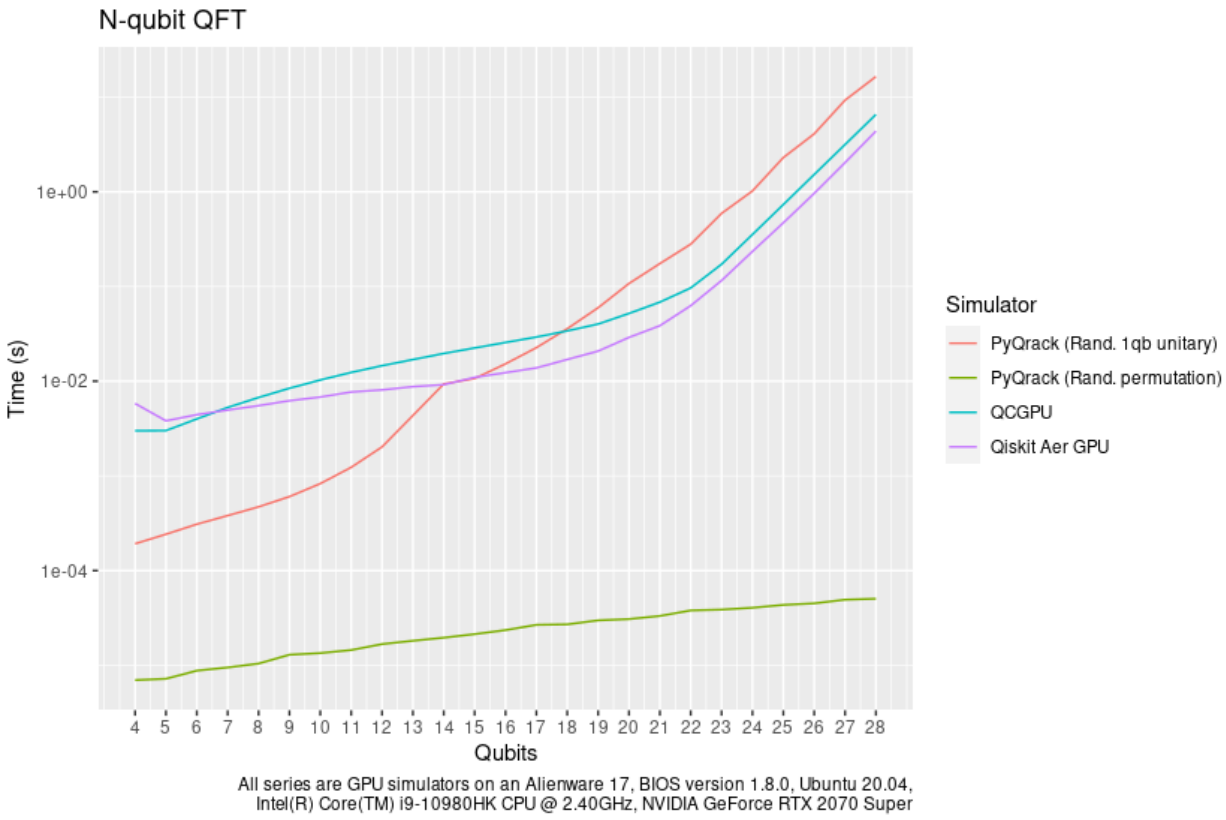
Random universal circuits carried out layers of single qubit gates on every qubit in the width of the test, followed by layers randomly selected couplings of (2-qubit) SWAP, CZ, CNOT, or (3-qubit) CCNOT, eliminating each selected bit for the layer. 20 layers of 1-qubit-plus-multi-qubit iterations were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

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

Qrack QEngine type heap usage was established as very closely matching theoretical expections, in earlier benchmarks, and this has not fundamentally changed. QUnit type heap usage varies greatly dependent on use case, though not in significant excess of QEngine types. No representative RAM benchmarks have been established for QUnit types, yet.

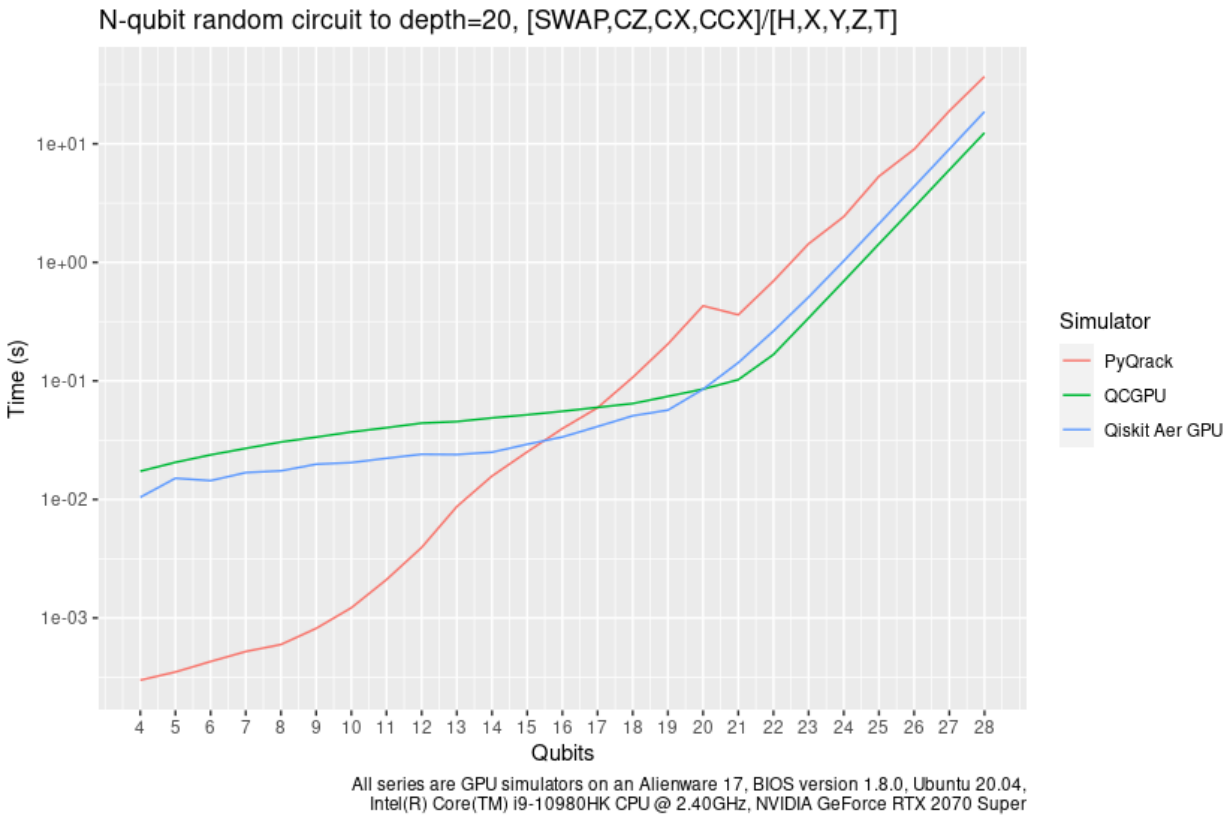
2.6.4 Results

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

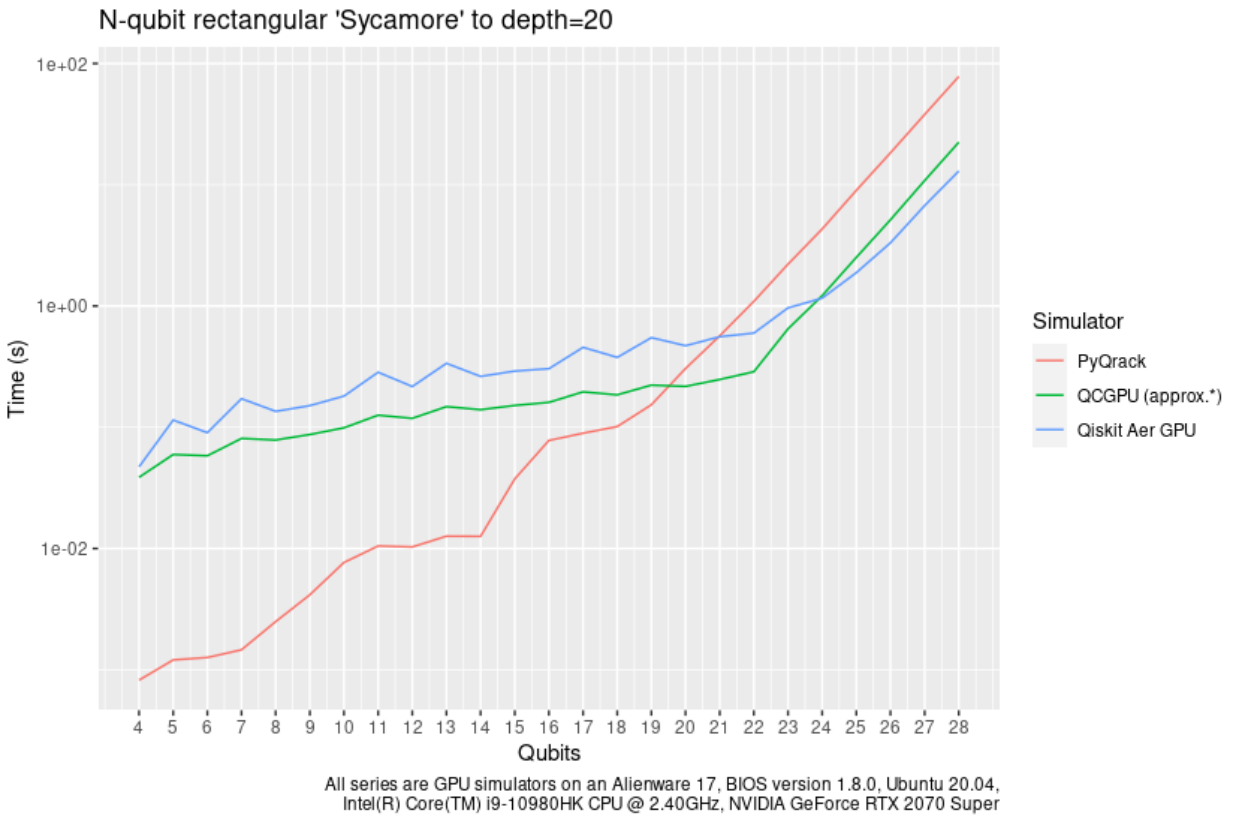


Likely due to a combination of all of its optimization "layers" and techniques, including Schmidt decomposition, "hybridization" of CPU with GPU simulation, and "hybridization" of stabilizer methods with "Schrödinger method," Qrack clearly outperforms purely GPU based simulations at low qubit widths. Recall that Qrack uses a representatively "hard" initialization with uniformly random single qubit unitary gates on this test as described above. We can see on the faster PyQrack series, permutation basis eigenstate inputs are much more quickly executed, for example. Qrack has historically been the only candidate tested which exhibits special case performance on the QFT, as for random permutation basis eigenstate initialization.

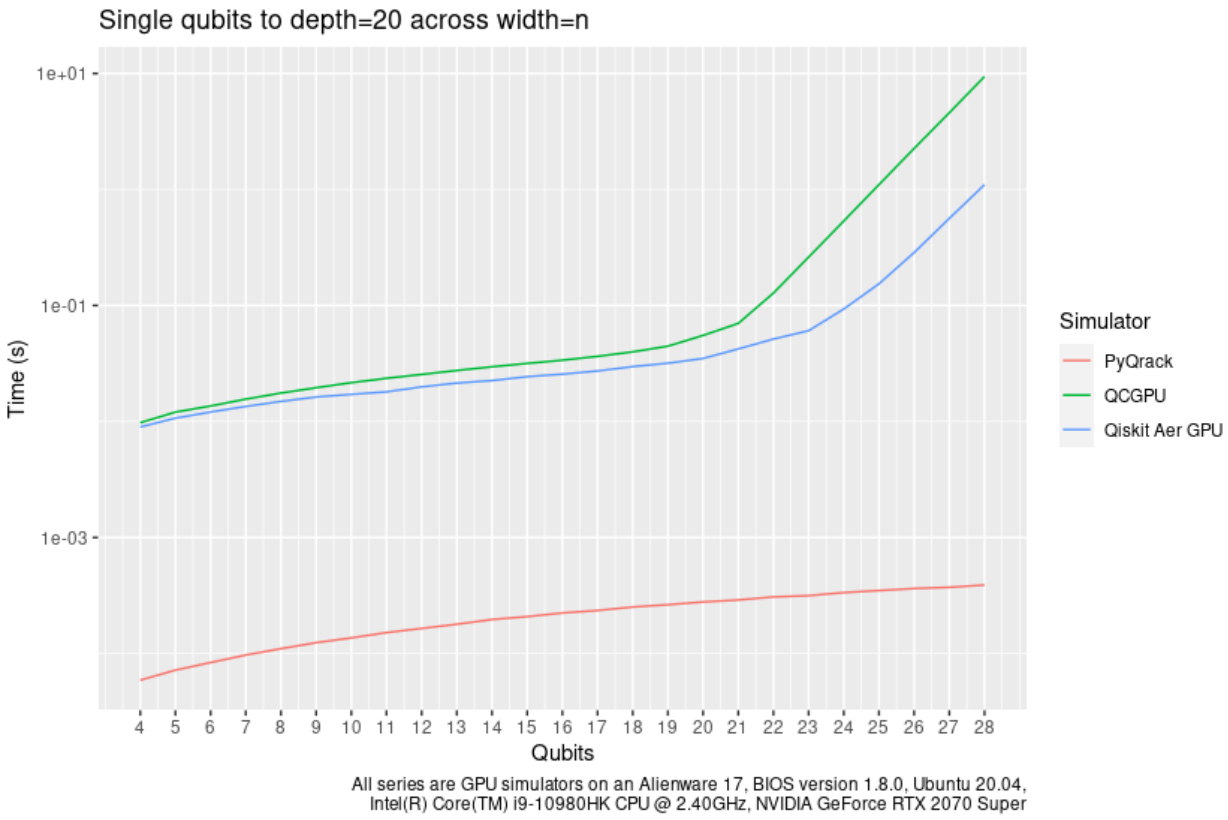
Similarly, on random universal circuits, defined above and in the benchmark repository, Qrack leads at low qubit widths.



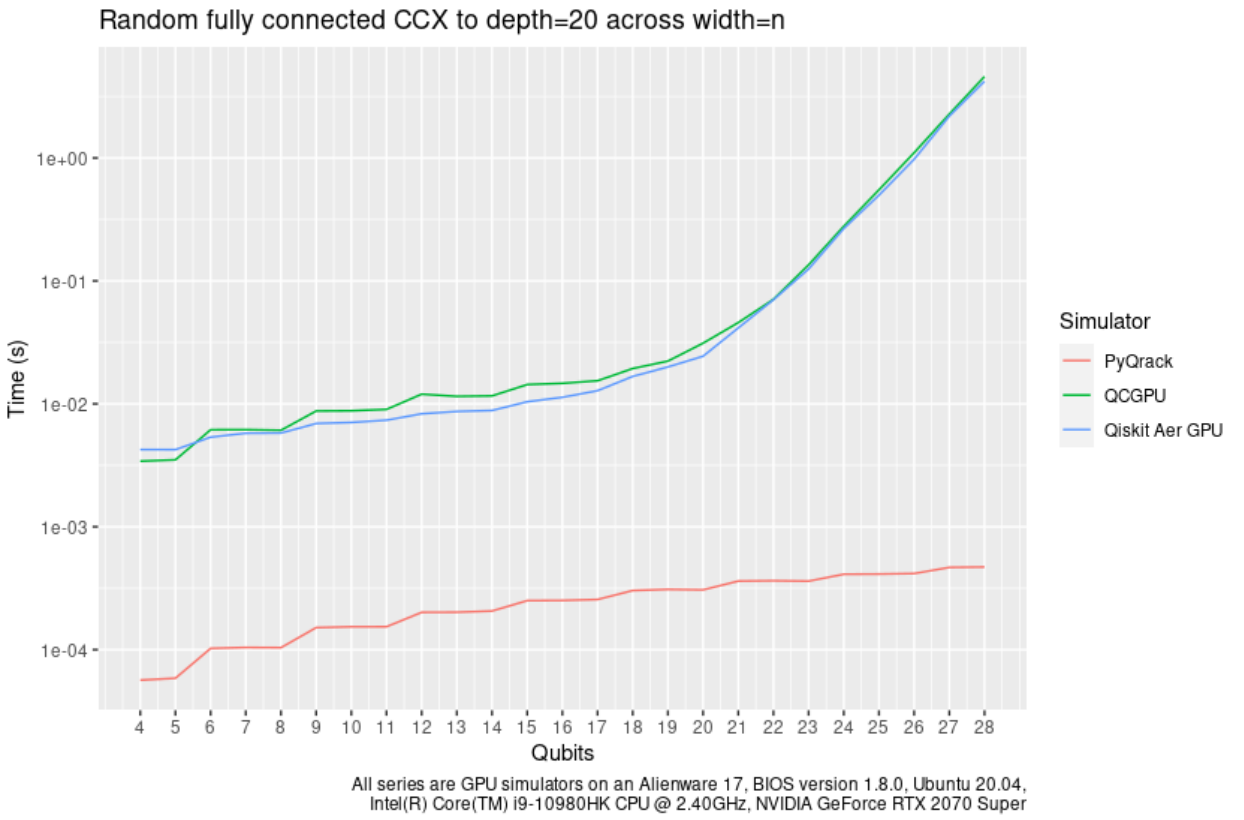
For “Sycamore” circuits, argued by other authors to establish “quantum supremacy” of native quantum hardware, Qrack maintains is low-width relative performance edge, (with PyQrack optimization options disabled except CPU/GPU hybridization, but still using “paging” with the C++ “QPager” layer).



However, we can furnish examples of circuits where Qrack has a commanding natural edge over naive "Schrödinger method," but also state-of-the-art in optimizing transpilation, (which could be additionally "layered" over Qrack itself).



The above graph is a test of (non-Clifford) single qubit gates, to depth of 20 on each qubit in simulator width, in parallel across the full width of a simulator instance. Qrack is transparently able to handle this circuit via "Schmidt decomposition," as single separable qubit subsystems, (similar to tensor network "matrix product states,") completing the test in linear time, over qubit widths. Surprisingly, Qiskit transpilation makes no great difference in execution time, in this case, despite successfully reducing depth of 20 qubits to depth of 1. We could guess, Qiskit might handle this via conventional "gate fusion" techniques, even without transpilation. However, even reducing circuit depth to 1 for each qubit in the width, Qiskit still suffers an exponential complexity disadvantage, since it still relies on fully connected "Schrödinger method" at base.



Our last case for consideration is CCNOT (or CCX) gate input qubit combinations selected at random (with elimination) from an implicitly fully-connected topology, across the width of the simulator, repeated for 20 layers of depth, initialized with a random permutation basis eigenstate, so as to produce a non-trivial change in the state of the qubits. Remember, CCNOT is specifically non-Clifford, but it is also sufficient in itself to serve as a single gate basis for the entirety of "classical" computation, "universally." So, we see that Qrack is capable of simulating a quantum computer that is emulating a "classical" computer, perhaps trivially. Rather, while we have no direct need to simulate the case of a quantum computer emulating classical computation, it is not granted that any other major quantum computer simulator is able to recognize and handle this case, "transparently," with full and automatic interoperability with all parts of the simulator API, without quite a bit of "cleverness" upon the part of the end-user, and user code labor. Again, Qiskit transpilation, not depicted, will reduce circuit depth, but rather signficantly increase simulator execution time, in this case.

2.6.5 Discussion

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

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

Qrack maintains a low-width edge over other GPU simulations by "hybridizing" CPU simulation with GPU simulation. Below system-responsive default thresholds, Qrack is simulating via CPU only, with a transparent transition to GPU simulation (and then "paged" GPU simulation) as qubit width is increased.

2.6.6 Further Work

The above results will be presented at the "Advanced Simulations of Quantum Computations Workshop," at QCE'21.

An option to simulate with CUDA, as opposed to OpenCL, might benefit execution time on systems with NVIDIA devices, such as the one used to collect test results for this page.

With the PyQrack layer functioning well, we have optionally wrapped it in a Qiskit ProviderV1 module. The provider module has not had an "official release," pending performance and stability improvements, but it is publicly available as open source software on GitHub. Further development and experiments will be done to assess the feasibility of improving Qrack and PyQrack performance with the Qiskit framework.

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

2.6.7 Conclusion

Per [Pednault2017], and many other attendant and synergistic optimizations engineered specifically in Qrack's QUnit, explicitly separated subsystems of qubits in QUnit have a significant RAM and speed edge in many cases over the Schrödinger algorithm of most popular quantum computer simulators. With QPager, it is possible to achieve even higher qubit widths and execution speeds. Qrack gives very efficient performance on a single node past 30 qubits, up to the limit of maximal entanglement.

2.6.8 Citations

2.7 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.

Inherits from Qrack::ParallelFor

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

2.7.1 Creating a QInterface

These are the primary implementations of a `QInterface`:

enum `Qrack::QInterfaceEngine`

Enumerated list of supported engines.

Use QINTERFACE_OPTIMAL for the best supported engine.

Values:

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

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

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

QINTERFACE_BDT
Create a QBinaryDecisionTree, (CPU-based).

QINTERFACE_MASK_FUSION
Create a QMaskFusion, coalescing Pauli gates.

QINTERFACE_STABILIZER
Create a QStabilizer, limited to Clifford/Pauli operations, but efficient.

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

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

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

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

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

QINTERFACE_OPTIMAL_SCHROEDINGER = QINTERFACE_CPU

QINTERFACE_OPTIMAL_SINGLE_PAGE = QINTERFACE_MASK_FUSION

QINTERFACE_OPTIMAL_BASE = QINTERFACE_CPU

QINTERFACE_OPTIMAL = QINTERFACE_QUNIT

QINTERFACE_OPTIMAL_MULTI = QINTERFACE_QUNIT_MULTI

QINTERFACE_MAX

These enums can be passed to an allocator to create a `QInterface` of that specified implementation type:
template <typename... *Ts*>
QInterfacePtr `Qrack::`**CreateQuantumInterface**(std::vector<*QInterfaceEngine*> *engines*, Ts... *args*)

2.7.2 Constructors

`Qrack::`*`QInterface`*`::`**QInterface**(bitLenInt *n*, qrack_rand_gen_ptr *rgp* = nullptr, bool *doNorm* = false, bool *useHardwareRNG* = true, bool *randomGlobalPhase* = true, real1_f *norm_thresh* = REAL1_EPSILON)

`Qrack::`*`QInterface`*`::`**QInterface**()
Default constructor, primarily for protected internal use.

2.7.3 Copying

virtual QInterfacePtr Qrack::*QInterface*::**Clone**() = 0
Clone this *QInterface*.

2.7.4 Members

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

2.7.5 Configuration Methods

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

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

virtual bool Qrack::*QInterface*::**isBinaryDecisionTree**()
Returns "true" if current state representation is definitely a binary decision tree, "false" if it is definitely not, or "true" if it cannot be determined.

virtual bool Qrack::*QInterface*::**isClifford**()
Returns "true" if current state is identifiably within the Clifford set, or "false" if it is not or cannot be determined.

virtual bool Qrack::*QInterface*::**isClifford**(bitLenInt *qubit*)
Returns "true" if current qubit state is identifiably within the Clifford set, or "false" if it is not or cannot be determined.

virtual void Qrack::*QInterface*::**SetReactiveSeparate**(bool *isAggSep*)
Set reactive separation option (on by default if available)

If reactive separation is available, as in Qrack::QUnit, then turning this option on attempts to more-aggresively recover separability of subsystems. It can either hurt or help performance, though it commonly helps.

virtual bool Qrack::*QInterface*::**GetReactiveSeparate**()
Get reactive separation option.

If reactive separation is available, as in Qrack::QUnit, then turning this option on attempts to more-aggresively recover separability of subsystems. It can either hurt or help performance, though it commonly helps.

virtual void Qrack::*QInterface*::**SetDevice**(int *dID*, bool *forceReInit* = false)
Set the device index, if more than one device is available.

virtual int64_t Qrack::*QInterface*::**GetDevice**()
Get the device index.

("-1" is default).

bitCapIntOcl Qrack::*QInterface*::**GetMaxSize**()
Get maximum number of amplitudes that can be allocated on current device.

2.7.6 State Manipulation Methods

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

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

Warning PSEUDO-QUANTUM

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

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

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

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

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

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

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

```
- virtual QInterfacePtr Qrack::QInterface::Decompose(bitLenInt, bitLenInt) = 0
- virtual void Qrack::QInterface::Decompose(bitLenInt, QInterfacePtr) = 0
```

virtual void Qrack::*QInterface*::**Dispose**(bitLenInt *start*, bitLenInt *length*) = 0
Minimally decompose a set of contiguous bits from the separably composed unit, and discard the separable bits from index "start" for "length.".

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

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

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

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

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

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

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

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

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

Warning PSEUDO-QUANTUM

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

Direct measure of full permutation probability.

Warning PSEUDO-QUANTUM

real1_f Qrack::*QInterface*::**ProbReg**(bitLenInt *start*, bitLenInt *length*, bitCapInt *permutation*)

Direct measure of register permutation probability.

Returns probability of permutation of the register.

Warning PSEUDO-QUANTUM

real1_f Qrack::*QInterface*::**ProbMask**(bitCapInt *mask*, bitCapInt *permutation*)

Direct measure of masked permutation probability.

Returns probability of permutation of the mask.

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

Warning PSEUDO-QUANTUM

void Qrack::*QInterface*::**ProbMaskAll**(bitCapInt *mask*, real1 **probsArray*)

Direct measure of masked permutation probability.

"mask" masks the bits to check the probability of. The probabilities of all permutations of the masked bits, from left/low to right/high are returned in the "probsArray" argument.

Warning PSEUDO-QUANTUM

void Qrack::*QInterface*::**ProbBitsAll**(**const** bitLenInt **bits*, bitLenInt *length*, real1 **probsArray*)

Direct measure of listed permutation probability.

The probabilities of all included permutations of bits, with bits valued from low to high as the order of the "bits" array parameter argument, are returned in the "probsArray" parameter.

Warning PSEUDO-QUANTUM

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

Warning PSEUDO-QUANTUM

real1_f Qrack::*QInterface*::**ExpectationBitsAll**(**const** bitLenInt **bits*, bitLenInt *length*, bitCapInt *offset* = 0)
Get permutation expectation value of bits.

The permutation expectation value of all included bits is returned, with bits valued from low to high as the order of the "bits" array parameter argument.

Warning PSEUDO-QUANTUM

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

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

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

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

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

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

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

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

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

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

virtual void Qrack::*QInterface*::**FSim**(real1_f *theta*, real1_f *phi*, bitLenInt *qubitIndex1*, bitLenInt *qubitIndex2*) = 0
The 2-qubit "fSim" gate, (useful in the simulation of particles with fermionic statistics)

virtual void Qrack::*QInterface*::**FSim**(real1_f *theta*, real1_f *phi*, bitLenInt *qubitIndex1*, bitLenInt *qubitIndex2*) = 0
The 2-qubit "fSim" gate, (useful in the simulation of particles with fermionic statistics)

void Qrack::*QInterface*::**UniformlyControlledRY**(**const** bitLenInt **controls*, bitLenInt *controlLen*, bitLenInt *qubitIndex*, **const** real1 **angles*)
Apply a "uniformly controlled" rotation of a bit around the Pauli Y axis.

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

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

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

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

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

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

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

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

Warning: doxygenfunction: Cannot find function "Qrack::QInterface::UniformParityRZ" in doxygen xml output for project "qrack" from directory: /tmp/qrack/doc/xml

Warning: doxygenfunction: Cannot find function "Qrack::QInterface::CUniformParityRZ" in doxygen xml output for project "qrack" from directory: /tmp/qrack/doc/xml

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

Reverse all of the bits in a sequence.

virtual bool Qrack::*QInterface*::**TrySeparate**(bitLenInt *qubit*)

Single-qubit TrySeparate()

virtual bool Qrack::*QInterface*::**TrySeparate**(bitLenInt *qubit1*, bitLenInt *qubit2*)

Two-qubit TrySeparate()

bool Qrack::*QInterface*::**TryDecompose**(bitLenInt *start*, QInterfacePtr *dest*, real1_f *error_tol* = TRYDECOMPOSE_EPSILON)

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

Statistical measure of masked permutation probability.

"qPowers" contains powers of 2^n, each representing *QInterface* bit "n." The order of these values defines a mask for the result bitCapInt, of 2^0 ~ qPowers[0] to 2^(qPowerCount - 1) ~ qPowers[qPowerCount - 1], in contiguous ascending order. "shots" specifies the number of samples to take as if totally re-preparing the pre-measurement state. This method returns a dictionary with keys, which are the (masked-order) measurement results, and values, which are the number of "shots" that produced that particular measurement result. This

method does not "collapse" the state of this *QInterface*. (The idea is to efficiently simulate a potentially statistically random sample of multiple re-preparations of the state right before measurement, and to collect random measurement resutls, without forcing the user to re-prepare or "clone" the state.)

Warning PSEUDO-QUANTUM

void Qrack::*QInterface*::**MultiShotMeasureMask**(**const** bitCapInt **qPowers*, bitLenInt *qPowerCount*, unsigned *shots*, unsigned **shotsArray*)

Statistical measure of masked permutation probability (returned as array)

Same `Qrack::MultiShotMeasureMask()`, except the shots are returned as an array.

Warning PSEUDO-QUANTUM

virtual bool Qrack::*QInterface*::**ApproxCompare**(QInterfacePtr *toCompare*, real1_f *error_tol* = TRYDECOMPOSE_EPSILON)

Compare state vectors approximately, component by component, to determine whether this state vector is the same as the target.

Warning PSEUDO-QUANTUM

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

To define a Hamiltonian, give a vector of controlled single bit gates ("HamiltonianOp" instances) that are applied by left-multiplication in low-to-high vector index order on the state vector.

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

$$e^{i(A+B)t} = e^{iAt}e^{iBt} \quad (2.8)$$

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

$$e^{-iH_{N-1}t}e^{-iH_{N-2}t}\dots e^{-iH_0t}\left|\psi\right\rangle \quad (2.9)$$

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

Warning Hamiltonian components might not commute.

2.7.7 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.

Note: Qrack::QInterface also offers arithmetic logic unit (ALU) gates. See the Doxygen.

Single Gates

virtual void Qrack::*QInterface*::**Mtrx**(**const** complex **mtrx*, bitLenInt *qubitIndex*) = 0
Apply an arbitrary single bit unitary transformation.

virtual void Qrack::*QInterface*::**MCMtrx**(**const** bitLenInt **controls*, bitLenInt *controlLen*, **const** complex **mtrx*, bitLenInt *target*) = 0
Apply an arbitrary single bit unitary transformation, with arbitrary control bits.

void Qrack::*QInterface*::**MACMtrx**(**const** bitLenInt **controls*, bitLenInt *controlLen*, **const** complex **mtrx*, bitLenInt *target*)
Apply an arbitrary single bit unitary transformation, with arbitrary (anti-)control bits.

void Qrack::*QInterface*::**Phase**(**const** complex *topLeft*, **const** complex *bottomRight*, bitLenInt *qubitIndex*)
Apply a single bit transformation that only effects phase.

void Qrack::*QInterface*::**MCPhase**(**const** bitLenInt **controls*, bitLenInt *controlLen*, complex *topLeft*, complex *bottomRight*, bitLenInt *target*)
Apply a single bit transformation that only effects phase, with arbitrary control bits.

void Qrack::*QInterface*::**MACPhase**(**const** bitLenInt **controls*, bitLenInt *controlLen*, complex *topLeft*, complex *bottomRight*, bitLenInt *target*)
Apply a single bit transformation that only effects phase, with arbitrary (anti-)control bits.

void Qrack::*QInterface*::**Invert**(**const** complex *topRight*, **const** complex *bottomLeft*, bitLenInt *qubitIndex*)
Apply a single bit transformation that reverses bit probability and might effect phase.

void Qrack::*QInterface*::**MCInvert**(**const** bitLenInt **controls*, bitLenInt *controlLen*, complex *topRight*, complex *bottomLeft*, bitLenInt *target*)
Apply a single bit transformation that reverses bit probability and might effect phase, with arbitrary control bits.

void Qrack::*QInterface*::**MACInvert**(**const** bitLenInt **controls*, bitLenInt *controlLen*, complex *topRight*, complex *bottomLeft*, bitLenInt *target*)
Apply a single bit transformation that reverses bit probability and might effect phase, with arbitrary (anti-)control bits.

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

(Assumes the outputBit is in the 0 state)

void Qrack::*QInterface*::**CLAND**(bitLenInt *inputQBit*, bool *inputClassicalBit*, bitLenInt *outputBit*)
Quantum analog of classical "AND" gate.

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

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

(Assumes the outputBit is in the 0 state)

void Qrack::*QInterface*::**CLOR**(bitLenInt *inputQBit*, bool *inputClassicalBit*, bitLenInt *outputBit*)
Quantum analog of classical "OR" gate.

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

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

(Assumes the outputBit is in the 0 state)

void Qrack::*QInterface*::**CLXOR**(bitLenInt *inputQBit*, bool *inputClassicalBit*, bitLenInt *outputBit*)
Quantum analog of classical "XOR" gate.

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

virtual bool Qrack::*QInterface*::**M**(bitLenInt *qubitIndex*)
Measurement gate.

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

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

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

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

virtual bool Qrack::*QInterface*::**ForceM**(bitLenInt *qubit*, bool *result*, bool *doForce* = true, bool *doApply* = true) = 0
Act as if is a measurement was applied, except force the (usually random) result.

Warning PSEUDO-QUANTUM

void Qrack::*QInterface*::**U**(bitLenInt *target*, real1_f *theta*, real1_f *phi*, real1_f *lambda*)
General unitary gate.

Applies a gate guaranteed to be unitary, from three angles, as commonly defined, spanning all possible single bit unitary gates, (up to a global phase factor which has no effect on Hermitian operator expectation values).

virtual void Qrack::*QInterface*::**U2**(bitLenInt *target*, real1_f *phi*, real1_f *lambda*)
2-parameter unitary gate

Applies a gate guaranteed to be unitary, from two angles, as commonly defined.

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

Applies a Hadamard gate on qubit at "qubitIndex."

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

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

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

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

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

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

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

Apply 1/4 phase rotation.

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

void Qrack::*QInterface*::**IS**(bitLenInt *qubitIndex*)
Inverse S gate.

Apply inverse 1/4 phase rotation.

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

virtual void Qrack::*QInterface*::**SH**(bitLenInt *qubitIndex*)
Y-basis transformation gate.

Converts from Pauli Z basis to Y, (via H then S gates).

virtual void Qrack::*QInterface*::**HIS**(bitLenInt *qubitIndex*)
Y-basis (inverse) transformation gate.

Converts from Pauli Y basis to Z, (via IS then H gates).

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

Apply 1/8 phase rotation.

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

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

Apply inverse 1/8 phase rotation.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

void Qrack::*QInterface*::**PhaseRootN**(bitLenInt *n*, bitLenInt *qubitIndex*)
"PhaseRootN" gate

Apply 1/(2^N) phase rotation.

Applies a 1/(2^N) phase rotation to the qubit at "qubitIndex."

void Qrack::*QInterface*::**IPhaseRootN**(bitLenInt *n*, bitLenInt *qubitIndex*)
Inverse "PhaseRootN" gate.

Apply inverse 1/(2^N) phase rotation.

Applies an inverse 1/(2^N) phase rotation to the qubit at "qubitIndex."

void Qrack::*QInterface*::**CPhaseRootN**(bitLenInt *n*, bitLenInt *control*, bitLenInt *target*)
Controlled "PhaseRootN" gate.

Apply controlled "PhaseRootN" gate to bit.

If the "control" bit is set to 1, then the "PhaseRootN" gate is applied to "target."

void Qrack::*QInterface*::**CIPhaseRootN**(bitLenInt *n*, bitLenInt *control*, bitLenInt *target*)
Controlled inverse "PhaseRootN" gate.

Apply controlled "IPhaseRootN" gate to bit.

If the "control" bit is set to 1, then the inverse "PhaseRootN" gate is applied to "target."

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

Controlled not.

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

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

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

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

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

Doubly-controlled not.

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

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

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

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

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

Apply controlled Pauli Y matrix to bit.

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

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

Apply controlled Pauli Z matrix to bit.

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

void Qrack::*QInterface*::**RT**(real1_f *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*::**RX**(real1_f *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*::**RY**(real1_f *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*::**RZ**(real1_f *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*::**CRZ**(real1_f *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.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

2.7.8 Arithmetic

Qrack can build with quantum arithmetic methods, using CMake option ‘‘-DENABLE_ALU=ON‘.

void Qrack::*QInterface*::**INC**(bitCapInt *toAdd*, bitLenInt *start*, bitLenInt *length*)

Add integer (without sign)

void Qrack::*QInterface*::**DEC**(bitCapInt *toSub*, bitLenInt *start*, bitLenInt *length*)
Subtract classical integer (without sign)

Subtract integer (without sign)

void Qrack::*QInterface*::**CINC**(bitCapInt *toAdd*, bitLenInt *inOutStart*, bitLenInt *length*, **const** bitLenInt *controls, bitLenInt *controlLen*)
Add integer (without sign, with controls)

void Qrack::*QInterface*::**CDEC**(bitCapInt *toSub*, bitLenInt *inOutStart*, bitLenInt *length*, **const** bitLenInt *controls, bitLenInt *controlLen*)
Subtract classical integer (without sign, with controls)

Subtract integer (without sign, with controls)

void Qrack::*QInterface*::**INCC**(bitCapInt *toAdd*, bitLenInt *start*, bitLenInt *length*, bitLenInt *carryIndex*)
Add integer (without sign, with carry)

void Qrack::*QInterface*::**INCS**(bitCapInt *toAdd*, bitLenInt *start*, bitLenInt *length*, bitLenInt *overflowIndex*)
Add a classical integer to the register, with sign and without carry.

void Qrack::*QInterface*::**DECS**(bitCapInt *toSub*, bitLenInt *start*, bitLenInt *length*, bitLenInt *overflowIndex*)
Subtract a classical integer from the register, with sign and without carry.

Subtract an integer from the register, with sign and without carry.

Because the register length is an arbitrary number of bits, the sign bit position on the integer to add is variable. Hence, the integer to add is specified as cast to an unsigned format, with the sign bit assumed to be set at the appropriate position before the cast.

Warning: doxygenfunction: Cannot find function “Qrack::QInterface::INCSC” in doxygen xml output for project “qrack” from directory: /tmp/qrack/doc/xml

Warning: doxygenfunction: Cannot find function “Qrack::QInterface::INCSC” in doxygen xml output for project “qrack” from directory: /tmp/qrack/doc/xml

Warning: doxygenfunction: Cannot find function “Qrack::QInterface::MUL” in doxygen xml output for project “qrack” from directory: /tmp/qrack/doc/xml

Warning: doxygenfunction: Cannot find function “Qrack::QInterface::DIV” in doxygen xml output for project “qrack” from directory: /tmp/qrack/doc/xml

Warning: doxygenfunction: Cannot find function “Qrack::QInterface::CMUL” in doxygen xml output for project “qrack” from directory: /tmp/qrack/doc/xml

Warning: doxygenfunction: Cannot find function “Qrack::QInterface::CDIV” in doxygen xml output for project “qrack” from directory: /tmp/qrack/doc/xml

Warning: doxygenfunction: Cannot find function “Qrack::QInterface::MULModNOut” in doxygen xml output for project “qrack” from directory: /tmp/qrack/doc/xml

Warning: doxygenfunction: Cannot find function “Qrack::QInterface::IMULModNOut” in doxygen xml output for project “qrack” from directory: /tmp/qrack/doc/xml

Warning: doxygenfunction: Cannot find function “Qrack::QInterface::CMULModNOut” in doxygen xml output for project “qrack” from directory: /tmp/qrack/doc/xml

Warning: doxygenfunction: Cannot find function “Qrack::QInterface::CIMULModNOut” in doxygen xml output for project “qrack” from directory: /tmp/qrack/doc/xml

Warning: doxygenfunction: Cannot find function “Qrack::QInterface::POWModNOut” in doxygen xml output for project “qrack” from directory: /tmp/qrack/doc/xml

Warning: doxygenfunction: Cannot find function “Qrack::QInterface::CPOWModNOut” in doxygen xml output for project “qrack” from directory: /tmp/qrack/doc/xml

void Qrack::*QInterface*::**FullAdd**(bitLenInt *inputBit1*, bitLenInt *inputBit2*, bitLenInt *carryInSumOut*, bitLenInt *carryOut*)

Quantum analog of classical “Full Adder” gate.

(Assumes the outputBit is in the 0 state)

void Qrack::*QInterface*::**IFullAdd**(bitLenInt *inputBit1*, bitLenInt *inputBit2*, bitLenInt *carryInSumOut*, bitLenInt *carryOut*)

Inverse of FullAdd.

(Can be thought of as “subtraction,” but with a register convention that the same inputs invert FullAdd.)

void Qrack::*QInterface*::**CFullAdd**(**const** bitLenInt **controls*, bitLenInt *controlLen*, bitLenInt *inputBit1*, bitLenInt *inputBit2*, bitLenInt *carryInSumOut*, bitLenInt *carryOut*)

Controlled quantum analog of classical “Full Adder” gate.

Quantum analog of classical “Full Adder” gate.

(Assumes the outputBit is in the 0 state)

void Qrack::*QInterface*::**CIFullAdd**(**const** bitLenInt **controls*, bitLenInt *controlLen*, bitLenInt *inputBit1*, bitLenInt *inputBit2*, bitLenInt *carryInSumOut*, bitLenInt *carryOut*)

Inverse of CFullAdd.

Inverse of FullAdd.

(Can be thought of as "subtraction," but with a register convention that the same inputs invert CFullAdd.)

void Qrack::*QInterface*::**ADC**(bitLenInt *input1*, bitLenInt *input2*, bitLenInt *output*, bitLenInt *length*, bitLenInt *carry*)
Add a quantum integer to a quantum integer, with carry.

(Assumes the output register is in the 0 state)

void Qrack::*QInterface*::**IADC**(bitLenInt *input1*, bitLenInt *input2*, bitLenInt *output*, bitLenInt *length*, bitLenInt *carry*)
Inverse of ADC.

(Can be thought of as "subtraction," but with a register convention that the same inputs invert ADC.)

void Qrack::*QInterface*::**CADC**(**const** bitLenInt **controls*, bitLenInt *controlLen*, bitLenInt *input1*, bitLenInt *input2*, bitLenInt *output*, bitLenInt *length*, bitLenInt *carry*)
Add a quantum integer to a quantum integer, with carry and with controls.

(Assumes the output register is in the 0 state)

void Qrack::*QInterface*::**CIADC**(**const** bitLenInt **controls*, bitLenInt *controlLen*, bitLenInt *input1*, bitLenInt *input2*, bitLenInt *output*, bitLenInt *length*, bitLenInt *carry*)
Inverse of CADC.

(Can be thought of as "subtraction," but with a register convention that the same inputs invert CADC.)

Warning: doxygenfunction: Cannot find function "Qrack::QInterface::IndexedLDA" in doxygen xml output for project "qrack" from directory: /tmp/qrack/doc/xml

Warning: doxygenfunction: Cannot find function "Qrack::QInterface::IndexedADC" in doxygen xml output for project "qrack" from directory: /tmp/qrack/doc/xml

Warning: doxygenfunction: Cannot find function "Qrack::QInterface::IndexedSBC" in doxygen xml output for project "qrack" from directory: /tmp/qrack/doc/xml

Warning: doxygenfunction: Cannot find function "Qrack::QInterface::Hash" in doxygen xml output for project "qrack" from directory: /tmp/qrack/doc/xml

2.7.9 Algorithmic Implementations

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

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

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

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

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

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

void Qrack::*QInterface*::**QFTR**(**const** bitLenInt **qubits*, bitLenInt *length*, bool *trySeparate* = false)
Quantum Fourier Transform (random access) - Apply the quantum Fourier transform to the register.

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

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

void Qrack::*QInterface*::**IQFTR**(**const** bitLenInt **qubits*, bitLenInt *length*, bool *trySeparate* = false)
Inverse Quantum Fourier Transform (random access) - Apply the inverse quantum Fourier transform to the register.

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

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

2.8 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.

2.8.1 Creating an OCLEngine

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

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

2.8.2 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.

2.9 QEngine

Defined in qengine.hpp.

This is an (abstract) intermediate specialization that inherits from Qrack::QInterface. This type is specifically a "state vector" simulation, with corresponding special methods.

Qrack::QEngine::**QEngine**(bitLenInt *qBitCount*, qrack_rand_gen_ptr *rgp* = nullptr, bool *doNorm* = false, bool *randomGlobalPhase* = true, bool *useHostMem* = false, bool *useHardwareRNG* = true, real1_f *norm_thresh* = REAL1_EPSILON)

Qrack::QEngine::**QEngine**()

Default constructor, primarily for protected internal use.

virtual void Qrack::QEngine::**ZeroAmplitudes**() = 0

Set all amplitudes to 0, and optionally temporarily deallocate state vector RAM.

virtual void Qrack::QEngine::**CopyStateVec**(QEnginePtr *src*) = 0

Exactly copy the state vector of a different QEngine instance.

virtual bool Qrack::QEngine::**IsZeroAmplitude**() = 0

Returns "true" only if amplitudes are all totally 0.

virtual void Qrack::QEngine::**GetAmplitudePage**(complex **pagePtr*, bitCapIntOcl *offset*, bitCapIntOcl *length*) = 0

Copy a "page" of amplitudes from this QEngine's internal state, into `pagePtr`.

Warning: doxygenfunction: Unable to resolve multiple matches for function "Qrack::QEngine::SetAmplitudePage" with arguments (const complex *, const bitCapIntOcl, const bitCapIntOcl) in doxygen xml output for project "qrack" from directory: /tmp/qrack/doc/xml. Potential matches:

```
- virtual void Qrack::QEngine::SetAmplitudePage(QEnginePtr, bitCapIntOcl,␣
↪bitCapIntOcl, bitCapIntOcl) = 0
- virtual void Qrack::QEngine::SetAmplitudePage(const complex *, bitCapIntOcl,␣
↪bitCapIntOcl) = 0
```

Warning: doxygenfunction: Unable to resolve multiple matches for function "Qrack::QEngine::SetAmplitudePage" with arguments (QEnginePtr, const bitCapIntOcl, const bitCapIntOcl, const bitCapIntOcl) in doxygen xml output for project "qrack" from directory: /tmp/qrack/doc/xml. Potential matches:

```
- virtual void Qrack::QEngine::SetAmplitudePage(QEnginePtr, bitCapIntOcl,␣
↪bitCapIntOcl, bitCapIntOcl) = 0
- virtual void Qrack::QEngine::SetAmplitudePage(const complex *, bitCapIntOcl,␣
↪bitCapIntOcl) = 0
```

virtual void Qrack::QEngine::**ShuffleBuffers**(QEnginePtr *engine*) = 0
Swap the high half of this engine with the low half of another.

This is necessary for gates which cross sub-engine boundaries.

virtual void Qrack::QEngine::**QueueSetDoNormalize**(bool *doNorm*) = 0
Add an operation to the (OpenCL) queue, to set the value of doNormalize, which controls whether to automatically normalize the state.

virtual void Qrack::QEngine::**QueueSetRunningNorm**(real1_f *runningNrm*) = 0
Add an operation to the (OpenCL) queue, to set the value of runningNorm, which is the normalization constant for the next normalization operation.

2.10 QEngineCPU

Defined in qengine_cpu.hpp.

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

Warning: doxygenfunction: Cannot find function "Qrack::QEngineOCL::QEngineCPU" in doxygen xml output for project "qrack" from directory: /tmp/qrack/doc/xml

2.11 QEngineOCL

Defined in qengine_opencl.hpp.

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

Qrack::QEngineOCL::**QEngineOCL**(bitLenInt *qBitCount*, bitCapInt *initState*, qrack_rand_gen_ptr *rgp* = nullptr, complex *phaseFac* = CMPLX_DEFAULT_ARG, bool *doNorm* = false, bool *randomGlobalPhase* = true, bool *useHostMem* = false, int *devID* = -1, bool *useHardwareRNG* = true, bool *ignored* = false, real1_f *norm_thresh* = REAL1_EPSILON, std::vector<int> *ignored2* = {}, bitLenInt *ignored4* = 0, real1_f *ignored3* = FP_NORM_EPSILON)
Initialize a Qrack::QEngineOCL object.

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

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

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

2.12 QHybrid

Defined in qhybrid.hpp.

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

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

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

`virtual` void `Qrack::QHybrid::`**`SwitchModes`**(bool *useGpu*)

Switches between CPU and GPU used modes.

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

2.13 QStabilizerHybrid

Defined in qstabilizerhybrid.hpp.

The API is provided by Qrack::QInterface. This is an extended "stabilizer" or "Clifford" simulation. If a Qrack::QInterface method call cannot be carried out by this simulator via "stabilizer" simulation methods, it will automatically convert to traditional state vector representation, (while keep its Qrack::QStabilizerHybrid class type).

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

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

2.14 QUnit

Defined in qunit.hpp.

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

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

virtual bool Qrack::*QInterface*::**TrySeparate**(bitLenInt *qubit*)
Single-qubit TrySeparate()

virtual bool Qrack::*QInterface*::**TrySeparate**(bitLenInt *qubit1*, bitLenInt *qubit2*)
Two-qubit TrySeparate()

Warning: doxygenfunction: Unable to resolve multiple matches for function "Qrack::QInterface::TrySeparate" with arguments (bitLenInt *, bitLenInt, real1_f) in doxygen xml output for project "qrack" from directory: /tmp/qrack/doc/xml. Potential matches:

```
- virtual bool Qrack::QInterface::TrySeparate(bitLenInt)
- virtual bool Qrack::QInterface::TrySeparate(bitLenInt, bitLenInt)
- virtual bool Qrack::QInterface::TrySeparate(const bitLenInt *, bitLenInt, real1_f)
```

2.15 QUnitMulti

Defined in qunitmulti.hpp.

Qrack::QUnitMulti is a direct subclass of Qrack::QUnit that attempts to dispatch its separable internal subsystem work to multiple OpenCL devices, including all devices visible to the OpenCL context by default. (Qrack::QUnit maintains explicit separation of representation between separable subsystems, when possible and efficient, greatly reducing memory and execution time overhead.)

2.16 QBinaryDecisionTree

Defined in qbinary_decision_tree.hpp and qbinary_decision_node.hpp.

The API is provided by Qrack::QInterface. Qrack::QBinaryDecisionTree is an attempt at an alternative representation of quantum pure states, in terms of binary decision trees, inspired by other open source work.

Warning: doxygenfunction: Cannot find function “Qrack::QBinaryDecisionTree::QBinaryDecisionTree” in doxygen xml output for project “qrack” from directory: /tmp/qrack/doc/xml

Warning: doxygenfunction: Cannot find function “Qrack::QBinaryDecisionTree::QBinaryDecisionTree” in doxygen xml output for project “qrack” from directory: /tmp/qrack/doc/xml

2.17 MOS-6502Q Opcodes

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

Table 1: 6502Q New Opcodes

OP	Byte	Mode	Description
HAA	0x02	Implied	Bitwise Hadamard on the Accumulator
HAX	0x03	Implied	Bitwise Hadamard on the X Register
HAY	0x04	Implied	Bitwise Hadamard on the Y Register
SEN	0x0F	Implied	SEt the Negative flag
PXA	0x12	Implied	Apply a bitwise Pauli X on the Accumulator
PXX	0x13	Implied	Apply a bitwise Pauli X on the X Register
PXY	0x0C	Implied	Apply a bitwise Pauli X on the Y Register
HAC	0x17	Implied	Apply a Hadamard gate on the carry flag
PYA	0x1A	Implied	Apply a bitwise Pauli Y on the Accumulator
PYX	0x1B	Implied	Apply a bitwise Pauli Y on the X Register
PYY	0x1C	Implied	Apply a bitwise Pauli Y on the Y Register
CLQ	0x1F	Implied	CLear Quantum mode flag
SEV	0x27	Implied	SEt the oVerflow flag
SEZ	0x2B	Implied	SEt the Zero flag
CLN	0x2F	Implied	CLear the Negative flag
PZA	0x32	Implied	Apply a bitwise Pauli Z on Accumulator
PZX	0x33	Implied	Apply a bitwise Pauli Z on the X Register
PZY	0x34	Implied	Apply a bitwise Pauli Z on the Y Register
RTA	0x3A	Implied	Bitwise quarter rotation on $\|1\rangle$ axis for Accumulator
RTX	0x3B	Implied	Bitwise quarter rotation on $\|1\rangle$ axis for the X Register
RTY	0x3C	Implied	Bitwise quarter rotation on $\|1\rangle$ axis for the Y Register
SEQ	0x1F	Implied	SEt the Quantum mode flag
RXA	0x42	Implied	Bitwise quarter rotation on X axis for Accumulator
RXX	0x43	Implied	Bitwise quarter rotation on X axis for the X Register
RXY	0x44	Implied	Bitwise quarter rotation on X axis for the Y Register
CLZ	0x47	Implied	CLear the Zero flag
RZA	0x5A	Implied	Bitwise quarter rotation on Z axis for Accumulator
RZX	0x5B	Implied	Bitwise quarter rotation on Z axis for the X Register
RZY	0x5C	Implied	Bitwise quarter rotation on Z axis for the Y Register
FTA	0x62	Implied	Quantum Fourier Transform on Accumulator

Continued on next page

Table 1 – continued from previous page

OP	Byte	Mode	Description
FTX	0x63	Implied	Quantum Fourier Transform on the X register
FTY	0x64	Implied	Quantum Fourier Transform on the Y register
ADC	0x75	Zero page X addressing	ADd with Carry, Zero Page indexed, will add in superposition if the X register is superposed. Results in the Accumulator and carry flag become entangled with the X register, such that the result of the addition is entangled with the address loaded from in the X register. (Addressing past the zero page loops to the start.)
ADC	0x7D	Absolute X addressing	ADd with Carry, Zero Page indexed, will add in superposition if the X register is superposed. Results in the Accumulator and carry flag become entangled with the X register, such that the result of the addition is entangled with the address loaded from in the X register.
ADC	0x71	Implied Y addressing	ADd with Carry, Implied Y indexed, will add in superposition if the Y register is superposed. Results in the Accumulator and carry flag become entangled with the Y register, such that the result of the addition is entangled with the address loaded from in the Y register.
ADC	0x79	Absolute Y addressing	ADd with Carry, Zero Page indexed, will add in superposition if the Y register is superposed. Results in the Accumulator and carry flag become entangled with the Y register, such that the result of the addition is entangled with the address loaded from in the Y register.
TXA	0x8A	Implied	Transfer X register to Accumulator, will maintain superposition of the X register, entangling it to be the same as the Accumulator when measured
TXS	0x9A	Implied	Transfer X register to Stack pointer, will also collapse superposition of the X register
TAY	0xA8	Implied	Transfer Accumulator Y register, will maintain superposition of the Accumulator, entangling it to be the same as the Y register when measured
TAX	0x8A	Implied	Transfer Accumulator to X register, will maintain superposition of the Accumulator, entangling it to be the same as the X register when measured
LDA	0xB5	Zero page X addressing	LoaD Accumulator, Zero Page indexed, will load in superposition if the X register is superposed. Results loaded in the Accumulator become entangled with the X register, such that the result of the load is entangled with the address loaded from in the X register. (Addressing past the zero page loops to the start.
LDA	0xBD	Absolute X addressing	LoaD Accumulator, Zero Page indexed, will load in superposition if the X register is superposed. Results loaded in the Accumulator become entangled with the X register, such that the result of the load is entangled with the address loaded from in the X register.
SBC	0xF5	Zero page X addressing	SuBtract with Carry, Zero Page indexed, will subtract in superposition if the X register is superposed. Results in the Accumulator and carry flag become entangled with the X register, such that the result of the addition is entangled with the address loaded from in the X register. (Addressing past the zero page loops to the start.
SBC	0xFD	Absolute X addressing	SuBtract with Carry, Zero Page indexed, will subtract in superposition if the X register is superposed. Results in the Accumulator and carry flag become entangled with the X register, such that the result of the addition is entangled with the address loaded from in the X register.
ADC	0xF1	Implied Y addressing	SuBtract with Carry, Implied Y indexed, will subtract in superposition if the Y register is superposed. Results in the Accumulator and carry flag become entangled with the Y register, such that the result of the subtraction is entangled with the address loaded from in the Y register.

Continued on next page

Table 1 – continued from previous page

OP	Byte	Mode	Description
ADC	0xF9	Absolute Y addressing	ADd with Carry, Zero Page indexed, will subtract in superposition if the Y register is superposed. Results in the Accumulator and carry flag become entangled with the Y register, such that the result of the subtraction is entangled with the address loaded from in the Y register.
QZZ	0xF7	Implied	Apply Pauli Z operator to zero flag
QZS	0xFA	Implied	Apply Pauli Z operator to negative flag
QZC	0xFB	Implied	Apply Pauli Z operator to carry flag

Table 2: 6502Q Modified Opcodes

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

Bibliography

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

[WiredSummary] Wired's Overview of the Industry

[AlgoZoo] Quantum Algorithm Zoo

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

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

[Grover] Grover Search Algorithm

[GroverSummary] Introduction to Implementing Grover's Search Algorithm

[GroverVisual] Visualization of Grover's Search Algorithm

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

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

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

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

[Pednault2017] Pednault, Edwin, et al. "Pareto-Efficient Quantum Circuit Simulation Using Tensor Contraction Deferral" arXiv preprint arXiv:1710.05867 (2017).

[QSharp] Q#

[QHiPSTER] QHipster

[Quantiki] Quantiki: List of QC simulators

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

Index

Q