A Framework for Efficient QC Simulation

Preview Draft 1, February 29 2012

Fine Print: This document is not finished, but you're welcome to read it anyway. There are still diagrams missing. If you have questions, comments, or really good sci-fi novel recommendations, please contact me at ej@machinelevel.com

Overview

The purpose of this section is to describe a software framework which facilitates quantum computer simulation reasonably efficiently, while being able to take advantage of a variety of computer architectures. An implementation of this framework can be seen at http://qc.machinelevel.com.

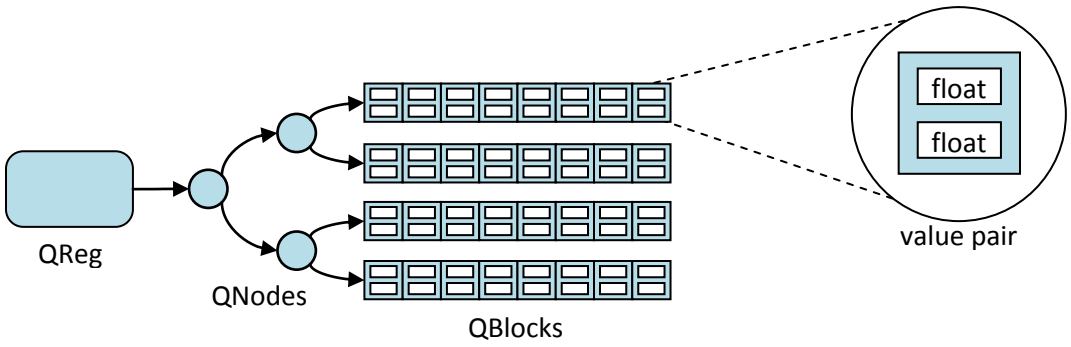
The simulation described here does not attempt to take heuristic shortcuts, so the time and space it consumes increase exponentially with the number of qubits being simulated. Specifically, to simulate *n* qubits, this framework requires $8*2^n$ bytes of memory (assuming single-precision floating point complex numbers)

The key advantages of this framework are:

- Many common operations can be done at very low (sometimes negligible) cost.
- Nearly perfect parallelization is easy to achieve.
- Very little extra buffer space is needed.
- Implementation can be done using a variety of architectures including standard CPU, SIMD-assisted CPU, GPU, and computing cluster. Combinations of these architectures may also be used.

Basic Terminology

This framework is made up of the following components:



QReg (quantum register) – The control structure and code interface for the collection of qubits to be simulated. This can in theory be any size, though in practice the exponential storage requirement will usually limit it to 30-40 qubits.

Value Pair – A single complex number which represents one possible value of the QReg. This is represented by a pair of floating point numbers (typically 8 or 16 bytes per pair, depending on whether single or double precision is used). A one-qubit QReg will require only two values: $|0\rangle$ and $|1\rangle$. An 8-qubit QReg needs 256 value pairs. For 32 qubits, about four billion value pairs are needed, requiring about 8 GB or so.

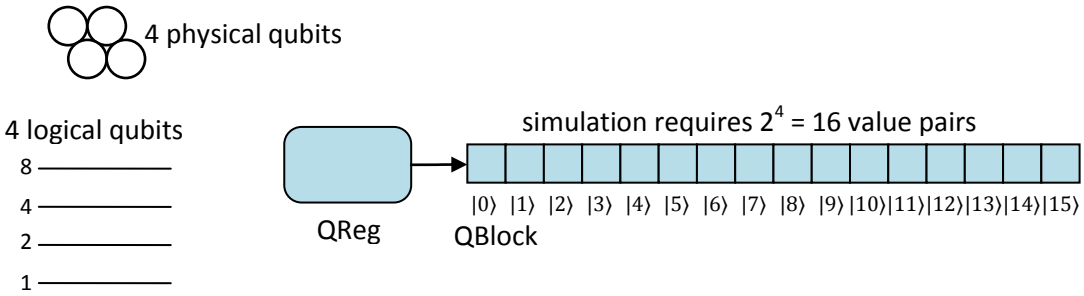
> **Important note**: All together, these value pairs simply represent one big multidimensional vector (2 components contributed by each pair). **At all times, this vector will have length 1.0**. It will be initialized to something like [1.0, 0.0, 0.0, 0.0, 0.0, …] and *no operation* will ever be allowed to change the length of the vector.

QBlock – An array of value pairs, stored as a contiguous block of memory. The number of value pairs on a QBlock is always a power of 2.

QNode – A binary tree node whose children are either both QNodes, or are both QBlocks.

Sample Case 1: Single Block

We'll start simple, and then add components and complexity in later sample cases. Consider the case where a QReg is 4 qubits, all stored in a single block. As data structures go, this is not very interesting. The QReg control struct has a pointer to a single QBlock. With 4 qubits, we'll need 2^4 = 16 value pairs. So the entire block will be 32 floats, which will occupy 128 bytes if we use single-precision.

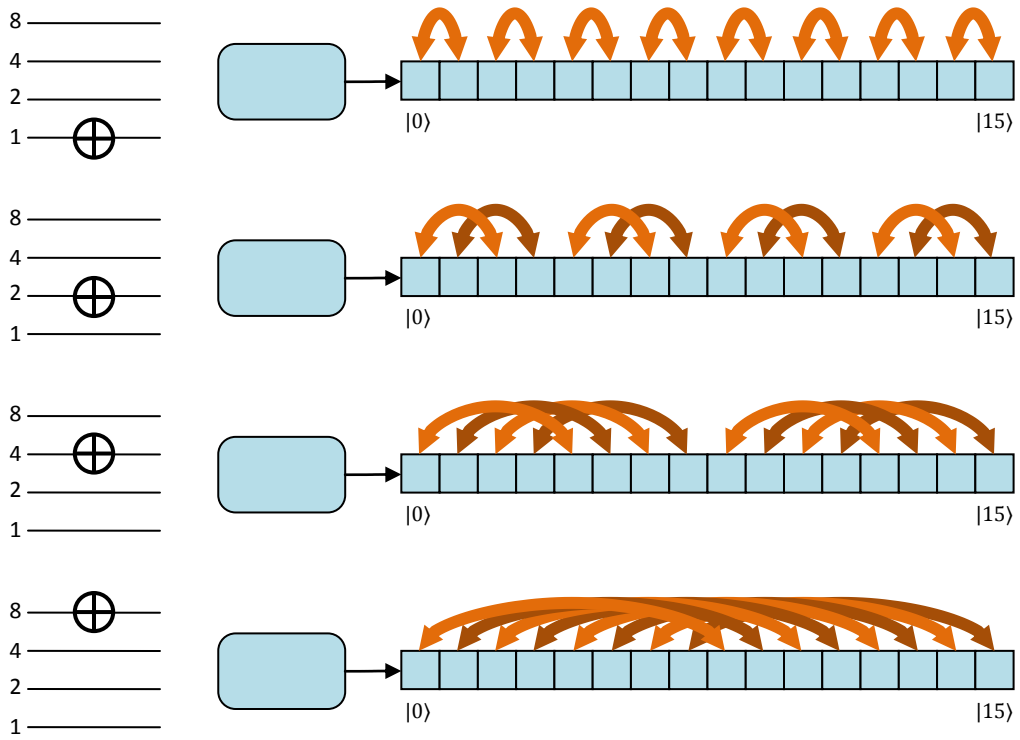


As mentioned above, the 32 floating-point numbers in our QBlock make up a vector which will *always* have unit length. Before starting the next section, we'll initialize the first float in this array to 1.0, and all of the others to 0.0, which represents the state $|0\rangle$. Now let's take a short look at doing some operations.

Operations: NOT ⊕

If you're familiar with quantum computation, you'll know this one already. In order to perform a NOT of qubit X, we simply swap each value pair with the pair whose value is the same, but with bit X flipped. For example, performing NOT on qubit 2 means we swap value pair 0 with 2, swap 1 with 3, swap 4 with 6, etc.

NOT Gate implemented by swapping value pairs

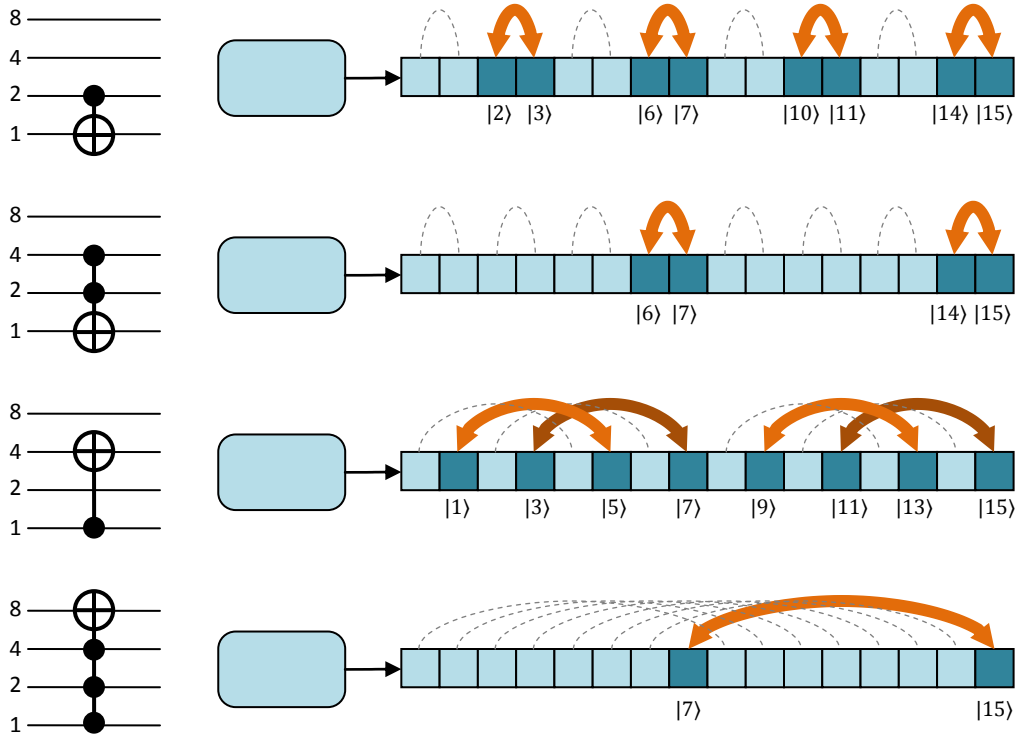


Although this operation is simple, it still may take a very long time, if the number of qubits is large. For 32 qubits, there will be 4 billion pairs to swap. Later on (in Sample Case 2), we'll look at a way to speed this up in some cases.

Operations: CNOT

For a controlled-not, we perform a NOT for any qubit pair where all of the condition bits are set.

CNOT Gate implemented by *selectively* swapping value pairs

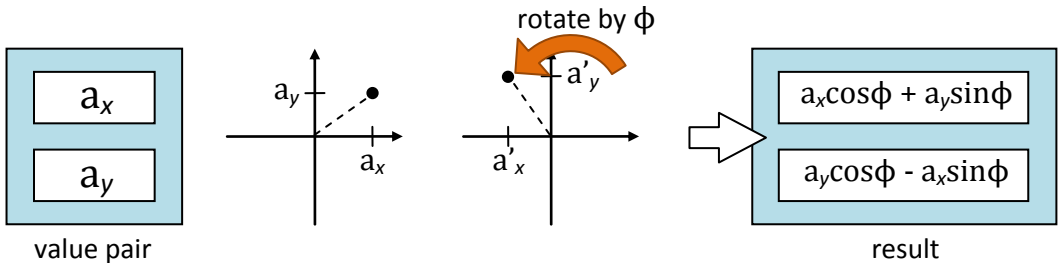


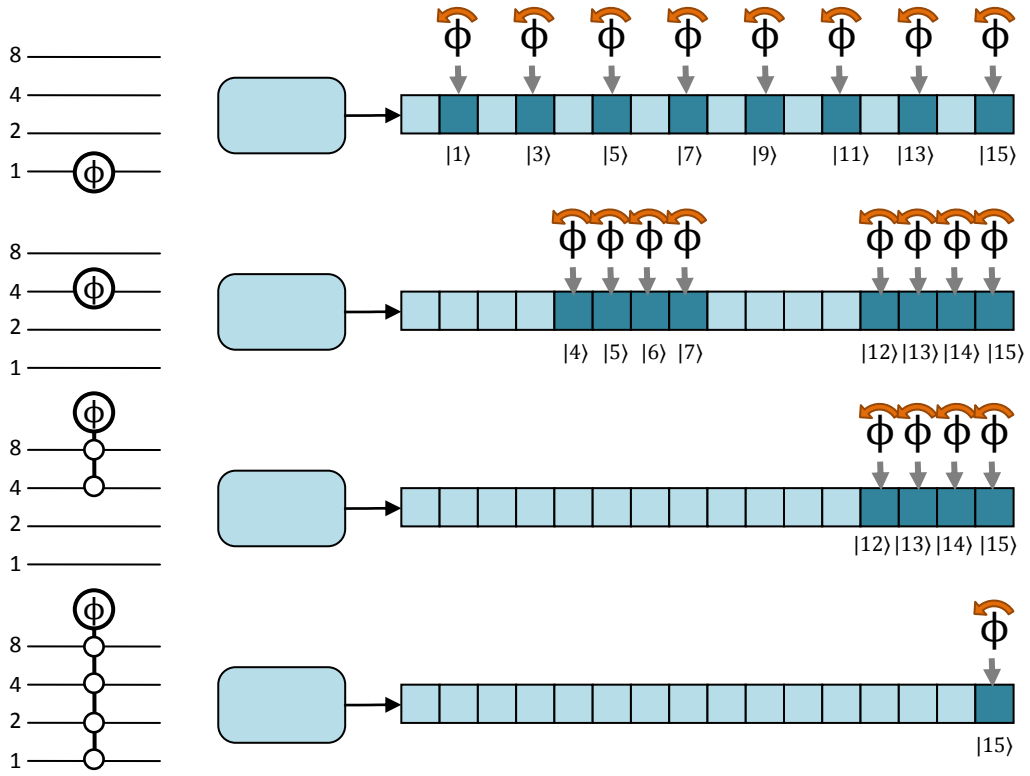
Operations: Phase Shift ϕ

To perform a phase shift of any number of qubits, first identify all value pairs which have the target bits set to 1. For each identified pair, consider the real and imaginary components to be X and Y components of a 2D vector, and rotate it.

When implementing this operation, it helps to keep in mind that the sin() and cos() functions only need to be called once, as phi is the same for all value pairs.

Phase Shift Gate implemented by *selectively* rotating value pair components



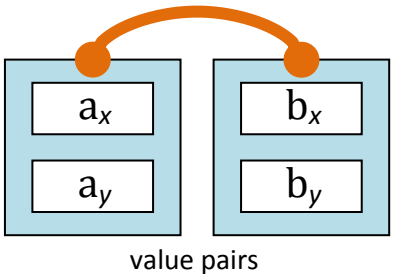


Note that I've chosen to draw the phase shift operator differently, because it's a bit of a different gate. There is no target qubit for the operation, and all of the "selected" qubits are effectively condition qubits.

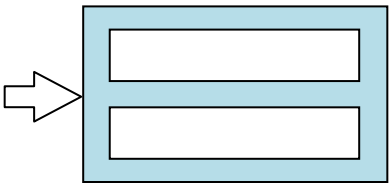
Operations: Rotate, Hadamard, and General 2x2 Matrix

Any quantum operator which can be expressed as a 2x2 complex matrix (that's a common form in the QC community) can be done by taking matched value pairs (as you would with NOT), but instead of exchanging them, create a 1x2 complex vector and multiply it by the matrix.

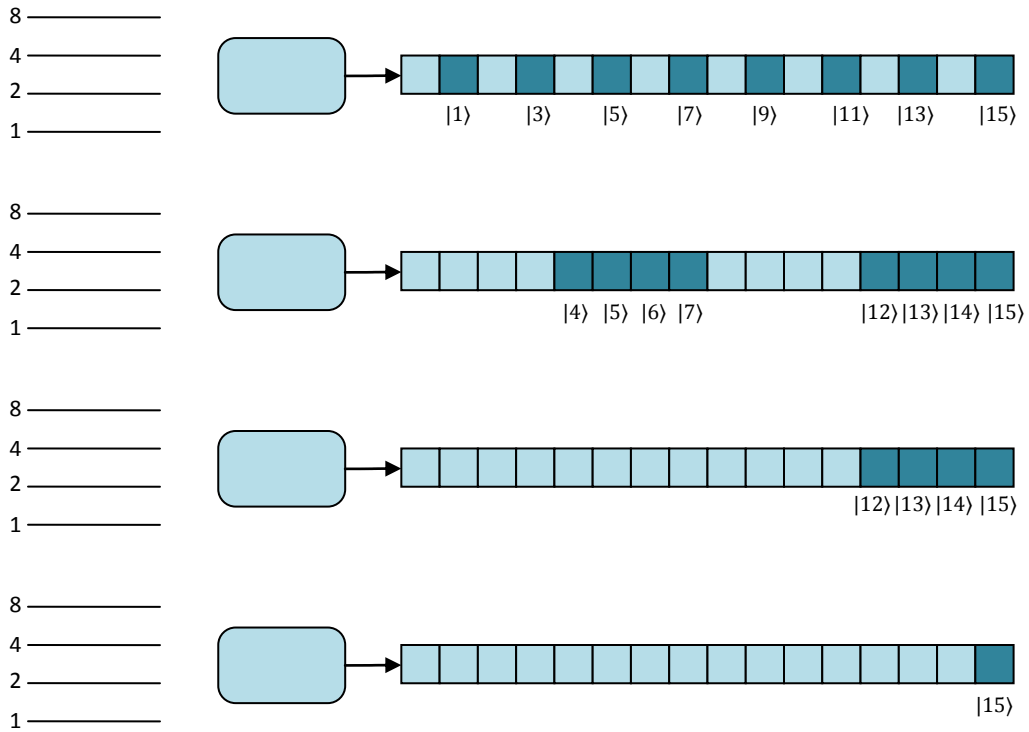
2x2 Matrix Gate implemented operating on pairs of value pairs



$$\begin{bmatrix} m_{0r} & m_{0i} & m_{1r} & m_{1i} \\ m_{2r} & m_{2r} & m_{3r} & m_{3r} \end{bmatrix} \begin{bmatrix} a_x \\ a_y \\ b_x \\ b_y \end{bmatrix}$$



(this diagram is under construction)



For the conditional version of these, only perform the operations on values which have the condition bits set.

Operations: Read and Write [pic of symbol]

To read a qubit, we need to have our simulator determine the probability of the qubit, then randomly choose a value, and then collapse the QReg into the chosen state.

To determine the probability for a qubit, we just need the squared length of all value pairs where the target bit is 1. This work can be distributed just like everything else we've done so far. The only difference is that now we actually need a result to be returned.

Collapsing the qubit is a two-step process. First, we go through each QBlock (in parallel if possible) and zero out every value pair in which the target bit disagrees with the result. For example, if the target bit is 1 (the lowest order bit), and the result of the read is 0, then all odd values in our QReg get set to zero.

Then, we re-normalize the remaining vector, so that its length is 1.

[diagram]

Note that the normalization isn't *strictly* necessary. That is to say, if we switch to using a probability-peek implementation which works for any length vector, then the rest of the simulation won't ever know or care whether the vector length is actually 1.0. However, if we don't normalize, then each time we read, our numbers will get smaller and smaller (possibly orders of magnitude per read), and the precision loss is likely to become a problem.

Also note that we can speed up out normalization by using the fact that we already know (from the probability result during the read) the squared length of the vector we're normalizing.

Once reading works, writing is easy. To write a value into a qubit, first read the qubit, and if your result doesn't match what you want, perform a NOT on the qubit. That's it.

The reason to go further

At this point, we've got a functioning QC simulation. It can perform any operation we want within the limitations of our available memory and compute time.

There are a few disadvantages to this simplistic arrangement. First of all, we need a single contiguous block large enough to store the whole simulation.

Second, if we're going to use multiple cores, they all need access to the same storage or memory.

Third, the time it takes to do the NOT and CNOT operations (probably the most common ones we'll do) always requires moving every single affected value pair from one memory location to another. That will be a lot of memory thrashing, so we'd like to improve it, at least in some cases.

Sample Case 2: Two Blocks and a Node

For the next step, we'll cut our QBlock in half, and place each of our new smaller blocks under a QNode. We can draw a dotted line on the qubit staff to show how many of the qubits each block effectively contains.

[diagram]

For any operations entirely below the dotted line, the work we need to do is exactly the same as it was before, except that since the work is all intra-block, and the blocks don't depend on one another. Because of this, we can process each block in parallel, even on separate computers if necessary. The two blocks can be processed in any order, or simultaneously. Already, we've gained a multiprocessing advantage.

Performance of NOT Gates

[diagram of below-the-line operations]

Now for something interesting. Consider a NOT tone on the qubit above the line. We *could* copy both blocks entirely over (we might need a lot of temp storage), but we can just as easily just swap the two pointers of the QNode.

[diagram of pointer swapping]

Suddenly, performing a NOT on that qubit takes negligible time, *no matter how many qubits there are below the line*. F we're running a QC program on this simulator, and it's going to do a lot of NOT operations on a single qubit, we can speed it up tremendously by using our above-the-line qubit.

Side note: Having a speed difference from one qubit to the next *may* be something we encounter on real QC's, for physical implementation reasons. With this in mind, the compilers should be able to gain performance benefits by taking qubit speed into account while optimizing.

Performance of CNOT Gates

In the case where a condition qubit is above the dotted line, we only need to operate on one of the QBlocks.

[diagram of]

When the target qubit is above the line, we actually need to do a cross-block operation. In this case, we're always guaranteed that the value pairs swap straight across, getting matched up with the corresponding value pair of the other block.

[diagram of cross-block operations]

Performance of Other Gates

Unfortunately, no other gates benefit from the super-optimization of that above-the-line NOT, but in all cases, the operations are broken neatly into independent intra-block or cross-block operations, which can be done in any order, on any computer.

Scheduling Parallel Operations

One thing to be aware of here is that if the blocks *are* located in separate memory spaces, then there will be some need to transfer data between computers in order to complete cross-block operations. Assuming we're going to run more than one instruction through our QC simulator, the transfers can be scheduled in advance.

Dependency and Results

Operations on a block are only ever dependent on other operations on that same block, which helps with scheduling. The block operations can by computed by any available means, so using a GPU to accelerate QC simulation becomes as simple as implementing some intra-block and/or cross-block operations in a shader, and letting the CPU handle the QNode level.

The result is not needed until the next time that block is read.

Sample Case 3: Multi-level QNode Tree

The last case to look at here just involves putting more qubits above the dotted line by making our QNode tree deeper. The only thing which really changes now is that we have more blocks, a bit of complex block pairing, and the possibility of having both target and condition qubits above the dotted line.

[diagram of some cases]

Tuning the System

The size of the QBlocks will have a substantial effect on performance, but the best block size will probably depend not only on the system you're running on, but also the actual programs you're running.

Larger QBlocks means more cache-coherent operation and less pointer overhead, but also a more constant operation time and fewer opportunities for parallelism.

Taking it Too Far

Since we gain advantages from splitting up our blocks, why not go all the way, having just two value pairs (or even one) in a block? Clearly, as the QNode tree grows, it loses its advantages and becomes just a big QBlock shattered all over memory.