Machine Level Optimization for Actual and Simulated Quantum Computers

Draft 1 (February 28, 2012)

Fine Print: This document is not finished, but you're welcome to read it anyway. It has not been peer-reviewed, or even thoroughly researched. These are my own notes and speculations, with shiny pictures added. If you have questions, comments, or really good jokes, please contact me at ej@machinelevel.com

Overview

The purpose of this document is to explore issues of hardware-level optimization as they apply to physical quantum computers. At the time of writing (January 2012) QC's are still very simple, having just a few qubits which lose coherence fairly quickly. So now is a great time to explore this. While the focus is on physical QC's, some of the issues are also useful for QC simulators. My own simulator (eternally under construction) can be found at http://qc.machinelevel.com

Machine Level Optimization

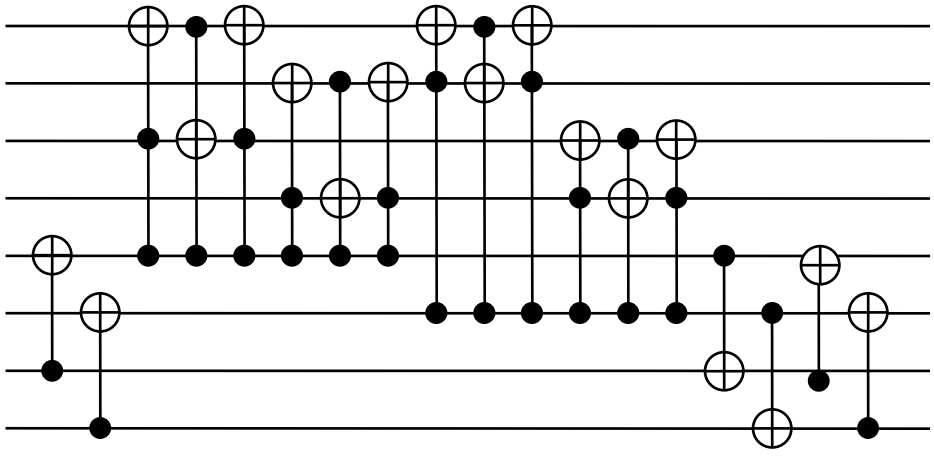
In conventional programming, the computer does a great deal of behind-the-scenes work to satisfy the desires of the programmer. For example, when the program says…

```
thisTurtle->speed += thrust[currentEngine];
```

…there's a lot of work done, fetching values from memory (or from a fast on-chip cache, or possibly a slow disk), dereferencing and fetching again, loading, adding and then storing out the answer. How many instructions are actually required to execute this line, and how much time do they take? Does this line of code cause function calls? If so, did the compiler inline them, or are we going to be fetching instructions from some other location? The best way to know for sure is to look at the assembly instructions. Most debuggers have a way to do this, and the process can reveal optimizations which are invisible at the source code level.

By being aware of how the physical machine differs from the logical ideal, the programmer can often speed up execution substantially, without changing functionality at all.

In QC programming, consider the following program: (This is the QQRAM indexer from another one of my whitepapers.)



There are 18 gates in this program. Assuming the QC performs one operation per clock cycle, it looks like it should take 18 cycles to execute this. Depending on the physical constraints of the machine, behind-the-scenes work might make it take much longer. There's also the possibility of building a machine which can execute it faster.

Why speed matters in QC programming

Besides the ultra-obvious desire to have a program run faster to save time, energy, and money, there's a special reason we want QC software to be efficient. Decoherence (the corruption of quantum states over time) is a major limitation to quantum computing in general. If the computation takes too long, you actually start losing data, and the computation will fail altogether. This might take much less than one second, depending on the physical computer. (As of Feb 28 2012, decoherence times are at about 100 microseconds. See IBM in the reference section at the end of this document.)

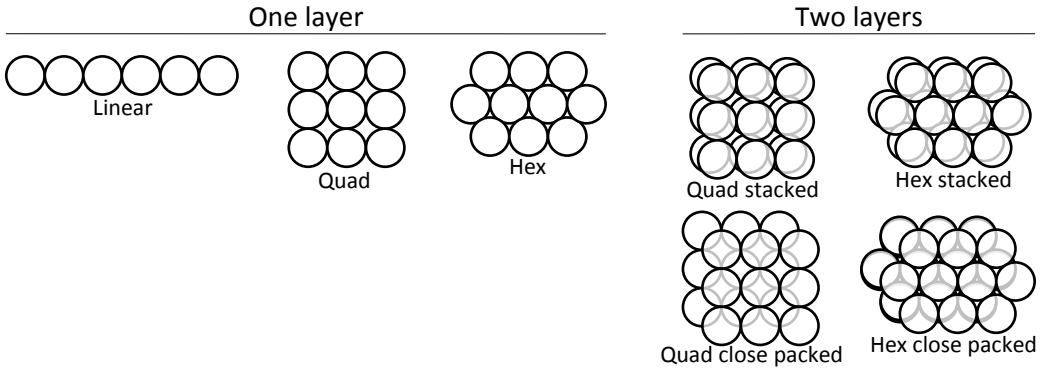
The classical analog would be something like this: imagine writing software on a computer which randomly started flipping bits in your variables a few milliseconds after they're initialized. In this case, a 2x speedup may make the difference between a program's success and failure.

Physical Constraints of QC's

It's hard to know exactly what physical limitations there will be in large-scale QC's, so **we're going to guess**.

Constraint #1: The qubits are arranged close together on a chip.

This seems reasonable, given modern chip fabrication techniques and QC solutions. There's some physical layout to the qubits, and we can't "run wires" to connect them. They must be physically close together, but any layout will do. Even company logos and animal shapes will work, though these may not achieve optimal efficiency. For the purpose of this exploration, we'll stick to a few arrangements which are likely to be common.

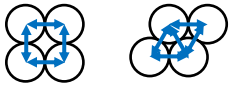


In a simple chip, the qubits may be all in a line. For reasons we'll see soon, arranging them into a 2D plane will add efficiency. Once the technology is sufficient to stack two planes, we can gain advantage from that as well.

We could of course add even more layers, in which case 'Quad close packed' and 'Hex close packed' become the same thing, in different orientations. This presents problems with Constraint #2 below, so we're going to stick to two planes for now, and note that if we *could* do it, we'd gain speed yet again.

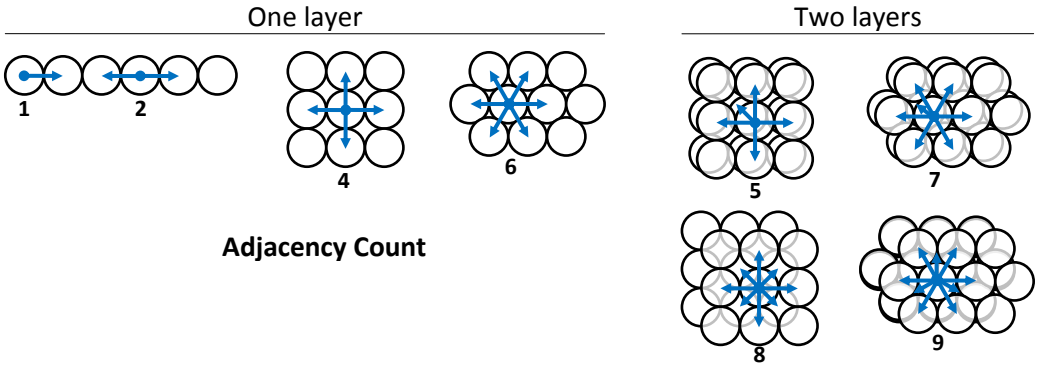
Some terminology becomes useful at this point.

Adjacency – Two physical qubits are considered adjacent if their spheres **actually touch** in the diagram.

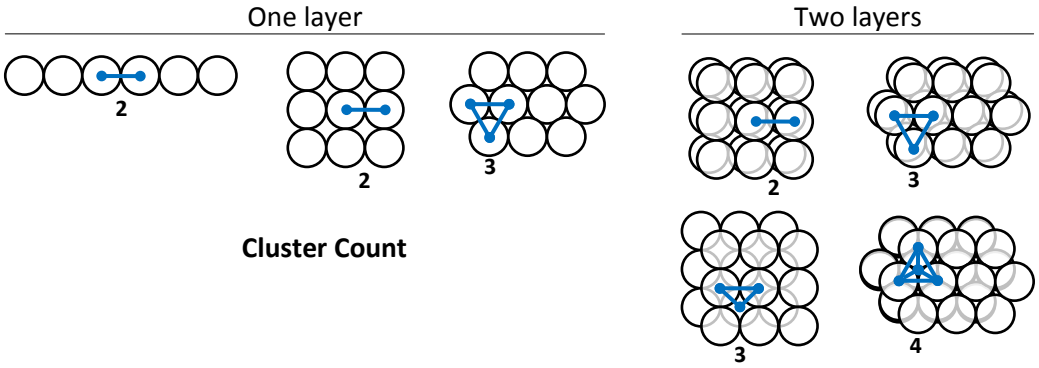


Adjacent qubits

Adjacency Count – The number of qubits adjacent to a given physical qubit.



Cluster Count – The maximum number of qubits which are all adjacent to each other.

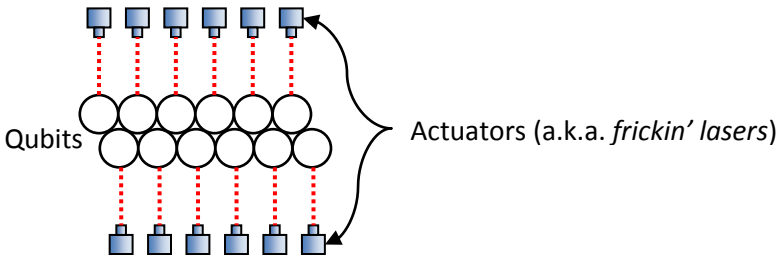


If it becomes important to have a high cluster count, we've got a few options. If we could get ultra-spiffy and add more than two layers, we could get our adjacency count up to 12, but, our cluster count still tops out at 4.

With those terms covered, we can move on to our next constraint.

Constraint #2: All qubits must have line-of-sight to an actuator (laser, electrode, or some other device for reading and writing).

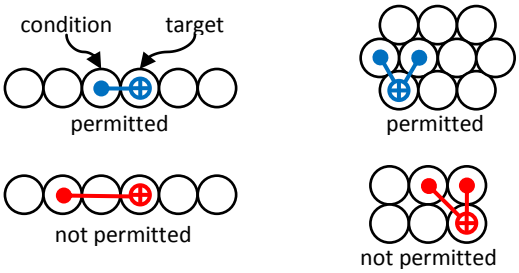
For reading and manipulating qubits, we've got to have some physical way to access them. This won't affect the programmer directly, but it certainly affects our choice of geometry when it comes to three-dimensional chip layouts. That's why we're sticking with two-layer arrangements, where a set of laser diodes on each side provides easy access.



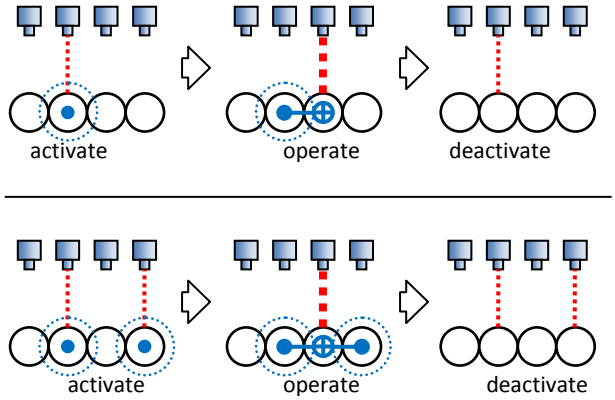
One could imagine a four-layer close packed setup where lasers aim through the little gaps in the top layer, we'll skip that case for now.

Constraint #3: An active **condition** qubit will affect all active adjacent **target** qubits, and no others.

Aha, this one is huge. Here we're taking an educated guess that in order to do a controlled-not (for example), the control qubit must be *physically next to* the target qubit.



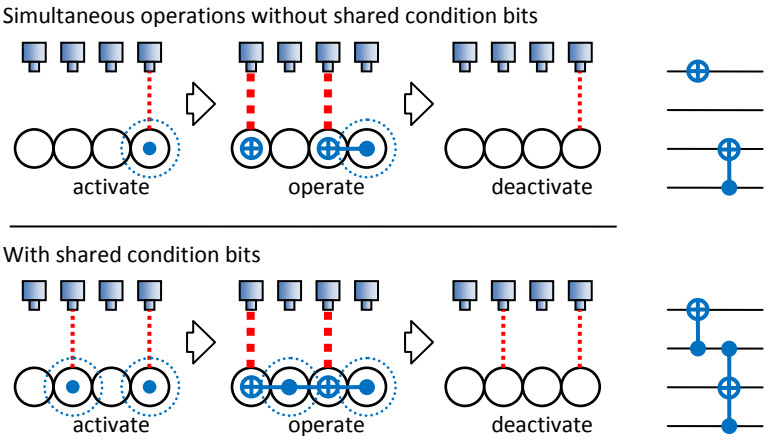
This constraint arises from the way that some (not all) physical quantum computers are implemented. It's a serious enough limitation that it's worth assuming for this machine. The typical process of performing an operation is to hit the condition qubit with a weak laser pulse to activate it, then hit the target qubit with a strong pulse to perform the operation, followed by a weak-pulse to deactivate the condition qubit. Basically what it says is that during an operation, any condition qubits are 'activated' and nearby operations are affected by them.



This puts some limitations on operations a QC can physically perform. The linear arrangement, for example, cannot support a native exchange at all, because that operation uses two target qubits. That doesn't mean the linear QC can't perform an exchange; it just takes longer because it's performed using three CNOT gates.

Constraint #4: Operations on separate target qubits may happen simultaneously.

That's a nice one (more of an ability than a constraint), and it makes sense. If two operations don't involve the same target qubits, we can do them both at the same time. This assumes that we have separate actuators (lasers/electrodes) per qubit, which is likely.

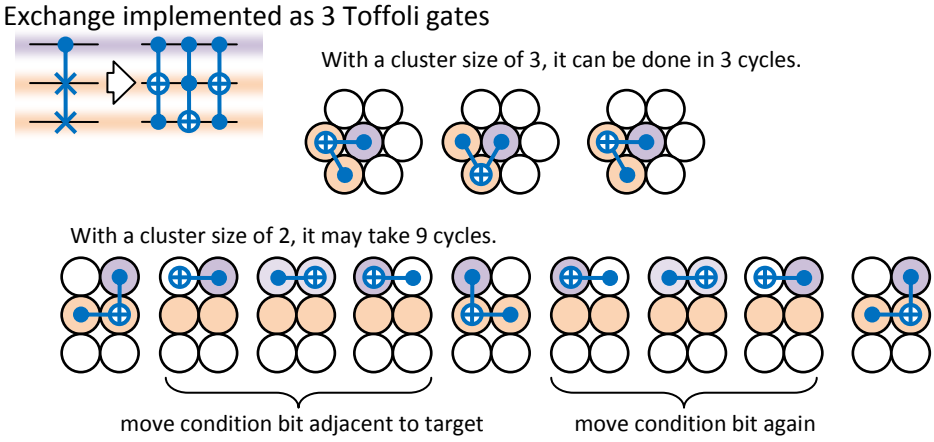


Some machines may not support simultaneous operations with shared condition qubits, but we'll assume ours does, because it's plausible and it makes the optimization task more interesting.

Constraint #5: Every QC has a set of 'native gates' which it uses to implement all other gates.

It's unlikely that any QC will be able to implement all of the different operations programmers want to use, so they'll construct them from whatever basic set is actually implemented in hardware. For example, a QC might have just CNOT and Rotate, or it may be able to do a Toffoli (CCNOT) or an Fredkin (exchange) gate.

Another possibility is that it supports exchange, but actually implements it on-chip as three Toffoli gates, causing the operation to take three times as long as a similar machine which has an actual exchange operation. In this case, the cluster count becomes very important.



This is our first real glimpse of the "extra behind-the-scenes work" the QC might need to do to run a program. None of this affects the programmer's ability to write software, but it will have a significant effect on speed.

Constraint #6: Any QC or simulator may have complex speed rules.

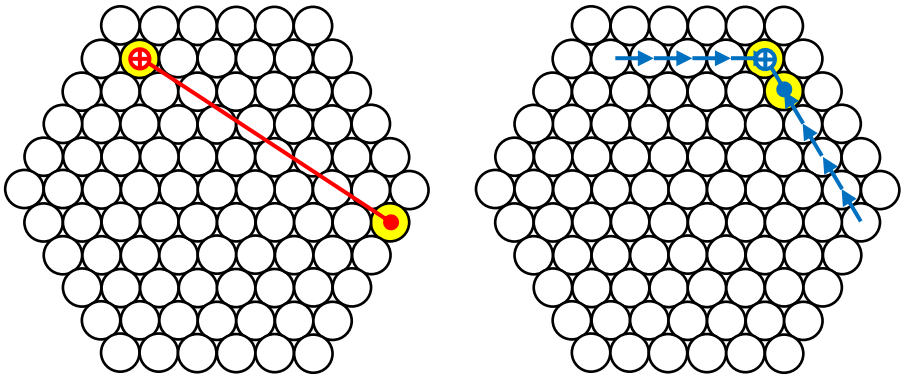
For a simulator, the speed of an operation may depend on parallelization characteristics of the simulator, which are likely to be different for different qubits. See the Model for Efficient Simulation paper for details on one possible set of characteristics.

On a real QC, some operations may have a “settle” time, which may depend on the number of qubits in the operation, or even the noise level or temperature in that area of the QC chip. These complex speed attributes will certainly vary from one QC model to another, but may even be dynamic on a single machine, changing with the machine’s environment.

One Spiffy Trick

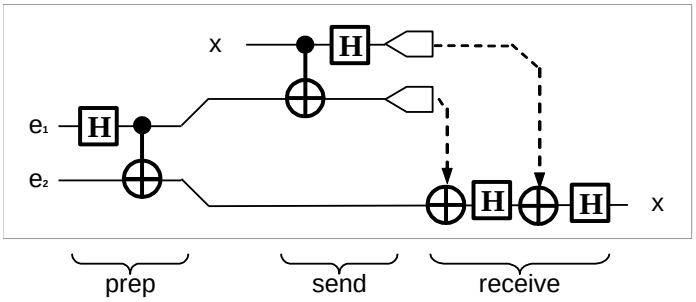
Many of the constraints mentioned so far involve adjacency. In addition, it has been noted that some qubits or groups of qubits might be faster than others. Just as classical computers spent much of their time waiting for memory fetches, one could easily imagine QC’s waiting for thousands of Fredkin operations to shuffle the right qubits together.

To do this operation, these qubits need to be exchange-moved a long way.



Even with a great path finding algorithm, it may be many cycles before the qubits are adjacent to one another. A very clever compiler might be able to arrange the operations to minimize the travel waste, but sometimes it will be unavoidable. Even worse, if the machine you’re using has no native exchange operation, you’ll need three gate cycles per swap.

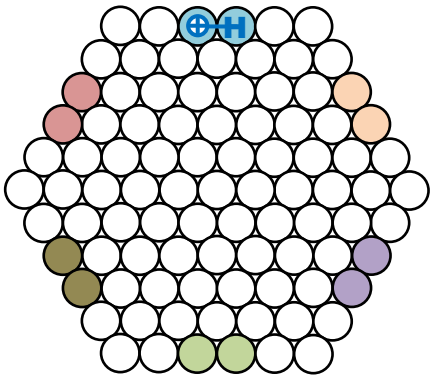
If only there were some way we could simply teleport one qubit over to the other. Hey wait, teleportation is actually something QC’s are good at! See the teleportation section for details, but the action is fairly simple.



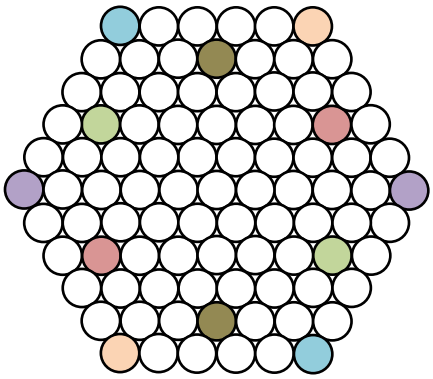
On-chip teleportation? That’s handy.

First we need an entangled pair of qubits, already in position. We can have the QC generate those using unused qubits as the rest of the computation is going on. In fact, it will probably be doing this constantly.

Create a bunch of entangled pairs.

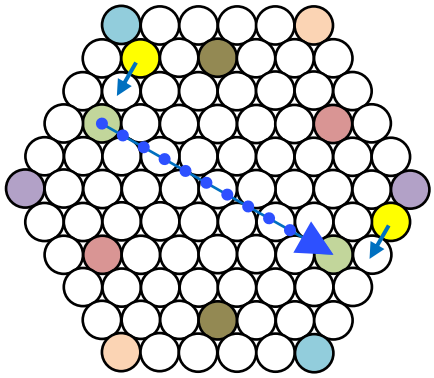


Get them into position.

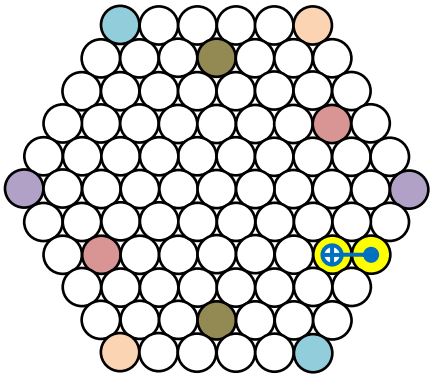


This is looking like a board game, and that's appropriate. When a qubit is ready to travel a long distance, all it needs to do is travel to (or bring to it) the most convenient teleport pair.

Get to the teleporter, beam over….



And… ready to operate.



…and bang! Suddenly distance is no problem. We've done an on-chip teleportation, resulting in a good optimization for our QC software. That entangled pair has been used up, but we'll be able to make one to replace it.

Conclusion

Machine level software optimization for QC's will be essential, just as it is for conventional computers. Awareness of the geometry and characteristics of the available qubits will allow a compiler or programmer to gain the advantage of substantially faster code.

Useful References

- Minds, Machines, and the Multiverse, by Julian Brown
- Elementary Gates for Quantum Computation (1995) arXiv:quant-ph/9503016v1
- Quantum Computing for Computer Scientists (Yanofsky and Mannucci), 2008
- IBM QC advances 2/28/2012:
    - http://ibmquantumcomputing.tumblr.com/
    - http://www.nytimes.com/2012/02/28/technology/ibm-inch-closer-on-quantum-computer.html
- [TODO: add other papers and books I've found useful]

About the Author

EJ and his muse live in a secret laboratory in San Francisco. Everything else you really need to know is either posted here, or can be obtained by sending an email to ej@machinelevel.com.