Statevector Based Quantum Circuit Simulator

Michael Tauraso
University of Washington
(Dated: December 9, 2022)

A state based quantum computer simulator written in the Rust programming language for windows computers, which takes quantum circuit input written in OpenQASM 2.0. It supports all quantum behavior defined in the specification, including the full scope of expressable quantum/classical interaction.

I. INTRODUCTION

This project arose out of a desire to have a desktop calculator of sorts for analyzing the correctness of circuits on homework assignments, as well as a platform for exploring the various ways that error can enter quantum circuits. After exploring several online quantum circuit simulators, and seeing their limitations I became interested in how difficult it was to construct one, and what sorts of trade-offs I might encounter in the attempt.

II. TECHNOLOGY CHOICE

Quantum simulations are usually limited by the amount of memory available to store amplitude information. In order to be able to have some direct contact with this issue, I decided to use Rust as the implementation language. Rust is similar to other systems programming languages like C/C++, and Fortran in that it allows the programmer to have some very direct access to the usage of memory, and supports a strong type system. The downside of using most systems languagesis that in getting greater control over what the computer is doing, you give up the ability to program higher level concepts without writing a lot of code. Systems languges often make it easy to introduce bugs that are difficult to track down, or only occur in a small number of runtime scenarios. Rust has several features that address these issues, borrowed from functional programming languages like SML/NJ, OCAML, and Haskell. Among these features that were useful on this project: Type inference saved many keystrokes, and the borrow checker, and memory safety types saved many headaches when manipulating n-dimensional arrays.

Since the focus of this project is simulating quantum computation, I decided to use libraries to perform steps like parsing and translation of the OpenQASM language, Linear algebra/Tensor operations, User interface, and error handling. With the exception of the parsing and translation library for OpenQASM, all of these libraries are general purpose utilities without any particular focus on quantum computing.

A excellent and readable summary of the OpenQASM language can be found in the language spec/citecrossOpen2017. The 2.0 version of OpenQASM has minimal features of traditional programming languages; however, it offers a machine readable way to implement any quantum circuit. The language requires the underlying quantum hardware to be able to perform two gates: a controlled not, and a unitary rotation gate defined in the paper. It allows more complex gates or circuits to be defined in terms of those basic gates in a recursive manner. It also supports expressing non-deferred measurement and classical control of quantum operations. Since 2017 OpenQASM has been extended to a 3.0 version that includes support for many more classical programming control flow constructs. The grammar of OpenQASM 3.0 is still in flux, and represents a much larger implementation target; however, the base quantum gates are the same as 2.0[1]. Given that the focus is on quantum circuits, only OpenQASM 2.0 is imlemented.

III. PARSING AND TRANSLATION

IV. QUANTUM BITS

The approach taken for this simulator was relatively simple in the sense that it is unsophisticated, and one of the more naieve possible approaches. The program keeps record of every possible complex amplitude, so the space complexity is $\mathcal{O}(2^n)$ where n is the total number of bits across all distinct registers defined in the quantum circuit. Classical bits are included in this count, which allows a straightforward implementation of remixing as we will see.

The usual manner of working out a quantum circuit by hand uses Dirac notation to define finite-dimensional state vectors and matrix transitions in the Hilbert space of the quantum computer. This simulator represents the same amplitudes as tensor components more in the manner that General Relativity is exposited. The amplitudes that describe the quantum computer's state are a rank-n tensor with complex-valued components where each index of the tensor can be 0 or 1[2]. Each index of the tensor corresponds to a quantum bit. Each quantum operation is a tensor contraction, which sums across the amplitudes associated with each quantum bit involved in the operation. In this paper I will be following the convention that ket vectors correspond to a lower tensor index, and bra vectors correspond to an upper index. I will also be using greek letters $\alpha, \beta, ..$ for quantum bits, and latin letters $u, v, ...$ for classical bits.

The Hadamard gate is traditionally represented as the matrix $\frac{1}{\sqrt{2}}\left(\begin{smallmatrix}1 & 1\\ 1 & -1\end{smallmatrix}\right)$, which is equivalently written in dirac notation as $\frac{1}{\sqrt{2}}\big(|0\rangle\langle 0| + |0\rangle\langle 1| + |1\rangle\langle 0| - |1\rangle\langle 1|\big)$ In the tensor notation, the hadamard would be ${H^\alpha}_\beta$ where ${H^0}_0 = {H^1}_0 = {H^0}_1 = \frac{1}{\sqrt{2}}$ and ${H^1}_1 = -\frac{1}{\sqrt{2}}$. Likewise a two qubit quantum computer with the first bit initialized to one, would have the amplitude matrix $A_{\alpha\beta}$ where $A_{10} = 1$ and all other components are zero, corresponding to the state $\Psi = |10\rangle$ in Dirac notation with the implicit tensor product. In order to compute a hadamard of the first qubit, the simulator contracts the lower index of the qubit (ket vectors) with the upper index of the Hadamard tensor (bra vectors). Using the einstein summation convention, the new amplitudes are

$$A'_{\alpha\beta} = A_{\gamma\beta}{H^\gamma}_\alpha.$$

This new set of amplitudes has two lower indicies, α and β which correspond to the new amplitudes for the two qubits after the Hadamard. The only non-zero components of A' are $A'_{00} = \frac{1}{\sqrt{2}}$ and $A'_{10} = -\frac{1}{\sqrt{2}}$, which correspond to the dirac state $\Psi' = \frac{1}{\sqrt{2}}(|00\rangle - |10\rangle)$, which is exactly what we would expect.

This formalism works similarly for multiple qubit gates. A CNOT gate can be represented by the tensor ${C^{\alpha\beta}}_{\gamma\delta}$ where ${C^{00}}_{00} = {C^{01}}_{01} = {C^{10}}_{11} = {C^{11}}_{10} = 1$ and all other components are zero. In this exposition it tempting to read the upper indicies as the "input" bit patterns, and the lower indicies as the "output" bit pattern; however, this only works because CNOT does not mix our basis states. The order of contraction of the CNOT tensor with our amplitudes controls which bit is the control bit and which is the target bit. We can take the prime state above and evolve it further by applying a CNOT targetting the second qubit with the first qubit as control. The resulting amplitudes can be written

$$A''_{\alpha\beta} = A'_{\gamma\delta}{C^{\gamma\delta}}_{\alpha\beta}.$$

The only non-zero components are $A''_{00} = \frac{1}{\sqrt{2}}$ and $A''_{11} = -\frac{1}{\sqrt{2}}$, which corresponds to the bell state we would expect: $\Psi'' = \frac{1}{\sqrt{2}}(|00\rangle - |11\rangle)$. Note that reversing the order of the γ and δ indicies one one tensor in the sum above results in no change to the amplitudes, which is what we would expect if the control and target bits of the CNOT gate were reversed.

This method is easy to extend mechanically to many more qubits, as is needed to accomodate large circuits. In the quantum simulator `ndarray` and `einsum` packages are used to perform these tensor contractions, and the examples above follow how the two builtin gates `CX` and `U` are implemented in the simulation. The functions `apply_u`, and `apply_cx` in `src/register.rs` implement the mathematics of these gates.

This formalism can also be extended to perform both measurement and the remixing of classical bits back in to quantum bits, without falling back to a sampling approach for measurement.

V. MEASUREMENT

When qubits are measured, there are many potential things that can mean in a simulation context. The desired outcome may be probabilities for various bit patterns, post-measurement selection of certain bit patterns. The circuit may go on to use of a measured bit to control a quantum operation. As we have seen on the homework, classical control of quantum gates, can replicate a quantum gate given certain input states. OpenQASM even supports doing multiple measurements into the same classical bit, where the earlier measurements are no longer present in the circuit at the end of execution time. For the sake of modularity, it is desirable to both separate the evolution of the quantum circuit and measurement from the format that the measurement is surfaced in the interface. It is also desirable to have an evolution algorithm that can handle even the more complex circuit cases.

In order to achieve these objectives, measurement and classical bits are implemented in the same tensor formalism used for quantum operations. Each tensor of amplitudes is extended with additional indicies corresponding to the value of classical bit registers defined in the OpenQASM input file. The OpenQASM `measure` statement can occur at any point in the program, and it measures a quantum bit on to a classical bit. In the simulator we begin by projecting the full amplitude state onto each of the basis states for the qubit we are measuring, and summing over the values of the cbit we are overwriting. This yields two tensors with rank one less than the total number of bits in the computer (classical and quantum). Each of these tensors is then set equal to the slice of amplitudes where the classical bit is the corresponding value.

Returning to the example from Section IV and dropping the primes, consider the state $\Psi = \frac{1}{\sqrt{2}}(|00\rangle - |11\rangle)$ which will be represented by the tensor $A_{u\alpha\beta}$. The u index is a classical bit set equal to zero, such that the amplitude components are all zero except for $A_{000} = \frac{1}{\sqrt{2}}$ and $A_{011} = -\frac{1}{\sqrt{2}}$. In order to measure the first qubit, we'll start by projecting across the zero and one basis states. ${P_{(0)}{}^\alpha}_\beta$ and ${P_{(1)}{}^\alpha}_\beta$ will be the respective projection tensors, such that the only nonzero components are ${P_{(0)}{}^0}_0 = {P_{(1)}{}^1}_1 = 1$. The two projected amplitudes will also be summed across the possible initial values of the classical bit:

$$A_{(0)\alpha\beta} = \big(A_{0\gamma\beta} + A_{1\gamma\beta}\big){P_{(0)}{}^\gamma}_\alpha$$
$$A_{(1)\alpha\beta} = \big(A_{0\gamma\beta} + A_{1\gamma\beta}\big){P_{(1)}{}^\gamma}_\alpha$$

The additional summing over the two values of the classical bit is necessary in the case where the classical bit was measured previously. Since OpenQASM does not specify selection rules base on measurement, when a cbit is overwritten by quantum information, we need to take into account both possible qubit states that could lead to that outcome.

The evolved amplitude tensor is constructed by assigning the projected amplitudes to the relevant slices such that ${A'}_{0\alpha\beta} = A_{(0)\alpha\beta}$ and ${A'}_{1\alpha\beta} = A_{(1)\alpha\beta}$. After the measurement, the overall amplitude tensor has the same meaning it had before: ${A'}_{0\alpha\beta}$ are the amplitudes of the qubits α and β in the case where the classical bit u was measured to be $|0\rangle$, and likewise for ${A'}_{1\alpha\beta}$.

These states retain their normalization across all possible cbit values, as if they were also qubits. The simulator uses these post measurement amplitudes to answer probability questions. After the measurement ${A'}_{0\alpha\beta}$ is the projection where the α qubit is in state $|0\rangle$. The probability of this outcome is simply the squared norm ${A'}_{0\alpha\beta}{A'}^{0\alpha\beta} = \frac{1}{2}$. Note that following the convention for mapping upper and lower indicies to dirac notation, ${A'}^{u\alpha\beta} = ({A'}_{u\alpha\beta})^*$. Therefore the contraction above will always be real.

Because all probabilities computed this way are normalized to cover all possible outcomes from running the circuit, conditional probabilities can be assembled from them by dividing the probability representing the outcome of interest by the probability for the condition.

The code that performs the amplitude updates associated with measurement is in the `apply_measure` function in `src/register.rs`, and the computation of squared norms is performed in the `probability` and `norm_sqr` functions of that same file.

VI. CLASSICAL CONTROL

[1] A. W. Cross, L. S. Bishop, J. A. Smolin, and J. M. Gambetta, OpenQASM 3.x Live Specification, https://openqasm.com/index.html.

[2] J. Kattemölle, Quantum Circuits in Python using nothing but Numpy, https://www.kattemolle.com/other/QCinPY.html.

[3] A. W. Cross, L. S. Bishop, J. A. Smolin, and J. M. Gambetta, Open Quantum Assembly Language (2017), arXiv:1707.03429 [quant-ph].