State Vector Based Quantum Circuit Simulator

Michael Tauraso
University of Washington
(Dated: December 13, 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. The Quantum Simulator supports all behavior defined in the OpenQASM 2.0 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. The quantum simulator itself is available at `https://mtauraso.github.io/QuantumSimulator/`, and is a windows command line application that implements the OpenQASM 2.0 language.

This paper starts with an overview the technology the simulator is built with in sections II and III. Then the mathematical formalism that is used internally will be introduced in section IV, followed by examples of how quantum gates are implemented in section V. Measurement and classical control of quantum information are covered in sections VI and VII respectively. In sections describing the implementation of the simulation algorithm, there are specific references to relevent sections of the program code. Section VIII concludes with some thoughts on extending the program, none of which have been implemented.

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 languages is 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 languages also 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, and OCAML. Ultimately these language features made it easier to know the program was working correctly, but didn't necessarily speed up development.

One language feature that was particularly useful was the integrated testing in rust. `src/main.rs` has several sample programs which are run in batch mode with the `cargo test` command. These tests allowed quick checking that no existing functionality had broken as the program grew in size and complexity.

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, command line 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[1]. 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.

OpenQASM requires the underlying quantum hardware to be able to perform two gates: a controlled not, and a unitary rotation gate defined in the paper. OpenQASM 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 OpenQASM 2.0[2]. Given that the focus is on quantum circuits, only OpenQASM 2.0 is implemented in the simulator.

III. PARSING AND TRANSLATION

When the simulator opens a circuit file, the raw OpenQASM code is first parsed and translated by the OpenQASM parser library. This library provides hooks for several types of program syntax errors, and handles things like the inclusion of the quantum experience header which defines many useful gates in OpenQASM in terms of the basic U and CX gates that are built into the language.

Once the circuit file has been parsed, the library offers two ways to programmatically access the parsed file. There is a low-level interface which allows access to the syntactic structure of the file, and a high level linearizer interface. The linearizer allows a program using the library to traverse the OpenQASM program recursively, accessing first the declarations of all quantum registers, and then the basic U and CX gates in order along with the other basic statements.

All of the simulation evolution and setup logic is invoked by the recursive enumeration done by the parser. The `QuantumRegisterGateWriter` object defined in `src/register.rs` recieves calls from this recursive enumeration. Recursive enumeration is initiated in `openqasm_run_program` function in `src/main.rs`. Errors from the parsing library, and calls to parse and check the program are handled in the `openqasm_parse_*` and `openqasm_check_program` functions.

IV. QUANTUM BITS

The approach taken for this simulator was relatively simple in the sense that it is unsophisticated, and one of the more naive possible approaches. The program keeps record of every possible complex amplitude, so the memory usage 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 classically controlled quantum gates.

The usual manner of working out a quantum circuit by hand uses Dirac notation to define 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 usually exposited.

The amplitudes that describe the quantum computer's state are a rank-n tensor with complex-valued components. Each index of the tensor can be 0 or 1 and corresponds to a single quantum bit[3]. Each quantum operation is a tensor contraction, which sums across the amplitudes associated with each quantum bit involved. 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. If a tensor is written with upper indicies, it is implied that the components of the upper index version are the complex conjugate of the corresponding components in the lower indexed version.

V. QUANTUM GATES

The Hadamard gate is usually 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 $H_0{}^0 = H_0{}^1 = H_1{}^0 = \frac{1}{\sqrt{2}}$ 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.

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

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

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 $C_{\gamma\delta}{}^{\alpha\beta}$ where $C_{00}{}^{00} = C_{01}{}^{01} = C_{11}{}^{10} = C_{10}{}^{11} = 1$ 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

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

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 [4].

This formalism can also be extended to perform both measurement and the classical control of quantum gates, without falling back to a sampling approach for measurement. It is also possible to at any point generate the density matrix for the system by computing $\rho_{\alpha\beta}{}^{\gamma\delta} = A_{\alpha\beta}A^{\gamma\delta}$. In this way the program is essentially manipulating half of a density matrix on each operation. Note that following the convention for mapping upper and lower indicies to dirac notation, $A^{\alpha\beta} = (A_{\alpha\beta})^*$

VI. 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 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. The simulator begins to evalute the `measure` statement by projecting the full amplitude state onto each of the basis states for the qubit we are measuring. Then the simulation uses these projections to assemble a tensor that represents the amplitudes after a measurement.

Returning to the example from Section IV and dropping the primes, consider the state $\Psi = \frac{1}{\sqrt{2}}(|00\rangle - |11\rangle)$, represented by the tensor $A_{u\alpha\beta}$. The u index is a classical bit set equal to zero, such that all tensor components are zero except $A_{000} = \frac{1}{\sqrt{2}}$ and $A_{011} = -\frac{1}{\sqrt{2}}$. In order to measure the first qubit, the simulation starts by projecting across the zero and one basis states.

${P^{(0)}}_{\beta}{}^{\alpha}$ and ${P^{(1)}}_{\beta}{}^{\alpha}$ will be the respective projection tensors, such that the only nonzero components are ${P^{(0)}}_{0}{}^{0} = {P^{(1)}}_{1}{}^{1} = 1$. Since the classical bit is zero at the beginning of the operation we only need consider amplitude components where $u = 0$:

$${A^{(0)}}_{\alpha\beta} = {P^{(0)}}_{\alpha}{}^{\gamma} A_{0\gamma\beta} \quad , \quad {A^{(1)}}_{\alpha\beta} = {P^{(1)}}_{\alpha}{}^{\gamma} A_{0\gamma\beta}$$

The classical bit may however have been the result of some quantum measurement. Had the classical bit u not been equal to zero, the simulation would simply swap the classical bit index u with a newly created classical bit index. This newly created classical bit index is defined to only have nonzero amplitudes on the zero valued components. When calculating measurement probabilities, this "hidden bit" gives us another index to trace over; however, it preserves quantum state that may be entangled with earlier measurements. In the simulator, the `hide_bit` function in `src/register.rs` handles the swapping in of new zeroed bits as needed. Notably, OpenQASM's `reset` command is entirely handled by calling `hide_bit`.

After contending with any newly introduced bit indicies, 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 ${A'}_{0\alpha\beta}{A'}^{0\alpha\beta} = \frac{1}{2}$. Since the indicies match upper and lower, this tensor contraction will always be real.

Because 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.

VII. CLASSICAL CONTROL

OpenQASM supports a limited control flow functionality where a classical bit register can control a quantum operation via the `if` command. Within the tensor formalism this is achieved by taking the relevant classical indicies and holding them to be equal to the target value of the `if` statement, and then performing other quantum gates as before.

For example, consider the simulator evaluating the statement `if (u == 0) h q[0];` . This statement asks for the quantum bit `q[0]` to be acted on with a hadamard gate `h` only if the classical register `u` has the value zero. Returning to the tensor from before, the simulator takes the $A_{u\alpha\beta}$ tensor and saves it. Then the simulator makes $A_{0\alpha\beta}$ active for gate evolution, pinning the value of the u index to zero. After the gates specified in the condition statement complete, the simulation updates only the $u = 0$ components of the original $A_{u\alpha\beta}$ tensor, and uses that same tensor for future gate evolution.

VIII. POSSIBLE EXTENSIONS

Performance of this program would be vastly improved on large programs by two changes: Allocation of all memory at the start, and the use of a sparse representation. Allocating the required number of tensor indicies at the start of the program would require a pre-pass, because `measure` and `reset` operations can each introduce a new index to the amplitude tensor under certain conditions. Currently the allocation of new memory for indicies and copying of prior tensor components is done on an as-needed basis. Pre-allocating the correct amount of space would be significantly faster. Using a sparse matrix library to hold the amplitude tensor would also speed things up. Currently a large amount of compute time and allocated memory are keeping track of the number zero in various tensor components.

The simulator currently operates on complex amplitudes as pairs of 32-bit floating point numbers. This has sufficient precision for the algorithms included in the `/sample/` directory; however, much of the by-hand analysis of circuits is done algebraically. It would be interesting to replace this 32-bit floating point type with an expression type which could accomodate symbolic notation. Because of the design choice to avoid implicitly sampling during measurement, it is possible to carry through this symbolic representation to the end of the calculation. This would yield symbolic expressions for the probability values, similar to what one might calculate by hand.

Unfortunately there was not time on this project to implement the introduction of errors; however, this representation is fairly close to a density matrix formalism. The expositions of idealized quantum error in [5, 6] use the density matrix formalism and take an implicit trace over the part of the environment that is responsible for the noise. Keeping with the current formulation used by the simulator may result in too many indicies being added to the amplitude tensor due to various error operations as hidden bits. This perfusion of extra bits may make the program too slow or take up too much memory to be useful.

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

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

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

[4] There is a slight difference in the definition of the U gate between OpenQASM 2.0 and OpenQASM 3.0. Despite the rest of the program implementing OpenQASM 2.0, I chose to use OpenQASM 3.0's definition of the U gate, because it allowed the Hadamard gate in the simulator to be $\frac{1}{\sqrt{2}}\big(|0\rangle\langle 0| + |0\rangle\langle 1| + |1\rangle\langle 0| - |1\rangle\langle 1|\big)$, which made it easier to check the simulator against outside references.

[5] M. Nielsen and I. Chuang, *Quantum Computation and Quantum Information: 10th Anniversary Edition* (Cambridge University Press, 2010).

[6] M. Le Bellac, *A Short Introduction to Quantum Information and Quantum Computation* (Cambridge University Press, Cambridge, 2006).