QUANTUM CIRCUIT SIMULATOR

QUANTUM PROGRAMMING

Parth Shettiwar parthshettiwar@g.ucla.edu

March 13, 2022

Contents

ABSTRACT

As part of this homework, a quantum circuit simulator was created in Python language where the task was to generate the resulting state vector for any circuit of any number of qubits mainly equipped with 5 types of gates: `NOT, Hadamard,T, TDag, and Control NOT`. First the implementation details are discussed, followed by analysis where run times are observed for various QASM programs. Finally the outputs are compared with the inbuilt simulator in cirq.

1 Design and Evaluation

1.1 Low Level Design / Code Walkthrough:

The objective was to create a quantum circuit simulator which would generate the state vector given a circuit, written in a qasm program line by line as string.

Given a QASM file, the whole problem of generating the exact state vector was broken down into following tasks:

- Parse the QASM program, and understand the various semantics of it (number of qubits, gates etc)
- Creating the functions for each type of gate

The following flow of code was adopted to create the quantum circuit simulator:

- Read the QASM file
- Preprocess the whole string as part of file to first determine the exact number of qubits being used as part of circuit and create a global state vector initialised to 0. The state vector will be of size 2^n where n is the number of qubits determined.
- Get the exact sequence of gates applied and the qubits on which they are applied to understand the circuit.
- Call the gate functions appropriately depending on the sequence derived above and update the state vector.
- Return the state vector after all sequence of operations. Compare the state vector with cirq simulator output.

Now we look over all the above steps with code.

- **Simulate function**

```
def simulate(qasm_string):
  global curr
  curr = -1e10
  qasm_string = qasm_string.replace('\n', '') #removing the new line literals
  qasm_string = qasm_string.split(";") #splitting on ;

  for i in range(4,len(qasm_string)):
    qasm_string[i] = qasm_string[i].replace(","," ") #replacing the ,
    temp = qasm_string[i].split(" ") # splitting on spaces

    #fetching the number of qubits in program
    if(len(temp)>0):
      for j in range(len(temp)):
        if(len(temp[j])>0 and temp[j][0]=="q"):
          l = temp[j].index('[')
          h = temp[j].index(']')

          curr = max(int(temp[j][l+1:h]),curr)

  state = np.zeros(2**(curr+1),dtype=complex)
  state[0] = 1


  # iterating over all lines and calling the gate functions
  for i in range(4,len(qasm_string)-1):
    temp = qasm_string[i].split(" ")

    if(len(temp)>0):
      op = temp[0]
      l1 = temp[1].index('[')
      h1 = temp[1].index(']')
      m1 = int(temp[1][l1+1:h1])
      if(len(temp)>2):
        l2 = temp[2].index("[")
        h2 = temp[2].index("]")
        m2 = int(temp[2][l2+1:h2])

      if(op=="x"):
        state = Xgate(state,m1)
      if(op=="h"):
        state = Hgate(state,m1)
      if(op=="t"):
        state = Tgate(state,m1)
      if(op=="tdg"):
        state = TDaggate(state,m1)
      if(op=="cx"):
        state = CXgate(state,m2,m1)

# rounding off since cirq output is also rounded off
  state = np.round_(state, decimals = 3)
  return state
```

The simulate function takes in QASM string and outputs the state vector Following are the key steps as part of this function

- First process the QASM program. Remove the new line literals. Split on ";" character to get the list of sequence of gates applied.
- Now starting from 4th item in list (since first 4 lines are not required at all t generate the circuit), we first find the exact number of qubits used as part of our program. We simply iterate through the lines and do some further processing to read the indices where we get the open and closed brackets and read the integer enclosed in it to extract the qubit number. We do this for all lines and find the maximum qubit number used
- We create a state vector of number of qubits size
- We again iterate through the qasm program. Everytime we extract the gate function being applied, and on which all qubits. Then call the appropriate function for that gate and get the updated state vector. For CX gate we parse 2 qubits and pass them to the `CXgate` function.
- Return the final state vector

• **Gate functions** We will see implmenetation for one of the gates, specifically the X gate, the rest all gate implmentations are similar

```
def Xgate(state,m):
  n = int(np.log2(len(state))) #get the number of qubits
  new_state = np.zeros(len(state),dtype=complex) #create a new state complex vector
  for i in range(len(state)): #iterate over all non zero entries and do the necessary operation by
```

```
    if(state[i]!=0):
      temp = state[i]
      state[i] = 0
      val = bin(i)[2:].zfill(n)
      bin_val = np.asarray(list(map(int,val)))
      bin_val[m] = -bin_val[m]+1
      new_val = int("".join(map(str,bin_val)),2)
      new_state[new_val] += temp
  return new_state
```

The function takes the current state and the qubit m on which operation has to be applied. Folllowing is the flow:

- Get the number of qubits n by observing the state vector length
- Create new state vector of length original state and initialise it to 0.
- Iterate through all non-zero entries in the state vector. We iterate through such entries only because in the Weighted Ket notation, these entries will only have a non-zero coefficient. For 0 coefficient, we dont care as any operartion wont affect them.
- As we iterate through such entries, we first convert the iterating index to binary value (this is how mapping is done between the state vector representation and Weighted Ket notation). We then apply our operation of X gate, that is reversing the qubit.
- After reversing,we convert it back to decimal value and set that entry in new state vector to the same coefficient which we started with for.
- For all other gates, we adopt similar set of operations, with if else conditions.

• **Analysis** We finally discuss the analysis code, which was written to perform the analysis of the simulator. The following analysis was performed:

  - Time taken vs number of qubits for my code
  - Time taken vs number of qubits for cirq simulator
  - Time taken for each gate individually
  - time taken vs number of gates for `miller_11.qasm` program

  All the plots are saved in the results folder (folder will be created automatically when run the code)

1.2 Random Circuit Generation

In this function, I created a code to generate qasm circuits at random. The goal was to test the simulator in rigorous manner. There are 2 options provided as part of this random circuit generation. Either user can provide the exact number of qubits and number of gates the circuit should be created or he can just avoid stating them and the code will randomly create a circuit of random number of gates and qubits. An option of verbose is also put in case user wants to see the random qasm program created. At the end of circuit generation, we pass the qasm circuit to my simulator and cirq simulator and compare the output. Following is the code:

```
def random_circuit_generate():

  lis = []
  lis.append('OPENQASM 2.0;\n')
  lis.append('include "qelib1.inc";\n')
  lis.append('qreg q[16];\n')
  lis.append('creg c[16];\n')


  qub = np.random.randint(low = 2, high = 16, size=1)[0]
  num_gates = np.random.randint(low = qub, high = 1000, size=1)[0]
```

```
if(len(sys.argv)>3):
  qub = int(sys.argv[3])
  num_gates = int(sys.argv[4])
print("Generaing circuit with ",qub,"qubits and",num_gates," gates in total")
print("\n")

dic = {0:"h",1:"x",2:"t",3:"tdg",4:"cx"}

for i in range(qub):
  gate = np.random.randint(low = 0, high = 5, size=1)[0]
  if(gate==4):
    qub_arr = np.arange(qub)
    qub_arr = np.delete(qub_arr, i)
    qub_num = np.random.choice(qub_arr, 1)[0]
    temp = dic[gate]+" q[{}".format(str(i))+"],q[{}".format(str(qub_num))+"]"+";\n"
    lis.append(temp)

  else:
    temp = dic[gate]+" q[{}]".format(str(i))+";\n"

    lis.append(temp)
for i in range(qub,num_gates):
  gate = np.random.randint(low = 0, high = 5, size=1)[0]
  if(gate==4):
    qub_arr = np.arange(qub)
    qub_num = np.random.choice(qub, 2, replace = False)
    temp = dic[gate]+" q[{}".format(str(qub_num[0]))+"],q[{}".format(str(qub_num[1]))+"];\n"
    lis.append(temp)

  else:
    qub_num = np.random.randint(low = 0, high = qub, size=1)[0]
    temp = dic[gate]+" q[{}]".format(str(qub_num))+";\n"
    lis.append(temp)


lis_str = "".join(lis)
if(int(sys.argv[2])==1):
  print("Following circuit generated")
  print(lis_str)
state = simulate(lis_str)

circuit = circuit_from_qasm(lis_str)
result = cirq.Simulator().simulate(circuit)
statevector = list(np.around(result.state_vector(), 3))
print("Running cirq simulator and my simulator on this random circuit")
print("Output: ",np.all(np.isclose(state, statevector)))


return
```

Overall the flow was that first I generated 2 numbers: number of qubits and number of gates (in case user doesnt give it). The upper limit for number of qubits was 16 and number of gates was 500. User can change this if he wants. It was ensured that number of gates are greater than number of gates. This is to ensure that all qubits get at 1 least gate operated on. This was to ensure the semantics are correctly aligning with what we did for given qasm programs.

Hence I first iterated through all qubits and assigned 1 gate randomly to each of them. In case of controlled not gate, I randomly generated a second qubit which will be our target qubit. After assigning 1 gate for each of qubits, I then randomly created new operations on random qubits with random gates until number of gates total become equal to the specified. This generated qasm program was totally correct with grammer aligning with the given qasm programs and hence was passed to both cirq and my simulator. Rigorous testing was done to see the validity of my simulator. User can play around with this random qasm circuit generator to generate random qasm programs of number of qubits and number of gates, I have put the instructions on how to run in the README.

1.3 Code readability:

We have modularized our code and used the name of variables and functions, following the conventions of the problem statement or the names are self indicated of the utility they perform. All the code is commented. An exhaustive description of the functions used for implementation is enlisted below for reference:

- `simulate(qasm_string)`: This is the function which is called by `compare_simulator.py`. The function takes in qasm string and does all the work to generate the state vector.
- `Xgate(state,m)`: Takes in the state vector and qubit number on which it has to apply the Not gate. Outputs the new state vector after operation.
- `Hgate(state,m)`: Takes in the state vector and qubit number on which it has to apply the Hadamard gate. Outputs the new state vector after operation.
- `Tgate(state,m)`: Takes in the state vector and qubit number on which it has to apply the T gate. Outputs the new state vector after operation.
- `TDaggate(state,m)`: Takes in the state vector and qubit number on which it has to apply the $T^\dagger$ gate. Outputs the new state vector after operation.
- `CXgate(state,m,t)`: Takes in the state vector and qubit number and control qubit number. Applies the Not gate on qubit m depending on qubit t state and returns the new state vector
- `analysis()`: Does all the analysis for our simulator. Called by main function.
- `random_circuit_generate()`: Generates a random circuit. The exact command line command is written in the readme. The task is to generate a random circuit with any number of qubits and number of gates and check whether it gives same output as cirq simulator.

1.4 Parametrizing the solution in n

As discussed before in simulate function, we created a state vector of size 2^n where n is the number of qubits which is obtained by traversing through the whole program and finding the biggest qubit number in program. This state vector was initialised to [1,0,0,..],since this is the vector which corresponds to all qubits initialised to 0 in weighted ket notation. The parametrization in n is kept throughout the code and the output state vector is also of the size 2^n which is returned.

1.5 Debugging and Testing

To ensure correctness of my simulator, I used the cirq simulator and used it for debugging purposes too. I started with correctness of each gate implmentation and checked the output of cirq vs mine. After doing this for all gates, I went on to write the parsing program, and using standard print function saw the output at every stage to ensure everything was working fine. Finally when I ran on the complete qasm programme, I ran it line by line of the qasm programme to see where my output was coming different from the cirq output. After doing this, I ran the `random_circuit_generate` to generate random qasm programs and tested my simulator output with cirq output. The overall simulator once when passed all test cases, was driven through further following time analysis to get an idea of how valid and optimized my simulator code is.

1.6 Cirq simulator precision loss

After running `random_circuit_generate` for many many times, an interesting finding was made. For the following random qasm program:

```
OPENQASM 2.0;
include "qelib1.inc";
qreg q[16];
creg c[16];
tdg q[0];
h q[1];
cx q[2],q[0];
cx q[3],q[4];
t q[4];
t q[5];
t q[1];
h q[1];
tdg q[3];
cx q[0],q[3];
x q[5];
h q[5];
h q[1];
h q[3];
h q[4];
x q[2];
cx q[2],q[0];
cx q[2],q[3];
t q[4];
t q[4];
tdg q[1];
tdg q[3];
tdg q[3];
x q[1];
t q[0];
cx q[0],q[3];
cx q[1],q[2];
tdg q[4];
h q[4];
h q[0];
t q[0];
cx q[2],q[1];
cx q[4],q[5];
x q[5];
cx q[2],q[3];
cx q[2],q[3];
cx q[5],q[3];
cx q[5],q[0];
h q[5];
tdg q[5];
t q[3];
cx q[3],q[0];
t q[0];
t q[3];
x q[1];
tdg q[5];
```

```
h q[5];
tdg q[0];
tdg q[0];
h q[5];
x q[1];
tdg q[3];
cx q[2],q[1];
x q[3];
t q[0];
x q[0];
x q[2];
cx q[0],q[3];
h q[5];
t q[5];
tdg q[0];
cx q[0],q[2];
h q[4];
h q[1];
h q[0];
x q[3];
tdg q[2];
x q[0];
x q[1];
t q[1];
cx q[1],q[5];
x q[0];
t q[2];
h q[0];
```

The above program gives False as output. The output of cirq simulator for 9th qubit is **-0.02588835-0.06250001j** whereas for my simulator is **-0.02588835-0.0625j**. After rounding, these give different answers. I checked further and analysed why was this the case and realised that cirq was messing up on with the precision while applying hadamard gates. As we know, the cirq simulator output **-0.02588835-0.06250001j** is incorrect and its a precision loss from the simulator side (incorrect because it shoould have been 0.0625 which comes from $1/\sqrt(2)$) of hadamard gate applied at end and cant be 0.0625001). There are many other values where it becomes 0.062499 but after rounding off they become equal, hence dont affect. This is clearly a bug in cirq simulator. Hence its better to compare without rounding off, since then it the state vectors will be close to each other.
All subsequent sections does the analysis part which we described before.

1.7 Scalibility with n

First I checked my simulator passes all 14 test cases. For this part, I computed the run times for these given qasm programs. The qasm programs are there for each possible value of number of qubits in a program starting from 3 and goes till 16 (14 programs are there). We first compare our output with the cirq simulator output and we get all outputs as `True` implying state vectors were same in all 14 qasm programs. We then compare the times taken and plot the times taken vs number of qubits for both our and cirq simulator. Following are the plots:

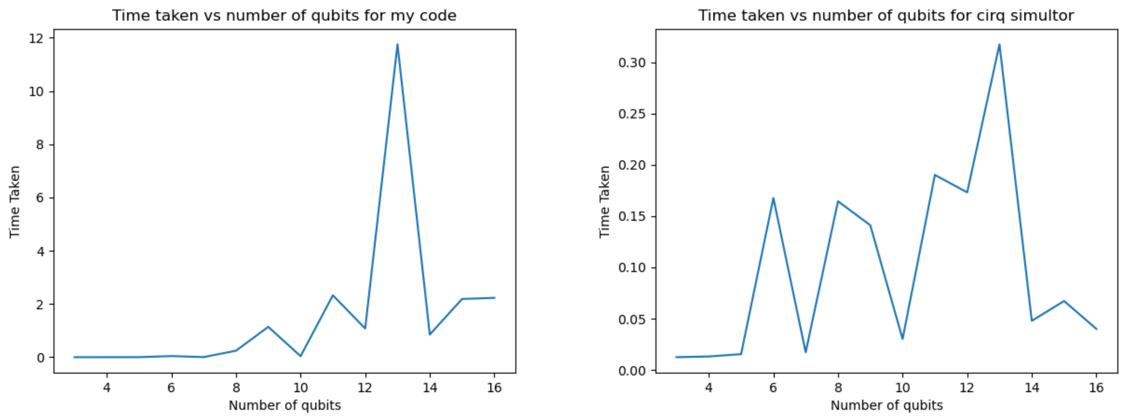

Figure 1: Times taken vs Number of qubits for my simulator and cirq simulator

We observe the following:

- As was ex[ected the times taken wont increase with n. This is because, though it depends on n, a large factor of time is contributed by how many. gates are present in the program. A higher n doesnt imply more numper of gates. Hence we dont see a linear or exponential increase in time taken with n for both my and cirq simulator.
- The times taken by mine and cirq simulator have peaks at same locations (except for qubit 6 where cirq simulator has a peak). Furthermore the plots are similar to each other. This clearly validates our simulator working.
- As we see for qubit 13, highest time is taken for both ours and cirq simulator. This is confiedm further by fact that 13 qubits file correspond to `squar5_261.qasm` file which contains the maximum number of lines = 1997 lines and hence maximum number of gates.
- The times taken by cirq simulators are relatively much less than mine, this is due to the fact that cirq simulator is highly optimized and packaged as a library.

1.8 Analysing the times taken by each gate

In this analysis, we observe the times taken by each gate. The following is the line plot for the same.

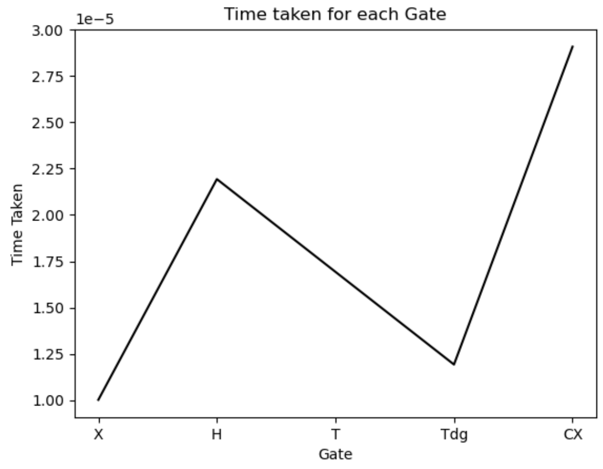

Figure 2: Times taken by each gate

We observe the following:

- As was expected the time taken by C-Not gate is highest since its a multi-qubit gate and hence more processing required to perform the computation.
- The Hadamard gate is second highest since it has to perform updates on 2 vectors of weighted ket, unlike X and T gates which directly put update on single vector fo weighted ket.
- The T, X and Tdg take least time as was expected as they are single qubit gates and apply operation only on single vector.

1.9 Time taken vs Number of gates

It was crucial to analyse the times taken by my simulator as function of number of qubits to verify its validity. As the number of gates increase, its expected that times taken also increase, as we have to call the gate functions that many number of times. For this analysis, we took the `miller_11.qasm` program and computed the times taken as we iterate through the program. Starting with only 1 line, we added 1 line everytime and computed run times each time. Following is the plot observed:

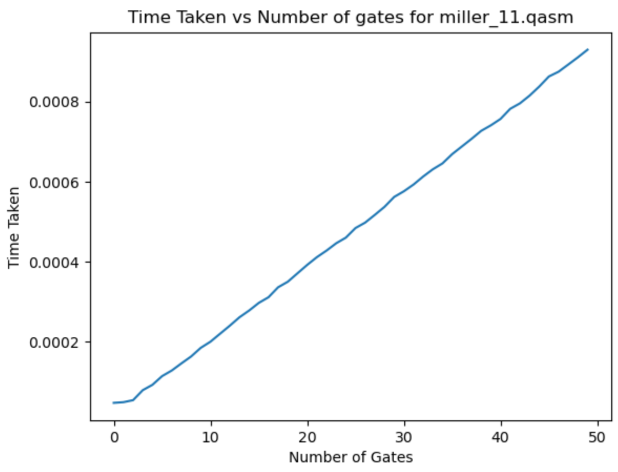

Figure 3: Time taken vs Number of gates in `miller`$_1 1.qasm|$

We observe the following:

- As we observe, as number of gates increase, the times taken also increase to run the program.
- This was expected since number of gates decide the number of calls to the gate functions and hence that much processing needed.
- Similar behaviour is expected for any other qasm program.

2 README

1. Download the python code. Install cirq, numpy and matplotlib.
2. Follow these links: Install Cirq, Install numpy and Install matplotlib.
3. For doing the compare simulator, just run the compare simulator code and keep the `cs238.py` code in same folder as `compare_simulators.py` code. Use the following command to run:

```
python compare_simulators.py qasm-folder
```

where ”qasm-folder” is an arguement passed, the name of folder, where the list of qasm programs are stored. The folder can contain any qasm files.

4. For generating the plots of analysis section in this report, run the following in terminal:

   ```
   python cs238.py qasm-folder
   ```

   where ”qasm-folder” again is folder which contains the list of qasm programs provided before (needed for doing the time analysis). **Do include all the 14 qasm programs with same name**. No such necessity if you just want to run the compare simulators.

5. For testing simulator on qasm program, created at random, run the following command:

   ```
   python cs238.py -gen [verbose] [number of qubits] [number of gates]
   ```

   Here gen implies, we want to generate a random circuit. The next input is whether we want verbose. If 1, then random circuit created will be shown on terminal. The next 2 inputs are number of qubits and number of gates that you want to be in the random circuit. If not specified, code will randomly sample these 2 values. Please keep the number of gates more than number of qubits if providing in command. Following are examples of commands you can try:

   - `python cs238.py -gen 1 2 3`

     Will generate a random qasm program of 2 qubits and 3 gates. Circuit generated will be shown on screen
   - `python cs238.py -gen 1`

     Will generate a random qasm program of random number of qubits and gates. Circuit generated will be shown on screen
   - `python cs238.py -gen 0`

     Will generate a random qasm program of random number of qubits and gates. Circuit generated won’t be shown on screen