ECE786: PROGRAMMING ASSIGNMENT THREE B

Task 1: Apply six single-qubit quantum gates to six different qubits in n-qubit quantum circuit

ALGORITHM

1. Problem Statement Analysis

The problem statement computes matrix multiplication of a one-bit quantum gate (U) on the nth bit of an N qubit quantum state (A). This is done six times to apply the single qubit gates on the quantum circuit.

This is done, by extending the code from Programming Assignment One. The only difference is the source code to read the input file which is modified to read six quantum gates and six bits on which they are applied.

2. Thread Organization

The proposed solution uses a one-dimensional thread organization across gird as well as the thread blocks. This implies each gird will have a 1D array of thread blocks and each corresponding thread block will have a 1D array of threads.

3. Kernel Function

Each thread will perform matrix multiplication on one set of input pairs. The global thread id will be mapped to its corresponding pairs.

$$index_1 = (tid \,\%\, 2^t) + (2^{t+1} * (tid/2^t))$$
$$index_2 = index_1 + 2^t$$

tid: global thread id
t: position of bit on which gate is applied.

SIMULATION ON GPGPUSIM

Input	Global Memory Accesses
input_for_qc7_q0_q2_q3_q4_q5_q6	324
input_for_qc10_q0_q1_q3_q5_q7_q9	1440
input_for_qc12_q6_q7_q8_q9_q10_q11	5184
input_for_qc16_q0_q2_q4_q6_q8_q10	80064

Task 2: Use shared memory to optimize the code in task 1

ALGORITHM

1. Problem Statement Analysis
   The objective of the task is to use shared memory of each SM to optimize the application of each single qubit gate to the input quantum circuit. Instead of loading the data for each computation from the host memory, its copied to the shared memory. Shared memory is then accessed to do the computation. Finally, the result is copied back to the host memory.

2. Thread Organization
   The proposed solution uses a one-dimensional thread organization across gird as well as the thread blocks. This implies each gird will have a 1D array of thread blocks and each corresponding thread block will have a 1D array of threads.

   $K = \{t_1, t_2 ... t_n\}$
   # thread_blocks = $2^N/2^k$
   # threads_per_block = $2^k/2$
   # elements_in_shared_memory = 2^k
   N – number of bits in quantum circuit
   k – number of qubits in K

3. Kernel Function
   The kernel function is mainly divided into two major parts.
   - Loading elements into shared memory.
     - Let K consist of active qubit positions, any qubit that is not in K is called inactive qubit position. Each thread will load two elements into the shared memory by toggling the active and inactive bit position to generate the indices for elements to be loaded.
     - More precisely, thread block id will be mapped to the inactive bits, that will be common for both the elements loaded by the thread.
     - The active bits will be mapped based on the thread id. This combination will allow each thread to load two unique values into the shared memory.
   - Apply quantum gate to respective qubits.
     - Once the shared memory is loaded, it is observed that the six quantum gates are serially applied on incremental qubit indices of the quantum circuit irrespective of the actual values in K. This pattern is repeated across all thread blocks.
     - This is primarily because the data is rearranged and loaded into the shared memory. The code from programming assignment one is used to apply quantum gates to the first six qubits of the quantum circuit.
     - After all the gates have been applied each thread loads data back from shared memory to the host memory.

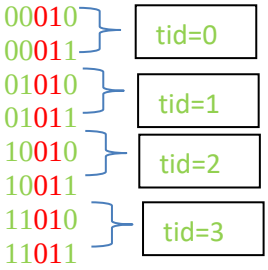
4. Thread Mapping Example

   A_{32x1}
   K = {0,3,4}
   Inactive = {1,2}
   # blocks = 32/8 = 4
   # threads/block = 8/2 = 4

   For, thread block id = 1
   shared_memory = {2, 3, 10, 11, 18, 19, 26, 27}

thread Id	t = 0	t = 3	t = 4
0	(2,3) [(0,1)]	(2,10) [(0,2)]	(2,18) [(0,4)]
1	(10,11) [(2,3)]	(3,11) [(1,3)]	(3,19) [(5,21)]
2	(18,19) [(4,5)]	(18,26) [(4,6)]	(10,26) [(12,28)]
3	(26,27) [(6,7)]	(19,27) [(5,7)]	(11,27) [(13,29)]

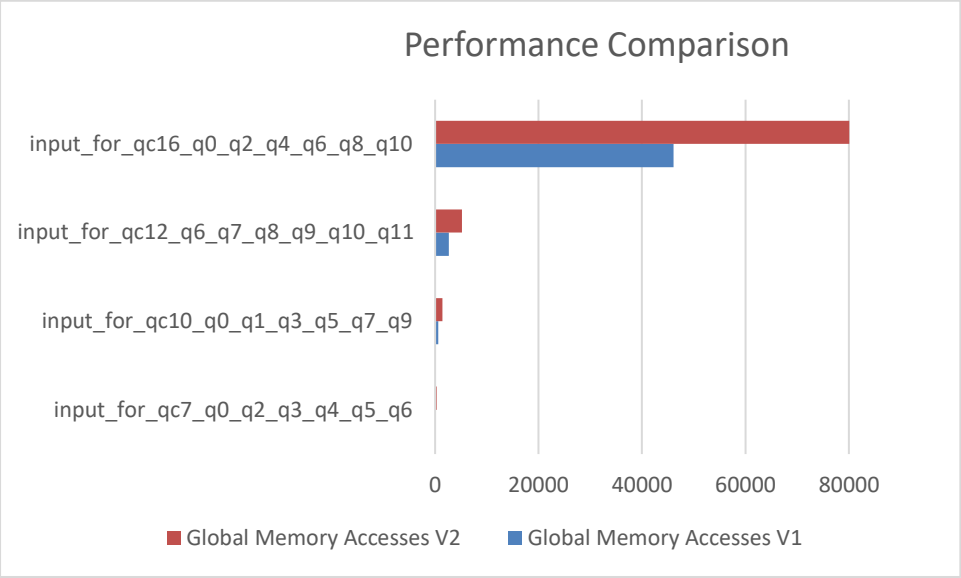
The numbers in red, denote the global indices.
The numbers in green, denote the shared memory indices of the corresponding global indices.



SIMULATION ON GPGPUSIM

Input	Global Memory Accesses
input_for_qc7_q0_q2_q3_q4_q5_q6	72
input_for_qc10_q0_q1_q3_q5_q7_q9	624
input_for_qc12_q6_q7_q8_q9_q10_q11	2624
input_for_qc16_q0_q2_q4_q6_q8_q10	46080

Memory Access Comparison of Task1 and Task2



The above graph summarizes the global memory accesses between task 2 and task 3. The following are the key points to note:

- It is evident that utilizing shared memory reduced the number of global memory accesses for all input use cases. This in return reduced the number of cache misses as the data accesses by each thread has poor spatial locality. This improves the performance of the code.
- It is also evident that as the size of the quantum circuit increases, the difference in the performance of the cuda kernel becomes more apparent.