Name: Nirmal Kumar Sedhumadhavan

ECE 786 Programming Assignment 3B

Optimize the quantum circuit simulation process of applying six single-qubit gates to six different qubits in an n-qubit quantum circuit.

Task 1

CUDA Kernel Function:

```
__global__ void matrix_multiply(const float *input, float *output, const float *Umatrix, int size, int qbit)
{
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  int mask = 1 << qbit; //Calculates the value of 2 to the power of qbit(2^qbit)
  int index = i ^ mask; //Calcultes the position of the 2nd element
  if (i <= (size - mask))
  {
    if((i/mask) % 2 != 1) // Check if the 2nd element is already calculated
    {
     output[i] = Umatrix[0] * input[i] + Umatrix[1] * input[index]; //Update the 1st element
     output[index] = Umatrix[2] * input[i] + Umatrix[3] * input[index]; //Update the 2nd element
    }
  }
}
```

Explanation:

The implemented kernel function matrix_multiply takes input vector (float *input), unitary matrix vector (float *Umatrix), qbit position (int qbit), size of the input vector (int size), output vector (float *output) as arguments.

Inside the function, the index for the threads is calculated based on the thread hierarchy. Each thread processes 2 elements in the output vector.

Task 2

CUDA Kernel Function:

```
__global__ void matrix_multiply(float *d_A, float *d_B, float *d_U, int *d_indices, int *d_TB_indices, int *d_qbit_indices)
{
    __shared__ float s[64];
    float temporary = 0;
    int i = 2*threadIdx.x;
    int j = blockIdx.x;
    //Copy the input vector to the shared variable
    s[i] = d_A[d_indices[i] + d_TB_indices[j]];
    s[i+1] = d_A[d_indices[i+1] + d_TB_indices[j]];
    __syncthreads();
    //Choose the pair of elements where the qubit gate need to be applied
    //Base address = (2*threadIdx.x - threadIdx.x % d_qbit_indices[k])
    //Base address + qubit offset = (2*threadIdx.x - threadIdx.x % d_qbit_indices[k]) + d_qbit_indices[k]
    //Where d_qbit_indices[6]={1,2,4,8,16,32}
    for(int k = 0; k < 6; k++)
    {
        temporary = d_U[(k*4)] * s[i - threadIdx.x % d_qbit_indices[k]] + d_U[(k*4) + 1] * s[(i - threadIdx.x % d_qbit_indices[k]) + d_qbit_indices[k]];
        s[(i - threadIdx.x % d_qbit_indices[k]) + d_qbit_indices[k]] = d_U[(k*4) + 2] * s[i - threadIdx.x % d_qbit_indices[k]] + d_U[(k*4) + 3] * s[(i - threadIdx.x % d_qbit_indices[k]) +
        s[(i - threadIdx.x % d_qbit_indices[k])] = temporary;
    }
    __syncthreads();
    //Copy the shared variable to the output vector
    d_B[d_indices[i] + d_TB_indices[j]] = s[i];
    d_B[d_indices[i+1] + d_TB_indices[j]] = s[i+1];
}
```

Explanation:

The implemented kernel function matrix_multiply takes input vector (float *d_A), unitary matrix vector (float *d_U), indices for the shared memory position (int *d_indices), Thread Block indices for the shared memory position (int *d_TB_indices), indices for the qubit offset position (int *d_qbit_indices), output vector (float *d_b) as arguments.

Indices for the shared memory are calculated by the following logic.

Starting from the lowest qubit. The first qubit is varied 01010101, the second qubit is varied 001100110011, the third qubit is varied 0000111100001111 till sixth qubit.

Thread block indices for the offset of indices are calculated by the following logic.

Starting from the lowest non qubit. The first TB is set to 0. TB 1 can be represented as 01 so the first non-qubit is set as offset (2^non-qubit [0]). TB 2 can be represented as 10 so the second non-qubit is set as offset (2^non-qubit [1]). TB 3 can be represented as 11 so the first & second non-qubit is set as offset (2^non-qubit [0] + 2^non-qubit [1]).

Global Memory Access (gpgpu_n_mem_read_global + gpgpu_n_mem_write_global)

Input	Task 1	Task 2
input_for_qc7_q0_q2_q3_q4_q5_q6	839	112
input_for_qc10_q0_q1_q3_q5_q7_q9	6,636	976
input_for_qc12_q6_q7_q8_q9_q10_q11	21,120	9,024
input_for_qc16_q0_q2_q4_q6_q8_q10	435,656	51,136

From the above data it is clear that the use of ShareMem method has reduced a significant number of Global Memory Access.

Task 3

CUDA Kernel Function:

```
__global__ void matrix_multiply(float *input, float *output, float *Umatrix, int size, int qbit)
{
    int i = blockDim.x * blockIdx.x + threadIdx.x;
    int segment1 = i/qbit;
    int j= i + (size/4);
    int segment2 = j/qbit;

    i = i+segment1*qbit; //Indexing i according to the qubit gate applied
    j = j+segment2*qbit; //Indexing j according to the qubit gate applied

    output[i] = (Umatrix[0]*input[i]) + (Umatrix[1]*input[i+qbit]); //Update the 1st element
    output[i+qbit] = (Umatrix[2]*input[i])+(Umatrix[3]*input[i+qbit]); //Update the 2nd element
    output[j] = (Umatrix[0]*input[j]) + (Umatrix[1]*input[j+qbit]); //Update the 3rd element
    output[j+qbit] = (Umatrix[2]*input[j])+(Umatrix[3]*input[j+qbit]); //Update the 4th element
}
```

Explanation:

The implemented kernel function matrix_multiply takes input vector (float *input), unitary matrix vector (float *Umatrix), qbit position (int qbit), size of the input vector (int size), output vector (float *output) as arguments.

Inside the function, the index for the threads is calculated based on the thread hierarchy. Each thread processes 4 elements in the output vector.