NC State University

Department of Electrical and Computer Engineering

ECE 786: Spring 2023

Programming Assignment #3b

CUDA Programming

by

RAGHUL SRINIVASAN

UnityID: rsriniv7

(200483357)

1. **TASK 1: Write the code to read in the input file and generate the expected output result of applying six single-qubit quantum gates to six different qubits in an n-qubit quantum circuit**

Kernel Function – Qubit Matrix multiplication

```
__global__ void QuantumGate(float *A, float *B, float *C, uint64_t Alength, uint64_t Blength)
{
    uint64_t Aposition = threadIdx.x;
    uint64_t Bposition = blockIdx.x;

    float MatrixResult = 0;
    if ((Aposition < Alength) && (Bposition < Blength))
    {
        for (uint64_t Iteration = 0; Iteration < Alength; Iteration++)
            MatrixResult += A[(Aposition * Alength) + Iteration] * B[(Iteration * Blength) + Bposition];
        C[(Aposition * Blength) + Bposition] = MatrixResult;
    }
    __syncthreads();
}
```

Calling Kernel Function in int main()

```
dim3 BlockCount(InputMatrixSize / 2, 1);
dim3 ThreadCount(2, 1);

struct timeval begin, end;
gettimeofday(&begin, NULL);
QuantumGate<<<BlockCount, ThreadCount>>>(d_A, d_B, d_C, 2, InputMatrixSize / 2);
gettimeofday(&end, NULL);
uint64_t time_in_us = 1e6 * (end.tv_sec - begin.tv_sec) + (end.tv_usec - begin.tv_usec);
// cout << "Run Time -> " << time_in_us << " us" << '\n';
```

Explanation:

- The kernel function **QuantumGate()** calculates matrix multiplication between A and B. Result is stored in C.
- Number of threads and blocks assigned are 2 and (InputMatrixSize/2) respectively, where InputMatrixSize will always be in powers of 2.
- **InputMatrixSize** is calculated from number on lines in the input text files.
- The appropriate indexes of A and B are calculated using **blockIdx.x** and **threadIdx.x**
- **Alength** will always be 2, as we are doing 2x2 matrix multiplication to calculate qbits.
- **if ((Aposition < Alength) && (Bposition < Blength))** -> we are using this condition so as to check if the input index calculated is within the expected size.
- Input matrixes A and B are sent via 1D array and the calucated output is also stored in 1D array C.
- **Alength** and **Blength** are also passed as arguments to the Kernel function along with the Input Matrixes.
- The kernel function **Quantumgate** is called in the main() function 6 times in for loop and the output matrix of the previous iteration is the input matrix for next iteration.

```
    }
    for (uint64_t i = 0; i < InputMatrixSize; i++)
        InputMatrixfromFile[i] = OutputMatrix[i];
}
```

2. TASK 2: Use shared memory to optimize the above code you have written

Kernel Function – Qubit Matrix multiplication

```
__global__ void QuantumGate(float *A, float *B, float *C, uint64_t Alength, uint64_t Blength)
{
    uint64_t Aposition = threadIdx.x;
    uint64_t Bposition = blockIdx.x;
    float MatrixResult[2];
    __shared__ float SharedMemMatrix[64];

    if ((Aposition < Alength) && (Bposition < Blength))
    {
        if (Aposition == 0)
        {
            for (uint64_t Index = 0; Index < 64; Index++)
                SharedMemMatrix[Index] = B[(Bposition * 64) + Index];
        }
        __syncthreads();

        for (uint64_t QuBitOperationCount = 0; QuBitOperationCount < 6; QuBitOperationCount++)
        {
            uint64_t QbitPower = 1 << QuBitOperationCount;
            uint64_t Remainder = Aposition % QbitPower;

            MatrixResult[0] = 0;
            MatrixResult[1] = 0;
            for (uint64_t Iteration = 0; Iteration < 2; Iteration++)
            {
                MatrixResult[0] += A[(QuBitOperationCount * 4) + Iteration + 0] * SharedMemMatrix[(Iteration * QbitPower) + ((Aposition - Remainder) * 2) + Remainder];
                MatrixResult[1] += A[(QuBitOperationCount * 4) + Iteration + 2] * SharedMemMatrix[(Iteration * QbitPower) + ((Aposition - Remainder) * 2) + Remainder];
            }

            SharedMemMatrix[(0 * QbitPower) + ((Aposition - Remainder) * 2) + Remainder] = MatrixResult[0];
            SharedMemMatrix[(1 * QbitPower) + ((Aposition - Remainder) * 2) + Remainder] = MatrixResult[1];
            C[(Bposition * 64) + (0 * QbitPower) + ((Aposition - Remainder) * 2) + Remainder] = MatrixResult[0];
            C[(Bposition * 64) + (1 * QbitPower) + ((Aposition - Remainder) * 2) + Remainder] = MatrixResult[1];
             __syncthreads();
        }
    }
    __syncthreads();
}
```

Calling Kernel Function in int main()

```
dim3 BlockCount(InputMatrixSize / (32 * 2), 1);
dim3 ThreadCount(32, 1);

struct timeval begin, end;
gettimeofday(&begin, NULL);
QuantumGate<<<BlockCount, ThreadCount>>>(d_A, d_B, d_C, 32, InputMatrixSize / (32 * 2));
gettimeofday(&end, NULL);
uint64_t time_in_us = 1e6 * (end.tv_sec - begin.tv_sec) + (end.tv_usec - begin.tv_usec);
```

Explanation:

- In the main function, input matrixes are sorted in a group of 64 (2^6) which would be accessed by each of the threads in the same block.
- Each block has 32 threads and there are (InputMatix size/64) Thread blocks.
- Inside the kernel, each of the threads with threadID = 1 will copy the corresponding (resp to block ID) 64 input matrix elements to shared memory.
- Each of the threads calculates the corresponding indexes for the input matrix to be considered to perform matrix multiplication depending on the QuBit position.
- Each thread calculates two output matrix elements and stores the result in the corresponding location in shared memory which is to be used as input for iteration.
- The above two steps are repeated 6 times, but the QuBit position will vary.
- Finally, the result is stored in the C matrix and copied to the device memory.

3. TASK 3: Thread coarsening optimization

Kernel Function – Qubit Matrix multiplication

```
__global__ void QuantumGate(float *A, float *B, float *C, uint64_t Alength, uint64_t Blength)
{
    uint64_t Aposition = 2 * threadIdx.x;
    uint64_t Bposition = blockIdx.x;
    float MatrixResult[4];
    __shared__ float SharedMemMatrix[64];

    if ((Aposition < Alength) && (Bposition < Blength))
    {
        if (Aposition == 0)
        {
            for (uint64_t Index = 0; Index < 64; Index++)
                SharedMemMatrix[Index] = B[(Bposition * 64) + Index];
        }
        __syncthreads();
        for (uint64_t QuBitOperationCount = 0; QuBitOperationCount < 6; QuBitOperationCount++)
        {
            uint64_t QbitPower = 1 << QuBitOperationCount;
            uint64_t Remainder = Aposition % QbitPower;
            uint64_t Remainder1 = (Aposition + 1) % QbitPower;

            MatrixResult[0] = 0;
            MatrixResult[1] = 0;
            MatrixResult[2] = 0;
            MatrixResult[3] = 0;
            for (uint64_t Iteration = 0; Iteration < 2; Iteration++)
            {
                MatrixResult[0] += A[(QuBitOperationCount * 4) + Iteration + 0] * SharedMemMatrix[(Iteration * QbitPower) + ((Aposition - Remainder) * 2) + Remainder];
                MatrixResult[1] += A[(QuBitOperationCount * 4) + Iteration + 2] * SharedMemMatrix[(Iteration * QbitPower) + ((Aposition - Remainder) * 2) + Remainder];
                MatrixResult[2] += A[(QuBitOperationCount * 4) + Iteration + 0] * SharedMemMatrix[(Iteration * QbitPower) + ((Aposition + 1 - Remainder1) * 2) + Remainder1];
                MatrixResult[3] += A[(QuBitOperationCount * 4) + Iteration + 2] * SharedMemMatrix[(Iteration * QbitPower) + ((Aposition + 1 - Remainder1) * 2) + Remainder1];
            }
            SharedMemMatrix[(0 * QbitPower) + ((Aposition - Remainder) * 2) + Remainder] = MatrixResult[0];
            SharedMemMatrix[(1 * QbitPower) + ((Aposition - Remainder) * 2) + Remainder] = MatrixResult[1];
            SharedMemMatrix[(0 * QbitPower) + ((Aposition + 1 - Remainder1) * 2) + Remainder1] = MatrixResult[2];
            SharedMemMatrix[(1 * QbitPower) + ((Aposition + 1 - Remainder1) * 2) + Remainder1] = MatrixResult[3];

            C[(Bposition * 64) + (0 * QbitPower) + ((Aposition - Remainder) * 2) + Remainder] = MatrixResult[0];
            C[(Bposition * 64) + (1 * QbitPower) + ((Aposition - Remainder) * 2) + Remainder] = MatrixResult[1];
            C[(Bposition * 64) + (0 * QbitPower) + ((Aposition + 1 - Remainder1) * 2) + Remainder1] = MatrixResult[2];
            C[(Bposition * 64) + (1 * QbitPower) + ((Aposition + 1 - Remainder1) * 2) + Remainder1] = MatrixResult[3];

            __syncthreads();
        }
    }
    __syncthreads();
}
```

Calling Kernel Function in int main()

```
dim3 BlockCount(InputMatrixSize / (32 * 2), 1);
dim3 ThreadCount(16, 1);

struct timeval begin, end;
gettimeofday(&begin, NULL);
QuantumGate<<<BlockCount, ThreadCount>>>(d_A, d_B, d_C, 32, InputMatrixSize / (32 * 2));
gettimeofday(&end, NULL);
```

Explanation:

- The procedure is same as task 2, except that there are only 16 threads per thread block but the number of thread blocks is the same.
- And each thread would compute four output matrix elements (Thread coarsening)
- On comparing with Task2, here the computation done by each thread is doubled.

Global Memory Count Analysis:

Input File Considered: **input_for_qc16_q0_q2_q4_q6_q8_q10.txt**

CUDA File	Global Memory Accesses
quamsimV1.cu (Task1)	1,179,648
quamsimV2.cu (Task2)	163,840
quamsimV3.cu (Task3)	237,568

- By using the SharedMem method in Task2, we have reduced the number of accesses to shared memory thereby saving computation time.
- But by using thread coarsening, there is slight increase in shared memory access count, as the number of threads/block is less.