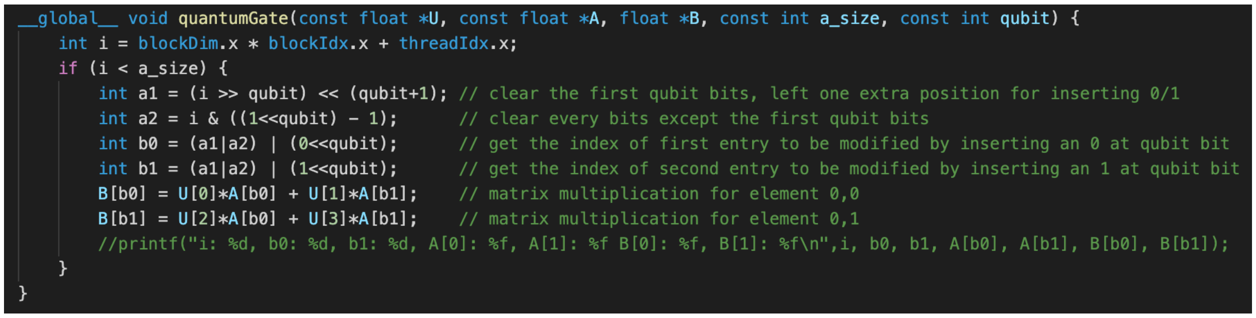
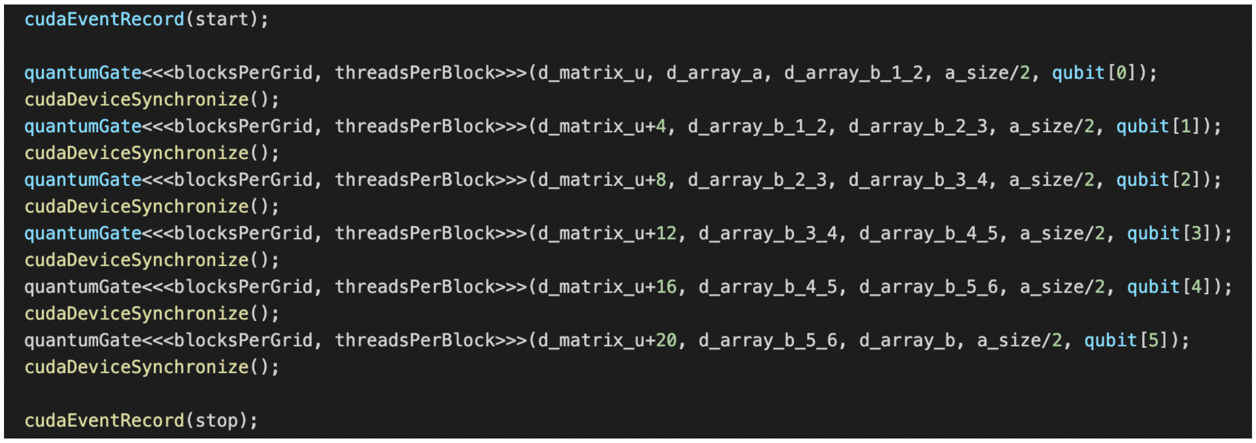
Project 3B report

Task 1:

Kernel function implementation is identical as project1, shown as below. The only difference is now the kernel function is called 6 times, with 6 different qubit & 2x2 matrix as input, plus the synchronization in-between them, shown as below.



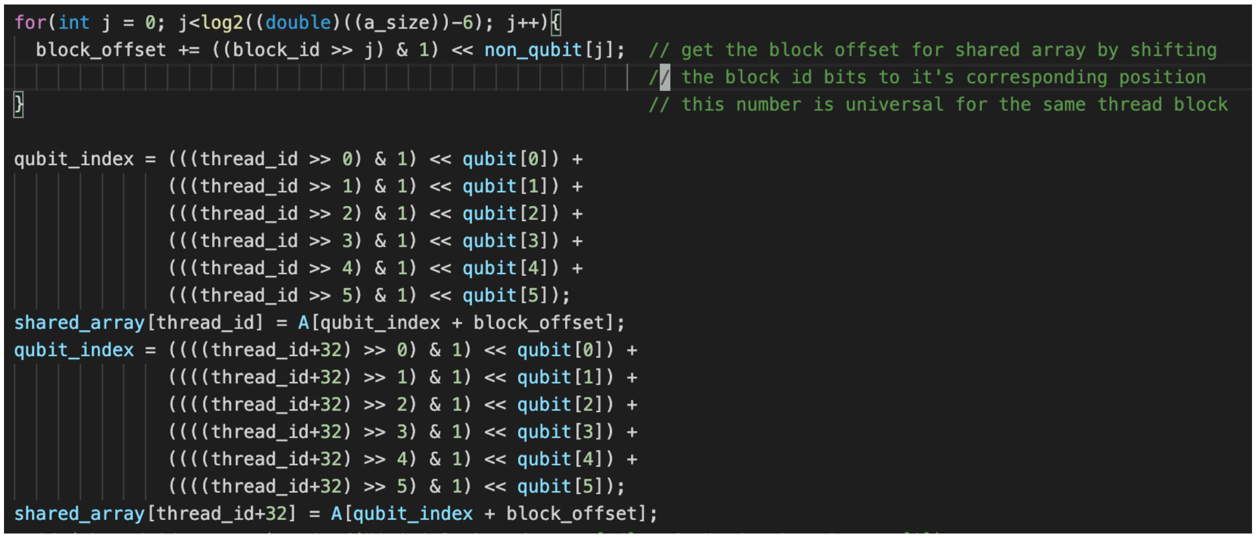
```
__global__ void quantumGate(const float *U, const float *A, float *B, const int a_size, const int qubit) {
    int i = blockDim.x * blockIdx.x + threadIdx.x;
    if (i < a_size) {
        int a1 = (i >> qubit) << (qubit+1); // clear the first qubit bits, left one extra position for inserting 0/1
        int a2 = i & ((1<<qubit) - 1);      // clear every bits except the first qubit bits
        int b0 = (a1|a2) | (0<<qubit);      // get the index of first entry to be modified by inserting an 0 at qubit bit
        int b1 = (a1|a2) | (1<<qubit);      // get the index of second entry to be modified by inserting an 1 at qubit bit
        B[b0] = U[0]*A[b0] + U[1]*A[b1];    // matrix multiplication for element 0,0
        B[b1] = U[2]*A[b0] + U[3]*A[b1];    // matrix multiplication for element 0,1
        //printf("i: %d, b0: %d, b1: %d, A[0]: %f, A[1]: %f B[0]: %f, B[1]: %f\n",i, b0, b1, A[b0], A[b1], B[b0], B[b1]);
    }
}
```



```
cudaEventRecord(start);

quantumGate<<<blocksPerGrid, threadsPerBlock>>>(d_matrix_u, d_array_a, d_array_b_1_2, a_size/2, qubit[0]);
cudaDeviceSynchronize();
quantumGate<<<blocksPerGrid, threadsPerBlock>>>(d_matrix_u+4, d_array_b_1_2, d_array_b_2_3, a_size/2, qubit[1]);
cudaDeviceSynchronize();
quantumGate<<<blocksPerGrid, threadsPerBlock>>>(d_matrix_u+8, d_array_b_2_3, d_array_b_3_4, a_size/2, qubit[2]);
cudaDeviceSynchronize();
quantumGate<<<blocksPerGrid, threadsPerBlock>>>(d_matrix_u+12, d_array_b_3_4, d_array_b_4_5, a_size/2, qubit[3]);
cudaDeviceSynchronize();
quantumGate<<<blocksPerGrid, threadsPerBlock>>>(d_matrix_u+16, d_array_b_4_5, d_array_b_5_6, a_size/2, qubit[4]);
cudaDeviceSynchronize();
quantumGate<<<blocksPerGrid, threadsPerBlock>>>(d_matrix_u+20, d_array_b_5_6, d_array_b, a_size/2, qubit[5]);
cudaDeviceSynchronize();

cudaEventRecord(stop);
```

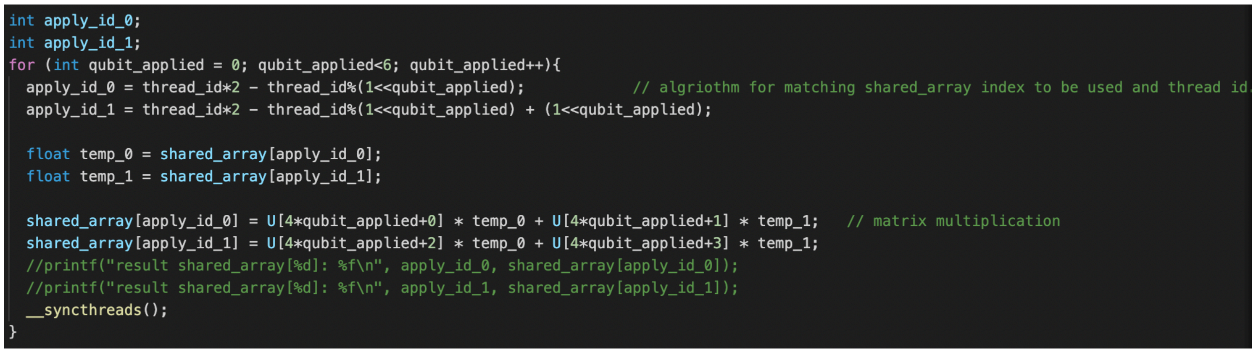
Task 2:

Array is split into multiple partitions with size of 2^6. The bit index for original array is partitioned into two parts - the bits that's mapped to block ID and the bits that's mapped to qubits. Threads in the same thread block shares the same block ID and thus share the same 2^6 array and same "block offset". A 64 sized shared array is fetched based on the unchanging "block offset" and the changing 6 bits qubit as its index, shown as below.



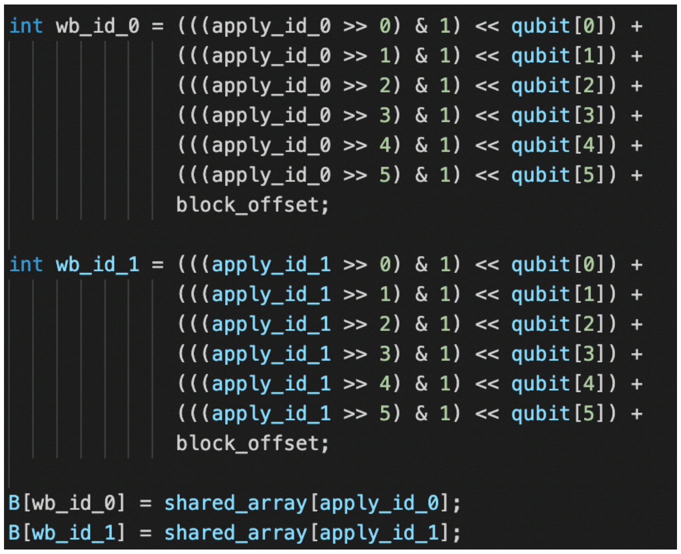
```
for(int j = 0; j<log2((double)((a_size))-6); j++){
  block_offset += ((block_id >> j) & 1) << non_qubit[j];  // get the block offset for shared array by shifting
                                                          // the block id bits to it's corresponding position
}                                                         // this number is universal for the same thread block

qubit_index = (((thread_id >> 0) & 1) << qubit[0]) +
              (((thread_id >> 1) & 1) << qubit[1]) +
              (((thread_id >> 2) & 1) << qubit[2]) +
              (((thread_id >> 3) & 1) << qubit[3]) +
              (((thread_id >> 4) & 1) << qubit[4]) +
              (((thread_id >> 5) & 1) << qubit[5]);
shared_array[thread_id] = A[qubit_index + block_offset];
qubit_index = ((((thread_id+32) >> 0) & 1) << qubit[0]) +
              ((((thread_id+32) >> 1) & 1) << qubit[1]) +
              ((((thread_id+32) >> 2) & 1) << qubit[2]) +
              ((((thread_id+32) >> 3) & 1) << qubit[3]) +
              ((((thread_id+32) >> 4) & 1) << qubit[4]) +
              ((((thread_id+32) >> 5) & 1) << qubit[5]);
shared_array[thread_id+32] = A[qubit_index + block_offset];
```

After then, 6 matrix multiplication is performed similar to Task1, but with synchronization within the kernel instead of in-between. Shown as below.



```
int apply_id_0;
int apply_id_1;
for (int qubit_applied = 0; qubit_applied<6; qubit_applied++){
  apply_id_0 = thread_id*2 - thread_id%(1<<qubit_applied);          // algriothm for matching shared_array index to be used and thread id
  apply_id_1 = thread_id*2 - thread_id%(1<<qubit_applied) + (1<<qubit_applied);

  float temp_0 = shared_array[apply_id_0];
  float temp_1 = shared_array[apply_id_1];

  shared_array[apply_id_0] = U[4*qubit_applied+0] * temp_0 + U[4*qubit_applied+1] * temp_1;   // matrix multiplication
  shared_array[apply_id_1] = U[4*qubit_applied+2] * temp_0 + U[4*qubit_applied+3] * temp_1;
  //printf("result shared_array[%d]: %f\n", apply_id_0, shared_array[apply_id_0]);
  //printf("result shared_array[%d]: %f\n", apply_id_1, shared_array[apply_id_1]);
  __syncthreads();
}
```

When all computation is done, the shared array's index is combined with block offset to compute the real index on the complete output array, and the shared array is written back based off it. Shown as below.

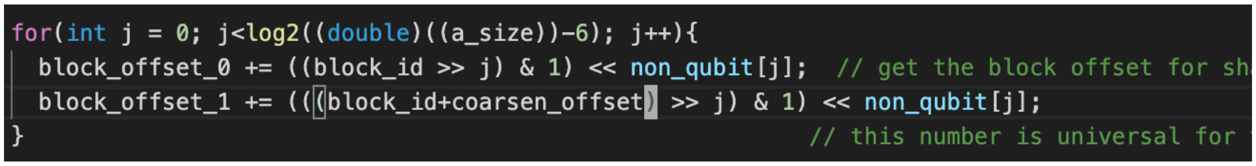


```
int wb_id_0 = (((apply_id_0 >> 0) & 1) << qubit[0]) +
              (((apply_id_0 >> 1) & 1) << qubit[1]) +
              (((apply_id_0 >> 2) & 1) << qubit[2]) +
              (((apply_id_0 >> 3) & 1) << qubit[3]) +
              (((apply_id_0 >> 4) & 1) << qubit[4]) +
              (((apply_id_0 >> 5) & 1) << qubit[5]) +
              block_offset;

int wb_id_1 = (((apply_id_1 >> 0) & 1) << qubit[0]) +
              (((apply_id_1 >> 1) & 1) << qubit[1]) +
              (((apply_id_1 >> 2) & 1) << qubit[2]) +
              (((apply_id_1 >> 3) & 1) << qubit[3]) +
              (((apply_id_1 >> 4) & 1) << qubit[4]) +
              (((apply_id_1 >> 5) & 1) << qubit[5]) +
              block_offset;

B[wb_id_0] = shared_array[apply_id_0];
B[wb_id_1] = shared_array[apply_id_1];
```

Task 3:

Coarsening is done by combining the work of some thread blocks into one. Thus first change to be bad is the number of block per grid. In this case, a coarsening degree of 2 is implemented, thus number of block per grid is halved.

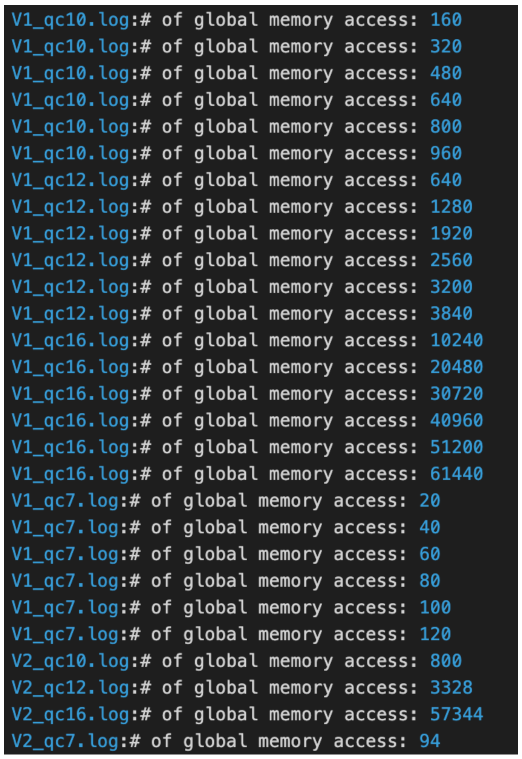
Next, add another set of shared array & block offset to compute another set of array that was originally mapped to the other half of the thread block. One more variable is needed here to find the original block ID of such thread block, and that variable is equal to (the original number of blocks - the current number of blocks), which equals to current number of blocks since the degree of coarsening here is 2.



```
for(int j = 0; j<log2((double)((a_size))-6); j++){
  block_offset_0 += ((block_id >> j) & 1) << non_qubit[j];  // get the block offset for sh
  block_offset_1 += (((block_id+coarsen_offset) >> j) & 1) << non_qubit[j];
}                                                          // this number is universal for
```

Global Memory Access:

Number of global memory access using GPGPU-SIM is shown below.



```
V1_qc10.log:# of global memory access: 160
V1_qc10.log:# of global memory access: 320
V1_qc10.log:# of global memory access: 480
V1_qc10.log:# of global memory access: 640
V1_qc10.log:# of global memory access: 800
V1_qc10.log:# of global memory access: 960
V1_qc12.log:# of global memory access: 640
V1_qc12.log:# of global memory access: 1280
V1_qc12.log:# of global memory access: 1920
V1_qc12.log:# of global memory access: 2560
V1_qc12.log:# of global memory access: 3200
V1_qc12.log:# of global memory access: 3840
V1_qc16.log:# of global memory access: 10240
V1_qc16.log:# of global memory access: 20480
V1_qc16.log:# of global memory access: 30720
V1_qc16.log:# of global memory access: 40960
V1_qc16.log:# of global memory access: 51200
V1_qc16.log:# of global memory access: 61440
V1_qc7.log:# of global memory access: 20
V1_qc7.log:# of global memory access: 40
V1_qc7.log:# of global memory access: 60
V1_qc7.log:# of global memory access: 80
V1_qc7.log:# of global memory access: 100
V1_qc7.log:# of global memory access: 120
V2_qc10.log:# of global memory access: 800
V2_qc12.log:# of global memory access: 3328
V2_qc16.log:# of global memory access: 57344
V2_qc7.log:# of global memory access: 94
```

	qc7	Qc10	qc12	qc16
V1 (original)	420	3360	13440	215040
V2 (shared memory optimized)	94	800	3328	57344

Before Optimizing, each thread will access global memory 12 times (one read and one write for each qubit). After optimization, each thread will only need to access 2 times (one read at the beginning and one write at the end). Thus theoretically, the difference between number of global memory access should be 6 times difference. However, the result yields ~4 times difference. The extra global memory access from shared memory optimized code might be caused by the overhead of synchronization between threads.