ECE 786
Project

Programming assignment 3B report

Chandu Yuvarajappa
cyuvara

TASK 1

In an n-bit quantum circuit, it performs six single-qubit quantum gates to six separate qubits. In order to accomplish this, the CUDA kernel function is called six times in a row, computing a single qubit quantum gate to a different bit each time. Each kernel call's output is supplied into the following stage as input by being written back into an intermediate array with the same size as the input. To match the TB size in task2, I set the thread block size to 32 in this instance. Despite the fact that the total number of threads (TB) is 32, only 16 threads really conduct computations since each thread reads the value of its counterpart thread and its corresponding value.
Assume that the q-bits (0, 2, 4, 8, 9, 10) in a 12 bit quantum circuit. The first kernel call computes a single qubit quantum gate on bit 0, the second kernel call computes a quantum gate on bit 2, and the third call will soon compute a quantum gate on bit 4.

the kernel code

The qubit location is represented by this kernel function's final parameter, "quamsim." A is the input vector, B is the output vector, and U is the quantum matrix.

```
__global__ void
quamsim(const float *U, const float *A, float *B, int number_elem,int q)
{
    int i = blockDim.x * blockIdx.x + threadIdx.x;
    int quad_i = i^(1<<q);

    if (i < number_elem)
    {if((i& 1<<q )==0){
          B[i] =  (U[0]*A[i]) + (U[1]*A[quad_i]);
          B[quad_i] =  (U[2]*A[i]) + (U[3]*A[quad_i]);   }
    }}
```

```
CallKernelFunction(d_U,h1_U,d_A, d_B1,threadsPerBlock,blocksPerGrid,number_elem,q_bit[0],itert++);
CallKernelFunction(d_U,h2_U,d_B1,d_B2,threadsPerBlock,blocksPerGrid,number_elem,q_bit[1],itert++);
CallKernelFunction(d_U,h3_U,d_B2,d_B3,threadsPerBlock,blocksPerGrid,number_elem,q_bit[2],itert++);
CallKernelFunction(d_U,h4_U,d_B3,d_B4,threadsPerBlock,blocksPerGrid,number_elem,q_bit[3],itert++);
CallKernelFunction(d_U,h5_U,d_B4,d_B5,threadsPerBlock,blocksPerGrid,number_elem,q_bit[4],itert++);
CallKernelFunction(d_U,h6_U,d_B5,d_B6,threadsPerBlock,blocksPerGrid,number_elem,q_bit[5],itert++);
```

Here, the output from the first stage, denoted by the letter d B1, is sent as input to the second stage, denoted by the letter d B2, and so on.

TASK 2

Here, 2n values are divided into 64 (6 qubits, or 26) independent fragments, each of which is given a TB. Each block of threads has its own separate shared memory of fragment size 26, which is only accessible

by the threads in that block and not by threads in other blocks. 32 threads can calculate 64 values since each thread can compute the results of two complement threads. These 64 variables are read into shared memory in TB from global memory. All non-qubit locations read from global memory will have identical location indices, with the exception of q_bits. Each thread block loads its fragment from global memory and copies them to shared memory, apply the gates on it, and store the fragment back to global memory.

If n represents the size of a quantum circuit and k represents the number of q_bits, then n-k bits will be constant for all 2k indices. So, we must set global memory indices so that n-k bits remain the same for 2K read values.

The problem with task 1 is that every kernel call reads and writes to global memory. Because we are already storing the data from indices with n-k identical bits, this is resolved in job 2. As shared memory already contains all the values needed for computation, q_bit computation is carried out on each quantum gate individually and is finished by writing the results back to global memory.
Thread 0 computes the base memory idx value necessary for reading from global memory in the following kernel function. The rest of the threads generate their own Q bit vector after waiting for that computation to complete. Oring base mem index and Q bit vector perform the function mem index to find the value or location of a memory index.

In essence, each thread reads two values from global memory using its own mem index and the complement's mem index before storing them in shared memory.

32 threads in all, each reading 2 values.

Hence, 64 values are read from shared memory and stored there.

```
__shared__ float A_shared[64];
```

Because it is a need for all threads, base mem index is also specified as a shared property.

```
__shared__ int base_memory_idx ;
```

As soon as all appropriate values are kept in shared memory, the following threads synchronize.

On each bit, the next quantum gate computation is run. Before beginning the subsequent computation, threads synchronize after each computation.

Value is written back to memory once all threads have completed their six single bit quantum operations. The write index is located using the memory_idx that each thread calculated above.

```
int threadsPerBlock = 32;
int blocksPerGrid =(number_elem>>1 + threadsPerBlock - 1) / threadsPerBlock;
CallKernelFunction(d_U,d_A,d_B,d_Q,threadsPerBlock,blocksPerGrid,number_elem);
```

If there are 65536 elements, then the computation can be done with only 65536/2 threads. Hence, the number of blocks in each grid is cut in half because we used number_elem>>1 rather than number_elem.

```
__global__ void
quamsim(const float *U, const float *A, float *B, const int *Q, int number_elem)
{    int i = blockDim.x * blockIdx.x + threadIdx.x;
    __shared__ int base_memory_idx ;
    __shared__ float A_shared[64];
    int memory_idx;
    int Q_bit_vector=0;
    int Q_bit_mask ;
    if(threadIdx.x==0){
        base_memory_idx = blockIdx.x;
        for(int it=0;it<6;it++)
           base_memory_idx = ((base_memory_idx >> Q[it])<<(Q[it]+1)) | ((1<< Q[it])-1)& base_memory_idx;     }
    __syncthreads();
    Q_bit_mask = threadIdx.x;
    for(int it=0;it<5;it++){
      if((Q_bit_mask & 1)== 1)
       Q_bit_vector =  Q_bit_vector | 1<<Q[it];
      else
       Q_bit_vector =  Q_bit_vector & ~(1<<Q[it]);
      Q_bit_mask = Q_bit_mask >>1;    }
    memory_idx = base_memory_idx | Q_bit_vector;
    A_shared[threadIdx.x] = A[memory_idx];
    A_shared[threadIdx.x|1<<5] = A[memory_idx | 1<<Q[5]];
    __syncthreads();
    if (i < number_elem)
    {        for(int it=0;it<6;it++){
           int index = ((threadIdx.x >> it)<<(it+1)) | ((1<<it)-1)& threadIdx.x;
           float temp =A_shared[index] ;
           A_shared[index]= (U[it*4]*A_shared[index]) + (U[it*4+1]*A_shared[index ^ (1<<it)]);
           A_shared[index ^ (1<<it)] =  (U[it*4+2]*temp) + (U[it*4+3]*A_shared[index ^ (1<<it)]);
         __syncthreads();
        }
        B[memory_idx]=A_shared[threadIdx.x];
        B[memory_idx| 1<<Q[5]] = A_shared[threadIdx.x|1<<5];          }}
```

TASK 3

Thread coarsening is implemented in task 3. Due to thread coarsening, one thread computes two elements in task 2 but four elements are computed in task 3.

In essence, I created q-bit vectors for each of the 16 threads and ORed base mem index with q-bit vectors to create the global memory address. Effective reading and writing of four memory pieces is accomplished by manipulating the mem index determined for each thread.

For four components in task3, the kernel function computes six single-q_bit quantum gates to six separate qubits for each thread (in task2 each thread computes only 2 elements).

In this method, thread coarsening does not result in a halving of resources or TBs.

In hydra, task 3's effective number of TBs is 1/4 of task 1's and 1/2 of task 2's effective number of threads. Each TB now has half as many threads, or 16. However, I kept the thread count at 32 because that is the minimum number of threads needed for TB in the GPGPU simulator. If I set the TB size to 16, I receive a configuration error.

```
__global__ void
quamsim(const float *U, const float *A, float *B, const int *Q, int number_elem)
{    int i = blockDim.x * blockIdx.x + threadIdx.x;
    __shared__ int base_memory_idx ;
    __shared__ float A_shared[64];
    int memory_idx;
    int Q_bit_vector=0;
    int Q_bit_mask ;
    if(threadIdx.x==0){
        base_memory_idx = blockIdx.x;
        for(int it=0;it<6;it++)
           base_memory_idx = ((base_memory_idx >> Q[it])<<(Q[it]+1)) | ((1<< Q[it])-1)& base_memory_idx;     }
 if(threadIdx.x <16){
    __syncthreads();
    Q_bit_mask = threadIdx.x;
    for(int it=0;it<4;it++){
      if((Q_bit_mask & 1)== 1)
       Q_bit_vector =  Q_bit_vector | 1<<Q[it];
      else
       Q_bit_vector =  Q_bit_vector & ~(1<<Q[it]);
      Q_bit_mask = Q_bit_mask >>1;    }
    memory_idx = base_memory_idx | Q_bit_vector;
    A_shared[threadIdx.x & ~(1<<5) & ~(1<<4)] = A[memory_idx & ~(1<<Q[5]) & ~(1<<Q[4]) ];
    A_shared[threadIdx.x & ~(1<<5) |  (1<<4)] = A[memory_idx & ~(1<<Q[5]) |  (1<<Q[4]) ];
    A_shared[threadIdx.x |  (1<<5) & ~(1<<4)] = A[memory_idx |  (1<<Q[5]) & ~(1<<Q[4]) ];
    A_shared[threadIdx.x |  (1<<5) |  (1<<4)] = A[memory_idx |  (1<<Q[5]) |  (1<<Q[4]) ];
    __syncthreads();
    if (i < number_elem)
    {        for(int it=0;it<6;it++){
           int index1 = (((2*threadIdx.x) >> it)<<(it+1)) | ((1<<it)-1)& (2*threadIdx.x);
           int index2 = (((2*threadIdx.x+1) >> it)<<(it+1)) | ((1<<it)-1)& (2*threadIdx.x+1);
           float temp1 =A_shared[index1] ;
           float temp2 =A_shared[index2] ;
           A_shared[index1]= (U[it*4]*A_shared[index1]) + (U[it*4+1]*A_shared[index1 ^ (1<<it)]);
           A_shared[index1 ^ (1<<it)] =  (U[it*4+2]*temp1) + (U[it*4+3]*A_shared[index1 ^ (1<<it)]);
           A_shared[index2]= (U[it*4]*A_shared[index2]) + (U[it*4+1]*A_shared[index2 ^ (1<<it)]);
           A_shared[index2 ^ (1<<it)] =  (U[it*4+2]*temp2) + (U[it*4+3]*A_shared[index2 ^ (1<<it)]);
         __syncthreads();        }
       B[memory_idx & ~(1<<Q[5]) & ~(1<<Q[4]) ]  =A_shared[threadIdx.x & ~(1<<5) & ~(1<<4)] ;
       B[memory_idx & ~(1<<Q[5]) |  (1<<Q[4]) ]  =A_shared[threadIdx.x & ~(1<<5) |  (1<<4)]  ;
       B[memory_idx |  (1<<Q[5]) & ~(1<<Q[4]) ]  =A_shared[threadIdx.x |  (1<<5) & ~(1<<4)]  ;
       B[memory_idx |  (1<<Q[5]) |  (1<<Q[4]) ]  =A_shared[threadIdx.x |  (1<<5) |  (1<<4)]  ;          } }}
```

Analysis

Number of global memory reads are represented by gpgpu_n_mem_read_global .
Number of global memory writes are represented by gpgpu_n_mem_write_global.

with out shared memory:
gpgpu_n_mem_read_global = 49552 ,912
gpgpu_n_mem_write_global = 65536 ,1024
with shared memory:
gpgpu_n_mem_read_global = 33082 ,592
gpgpu_n_mem_write_global = 16384, 256

Every kernel call in task 1 reads from and writes to global memory. Because we are already storing the values from indices with n-k identical bits, task 2 optimizes this. As shared memory already contains all the values needed for computation, qubit computation is carried out on each quantum gate individually and is finished by writing the results back to global memory. Only shared memory locations are used for the intermediate writes and reads, which lowers the overall global write and read counts.