ECE 786
Programming Assignment 3
Part B

Shawn Salekin

Task 1

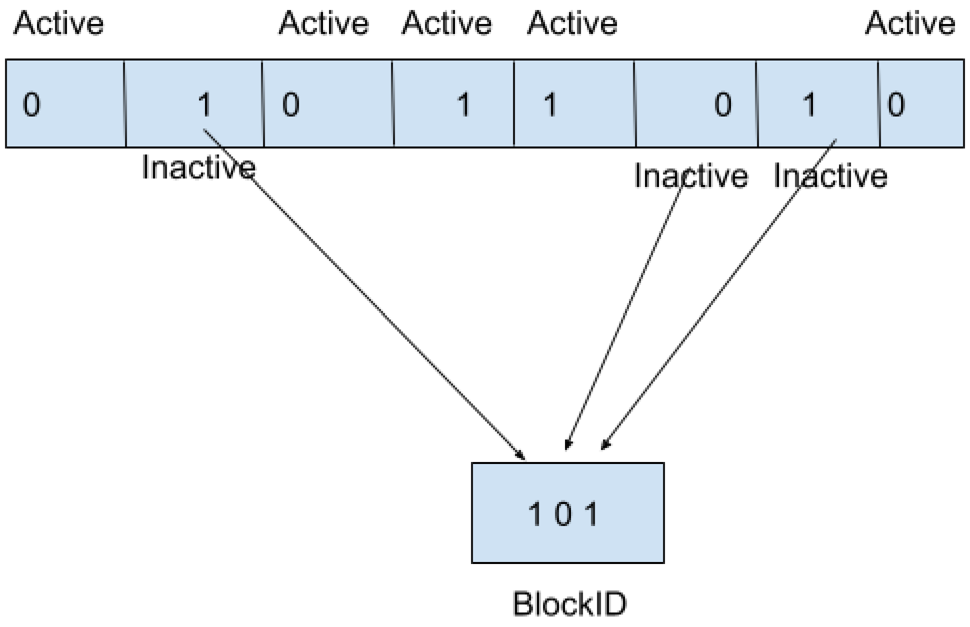
As mentioned in the project specification, this part is really a wrapper around the code developed in PA1. The CUDA kernel function takes two arrays, A and C for input and output storage respectively. Then we take advantage of the “global” threadID to identify which piece of data this thread should perform operations on. Then we calculate the other thread that will be paired with this thread, using bitwise operations. Once we identified both threads, we do the matrix multiplication and store the data in the output array.

The kernel code is run within a while loop that selects different bits that to apply the quantum gate on. At the end of every iteration, we copy over the output array into the input array so the next iteration can perform its operation on the state calculated in the previous loop.

Task 2

This was much more complicated compared to Task1. The crux of the ShareMem method was to map the elements to the correct thread block. We use the active qubits and inactive qubits to perform the grouping.

To correctly map an element in the quantum circuit, we use the binary representation of that number. We are given the active qubits as input, so we extrapolate the inactive qubits using the active sets. Once we have the inactive bits, we generate a number using the inactive bits. This essentially serves as the block that this element(thread) should belong to.



The above figure shows how we map each element of the state vector (and the thread) is mapped into the correct 2^6-sized fraction, identified by BlockID.

In our implementation, we use a second array, B, to store this data (element to BlockID mapping). In the kernel function, we move data from global to shared memory to perform calculations on based on this mapping. Once the fractioning is done, we apply gates 1 to 6 on *active* qubit 0 to 5 (in that order). After that, the data is put back into the global result array, using the same mechanism as when we copied data from the global memory.

Task3

We implemented thread coarsening by taking the code on Task 2 and reducing the number of threads fourfold. So instead of using 64 threads, we use 16 threads and make each thread do the job of 4 threads. Using modulo arithmetic, we map 4 elements to a single thread, like this:

```
tid == (m % 16)
```

This maps the elements that would have been handled by 3 other threads to the current thread. The rest of the code stays the same.

Comparison

We used the quantum circuit with size 10 as the sample data to perform this calculation. The configuration used was SM7_TITANV.

Task1	**Task2**

gpgpu_n_mem_read_global = 49152
gpgpu_n_mem_write_global = 65536
Total = 114688
	gpgpu_n_mem_read_global = 2352
	gpgpu_n_mem_write_global = 2048
	Total = 4400

The number of global memory accesses for Task1 is 26x higher than for Task2. We suspect the Task 2 number would have been even lower if we had a more optimized *threadID ⇔ BlockID* mapping mechanism. The current code iterate over all the elements in a global array (B) to identify an element that is mapped to the current block. In a more optimized implementation, which can reduce the number of comparisons to the number of elements, the global memory access would have been roughly 2x the number of elements.