Profiling command:

```
# Profling command
# sudo env PATH=/usr/local/cuda-11.8/bin:$PATH /usr/local/cuda-11.8/bin/nv-nsight-cu-cli --target-processes all --set full --log-file cnn_report ./run_cnn.sh
```

Parallelization strategies:

1. Data distribution:
    a. The parallelization strategies I adopted was to divide the output (I, h, w) in to a three-dimensional grid, where each block is computes a 16 * 16 (x and y dimension of each block) tile of output elements for a particular output channel I (Total output size equals to 112 * 112 after max pooling). This way, each block only has to store a 16 * 16 output array and load it back to the host without any cross-block interferences.
    b. In terms of data distribution, each block will have to get its own tile and weights for each input channel.
2. Computation:
    a. In terms of computation, each thread is responsible for computing 4 neighboring pooling element in the output size of 224 * 224 across 256 different input channels (The z dimension of each block). Then each of the four elements go through adding bias, relu and then pooling. The result is written back to the output array.
    b. One important reasoning for choosing the 16 * 16 tile is because I will be able to fit four blocks of 256 threads, getting 1024 threads in total, taking full advantage of the thread limit. Furthermore, this approach requires only around 6 KB of shared memory per block, preventing me from ever reaching the 64 KB shared memory boundary while maintaining a 100% occupancy in profiling.

Optimizations:

1. Weights in shared memory:
    a. This actually reduces my performance by about 30 GFlops, I believe the reason is because the loaded weight is reused by all 256 threads.
        i. Original code: 256 threads each load 5 * 5 weights for 256 input channels.
            1. 1638400 Total global reads.
        ii. Shared memory: 256 threads each load 5 * 5 weights for 256 input channels, then save into shared memory and reuse shared memory cache.
            1. 6400 Total global reads (256 times reduction)
            2. Shared memory writes latency (6400 writes)
            3. Shared memory read latency.
2. Inputs in shared memory:
    a. Performance not tested yet
        i. Original code: 256 threads, each thread 256 * 5 *5 iterations, 4 data loaded each iteration.
            1. 6553600 global reads
        ii. Shared memory: 256 threads, each read 256 elements.
            1. 65536 global reads
            2. Shared memory write latency (65536 writes)
            3. 5*5*4=100 times reduction
        iii. Analysis on reasoning for worsened performance:
            1. Using profiler, the shared memory weights turned the Sectors/half-warp request from 6.1 (195 B) to 9.3 (297 B)
            2. Occupancy dropped from 99% to 50% (32 Warps into 16 Warps)
                a. The reason for drop in occupancy happen because the theoretical maximum number of concurrent warps per SM is 32. With weight 16/32, without weight 32/32.
                b. Without weight in shared memory, total shared memory usage is 5 KB(Input tile), Total shared memory is 64 KB, thus can tolerate 12 thread blocks per SM (64/5=12). Since each thread blocks is 256 threads, 8 warps. There can be in total of 96 warps possible, but only 32 concurrent warps. Thus, all warps can concurrently execute 32/32=100%.
                c. With weights in shared memory, total shared memory usage is 5 KB(Input tile) + 30 KB (Weights), Total shared memory is 64 KB, thus can tolerate 2 thread blocks per SM ((64-5)/30=2). Since each thread blocks is 256 threads, 8 warps. Max warps per SM is 16 / 32 theoretical maximum. Thus 50% occupancy

    d. Having 16 warps reduces the number of outstanding memory fetches possible (From 16* 32 = 512 loads to 16 * 16 = 256) (16 outstanding global loads per warp because of hardware limitation on 16 load buffer)
3. Loading weights into registers:
    a. Assuming 25 weight elements and 256 channels.
    b. Assuming 400 cycles global read, 30 cycles shared memory read
    c. Assuming each syncthreads of 30 cycles latency.
    d. Shared memory:
        i. 25 * 400 + 30 * 25 = 10750 cycles
    e. Register usage (Syncthreads overhead):
        i. 25 * 400 + 25  + 30 = 10055 cycles
    f. Saves 725 cycles per input channel, in total saves 725 * 256 = 185600 cycles read.
    g. Result:
        i. This approach actually costed me more performance (70 GFlops) because extra synchronization needed between warps
4. Loading individual weights on the fly:
    a. This approach helped me gained more than 200 GFlops of performance because this approach reduced my block shared memory back to around 10 KB, and each SM now can fit 4 blocks with a 100% occupancy, before it was only 50% occupancy because each block took 30 KB of shared memory.
5. Loop fusion to have each thread computes output of 4 neighboring pooling element. This avoids further loops.
6. Parallelization: taking advantage of the excessive number of threads per block to parallelize the computation for each i, h , w elements with different threads inside each block.

Block Size: (7, 7, 256)

Block Dimension: (16, 16, 1)

Number of SM: 40

Number of cores per SM: 64

Total cores = 2560

Yes. 16 * 16 = 256 = 8 warps, there is no intra-warp underutilization. 256 * 4 = 1024 threads, by having 1024 threads, we take full advantage of threads limit, while at the same time we are taking advtange of the 4 processing block per SM.

Actions:

1. Load weights for 256 inputs in two phases. This way 4 blocks of 32 concurrent warps can be fully utilized.
2. Reduce the 6.1 sector per request from without weight.
    a. Fixed
3. Use Shared memory for inputs:
    a. Done
    b. Increased about 100 GFlops