ECE 277, WINTER 2021
GPU Programming
Department of Electrical and Computer Engineering
University of California, San Diego

LAB 3: Reinforcement learning: Q-learning (Multi-Agent, CUDA multithreads)

This lab requires to design multiple agents to interact with the environment using the reinforcement learning algorithm in Figure 1. Specifically, you need to design multiple agents to maximize rewards from the mine game environment using Q-learning. The agents should interact with the given mine game environment in Figure 2.

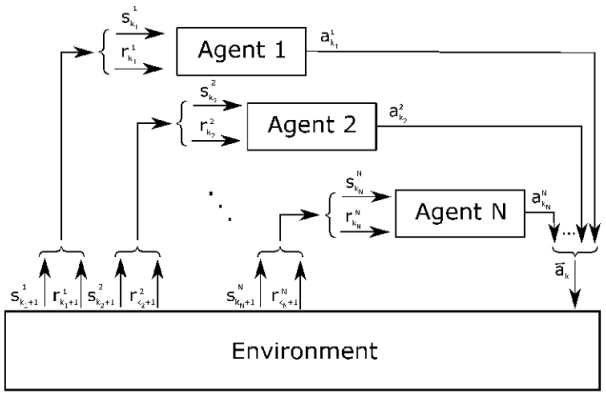

Figure 1: Parallel reinforcement learning.

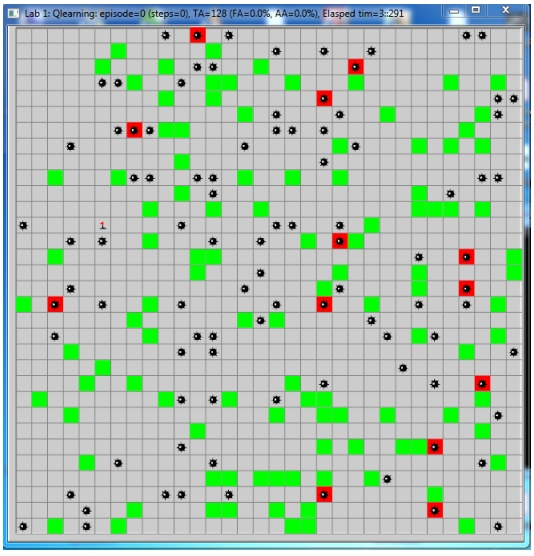

Figure 2: 32x32 mine game environment.

- The number of agents: 128
- Action: right:0, down:1, left:2, up:3
- Reward: flag: +1, mine: -1, otherwise: 0
- State: (x, y) current position of an agent in the coordinator of (0,0) at the top-left corner
- Every episode restarts after the number of active agents reaches less than 20%.
- You need to maintain active agents in each episode. **You should prevent inactive agents from taking an action and updating the Q-table** since environment returns wrong rewards to inactive agents.
- Initial states of environment are randomized every episode.
- Environment elements such as mine distributions and a flag position are randomized every game.
- $Agent_i$ should return $action[Agent_i]$ to the corresponding current state, $cstate[Agent_i]$.
- $Agent_i$ should update a centralized Q table along with a current state, $cstate[Agent_i]$, a next state, $nstate[Agent_i]$, and a reward, $rewards[Agent_i]$.
- Share a single centralized Q table for all agents (a centralized learning and decentralized execution approach).
- You should initialize the Q table using a multithreads kernel instead of using CPU functions such as "cudaMemSet".
- In the learning environment, TA means the total number of agents, FA indicates a percentile of agents catching flag, and AA shows a percentile of active agents in a current episode.

You should not modify any given codes except CMakeLists to add your codes. You only need to add your agent code to the lab project.

You have to use CUDA to program a multi-agent reinforcement learning algorithm.

Interface pointers of all the extern functions are allocated to the Device (GPU) memory (not CPU memory) The below function is an informative RL environment routine to show when and how agent functions are called

```
extern void agent_init();
extern void agent_init_episode();
extern float agent_adjustepsilon();
extern short* agent_action(int2* cstate);
extern void agent_update(int2* cstate, int2* nstate, float* rewards);

int qlearningCls::learning(int *board, unsigned int &episode, unsigned int &steps)
    if (m_episode == 0 && m_steps==0) {// only for first episode
        env.reset(m_sid);
```

```
        agent_init(); // clear action + initQ table + self initialization
    }else {
        active_agent = checkstatus(board, env.m_state, flag_agent);

        if (m_newepisode) {
            env.reset(m_sid);
            agent_init_episode(); // set all agents in active status
            float epsilon = agent_adjustepsilon(); // adjust epsilon
            m_steps = 0;
            printf("EP=%4d, eps=%4.3f\n", m_episode, epsilon);
            m_episode++;
        }else {
            short* action = agent_action(env.d_state[m_sid]);
            env.step(m_sid, action);
            agent_update(env.d_state[m_sid], env.d_state[m_sid ^ 1], env.d_reward);

            m_sid ^= 1;
            episode = m_episode;
            steps = m_steps;
        }}
    m_steps++;
    env.render(board, m_sid);
    return m_newepisode;
```

The provided parameters are just for reference.

$$\gamma = 0.9, \quad \alpha = 0.1, \quad 0.1 \leq \epsilon - \delta\epsilon \leq 1.0, \quad \delta\epsilon = 0.001$$

Submit only your agent files into the assignment.
Programming language: CUDA