SystemVerilog Hardware Acceleration for Q-Learning

Transcript Date: April 19, 2025

This document captures the technical conversation between a user and ChatGPT regarding the profiling, optimization, and hardware implementation of a Python-based Q-learning algorithm applied to a deterministic FrozenLake environment. The objective was to identify computational bottlenecks in the code and provide equivalent SystemVerilog modules to accelerate the Q-learning process in hardware, suitable for FPGA or ASIC deployment.

1. Repository Analysis

The GitHub repository analyzed is: https://github.com/ronanmmurphy/Q-Learning-Algorithm

It implements a Q-learning agent trained to solve a deterministic version of FrozenLake - a 5x5 grid world environment. The agent navigates from the top-left (0,0) to the bottom-right (4,4) while avoiding holes. It employs:

- A State class for managing positions, hole detection, and rewards.
- An Agent class that performs Q-learning with a Q-table structured as nested Python dictionaries.
- Epsilon-greedy policy for exploration-exploitation (e = 0.1).
- Training for 10,000 episodes.

2. Identified Bottlenecks (with Line Numbers)

1. Q-table Dictionary (Lines 110, 176-194): Nested dictionaries (Q[state][action]) introduce hashing overhead and poor cache performance. Use fixed-size arrays for fast access.

2. State Object Recreation (Line 166): Re-instantiating State class each episode leads to memory

churn. Reuse one instance with a reset() method.

3. Linear Hole Search (Lines 31, 46): Repeated list scans for hole detection can be improved by using a set for constant-time lookup.

4. Random Action Sampling (Line 178): Frequent np.random.uniform and random.choice calls are relatively slow in tight loops. Consider vectorized pre-generation.

3. Hardware Proposals

3.1 Q-table in SRAM:

- Fixed-size 25x4 Q-table stored in dual-port SRAM.
- Indexed by (state, action) pair using 5-bit and 2-bit indices.
- Read/write supported via constant-time memory access.
- Updates computed in a small ALU: Q <- (1 - alpha) * Q + alpha * (r + gamma * max Q').

3.2 State Tracker FSM:

- FSM maintains x and y using 3-bit registers.
- Hole and goal detection via combinational logic or LUT.
- Uses an action decoder (2 bits) to trigger grid movement.
- Reset restores state to (0,0); edge-limited actions check boundaries.

These modules are suitable for synthesis in FPGA or ASIC environments.

4. SystemVerilog Implementations

```
// Q-Table Hardware: SRAM-Based Implementation for 25 states × 4 actions
// Fixed-point 16-bit Q-values, dual-port SRAM for simultaneous read/write

module QTable #(parameter STATE_BITS = 5, // log2(25) = 5
                parameter ACTION_BITS = 2, // log2(4) = 2
                parameter Q_WIDTH = 16)(
    input logic clk,
    input logic rst,
```

```
    // Read interface
    input logic [STATE_BITS-1:0] read_state,
    input logic [ACTION_BITS-1:0] read_action,
    output logic [Q_WIDTH-1:0] q_read_data,

    // Write interface
    input logic we, // write enable
    input logic [STATE_BITS-1:0] write_state,
    input logic [ACTION_BITS-1:0] write_action,
    input logic [Q_WIDTH-1:0] q_write_data
);

    // 25 states x 4 actions
    localparam ENTRY_COUNT = 25 * 4;
    logic [Q_WIDTH-1:0] q_table [0:ENTRY_COUNT-1];

    logic [6:0] read_addr, write_addr;
    assign read_addr  = {read_state, read_action};
    assign write_addr = {write_state, write_action};

    // Read
    assign q_read_data = q_table[read_addr];

    // Write
    always_ff @(posedge clk) begin
        if (we) begin
            q_table[write_addr] <= q_write_data;
        end
    end

endmodule
```

```
// State Tracker FSM for 5x5 Grid (FrozenLake environment)
// Handles (x, y) position, obstacle detection, and reset logic

module StateFSM #(parameter X_BITS = 3,  // log2(5) = 3
                  parameter Y_BITS = 3)(
    input logic clk,
    input logic rst,
    input logic [1:0] action,  // 2-bit input: 00=up, 01=down, 10=left, 11=right
    input logic step,         // step trigger (advance 1 time step)

    output logic [X_BITS-1:0] pos_x,
    output logic [Y_BITS-1:0] pos_y,
    output logic is_hole,
    output logic is_goal
);

    // Internal position registers
    logic [X_BITS-1:0] x;
    logic [Y_BITS-1:0] y;

    // Hole ROM: hardcoded 4 positions
    function logic is_hole_pos(input logic [2:0] x_in, input logic [2:0] y_in);
        return (x_in == 1 && y_in == 0) ||
               (x_in == 3 && y_in == 1) ||
               (x_in == 4 && y_in == 2) ||
               (x_in == 1 && y_in == 3);
    endfunction

    function logic is_goal_pos(input logic [2:0] x_in, input logic [2:0] y_in);
        return (x_in == 4 && y_in == 4);
    endfunction

    // Position update
    always_ff @(posedge clk or posedge rst) begin
        if (rst) begin
            x <= 0;
            y <= 0;
        end else if (step) begin
            case (action)
                2'b00: begin
                    if (y > 0) y <= y - 1;
                    else y <= y;
                end
                2'b01: begin
                    if (y < 4) y <= y + 1;
                    else y <= y;
                end
                2'b10: begin
                    if (x > 0) x <= x - 1;
                    else x <= x;
                end
                2'b11: begin
                    if (x < 4) x <= x + 1;
                    else x <= x;
                end
                default: begin
                    x <= x;
                    y <= y;
                end
            endcase
        end
    end

    assign pos_x = x;
```

```
    assign pos_y = y;
    assign is_hole = is_hole_pos(x, y);
    assign is_goal = is_goal_pos(x, y);

endmodule
```