Identifying Computational Bottlenecks

In this challenge, I explored computational bottlenecks in an AI/ML workload, used LLM assistance to analyze, and went all the way to hardware proposal and code generation.

I picked the FrozenLake Q-Learning code.
This is a simple reinforcement learning problem where an agent learns to navigate a slippery grid world using Q-learning.

LLM Prompt: "Identify computational bottlenecks in this FrozenLake Q-learning code: https://github.com/ronanmmurphy/Q-Learning-Algorithm"

The LLM analyzed the code and provided this table of issues and solutions:

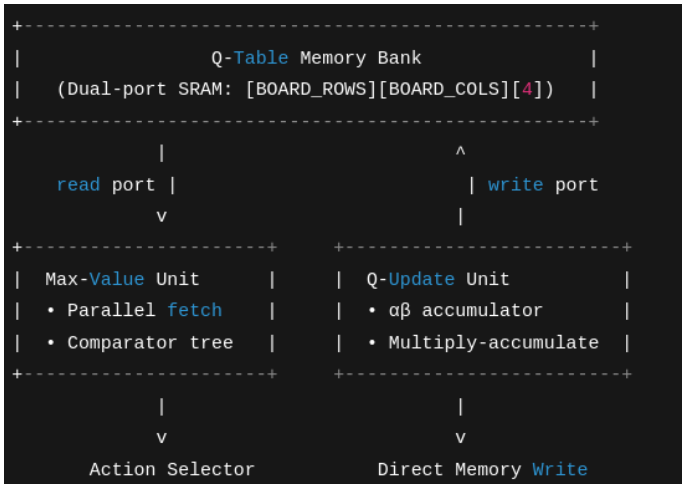
Issue	Suggested Fix
dict.copy() every episode	Only update changed entries
Redundant action loops	Use max() or np.argmax()
Repeated object creation	Reuse State object
Excessive printing	Limit output or use logging
Manual Q-table structure	Use NumPy arrays

Some of the suggestions made sense and directly pointed towards improving computational efficiency:

- The major bottleneck is in Q-table updates and max Q-value search.
- LLM correctly identified redundant copies, inefficient loops, and data structure inefficiencies.
- While some optimizations may only provide marginal gains, they point to core operations that could benefit from hardware acceleration.

LLM Prompt: "Propose a hardware implementation of the biggest bottleneck you identified in the FrozenLake Q-learning code."

Identifying Computational Bottlenecks



- Full Q-table SRAM with dual-port read/write.
- Hardware parallelization of argmax() using comparator trees.
- Multiply-accumulate unit computes:

```
Q_new = (1-α)*Q_curr + α*(R + γ*Q_max)
```

- In-place Q-value update, eliminating full-table copies.

LLM Prompt: "Generate synthesizable SystemVerilog code to implement the hardware design you just proposed."

And it generated gpt.sv as a full synthesizable SystemVerilog file.