Bootstrapping the main project!

I started this challenge by deeply exploring the Viterbi algorithm, which I had selected for my accelerator project. My first step was to understand the algorithm's structure, how it computes forward paths, keeps track of back-pointers, and eventually backtracks to decode the most probable state sequence.

I used both online references and LLM (ChatGPT) assistance to:

- Break down the core computations involved (maximization, addition, state transitions).
- Understand where the computation bottlenecks typically lie.
- Visualize data dependencies and control flow.

I realized that Viterbi involves both:

- Data-dependent control flow (especially during backtracking).
- Highly parallelizable arithmetic (during forward path updates).

For Profiling,
I wrote the Viterbi code in Python, which allowed me to leverage rich profiling tools.

- Used cProfile to profile the full execution time and function-wise breakdown.
- Used line_profiler to get fine-grained time breakdown at line level.
- Used memory_profiler to check memory usage at each stage.

```
python3 -m cProfile -o output.prof my_viterbi.py
snakeviz output.prof
```

I then used snakeviz to visualize the profiling data interactively.

Through this:

- I identified that forward_log() and viterbi() functions consumed the majority of execution time.
- Most time was spent inside loops performing repeated arithmetic: max() operations, additions, and lookups in matrices (logA, logB, logC).
- Memory usage was minimal, confirming the algorithm is compute-bound, not memory-bound.

<u>Bytecode Analysis:</u>
With LLM guidance, I compiled the code to bytecode using:

```
import py_compile
py_compile.compile('my_viterbi.py')
```

Then I disassembled the .pyc file using Python's built-in dis module:

```
import dis
dis.dis(my_viterbi.forward_log)
```

This showed me:

- Python uses a simple stack-based virtual machine.

Bootstrapping the main project!

- A high volume of BINARY_ADD, COMPARE_OP, LOAD_FAST, and CALL_FUNCTION instructions.
- The VM executes many instructions for every arithmetic operation, making Python inefficient for this type of numerical workload.

I also wrote and ran a helper script (count_instructions.py) to count and analyze the instruction distribution. This showed me how much room there is for hardware acceleration.

<u>Visualization:</u>

I attempted various tools that LLM suggested:

- ast module (Python's Abstract Syntax Tree) to parse the code.
- NetworkX to build simplified data flow graphs.
- Explored PYCFG and StaticFG for control flow graph extraction (though I didn't fully use them due to complexity for this case).

The clear separation between initialization, forward recursion, and backtracking phases.
Forward phase being embarrassingly parallel (per state per time step).

<u>What I understood?</u>
Based on the analysis, I identified:

- Forward phase is highly parallelizable.
- Backtracking is sequential but lightweight.
- A systolic array structure would efficiently compute the forward recursion (max + add operations).
- Hardware accelerator focus: processing element (PE) design for the forward phase.

During this stage, I also experimented with:

- Cocotb: Learned Python-based testbenches for hardware verification (used later during hardware prototyping).
- Verilog implementations: Gradually built up RTL designs for the processing element (viterbi_pe) and full decoder (viterbi_top).
- Profiling-guided co-design: Used Python profiling data to inform hardware block diagram decisions.

Prompts which I used:

"Generate a cProfile script to profile my Viterbi function"
"How do I analyze Python bytecode?"
"Write code to count Python bytecode instructions"
"How to visualize Python AST"

Bootstrapping the main project!

"Suggest control/data flow analysis tools for Python"

"Which parts of Viterbi algorithm benefit from hardware acceleration?"

"Can you generate Verilog for systolic processing element for Viterbi forward pass?"