# Updated Cursor AI Prompt for LangOne Project Setup

I am building a new programming language called **LangOne**. Please help me set up a new Rust project (Cargo package) named langone that will serve as the compiler/interpreter for this language. In addition to the basic compiler structure, **LangOne must be optimized for RAM, GPU, and other hardware resources from the very beginning.** This means the project should include modules and placeholders dedicated to **memory management, GPU resource management, and compile‑time resource analysis**, enabling the language to run efficiently on a wide range of devices—from small IoT boards to multi‑GPU servers.

## Requirements and Structure

1. **Use Rust for Implementation**  
   Set up a Rust project that uses the inkwell crate for LLVM code generation, along with any other crates needed (e.g., anyhow for error handling or clap for CLI parsing). Ensure Cargo.toml includes inkwell with the appropriate LLVM feature enabled (e.g., llvm16-0 or a suitable version).
2. **Core Compiler Components**  
   Create the following modules/files, each with basic struct or function definitions and // TODO comments for future development:
3. **main.rs** – Entry point that parses command‑line arguments (using std::env or clap) and calls the compiler pipeline (lexer → parser → codegen). For now, just demonstrate the flow and print a message or dummy output.
4. **lexer.rs** – Define a Token enum and a stub function lex() that returns a list of tokens. Include TODOs for lexical analysis logic.
5. **parser.rs** – Define a Parser struct or function and a stub parse() that produces an AST. Include TODOs for parsing logic.
6. **ast.rs** – Define simple AST node structs or enums (e.g., for expressions, statements) with derived traits (Debug, Clone). These are placeholders for the abstract syntax tree data structures.
7. **codegen.rs** – Initialize an LLVM context and module using inkwell. Provide a placeholder codegen() function that will traverse the AST and generate IR. Include a TODO note for implementing actual code generation.
8. **errors.rs** (optional) – Define error types and diagnostic utilities. You may include a basic error struct and an error reporting function with TODOs for richer diagnostics.
9. **Memory and Resource Optimization Module**  
   To ensure that LangOne optimizes RAM, GPU, and other hardware resources from the start, add a new module **resource.rs** with the following elements:
10. **Memory Pooling and Allocators**: Define a MemoryPool struct and associated functions for pooling and reusing memory buffers. Include a TODO to implement arenas or bump allocators that reduce heap fragmentation and manage memory efficiently.
11. **GPU Memory Management**: Define a GPUMemoryManager struct with stub methods to allocate, reuse, and release GPU memory buffers. Include TODOs for mixed‑precision control (e.g., FP16 vs. FP32) and for implementing automatic device placement of tensors.
12. **Resource Budgets and Monitoring**: Create a ResourceBudget struct or type that represents memory and GPU budgets for a program. Add stub functions to set and enforce these budgets, and placeholders for runtime monitoring that will record real‑time memory and GPU usage statistics.
13. **Compile‑Time Resource Analysis**: Include a placeholder function analyze\_resources() that the compiler will call to estimate stack and heap usage, and identify potentially expensive allocations. Document that this function will warn developers about high memory or GPU usage at compile time.
14. **Integration Points**: In comments, explain how this resource module will integrate with the compiler pipeline—for example, by sharing the budget data with the codegen stage and by exposing an API for the runtime to monitor resource usage.
15. **Wiring Everything Together**
16. Reference all modules using mod statements in main.rs or lib.rs so they compile as one project.
17. In Cargo.toml, besides inkwell, add any crates that may help with future resource optimization (like profiling for runtime metrics or clap for CLI parsing).
18. Use // TODO comments liberally to mark where functionality will be filled in later.
19. **Compile‑Ready Scaffold**  
    Ensure that the generated project compiles successfully with cargo build, even though many functions only contain stubs. This gives us a solid scaffold to iterate on as we implement the full compiler, resource management, and other language features.

Please create the project structure and files as described, populating them with the requested placeholders and comments. Once complete, provide the output and ensure that it builds without errors.