Performance Benchmarking Methodology

Benchmark Categories and Rationale

I employed a multi-dimensional benchmark approach designed to capture the performance characteristics of Rust and C across different domains. The selected benchmark categories were chosen to provide a holistic assessment of language performance:

1. **Memory Management Performance**
   This category evaluates the efficiency and overhead associated with dynamic memory allocation and deallocation. The implementation of this benchmark is in **`malloc-c`** and **`malloc-rs`** for C and Rust, respectively.

2. **Computational Efficiency**
   Computational benchmarks assess the raw processing capabilities of both languages across different challenges:
   - Numerical computing performance through matrix-matrix multiplication in **`matrix-c`** and **`matrix-rs`**
   - Algorithmic complexity handling through different sorting algorithms in **`sort-c`** and **`sort-rs`**

3. **Concurrency Performance**
   Given the increasing importance of parallel computing, this category examines thread creation and management overhead in **`concurrency-c`** and **`concurrency-rs`**.

Experimental Setup and Methodology

Hardware Configuration

My benchmark infrastructure utilized a laptop computer with the following specifications:

- **Processor:** AMD Ryzen 5 5500U
  - 6 physical cores (12 threads)
  - Base clock: 2.1 GHz
  - Turbo boost up to 4.0 GHz
- **RAM:** 16 GB DDR4 2400 MT/s
- **Storage:** Toshiba KXG5AZNV512G NVMe SSD
- **Operating System:** Linux 6.6.63 (Kernel), NixOS distribution, with performance governor enabled

Compiler Configuration

To ensure fair and representative comparisons, I employed the following compiler versions:

- C Compiler: Clang (LLVM C frontend) version 18.1.8
  - Configured with full optimization support
- Rust Compiler: Rust 1.83.0 Stable Release
  - Default stable channel
  - Configured with release optimization profile

Benchmark Optimization and Configuration

- Optimization Level: `-O3` for both languages
  - Enables maximum compiler optimizations
  - Allows aggressive inlining and code generation
- Compilation Flags:
  - C: `-march=native -mtune=native -flto`
  - Rust: `-C target-cpu=native lto=true`

Reproducibility and Transparency

All benchmark source code, raw data, and detailed methodology are publicly available in the associated GitHub repository to ensure complete transparency and independent verification.