Consolidated Knowledge Base for ohhmm/openmind

Project Structure

The ohhmm/openmind repository is organized into the following main directories:

- `/OpenMind` - Main application code
- `/omnn` - Core libraries and modules:
  - `/math` - Mathematical operations and types
  - `/extrapolator` - Matrix operations and extrapolation
  - `/logic` - Logical operations
  - `/rt` - Runtime and neural network components
  - `/ct` - Compile-time utilities
  - `/storage` - Caching and persistence
- `/Examples` - Sample implementations
- `/lang` - Language processing

Core Systems

1. Goal Management System

- Central `Mind` class coordinates goals and generators
- Asynchronous goal processing with state machines
- Multiple generator types (GeneralGoalGenerator, SingletonGoalGenerator, IdleTimeGoalGenerator)
- Uses facilities for reusable operations

2. Mathematical Framework (`omnn/math`)

- Extensive type system for mathematical operations
- Support for variables, equations, matrices
- Operations: arithmetic, logarithms, exponentials, fractions
- Constants (pi, e, i) and special values
- Heavy template usage for generic operations

3. Neural Network Components (`omnn/rt`)

- Async neuron implementation
- Task queues and parallel processing
- GPU acceleration via OpenCL/OpenGL
- Memory management with custom allocators

4. Storage/Caching System

- Multiple backend support (LevelDB, FoundationDB)
- Thread-safe operations
- Abstract CacheBase class with concrete implementations

Build System & Dependencies

Build Configuration:

```
mkdir build && cd build
cmake .. -DCMAKE_BUILD_TYPE=Debug \
         -DBOOST_INCLUDE_DIR=/usr/include \
         -DOPENMIND_BUILD_TESTS=ON \
         -DOPENMIND_USE_OPENCL=ON
make -j8
```

Key Dependencies:

- Boost (1.81+): MPL, compute, filesystem, serialization
- LevelDB/FoundationDB
- OpenCL/Vulkan
- Python3 with pip
- CMake 3.15+
- C++20 compiler
- For Ubuntu/Debian:

  ```
  sudo apt install -y cmake ninja-build g++-10 git cmake-curses-gui ninja-build cmake-qt-
  ```

Development Environment

- Docker containers available:
  - Ubuntu-based
  - Gentoo-based
  - FoundationDB-specific
- VS Code integration
- Clang format configuration
- Git workflow utilities

Testing

- Boost Test framework
- Tests organized per component
- Image processing tests
- Neural network tests (FANN)
- Mathematical operation tests
- Storage backend tests

Important Implementation Details

1. Threading & Concurrency:

- Heavy use of async operations
- Thread-safe storage operations
- Lockfree data structures in runtime
- Parallel task execution

2. Memory Management:

- Custom allocators
- Smart pointer usage throughout
- Garbage collection in runtime

3. Cross-Platform Considerations:

- Windows/POSIX compatibility
- Platform-specific process management
- Conditional compilation for GPU features

4. Performance:

- Caching systems for optimization
- Template metaprogramming
- GPU acceleration where available
- Compile-time computations

Development Guidelines

Data Structures

- All getter methods must maintain O(1) time complexity
- Lazy evaluation patterns should be avoided as they interfere with O(1) access requirements
- Use bit masks for implementing tags to optimize memory usage
- All getter operations, including value operations and tag lookups, must maintain O(1) complexity

Root Cause Analysis (RCA)

- RCA must be completed and approved by the maintainer before creating PRs
- RCA must be presented in a specific three-part format:
  1. Single call stack from gdb output pointing to the root cause call
  2. Single code chunk of the pointed function containing the problematic code
  3. Variables that have unexpected values within that chunk of code

- Apply Occam's razor principle by starting with the simplest possible explanation
- Avoid complex assumptions without direct supporting evidence
- Verify each assumption before moving to more complex explanations
- Focus on direct observations from debugging tools

Method Implementation

- When implementing specific method overloads, treat each overload as separate and unrelated
- Each overload should be implemented and tested independently

Refactoring

- Even "refactoring-only" changes must be thoroughly verified through test execution
- Structural changes can unexpectedly affect code logic
- Each refactoring change must be validated through the test suite

Multiple Related Changes

- Split changes involving multiple components into separate, incremental PRs
- Follow an incremental approach:
  1. First PR: Add the base method/interface
  2. Subsequent PRs: Add individual class implementations one at a time
  3. Final PRs: Add any remaining implementations or cleanup

Cross-Session Communication

- The repository has a dedicated context branch (https://github.com/ohhmm/_/tree/context)
- Contains a SQLite-based communication system with tables for:
  - active_work: Current tasks and status tracking
  - session_knowledge: Priority-based information storage
  - session_notes: Observations and decisions
- Can be used to store and retrieve structured information between sessions

Storage Implementation

- New storage or caching implementations should begin in the storage subdirectory (omnn/storage/)
- Core storage functionality should be implemented and tested in the storage layer first
- Integration with higher-level modules should follow after storage layer implementation

Verification Tasks

- All verification tasks must have a structured plan with explicit steps
- Include measurable completion criteria for each step
- Show evidence of the actual verification process
- Document verification steps and outcomes
- Execute as formal plan items rather than ad-hoc checks

System Optimization

- Time savings have a direct impact on business sustainability through user growth
- Optimizations that reduce user time investment have multiplicative value
- System effectiveness is measured by comparing time metrics

Git Workflow

- Before pushing changes, perform `git pull --rebase --autostash origin main`
- This helps resolve conflicts and ensure a clean commit history

License