Consolidated Knowledge Base for ohhmm/openmind

Table of Contents

Project Structure and Organization

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

5. Quantum Computing (`omnn/quantum`)

- Quantum state manipulation and operations
- QuantumRegister class for quantum state representation

- Support for controlled phase rotation and normalization
- Framework for quantum algorithm implementation
- Integration with classical computation

6. Logic System (`omnn/logic`)

- Propositional reasoning and logical operations
- Equality constraints and inference rules
- Mathematical representation of logical expressions
- Integration with equation solving
- Support for logical inference and constraint satisfaction

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
- Derivative expression format follows the pattern "(expression)'variable", where 'variable represents the variable to differentiate with respect to
- Advanced mathematical operations including integration, differentiation, and special functions
- Bit manipulation operations for low-level optimization
- Copy-on-Write (COW) memory model for efficient object sharing and modification
- Conditional operations (Ifz, IfNZ, IfEq) for complex mathematical expressions
- Symbolic computation capabilities for mathematical precision

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
- Efficient caching and persistence of data
- Optimized storage and retrieval of computed mathematical expressions
- LevelDbCache implementation with custom connection options
- Asynchronous operations for improved performance
- Memory caching for frequently accessed items

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
- Each issue resolution should start with root cause analysis to identify the fundamental problem before implementing any fixes
- All assumptions made during code inspection must be verified through actual debugging

Method Implementation

- When implementing specific method overloads, treat each overload as separate and unrelated
- Each overload should be implemented and tested independently
- When implementing new methods, follow strict TDD workflow:
  1. First isolate the problem that necessitates the new method
  2. Create a failing test that demonstrates the issue
  3. Submit initial PR with the disabled failing test
  4. Create separate PR with the fix that enables and passes the test
- Mathematical operations should follow the organization pattern established by the Logarithm implementation

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

Investigation Approach

- When investigating issues, prefer direct testing and reproduction over asking for detailed problem descriptions
- This saves time and allows for more efficient problem identification
- If you have the ability to reproduce and investigate an issue directly, do so rather than waiting for detailed user descriptions

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 add-apt-repository -y ppa:mhier/libboost-latest && sudo apt update && sudo apt upg
  ```

Dependency Management Systems:

- Dependencies are managed through multiple systems:
  - Conan: Uses conanfile.txt
  - vcpkg: Uses vcpkg.json
  - Native package managers (apt, brew)

CI/CD Configuration

```yaml
jobs:
  build-in-ubuntu:
    runs-on: ubuntu-22.04+
    steps:
    - uses: actions/checkout@v3
    - name: OS prerequisites
      run: sudo add-apt-repository -y ppa:mhier/libboost-latest && sudo apt update && sudo a
    - name: Create Build Dir
      run: cmake -E make_directory ${{github.workspace}}/build
    - name: Configure
      working-directory: ${{github.workspace}}/build
      run: cmake ${{github.workspace}} -DOPENMIND_BUILD_SAMPLES=NO -DOPENMIND_BUILD_TESTS=ON
    - name: Install prerequisites
      working-directory: ${{github.workspace}}/build
      run: cmake --build ${{github.workspace}}/build --target prerequisites -j `nproc`
    - name: ReConfigure to detect fetched prerequisites
      working-directory: ${{github.workspace}}/build
      run: cmake ${{github.workspace}}
    - name: Build
      working-directory: ${{github.workspace}}/build
      run: cmake --build ${{github.workspace}}/build -j `nproc` --config Release
    - name: Check
      working-directory: ${{github.workspace}}/build
      run: ctest . -j`nproc` -C Release -E "image_codec_test|ts|Polyfit_test" --rerun-failed
```

Testing

Testing Framework

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

Testing Guidelines

- Tests should be enabled and fixed to pass successfully rather than being left in a disabled state
- When encountering a disabled test, the priority should be to:
  1. Enable the test by removing the disabled flag
  2. Fix any issues that arise to make the test pass
  3. Verify the test passes successfully in CI
- Before running tests on a specific branch, always verify that the branch is up-to-date with its remote counterpart
- When creating a new test to address an issue, the test must first demonstrate the failing behavior before any fixes are implemented
- Tests should not be excluded from CI or disabled to work around failures
- Before pushing changes that enable or modify tests, verify that the tests pass successfully in a local environment first
- When fixing issues in mathematical operations, use Test-Driven Development (TDD) to implement the fix before enabling any disabled tests that depend on it

Implementation Details

1. Threading & Concurrency:

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

5. Python Bindings (`omnn/variable`):

- Integration with Python through Boost.Python
- Exposes mathematical operations to Python scripts
- Allows seamless integration of OpenMind's functionality into Python workflows
- Supports mathematical operations, neural networks, and extrapolator functionality
- Built when the `OPENMIND_BUILD_PYTHON_BINDINGS` CMake option is enabled

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

Git Workflow

- Before pushing changes, perform `git pull --rebase --autostash origin main`
- This helps resolve conflicts and ensure a clean commit history
- When rebasing pull requests, always update the existing branches directly rather than creating new branches
- When resolving merge conflicts in pull requests, squash commits to remove the conflict resolution history before merging
- When making changes, do not create new commits until the existing commit is amended and successfully builds on CI
- Each PR must contain exactly one commit, even when working with multiple related commits
- When creating PRs from existing commits in a branch, verify that each PR contains only its target commit
- Test changes must be isolated in their own dedicated commits and pull requests
- Create pull requests directly from existing commits first, without making any local code changes or running tests
- When creating multiple PRs from a series of commits, prefer creating independent PRs based directly off the main branch
- Only rebase and push pull requests that were created by yourself
- Always create pull requests targeting the main branch
- Each individual commit should be submitted as a separate pull request, even when multiple commits are related to the same feature

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
- The ohhmm/_ repository's context branch is designated for storing structured data (particularly SQLite databases)

- This branch can be freely modified without requiring explicit approval
- The system uses SQLite for git-friendly versioning and efficient querying
- When verification tasks need to be performed, they should be stored in branch_tracking.db for other Devin instances to check
- Definition of Done (DoD) procedures and verification checks should be stored in the session_knowledge table with knowledge_type='definition_of_done'

Mathematical Operations

- When implementing mathematical operations in the `ohhmm/openmind` repository:
  - Mathematical operation simplification methods (like IsModSimplifiable, IsMultiplicationSimplifiable) can be implemented across different numeric types
  - The Integer class is noted as a candidate for implementing its own IsModSimplifiable method
  - Consider whether other numeric types could benefit from similar implementations
- When implementing new mathematical operations:
  - Follow the organization pattern established by the Logarithm implementation
  - Operations are expected to support string-based input formats (e.g. "lim(x->0, expression)" for limits)
- Before reporting progress on mathematical operation implementations, verify that:
  1. The operation produces correct numerical results through test cases
  2. Type conversions work correctly (especially float/double conversions)
  3. All edge cases are handled properly
  4. The implementation integrates properly with the existing mathematical framework
- Simply compiling without errors is not sufficient to consider a mathematical operation implementation complete or working

Documentation Standards

- Documentation must prioritize technical precision and reader efficiency:
  1. Avoid imprecise or ambiguous technical statements
  2. Focus on saving reader's time through concise, accurate descriptions
  3. Prioritize specific technical details over general descriptions
  4. Documentation should be verifiably accurate against the codebase
- This standard applies to all documentation including:
  - API references
  - Architecture descriptions
  - Component documentation
  - Configuration guides

- Documentation must be organized with a focus on integrity through two key components:
  1. A consolidated knowledge base that presents all information in a unified, coherent structure
  2. A separate contradictions analysis that explicitly identifies and resolves any inconsistencies in the knowledge base
- This dual-document approach ensures that:
  - All knowledge is properly consolidated in one place
  - Contradictions are explicitly tracked and resolved
  - Documentation maintains internal consistency
  - Information integrity is actively maintained
- Both documents should be maintained in PDF format in the repository's documentation directory
- Documentation can be tested by accessing the rendered content at the branch-specific path: docs/modules/ROOT/pages/index.adoc

Integrity and Verification

- Before making any statements about code changes or CI checks, always verify the actual changes by checking the git diff first
- Before reporting any test results or status updates, each statement must be verified with concrete evidence
- Any claims about test results or the effectiveness of fixes must be accompanied by a proof link
- Progress updates should be clear, concise, and focused on key developments
- When making statements about code matching a reference implementation, every single line must be verified to match exactly
- Mathematical model theory, particularly consolidated contradictionless formats, should be used as a foundation for implementing system integrity features
- Integrity must be inherent in the reasoning process itself, not just a compliance check
- All development work, including root cause analysis (RCA), must be performed without speculation or assumptions
- When making statements about technical issues or system states, especially those involving causality ("because"), always verify the claim through direct observation or testing before communicating it
- Each technical assertion must be backed by concrete evidence that you have personally verified, not assumptions or indirect observations

Task Management and Workflow

Starting Tasks

- Follow a strict three-step process for all tasks:
  1. Understand everything possible about the requirements before pro-

ceeding
    2. Communicate understanding back to user and obtain clarification if needed
    3. Only implement changes after complete understanding is verified
- This process is vital for saving time and ensuring work aligns with requirements
- For test-related tasks, this means understanding exactly what scenario needs to be tested before writing any test code

Definition of Done (DoD)

- Any changes to the Definition of Done (DoD) document require explicit approval from the user
- After completing significant changes, push to the PR and notify the user for review
- The DoD must include verification of CI/CD workflow status
- Before considering a task complete, verify the status of CI/CD workflows
- Before considering a task complete, it is essential to verify the status of CI/CD workflows running on the pull request to ensure that the changes are being tested correctly and that there are no issues with the build or tests

Documentation Organization

- Documentation must be organized with a focus on integrity through two key components:
    1. A consolidated knowledge base that presents all information in a unified, coherent structure
    2. A separate contradictions analysis that explicitly identifies and resolves any inconsistencies
- Both documents should be maintained in PDF format in the repository's documentation directory
- Documentation can be tested by accessing the rendered content at the branch-specific path: docs/modules/ROOT/pages/index.adoc

Project Fundamentals

The OpenMind project is fundamentally an AGI infrastructure framework with three core capabilities that should guide all development: 1. Knowledge accumulation in universal distributed forms 2. Deduction capabilities for processing and deriving new information 3. Self-modification capabilities while maintaining system integrity and consistency

All changes and features should be evaluated based on how they support or enhance these core capabilities rather than just their immediate functional requirements.

License

- BSD 3-Clause License (2019, Sergei Krivonos)