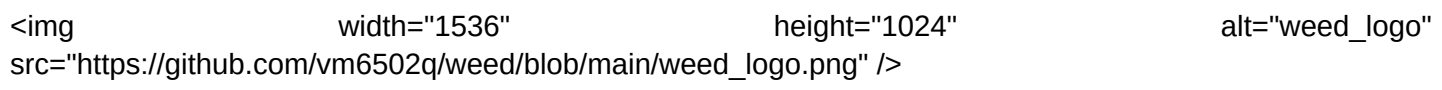
Weed Repository Documentation

./README.md



Weed

Minimalist AI/ML inference and backprogation in the style of [Qrack](https://github.com/unitaryfoundation/qrack)

Development Status

- *Weed** is a rapidly-developing **work-in-progress**. Its ABI may change drastically and without notice.

The project provides a set of essential CPU and GPU **kernels**, used by `Tensor` instances that perform _autograd._ We also provide _stochastic gradient descent (SGD)_ and _Adam_ optimizer implementations. (Build and check the API reference to get started.)

Why try Weed?

With the growing popularity of AI/ML tools and workflows (including LLMs), legacy frameworks often carry "code debt" from over a decade of rapidly developing research history. This has led them to "bolt on" new features and advancements to design principles decided before the latest research. Popular frameworks also commonly started based in Python (maybe to capture early adoption), only later potentially "tacking on" a C++ library for special-case deployment needs. These conditions have produced libraries and frameworks with complicated dependency trees that occupy upward of a GB of disk footprint. This entire ecosystem might be due for a "refresh."

- *Weed** does not seek to fully replace or supplant established frameworks. However, it aims for **minimalist complete closure** on the primitives necessary for high-performance AI/ML inference and back-propagation. Chiefly, this includes **kernels**, and a `Tensor` interface that immediately produces an **autograd** graph appropriate for training. Allowing **optional** OpenCL (and/or CUDA) for **hardware acceleration**, it will remain **free of required dependencies** outside of C++(11) language standard.

Rethinking AI/ML library design this way, `Weed` has realized a rather unique and powerful form of _sparsification_ of `Tensor` **storage**. _Sparseness_ should **not** be a **`Tensor` interface concern**, but rather a **`Storage` concern**. Inspired by the design of the [Qrack](https://github.com/unitaryfoundation/qrack) quantum computer simulation framework, the `Tensor` interface treats **sparse and dense** tensors as **functionally equivalent**. Sparse optimization is so "transparently streamlined," this way, that it defaults to enabled for CPU-based tensors, and we recommend you leave it enabled at all times.

Much like `Qrack`, `Weed` is designed to make the correct thing the default-and the expensive thing explicit.

Building the API reference

```sh
$ doxygen doxygen.config
```

Performing code coverage

```sh
$ cd _build
$ cmake -DENABLE_CODECOVERAGE=ON ..
$ make unittest
$ ./unittest
$ make coverage
$ cd coverage_results
```

$ python -m http.server
```

**Directory Structure**

- **cmake/**: CMake modules for build configuration.
- **debian/**: Debian packaging files.
- **examples/**: Example code demonstrating usage.
- **include/**: Public API header files, organized by module.
- `autograd/`: Optimizers and loss functions.
- `common/`: Common utilities and definitions.
- `devices/`: Device abstraction.
- `enums/`: Enumerations.
- `modules/`: Neural network modules.
- `ops/`: Tensor operations.
- `storage/`: Tensor storage implementations.
- `tensors/`: Tensor interface.
- **src/**: Source code implementations, mirroring the `include/` structure.
- **test/**: Unit tests.

**Copyright, License, and Acknowledgments**



The Weed logo was produced with assistance from "Elara," an OpenAI custom GPT, and it is in the **public domain**.



## ./cmake/README.md

**CMake Modules**

This directory contains CMake modules and scripts used to configure the build process of the Weed library.

**Files**

- **Boost.cmake**: Configures the Boost C++ libraries dependency. It defines `BOOST_AVAILABLE` if found.
- **CUDA.cmake**: Handles the detection and configuration of CUDA for GPU acceleration. It sets `ENABLE_CUDA` and configures target architectures.
- **Complex_x2.cmake**: Likely configures double-precision complex number support or SIMD extensions.
- **Coverage.cmake**: Sets up code coverage analysis (likely used with `ENABLE_CODECOVERAGE`).
- **CppStd.cmake**: Ensures the compiler supports the required C++ standard (C++11 or later).
- **EnvVars.cmake**: Helper for handling environment variables.
- **Examples.cmake**: Configures the building of example executables.
- **Format.cmake**: Configures `clang-format` targets for code formatting.
- **FpMath.cmake**: Configures floating-point math optimizations (e.g., `-ffast-math`).
- **OpenCL.cmake**: Handles the detection and configuration of OpenCL. It supports fetching headers for Apple/PPC, configures SnuCL if enabled, and adds custom commands to compile `.cl` kernels into headers.
- **Pstridepow.cmake**: Configuration related to specific math optimizations or stride power functions
```

(project-specific).
- **Pthread.cmake**: Configures POSIX threads support.
- **TCapPow.cmake**: Configuration for `tcapint` (tensor capacity integer) power functions.

./debian/README.md

Debian Packaging

This directory contains the necessary files to build a Debian package (`.deb`) for the Weed library.

Files

- **control.in**: Template for the Debian `control` file, defining package metadata, dependencies, and descriptions for the library and its development headers (`libqrack-dev` equivalent).
- **rules**: The `make` rules for building the package.
- **copyright**: Copyright information for the package.
- **README.Debian**: specific notes for Debian users.
- **README.source**: Information about the source package.
- **changelog**: The changelog for the Debian package.
- **files**, **libqrack*.dirs.in**, **libqrack*.install.in**: Helper files for installation directories and file mappings.
- **triggers**: Package triggers.

These files are typically used by packaging tools (like `dpkg-buildpackage` or CPack's Debian generator) to create installable packages.

./examples/README.md

Examples

This directory contains example code demonstrating how to use the Weed library.

Files

- **nor.cpp**: A simple example that trains a small neural network to learn the logical NOR function.
- It demonstrates how to:
- Create `Tensor` objects for input data and labels.
- Instantiate `Linear` layers and extract their parameters.
- Use the `Adam` optimizer.
- Construct a forward pass using activation functions (`relu`, `sigmoid`).
- Calculate loss (`mse_loss` or `bci_loss`).
- Perform backpropagation (`Tensor::backward`) and optimizer steps (`adam_step`).
- Reset gradients (`zero_grad`).

./include/README.md

Public API Headers

This directory contains the public header files for the Weed library, organized by module.

Subdirectories

- **autograd/**: Optimizers, loss functions, and computation graph nodes.
- **common/**: Common utilities, type definitions, and device API wrappers.
- **devices/**: Hardware device abstraction (GPU management).

- **enums/**: Enumerations for devices and data types.
- **modules/**: Neural network layers (e.g., `Linear`).
- **ops/**: Tensor operation definitions (`add`, `mul`, `matmul`, etc.).
- **storage/**: Data storage classes handling memory management and device transfers.
- **tensors/**: The core `Tensor` class interface.

./include/autograd/README.md

Autograd

This module provides the necessary components for automatic differentiation and optimization in Weed.

Files

- **node.hpp**: Defines the `Node` struct, which is the building block of the autograd computation graph. Each node stores references to its parent tensors and a closure (`std::function`) to execute the backward pass.
- **adam.hpp**: Implements the **Adam** optimizer.
- `Adam`: Structure holding optimizer state (first and second moments) and hyperparameters (`lr`, `beta1`, `beta2`, `eps`).
- `adam_step`: Function to perform a single optimization step.
- **sgd.hpp**: Implements **Stochastic Gradient Descent (SGD)**.
- `sgd_step`: Function to perform a simple gradient descent update.
- **mse_loss.hpp**: Implements **Mean Squared Error (MSE)** loss function (`mse_loss`).
- **bci_loss.hpp**: Implements **Binary Cross-Entropy (BCI)** loss function (`bci_loss`).
- **zero_grad.hpp**: Provides the `zero_grad` helper function to reset gradients of parameters to zero before a new training step.

./include/common/README.md

Common Utilities

This directory contains common definitions, types, and utility classes used throughout the Weed library.

Files

- **weed_types.hpp**: The core type definitions for the library.
- `real1`: The floating-point scalar type (configurable via `FPPOW` to be `half`, `float`, `double`, or `float128`).
- `complex`: Complex number type based on `real1`.
- `tcapint`: Integer type for tensor dimensions and capacities (configurable via `TCAPPOW`).
- Constants: Mathematical constants (`PI_R1`, `ONE_R1`, etc.) and configuration flags.
- **parallel_for.hpp**: Provides the `ParallelFor` class for multi-threaded execution on the CPU. It supports parallel loops over dense ranges and sparse containers.
- **oclapi.hpp**: Defines the `OCLAPI` enum, which lists the available OpenCL kernels (e.g., `OCL_API_ADD_REAL`, `OCL_API_MATMUL_COMPLEX`).
- **oclengine.hpp**: The OpenCL runtime manager.
- `OCLEngine`: A singleton that manages OpenCL devices, contexts, and program compilation.
- `OCLDeviceContext`: Encapsulates an OpenCL context, command queue, and device properties.
- **weed_functions.hpp**: Declarations for common mathematical and utility functions.
- **complex*x2simd.hpp**: SIMD vectorization helpers for complex numbers.
- **config.h.in**: Template for the generated `config.h`.

./include/devices/README.md

Device Abstraction

This directory contains classes for managing hardware devices, specifically GPUs (via OpenCL or CUDA).

Files

- **gpu_device.hpp**: Defines the `GpuDevice` class.
- `GpuDevice`: The primary interface for interacting with a GPU. It manages the OpenCL/CUDA context, command queue, and memory allocations. It provides high-level methods to execute kernels (e.g., `FillValueReal`, `DispatchQueue`) and manage buffers (`MakeBuffer`, `LockSync`, `UnlockSync`).
- **pool_item.hpp**: Defines `PoolItem`, which manages reusable, pre-allocated buffers for passing scalar arguments (like complex numbers or dimensions) to kernels, reducing allocation overhead.
- **queue_item.hpp**: Defines `QueueItem`, a simple struct that encapsulates the parameters for a pending kernel execution request (API call ID, work sizes, buffers) before it is processed by the device queue.

./include/enums/README.md

Enumerations

This directory contains enumeration definitions used throughout the Weed library.

Files

- **device_tag.hpp**: Defines the `DeviceTag` enum, specifying the backend device type for tensors.
- `CPU`: CPU-based storage and execution.
- `GPU`: GPU-based storage and execution (OpenCL/CUDA).
- `Qrack`: (Experimental/Future) Qrack-based execution.
- **dtype.hpp**: Defines the `DType` enum, specifying the data type of the tensor elements.
- `REAL`: Real numbers (precision defined by `real1`).
- `COMPLEX`: Complex numbers.

./include/modules/README.md

Modules

This directory contains neural network module definitions, providing composable layers for building models.

Files

- **module.hpp**: Defines the abstract `Module` base class.
- `forward`: Pure virtual function to perform the forward pass of the module.
- `parameters`: Pure virtual function to return a list of trainable parameters (`ParameterPtr`) in the module.
- **linear.hpp**: Defines the `Linear` module (fully connected layer).
- `Linear`: Represents a linear transformation $y = xW + b$.
- Manages `weight` and optional `bias` parameters.
- Supports random initialization or zero initialization.
- Supports both real and complex data types.

./include/ops/README.md

Tensor Operations

This directory contains the definitions and dispatch logic for tensor operations. It defines kernel structures that abstract the execution of operations across different devices (CPU/GPU) and data types (Real/Complex).

Files

- **abs.hpp**: Absolute value operation (`abs`).
- **clamp.hpp**: Clamping operation (`clamp`), limiting values to a specified range.
- **commuting.hpp**: Commutative binary operations (`add`, `mul`). Defines the `CommutingKernel` struct used to dispatch these operations.
- **div.hpp**: Division operation (`div`).
- **in_place.hpp**: In-place binary operations (`add_in_place`, `sub_in_place`). Defines the `InPlaceKernel` struct.
- **matmul.hpp**: Matrix multiplication (`matmul`, `>>`, `<<`).
- **pow.hpp**: Power (`pow`, `^`) and exponential (`exp`) operations. Also includes logarithm (`log`).
- **real_unary.hpp**: Unary operations that might have specific real-valued behaviors or constraints.
- **reduce.hpp**: Reduction operations base definitions.
- **sub.hpp**: Subtraction operation (`sub`).
- **sum.hpp**: Summation operation (`sum`, `mean`).
- **unary.hpp**: Common unary operations like activation functions (`relu`, `sigmoid`, `tanh`) and their gradients. Defines the `UnaryKernel` struct.

./include/storage/README.md

Storage

This directory contains the classes responsible for managing the actual data of tensors. It abstracts over the device (CPU/GPU), data type (Real/Complex), and layout (Dense/Sparse).

Files

- **storage.hpp**: Defines the abstract `Storage` base class.
- Manages metadata: `DeviceTag`, `DType`, and `size`.
- Defines virtual interface for data movement: `cpu()`, `gpu()`.
- Defines utility methods: `FillZeros()`, `FillOnes()`, `Upcast()`.
- **all_storage.hpp**: Convenience header including all storage types.
- **real_storage.hpp**, **complex_storage.hpp**: Intermediate interfaces for Real and Complex data types, defining virtual methods for element access (`operator[]`, `write`, `add`).
- **cpu_real_storage.hpp**, **cpu_complex_storage.hpp**: Dense implementations for CPU.
- **sparse_cpu_real_storage.hpp**, **sparse_cpu_complex_storage.hpp**: Sparse implementations for CPU, using hash maps (`std::unordered_map`) to store only non-zero elements.
- **gpu_storage.hpp**: Interface for GPU storage.
- **gpu_real_storage.hpp**, **gpu_complex_storage.hpp**: Dense implementations for GPU, managing OpenCL/CUDA buffers.

./include/tensors/README.md

Tensors

This directory contains the core tensor class definitions and their specializations.

Files

- **tensor.hpp**: Defines the `Tensor` struct, the central data structure in Weed.
- **Properties**:

- `storage`: Pointer to the underlying data storage (`StoragePtr`).
- `shape` & `stride`: Vectors defining the tensor's dimensions and memory layout.
- `grad`: Gradient tensor (if `requires_grad` is true).
- `grad_node`: Pointer to the node in the autograd computation graph.
- **Operations**: Static methods for math operations (`add`, `matmul`, `relu`, etc.) that execute the operation and build the computation graph.
- **Operators**: Overloads for `+`, `-`, `*`, `/`, `>>` (matmul), `^` (pow).
- **parameter.hpp**: Defines `Parameter`, a subclass of `Tensor` that defaults to `requires_grad=true`. Used for learnable weights in `Module`s.
- **scalar.hpp**: Defines `Scalar`, a subclass of `Tensor` representing a single value (rank-0 tensor).
- **real_tensor.hpp**, **complex_tensor.hpp**: (If used directly) Specializations for real and complex tensors.
- **real_scalar.hpp**, **complex_scalar.hpp**: Specializations for real and complex scalars.
- **flat_tensors.hpp**: Utilities for flattening tensors (likely for serialization or specific operations).

./src/README.md

Source Code

This directory contains the implementation of the Weed library.

Subdirectories

- **common/**: Implementations of common utilities and OpenCL/CUDA kernels.
- **devices/**: Implementations of device management.
- **modules/**: Implementations of neural network modules.
- **ops/**: Implementations of tensor operation dispatch logic.
- **storage/**: Implementations of storage classes (CPU/GPU, Sparse/Dense).
- **tensors/**: Implementations of the `Tensor` logic.

Files

- **weed_cl_precompile.cpp**: A utility program that precompiles OpenCL kernels and saves them to disk to speed up subsequent load times.

./src/common/README.md

Common Implementations

This directory contains the implementations of the common utilities and the OpenCL/CUDA kernels.

Files

- **weed_types.cpp**: (If present) Implementation of type-related utilities.
- **parallel_for.cpp**: Implementation of the `ParallelFor` class, handling thread pool management (implied).
- **oclengine.cpp**: Implementation of the OpenCL engine, handling device discovery, context creation, and kernel execution.
- **cudaengine.cu**: Implementation of the CUDA engine (if CUDA is enabled).
- **qengine.cl**: The main OpenCL source file containing the kernel implementations for tensor operations.
- **qengine.cu**: The main CUDA source file containing the kernel implementations.
- **qheader_*.cl**: Helper OpenCL headers for different data types (float, double, half, etc.).
- **functions.cpp**: Implementation of common math functions.
- **dispatchqueue.cpp**: Implementation of a thread-safe dispatch queue (likely used by `ParallelFor` or

similar).

./src/devices/README.md

Device Implementations

This directory contains the implementation of the device management classes.

Files

- **gpu_device.cpp**: Implements the `GpuDevice` class. It handles the details of:
- Creating and managing OpenCL/CUDA buffers.
- Enqueuing kernel execution commands to the command queue.
- Synchronizing memory access between host and device (`LockSync`, `UnlockSync`).
- Dispatching specific operations (like filling buffers or copying data) to the underlying engine.

./src/modules/README.md

Module Implementations

This directory contains the implementations of the neural network modules.

Files

- **linear.cpp**: Implements the `Linear` module.
- **Constructor**: Initializes weights and biases. Can perform random initialization (uniform distribution scaled by $1/\sqrt{in\_features}$) or zero initialization.
- **forward**: Computes the linear transformation using matrix multiplication (`operator>>`) and addition (for bias).
- **parameters**: Returns the weight and bias (if present) parameters for optimization.

./src/ops/README.md

Operation Implementations

This directory contains the source code for dispatching tensor operations to the appropriate kernels (CPU or GPU).

Files

- **abs.cpp**: Implementation of absolute value dispatch.
- **clamp.cpp**: Implementation of clamp dispatch.
- **commuting.cpp**: Implementation of addition and multiplication dispatch. It handles broadcasting and type promotion (Real + Complex -> Complex).
- **div.cpp**: Implementation of division dispatch.
- **in_place.cpp**: Implementation of in-place addition and subtraction dispatch.
- **matmul.cpp**: Implementation of matrix multiplication dispatch.
- **pow.cpp**: Implementation of power, exponentiation, and logarithm dispatch.
- **real_unary.cpp**: Implementation of real-valued unary operations.
- **reduce.cpp**: Implementation of reduction operations (sum, mean).
- **sub.cpp**: Implementation of subtraction dispatch.
- **sum.cpp**: Implementation of summation dispatch (often alias or specific reduction).
- **unary.cpp**: Implementation of unary operations (ReLU, Sigmoid, Tanh) and their gradient computations.

./src/storage/README.md

Storage Implementations

This directory contains the implementations of the storage classes.

Files

- **cpu_real_storage.cpp**, **cpu_complex_storage.cpp**: Implementations for dense CPU storage.
- Handles memory allocation (`Alloc`) and deallocation.
- Implements data transfer logic (to/from GPU).
- **gpu_real_storage.cpp**, **gpu_complex_storage.cpp**: Implementations for GPU storage.
- Interacts with `GpuDevice` to manage device buffers.
- Implements data transfer logic (to/from CPU).
- Uses kernels for operations like `FillZeros`.

./src/tensors/README.md

Tensor Implementations

This directory contains the implementation of the tensor logic.

Files

- **tensor.cpp**: Implements the `Tensor` class methods.
- **Constructors**: Initializes storage, shape, and stride.
- **Operation Factories**: Methods like `add`, `mul`, `matmul` which:

1. Check input shapes/types.
2. Dispatch to the appropriate `ops` kernel.
3. If gradients are required, create a `Node` in the autograd graph with the appropriate backward closure.
   - **Autograd**: `backward()` triggers the reverse-mode automatic differentiation from the given tensor (usually loss).

./test/README.md

Tests

This directory contains the unit tests for the Weed library, built using the [Catch2](https://github.com/catchorg/Catch2) framework.

Files

- **test_main.cpp**: The main entry point for the test runner. It parses command-line arguments to configure the test environment, such as selecting the device (CPU/GPU) to run tests on.
- **tests.cpp**: Contains the actual test cases covering:
- Tensor storage and device transfers.
- Arithmetic operations (add, sub, mul, div).
- Activation functions (ReLU, Sigmoid, Tanh).
- Matrix multiplication.
- Autograd functionality (verifying gradients).
- **tests.hpp**: Header file defining common macros, globals (like the current `DeviceTag` being tested), and includes.
- **catch.hpp**: The single-header Catch2 library.

Running Tests

Tests are typically built and run via CMake:

```sh
cd _build
make unittest
./unittest
```

You can select specific devices using command line flags:

- `--device-cpu`: Run CPU tests.
- `--device-gpu`: Run GPU tests.