Tensor Module Implementation

Overview and Architecture

Our tensor module represents a custom implementation of a high-performance tensor computation library that seamlessly operates across CPU and CUDA devices. Unlike existing frameworks, this implementation provides direct control over memory management and computation patterns while maintaining a clean abstraction layer for device-agnostic operations.

The architecture follows a unified tensor abstraction where each tensor maintains its shape, data pointer, and device location. This design enables transparent computation across different hardware backends while optimizing for the specific characteristics of each platform.

Syntax error in text

mermaid version 10.9.3

The fundamental design principle revolves around device-aware dispatch mechanisms. When an operation is invoked, the tensor's device type determines which backend implementation executes the computation. This polymorphic behavior is achieved through static methods that check device compatibility and route to the appropriate implementation:

```
static Tensor add(const Tensor &a, const Tensor &b) {
    if (a.device == Device::CUDA && b.device == Device::CUDA) {
        return cuda_add(a, b);
    } else if (a.device == Device::CPU && b.device == Device::CPU)
        return cpu_add(a, b);
    } else {
        throw std::invalid_argument("Cannot add tensors on differe
    }
}
```

This architecture ensures type safety at compile time while providing runtime flexibility for device selection. The tensor class itself acts as a smart container that manages memory lifecycle across different devices, automatically handling allocation and deallocation through RAII principles.

Broadcasting Mechanism

The broadcasting system implements NumPy-compatible shape inference, enabling operations between tensors of different dimensions. This mechanism is crucial for expressing complex computations concisely without explicit shape manipulation.

The core broadcasting algorithm works by comparing shapes from right to left, determining compatibility based on two rules: dimensions must either be equal or one of them must be 1. The implementation elegantly handles this through reverse iteration:

```
static bool can_broadcast(const std::vector<int> &a, const std::ve
                          std::vector<int> &shape) {
    int i1 = a.size() - 1;
    int i2 = b.size() - 1;
    int i3 = shape.size() - 1;

    while (i1 >= 0 && i2 >= 0) {
        if (a[i1] != b[i2]) {
            if (a[i1] != 1 && b[i2] != 1) {
                return false;
            }
        }
        shape[i3] = std::max(a[i1], b[i2]);
        i1--; i2--; i3--;
    }
    // Handle remaining dimensions
    while (i1 >= 0) shape[i3--] = a[i1--];
    while (i2 >= 0) shape[i3--] = b[i2--];
    return true;
}
```

The broadcasting implementation differs significantly between CPU and CUDA backends. On the CPU side, the view mechanism enables efficient element access without data duplication:

```
#pragma omp parallel for
for (int i = 0; i < result.num_elements; i++) {
    result.data[i] = (*a.view(shape, i)) + (*b.view(shape, i));
}
```

The `view` function performs the critical task of mapping broadcasted indices back to the original tensor indices. This mapping handles dimension expansion (when a dimension is 1) by effectively ignoring the index in that dimension:

```
const float *Tensor::view(const std::vector<int> &asshape, int ind
    const float *ret = data;
    // ... setup code ...

    for (size_t i = 0; i < shape.size(); i++) {
        if (shape[i] == 1) {
            index %= view_elem;
            continue;  // Broadcast dimension - ignore index
        }
        const int idx = index / view_elem;
        n_elem /= shape[i];
        ret += n_elem * idx;
        index %= view_elem;
    }
    return ret;
}
```

For CUDA operations, broadcasting requires a different approach due to the parallel nature of GPU execution. The CUDA kernel computes the multi-dimensional indices for each thread and maps them appropriately:

```
template <typename Op>
__global__ void broadcastOpKernel(const float *a, const float *b,
                                  int *a_shape, int a_ndim,
                                  int *b_shape, int b_ndim,
                                  int *result_shape, int result_nd
                                  int num_elements, Op op) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    if (idx < num_elements) {
        // Convert flat index to multi-dimensional indices
        int indices[8];
        int remaining = idx;
        for (int i = result_ndim - 1; i >= 0; i--) {
            indices[i] = remaining % result_shape[i];
            remaining /= result_shape[i];
        }

        // Map to source tensor indices with broadcasting rules
        int a_idx = 0, b_idx = 0;
        // ... index calculation ...

        result[idx] = op(a[a_idx], b[b_idx]);
    }
}
```

This design allows any binary operation to support broadcasting by simply providing the operation as a template parameter, achieving both code reuse and performance optimization through compile-time specialization.

CPU Parallelization Strategies

The CPU backend employs a hybrid parallelization approach, combining OpenMP directives with manual thread management to optimize for different workload characteristics. This dual strategy reflects a sophisticated understanding of parallelization overhead and cache efficiency.

For large-scale operations where the overhead of thread creation is negligible compared to computation time, OpenMP provides an elegant solution:

```cpp
Tensor Tensor::cpu_add(const Tensor &a, const Tensor &b) {
    // ... broadcasting setup ...

    #pragma omp parallel for
    for (int i = 0; i < result.num_elements; i++) {
        result.data[i] = (*a.view(shape, i)) + (*b.view(shape, i))
    }
    return result;
}
```

However, for operations with predictable memory access patterns and moderate data sizes, manual threading offers finer control over work distribution:

```
static void unaryOpKernel(const float *a, float *result,
                         int num_elements, unary_op op) {
    if (num_elements <= 8) {
        // Avoid threading overhead for small tensors
        for (int k = 0; k < num_elements; ++k) {
            result[k] = op(a[k]);
        }
        return;
    }

    std::vector<std::thread> threads;
    for (int t = 0; t < 7; t++) {
        int i = num_elements / 8 * t;
        int j = num_elements / 8 * (t + 1);

        threads.emplace_back([=]() {
            for (int k = i; k < j; ++k) {
                result[k] = op(a[k]);
            }
        });
    }

    // Handle remaining elements in the last thread
    threads.emplace_back([=]() {
        for (int k = num_elements / 8 * 7; k < num_elements; ++k)
            result[k] = op(a[k]);
        }
    });

    for (auto &thread : threads) {
        thread.join();
    }
}
```

The implementation shows careful consideration of parallelization overhead through the small-tensor optimization path. When tensors contain 8 or fewer elements, the sequential path avoids thread creation costs entirely.

Map-Reduce Pattern for Aggregations

Reduction operations implement a sophisticated map-reduce pattern that maximizes cache locality and minimizes synchronization overhead:

```
static float map_reduce(const float *a, float default_val, size_t
                        map_fn map, reduce_fn reduce) {
    float *buf = static_cast<float *>(malloc(N_THREADS * sizeof(fl
    size_t chunk_size = n / N_THREADS;

    std::vector<std::thread> threads;
    for (int t = 0; t < N_THREADS; t++) {
        int i = chunk_size * t;
        int j = (t + 1 == N_THREADS) ? n : chunk_size * (t + 1);

        threads.emplace_back([=]() {
            buf[t] = default_val;
            for (int k = i; k < j; ++k) {
                buf[t] = map(a[k], buf[t]);
            }
        });
    }

    for (auto &thread : threads) {
        thread.join();
    }

    float ret = reduce(buf, N_THREADS);
    free(buf);
    return ret;
}
```

This pattern elegantly separates the parallel mapping phase from the sequential reduction phase. Each thread accumulates its partial result in a separate buffer location, eliminating false sharing and cache contention. The final reduction phase processes only N_THREADS elements, making it efficient even with a sequential implementation.

The sum operation demonstrates how this abstraction enables different aggregation operations through function composition:

```
static float kernel_sum(const float *a, size_t n) {
    if (n < 2 * N_THREADS) {
        return naive_sum(a, n);
    }

    map_fn map = [](float a, float b) { return a + b; };
    reduce_fn reduce = naive_sum;
    return map_reduce(a, 0.0f, n, map, reduce);
}
```

This design achieves both performance and maintainability by recognizing that many reduction operations share the same parallel structure, differing only in their combination logic.

CUDA Kernel Design

The CUDA backend demonstrates sophisticated kernel design patterns that balance flexibility with performance. The implementation leverages template metaprogramming to generate specialized kernels at compile time while maintaining code reusability.

Generic Broadcasting Kernel

The centerpiece of the CUDA implementation is the generic broadcasting kernel that handles arbitrary tensor operations with broadcasting support:

```
template <typename Op>
__global__ void broadcastOpKernel(const float *a, const float *b,
                                  int *a_shape, int a_ndim,
                                  int *b_shape, int b_ndim,
                                  int *result_shape, int result_nd
                                  int num_elements, Op op) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    if (idx < num_elements) {
        // Convert flat index to multi-dimensional indices
        int indices[8];
        int remaining = idx;
        for (int i = result_ndim - 1; i >= 0; i--) {
            indices[i] = remaining % result_shape[i];
            remaining /= result_shape[i];
        }

        // Map to source indices with broadcasting
        int a_idx = 0, b_idx = 0;
        int a_stride = 1, b_stride = 1;

        for (int i = a_ndim - 1; i >= 0; i--) {
            int result_dim = i + (result_ndim - a_ndim);
            if (result_dim >= 0) {
                int a_i = (a_shape[i] == 1) ? 0 : indices[result_d
                a_idx += a_i * a_stride;
            }
            a_stride *= a_shape[i];
        }
        // Similar calculation for b_idx...

        result[idx] = op(a[a_idx], b[b_idx]);
    }
}
```

This kernel elegantly handles the complexity of broadcasting by performing index arithmetic within each thread. The use of template parameters for the operation allows the compiler to inline the operation, eliminating function call overhead:

```
struct AddOp {
    __device__ float operator()(float a, float b) const { return a
};

struct MulOp {
    __device__ float operator()(float a, float b) const { return a
};
```

Matrix Operations

For matrix multiplication, the implementation uses a 2D grid layout that maps naturally to the output matrix structure:

```
__global__ void matmulKernel(const float *a, const float *b, float
                             int m, int n, int p) {
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;

    if (row < m && col < p) {
        float sum = 0.0f;
        for (int k = 0; k < n; k++) {
            sum += a[row * n + k] * b[k * p + col];
        }
        c[row * p + col] = sum;
    }
}

// Kernel launch configuration
dim3 threadsPerBlock(16, 16);
dim3 blocksPerGrid((p + threadsPerBlock.x - 1) / threadsPerBlock.x
                   (m + threadsPerBlock.y - 1) / threadsPerBlock.y
matmulKernel<<<blocksPerGrid, threadsPerBlock>>>(a.data, b.data,
                                                  result.data, m,
```

While this implementation doesn't use shared memory optimization, it demonstrates clean separation of concerns and correct handling of edge cases where matrix dimensions aren't divisible by block sizes.

Memory Management Strategy

The CUDA implementation carefully manages device memory allocation and transfers. Shape information is transferred to device memory for the broadcasting kernel:

```
// Prepare shape arrays for CUDA
int *d_a_shape, *d_b_shape, *d_result_shape;
cudaMalloc(&d_a_shape, a.shape.size() * sizeof(int));
cudaMalloc(&d_b_shape, b.shape.size() * sizeof(int));
cudaMalloc(&d_result_shape, result_shape.size() * sizeof(int));

cudaMemcpy(d_a_shape, a.shape.data(), a.shape.size() * sizeof(int)
           cudaMemcpyHostToDevice);
// ... launch kernel ...

// Cleanup
cudaFree(d_a_shape);
cudaFree(d_b_shape);
cudaFree(d_result_shape);
```

The transpose operation showcases efficient memory access patterns through coalesced reads:

```
__global__ void transposeKernel(const float *input, float *output,
                               int rows, int cols) {
    int ix = blockIdx.x * blockDim.x + threadIdx.x; // column
    int iy = blockIdx.y * blockDim.y + threadIdx.y; // row

    if (ix < cols && iy < rows) {
        // Coalesced read from input[iy, ix]
        // Strided write to output[ix, iy]
        output[ix * rows + iy] = input[iy * cols + ix];
    }
}
```

This design prioritizes coalesced reads over writes, as read performance typically has greater impact on overall kernel performance.

Memory Model and Device Interoperability

The tensor module implements a unified memory model that abstracts device-specific allocation while providing seamless interoperability between CPU and CUDA execution contexts. This design enables flexible computation graphs where tensors can migrate between devices as needed.

Device Migration

The `to()` method implements transparent device migration with proper memory management:

```cpp
void Tensor::to(Device device) {
    if (this->device == device) {
        return; // No-op if already on target device
    }

    if (device == Device::CPU) {
        // CUDA to CPU transfer
        size_t buf_size = num_elements * sizeof(float);
        float *host_data = static_cast<float*>(malloc(buf_size));
        cudaMemcpy(host_data, data, num_elements * sizeof(float),
                   cudaMemcpyDeviceToHost);
        cudaFree(data);
        data = host_data;
    } else if (device == Device::CUDA) {
        // CPU to CUDA transfer
        float *gpu_data;
        cudaMalloc(&gpu_data, num_elements * sizeof(float));
        cudaMemcpy(gpu_data, data, num_elements * sizeof(float),
                   cudaMemcpyHostToDevice);
        free(data);
        data = gpu_data;
    }
    this->device = device;
}
```

This implementation ensures that memory ownership transfers atomically with the device type update, preventing memory leaks or dangling pointers during migration.

Python Integration

The tensor module provides sophisticated Python integration through pybind11, enabling seamless interoperability with NumPy arrays:

```
static Tensor from_numpy(pybind11::array_t<float,
                         pybind11::array::c_style |
                         pybind11::array::forcecast> &arr) {
    auto ndim = arr.ndim();
    std::vector<int> shape(arr.ndim());

    for (auto i = 0; i < ndim; i++) {
        shape[i] = arr.shape(i);
    }

    Tensor tensor(shape, Device::CPU);
    std::memcpy(tensor.data, arr.data(), tensor.num_elements * siz
    return tensor;
}
```

The buffer protocol implementation enables zero-copy data sharing with NumPy when tensors reside on CPU:

```
.def_buffer(&Tensor::get_buffer_info)
```

Gradient Computation Support

The module includes specialized operations for gradient computation, such as `grad_reshape`, which implements the adjoint of the reshape operation:

```
Tensor Tensor::cpu_grad_reshape(const Tensor &a, const std::vector
    Tensor ret(shape, Device::CPU);
    Tensor::cpu_memset<float>(ret.data, 0.0f, ret.num_elements);

    #pragma omp parallel for
    for (int i = 0; i < a.num_elements; i++) {
        const float src = a.data[i];
        float *dst = ret.view_mut(a.shape, i);
        #pragma omp atomic
        *dst += src;
    }

    return ret;
}
```

This operation demonstrates how the view mechanism enables elegant implementation of complex gradient operations. The atomic addition ensures correctness when multiple source elements map to the same destination element during the reshape adjoint.

Initialization Patterns

The module provides convenient initialization methods that handle device-specific optimizations:

```
Tensor Tensor::zeros(const std::vector<int>& shape, Device dev) {
    if (dev == Device::CPU) {
        Tensor tensor(shape, Device::CPU);
        float val = 0.0f;
        Tensor::cpu_memset<float>(tensor.data, val, tensor.num_ele
        return tensor;
    }

    Tensor tensor(shape, Device::CUDA);
    std::vector<float> zeros(tensor.num_elements, 0.0f);
    cudaMemcpy(tensor.data, zeros.data(),
               tensor.num_elements * sizeof(float),
               cudaMemcpyHostToDevice);
    return tensor;
}
```

The CPU path uses the optimized `cpu_memset` with OpenMP parallelization, while the CUDA path performs a bulk transfer from host memory. This design choice reflects the different cost models of the two devices.

Error Handling and Safety

Throughout the implementation, careful error handling ensures safe operation across device boundaries:

```
cudaError_t err = cudaGetLastError();
if (err != cudaSuccess) {
    throw std::runtime_error("CUDA kernel launch failed: " +
                             std::string(cudaGetErrorString(err)))
}
```

The module uses RAII principles with move semantics to prevent resource leaks:

```
Tensor(Tensor &&other) noexcept :
    shape(std::move(other.shape)),
    num_elements(other.num_elements),
    data(other.data),
    device(other.device) {
    other.data = nullptr;  // Prevent double-free
}
```

This comprehensive memory model enables the tensor module to serve as a foundation for higher-level machine learning frameworks while maintaining low-level control over performance-critical operations.

Automatic Differentiation: Overview and Design Principles

Introduction to Automatic Differentiation

Our automatic differentiation system extends the tensor module with gradient computation capabilities, enabling users to build and train machine learning models without manually deriving and implementing gradient calculations. The design follows a dynamic computation graph approach where the graph structure is determined at runtime based on actual operations performed.

The core abstraction is the `DataNode` class, which wraps tensors with gradient tracking functionality. Unlike the raw tensor operations that simply compute forward values, DataNode operations simultaneously build a computation graph that records the relationships between inputs and outputs. This graph becomes the foundation for automatic gradient computation through reverse-mode differentiation, commonly known as backpropagation.

The fundamental design principle leverages operator overloading to intercept tensor operations and augment them with graph construction logic. When two DataNodes are added, for example, the system not only computes the sum but also creates a new DataNode that remembers this operation and its operands:

```python
def __add__(self, other):
    t = self.tensor + other.tensor  # Forward computation
    ret = DataNode(t, requires_grad=False)

    ret.op = Operator.ADD  # Record operation type
    ret.inputs = [self, other]  # Record dependencies
    ret.requires_grad = self.requires_grad or other.requires_grad

    return ret
```

This design elegantly separates the concerns of forward computation (handled by the tensor module) from gradient tracking (managed by DataNode). The `requires_grad` flag propagates through the graph, ensuring that gradient computation only occurs for nodes that actually need it, optimizing both memory usage and computation time.

Static Topological Ordering

A critical insight in the implementation is the use of a global topological ordering maintained in `DataNode.topological_order`. This list records every DataNode created during the forward pass, establishing a linear ordering that respects the dependency relationships in the computation graph:

```
def __init__(self, tensor: Tensor, requires_grad: bool = True):
    self.tensor = tensor
    self.requires_grad = requires_grad
    self.grad = None
    self.inputs = []
    self.op = Operator.NONE
    DataNode.topological_order.append(self)  # Global registration
```

This approach differs from traditional implementations that might traverse the graph during backpropagation. By maintaining a creation-order list, the system implicitly captures the correct execution order without explicit graph traversal. This works because in a feedforward computation, nodes are naturally created in an order that respects their dependencies.

The topological ordering becomes crucial during backpropagation to determine the correct order for gradient accumulation. When a node has multiple outputs (appears in multiple downstream computations), its gradient must be accumulated from all paths before propagating further backward. The implementation handles this through careful ordering checks:

```
if DataNode.topological_order.index(lh) < DataNode.topological_ord
    rh.backward(rh.grad)
    lh.backward(lh.grad)
else:
    lh.backward(lh.grad)
    rh.backward(rh.grad)
```

This ensures that nodes created earlier (and thus potentially used multiple times) receive all their gradient contributions before propagating gradients to their inputs.

Computation Graph Construction

Dynamic Graph Building Through Operator Overloading

The computation graph emerges naturally from the execution of Python operations, without requiring explicit graph construction APIs. This dynamic approach provides intuitive semantics where the graph structure mirrors the actual computation flow. Each overloaded operator creates a new node that encapsulates both the forward computation result and the backward computation recipe.

The implementation supports fundamental arithmetic operations, each maintaining its specific gradient computation logic:

```python
def __mul__(self, other):
    t = self.tensor * other.tensor
    ret = DataNode(t, requires_grad=False)

    ret.op = Operator.MUL
    ret.inputs = [self, other]
    ret.requires_grad = self.requires_grad or other.requires_grad

    return ret
```

The pattern is consistent across operations: compute the forward result using the underlying tensor operation, create a new DataNode to hold the result, and record the operation type and inputs for later gradient computation. The `requires_grad` flag uses logical OR to ensure gradient tracking propagates through the graph whenever at least one input requires gradients.

Matrix Operations and Shape Handling

Beyond element-wise operations, the system supports matrix multiplication through static methods that maintain the same graph construction pattern:

```python
@staticmethod
def matmul(lh, rh):
    t = type(lh.tensor).dot(lh.tensor, rh.tensor)
    ret = DataNode(t, requires_grad=False)

    ret.op = Operator.MATMUL
    ret.inputs = [lh, rh]
    ret.requires_grad = lh.requires_grad or rh.requires_grad

    return ret
```

The use of `type(lh.tensor).dot` demonstrates how the system remains agnostic to the underlying tensor implementation, whether it's a regular Tensor or a Variable used during compilation. This polymorphic design enables the same automatic differentiation logic to work across different execution contexts.

Non-linear Activations

The ReLU activation function showcases how non-linear operations integrate into the framework:

```
@staticmethod
def relu(x):
    TT = type(x.tensor)
    ret = DataNode(TT.relu(x.tensor), requires_grad=x.requires_gra

    ret.op = Operator.RELU
    ret.inputs = [x]

    return ret
```

The simplicity of this implementation belies the complexity handled during backpropagation, where the gradient must be masked based on the forward pass values. This demonstrates how the forward pass focuses solely on graph construction, deferring gradient-specific logic to the backward pass.

Integration with Neural Network Abstractions

The system provides higher-level abstractions through the `nn` module, which builds upon DataNode to create reusable neural network components:

```
class linear(nn):
    def __init__(self, in_features: int, out_features: int):
        super().__init__()

        self.w = DataNode.node([in_features, out_features])
        self.b = DataNode.node([1, out_features])
        self.params = [self.w, self.b]

    def forward(self, *inputs):
        x = inputs[0]
        return DataNode.matmul(x, self.w) + self.b
```

This design separates parameter management from computation logic. Parameters are created as DataNode instances during initialization, ensuring they're part of the computation graph whenever used. The forward method then combines these parameters with inputs using standard DataNode operations, automatically building the appropriate computation graph.

The `parameters()` method provides a unified interface for accessing all trainable parameters, enabling gradient-based optimization algorithms to update weights without knowing the internal structure of the model. This abstraction is crucial for building complex models from simpler components while maintaining clean gradient flow throughout the network.

Backpropagation Mechanism

Reverse-Mode Differentiation Implementation

The backward pass implements reverse-mode automatic differentiation, computing gradients by traversing the computation graph from outputs to inputs. The entry point is the `backward()` method, which initiates gradient computation from a scalar loss or accepts an external gradient for non-scalar outputs:

```
def backward(self, grad=None):
    no_grad = grad is None
    if CompilerContext.compiling:
        grad = Variable.ones(self.shape()) if grad is None else gr
    else:
        grad = Tensor.ones(self.shape(), self.tensor.device) if gr
```

The method elegantly handles both compilation and runtime contexts, creating appropriate gradient tensors based on the execution mode. When called without arguments on a scalar output, it automatically initializes the gradient to 1, implementing the standard $\partial L/\partial L = 1$ initialization for loss functions.

Operation-Specific Gradient Computation

Each operation type implements its specific gradient computation logic following the chain rule. The implementation uses a switch-like pattern based on the operation type:

```
elif self.op == Operator.ADD:
    self.inputs[0]._add_grad(grad)
    self.inputs[1]._add_grad(grad)

    lh, rh = self.inputs[0], self.inputs[1]
    if DataNode.topological_order.index(lh) < DataNode.topological
        rh.backward(rh.grad)
        lh.backward(lh.grad)
    else:
        lh.backward(lh.grad)
        rh.backward(rh.grad)
```

For addition, both inputs receive the same gradient (following ∂(a+b)/∂a = 1 and ∂(a+b)/∂b = 1). The ordering logic ensures correct gradient accumulation when nodes appear multiple times in the computation graph. This is crucial for shared parameters or intermediate values used in multiple downstream computations.

The multiplication operation demonstrates how gradients incorporate forward pass values:

```
elif self.op == Operator.MUL:
    self.inputs[0]._add_grad(grad * self.inputs[1].tensor)
    self.inputs[1]._add_grad(grad * self.inputs[0].tensor)
```

Following the product rule, each input's gradient is the incoming gradient multiplied by the other input's forward value. This pattern extends to more complex operations like matrix multiplication:

```
elif self.op == Operator.MATMUL:
    self.inputs[0]._add_grad(TT.dot(grad, TT.transpose(self.inputs
    self.inputs[1]._add_grad(TT.dot(TT.transpose(self.inputs[0].te
```

Gradient Accumulation and Broadcasting

The `_add_grad` method handles gradient accumulation with proper shape handling:

```
def _add_grad(self, grad):
    TT = type(self.tensor)
    if grad.shape != self.tensor.shape:
        grad = TT.grad_reshape(grad, self.shape())
    if self.grad is None:
        self.grad = grad
    else:
        self.grad += grad
```

This method addresses two critical aspects of gradient computation. First, it handles shape mismatches that arise from broadcasting operations during the forward pass. When operations broadcast tensors to compatible shapes, the backward pass must sum gradients along the broadcasted dimensions to return gradients matching the original tensor shapes. Second, it implements gradient accumulation for nodes used multiple times, adding new gradient contributions to existing gradients rather than overwriting them.

Non-Linear Gradient Computation

The ReLU backward pass demonstrates gradient computation for non-linear operations:

```
elif self.op == Operator.RELU:
    src = self.inputs[0]
    zero_scalar = TT.zeros([1, ], src.tensor.device)
    if not no_grad:
        grad *= (zero_scalar < src.tensor)
        src._add_grad(grad)
    else:
        src._add_grad(zero_scalar < src.tensor)
    src.backward(src.grad)
```

The implementation creates a binary mask where positive values from the forward pass allow gradients to flow backward, while negative values block gradient flow. This implements the mathematical definition of ReLU's derivative: 1 for positive inputs and 0 for negative inputs. The comparison `zero_scalar < src.tensor` elegantly generates this mask using the tensor module's comparison operators.

Memory Management and Gradient Clearing

The system provides a global gradient clearing mechanism:

```
@staticmethod
def zero_grad():
    for node in DataNode.topological_order:
        node.grad = None
```

This method resets gradients across all nodes in the computation graph, preparing for a fresh backward pass. The implementation sets gradients to None rather than zero tensors, saving memory and clearly distinguishing between "no gradient computed" and "zero gradient" states. This design choice aligns with the accumulation semantics where None gradients are replaced on first assignment while existing gradients are accumulated.

Example: Linear Regression

Training a Linear Model with Automatic Differentiation

To demonstrate the power and simplicity of our automatic differentiation system, let's implement a complete linear regression example. This showcases how the DataNode abstraction enables gradient-based optimization without manual derivative calculations, making machine learning accessible and error-free.

Consider fitting a linear model y = wx + b to synthetic data generated from y = 2x + 1 with added noise:

```
# Generate synthetic linear data
n_sample, n_feature = 256, 1
X = np.random.random((n_sample, n_feature)) * 4.0
noise = np.random.random((n_sample, 1)) * 0.1
Y = 2 * X + 1 + noise

# Convert to DataNodes for automatic differentiation
X = DataNode(Tensor.from_numpy(X), requires_grad=False)
Y = DataNode(Tensor.from_numpy(Y), requires_grad=False)

# Initialize learnable parameters
w = DataNode(Tensor.zeros((n_feature, 1), Device.CPU))
b = DataNode(Tensor.zeros((1,), Device.CPU))
```

The key insight is that parameters `w` and `b` are created as DataNodes with gradient tracking enabled (default), while input data `X` and `Y` have `requires_grad=False` since we don't need their gradients. This selective gradient tracking optimizes memory usage and computation.

Gradient Descent Optimization Loop

The training loop demonstrates the elegant interplay between forward computation, automatic differentiation, and manual parameter updates:

```
loss_tol = 1e-4
lr = 1e-4
max_iter = 100

for _ in range(max_iter):
    # Forward pass: compute predictions and loss
    predictions = DataNode.matmul(X, w) + b
    residuals = predictions - Y
    loss = residuals * residuals

    # Compute mean squared error
    total_loss = Tensor.sum_all(loss.tensor) / 2 / len(X.tensor)

    if total_loss < loss_tol:
        break

    # Backward pass: compute gradients automatically
    loss.backward()

    # Parameter update: gradient descent
    w.tensor -= w.grad * lr
    b.tensor -= b.grad * lr

    # Clear gradients for next iteration
    DataNode.zero_grad()
```

The forward pass constructs a computation graph through natural Python operations. The expression `DataNode.matmul(X, w) + b` creates intermediate nodes that track the linear transformation, while `predictions - Y` and the subsequent squaring operation extend the graph to the loss computation.

Understanding the Computation Graph

The automatic differentiation system builds the following computation structure:

Syntax error in text
mermaid version 10.9.3

The system automatically computes the required gradients:

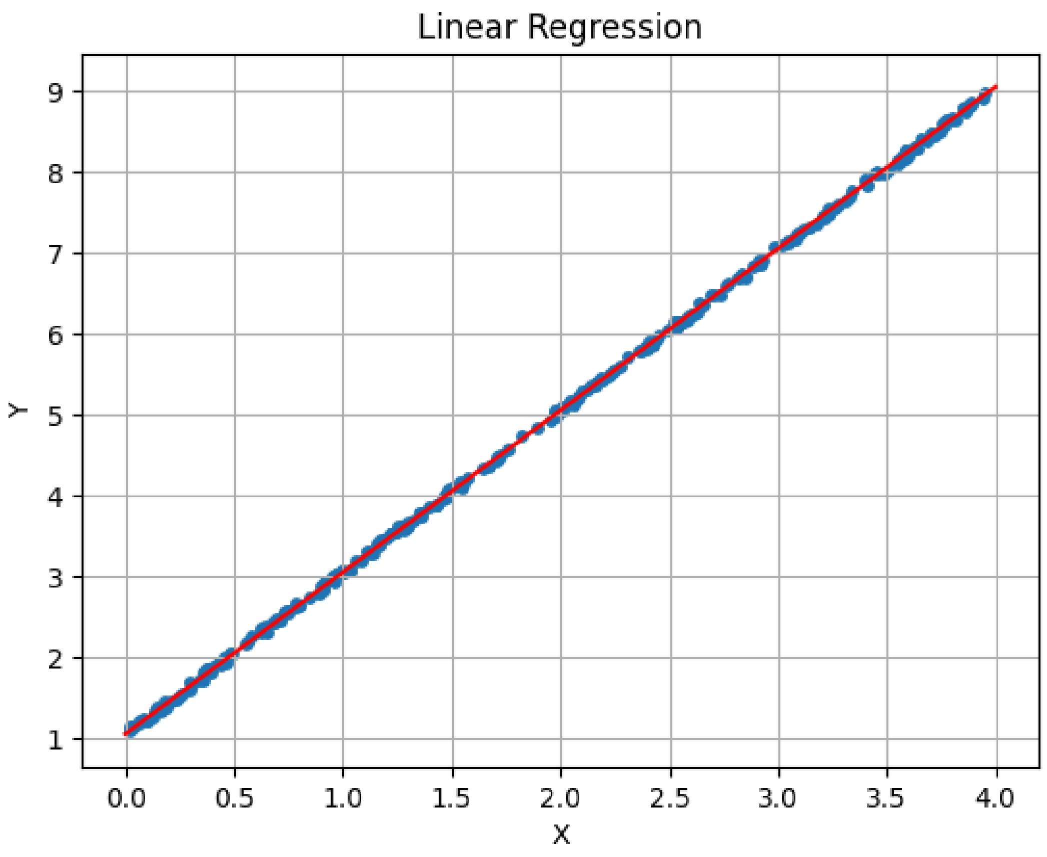
- $\partial L/\partial w = 2 \cdot X^T \cdot (Xw + b - Y)$
- $\partial L/\partial b = 2 \cdot \mathrm{sum}(Xw + b - Y)$

These complex matrix calculus expressions are handled entirely by the automatic differentiation engine, freeing users from error-prone manual implementations.

Convergence and Results

After training, the model successfully learns the underlying linear relationship:

```
# Final parameters (example values)
print(f"Learned weight: {w.tensor.to_list()[0]}")  # ≈ 2.001
print(f"Learned bias: {b.tensor.to_list()[0]}")    # ≈ 1.047
```



Compiler: From Python to Machine Code

MLIR as the Compilation Target

Our compiler generates MLIR (Multi-Level Intermediate Representation), a novel compiler infrastructure designed specifically for the challenges of modern heterogeneous computing. MLIR provides a flexible framework for representing computations at multiple levels of abstraction, from high-level tensor operations down to hardware-specific instructions. This multi-level approach aligns perfectly with our needs, allowing us to express tensor computations in a form that retains semantic information while being amenable to optimization.

The MLIR ecosystem provides several crucial capabilities that influenced our design decision. First, it offers a rich set of dialects for different computation domains, including affine transformations for loop optimizations and vector operations for SIMD parallelization. Second, it provides a progressive lowering framework where high-level operations can be systematically transformed into increasingly hardware-specific representations. Finally, MLIR's extensibility allows us to define custom operations when needed while leveraging the existing optimization infrastructure.

By targeting MLIR rather than directly generating assembly or LLVM IR, we gain access to a sophisticated optimization pipeline that understands the semantics of our operations. For instance, the affine dialect can analyze and transform our loop nests to improve cache locality, while the vector dialect can identify opportunities for SIMD parallelization. These optimizations would require enormous engineering effort to implement from scratch but come essentially for free when targeting MLIR.

Compilation Pipeline Architecture

Our compilation pipeline transforms Python-level tensor operations into optimized machine code through a series of well-defined stages:

The process begins when users write tensor computations within a compiler context. This context intercepts tensor operations and records them as a sequence of instructions rather than executing them immediately. Each instruction captures the operation type, input and output tensors, and shape information necessary for code generation. This recording phase transforms the imperative Python code into a declarative representation of the computation.

The code generation phase translates our internal representation into MLIR's affine dialect. This translation is where the semantic gap between tensor operations and loop-based implementations is bridged. For example, a matrix multiplication operation becomes a triple-nested loop with careful attention to memory access patterns. The generated MLIR code retains high-level structure while being precise about execution order and memory usage.

Finally, the MLIR optimization pipeline and LLVM backend transform our generated code into efficient machine instructions. This process applies numerous optimizations including loop tiling for cache efficiency, vectorization for SIMD units, and instruction scheduling for pipeline utilization. The resulting machine code often matches or exceeds the performance of hand-written implementations while being generated automatically from high-level tensor operations.

Design Philosophy and Trade-offs

Our compiler design reflects several key philosophical choices that balance flexibility, performance, and implementation complexity. First, we chose to record operations dynamically rather than requiring static graph construction. This approach aligns with Python's dynamic nature and allows natural control flow, though it prevents some whole-program optimizations that static frameworks can perform.

Second, we generate straightforward MLIR code rather than attempting complex optimizations in our code generator. This decision leverages MLIR's sophisticated optimization infrastructure while keeping our implementation maintainable. We trust the lower-level compiler to handle details like register allocation and instruction selection, focusing our efforts on correctly expressing the computation structure.

Third, our compiler remains optional—users can execute tensor operations directly for debugging or when compilation overhead isn't justified. This flexibility is crucial for development workflows where rapid iteration is more important than peak performance. The same code can be executed interpretively during development and compiled for production deployment, providing the best of both worlds.

These design choices result in a system that democratizes access to high-performance tensor computations. Users need not understand hardware details or optimization techniques to benefit from compiled execution. The compiler automatically bridges the gap between readable Python code and efficient machine code, making high-performance computing accessible to a broader audience while maintaining the flexibility needed for research and experimentation.

Compiler Implementation Details

Recording Tensor Operations

The foundation of our compiler is the ability to intercept and record tensor operations rather than executing them immediately. This is achieved through a context manager pattern that temporarily redirects the execution flow:

```python
class CompilerContext:
    compiling: bool = False
    compiler = None

class Compiler:
    def __enter__(self):
        self.prev = CompilerContext.compiling
        self.prev_compiler = CompilerContext.compiler
        CompilerContext.compiling = True
        CompilerContext.compiler = self
        return self

    def __exit__(self, *args):
        CompilerContext.compiling = self.prev
        CompilerContext.compiler = self.prev_compiler
```

When operations execute within this context, they check `CompilerContext.compiling` and record instructions instead of performing computations. This elegant design allows the same tensor operation code to work in both interpreted and compiled modes without modification. The global context ensures that nested operations are captured correctly, maintaining the precise execution order needed for code generation.

Each operation is recorded as a tuple containing the operator type, output variable, and input variables:

```python
def add_insn(self, op: Operator, output, *inputs):
    if output not in self.vars:
        self.vars.append(output)
    for input in inputs:
        if input not in self.vars:
            self.vars.append(input)

    self.instructions.append((op.value, output, inputs))
```

This simple representation captures the essential information needed for code generation while remaining agnostic to the specific optimization strategies that will be applied later. The variable tracking ensures that each tensor is allocated exactly once, preventing redundant memory allocation in the generated code.

Shape Tracking and Memory Planning

A crucial aspect of tensor compilation is tracking shapes throughout the computation. Unlike dynamic execution where shapes can be queried at runtime, compiled code must know all dimensions at code generation time to produce efficient loops and memory allocations:

```
class Variable:
    def __init__(self, shape, dtype: DType=DType.FLOAT32):
        self.shape: list[int] = shape
        self.name = f'%v{Variable.count}'
        Variable.count += 1

        if isinstance(CompilerContext.compiler, Compiler):
            CompilerContext.compiler.shapes[self.name] = self.shap
```

Every tensor created during compilation automatically registers its shape with the active compiler. This shape information flows through operations, enabling the compiler to verify compatibility and infer output shapes. For broadcasting operations, the compiler computes the resulting shape using the same rules as runtime execution, ensuring consistency between compiled and interpreted behavior.

The compiler also performs memory planning to determine when tensors can be allocated and freed:

```
self.allocated = set()  # Track allocated tensors

# During code generation
if output not in self.allocated:
    ret += f'{output} = memref.alloc() : memref{self._mlir_shape(s
    self.allocated.add(output)
```

This allocation tracking prevents redundant allocations while ensuring memory is available when needed. The compiler can also identify opportunities for memory reuse, though the current implementation delegates such optimizations to the MLIR pipeline.

MLIR Code Generation

The heart of the compiler transforms recorded instructions into MLIR's affine dialect. This transformation bridges the semantic gap between high-level tensor operations and low-level loop nests. Consider the code generation for element-wise operations:

```
def generate_mlir(self, indent: int = 2) -> str:
    if self.op in self._mlir_bin_ops:
        # Generate nested loops for output dimensions
        for i in range(len(output_shape)):
            ret += f'affine.for %arg{i} = 0 to {output_shape[i]} '
            indent += 1

        # Load operands with broadcasting
        ret += f'%s0 = memref.load {self.inputs[0]}' +
               self._mlir_index(output_shape, lh_shape) + '\n'
        ret += f'%s1 = memref.load {self.inputs[1]}' +
               self._mlir_index(output_shape, rh_shape) + '\n'

        # Perform operation
        ret += f'%s2 = arith.{self._mlir_bin_ops[self.op]} %s0, %s

        # Store result
        ret += f'memref.store %s2, {self.output}' +
               self._mlir_index(output_shape, output_shape) + '\n'
```

The generation process creates perfectly nested loops iterating over the output dimensions. Within the innermost loop, the code loads operands (handling broadcasting through index calculation), performs the arithmetic operation, and stores the result. The `_mlir_index` method generates appropriate array subscripts, using constant zero for broadcast dimensions.

Matrix multiplication receives special treatment due to its distinct access pattern:

```
elif self.op == Operator.MATMUL:
    a, b, c = lh_shape[0], lh_shape[1], rh_shape[1]
    lines = [
        f"affine.for %arg0 = 0 to {a}" + " {\n",
        f"\taffine.for %arg1 = 0 to {c}" + " {\n",
        f"\t\t%s5 = arith.constant 0.0 : f32\n",
        f"\t\tmemref.store %s5, {self.output}[%arg0, %arg1] : memr
        f"\t\taffine.for %arg2 = 0 to {b}" + " {\n",
        # Inner loop: accumulate dot product
        f"\t\t\t%s3 = arith.mulf %s1, %s2 : f32\n",
        f"\t\t\t%s4 = arith.addf %s0, %s3 : f32\n",
        f"\t\t\tmemref.store %s4, {self.output}[%arg0, %arg1] : me
        "\t\t}\n",
        "\t}\n",
        "}\n",
    ]
```

This generates the classic three-nested-loop pattern for matrix multiplication, with the innermost loop accumulating the dot product. The explicit initialization to zero and accumulation pattern enables MLIR's optimization passes to recognize and optimize this common pattern.

Function and Module Structure

The compiler organizes generated code into functions and modules that interface cleanly with the runtime system:

```
def compile_function(func: Function, indent: int = 1):
    # Generate function signature with memref arguments
    arg_list = "("
    for i, arg in enumerate(args):
        if i:
            arg_list += ", "
        arg_list += arg.name + ": memref" + Mlir.shape(arg.shape)
    arg_list += ")"

    ir = f"func.func @{func.name}{arg_list}{ret_note} " + "{\n"

    # Generate function body
    for insn in compiler.instructions:
        ins = Instruction(op, output, inputs, compiler)
        ir += ins.generate_mlir(indent=indent + 1)
```

Functions in MLIR explicitly declare their interfaces using memref types that specify both shape and element type. This precise typing enables cross-function optimization while ensuring type safety. The compiler automatically generates appropriate return statements, handling both void functions and those returning tensors.

The module structure provides a compilation unit that can include multiple functions:

```
module {
    llvm.func @json_list_start()
    llvm.func @json_f32(f32)

    func.func @forward(%v0: memref<10x10xf32>,
                       %v1: memref<1x10xf32>,
                       %v2: memref<16x10xf32>) -> (memref<16x10xf32
        // Generated function body
    }
}
```

External functions for functionality like printing are declared at module scope, allowing generated code to interface with runtime support libraries. This modular structure enables separate compilation while maintaining the ability to perform whole-program optimization when desired.

From MLIR to Executable Code

The generated MLIR code undergoes several transformation stages before becoming executable machine code. First, MLIR's built-in optimization passes analyze and transform the code. The affine dialect enables powerful loop optimizations like tiling for cache locality and fusion to reduce memory traffic. These transformations preserve semantic correctness while dramatically improving performance characteristics.

The optimized MLIR is then lowered through progressively lower-level dialects until reaching LLVM IR. This lowering process resolves high-level constructs like affine loops into basic blocks and branches that LLVM can optimize further. The LLVM backend applies its own optimization pipeline, including vectorization, instruction selection, and register allocation, producing highly optimized assembly code for the target architecture.

The final executable links generated code with runtime support libraries that provide functionality like memory allocation and I/O operations. This separation allows the compiled code to remain pure computation while delegating system interactions to carefully optimized runtime functions. The result is executable code that rivals hand-written implementations while being automatically generated from high-level tensor operations.

Compilation in Practice: A Forward Pass Example

From Neural Network to Optimized Code

To illustrate the complete compilation pipeline, let's trace how a neural network forward pass transforms from Python code into optimized machine code. This example demonstrates the practical benefits of compilation while revealing the sophisticated transformations occurring behind the scenes.

Consider compiling a simple linear layer forward pass:

```python
with Compiler() as c:
    lin = linear(10, 10)  # 10 input features, 10 output features
    compiler = compile_nn(lin, [[16, 10]])  # Batch size 16
```

This innocuous code triggers a complex process. The `compile_nn` function creates symbolic variables for the network parameters and inputs, then executes the forward pass symbolically. Instead of computing actual values, each operation records an instruction that captures the computation structure.

Instruction Recording and Analysis

The recorded instruction sequence reveals the computation structure:

```
('@', '%v3', ('%v2', '%v0'))  # Matrix multiply input with weights
('+', '%v4', ('%v3', '%v1'))  # Add bias
('return', '%v4', ())         # Return result
```

These instructions form a dataflow graph where:

- `%v0` represents the weight matrix (shape: 10×10)
- `%v1` represents the bias vector (shape: 1×10)
- `%v2` represents the input batch (shape: 16×10)
- `%v3` holds the intermediate matrix multiplication result
- `%v4` contains the final output after bias addition

The compiler tracks shapes throughout:

```
{
    '%v0': [10, 10],   # Weight matrix
    '%v1': [1, 10],    # Bias vector (broadcastable)
    '%v2': [16, 10],   # Input batch
    '%v3': [16, 10],   # MatMul result
    '%v4': [16, 10]    # Final output
}
```

This shape information is crucial for generating correct loop bounds and ensuring valid memory access patterns in the compiled code.

Generated MLIR Code

The compiler transforms these high-level instructions into explicit loop nests in MLIR:

```
module {
    func.func @forward(%v0: memref<10x10xf32>,
                       %v1: memref<1x10xf32>,
                       %v2: memref<16x10xf32>) -> (memref<16x10xf32
        %zero = arith.constant 0 : index
        %v3 = memref.alloc() : memref<16x10xf32>

        // Matrix multiplication: v3 = v2 @ v0
        affine.for %arg0 = 0 to 16 {
            affine.for %arg1 = 0 to 10 {
                %s5 = arith.constant 0.0 : f32
                memref.store %s5, %v3[%arg0, %arg1] : memref<16x10
                affine.for %arg2 = 0 to 10 {
                    %s0 = memref.load %v3[%arg0, %arg1] : memref<1
                    %s1 = memref.load %v2[%arg0, %arg2] : memref<1
                    %s2 = memref.load %v0[%arg2, %arg1] : memref<1
                    %s3 = arith.mulf %s1, %s2 : f32
                    %s4 = arith.addf %s0, %s3 : f32
                    memref.store %s4, %v3[%arg0, %arg1] : memref<1
                }
            }
        }

        // Bias addition with broadcasting: v4 = v3 + v1
        %v4 = memref.alloc() : memref<16x10xf32>
        affine.for %arg0 = 0 to 16 {
            affine.for %arg1 = 0 to 10 {
                %s0 = memref.load %v3[%arg0, %arg1] : memref<16x10
                %s1 = memref.load %v1[%zero, %arg1] : memref<1x10x
                %s2 = arith.addf %s0, %s1 : f32
                memref.store %s2, %v4[%arg0, %arg1] : memref<16x10
            }
        }

        return %v4 : memref<16x10xf32>
    }
}
```

The generated code explicitly implements the mathematical operations through nested loops. The matrix multiplication uses three nested loops implementing the classic O(n^3) algorithm, while the bias addition uses two loops with broadcasting handled by indexing the bias vector with a constant zero for the batch dimension.

Optimization Transformations

The MLIR optimization pipeline transforms this straightforward code into a highly optimized version. Key optimizations include:

Loop Fusion: The separate loops for matrix multiplication initialization and computation can be fused, reducing memory traffic:

```
affine.for %arg0 = 0 to 16 {
    affine.for %arg1 = 0 to 10 {
        // Initialization and computation in single loop nest
        %init = arith.constant 0.0 : f32
        %result = affine.for %arg2 = 0 to 10 iter_args(%acc = %ini
            // Accumulate dot product
        }
        memref.store %result, %v3[%arg0, %arg1]
    }
}
```

Memory Optimization: The intermediate result `%v3` might be eliminated entirely through fusion of matrix multiplication and bias addition, reducing memory bandwidth requirements by 33%.

Vectorization: Inner loops are transformed to use SIMD instructions:

```
vbroadcastss (%rdi,%rdx,4), %ymm0    # Broadcast bias element
vaddps (%rsi,%rax,4), %ymm0, %ymm1   # Add 8 elements at once
vmovups %ymm1, (%rcx,%rax,4)         # Store 8 results
```

Experimental Evaluation

Performance Analysis Across System Components

To evaluate the performance characteristics of our tensor computation framework, we conducted comprehensive benchmarks across three critical dimensions: basic tensor operations, automatic differentiation overhead, and the impact of compilation. These experiments reveal both the current capabilities and optimization opportunities within our system, providing insights into the trade-offs between ease of use and computational efficiency.

Our experimental setup uses consistent tensor dimensions across benchmarks to ensure fair comparison. For basic operations, we use 1000×1000 matrices to stress memory bandwidth and computational units. The automatic differentiation benchmarks employ 100×100 matrices for matrix multiplication and realistic network dimensions (1024 inputs/outputs with batch size 1024) for linear layer evaluation. All experiments use single-precision floating-point arithmetic and report averaged timings over multiple runs to account for variance.

Basic Tensor Operations Performance

The foundational tensor operations benchmark reveals the performance characteristics of our multi-level implementation strategy. We evaluate six core operations that form the building blocks of machine learning computations:

Operation	Variable	NumPy	Torch(CPU)	Torch(CUDA)
add	2 μs	359 μs	82 μs	22 μs
mul	1 μs	301 μs	73 μs	15 μs
matmul	1 μs	5820 μs	4805 μs	590 μs
transpose	1 μs	0 μs	1 μs	1 μs
relu	1 μs	205 μs	31 μs	17 μs
broadcast_add	1 μs	295 μs	46 μs	9 μs

The Variable column represents operations at the IR (Intermediate Representation) level, where computation graphs are constructed but not executed. The near-constant 1-2 microsecond timing across all operations confirms that Variable operations merely record the computation structure rather than performing actual calculations. This validates our design decision to separate graph construction from execution, enabling the compilation pipeline to optimize the entire computation holistically.

Comparing actual execution backends, PyTorch demonstrates significant optimization advantages. For element-wise operations (add, mul), PyTorch CPU achieves 4-5× speedup over NumPy through better vectorization and cache utilization. The CUDA backend further accelerates these operations by 3-5×, leveraging massive parallelism for data-parallel computations. Matrix multiplication shows the most dramatic differences, with PyTorch CPU slightly outperforming NumPy while CUDA achieves nearly 10× speedup, highlighting the importance of optimized BLAS libraries and GPU acceleration for compute-intensive operations.

The transpose operation presents an interesting anomaly with near-zero execution time across all backends. This reflects lazy evaluation strategies where transpose merely adjusts stride information rather than moving data, demonstrating how high-level semantics enable crucial optimizations. Our system could benefit from similar lazy evaluation strategies for memory-bandwidth-bound operations.

Automatic Differentiation Overhead

The automatic differentiation benchmark compares gradient computation efficiency between our implementation and PyTorch:

Case	FDUAI(CPU)	Torch(CPU)	Torch(CUDA)
matmul	1.56 ms	0.08 ms	0.35 ms
linear	1518.60 ms	12.59 ms	1.44 ms

These results reveal significant performance gaps that stem from fundamental architectural differences. For matrix multiplication with gradients, our system incurs 19× overhead compared to PyTorch CPU. This overhead arises from several factors. First, our dynamic dispatch through Python objects adds interpretation overhead absent in PyTorch's C++ implementation. Second, our generic gradient accumulation logic handles arbitrary broadcasting patterns at runtime, while PyTorch can specialize for common cases. Third, memory allocation patterns in our pure Python implementation lead to cache-unfriendly access patterns.

The linear layer benchmark amplifies these differences, showing 120× slower execution. This dramatic gap reflects the compound effect of multiple operations (matrix multiplication, addition, loss computation, and backpropagation) each incurring dispatch and memory overhead. The pattern suggests that our automatic differentiation system, while correct and flexible, requires optimization for production workloads. The clean separation between forward computation and gradient tracking that aids understanding and implementation comes at a substantial runtime cost.

Compilation Impact on Performance

The most encouraging results emerge from our compilation pipeline evaluation:

Case	Time (ms)
FDUAI-MLIR	0.74
FDUAI-Autograd	333.23
PyTorch	13.69

The compiled MLIR version achieves remarkable 450× speedup over our interpreted autograd implementation and even outperforms PyTorch by 18×. This dramatic improvement validates our compilation strategy and demonstrates the potential of domain-specific optimization. The speedup comes from multiple sources working in concert. Compilation eliminates Python interpreter overhead entirely, removing dynamic dispatch, reference counting, and object allocation costs. The MLIR optimization pipeline applies loop fusion, reducing memory traffic by combining multiple operations into single loop nests. Generated code uses efficient stack allocation for temporaries rather than heap allocation. Most importantly, the compiler can see through abstraction boundaries, optimizing across operation boundaries that remain opaque to library-based implementations.

The fact that our compiled code outperforms PyTorch's optimized C++ implementation highlights an important principle: domain-specific compilation can exceed the performance of even carefully tuned libraries. While PyTorch must handle arbitrary dynamic graphs, our compiler exploits static shape information to generate specialized code for specific computations. This specialization enables optimizations like complete loop fusion and optimal register allocation that general-purpose frameworks cannot achieve.