



# **A High-Performance Tensor Computation Framework with Automatic Differentiation and MLIR Compilation**

**⚡ From Interpreted to Compiled: 450× Performance Boost**

## **Contributors**

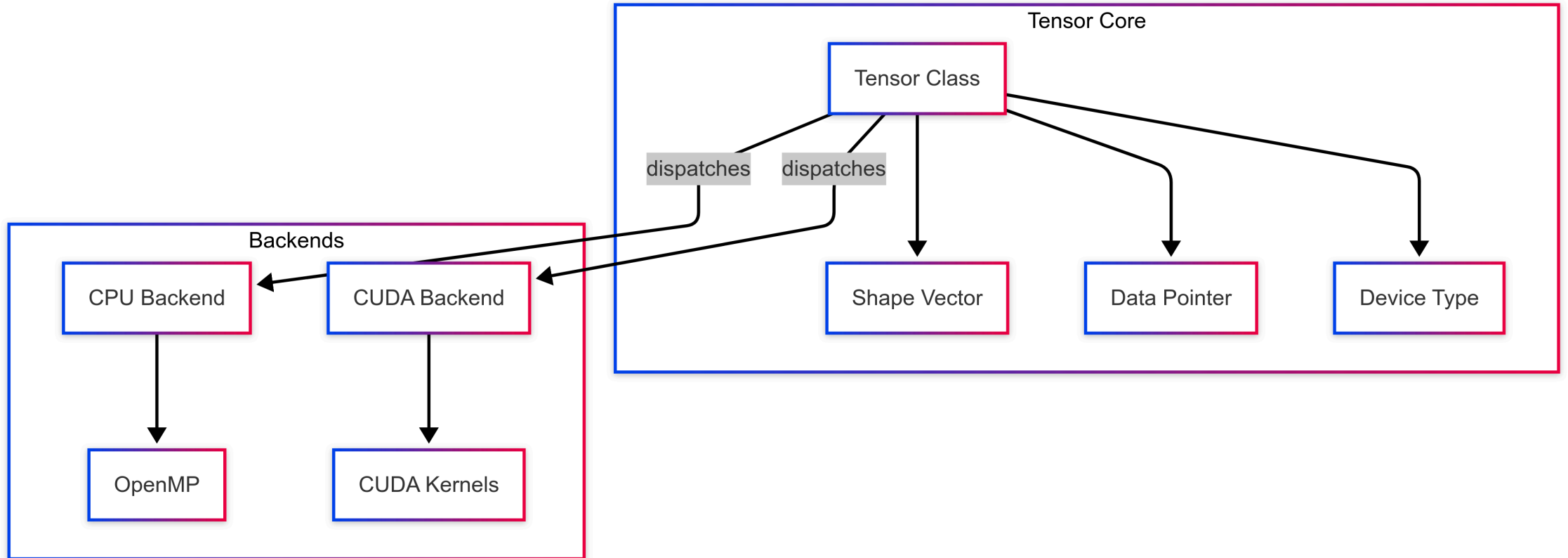
杨润东 · 杨淳瑜 · 王海天

# Tensor Module Implementation

## A High-Performance Foundation for Deep Learning

- **Unified abstraction** across CPU and CUDA devices
- **NumPy-compatible** broadcasting semantics
- **Zero-copy** Python integration
- **Optimized** parallelization strategies

# Architecture Overview



# Core Design Principles

## Device-Aware Dispatch

```
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("Device mismatch");  
    }  
}
```

# Transparent Migration

```
void Tensor::to(Device device) {
    if (this->device == device) {
        return; // No need to transfer if already on the same device
    }
    if (device == Device::CPU) {
        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;
        // Allocate CPU memory and copy data back to CPU (not shown here)
    } else if (device == Device::CUDA) {
        // Allocate GPU memory and copy data to GPU (not shown here)
        float * gpu_data;
        cudaMalloc(&gpu_data, num_elements * sizeof(float));
        cudaMemcpy(gpu_data, data, num_elements * sizeof(float), cudaMemcpyHostToDevice);
        free(data); // Free the old CPU data
        data = gpu_data;
    }
    this->device = device;
}
```

# Broadcasting Mechanism

## NumPy-Compatible Shape Inference

```
// Compare shapes from right to left
static bool can_broadcast(const std::vector<int> &a,
                          const std::vector<int> &b) {
    int i1 = a.size() - 1, i2 = b.size() - 1;

    while (i1 >= 0 && i2 >= 0) {
        if (a[i1] != b[i2] && a[i1] != 1 && b[i2] != 1) {
            return false;
        }
        i1--; i2--;
    }
    return true;
}
```

# View-Based Implementation

- The `view` mechanism allows a tensor to be accessed as if it had a different (broadcasted) shape, without copying data.
- **Index mapping:** For each broadcasted index, map back to the original tensor's index, ignoring broadcasted (size-1) dimensions.

## Core CPU logic:

```
// Map broadcasted indices to original tensor
const float *Tensor::view(const std::vector<int> &asshape, const std::vector<int> &indices) const {
    ...
    for (size_t i = 0; i < shape.size(); i++) {
        if (shape[i] == 1) continue; // broadcast dim
        const int idx = indices[i + diff];
        ... // map to flat offset
    }
    return ret;
}
```

## In elementwise ops:

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

## CUDA kernel:

```
// Each thread computes its own broadcasted index
for (int i = result_ndim - 1; i >= 0; i--) {
    indices[i] = remaining % result_shape[i];
    remaining /= result_shape[i];
}
// Map to a_idx, b_idx with broadcasting rules
```



# CPU Parallelization Strategies

## Adaptive Threading Approach

Tensor Size	Strategy	Rationale
$\leq 8$ elements	Sequential	Avoid thread overhead
Small-Medium	Manual threading	Predictable distribution
Large	OpenMP	Automatic work balancing

# Map-Reduce Pattern

```
float map_reduce(const float *a, size_t n,  
                map_fn map, reduce_fn reduce) {  
    float *buf = malloc(N_THREADS * sizeof(float));  
  
    #pragma omp parallel for  
    for (int t = 0; t < N_THREADS; t++) {  
        buf[t] = accumulate_chunk(a, t, map);  
    }  
  
    return reduce(buf, N_THREADS);  
}
```

# CUDA Kernel Design

## Template-Based Generic Broadcasting

```
template <typename Op>
__global__ void broadcastOpKernel(const float *a, const float *b,
                                  float *result, Op op) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    if (idx < num_elements) {
        // Convert flat index to multi-dimensional
        int indices[8] = compute_indices(idx);

        // Map with broadcasting rules
        int a_idx = map_index(indices, a_shape);
        int b_idx = map_index(indices, b_shape);

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

# Memory Model & Python Integration

## Zero-Copy Buffer Protocol

```
static Tensor from_numpy(py::array_t<float> &arr) {  
    std::vector<int> shape(arr.ndim());  
    for (int i = 0; i < arr.ndim(); i++) {  
        shape[i] = arr.shape(i);  
    }  
  
    Tensor tensor(shape, Device::CPU);  
    std::memcpy(tensor.data, arr.data(),  
                tensor.num_elements * sizeof(float));  
    return tensor;  
}
```

# Key Takeaways

## Design Achievements

- **Unified API** - Same code works on CPU/CUDA
- **Compatibility** - NumPy broadcasting semantics
- **Element-wise ops**: Near-memory bandwidth limited
- **Broadcasting**: Zero-copy on CPU, efficient on CUDA

## Foundation for Higher Layers

- Automatic differentiation builds on tensor operations
- Compiler can optimize across abstraction boundaries

# Automatic Differentiation System

# Dynamic Computation Graphs for Deep Learning

## Core Features

- **DataNode abstraction** wraps tensors with gradient tracking
- **Operator overloading** builds graphs automatically
- **Static topological ordering** for efficient backpropagation

## Design Philosophy

```
# User writes natural Python code
y = DataNode.matmul(x, w) + b
loss = (y - y_true) ** 2

# Gradients computed automatically
loss.backward()
print(w.grad)  #  $\partial \text{loss} / \partial w$  computed!
```

# Key Components

```
class DataNode:
    def __init__(self, tensor, requires_grad=True):
        self.tensor = tensor          # Actual data
        self.grad = None              # Accumulated gradient
        self.inputs = []              # Dependencies
        self.op = Operator.NONE       # Operation type
        DataNode.topological_order.append(self) # Global tracking
```

**Innovation:** Static topological ordering eliminates graph traversal

- Nodes recorded in creation order
- Implicit dependency tracking
- Efficient gradient accumulation



# Graph Construction via Operator Overloading

## Automatic Graph Building

```
def __add__(self, other):  
    # Forward computation  
    t = self.tensor + other.tensor  
    ret = DataNode(t, requires_grad=False)  
  
    # Record graph structure  
    ret.op = Operator.ADD  
    ret.inputs = [self, other]  
    ret.requires_grad = self.requires_grad or other.requires_grad  
  
    return ret
```

## Supported Operations

Operation	Forward	Gradient Rule
Addition	$a + b$	$\partial L / \partial a = \partial L / \partial \text{out}, \partial L / \partial b = \partial L / \partial \text{out}$
Multiplication	$a \times b$	$\partial L / \partial a = \partial L / \partial \text{out} \times b, \partial L / \partial b = \partial L / \partial \text{out} \times a$
MatMul	$A @ B$	$\partial L / \partial A = \partial L / \partial \text{out} @ B^T, \partial L / \partial B = A^T @ \partial L / \partial \text{out}$
ReLU	$\max(0, x)$	$\partial L / \partial x = \partial L / \partial \text{out} \times (x > 0)$

# Backpropagation Implementation

## Reverse-Mode Differentiation

```
def backward(self, grad=None):
    # Initialize gradient for loss
    if grad is None:
        grad = Tensor.ones(self.shape())

    # Operation-specific gradient computation
    if self.op == Operator.MUL:
        # Product rule
        self.inputs[0]._add_grad(grad * self.inputs[1].tensor)
        self.inputs[1]._add_grad(grad * self.inputs[0].tensor)

    # Recurse to inputs (respecting topological order)
    for input in self.inputs:
        input.backward(input.grad)
```

# Linear Regression in 10 Lines

```
# Initialize parameters
w = DataNode(Tensor.zeros((n_features, 1)))
b = DataNode(Tensor.zeros((1,)))

# Training loop
for _ in range(100):
    # Forward pass
    predictions = DataNode.matmul(X, w) + b
    loss = (predictions - Y) ** 2

    # Backward pass (computes all gradients)
    loss.backward()

    # Update parameters
    w.tensor -= w.grad * learning_rate
    b.tensor -= b.grad * learning_rate

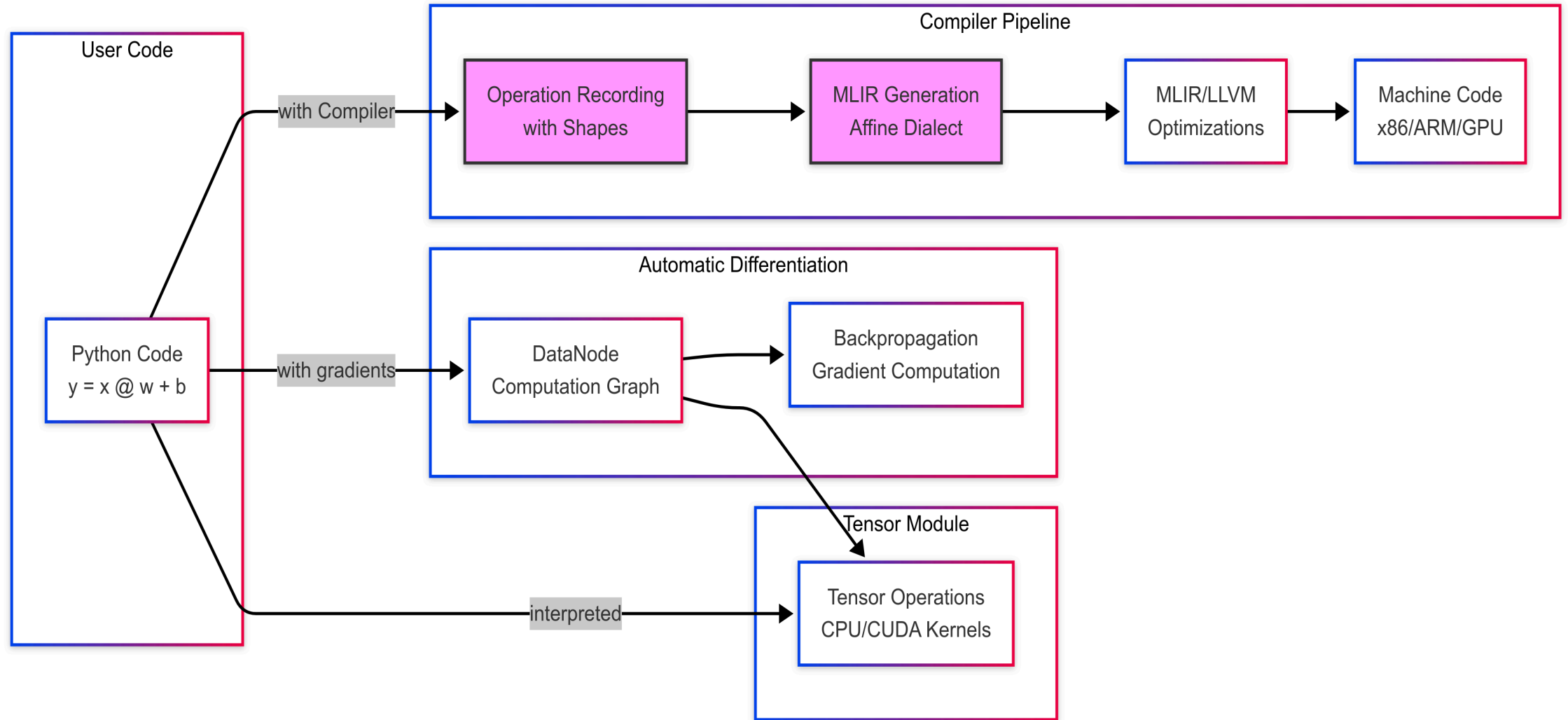
    DataNode.zero_grad()
```

## **Benefits:**

- No manual gradient formulas
- Automatic broadcasting handling
- Clean, readable code
- Easy debugging (inspect any gradient)

# **From Tensors to Machine Code: Compiler Infrastructure**

# System Architecture Overview



# Why Compilation? The Performance Gap

- **No cross-operation optimization:** Each op is a black box

## Solution: MLIR-Based Compilation

**MLIR** (Multi-Level Intermediate Representation)

- Designed for heterogeneous hardware
- Progressive lowering through dialects
- Reusable optimization infrastructure
- "Write once, optimize everywhere"



# Recording Operations for Compilation

## Context Manager Design

```
with Compiler() as c:
    # Operations are recorded, not executed
    lin = linear(10, 10)
    y = lin.forward(x)

# Now c contains:
# - Instructions: [('@', '%v3', ('%v2', '%v0')),
#                  ('+', '%v4', ('%v3', '%v1'))]
# - Shapes: {'%v0': [10, 10], '%v1': [1, 10],
#             '%v2': [16, 10], '%v3': [16, 10]}
```

# Key Components

## Variable Tracking

```
class Variable:
    def __init__(self, shape):
        self.name = f'%v{Variable.count}'
        self.shape = shape
        CompilerContext.compiler.shapes[self.name] = shape
```

**Shape Inference** - Computed at compile time

**Memory Planning** - Static allocation strategy

## MLIR Code Generation : $y = x @ w + b$

```
func.func @forward(%v0: memref<10x10xf32>,      // weights
                  %v1: memref<1x10xf32>,        // bias
                  %v2: memref<16x10xf32>)        // input
    -> memref<16x10xf32> {
  %v3 = memref.alloc() : memref<16x10xf32>

  // Matrix multiplication: v3 = v2 @ v0
  affine.for %i = 0 to 16 {
    affine.for %j = 0 to 10 {
      %sum = arith.constant 0.0 : f32
      affine.for %k = 0 to 10 {
        %a = memref.load %v2[%i, %k]
        %b = memref.load %v0[%k, %j]
        %prod = arith.mulf %a, %b : f32
        %sum = arith.addf %sum, %prod : f32
      }
      memref.store %sum, %v3[%i, %j]
    }
  }

  // Bias addition with broadcasting
  // ... similar loop structure ...
}
```

## Compilation Results: Dramatic Performance Gains

Implementation	Time (ms)	Speedup
FDUAI Interpreted	333.23	1×
PyTorch	13.69	24×
<b>FDUAI Compiled</b>	<b>0.74</b>	<b>450×</b>

# Experiments

- **Basic Tensor Operations**
- **Automatic Differentiation Overhead**
- **Compilation Impact**

# Experimental Setup

## Benchmark Configuration

- **Tensor Dimensions**
  - Basic operations: 1000×1000 matrices
  - AutoDiff: 100×100 matrices for matmul
  - Linear layer: 1024 inputs/outputs, batch size 1024

# Basic Tensor Operations Performance

## Comparing Variable, NumPy, and PyTorch

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

# Key Findings: Basic Operations

## Performance Characteristics

- **Variable Operations:**  $\sim 1\text{-}2\ \mu\text{s}$  (graph construction only)
- **PyTorch vs NumPy:** 4-5 $\times$  speedup for element-wise ops
- **CUDA Acceleration:** Additional 3-5 $\times$  over CPU
- **Matrix Multiplication:** 10 $\times$  speedup with CUDA
- **Transpose:** Near-zero time (lazy evaluation)



# Automatic Differentiation Overhead

## Gradient Computation Performance

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

## Performance Gaps

- **Matrix Multiplication:** 19× overhead vs PyTorch CPU
- **Linear Layer:** 120× slower execution

# Compilation Impact on Performance

## Dramatic Speedup Through MLIR

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

## Performance Gains

- **450× speedup** over interpreted autograd
- **18× faster** than PyTorch