Creating a Tensor

You can create a Tensor in two primary ways: by specifying its properties directly as arguments, or by bundling them into a TensorOptions object for convenience. The constructors have default parameters, allowing for multiple ways to create a tensor.

Method 1: Direct Construction

This method involves passing the tensor's properties, such as its shape and data type, directly to the constructor.

Syntax

The primary constructor can be called with 3 or 4 arguments.

```
 C++
 // 4-parameter version (fully specified)
Tensor(Shape shape, Dtype dtype, DeviceIndex device, bool requires_grad);

// 3-parameter version (requires_grad defaults to false)
Tensor(Shape shape, Dtype dtype, DeviceIndex device);
```

Examples

A. With 4 Parameters (Shape, Dtype, Device, and `requires_grad`)

This gives you full control over the tensor's properties, including enabling gradient tracking.

```
C++
// A 4x4 tensor of 32-bit floats on the first GPU, with gradients enabled
Shape shape = {4, 4};
Tensor gpu_tensor(shape, Dtype::Float32, Device::CUDA, true);
```

B. With 3 Parameters (Shape, Dtype, and Device)

This is the most common way to create a tensor for inference or general computation. The requires_grad flag defaults to false.

```
C++
// A 10x20 tensor of 32-bit integers on the CPU
Shape shape = {10, 20};
Tensor cpu_tensor(shape, Dtype::Int32, Device::CPU);
```

Method 2: Construction with TensorOptions

This method provides a more readable and organized way to define a tensor's properties by grouping them into a single TensorOptions object. It is the standard way to create a tensor with just two arguments.

Syntax

This constructor always takes 2 arguments.

```
C++
// 2-parameter version
Tensor(Shape shape, TensorOptions opts);
```

Examples

C. With 2 Parameters (Shape and Options)

You first configure a TensorOptions object and then pass it along with the shape to create the tensor. This is very useful when creating many tensors with the same configuration.

```
C++
 // 1. Configure the options
TensorOptions options;
options.dtype = Dtype::Float16;
options.device = Device::CUDA(0);

// 2. Create a 128x128 tensor using the defined options
Shape shape = {128, 128};
Tensor tensor_from_opts(shape, options);

// 3. Reuse the same options to create another tensor
Shape another_shape = {64, 32, 8};
Tensor another_tensor(another_shape, options);
```

Tensor Constructor Parameters

This guide details the parameters used to create a Tensor. Understanding these is key to configuring your tensors correctly.

1. shape - Shape{{...}}

- **Purpose**: Defines the dimensions and size of the tensor. It dictates the number of elements and how they are organized.
- **Type**: Shape
- **Accepted Values**: A Shape object is constructed using an initializer list {{...}} containing one or more positive integers.
    - `{10}`: A 1-dimensional tensor (vector) with 10 elements.
    - `{64, 128}`: A 2-dimensional tensor (matrix) with 64 rows and 128 columns.
    - `{16, 3, 224, 224}`: A 4-dimensional tensor, commonly used for batches of images in computer vision.
- **How to access specific dimension:** This shape object itself is just a vector, so to access those specific dimensions we can do like this,

---

2. dtype - Dtype::Name

- **Purpose**: Specifies the numerical data type for every element in the tensor. This affects memory usage and numerical precision.
- **Type**: Dtype
- **Accepted Values**: An enumeration of the supported data types.
    - `Dtype::Float32`: Standard 32-bit floating-point numbers.
    - `Dtype::Float64`: Double-precision 64-bit floating-point numbers.
    - `Dtype::Float16`: Half-precision 16-bit floating-point numbers.
    - `Dtype::Bfloat16`: An alternative 16-bit floating-point format.
    - `Dtype::Int16`: 16-bit signed integers.
    - `Dtype::Int32`: 32-bit signed integers.
    - `Dtype::Int64`: 64-bit signed integers.

---

3. device - DeviceIndex(Device::Name, index: number)

- **Purpose**: Determines the hardware device where the tensor's data will be allocated and where computations will be performed.
- **Type**: DeviceIndex
- **Accepted Values**: A DeviceIndex object is typically created from the Device class.
    - **With default device index:**
        - `Device::CPU`: Allocates the tensor on the main system memory (CPU).

        - `Device::CUDA`: Allocates the tensor on the first CUDA-enabled GPU (device index 0).
    - **With manual device index - telling which specific device you want to use:**
        - `DeviceIndex(Device::CPU, number)`: Allocates the tensor on the main system memory (CPU). using number on a multicore cpu system is possible, not fully configured for it
        - `DeviceIndex(Device::CUDA, number)`: Allocates the tensor on the first CUDA-enabled GPU device, uses cuda detect device functionality to do and we can allocate to any device if it exists.

---

4. requires_grad

- **Purpose**: A boolean flag that indicates whether the tensor should track its computational history to compute gradients with respect to it. This is essential for training neural networks.
- **Type**: bool
- **Accepted Values**:
    - `true`: Allocates an additional buffer to store gradients. Operations involving this tensor will be tracked by the automatic differentiation engine.
    - `false`: (Default) The tensor is a standard data container. No gradient buffer is allocated, and no operations are tracked. This is used for inference, input data, or any computation where gradients are not needed.

Accessing Tensor Information

1. Getting Tensor Properties (Metadata)

You can easily inspect the core properties of a tensor using its accessor methods. These methods do not perform any computation; they simply return the configuration you defined when the tensor was created.

.shape()

Returns the shape of the tensor.

- **Return Type**: `Shape`

Example:

```
 C++
 Shape shape = {{10, 20}};
Tensor my_tensor(shape, Dtype::Float32, Device::CPU);

// Get the shape
```

```
Shape retrieved_shape = my_tensor.shape(); // Will be {10, 20}
```

```
Shape square = {{10, 100}};
square.shape().dims[0] ⇒ 10
square.shape().dims[1] ⇒ 100
```

.dtype()

Returns the data type of the tensor's elements.

- **Return Type**: `Dtype`

Example:

```
C++
Tensor my_tensor({{4, 4}}, Dtype::Int16, Device::CPU);
Dtype tensor_type = my_tensor.dtype(); // Will be Dtype::Int16
```

.device()

Returns the device where the tensor is located.

- **Return Type**: `DeviceIndex`

Example:

```
C++
Tensor my_tensor(P{4, 4}}, Dtype::Float32, Device::CUDA(0));
DeviceIndex tensor_device = my_tensor.device(); // Will be Device::CUDA
```

.numel()

Returns the total number of individual elements in the tensor. This is the product of all its dimensions.

- **Return Type**: `size_t`

Example:

```
C++
Tensor my_tensor({{5, 10, 2}}, Dtype::Float32, Device::CPU);
size_t total_elements = my_tensor.numel(); // Will be 100 (5 * 10 * 2)
```

.nbytes()

Returns the total size of the tensor's data buffer in bytes.

- **Return Type**: `size_t`

Example:

```
 C++
// A 10-element tensor of Float32 (4 bytes each)
Tensor my_tensor({{10}}, Dtype::Float32, Device::CPU);
size_t total_bytes = my_tensor.nbytes(); // Will be 40 or slightly more due
to memory alignment (nearest 8 bytes alignment)
```

.ndim()

Returns the number of dimensions of the tensor. This is equivalent to the length of the shape vector.

- **Return Type**: `int64_t`
- **Example**:
  - For a tensor with shape `{10}`, **.ndim()** returns 1.
  - For a tensor with shape `{16, 3, 224, 224}`, **.ndim()** returns 4.

.stride()

Returns the stride of the tensor. The stride defines the number of elements to jump in memory to move one step along each dimension.

- **Return Type**: `const Stride&`
- **Example**: For a contiguous 3x4 tensor of floats, the stride would be {4, 1}. To move to the next row (dimension 0), you jump 4 elements. To move to the next column (dimension 1), you jump 1 element.

.dtype_size(Dtype d)

A static utility function that returns the size in bytes for a given Dtype.

- **Return Type**: `size_t`
- **Example**: `Tensor::dtype_size(Dtype::Float32)` would return 4.

.is_cpu()

Checks if the tensor is located on the CPU.

- **Return Type**: `bool`

- Returns: true if the tensor resides in CPU memory, false otherwise.

.is_cuda()

Checks if the tensor is located on a CUDA-enabled GPU.

- **Return Type**: `bool`
- **Returns**: true if the tensor resides in GPU memory, false otherwise.

.nbytes()

Returns the total number of bytes consumed by the tensor's elements if it were contiguous.

It is calculated as `numel() * dtype_size()`.

- **Return Type**: `size_t`

.grad_nbytes()

Returns the total number of bytes for the gradient buffer, if it exists.

- **Return Type**: `size_t`
- **Returns**: The size of the gradient buffer if requires_grad is true, otherwise returns 0.

.allocated_bytes()

Returns the actual size in bytes of the allocated memory block for the tensor's data. This can be slightly larger than .nbytes() due to memory alignment optimizations.

- **Return Type**: `size_t`

.grad_allocated_bytes()

Returns the actual size in bytes of the allocated memory block for the tensor's gradients.

- **Return Type**: `size_t`
- **Returns**: The allocated size of the gradient block if requires_grad is true, otherwise may return 0.

.owns_data()

Checks if this tensor object owns its underlying data memory.

- **Return Type**: `bool`
- **Returns**: true if the tensor allocated and is responsible for freeing its data memory. Returns false if it is a "view" that simply points to another tensor's memory.

.owns_grad()

Checks if this tensor object owns its underlying gradient memory.

- **Return Type**: `bool`
- **Returns**: true if requires_grad was set and this tensor is responsible for the gradient memory.

.is_contiguous()

Checks if the tensor's elements are stored sequentially in memory in row-major order.

- **Return Type**: `bool`
- **Returns**: true if the elements are laid out in one continuous block, false otherwise. Views created from operations like transpose() are often not contiguous.

2. Accessing Raw Tensor Data and Gradient

To read or modify the values inside a tensor, you need to get a raw pointer to its memory. The .data<T>() method provides this access.

Warning: Direct memory access is powerful but unsafe. You are responsible for using the correct data type and respecting the tensor's memory boundaries. Mismatched types will lead to incorrect data or crashes.

.data() [.grad()]

Returns a raw C++ pointer to the first element of the tensor's data. You must specify the data type T that matches the tensor's Dtype.

- **Return Type**: T* (e.g., float*, int*)
- **Template Parameter T**: The C++ type corresponding to the tensor's Dtype.

Common Type Mappings

Dtype	C++ Type T
Int64	int64_t
Int32	int32_t
Int16	int16_t
Float64	double
Float32	float

Float16	float16_t
Bfloat16	bfloat16_t

Example: Writing to a CPU Tensor

This example shows how to get a pointer to a tensor's data and fill it with values.

```
C++
// Create a 1D tensor with 5 elements
Tensor my_tensor({{5}}, Dtype::Float32, Device::CPU);

// 1. Get a pointers to the data and gradients, specifying the correct type
float* data_ptr = my_tensor.data<float>();
float* grad_ptr = my_tensor.grad();

// 2. Use the pointer to write values into the tensor's memory
for (int i = 0; i < my_tensor.numel(); ++i) {
    data_ptr[i] = 1.5f * i;
    grad_ptr[i] = -1.5f;
}

// The tensor now contains [0.0, 1.5, 3.0, 4.5, 6.0]
// The gradients would now be [-1.5, -1.5, -1.5, -1.5, -1.5]
```

Example: Reading from a CPU Tensor

This example shows how to read and print the values from a tensor.

```
C++
// Assume my_tensor is the tensor from the previous example
// 1. Get a read-only pointer to the data
const float* data_ptr = my_tensor.data<float>();
const float* grad_ptr = my_tensor.grad();

// 2. Read and print the values
for (int i = 0; i < my_tensor.numel(); ++i) {
    std::cout << data_ptr[i] << std::endl;
    std::cout << grad_ptr[i] << std::endl;
}
```

Note 1: Basically the gradients would be the same type as data and it would not break since we will use float types for model anyway, but we will look into making float type only for gradients irrespective of what the input datatype is.

Note 2: To access data on a CUDA tensor, you must first move it to the CPU using the .to_cpu() method before calling .data<T>(), as you cannot directly access GPU memory from standard C++ code. We will cover device transfers in the next section.

Factory Functions

Factory functions are static methods on the Tensor class that provide a convenient way to create tensors with pre-initialized data, such as all zeros, ones, or random numbers. These methods are easier to use than using the main constructor and manually filling the data.

Note: All factory functions accept a shape and a TensorOptions object which should definitely have a Datatype attribute, allowing you to easily specify the desired dtype and device.

Tensor::zeros()

Creates a tensor with all its elements initialized to 0.

Syntax

```
C++
static Tensor zeros(Shape shape, TensorOptions opts = {});
```

Parameters

- shape **(Shape)**: The desired shape for the output tensor (e.g., {2, 3}).
- opts **(TensorOptions, optional)**: A configuration object to specify the dtype and device. If omitted, the tensor will be a Float32 tensor on the CPU.

Examples

A. Basic CPU Zeros Tensor
Creates a 2x3 tensor of type Float32 on the CPU.

```
C++
// Creates a tensor: [[0., 0., 0.],
//                    [0., 0., 0.]]
Shape shape = {2, 3}; Tensor z = Tensor::zeros(shape);
```

B. GPU Zeros Tensor with a Specific Type
Creates a 4x4 tensor of 32-bit integers on the first GPU.

```cpp
Shape shape = {4, 4};
TensorOptions options;
options.dtype = Dtype::Int32;
options.device = Device::CUDA(0);

Tensor z_gpu = Tensor::zeros(shape, options);
```

Tensor::ones()

Creates a tensor with all its elements initialized to 1.

Syntax

```cpp
static Tensor ones(Shape shape, TensorOptions opts = {});
```

Parameters

- shape **(Shape)**: The desired shape for the output tensor.
- opts **(TensorOptions, optional)**: A configuration object to specify dtype and device.

Examples

A. Basic CPU Ones Tensor

```cpp
// Creates a tensor: [[1., 1., 1.],
//                    [1., 1., 1.]]
Shape shape = {2, 3};
Tensor o = Tensor::ones(shape);
```

B. GPU Ones Tensor

Creates a 10x10 tensor of Float32 on the first GPU.

```cpp
Shape shape = {10, 10};
TensorOptions options;
options.device = Device::CUDA(0);

Tensor o_gpu = Tensor::ones(shape, options);
```

Tensor::full()

Creates a tensor with all elements initialized to a specified scalar value.

Syntax

```
C++
static Tensor full(Shape shape, TensorOptions opts, float val);
```

Description

This is a more general version of zeros and ones, allowing you to create a tensor filled with any value you choose.

Parameters

- shape **(Shape)**: The desired shape for the output tensor.
- opts **(TensorOptions)**: A configuration object to specify dtype and device.
- val **(float)**: The scalar value to fill the tensor with.

Example

Creates a 3x3 tensor on the CPU, with dtype Float32, filled with the value 42.0.

```
C++
Shape shape = {3, 3};
TensorOptions options;
options.device = Device::CPU;
// Creates a tensor: [[42., 42., 42.],
//                    [42., 42., 42.],
//                    [42., 42., 42.]]
Tensor f = Tensor::full(shape, options, 42.0f);
```

Tensor::rand()

Creates a tensor with elements sampled from a **uniform distribution** on the interval [0, 1).

Syntax

```
C++
static Tensor rand(Shape shape, TensorOptions opts = {});
```

Description

Ideal for creating tensors with random noise or for initializing weights in a neural network where a uniform distribution is desired. The values will be evenly distributed between 0 (inclusive) and 1 (exclusive).

Parameters

- shape **(Shape)**: The desired shape for the output tensor.
- opts **(TensorOptions, optional)**: A configuration object. Note that rand typically works with floating-point dtypes.

Example

Creates a 2x2 tensor with random float values on the CPU.

```
C++
// Creates a tensor like: [[0.123, 0.987],
//                         [0.456, 0.789]]
// (Values will be different each time)
Shape shape = {2, 2};
Tensor r = Tensor::rand(shape);
```

Tensor::randn()

Creates a tensor with elements sampled from a **standard normal (Gaussian) distribution**.

Syntax

```
C++
static Tensor randn(Shape shape, TensorOptions opts = {});
```

Description

This function fills the tensor with random numbers from a distribution with a mean of 0 and a variance of 1. This is the most common method for initializing weights in deep learning models.

Parameters

- shape **(Shape)**: The desired shape for the output tensor.
- opts **(TensorOptions, optional)**: A configuration object. Like rand, this is intended for floating-point dtypes.

Example

Creates a 2x2 tensor with normally distributed random values on the GPU.

```
C++
// Creates a tensor like: [[ 0.45, -1.23],
//                        [-0.89,  1.78]]
// (Values will be different each time)
Shape shape = {2, 2};
TensorOptions options;
options.device = Device::CUDA(0);
Tensor rn = Tensor::randn(shape, options);]
```

Tensor Utilities:

display():

The display function is a utility method of the Tensor class that provides a human-readable, formatted representation of the tensor's data, printed to a specified output stream.

Function Signature

```
C++
void display(std::ostream& stream, int precision = 4) const;
```

Description

This function iterates through the elements of a tensor and prints them to the console or any other output stream. The output is formatted with nested square brackets [] to visually represent the shape and dimensions of the tensor.

Parameters

Parameter	Type	Description
stream	**std::ostream &**	The output stream where the tensor's content will be written. This is typically std::cout.
precision	**int**	**(Optional)** The number of decimal places to display for floating-point data types (Float32, Float64, etc.). The default value is 4.

Behavior and Limitations

CPU-Only Operation: The display function can only be called on a tensor that resides on the CPU. It needs direct host access to the data to iterate and format it. If you have a tensor on a GPU, you must first move it to the CPU before displaying it.

```
C++
 // Correct usage for a GPU tensor:
Tensor gpu_tensor = my_tensor.to_cuda();
gpu_tensor.to_cpu().display(std::cout, 4);
```

- **Prints All Elements:** This function will print every single element in the tensor. For very large tensors, this can result in a large amount of output.

Example Usage

Here is how to use the display function to inspect the contents of a tensor.

```
C++
#include <iostream>
#include "TensorLib.h"

int main() {
    // Create a 2x3 tensor with floating-point data
    OwnTensor::Tensor my_tensor(OwnTensor::Shape{{2, 3}},
OwnTensor::TensorOptions{OwnTensor::Dtype::Float32});
    my_tensor.set_data({1.123456f, 2.2f, 3.3f, 4.4f, 5.5f, 6.6f});

    // --- Example 1: Basic Display ---
    // Uses the default precision of 4.
    std::cout << "--- Basic Display (default precision) ---" << std::endl;
    my_tensor.display(std::cout);

    // --- Example 2: Display with Custom Precision ---
    // Show only 2 decimal places.
    std::cout << "\n--- Display with Custom Precision (2) ---" <<
std::endl;
    my_tensor.display(std::cout, 2);

    return 0;
}
```

Expected Output

```
Code
   --- Basic Display (default precision) ---
[[ 1.1235,      2.2000,      3.3000],
 [4.4000,      5.5000,      6.6000]]

--- Display with Custom Precision (2) ---
[[ 1.12,  2.20,  3.30],
 [ 4.40,  5.50,  6.60]]
```

Tensor Device Transfer Functions

These methods are used to move a tensor's data between different compute devices, such as from the CPU to a GPU. All these operations create and return a **new Tensor object**; they do not modify the original tensor.

to(Device)

This is the primary, most flexible function for moving a tensor to a specific device.

Function Signature

```
C++
Tensor to(const Device& device) const;
```

Description

Creates a new tensor on the specified device, containing a copy of the original tensor's data.

Parameters

Parameter	Type	Description
device	const Device&	The target device where the new tensor will be created (e.g., Device::CPU or Device::CUDA).

Behavior

- **Cross-Device Copy:** If the target device is different from the original tensor's device, this function performs a full data transfer (e.g., from system RAM to GPU VRAM).
- **Shallow Copy:** If the tensor is already on the target device, this function does **not** perform a costly data copy. Instead, it efficiently returns a new Tensor object that shares the same underlying data.

---

to_cpu()

This is a convenience function for moving a tensor to the CPU.

Function Signature

```
C++
Tensor to_cpu() const;
```

Description

A shortcut for to(Device::CPU). It creates a new tensor on the CPU with a copy of the original tensor's data. This is commonly used to bring results back from the GPU to the host for printing or saving.

Example Usage

```
C++
Tensor cpu_copy = gpu_result.to_cpu();

std::cout << "Results from GPU, copied to CPU:" << std::endl;
cpu_copy.display(std::cout);
```

---

to_cuda()

This is a convenience function for moving a tensor to the GPU.

Function Signature

```
C++
Tensor to_cuda() const;
```

Description

A shortcut for to(Device::CUDA). It creates a new tensor on the default CUDA device with a copy of the original tensor's data. This is used to prepare data for GPU-accelerated computations.

Availability

This function is only available if the library was compiled with CUDA support (i.e., the WITH_CUDA macro is defined).

Example Usage

```
C++
Tensor cpu_tensor(OwnTensor::Shape{{10, 10}});
cpu_tensor.fill_(5.0f);

Tensor gpu_tensor = cpu_tensor.to_cuda();
```

Operations:

Unary Operations:

Trigonometric Operations :

The **Trigonometry** module implements both **standard** and **hyperbolic** trigonometric functions (and their inverses). Each function supports **CPU** and **CUDA** backends and multiple floating-point types.

Supported Functions

- `sin`, `cos`, `tan`
- `asin`, `acos`, `atan`
- `sinh`, `cosh`, `tanh`
- `asinh`, `acosh`, `atanh`

Data type handling:

Input Dtype	CPU Execution Dtype	CUDA Execution Dtype	Out-of-place (`sin(x)`)	In-place (`sin_(x)`)	Notes
`Float64`	Float64 (native)	Float64 (native)	✅ Supported	✅ Supported	Full precision (double)
`Float32`	Float32 (native)	Float32 (native)	✅ Supported	✅ Supported	Standard precision
`Float16`	Promoted to Float32	Float16 (native)	✅ Supported	✅ Supported	CPU computes in Float32, result cast back
`BFloat16`	Promoted to Float32	BFloat16 (native)	✅ Supported	✅ Supported	CPU computes in Float32, result cast back
`Int16`	Promoted to Float32	Promoted to Float32	✅ Supported	❌ Not supported	Returns Float32 tensor
`Int32`	Promoted to Float32	Promoted to Float32	✅ Supported	❌ Not supported	Returns Float32 tensor
`Int64`	Promoted to Float64	Promoted to Float64	✅ Supported	❌ Not supported	Returns Float64 tensor

Out-of-place trigonometric ops accept integer tensors and automatically promote them to Float32, returning a Float32 tensor. And for FP16 and bf16 convert to fp32 then perform the operation and convert back to fp16 and bf16.

Function Signatures:

Each trigonometric operation provides two variants:

```
// Out-of-place (returns a new tensor)
Tensor sin(const Tensor& x);

// In-place (modifies the input tensor)
void sin_(Tensor& x);
```

The same naming convention applies to all trigonometric operations.

```cpp
#include "core/Tensor.h"
#include "dtype/Types.h"
#include "ops/UnaryOps/Trigonometry.h"

using namespace OwnTensor;

int main() {
    Shape sh{{2, 3}};

    // Create directly on CUDA device
    Tensor a(sh, Dtype::Bfloat16, DeviceIndex(Device::CUDA), false);
    a.fill<bfloat16_t>(bfloat16_t(0.5f));

    // Out-of-place trigonometric operation on GPU
    Tensor s = sin(a);

    // In-place operation example
    cos_(a); // modifies tensor 'a' in place

    // Move to CPU for display
    Tensor b1 = s.to_cpu();
    Tensor b2 = a.to_cpu();

    std::cout << "Out-of-place sin(a):\n";
    b1.display(std::cout, 6);

    std::cout << "In-place cos_(a):\n";
    b2.display(std::cout, 6);

    return 0;
}
```

When dealing with cuda code use .to_cpu() in order to print the output.

```cpp
#include "core/Tensor.h"
#include "dtype/Types.h"
#include "ops/UnaryOps/Trigonometry.h"

using namespace OwnTensor;

int main() {
    Shape sh{{2, 3}};
    Tensor a(sh, Dtype::Float32, DeviceIndex(Device::CPU), false);
    a.fill<float>(0.5f);

    // Out-of-place example
    Tensor s = sin(a);

    // In-place example
    cos_(a);

    std::cout << "Out-of-place sin(a):\n";
    s.display(std::cout, 6);

    std::cout << "In-place cos_(a):\n";
    a.display(std::cout, 6);

    return 0;
}
```

For out place, create a new tensor and perform the operation. For in place operations you can perform directly.

NaN values:

When the input values go beyond these ranges for the following will lead to NaN values

Function	Valid domain	Example of NaN input	Output
`acos(x)`	-1 ≤ x ≤ 1	x = 2 or -2	**NaN**
`asin(x)`	-1 ≤ x ≤ 1	x = 1.1	**NaN**
`atan(x)`	All real numbers	none	finite
`acosh(x)`	x ≥ 1	x = 0.5	**NaN**
`atanh(x)`	-1 < x < 1	x = 2 or -2	**NaN**

Basic Arithmetic Module

What is the Basic Arithmetic Module?

The Basic Arithmetic Module provides a set of **element-wise unary mathematical functions**. All functions in this module are **device-agnostic**, automatically dispatching the appropriate kernel for CPU or CUDA devices,

Key Features

- **Element-Wise Execution:** Operations are applied independently to each element.
- **In-Place and Out-of-Place Variants:** Each operation has two forms:
    1. **Out-of-Place (e.g., neg):** Returns a *new* Tensor, leaving the original input unchanged.
    2. **In-Place (e.g., neg_):** Modifies the input Tensor directly, saving memory and avoiding data copy overhead.
- **Automatic Dtype Promotion:** Certain functions (like square, sqrt, and reciprocal) automatically promote the input data type to a higher precision (usually a floating-point type) to prevent overflow or ensure correct mathematical results.

2. Core Concepts: Out-of-Place vs. In-Place

A core principle of the OwnTensor framework is the distinction between functions that create a new tensor and those that modify the original.

Feature	Out-of-Place (e.g., square(t))	In-Place (e.g., square_(t))
Syntax	Tensor result = func(input)	func_(input)
Memory	Allocates a **new** Tensor object and its backing data.	**No** new memory allocated; modifies the existing data.
Return Value	A new Tensor object.	void (no return value).
Naming Convention	Standard function name (e.g., abs, pow).	Function name suffixed with an underscore (e.g., abs_, pow_).

3. API Reference

The module provides functions for negation, absolute value, sign, square, square root, reciprocal, and power.

3.1. Square (square / square_)

Computes the element-wise square of the input tensor: output[i] = input[i]^2.

Out-of-Place Syntax	In-Place Syntax	Parameters	Return
Tensor square(const Tensor& input)	void square_(Tensor& input)	input (Tensor): The input tensor.	Tensor (out-of-place) or void (in-place).

Details on Dtype Promotion:

- To prevent overflow, integer inputs (Int16, Int32, Int64) are **automatically promoted to Float64** in the output tensor.
- Floating-point inputs (Float16, Bfloat16, Float32, Float64) retain their original floating-point type.

Example:

```
Tensor t_cpu({{2, 3}}, Dtype::Int32, DeviceIndex(Device::CPU));
  std::vector<int> data1 = {100, -200, 3000 , 400, 0, 600};
  t_cpu.set_data(data1);
  std::cout << "\n SQUARE ON CPU" << std::endl;
  Tensor t1 = square(t_cpu);
  t1.display(std::cout,6);
```

```
Tensor t_cpu({{2, 3}}, Dtype::Int32, DeviceIndex(Device::CPU));
  std::vector<int> data1 = {100, -200, 3000 , 400, 0, 600};
  t_cpu.set_data(data1);
  std::cout << "\n IN PLACE SQUARE ON CPU" << std::endl;
  square_(t_cpu);
  t_cpu.display(std::cout,6);
```

3.2. Square Root (sqrt / sqrt_)

Computes the element-wise square root of the input tensor:

Out-of-Place Syntax	In-Place Syntax	Parameters	Return
Tensor sqrt(const Tensor& input)	void sqrt_(Tensor& input)	input (Tensor): The input tensor.	Tensor (out-of-place) or void (in-place).

Details on Dtype Promotion:

- Since square root is primarily defined for non-negative numbers and often yields non-integer results, the output is **always floating-point**.
- Integer inputs are promoted to Float32 in the output tensor.
- Inputs must be non-negative; taking the square root of a negative value will result in NaN (Not a Number) in the output.

Example:

```
Tensor t_cpu({{2, 3}}, Dtype::Int64, DeviceIndex(Device::CPU));
   std::vector<int64_t> data1 = {100, -200, 3000 , 400, 0, 600};
   t_cpu.set_data(data1);
   std::cout << "\n SQUARE ROOT ON CPU" << std::endl;
   Tensor t1 = sqrt(t_cpu);
   t1.display(std::cout,6);
```

```
Tensor t_cpu({{2, 3}}, Dtype::Float32, DeviceIndex(Device::CPU));
   std::vector<float> data1 = {100.0f, -200.0f, 3000.0f , 400.0f, 0.0f,
600.0f};
```

```
    t_cpu.set_data(data1);
    std::cout << "\n IN PLACE SQUARE ROOT ON CPU" << std::endl;
    sqrt_(t_cpu);
    t_cpu.display(std::cout,6);
```

3.3. Negation (neg / neg_)

Computes the element-wise negation of the input tensor: output[i] = -input[i].

Out-of-Place Syntax	In-Place Syntax	Parameters	Return
Tensor neg(const Tensor& input)	void neg_(Tensor& input)	input (Tensor): The input tensor.	Tensor (out-of-place) or void (in-place).

Example:

```
Tensor t_cpu({{2, 3}}, Dtype::Float32, DeviceIndex(Device::CPU));
    std::vector<float> data1 = {100.0f, -200.0f, 3000.0f , 400.0f, 0.0f,
600.0f};
    t_cpu.set_data(data1);
    std::cout << "\n IN PLACE SQUARE ROOT ON CPU" << std::endl;
    sqrt_(t_cpu);
    t_cpu.display(std::cout,6);
```

```
Tensor t_cpu({{2, 3}}, Dtype::Int64, DeviceIndex(Device::CPU));
    std::vector<int64_t> data1 = {100, -200, 3000 , 400, 0, 600};
    t_cpu.set_data(data1);
    std::cout << "\nIN PLACE NEGATION ON CPU" << std::endl;
    neg_(t_cpu);
    t_cpu.display(std::cout,6);
```

3.4. Absolute Value (abs / abs_)

Computes the element-wise absolute value of the input tensor: output[i] = |input[i]|.

Out-of-Place Syntax	In-Place Syntax	Parameters	Return
Tensor abs(const Tensor& input)	void abs_(Tensor& input)	input (Tensor): The input tensor.	Tensor (out-of-place) or void (in-place).

Example:

```
Tensor t_cpu({{2, 3}}, Dtype::Int64, DeviceIndex(Device::CPU));
    std::vector<int64_t> data1 = {100, -200, 3000 , 400, 0, 600};
    t_cpu.set_data(data1);
    std::cout << "\n ABSOLUTE ON CPU" << std::endl;
```

```
    Tensor t1 = abs(t_cpu);
    t1.display(std::cout,6);
```

```
Tensor t_cpu({{2, 3}}, Dtype::Int64, DeviceIndex(Device::CPU));
    std::vector<int64_t> data1 = {100, -200, 3000 , 400, 0, 600};
    t_cpu.set_data(data1);
    std::cout << "\n IN PLACE ABSOLUTE ON CPU" << std::endl;
    abs_(t_cpu);
    t_cpu.display(std::cout,6);
```

3.5. Sign Function (sign / sign_)

Computes the element-wise sign of the input tensor. Returns **1** for positive numbers, **-1** for negative numbers, and **0** for zero.

Out-of-Place Syntax	In-Place Syntax	Parameters	Return
Tensor sign(const Tensor& input)	void sign_(Tensor& input)	input (Tensor): The input tensor.	Tensor (out-of-place) or void (in-place).

Example:

```
Tensor t_cpu({{2, 3}}, Dtype::Int64, DeviceIndex(Device::CPU));
    std::vector<int64_t> data1 = {100, -200, 3000 , 400, 0, 600};
    t_cpu.set_data(data1);
    std::cout << "\n SIGN ON CPU" << std::endl;
    Tensor t1 =sign(t_cpu);
    t1.display(std::cout,6);
```

```
Tensor t_cpu({{2, 3}}, Dtype::Int64, DeviceIndex(Device::CPU));
    std::vector<int64_t> data1 = {100, -200, 3000 , 400, 0, 600};
    t_cpu.set_data(data1);
    std::cout << "\n SIGN ON CPU" << std::endl;
    Tensor t1 =sign(t_cpu);
    t1.display(std::cout,6);
```

3.6. Reciprocal (reciprocal / reciprocal_)

Computes the element-wise reciprocal of the input tensor: output[i] = 1 / input[i].

Out-of-Place Syntax	In-Place Syntax	Parameters	Return
Tensor reciprocal(const Tensor& input)	void reciprocal_(Tensor& input)	input (Tensor): The input tensor.	Tensor (out-of-place) or void (in-place).

Details on Dtype Promotion:

- Similar to sqrt, the output is **always floating-point** to handle division results.
- Integer inputs are promoted to Float32 in the output tensor.

- Division by zero will result in **infinity** in the output tensor.

Example:

```
Tensor t_cpu({{2, 3}}, Dtype::Float32, DeviceIndex(Device::CPU));
   std::vector<float> data1 = {100.0f, -200.0f, 3000.0f , 400.0f, 0.0f, 600.0f};
   t_cpu.set_data(data1);
   std::cout << "\n OUT PLACE RECIPROCAL ON CPU" << std::endl;
   Tensor t1 = reciprocal(t_cpu);
   t1.display(std::cout,6);

Tensor t_cpu({{2, 3}}, Dtype::Float32, DeviceIndex(Device::CPU));
   std::vector<float> data1 = {100.0f, -200.0f, 3000.0f , 400.0f, 0.0f, 600.0f};
   t_cpu.set_data(data1);
   std::cout << "\n IN PLACE RECIPROCAL ON CPU" << std::endl;
   reciprocal_(t_cpu);
   t_cpu.display(std::cout,6);
```

3.7. Power (pow / pow_)

Computes the element-wise power of the input tensor: output[i] = input[i]^exponent. This function supports multiple overloads for the exponent type.

Out-of-Place Power (pow)

Syntax	Parameters	Return
Tensor pow(const Tensor& t, int exponent)	t (Tensor), exponent (int)	Tensor
Tensor pow(const Tensor& t, float exponent)	t (Tensor), exponent (float)	Tensor
Tensor pow(const Tensor& t, double exponent)	t (Tensor), exponent (double)	Tensor

In-Place Power (pow_)

Syntax	Parameters	Return
void pow_(Tensor& t, int exponent)	t (Tensor), exponent (int)	void
void pow_(Tensor& t, float exponent)	t (Tensor), exponent (float)	void
void pow_(Tensor& t, double exponent)	t (Tensor), exponent (double)	void

Details on Dtype Promotion:

- If the exponent is an **integer**, the output Dtype matches the input Dtype unless an overflow occurs, in which case it follows the square promotion rule (Int $\rightarrow$ Float64).
- If the exponent is a **floating-point type** (float or double), the output is **always a floating-point tensor** (Float32 or Float64) following similar rules to sqrt and reciprocal.

Example:

```
Tensor t_cpu({{2, 3}}, Dtype::Float32, DeviceIndex(Device::CPU));
   std::vector<float> data1 = {100.0f, -200.0f, 3000.0f , 400.0f, 0.0f,
600.0f};
   t_cpu.set_data(data1);
   std::cout << "\n OUT PLACE POWER ON CPU" << std::endl;
   Tensor t1 = pow(t_cpu,3);
   t1.display(std::cout,6);

Tensor t_cpu({{2, 3}}, Dtype::Float32, DeviceIndex(Device::CPU));
   std::vector<float> data1 = {100.0f, -200.0f, 3000.0f , 400.0f, 0.0f,
600.0f};
   t_cpu.set_data(data1);
   std::cout << "\n IN PLACE POWER ON CPU" << std::endl;
   pow_(t_cpu,3);
   t_cpu.display(std::cout,6);
```

Error causing parameters in pow_() function:

```
pow_(t_cpu,-2);
pow_(t_cpu,0.5);
```

Reduction Module

What is Reduction?

In tensor operations, **reduction** is the process of "collapsing" one or more dimensions of a tensor by applying an operation (like sum, mean, or max) across those dimensions.
Think of it as summarizing data.

Key Parameters

All reduction functions in this module share a common set of parameters:

- const Tensor& input: The tensor you want to reduce.
- const std::vector<int64_t>& axes = {}: A vector of dimension indices to reduce.
  - If you provide {} (the default), the operation performs a **full reduction** across all axes, resulting in a scalar tensor.
  - If you provide {0}, it reduces along the first dimension.
  - If you provide {0, 2}, it reduces along the first and third dimensions.
- bool keepdim = false:
  - false (default): The reduced dimensions are removed. A (3, 4) tensor reduced on axis=0 becomes (4,).
  - true: The reduced dimensions are kept, but with a size of 1. A (3, 4) tensor reduced on axis=0 becomes (1, 4). This is useful for broadcasting operations.

2. Setup for C++ Examples

All examples below use the following setup code. We define two base tensors:

1. t_simple_cpu: A standard tensor for most operations.
2. t_complex_cpu: A tensor containing NAN to demonstrate the "Nan-aware" operations.

```cpp
#include "TensorLib.h" // Your framework's main header
#include <iostream>
#include <vector>
#include <cmath> // For NAN
#include <iomanip> // For std::setprecision

using namespace OwnTensor;

// --- Base Tensors Used in Examples ---

// 1. Simple Tensor (for non-NaN ops)
std::vector<float> data_simple = {1.0f, 5.0f, 3.0f, 4.0f, 2.0f, 6.0f};
Tensor t_simple_cpu({{2, 3}}, Dtype::Float32, DeviceIndex(Device::CPU));
t_simple_cpu.set_data(data_simple);
// t_simple_cpu:
// [[ 1.00,  5.00,  3.00],
//  [ 4.00,  2.00,  6.00]]

// 2. Complex Tensor (with NaN) (for NaN-aware ops)
std::vector<float> data_complex = {1.0f, 5.0f, 3.0f, 4.0f, NAN, 6.0f};
Tensor t_complex_cpu({{2, 3}}, Dtype::Float32, DeviceIndex(Device::CPU));
t_complex_cpu.set_data(data_complex);
// t_complex_cpu:
// [[ 1.00,  5.00,  3.00],
//  [ 4.00,   nan,  6.00]]

// 3. GPU versions
#ifdef WITH_CUDA
Tensor t_simple_gpu = t_simple_cpu.to(DeviceIndex(Device::CUDA));
Tensor t_complex_gpu = t_complex_cpu.to(DeviceIndex(Device::CUDA));
#endif
```

3. Core Reduction Operations

These operations work on all data types. **If the input contains NaN, the result of the reduction will also be NaN**.

1. reduce_sum

Computes the sum of elements across specified dimensions.

Syntax

```
// Full reduction (on all axes)
Tensor result = reduce_sum(your_tensor);

// Partial reduction (on specific axes)
Tensor result = reduce_sum(your_tensor, {axis_0, axis_1});

// Partial reduction (keeping dimensions)
Tensor result = reduce_sum(your_tensor, {axis_0}, true);
```

Usage Examples (using t_simple_cpu)

```
// Base Tensor:
// [[ 1.00,  5.00,  3.00],
//  [ 4.00,  2.00,  6.00]]

// 1. Full reduction (1+5+3+4+2+6)
Tensor res_full = reduce_sum(t_simple_cpu);
// Output: [21.00]

// 2. Partial reduction (axis 0)
// [1+4, 5+2, 3+6]
Tensor res_ax0 = reduce_sum(t_simple_cpu, {0});
// Output: [[ 5.00,  7.00,  9.00]]

// 3. Partial reduction (axis 1, keepdim=true)
// [1+5+3], [4+2+6]
Tensor res_ax1k = reduce_sum(t_simple_cpu, {1}, true);
// Output: [[ 9.00],
//          [12.00]]

// 4. GPU reduction
#ifdef WITH_CUDA
Tensor res_gpu = reduce_sum(t_simple_gpu, {0}); // ax=0 on GPU
// Output: [[ 5.00,  7.00,  9.00]]
#endif
```

2. reduce_product

Computes the product of elements across specified dimensions.

Syntax

```
// Full reduction (on all axes)
```

```
Tensor result = reduce_product(your_tensor);

// Partial reduction (on specific axes)
Tensor result = reduce_product(your_tensor, {axis_0, axis_1});

// Partial reduction (keeping dimensions)
Tensor result = reduce_product(your_tensor, {axis_0}, true);
```

Usage Examples (using t_simple_cpu)

```
// 1. Full reduction (1*5*3*4*2*6)
Tensor res_full = reduce_product(t_simple_cpu);
// Output: [720.00]

// 2. Partial reduction (axis 0)
// [1*4, 5*2, 3*6]
Tensor res_ax0 = reduce_product(t_simple_cpu, {0});
// Output: [[ 4.00, 10.00, 18.00]]
```

3. reduce_min

Computes the minimum value of elements across specified dimensions.

Syntax

```
// Full reduction (on all axes)
Tensor result = reduce_min(your_tensor);

// Partial reduction (on specific axes)
Tensor result = reduce_min(your_tensor, {axis_0, axis_1});

// Partial reduction (keeping dimensions)
Tensor result = reduce_min(your_tensor, {axis_0}, true);
```

Usage Examples (using t_simple_cpu)

```
// 1. Full reduction
Tensor res_full = reduce_min(t_simple_cpu);
// Output: [1.00]

// 2. Partial reduction (axis 1)
// min(1,5,3), min(4,2,6)
Tensor res_ax1 = reduce_min(t_simple_cpu, {1});
// Output: [[ 1.00],
//          [ 2.00]]
```

4. reduce_max

Computes the maximum value of elements across specified dimensions.

Syntax

```
// Full reduction (on all axes)
Tensor result = reduce_max(your_tensor);

// Partial reduction (on specific axes)
Tensor result = reduce_max(your_tensor, {axis_0, axis_1});

// Partial reduction (keeping dimensions)
Tensor result = reduce_max(your_tensor, {axis_0}, true);
```

Usage Examples (using t_simple_cpu)

```
// 1. Full reduction
Tensor res_full = reduce_max(t_simple_cpu);
// Output: [6.00]

// 2. Partial reduction (axis 0)
// max(1,4), max(5,2), max(3,6)
Tensor res_ax0 = reduce_max(t_simple_cpu, {0});
// Output: [[ 4.00,  5.00,  6.00]]
```

5. reduce_mean

Computes the mean of elements across specified dimensions.

Syntax

```
// Full reduction (on all axes)
Tensor result = reduce_mean(your_tensor);

// Partial reduction (on specific axes)
Tensor result = reduce_mean(your_tensor, {axis_0, axis_1});

// Partial reduction (keeping dimensions)
Tensor result = reduce_mean(your_tensor, {axis_0}, true);
```

Usage Examples (using t_simple_cpu)

```
// 1. Full reduction (21 / 6)
Tensor res_full = reduce_mean(t_simple_cpu);
// Output: [3.50]

// 2. Partial reduction (axis 1)
// (1+5+3)/3, (4+2+6)/3
Tensor res_ax1 = reduce_mean(t_simple_cpu, {1});
// Output: [[ 3.00],
//          [ 4.00]]
```

4. NaN-Aware Reduction Operations

These operations are **only for floating-point types**. They behave identically to the core operations but **treat NaN values as if they do not exist**.

6. reduce_nansum

Computes the sum, skipping NaN values.

Syntax

```
// Full reduction (on all axes)
Tensor result = reduce_nansum(your_tensor);

// Partial reduction (on specific axes)
Tensor result = reduce_nansum(your_tensor, {axis_0, axis_1});

// Partial reduction (keeping dimensions)
Tensor result = reduce_nansum(your_tensor, {axis_0}, true);
```

Usage Examples (using t_complex_cpu)

```
// Base Tensor:
// [[ 1.00,  5.00,  3.00],
//  [ 4.00,   nan,  6.00]]

// 1. Full reduction (1+5+3+4+6, skip NaN)
Tensor res_full = reduce_nansum(t_complex_cpu);
// Output: [19.00]

// 2. Partial reduction (axis 0)
// [1+4, 5+0, 3+6]
Tensor res_ax0 = reduce_nansum(t_complex_cpu, {0});
// Output: [[ 5.00,  5.00,  9.00]]
```

```
// 3. Partial reduction (axis 1)
// [1+5+3], [4+6]
Tensor res_ax1 = reduce_nansum(t_complex_cpu, {1});
// Output: [[ 9.00],
//          [10.00]]
```

7. reduce_nanproduct

Computes the product, skipping NaN values.

Syntax

```
// Full reduction (on all axes)
Tensor result = reduce_nanproduct(your_tensor);

// Partial reduction (on specific axes)
Tensor result = reduce_nanproduct(your_tensor, {axis_0, axis_1});

// Partial reduction (keeping dimensions)
Tensor result = reduce_nanproduct(your_tensor, {axis_0}, true);
```

Usage Examples (using t_complex_cpu)

```
// 1. Full reduction (1*5*3*4*6)
Tensor res_full = reduce_nanproduct(t_complex_cpu);
// Output: [360.00]

// 2. Partial reduction (axis 1)
// [1*5*3], [4*6]
Tensor res_ax1 = reduce_nanproduct(t_complex_cpu, {1});
// Output: [[ 15.00],
//          [ 24.00]]
```

8. reduce_nanmin

Computes the minimum, skipping NaN values.

Syntax

```
// Full reduction (on all axes)
Tensor result = reduce_nanmin(your_tensor);

// Partial reduction (on specific axes)
Tensor result = reduce_nanmin(your_tensor, {axis_0, axis_1});

// Partial reduction (keeping dimensions)
```

```
Tensor result = reduce_nanmin(your_tensor, {axis_0}, true);
```

Usage Examples (using t_complex_cpu)

```
// 1. Full reduction
Tensor res_full = reduce_nanmin(t_complex_cpu);
// Output: [1.00]

// 2. Partial reduction (axis 0)
// min(1,4), min(5,NaN), min(3,6)
Tensor res_ax0 = reduce_nanmin(t_complex_cpu, {0});
// Output: [[ 1.00,  5.00,  3.00]]
```

9. reduce_nanmax

Computes the maximum, skipping NaN values.

Syntax

```
// Full reduction (on all axes)
Tensor result = reduce_nanmax(your_tensor);

// Partial reduction (on specific axes)
Tensor result = reduce_nanmax(your_tensor, {axis_0, axis_1});

// Partial reduction (keeping dimensions)
Tensor result = reduce_nanmax(your_tensor, {axis_0}, true);
```

Usage Examples (using t_complex_cpu)

```
// 1. Full reduction
Tensor res_full = reduce_nanmax(t_complex_cpu);
// Output: [6.00]

// 2. Partial reduction (axis 0)
// max(1,4), max(5,NaN), max(3,6)
Tensor res_ax0 = reduce_nanmax(t_complex_cpu, {0});
// Output: [[ 4.00,  5.00,  6.00]]
```

10. reduce_nanmean

Computes the mean, skipping NaN values (and not counting them in the denominator).

Syntax

```
// Full reduction (on all axes)
Tensor result = reduce_nanmean(your_tensor);

// Partial reduction (on specific axes)
Tensor result = reduce_nanmean(your_tensor, {axis_0, axis_1});

// Partial reduction (keeping dimensions)
Tensor result = reduce_nanmean(your_tensor, {axis_0}, true);
```

Usage Examples (using t_complex_cpu)

```
// Base Tensor:
// [[ 1.00,  5.00,  3.00],
//  [ 4.00,   nan,  6.00]]

// 1. Full reduction (19 / 5 elements)
Tensor res_full = reduce_nanmean(t_complex_cpu);
// Output: [3.80]

// 2. Partial reduction (axis 1)
// (1+5+3)/3, (4+6)/2
Tensor res_ax1 = reduce_nanmean(t_complex_cpu, {1});
// Output: [[ 3.00],
//          [ 5.00]]
```

5. Index Reduction Operations

These operations return the *index* (not the value) of the min/max element. The **output tensor is always Dtype::Int64**.

11. reduce_argmin

Computes the index of the minimum element.

Syntax

```
// Full reduction (on all axes)
Tensor result = reduce_argmin(your_tensor);

// Partial reduction (on specific axes)
Tensor result = reduce_argmin(your_tensor, {axis_0, axis_1});

// Partial reduction (keeping dimensions)
Tensor result = reduce_argmin(your_tensor, {axis_0}, true);
```

Usage Examples (using t_simple_cpu)

```
// Base Tensor:
// [[ 1.00,  5.00,  3.00],
//  [ 4.00,  2.00,  6.00]]

// 1. Full reduction (flattened tensor is [1,5,3,4,2,6]. Min is 1 at index
0)
Tensor res_full = reduce_argmin(t_simple_cpu);
// Output: [0] (Int64)

// 2. Partial reduction (axis 0)
// col 0: min(1,4) is 1 at index 0
// col 1: min(5,2) is 2 at index 1
// col 2: min(3,6) is 3 at index 0
Tensor res_ax0 = reduce_argmin(t_simple_cpu, {0});
// Output: [[ 0,  1,  0]] (Int64)

// 3. Partial reduction (axis 1, keepdim=true)
// row 0: min(1,5,3) is 1 at index 0
// row 1: min(4,2,6) is 2 at index 1
Tensor res_ax1k = reduce_argmin(t_simple_cpu, {1}, true);
// Output: [[ 0],
//          [ 1]] (Int64)
```

12. reduce_argmax

Computes the index of the maximum element.

Syntax

```
// Full reduction (on all axes)
Tensor result = reduce_argmax(your_tensor);

// Partial reduction (on specific axes)
Tensor result = reduce_argmax(your_tensor, {axis_0, axis_1});

// Partial reduction (keeping dimensions)
Tensor result = reduce_argmax(your_tensor, {axis_0}, true);
```

Usage Examples (using t_simple_cpu)

```
// Base Tensor:
// [[ 1.00,  5.00,  3.00],
//  [ 4.00,  2.00,  6.00]]

// 1. Full reduction (flattened tensor is [1,5,3,4,2,6]. Max is 6 at index
```

```
5)
Tensor res_full = reduce_argmax(t_simple_cpu);
// Output: [5] (Int64)

// 2. Partial reduction (axis 0)
// col 0: max(1,4) is 4 at index 1
// col 1: max(5,2) is 5 at index 0
// col 2: max(3,6) is 6 at index 1
Tensor res_ax0 = reduce_argmax(t_simple_cpu, {0});
// Output: [[ 1,  0,  1]] (Int64)
```

6. NaN-Aware Index Reduction Operations

Identical to Index Reductions, but **they skip NaN values** when finding the min/max index. **Output is Dtype::Int64**.

13. reduce_nanargmin

Computes the index of the minimum element, skipping NaN.
Syntax

```
// Full reduction (on all axes)
Tensor result = reduce_nanargmin(your_tensor);

// Partial reduction (on specific axes)
Tensor result = reduce_nanargmin(your_tensor, {axis_0, axis_1});

// Partial reduction (keeping dimensions)
Tensor result = reduce_nanargmin(your_tensor, {axis_0}, true);
```

Usage Examples (using t_complex_cpu)

```
// Base Tensor:
// [[ 1.00,  5.00,  3.00],
//  [ 4.00,   nan,  6.00]]

// 1. Full reduction (min is 1 at index 0)
Tensor res_full = reduce_nanargmin(t_complex_cpu);
// Output: [0] (Int64)

// 2. Partial reduction (axis 0)
// col 0: min(1,4) is 1 at index 0
// col 1: min(5,NaN) is 5 at index 0
// col 2: min(3,6) is 3 at index 0
Tensor res_ax0 = reduce_nanargmin(t_complex_cpu, {0});
// Output: [[ 0,  0,  0]] (Int64)
```

14. reduce_nanargmax

Computes the index of the maximum element, skipping NaN.

Syntax

```
// Full reduction (on all axes)
Tensor result = reduce_nanargmax(your_tensor);

// Partial reduction (on specific axes)
Tensor result = reduce_nanargmax(your_tensor, {axis_0, axis_1});

// Partial reduction (keeping dimensions)
Tensor result = reduce_nanargmax(your_tensor, {axis_0}, true);
```

Usage Examples (using t_complex_cpu)

```
// Base Tensor:
// [[ 1.00,  5.00,  3.00],
//  [ 4.00,   nan,  6.00]]

// 1. Full reduction (max is 6 at index 5)
Tensor res_full = reduce_nanargmax(t_complex_cpu);
// Output: [5] (Int64)

// 2. Partial reduction (axis 1)
// row 0: max(1,5,3) is 5 at index 1
// row 1: max(4,NaN,6) is 6 at index 2
Tensor res_ax1 = reduce_nanargmax(t_complex_cpu, {1});
// Output: [[ 1],
//          [ 2]] (Int64)
```

Exponent and Logarithm Module

What is the Exponent and Logarithm Module?

The Exponent and Logarithm Module provides a set of **element-wise unary mathematical functions** essential for deep learning computations, such as activation functions, probability distributions, and numerical stability techniques.
These functions include:

1. **Exponentials:** e^x, 2^x (base-e and base-2 exponents).
2. **Logarithms:** ln(x) (natural log), log_2(x), log_10(x) (logarithms to bases e, 2, and 10).

All functions are **device-agnostic** (CPU and CUDA) and support both out-of-place and in-place execution.

Key Computational Considerations

- **Floating-Point Output (Mandatory Dtype Promotion):** Since exponential and logarithmic operations rarely result in exact integers, all functions in this module enforce **Dtype Promotion**. The output tensor is always a floating-point type, even if the input tensor is an integer type. This ensures mathematical accuracy and handles potential non-integer results.
- **Domain Restrictions:** Logarithmic functions have strict input domain requirements (input(x) must be x > 0). Providing non-positive input will result in a numerical error (-inf or NaN).

2. Core Concepts: Out-of-Place vs. In-Place

These functions strictly adhere to the OwnTensor framework's convention:

- **Out-of-Place (e.g., exp):** Returns a *new* Tensor.
- **In-Place (e.g., exp_):** Modifies the input Tensor directly, often changing its Dtype to a floating-point type.

Feature	Out-of-Place (e.g., exp(t))	In-Place (e.g., exp_(t))
Dtype Change	The output tensor has the promoted floating-point Dtype.	The input tensor's Dtype is changed in-place to the promoted floating-point type.
Return Value	A new Tensor object.	void (no return value).

3. API Reference: Exponential Functions

3.1. Exponential (exp / exp_)

Computes the element-wise exponential of the input tensor using base e:
output[i] = e^{input[i]}.

Out-of-Place Syntax	In-Place Syntax	Parameters	Return
Tensor exp(const Tensor& input)	void exp_(Tensor& input)	input (Tensor): The input tensor.	Tensor (out-of-place) or void (in-place).

Details on Dtype Promotion:

- **Output Dtype is Floating-Point:** All integer inputs are promoted to Float32 in the output/in-place tensor. Existing floating-point types (Float32, Float64) are preserved.

Example:

```
Tensor t_cpu({{2, 3}}, Dtype::Float32, DeviceIndex(Device::CPU));
   std::vector<float> data1 = {100.0f, -200.0f, 3000.0f , 400.0f, 0.0f, 600.0f};
   t_cpu.set_data(data1);
   std::cout << "\n EXP ON CPU" << std::endl;
   Tensor t1 = exp(t_cpu);
   t1.display(std::cout,6);

//INPLACE OPERATION TEST
   std::cout << "\n IN PLACE EXP ON CPU" << std::endl;
   exp_(t_cpu); //inplace pow on cpu tensor
   t_cpu.display(std::cout,6);
```

3.2. Exponential Base 2 (exp2 / exp2_)

Computes the element-wise exponential of the input tensor using base 2:
output[i] = 2^{input_i}$.

Out-of-Place Syntax	In-Place Syntax	Parameters	Return
Tensor exp2(const Tensor& input)	void exp2_(Tensor& input)	input (Tensor): The input tensor.	Tensor (out-of-place) or void (in-place).

Details on Dtype Promotion:

- Follows the same Dtype Promotion rules as exp.

Example:

```
  Tensor t_cpu({{2, 3}}, Dtype::Float32, DeviceIndex(Device::CPU));
    std::vector<float> data1 = {100.0f, -200.0f, 3000.0f , 400.0f, 0.0f, 600.0f};
    t_cpu.set_data(data1);
    std::cout << "\n EXP_2 ON CPU" << std::endl;
    Tensor t1 = exp2(t_cpu);
    t1.display(std::cout,6);

 //INPLACE OPERATION TEST
    std::cout << "\n IN PLACE EXP_2 ON CPU" << std::endl;
    exp2_(t_cpu); //inplace pow on cpu tensor
    t_cpu.display(std::cout,6);
```

4. API Reference: Logarithmic Functions

4.1. Natural Logarithm (log / log_)

Computes the element-wise natural logarithm (log base e) of the input tensor:
output[i] = ln(input[i]).

Out-of-Place Syntax	In-Place Syntax	Parameters	Return
Tensor log(const Tensor& input)	void log_(Tensor& input)	input (Tensor): The input tensor.	Tensor (out-of-place) or void (in-place).

Details on Dtype Promotion and Domain:

- **Output Dtype is Floating-Point:** All integer inputs are promoted to Float32.
- **Domain Restriction:** The input elements **must be strictly positive** ($x > 0$).
    - Input of 0 results in **negative infinity** (-inf).
    - Input of a negative number results in **Not a Number** (NaN).

Example:

```
 Tensor t_cpu({{2, 3}}, Dtype::Float32, DeviceIndex(Device::CPU));
    std::vector<float> data1 = {100.0f, -200.0f, 3000.0f , 400.0f, 0.0f, 600.0f};
    t_cpu.set_data(data1);
    std::cout << "\n LOG ON CPU" << std::endl;
    Tensor t1 =log(t_cpu);
    t1.display(std::cout,6);

 //INPLACE OPERATION TEST
```

```
    std::cout << "\n IN PLACE LOG ON CPU" << std::endl;
    log_(t_cpu); //inplace pow on cpu tensor
    t_cpu.display(std::cout,6);
```

4.2. Logarithm Base 2 (log2 / log2_)

Computes the element-wise logarithm base 2 of the input tensor: output[i] = log_2(input[i]).

Out-of-Place Syntax	In-Place Syntax	Parameters	Return
Tensor log2(const Tensor& input)	void log2_(Tensor& input)	input (Tensor): The input tensor.	Tensor (out-of-place) or void (in-place).

Details on Dtype Promotion and Domain:

- Follows the same Dtype Promotion and Domain Restriction rules as log.

Example:

```
  Tensor t_cpu({{2, 3}}, Dtype::Float32, DeviceIndex(Device::CPU));
    std::vector<float> data1 = {100.0f, -200.0f, 3000.0f , 400.0f, 0.0f, 600.0f};
    t_cpu.set_data(data1);
    std::cout << "\n LOG_2 ON CPU" << std::endl;
    Tensor t1 =log2(t_cpu);
    t1.display(std::cout,6);

 //INPLACE OPERATION TEST
    std::cout << "\n IN PLACE LOG_2 ON CPU" << std::endl;
    log2_(t_cpu); //inplace pow on cpu tensor
    t_cpu.display(std::cout,6);
```

4.3. Logarithm Base 10 (log10 / log10_)

Computes the element-wise logarithm base 10 of the input tensor:
output[i] = log_{10}(input[i]).

Out-of-Place Syntax	In-Place Syntax	Parameters	Return
Tensor log10(const Tensor& input)	void log10_(Tensor& input)	input (Tensor): The input tensor.	Tensor (out-of-place) or void (in-place).

Details on Dtype Promotion and Domain:

- Follows the same Dtype Promotion and Domain Restriction rules as log.

Example:

```
  Tensor t_cpu({{2, 3}}, Dtype::Float32, DeviceIndex(Device::CPU));
    std::vector<float> data1 = {100.0f, -200.0f, 3000.0f , 400.0f, 0.0f, 600.0f};
    t_cpu.set_data(data1);
    std::cout << "\n LOG_10 ON CPU" << std::endl;
    Tensor t1 = log10(t_cpu);
    t1.display(std::cout,6);

 //INPLACE OPERATION TEST
    std::cout << "\n IN PLACE LOG_10 ON CPU" << std::endl;
    log10_(t_cpu); //inplace pow on cpu tensor
    t_cpu.display(std::cout,6);
```

Scalar Operations:

The **Scalar Operations** module enables arithmetic between a **Tensor** and a **Scalar** value. All standard arithmetic operators are supported both **out-of-place** and **in-place**.

Supported Operations:

Operation	Out-of-Place	In-Place
Addition	`tensor + scalar`	`tensor += scalar`
Subtraction	`tensor - scalar`	`tensor -= scalar`
Multiplication	`tensor * scalar`	`tensor *= scalar`
Division	`tensor / scalar`	`tensor /= scalar`

Out of place also supports:

```
scalar + tensor
scalar - tensor
scalar * tensor
scalar / tensor
```

-

```
Shape sh{{2, 3}};
Tensor a(sh, Dtype::Float32, DeviceIndex(Device::CPU));
a.fill<float>(2.0f);

Tensor result = a * 3.5f;  // Out-of-place
a += 1.5f;                 // In-place
```

Division by zero:

- If the tensor data type is float, it returns inf values
- If the tensor data type is integer, it throws an error.

Datatypes:

- Use the same datatype for scalar values as the Tensor for better results.

```
Float32 - 2.0f
Float64 - 2.0;
Int32 - 2;
Int64 - 2L
Bfloat16 - (bfloat16_t)2.0f
Float16 - (float16_t) 2.0f.
```

Tensor Operation:

The **Tensor Operations** module enables arithmetic between two Tensors.
All standard arithmetic operators are supported both **out-of-place** and **in-place**.

```cpp
#include <TensorLib.h>
#include <iostream>

using namespace OwnTensor;

int main() {
    // Create two 2x2 Float32 tensors on CPU
    Tensor a(Shape{{2, 2}}, Dtype::Float32, Device::CPU,false);
    a.fill(8.0f);

    Tensor b(Shape{{2, 2}}, Dtype::Float32, Device::CPU, false);
    b.fill(2.0f);

    // Elementwise operations (same shapes & SAME DTYPE required)
    Tensor p = a * b;   // multiply -> [[16, 16], [16, 16]]
    Tensor q = a + b;   // add      -> [[10, 10], [10, 10]]

    // Pretty-print (precision = 8 decimals)
    p.display(std::cout, 8);
}
```

Conditions:

- Make sure the datatypes of both tensors are the same.
- For more than 2D Tensors The shape should be same
- For 2D Tensor we can do operations with a 1D Tensor (vector) also.

```cpp
Tensor operator+(const Tensor& lhs, const Tensor& rhs);
Tensor operator-(const Tensor& lhs, const Tensor& rhs);
Tensor operator*(const Tensor& lhs, const Tensor& rhs);
Tensor operator/(const Tensor& lhs, const Tensor& rhs);

Tensor operator+=(Tensor& lhs, const Tensor& rhs);
Tensor operator-=(Tensor& lhs, const Tensor& rhs);
Tensor operator*=(Tensor& lhs, const Tensor& rhs);
Tensor operator/=(Tensor& lhs, const Tensor& rhs);
```

Broadcasting:

```cpp
Tensor a(Shape{{2, 3}}, Dtype::Float32, Device::CPU,false); a.fill(1.0f)
Tensor b(Shape{{1,3}},Dtype::Float32,Device::CPU,false); b.fill(2.0f);
Tensor c = a + b; // [[3,3,3], [3,3,3]]
```

Tensor Operations can be broadcasted only if:

- Both are 2 dimensional tensors
- One of the dimensions of the Tensor should be 1 and the other should be equal to the Tensor between which the operations are performed.

Matmul - Matrix Multiplication:

The matmul function provides the core functionality for performing matrix multiplication on Tensor objects. It is a high-level dispatch function that automatically selects the appropriate backend (CPU or CUDA) based on the device of the input tensors.

Function Signature

```
C++
// Located in: include/ops/Kernels.h
OwnTensor::Tensor matmul(const OwnTensor::Tensor& A, const
OwnTensor::Tensor& B);
```

Description

This function computes the matrix product of two tensors. The behavior intelligently adapts based on the number of dimensions in the input tensors:
2D x 2D Tensors: Performs standard matrix multiplication. A tensor of shape (N, M) multiplied by a tensor of shape (M, P) results in a new tensor of shape (N, P).
Batched Multiplication (N-D Tensors): If the input tensors have more than two dimensions, the function performs a batched matrix multiplication.
The last two dimensions of each tensor are treated as the matrices to be multiplied.
All preceding dimensions are treated as "batch" dimensions, which are broadcast against each other.

Broadcasting Rules for Batch Dimensions
For the operation to be valid, the batch dimensions of A and B must be "broadcastable." This means that when comparing the batch dimension sizes from right to left, one of the following must be true for each dimension:

- The dimension sizes are equal.
- One of the dimension sizes is 1.

The resulting tensor's batch dimension will be the element-wise maximum of the two input batch dimensions.
Broadcasting Examples:

```
A(5, 2, 3) @ B(5, 3, 4) -> Result(5, 2, 4) (Batch dimensions (5) and (5) are equal)
A(5, 2, 3) @ B(1, 3, 4) -> Result(5, 2, 4) (Batch dimension 1 is broadcast up to 5)
A(4, 1, 2, 3) @ B(5, 3, 2) -> Result(4, 5, 3, 2) (Broadcasting 1 to 4 and 5 to 5)
A(5, 2, 3) @ B(2, 3, 4) -> Error! (Batch dimensions 5 and 2 are not equal and
neither is 1)
```

Parameters

Parameter	Type	Description
A	const Tensor&	The first input tensor (left operand). Must have at least 2 dimensions.
B	const Tensor&	The second input tensor (right operand). Must have at least 2 dimensions

Returns

A new Tensor object containing the result of the matrix multiplication. The returned tensor will be on the same device as the input tensors.

Validation and Error Handling

The matmul function performs several critical checks before dispatching to the CPU or GPU backend. It will throw a std::runtime_error if any of the following conditions are not met:

- Data Type Mismatch: A and B must have the exact same Dtype.
- Insufficient Dimensions: Both A and B must have at least 2 dimensions.
- Matrix Multiplication Incompatibility: The size of the last dimension of A must be equal to the size of the second-to-last dimension of B.
- Batch Dimension Incompatibility: The batch dimensions (all dimensions except the last two) must be broadcastable.

Example Usage

Here is how to use the matmul function with both 2D and batched tensors.
code

```
C++
#include <iostream>
#include "TensorLib.h" // Your library's main header

int main() {
    // --- Example 1: Simple 2D Matrix Multiplication ---
    std::cout << "--- 2D Matmul Example ---" << std::endl;
    OwnTensor::Tensor a(OwnTensor::Shape{{2, 3}});
    OwnTensor::Tensor b(OwnTensor::Shape{{3, 4}});

    a.fill_(2.0f); // Fill with some data
    b.fill_(3.0f);

    OwnTensor::Tensor result_2d = OwnTensor::matmul(a, b);
```

```
    // Expected shape: (2, 4)
    // Expected values: 2.0 * 3.0 * 3 = 18.0
    std::cout << "Result of 2D matmul has shape: " <<
result_2d.shape().to_string() << std::endl;
    result_2d.display(std::cout);


    // --- Example 2: Batched Matrix Multiplication with Broadcasting ---
    std::cout << "\n--- Batched Matmul Example ---" << std::endl;
    // A has shape (1, 4, 2, 3)
    // B has shape (   5, 3, 4)
    OwnTensor::Tensor batched_a(OwnTensor::Shape{{1, 4, 2, 3}});
    OwnTensor::Tensor batched_b(OwnTensor::Shape{{   5, 3, 4}});

    batched_a.fill_(1.0f);
    batched_b.fill_(2.0f);

    OwnTensor::Tensor result_batched = OwnTensor::matmul(batched_a,
batched_b);

    // Expected shape after broadcasting: (1, 4, 5, 2, 4) --> (4, 5, 2, 4)
    // The batch dimension of A (1, 4) is broadcast against B's (5)
    // to produce a final batch dimension of (4, 5).
    std::cout << "Result of batched matmul has shape: " <<
result_batched.shape().to_string() << std::endl;

    return 0;
}
```