xtensor-julia

Johan Mabille and Sylvain Corlay

Jul 01, 2022

USAGE

Julia bindings for the xtensor C++ multi-dimensional array library.

`xtensor` is a C++ library for multi-dimensional arrays enabling numpy-style broadcasting and lazy computing.

`xtensor-julia` enables inplace use of julia arrays in C++ with all the benefits from xtensor

- C++ universal function and broadcasting
- STL - compliant APIs.
- A broad coverage of numpy APIs (see the numpy to xtensor cheat sheet).

The Julia bindings for xtensor are based on the CxxWrap.jl C++ library.

CHAPTER

ONE

LICENSING

We use a shared copyright model that enables all contributors to maintain the copyright on their contributions.

This software is licensed under the BSD-3-Clause license. See the LICENSE file for details.

1.1 Basic Usage

1.1.1 Example 1: Use an algorithm of the C++ library with a Julia array

C++ code

```
#include <numeric>                          // Standard library import for std::accumulate
#include "jlcxx/jlcxx.hpp"                  // CxxWrap import to define Julia bindings
#include "xtensor-julia/jltensor.hpp"       // Import the jltensor container definition
#include "xtensor/xmath.hpp"                // xtensor import for the C++ universal␣
↪functions

double sum_of_sines(xt::jltensor<double, 2> m)
{
    auto sines = xt::sin(m);  // sines does not actually hold values.
    return std::accumulate(sines.cbegin(), sines.cend(), 0.0);
}

JLCXX_MODULE define_julia_module(jlcxx::Module& mod)
{
    mod.method("sum_of_sines", sum_of_sines);
}
```

Julia code:

```
using xtensor_julia_test

arr = [[1.0 2.0]
       [3.0 4.0]]

s = sum_of_sines(arr)
s
```

Outputs

```
1.2853996391883833
```

1.1.2 Example 2: Create a numpy-style universal function from a C++ scalar function

C++ code

```
#include "jlcxx/jlcxx.hpp"
#include "xtensor-julia/jlvectorize.hpp"

double scalar_func(double i, double j)
{
    return std::sin(i) - std::cos(j);
}

JLCXX_MODULE define_julia_module(jlcxx::Module& mod)
{
    mod.method("vectorized_func", xt::jlvectorize(scalar_func));
}
```

Julia code:

```
using xtensor_julia_test

x = [[ 0.0  1.0  2.0  3.0  4.0]
     [ 5.0  6.0  7.0  8.0  9.0]
     [10.0 11.0 12.0 13.0 14.0]]
y = [1.0, 2.0, 3.0, 4.0, 5.0]
z = xt.vectorized_func(x, y)
z
```

Outputs

```
[[-0.540302  1.257618  1.89929   0.794764 -1.040465],
 [-1.499227  0.136731  1.646979  1.643002  0.128456],
 [-1.084323 -0.583843  0.45342   1.073811  0.706945]]
```

1.2 Arrays and tensors

`xtensor-julia` provides two container types wrapping Julia arrays: `jlarray` and `jltensor`. They are the counterparts to `xarray` and `xtensor` containers.

1.2.1 jlarray

Like `xarray`, `jlarray` has a dynamic shape. This means that one can reshape the array on the C++ side and see this change reflected on the Julia side. `jlarray` doesn't make a copy of the shape, but reads the values each time it is needed. Therefore, if a reference on a `jlarray` is kept in the C++ code and the corresponding Julia array is then reshaped on the Julia side, this modification will reflect in the `jlarray`.

1.2.2 jltensor

Like `xtensor`, `jltensor` has a static stack-allocated shape. This means that the shape of the Julia array is copied into the shape of the `jltensor` upon creation. As a consequence, reshapes are not reflected across languages. However, this drawback is offset by a more effective iteration and broadcasting.

1.3 API reference

1.3.1 Containers

jlarray

template<class **T**>

class **jlarray** : public xt::jlcontainer<*jlarray*<*T*>>, public xcontainer_semantic<*jlarray*<*T*>>

Constructors

inline **jlarray**()

Allocates a jlarray that holds 1 element.

inline **jlarray**(const value_type &t)

Allocates a jlarray with a nested initializer list.

inline explicit **jlarray**(const shape_type &shape)

Allocates an uninitialized jlarray with the specified shape and layout.

Parameters

shape – the shape of the jlarray

inline explicit **jlarray**(const shape_type &shape, const_reference value)

Allocates a jlarray with the specified shape and layout.

Elements are initialized to the specified value.

Parameters

- **shape** – the shape of the jlarray
- **value** – the value of the elements

inline **jlarray**(jl_array_t *jl)

Allocates a jlarray that holds 1 element.

Copy semantic

inline **jlarray**(const self_type&)

The copy constructor.

inline self_type &**operator=**(const self_type&)

The assignment operator.

Extended copy semantic

template<class **E**>
inline **jlarray**(const xexpression<*E*> &e)

The extended copy constructor.

template<class **E**>
inline auto **operator=**(const xexpression<*E*> &e) -> self_type&

The extended assignment operator.

jltensor

template<class **T**, std::size_t **N**>

class **jltensor** : public xt::jlcontainer<*jltensor*<*T*, *N*>>, public xcontainer_semantic<*jltensor*<*T*, *N*>>

Constructors

inline **jltensor**()

Allocates a jltensor that holds 1 element.

inline **jltensor**(nested_initializer_list_t<*T*, *N*> t)

Allocates a jltensor with a nested initializer list.

inline explicit **jltensor**(const shape_type &shape)

Allocates an uninitialized jltensor with the specified shape and layout.

Parameters
shape – the shape of the jltensor

inline explicit **jltensor**(const shape_type &shape, const_reference value)

Allocates a jltensor with the specified shape and layout.

Elements are initialized to the specified value.

Parameters

- **shape** – the shape of the jltensor
- **value** – the value of the elements

inline **jltensor**(jl_array_t *jl)

Allocates a jltensor that holds 1 element.

Copy semantic

inline **jltensor**(const self_type&)

The copy constructor.

inline self_type &**operator=**(const self_type&)

The assignment operator.

Extended copy semantic

template<class **E**>
inline **jltensor**(const xexpression<*E*> &e)

The extended copy constructor.

template<class **E**>
inline auto **operator=**(const xexpression<*E*> &e) -> self_type&

The extended assignment operator.

1.3.2 Numpy-style universal functions

jlvectorize

template<class **R**, class ...**Args**>
inline jlvectorizer<*R* (*)(*Args*...), *R*, *Args*...> **xt::jlvectorize**(*R* (*f)(*Args*...))

Create numpy-style universal function from scalar function.

INDEX

X