Manual of Mars

Mars Developers

Sep. 15, 2016

Abstract

Mars provides C++ data structure Array, a multi-dimensional tensor for high-performance computation.

"Over the years considerations along these lines grew into the C++ 'principle' that it was not sufficient to provide a feature, it had to be provided in an affordable form. Most definitely, 'affordable' was seen as meaning 'affordable on hardware common among developers' as opposed to 'affordable to researchers with high-end equipment' or 'affordable in a couple of years when hardware will be cheaper." – A History of C++: 1979-1991, Bjarne Stroustrup

1 Array

1.1 Design Goals

The Array class is designed as a multidimensional array for high performance computing with goals: efficiency, flexibility, modularity, easiness of debugging, and code simplicity. When there is a conflict between efficiency and code simplicity, we usually choose efficiency. Fortunately, most of the implementational complexities are transparent to the end users.

A general technique we use to optimize efficiency is memorization of, for example the size, rank, strides of an array object even if they can be computed on the fly. This costs slightly more memory with a big gain in run time. Another technique is to put as much data as possible on the stack to minimize memory allocation and deallocation. We use StackVector, a small "vector" on the stack, to represent array shape and index. The max rank of array is determinated by the max size of StackVector. It is currently set as 6.

For flexibility and modularity, we use Array as a uniform interface to represent array of any dimension, whether the array object owns memory and whether it is a sliced view or not. An array object can easily decay to raw memory, which makes interacting with packages such as Lapack and FFTW easier.

For easiness of debugging, we insert various assertions in our implementation to check array shape, array bounds, etc. It is usually wise to keep these assertions when you debug the code.

1.2 User Interface

Mars let you create C++ multidimensional array and do vectorized operations on the array easily.

1.2.1 Array construction

The code snippet below creates a one-dimensional array with 10 elements and assigns value one by one. The array is then printed. In the code snippet, { M } is implicitly converted to a StackVector object, which is a small vector on the stack. Since StackVector is declared on the stack, it can be fast created. We also use StackVector as array index.

```
const long M = 10;
mars::Array<double> A({ M });
for(long i = 0; i < M; i ++)
    A(i) = i;
A.print();
```

```
Array(10):
   0 1 2 3 4 5 6 7 8 9
  +--------------------+
0 |0 1 2 3 4 5 6 7 8 9 |
  +--------------------+
```

1.2.2 Slicing

Select the elements at 1, 3, 5, 7 and 9 and print them. Type Slice is used to make array slicing easier.

```
A.slice(mars::Slice(1,10,2)).print();
```

```
Array(5):
   0 1 2 3 4
  +----------+
0 |1 3 5 7 9 |
  +----------+
```

We can operate on the sliced array without affecting other elements of the original array.

```
mars::Array<double> B = A.slice(mars::Slice(1,10,2));
B *= -1;
A.print();
```

```
Array(10):
   0  1 2  3 4  5 6  7 8  9
  +------------------------+
0 |0 -1 2 -3 4 -5 6 -7 8 -9 |
  +------------------------+
```

1.2.3 Operators

Operators =, +=, −=, ∗=, /= , %= are supported.

```
A += A;
A.print();
```

```
Array(10):
   0 1 2 3 4  5  6  7  8  9
  +------------------------+
0 |0 2 4 6 8 10 12 14 16 18 |
  +------------------------+
```

1.2.4 Map in place

An array can be transformed by the map method with function pointer or lambda. Below we first apply sin function to the array and then floor the values at 0.

```
A.map(sin);
A.print();
```

```
Array(10):
       0      1       2       3      4       5       6      7       8
9
  +---------------------------------------------------------------------------+
0 |0.0000 0.9093 -0.7568 -0.2794 0.9894 -0.5440 -0.5366 0.9906 -0.2879 -0.7510 |
  +---------------------------------------------------------------------------+
```

```
A.map([](double x) -> double { return std::max(x, 0.0); });
A.print();
```

```
Array(10):
       0      1      2      3      4      5      6      7      8
9
  +---------------------------------------------------------------------+
0 |0.0000 0.9093 0.0000 0.0000 0.9894 0.0000 0.0000 0.9906 0.0000 0.0000 |
  +---------------------------------------------------------------------+
```

1.2.5 Two dimensional array

A two dimensional array is created when the parameter is a two dimensional StackVector. Method range(start, step) sets the first value as start and adds step sequentially. A slicing example of two dimensional array is also shown below.

```
const long M = 10;
mars::Array<double> A({ M, M });
A.range(0, 1);
```

```
Array(10,10):
     0  1  2  3  4  5  6  7  8  9
   +------------------------------+
00 | 0  1  2  3  4  5  6  7  8  9 |
01 |10 11 12 13 14 15 16 17 18 19 |
02 |20 21 22 23 24 25 26 27 28 29 |
03 |30 31 32 33 34 35 36 37 38 39 |
04 |40 41 42 43 44 45 46 47 48 49 |
05 |50 51 52 53 54 55 56 57 58 59 |
06 |60 61 62 63 64 65 66 67 68 69 |
07 |70 71 72 73 74 75 76 77 78 79 |
08 |80 81 82 83 84 85 86 87 88 89 |
09 |90 91 92 93 94 95 96 97 98 99 |
   +------------------------------+
```

```
    A.slice(mars::Slice(1, M, 2), mars::Slice(1, M, 2)).print();
```

```
Array(5,5):
    0  1  2  3  4
  +---------------+
0 |11 13 15 17 19 |
1 |31 33 35 37 39 |
2 |51 53 55 57 59 |
3 |71 73 75 77 79 |
4 |91 93 95 97 99 |
  +---------------+
```

1.2.6 Reshaping

A three dimensional array is reshaped from dimension $2 \times 3 \times 5$ to 5×6 require the array size is kept after reshaping.

```
    mars::Array<double> A({ 2, 3, 5 });
    A.range(0, 1);
    A.print();
    A.reshape( {5, 6} );
    A.print();
```

Before reshaping:

```
Array(2,3,5):
[0]:
    0  1  2  3  4
  +---------------+
0 | 0  1  2  3  4 |
1 | 5  6  7  8  9 |
2 |10 11 12 13 14 |
  +---------------+
[1]:
    0  1  2  3  4
  +---------------+
0 |15 16 17 18 19 |
1 |20 21 22 23 24 |
2 |25 26 27 28 29 |
  +---------------+
```

After reshaping:

```
    0  1  2  3  4  5
  +------------------+
0 | 0  1  2  3  4  5 |
1 | 6  7  8  9 10 11 |
2 |12 13 14 15 16 17 |
3 |18 19 20 21 22 23 |
4 |24 25 26 27 28 29 |
  +------------------+
```