

S

educative

EXPLORE

TRACKS

MY COURSES

EDPRESSO

REFER A FRIEND

From Python to Numpy

0% COMPLETED

Q

Search Course

Back to Python

NumPy & co

Scipy & co

Conclusion

Conclusion

About the Author

A Quick Overview

# Memory-aware Array: Array Subclass in GPUData class

In this lesson, we will learn how to subclass ndarray to use it in the GPUData class.

## We'll cover the following

- Computing extents
- Keeping track of pending data
- GPUData Class Implementation

As explained in the [Subclassing ndarray](#) documentation, subclassing `ndarray` is complicated by the fact that new instances of `ndarray` classes can come about in three different ways:

- Explicit constructor call
- View casting
- New from template

However, our case is simpler because we're only interested in the view casting. We thus only need to define the `__new__` method that will be called at each instance creation. As such, the `GPUData` class will be equipped with two properties:

- `extents`: This represents the full extent of the view relatively to the base array. It is stored as a byte offset and a byte size.
- `pending_data`: This represents the contiguous *dirty* area as (byte offset, byte size) relatively to the extents property.

```
1 import numpy as np
2
3 Memory tracked numpy array.
4
5 class GPUData(np.ndarray):
6     def __new__(cls, *args, **kwargs):
7         return np.ndarray.__new__(cls, *args, **kwargs)
8
9     def __init__(self, *args, **kwargs):
10         pass
11
12     def __array_finalize__(self, obj):
13         if not isinstance(obj, GPUData):
14             self._extents = 0, self.size*self.itemsize
15             self.__class__ = self.__init__(self)
16             self.pending_data = self._extents
17         else:
18             self._extents = obj._extents
19
```

## Computing extents

Each time a partial view of the array is requested, we need to compute the extents of this partial view while we have access to the base array.

```
1 def __getitem__(self, key):
2     """ FIXME: Need to take care of case where key is a list or array """
3     Z = np.ndarray.__getitem__(self, key)
4     if not hasattr(Z, 'shape') or Z.shape == ():
5         return Z
6     Z._extents = self._compute_extents(Z)
7     return Z
8
9 def _compute_extents(self, Z):
10     """
11     Compute extents (start, stop) in the base array.
12     """
13     if self.base is not None:
14         base = self.base.__array_interface__['data'][0]
15         view = Z.__array_interface__['data'][0]
16         offset = view - base
17         shape = np.array(Z.shape) - 1
18         strides = np.array(Z.strides)
19         size = (shape*strides).sum() + Z.itemsize
20         return offset, offset+size
21     else:
22         return 0, self.size*self.itemsize

```

## Keeping track of pending data

One extra difficulty is that we don't want all the views to keep track of the dirty area but only the base array. This is the reason why we don't instantiate the `self._pending_data` in the second case of the `__array_finalize__` method. This will be handled when we need to update some data as during a `__setitem__` call for example:

```
1 def __setitem__(self, key, value):
2     """ FIXME: Need to take care of case where key is a list or array """
3     Z = np.ndarray.__getitem__(self, key)
4     if Z.shape == ():
5         key = np.mod(np.array(key)+self.shape, self.shape)
6         offset = self._extents[0]+(key * self.strides).sum()
7         size = Z.itemsize
8         self._add_pending_data(offset, offset+size)
9         key = tuple(key)
10    else:
11        Z._extents = self._compute_extents(Z)
12        self._add_pending_data(Z._extents[0], Z._extents[1])
13        np.ndarray.__setitem__(self, key, value)
14
15
16 def _add_pending_data(self, start, stop):
17     """
18     Add pending data, taking care of previous pending data such that it
19     is always a contiguous area.
20     """
21     base = self.base
22     if isinstance(base, GPUData):
23         base._add_pending_data(start, stop)
24     else:
25         if self._pending_data is None:
26             self._pending_data = start, stop
27         else:
28             start = min(self._pending_data[0], start)
29             stop = max(self._pending_data[1], stop)
30             self._pending_data = start, stop

```

## GPUData Class Implementation

**GPU data** is the base class for any data that needs to co-exist on both CPU and GPU memory. It keeps track of the smallest contiguous area that needs to be uploaded to GPU to keep the CPU and GPU data synced. This allows to update the data in one operation. Even though this might be sub-optimal in a few cases, it provides a greater usage flexibility and most of the time, it will be faster.

This is done transparently and user can use a GPU buffer as a regular numpy array. The `pending_data` property indicates the region (offset/nbytes) of the base array that needs to be uploaded. Here is the complete implementation of the GPUData class after combining all the codes from above:

```
1 # -----
2 # Copyright (c) 2009-2016 Nicolas P. Rougier. All rights reserved.
3 # Distributed under the (new) BSD License.
4 # -----
5
6 import numpy as np
7
8
9 class GPUData(np.ndarray):
10     """
11     Memory tracked numpy array.
12     """
13
14     def __new__(cls, *args, **kwargs):
15         return np.ndarray.__new__(cls, *args, **kwargs)
16
17     def __init__(self, *args, **kwargs):
18         pass
19
20     def __array_finalize__(self, obj):
21         if not isinstance(obj, GPUData):
22             self._extents = 0, self.size*self.itemsize
23             self.__class__ = self.__init__(self)
24             self.pending_data = self._extents
25         else:
26             self._extents = obj._extents
27
28
29 @property
30 def pending_data(self):
31     """ Pending data region as (byte offset, byte size) """
32
33     if isinstance(self.base, GPUData):
34         return self.base.pending_data
35
36     if self._pending_data:
37         return self._pending_data
38         # start, stop = self._pending_data
39         # WARN: semantic is offset, nbytes
40         # extents semantic is start, stop
41         # return start, stop-start
42         return start, stop
43     else:
44         return None
45
46
47 @property
48 def stride(self):
49     """ Item stride in the base array. """
50
51     if self.base is None:
52         return self.ravel().strides[0]
53     else:
54         return self.base.ravel().strides[0]
55
56
57 @property
58 def offset(self):
59     """ Byte offset in the base array. """
60
61     return self._extents[0]
62
63
64 def _add_pending_data(self, start, stop):
65     """
66     Add pending data, taking care of previous pending data such that it
67     is always a contiguous area.
68     """
69     base = self.base
70     if isinstance(base, GPUData):
71         base._add_pending_data(start, stop)
72     else:
73         if self._pending_data is None:
74             self._pending_data = start, stop
75         else:
76             start = min(self._pending_data[0], start)
77             stop = max(self._pending_data[1], stop)
78             self._pending_data = start, stop
79
80
81 def _compute_extents(self, Z):
82     """
83     Compute extents (start, stop) in the base array.
84     """
85
86     if self.base is not None:
87         base = self.base.__array_interface__['data'][0]
88         view = Z.__array_interface__['data'][0]
89         offset = view - base
90         shape = np.array(Z.shape) - 1
91         strides = np.array(Z.strides)
92         size = (shape*strides).sum() + Z.itemsize
93         return offset, offset+size
94     else:
95         return 0, self.size*self.itemsize
96
97
98 def __getitem__(self, key):
99     """ FIXME: Need to take care of case where key is a list or array """
100
101     Z = np.ndarray.__getitem__(self, key)
102     if not hasattr(Z, 'shape') or Z.shape == ():
103         return Z
104     Z._extents = self._compute_extents(Z)
105     return Z
106
107
108 def __setitem__(self, key, value):
109     """ FIXME: Need to take care of case where key is a list or array """
110
111     Z = np.ndarray.__getitem__(self, key)
112     if Z.shape == ():
113         # WARN: Be careful with negative indices !
114         key = np.mod(np.array(key)+self.shape, self.shape)
115         offset = self._extents[0]+(key * self.strides).sum()
116         size = Z.itemsize
117         self._add_pending_data(offset, offset+size)
118         key = tuple(key)
119     else:
120         Z._extents = self._compute_extents(Z)
121         self._add_pending_data(Z._extents[0], Z._extents[1])
122         np.ndarray.__setitem__(self, key, value)
123
124
125 def __getslice__(self, start, stop):
126     return self.__getitem__(slice(start, stop))
127
128
129 def __setslice__(self, start, stop, value):
130     return self.__setitem__(slice(start, stop), value)
131
132
133 def __iadd__(self, other):
134     self._add_pending_data(self._extents[0], self._extents[1])
135     return np.ndarray.__iadd__(self, other)
136
137
138 def __isub__(self, other):
139     self._add_pending_data(self._extents[0], self._extents[1])
140     return np.ndarray.__isub__(self, other)
141
142
143 def __imul__(self, other):
144     self._add_pending_data(self._extents[0], self._extents[1])
145     return np.ndarray.__imul__(self, other)
146
147
148 def __idiv__(self, other):
149     self._add_pending_data(self._extents[0], self._extents[1])
150     return np.ndarray.__idiv__(self, other)
151
152
153 data = np.zeros((5,5)).view(GPUData)
154 print ("data:\n",data)
155 print ("data.pending_data:",data.pending_data)

```

RUN

SAVE

RESET

Output0.657s

data:  
[[0. 0. 0. 0.]  
 [0. 0. 0. 0.]  
 [0. 0. 0. 0.]  
 [0. 0. 0. 0.]  
 [0. 0. 0. 0.]]  
data.pending\_data: (0, 200)

Next, we will look at a few important library that can be used along with NumPy for multiple purposes.

← Back

Memory-aware Array: Glumpy

Mark as Completed

Next →

Conclusion