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

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.

```python
import numpy as np
"""
Memory tracked numpy array.
"""
class GPUData(np.ndarray):
  def __new__(cls, *args, **kwargs):
    return np.ndarray.__new__(cls, *args, **kwargs)

  def __init__(self, *args, **kwargs):
    pass

  def __array_finalize__(self, obj):
    if not isinstance(obj, GPUData):
      self._extents = 0, self.size*self.itemsize
      self.__class__.__init__(self)
      self._pending_data = self._extents
    else:
      self._extents = obj._extents

```

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.

```python
def __getitem__(self, key):
""" FIXME: Need to take care of case where key is a list or array """
  Z = np.ndarray.__getitem__(self, key)
  if not hasattr(Z,'shape') or Z.shape == ():
    return Z
  Z._extents = self._compute_extents(Z)
  return Z

def _compute_extents(self, Z):
"""
Compute extents (start, stop) in the base array.
"""
  if self.base is not None:
    base = self.base.__array_interface__['data'][0]
    view = Z.__array_interface__['data'][0]
    offset = view - base
    shape = np.array(Z.shape) - 1
    strides = np.array(Z.strides)
    size = (shape*strides).sum() + Z.itemsize
    return offset, offset+size
  else:
    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:

```python
def __setitem__(self, key, value):
""" FIXME: Need to take care of case where key is a list or array """
  Z = np.ndarray.__getitem__(self, key)
  if Z.shape == ():
    key = np.mod(np.array(key)+self.shape, self.shape)
    offset = self._extents[0]+(key * self.strides).sum()
    size = Z.itemsize
    self._add_pending_data(offset, offset+size)
    key = tuple(key)
  else:
    Z._extents = self._compute_extents(Z)
    self._add_pending_data(Z._extents[0], Z._extents[1])
  np.ndarray.__setitem__(self, key, value)


def _add_pending_data(self, start, stop):
"""
Add pending data, taking care of previous pending data such that it
is always a contiguous area.
"""
  base = self.base
  if isinstance(base, GPUData):
    base._add_pending_data(start, stop)
  else:
    if self._pending_data is None:
      self._pending_data = start, stop
    else:
      start = min(self._pending_data[0], start)
      stop = max(self._pending_data[1], stop)
      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:

```python
# -----------------------------------------------------------------------------
# Copyright (c) 2009-2016 Nicolas P. Rougier. All rights reserved.
# Distributed under the (new) BSD License.
# -----------------------------------------------------------------------------

import numpy as np


class GPUData(np.ndarray):
    """
    Memory tracked numpy array.
    """

    def __new__(cls, *args, **kwargs):
        return np.ndarray.__new__(cls, *args, **kwargs)

    def __init__(self, *args, **kwargs):
        pass

    def __array_finalize__(self, obj):
        if not isinstance(obj, GPUData):
            self._extents = 0, self.size*self.itemsize
            self.__class__.__init__(self)
            self._pending_data = self._extents
        else:
            self._extents = obj._extents


    @property
    def pending_data(self):
        """ Pending data region as (byte offset, byte size) """

        if isinstance(self.base, GPUData):
            return self.base.pending_data

        if self._pending_data:
            return self._pending_data
            # start, stop = self._pending_data
            # WARN: semantic is offset, nbytes
            # extents semantic is start, stop
            # return start, stop-start
            return start, stop
        else:
            return None

    @property
    def stride(self):
        """ Item stride in the base array. """

        if self.base is None:
            return self.ravel().strides[0]
        else:
            return self.base.ravel().strides[0]

    @property
    def offset(self):
        """ Byte offset in the base array. """

        return self._extents[0]


    def _add_pending_data(self, start, stop):
        """
        Add pending data, taking care of previous pending data such that it
        is always a contiguous area.
        """
        base = self.base
        if isinstance(base, GPUData):
            base._add_pending_data(start, stop)
        else:
            if self._pending_data is None:
                self._pending_data = start, stop
            else:
                start = min(self._pending_data[0], start)
                stop = max(self._pending_data[1], stop)
                self._pending_data = start, stop

    def _compute_extents(self, Z):
        """
        Compute extents (start, stop) in the base array.
        """

        if self.base is not None:
            base = self.base.__array_interface__['data'][0]
            view = Z.__array_interface__['data'][0]
            offset = view - base
            shape = np.array(Z.shape) - 1
            strides = np.array(Z.strides)
            size = (shape*strides).sum() + Z.itemsize
            return offset, offset+size
        else:
            return 0, self.size*self.itemsize


    def __getitem__(self, key):
        """ FIXME: Need to take care of case where key is a list or array """

        Z = np.ndarray.__getitem__(self, key)
        if not hasattr(Z,'shape') or Z.shape == ():
            return Z
        Z._extents = self._compute_extents(Z)
        return Z

    def __setitem__(self, key, value):
        """ FIXME: Need to take care of case where key is a list or array """

        Z = np.ndarray.__getitem__(self, key)
        if Z.shape == ():
            # WARN: Be careful with negative indices !
            key = np.mod(np.array(key)+self.shape, self.shape)
            offset = self._extents[0]+(key * self.strides).sum()
            size = Z.itemsize
            self._add_pending_data(offset, offset+size)
            key = tuple(key)
        else:
            Z._extents = self._compute_extents(Z)
            self._add_pending_data(Z._extents[0], Z._extents[1])
        np.ndarray.__setitem__(self, key, value)


    def __getslice__(self, start, stop):
        return self.__getitem__(slice(start, stop))

    def __setslice__(self, start, stop,  value):
        return self.__setitem__(slice(start, stop), value)

    def __iadd__(self, other):
        self._add_pending_data(self._extents[0], self._extents[1])
        return np.ndarray.__iadd__(self, other)

    def __isub__(self, other):
        self._add_pending_data(self._extents[0], self._extents[1])
        return np.ndarray.__isub__(self, other)

    def __imul__(self, other):
        self._add_pending_data(self._extents[0], self._extents[1])
        return np.ndarray.__imul__(self, other)

    def __idiv__(self, other):
        self._add_pending_data(self._extents[0], self._extents[1])
        return np.ndarray.__idiv__(self, other)



data = np.zeros((5,5)).view(GPUData)
print ("data:\n",data)
print ("data.pending_data:",data.pending_data)
```

RUN SAVE RESET

Output 0.657s

```
data:
 [[0. 0. 0. 0. 0.]
 [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.