Grid Coordinate Conversion with CuPy

Much of our data is provided with latitude and longitude coordinates, but for some of our machine learning tasks involving distance - identifying geographically dense clusters of infected people, locating the nearest hospital or clinic from a given person - it is convenient to have Cartesian grid coordinates instead. Our road data comes with those coordinates, as well. By using a region-specific map projection - in this case, the Ordnance Survey Great Britain 1936 - we can compute local distances efficiently and with good accuracy.

In this notebook you will use a user-defined function to perform data manipulation, generating grid coordinate values. In doing so, you will learn more about the powerful GPU-accelerated drop-in replacement library for NumPy called CuPy.

Objectives

By the time you complete this notebook you will be able to:

- Use CuPy to GPU-accelerate data transformations using user-defined functions

Imports

In [1]:
```
import cudf

import numpy as np
import cupy as cp
```

Read Data

For this notebook we will load the UK population data again. Here and later, when reading data from disk we will provide the named argument `dtype` to specify the data types we want the columns to load as.

In [2]:
```
%time gdf = cudf.read_csv('./data/pop_1-05.csv', dtype=['float32', 'str', 'str', 'floa
```

```
CPU times: user 2.1 s, sys: 574 ms, total: 2.67 s
Wall time: 2.67 s
```

In [3]:
```
gdf.dtypes
```

Out[3]:
```
age       float32
sex        object
county     object
lat       float32
long      float32
name       object
dtype: object
```

In [4]:
```
gdf.shape
```

Out[4]:
```
(58479894, 6)
```

Lat/Long to OSGB Grid Converter with NumPy

To perform coordinate conversion, we will create a function `latlong2osgbgrid` which accepts latitude/longitude coordinates and converts them to OSGB36 coordinates: "northing" and "easting" values representing the point's Cartesian coordinate distances from the southwest corner of the grid.

Immediately below is `latlong2osgbgrid`, which relies heavily on NumPy:

In [5]:
```
# https://www.ordnancesurvey.co.uk/docs/support/guide-coordinate-systems-great-britain

def latlong2osgbgrid(lat, long, input_degrees=True):
    '''
    Converts latitude and longitude (ellipsoidal) coordinates into northing and eastin

    Inputs:
    lat: latitude coordinate (north)
    long: longitude coordinate (east)
    input_degrees: if True (default), interprets the coordinates as degrees; otherwise

    Output:
    (northing, easting)
    '''

    if input_degrees:
        lat = lat * np.pi/180
        long = long * np.pi/180

    a = 6377563.396
    b = 6356256.909
    e2 = (a**2 - b**2) / a**2

    N0 = -100000                # northing of true origin
    E0 = 400000                 # easting of true origin
    F0 = .9996012717            # scale factor on central meridian
    phi0 = 49 * np.pi / 180     # latitude of true origin
    lambda0 = -2 * np.pi / 180  # longitude of true origin and central meridian

    sinlat = np.sin(lat)
    coslat = np.cos(lat)
    tanlat = np.tan(lat)

    latdiff = lat-phi0
    longdiff = long-lambda0

    n = (a-b) / (a+b)
    nu = a * F0 * (1 - e2 * sinlat ** 2) ** -.5
    rho = a * F0 * (1 - e2) * (1 - e2 * sinlat ** 2) ** -1.5
    eta2 = nu / rho - 1
    M = b * F0 * ((1 + n + 5/4 * (n**2 + n**3)) * latdiff -
                  (3*(n+n**2) + 21/8 * n**3) * np.sin(latdiff) * np.cos(lat+phi0) +
```

```
                15/8 * (n**2 + n**3) * np.sin(2*(latdiff)) * np.cos(2*(lat+phi0)) -
                35/24 * n**3 * np.sin(3*(latdiff)) * np.cos(3*(lat+phi0)))
    I = M + N0
    II = nu/2 * sinlat * coslat
    III = nu/24 * sinlat * coslat ** 3 * (5 - tanlat ** 2 + 9 * eta2)
    IIIA = nu/720 * sinlat * coslat ** 5 * (61-58 * tanlat**2 + tanlat**4)
    IV = nu * coslat
    V = nu / 6 * coslat**3 * (nu/rho - np.tan(lat)**2)
    VI = nu / 120 * coslat ** 5 * (5 - 18 * tanlat**2 + tanlat**4 + 14 * eta2 - 58 * t

    northing = I + II * longdiff**2 + III * longdiff**4 + IIIA * longdiff**6
    easting = E0 + IV * longdiff + V * longdiff**3 + VI * longdiff**5

    return(northing, easting)
```

Testing the NumPy Converter

To test the converter and check its performance, here we generate 10,000,000 normally distributed random coordinates within the rough bounds of the latitude and longitude ranges of the UK.

In [6]:
```
%%time
coord_lat = np.random.normal(54, 1, 10000000)
coord_long = np.random.normal(-1.5, .25, 10000000)
```

```
CPU times: user 485 ms, sys: 35.8 ms, total: 520 ms
Wall time: 520 ms
```

We now pass these latitude/longitude coordinates into the converter, which returns north and east values within the OSGB grid:

In [7]:
```
%time grid_n, grid_e = latlong2osgbgrid(coord_lat, coord_long)
print(grid_n[:5], grid_e[:5])
```

```
CPU times: user 4.35 s, sys: 1.09 s, total: 5.44 s
Wall time: 5.43 s
[293139.45469581 475895.80082272 382428.98823672 467035.08774159
 521095.60150287] [465140.25474169 426219.50138935 436864.53099719 426483.3887452
 442044.17914406]
```

Lat/Long to OSGB Grid Converter with CuPy

CuPy is a NumPy-like matrix library that can often be used as a drop in replacement for NumPy.

In the following `latlong2osgbgrid_cupy`, we simply swap `cp` in for `np`. While CuPy supports a wide variety of powerful GPU-accelerated tasks, this simple technique of being able to swap in CuPy calls for NumPy calls makes it an incredibly powerful tool to have at your disposal.

In [8]:
```
# https://www.ordnancesurvey.co.uk/docs/support/guide-coordinate-systems-great-britair

def latlong2osgbgrid_cupy(lat, long, input_degrees=True):
    '''
    Converts latitude and longitude (ellipsoidal) coordinates into northing and eastir
```

```
    Inputs:
    lat: latitude coordinate (north)
    long: longitude coordinate (east)
    input_degrees: if True (default), interprets the coordinates as degrees; otherwise

    Output:
    (northing, easting)
    '''

    if input_degrees:
        lat = lat * cp.pi/180
        long = long * cp.pi/180

    a = 6377563.396
    b = 6356256.909
    e2 = (a**2 - b**2) / a**2

    N0 = -100000                # northing of true origin
    E0 = 400000                 # easting of true origin
    F0 = .9996012717            # scale factor on central meridian
    phi0 = 49 * cp.pi / 180     # latitude of true origin
    lambda0 = -2 * cp.pi / 180  # longitude of true origin and central meridian

    sinlat = cp.sin(lat)
    coslat = cp.cos(lat)
    tanlat = cp.tan(lat)

    latdiff = lat-phi0
    longdiff = long-lambda0

    n = (a-b) / (a+b)
    nu = a * F0 * (1 - e2 * sinlat ** 2) ** -.5
    rho = a * F0 * (1 - e2) * (1 - e2 * sinlat ** 2) ** -1.5
    eta2 = nu / rho - 1
    M = b * F0 * ((1 + n + 5/4 * (n**2 + n**3)) * latdiff -
                  (3*(n+n**2) + 21/8 * n**3) * cp.sin(latdiff) * cp.cos(lat+phi0) +
                  15/8 * (n**2 + n**3) * cp.sin(2*(latdiff)) * cp.cos(2*(lat+phi0)) -
                  35/24 * n**3 * cp.sin(3*(latdiff)) * cp.cos(3*(lat+phi0)))
    I = M + N0
    II = nu/2 * sinlat * coslat
    III = nu/24 * sinlat * coslat ** 3 * (5 - tanlat ** 2 + 9 * eta2)
    IIIA = nu/720 * sinlat * coslat ** 5 * (61-58 * tanlat**2 + tanlat**4)
    IV = nu * coslat
    V = nu / 6 * coslat**3 * (nu/rho - cp.tan(lat)**2)
    VI = nu / 120 * coslat ** 5 * (5 - 18 * tanlat**2 + tanlat**4 + 14 * eta2 - 58 * t

    northing = I + II * longdiff**2 + III * longdiff**4 + IIIA * longdiff**6
    easting = E0 + IV * longdiff + V * longdiff**3 + VI * longdiff**5

    return(northing, easting)
```

Testing the CuPy Converter

Here we perform the same operations as we did with NumPy above, only the conversion runs significantly faster. Once you have run the cells below, try rerunning the NumPy converter

above (including random number generation) and then the CuPy converter - you may see even larger differences.

In [9]:
```
%%time
coord_lat = cp.random.normal(54, 1, 10000000)
coord_long = cp.random.normal(-1.5, .25, 10000000)
```

```
CPU times: user 15.1 ms, sys: 7.6 ms, total: 22.7 ms
Wall time: 22.4 ms
```

In [10]:
```
%time grid_n, grid_e = latlong2osgbgrid_cupy(coord_lat, coord_long)
print(grid_n[:5], grid_e[:5])
```

```
CPU times: user 3.62 s, sys: 198 ms, total: 3.82 s
Wall time: 3.81 s
[492863.2422679  596424.98137755 386517.03535701 511642.22598799
 472782.20534888] [446483.18774235 417903.26706023 414379.77879251 438642.46527973
 445424.69230954]
```

Adding Grid Coordinate Columns to Dataframe

Now we will utilize `latlong2osgbgrid_cupy` to add `northing` and `easting` columns to `gdf`. We start by converting the two columns we need, `lat` and `long`, to CuPy arrays with the `cp.asarray` method. Because cuDF and CuPy interface directly via the `__cuda_array_interface__`, the conversion can happen in nanoseconds.

In [11]:
```
%%time
cupy_lat = cp.asarray(gdf['lat'])
cupy_long = cp.asarray(gdf['long'])
```

```
CPU times: user 197 µs, sys: 67 µs, total: 264 µs
Wall time: 284 µs
```

Exercise: Create Grid Columns

For this exercise, now that you have GPU arrays for `lat` and `long`, you will create `northing` and `easting` columns in `gdf`. To do this:

- Use `latlong2osgbgrid_cupy` with `cupy_lat` and `cupy_long`, just created, to make CuPy arrays of the grid coordinates
- Create cuDF series out of each of these coordinate CuPy arrays and set the dtype to `float32`
- Add these two new series to `gdf`, calling them `northing` and `easting`

In [13]:
```
n_cupy_array, e_cupy_array = latlong2osgbgrid_cupy(cupy_lat, cupy_long)
gdf['northing'] = cudf.Series(n_cupy_array).astype('float32')
gdf['easting'] = cudf.Series(e_cupy_array).astype('float32')
print(gdf.dtypes)
gdf.head()
```

```
age          float32
sex           object
county        object
lat          float32
long         float32
name          object
northing     float32
easting      float32
dtype: object
```

Out[13]:

	age	sex	county	lat	long	name	northing	easting
0	0.0	m	Darlington	54.533638	-1.524400	Francis	515491.90625	430772.15625
1	0.0	m	Darlington	54.426254	-1.465314	Edward	503572.46875	434685.87500
2	0.0	m	Darlington	54.555199	-1.496417	Teddy	517903.65625	432565.53125
3	0.0	m	Darlington	54.547905	-1.572341	Angus	517059.90625	427660.65625
4	0.0	m	Darlington	54.477638	-1.605995	Charlie	509228.68750	425527.78125

Solution

In [14]:

```
# %load solutions/create_grid_columns
n_cupy_array, e_cupy_array = latlong2osgbgrid_cupy(cupy_lat, cupy_long)
gdf['northing'] = cudf.Series(n_cupy_array).astype('float32')
gdf['easting'] = cudf.Series(e_cupy_array).astype('float32')
print(gdf.dtypes)
gdf.head()
```

```
age          float32
sex           object
county        object
lat          float32
long         float32
name          object
northing     float32
easting      float32
dtype: object
```

Out[14]:

	age	sex	county	lat	long	name	northing	easting
0	0.0	m	Darlington	54.533638	-1.524400	Francis	515491.90625	430772.15625
1	0.0	m	Darlington	54.426254	-1.465314	Edward	503572.46875	434685.87500
2	0.0	m	Darlington	54.555199	-1.496417	Teddy	517903.65625	432565.53125
3	0.0	m	Darlington	54.547905	-1.572341	Angus	517059.90625	427660.65625
4	0.0	m	Darlington	54.477638	-1.605995	Charlie	509228.68750	425527.78125

Please Restart the Kernel

In [15]:

```
import IPython
app = IPython.Application.instance()
app.kernel.do_shutdown(True)
```

Out[15]:
```
{'status': 'ok', 'restart': True}
```

Next

In the next notebook we will return to fundamental cuDF operations, focusing on data analysis with grouping and sorting.