Introduction to cuDF

You will begin your accelerated data science training with an introduction to cuDF, the RAPIDS API that enables you to create and manipulate GPU-accelerated dataframes. cuDF implements a very similar interface to Pandas so that Python data scientists can use it with very little ramp up. Throughout this notebook we will provide Pandas counterparts to the cuDF operations you perform to build your intuition about how much faster cuDF can be, even for seemingly simple operations.

Objectives

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

- Read and write data to and from disk with cuDF
- Perform basic data exploration and cleaning operations with cuDF

Imports

Here we import cuDF and CuPy for GPU-accelerated dataframes and math operations, plus the CPU libraries Pandas and NumPy on which they are based and which we will use for performance comparisons:

In [1]:
```
import cudf
import cupy as cp

import pandas as pd
import numpy as np
```

Reading and Writing Data

Using cuDF, the RAPIDS API providing a GPU-accelerated dataframe, we can read data from a variety of formats, including csv, json, parquet, feather, orc, and Pandas dataframes, among others.

For the first part of this workshop, we will be reading almost 60 million records (corresponding to the entire population of England and Wales) which were synthesized from official UK census data. Here we read this data from a local csv file directly into GPU memory:

In [2]:
```
%time gdf = cudf.read_csv('./data/pop_1-03.csv')
gdf.shape
```

```
CPU times: user 2.33 s, sys: 580 ms, total: 2.9 s
Wall time: 2.9 s
```

Out[2]:
```
(58479894, 6)
```

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

Out[3]:
```
age          int64
sex         object
county      object
lat        float64
long       float64
name        object
dtype: object
```

Here for comparison we read the same data into a Pandas dataframe:

In [4]:
```
%time df = pd.read_csv('./data/pop_1-03.csv')
gdf.shape == df.shape
```

```
CPU times: user 25 s, sys: 3.74 s, total: 28.7 s
Wall time: 28.7 s
```

Out[4]:
```
True
```

Because of the sophisticated GPU memory management behind the scenes in cuDF, the first data load into a fresh RAPIDS memory environment is sometimes substantially slower than subsequent loads. The RAPIDS Memory Manager is preparing additional memory to accommodate the array of data science operations that you may be interested in using on the data, rather than allocating and deallocating the memory repeatedly throughout your workflow.

We will be using `gdf` regularly in this workshop to represent a GPU dataframe, as well as `df` for a CPU dataframe when comparing performance.

Writing to File

cuDF also provides methods for writing data to files. Here we create a new dataframe specifically containing residents of Blackpool county and then write it to `blackpool.csv` , before doing the same with Pandas for comparison.

cuDF

In [5]:
```
%time blackpool_residents = gdf.loc[gdf['county'] == 'BLACKPOOL']
print(f'{blackpool_residents.shape[0]} residents')
```

```
CPU times: user 87.2 ms, sys: 15.8 ms, total: 103 ms
Wall time: 102 ms
139305 residents
```

In [6]:
```
%time blackpool_residents.to_csv('blackpool.csv')
```

```
CPU times: user 10.2 ms, sys: 7.54 ms, total: 17.7 ms
Wall time: 16.9 ms
```

Pandas

In [7]:
```
%time blackpool_residents_pd = df.loc[df['county'] == 'BLACKPOOL']
```

```
CPU times: user 3.09 s, sys: 177 ms, total: 3.27 s
Wall time: 3.23 s
```

In [8]:
```
%time blackpool_residents_pd.to_csv('blackpool_pd.csv')
```

```
CPU times: user 450 ms, sys: 0 ns, total: 450 ms
Wall time: 449 ms
```

Exercise: Initial Data Exploration

Now that we have some data loaded, let's do some initial exploration.

Use the `head`, `dtypes`, and `columns` methods on `gdf`, as well as the `value_counts` on individual `gdf` columns, to orient yourself to the data. If you're interested, use the `%time` magic command to compare performance against the same operations on the Pandas `df`.

You can create additional interactive cells by clicking the `+` button above, or by switching to command mode with `Esc` and using the keyboard shortcuts `a` (for new cell above) and `b` (for new cell below).

If you fill up the GPU memory at any time, don't forget that you can restart the kernel and rerun the cells up to this point quite quickly.

In [10]:
```
# Begin your initial exploration here. Create more cells as needed.

print(gdf.head())

print(gdf.dtypes)

print(gdf.columns)

print(blackpool_residents_pd.value_counts())
```

```
   age sex      county        lat      long     name
0    0   m  DARLINGTON  54.533644 -1.524401  FRANCIS
1    0   m  DARLINGTON  54.426256 -1.465314   EDWARD
2    0   m  DARLINGTON  54.555200 -1.496417    TEDDY
3    0   m  DARLINGTON  54.547906 -1.572341    ANGUS
4    0   m  DARLINGTON  54.477639 -1.605995  CHARLIE
age         int64
sex        object
county     object
lat       float64
long      float64
name       object
dtype: object
Index(['age', 'sex', 'county', 'lat', 'long', 'name'], dtype='object')
age  sex  county     lat        long       name
0    f    BLACKPOOL  53.765935  -3.016223  AMAYA       1
55   f    BLACKPOOL  53.812061  -3.019956  AMELIE      1
                     53.811836  -3.070697  WILLOW      1
                     53.811958  -3.010995  ISABELLE    1
                     53.811959  -3.005332  SUSANNA     1
                                                      ..
28   m    BLACKPOOL  53.818232  -3.053205  GABRIEL     1
                     53.818227  -3.007675  JAYDEN      1
                     53.818206  -3.020755  VICTOR      1
                     53.818111  -2.980568  RORY        1
90   m    BLACKPOOL  53.878556  -3.030422  HASSAN      1
Length: 139305, dtype: int64
```

Basic Operations with cuDF

Except for being much more performant with large data sets, cuDF looks and feels a lot like Pandas. In this section we highlight a few very simple operations. When performing data operations on cuDF dataframes, column operations are typically much more performant than row-wise operations.

Converting Data Types

For machine learning later in this workshop, we will sometimes need to convert integer values into floats. Here we convert the `age` column from `int64` to `float32`, comparing performance with Pandas:

cuDF

In [11]:
```
%time gdf['age'] = gdf['age'].astype('float32')
```

```
CPU times: user 5.39 ms, sys: 322 µs, total: 5.71 ms
Wall time: 4.42 ms
```

Pandas

In [12]:
```
%time df['age'] = df['age'].astype('float32')
```

```
CPU times: user 56.9 ms, sys: 156 ms, total: 213 ms
Wall time: 212 ms
```

Column-Wise Aggregations

Similarly, column-wise aggregations take advantage of the GPU's architecture and RAPIDS' memory format.

cuDF

In [13]:
```
%time gdf['age'].mean()
```

```
CPU times: user 711 µs, sys: 7.23 ms, total: 7.94 ms
Wall time: 6.99 ms
```

Out[13]:
```
40.12419336806595
```

Pandas

In [14]:
```
%time df['age'].mean()
```

```
CPU times: user 85.7 ms, sys: 31.9 ms, total: 118 ms
Wall time: 116 ms
```

Out[14]:
```
40.12419
```

String Operations

Although strings are not a datatype traditionally associated with GPUs, cuDF supports powerful accelerated string operations.

cuDF

In [15]:
```
%time gdf['name'] = gdf['name'].str.title()
```

```
CPU times: user 79.3 ms, sys: 4.44 ms, total: 83.7 ms
Wall time: 82.3 ms
```

In [16]:
```
gdf.head()
```

Out[16]:

	age	sex	county	lat	long	name
0	0.0	m	DARLINGTON	54.533644	-1.524401	Francis
1	0.0	m	DARLINGTON	54.426256	-1.465314	Edward
2	0.0	m	DARLINGTON	54.555200	-1.496417	Teddy
3	0.0	m	DARLINGTON	54.547906	-1.572341	Angus
4	0.0	m	DARLINGTON	54.477639	-1.605995	Charlie

Pandas

In [17]:
```
%time df['name'] = df['name'].str.title()
```

```
CPU times: user 17 s, sys: 2.36 s, total: 19.4 s
Wall time: 19.3 s
```

In [18]:
```
df.head()
```

Out[18]:

	age	sex	county	lat	long	name
0	0.0	m	DARLINGTON	54.533644	-1.524401	Francis
1	0.0	m	DARLINGTON	54.426256	-1.465314	Edward
2	0.0	m	DARLINGTON	54.555200	-1.496417	Teddy
3	0.0	m	DARLINGTON	54.547906	-1.572341	Angus
4	0.0	m	DARLINGTON	54.477639	-1.605995	Charlie

Data Subsetting with `loc` and `iloc`

cuDF also supports the core data subsetting tools `loc` (label-based locator) and `iloc` (integer-based locator).

Range Selection

Our data's labels happen to be incrementing numbers, though as with Pandas, `loc` will include every value it is passed whereas `iloc` will give the half-open range (omitting the final value).

In [19]:
```
gdf.loc[100:105]
```

Out[19]:

	age	sex	county	lat	long	name
100	0.0	m	DARLINGTON	54.519527	-1.557723	Samuel
101	0.0	m	DARLINGTON	54.530248	-1.500405	Alden
102	0.0	m	DARLINGTON	54.515970	-1.628573	Samuel
103	0.0	m	DARLINGTON	54.543373	-1.664323	Muhammad
104	0.0	m	DARLINGTON	54.554589	-1.507385	Isaac
105	0.0	m	DARLINGTON	54.487209	-1.541073	Jayden

In [20]:
```
gdf.iloc[100:105]
```

Out[20]:

	age	sex	county	lat	long	name
100	0.0	m	DARLINGTON	54.519527	-1.557723	Samuel
101	0.0	m	DARLINGTON	54.530248	-1.500405	Alden
102	0.0	m	DARLINGTON	54.515970	-1.628573	Samuel
103	0.0	m	DARLINGTON	54.543373	-1.664323	Muhammad
104	0.0	m	DARLINGTON	54.554589	-1.507385	Isaac

`loc` with Boolean Selection

We can use `loc` with boolean selections:

cuDF

In [21]:
```
%time e_names = gdf.loc[gdf['name'].str.startswith('E')]
e_names.head()
```

```
CPU times: user 40.3 ms, sys: 4.06 ms, total: 44.3 ms
Wall time: 43.6 ms
```

Out[21]:

	age	sex	county	lat	long	name
1	0.0	m	DARLINGTON	54.426256	-1.465314	Edward
6	0.0	m	DARLINGTON	54.501872	-1.667874	Eamonn
34	0.0	m	DARLINGTON	54.483065	-1.501312	Ethan
45	0.0	m	DARLINGTON	54.640205	-1.558986	Elvin
49	0.0	m	DARLINGTON	54.575450	-1.600592	Edward

Pandas

In [22]:
```
%time e_names_pd = df.loc[df['name'].str.startswith('E')]
```

```
CPU times: user 18.9 s, sys: 652 ms, total: 19.5 s
Wall time: 19.5 s
```

Combining with NumPy Methods

We can combine cuDF methods with NumPy methods, just like Pandas. Here we use `np.logical_and` for element-wise boolean selection.

cuDF

In [23]:
```
%time ed_names = gdf.loc[np.logical_and(gdf['name'].str.startswith('E'), gdf['name'].s
ed_names.head()
```

```
CPU times: user 25.7 ms, sys: 11.6 ms, total: 37.4 ms
Wall time: 36.3 ms
```

Out[23]:

	age	sex	county	lat	long	name
1	0.0	m	DARLINGTON	54.426256	-1.465314	Edward
49	0.0	m	DARLINGTON	54.575450	-1.600592	Edward
106	0.0	m	DARLINGTON	54.488042	-1.640927	Edward
145	0.0	m	DARLINGTON	54.492810	-1.509049	Edward
170	0.0	m	DARLINGTON	54.577920	-1.436109	Edward

For better performance at scale, we can use CuPy instead of NumPy, thereby performing the element-wise boolean `logical_and` operation on GPU.

In [24]:
```
%time ed_names = gdf.loc[cp.logical_and(gdf['name'].str.startswith('E'), gdf['name'].s
ed_names.head()
```

```
CPU times: user 329 ms, sys: 12.1 ms, total: 341 ms
Wall time: 340 ms
```

Out[24]:

	age	sex	county	lat	long	name
1	0.0	m	DARLINGTON	54.426256	-1.465314	Edward
49	0.0	m	DARLINGTON	54.575450	-1.600592	Edward
106	0.0	m	DARLINGTON	54.488042	-1.640927	Edward
145	0.0	m	DARLINGTON	54.492810	-1.509049	Edward
170	0.0	m	DARLINGTON	54.577920	-1.436109	Edward

Pandas

In [25]:
```
%time ed_names_pd = df.loc[np.logical_and(df['name'].str.startswith('E'), df['name'].s
```

```
CPU times: user 27.5 s, sys: 530 ms, total: 28.1 s
Wall time: 28 s
```

Exercise: Basic Data Cleaning

For this exercise we ask you to perform two simple data cleaning tasks using several of the techniques described above:

1. Modifying the data type of a couple columns
2. Transforming string data into our desired format

1. Modify `dtypes`

Examine the `dtypes` of `gdf` and convert any 64-bit data types to their 32-bit counterparts.

In [27]:
```
for column in gdf.columns:
    if gdf[column].dtype == 'float64':
        gdf[column] = gdf[column].astype('float32')
```

Solution

In [28]:
```
# %load solutions/modify_dtypes
gdf['lat'] = gdf['lat'].astype('float32')
gdf['long'] = gdf['long'].astype('float32')
```

2. Title Case the Counties

As it stands, all of the counties are UPPERCASE:

In [29]:
```
gdf['county'].head()
```

Out[29]:
```
0    DARLINGTON
1    DARLINGTON
2    DARLINGTON
3    DARLINGTON
4    DARLINGTON
Name: county, dtype: object
```

Convert them to title case as we have already done with the `name` column.

In [31]:
```
gdf['county'] = gdf['county'].str.title()
```

Solution

In [32]:
```
# %load solutions/title_case_counties
gdf['county'] = gdf['county'].str.title()
```

Exercise: Counties North of Sunderland

This exercise will require to use the `loc` method, and several of the techniques described above. Identify the latitude of the northernmost resident of Sunderland county (the person with the maximum `lat` value), and then determine which counties have any residents north of this resident. Use the `unique` method of a cudf `Series` to de-duplicate the result.

In [34]:
```
sunderland_residents = gdf.loc[gdf['county'] == 'Sunderland']
northmost_sunderland_lat = sunderland_residents['lat'].max()
counties_with_pop_north_of = gdf.loc[gdf['lat'] > northmost_sunderland_lat]['county'].
```

Solution

In [35]:
```
# %load solutions/counties_north_of_sunderland
sunderland_residents = gdf.loc[gdf['county'] == 'Sunderland']
northmost_sunderland_lat = sunderland_residents['lat'].max()
counties_with_pop_north_of = gdf.loc[gdf['lat'] > northmost_sunderland_lat]['county'].
```

Please Restart the Kernel

In [36]:
```python
import IPython
app = IPython.Application.instance()
app.kernel.do_shutdown(True)
```

Out[36]:
```
{'status': 'ok', 'restart': True}
```

Next

In the next section, you will do some actual data preparation for use in our machine learning models later. As part of your work, we will create custom functions with CuPy, which can be used as a GPU-accelerated drop-in replacement for NumPy with drastic performance benefits.