Visualize the Population

In this notebook we visualize our UK population data with cuXfilter.

Objectives

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

- Use cuXfilter to visualize scatterplot data
- Use a cuXfilter widget to filter subsets of the data

Imports

RAPIDS can be used with a wide array of visualizations, both open source and proprietary. Bokeh, DataShader, and other open source visualization projects are connected with RAPIDS via the cuXfilter framework.

We import `cudf` as usual, plus `cuxfilter`, which we will be using to visualize the UK population data.

As `cuxfilter` loads, we will see empty rows appear underneath this cell; that is expected behavior.

In [ ]:
```
import cudf

import cuxfilter as cxf
```

Load Data

Here we load into a cuDF dataframe the grid coordinates and county from the England/Wales population data, already transformed from your work in the first section of the workshop.

In [ ]:
```
gdf = cudf.read_csv('./data/pop_2-02.csv', usecols=['easting', 'northing'
print(gdf.dtypes)
gdf.shape
```

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

Factorize Counties

cuXfilter widgets enable us to select entries from an integer column. To make that column, we can use cuDF's `factorize` method to convert a string column into an integer column, while keeping a map from the new integers to the old strings.

`factorize` produces the integer column and corresponding map as output, and we overwrite the old string column with the new integer column here. Alternatively, we could have appended the integer column as a new column, but the integer column is much more memory-efficient than the equivalent string column, and so we will often prefer to overwrite.

In [ ]:
```
gdf['county'], county_names = gdf['county'].factorize()
gdf.head()
```

The `county_names` Series is indexed by the integers that `factorize` created. The cuXfilter widget requires a dictionary to map from the integers to the strings, so we convert the index to a list, zip it together with the strings, and make the result into a dictionary. We use the `to_arrow` method to make the gdf series iterable for the Python `list` and `zip` functions.

In [ ]:
```
county_map = dict(zip(list(range(len(county_names))), county_names.to_arr
county_map[37]
```

Visualize Population Density and Distribution

Using cuXfilter has three main steps:

1. Associate a data source with cuXfilter
2. Define the charts and widgets to use with the data
3. Create and show a dashboard containing those charts and widgets

Associate a Data Source with cuXfilter

We need to know what our data source's column names will be for the next step, so it is usually helpful to define the data source first. However, note that the data source is not used in the *Define Charts and Widgets* step below--the same charts and widgets can be used with different data sources with the same column names!

In [ ]:
```
cxf_data = cxf.DataFrame.from_dataframe(gdf)
```

Define Charts and Widgets

We fix the chart `width` and then use the fact that `easting` and `northing` are both in meters to scale the chart `height` appropriately.

We also see how we will use the `county_map` made in the last step: it lets cuXfilter know how to display the selection widget, which is operating on the integer column behind the scenes.

In [ ]:
```
chart_width = 600
scatter_chart = cxf.charts.datashader.scatter(x='easting', y='northing',
                                              width=chart_width,
                                              height=int((gdf['northing']
                                                         (gdf['easting'].
                                                          chart_width))

county_widget = cxf.charts.panel_widgets.multi_select('county', label_map
```

Create and Show the Dashboard

At this point, we provide a list of the elements that we want on the dashboard and can provide it parameters that determine its appearance.

In [ ]:
```
dash = cxf_data.dashboard([scatter_chart, county_widget], theme=cxf.theme
```

Then, we can view individual charts non-interactively as a preview...

In [ ]:
```
scatter_chart.view()
```

And finally, push the whole dashboard up for interactive cross-filtering. To do so, we will need the IP address of the machine we are working on, which you can get by executing the next cell...

In [ ]:
```
%%js
var host = window.location.host;
element.innerText = "'"+host+"'";
```

Set `my_url` in the next cell to the value just printed, making sure to include the quotes.

Note: due to the cloud environment we are working in, you will need to ignore the "open cuxfulter dashboard" button that will appear, and instead, execute the following cell to generate a working link.

In [ ]:
```
my_url = # TODO: Set this value to the print out of the cell above, inclu
dash.show(my_url, port=8789)
```

... and you can run the next cell to generate a link to the dashboard:

In [ ]:
```
%%js
var host = window.location.host;
var url = 'http://'+host+'/lab/proxy/8789/';
element.innerHTML = '<a style="color:blue;" target="_blank" href='+url+'>
```

Finally, once you are done with the dashboard, run the next cell to end it:

In [ ]:
```
dash.stop()
```

Please Restart the Kernel

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

Next

In the next notebook, you will begin your use of GPU-accelerated machine learning algorithms, using K-means to identify the best locations for supply depots and then visualizing the results.