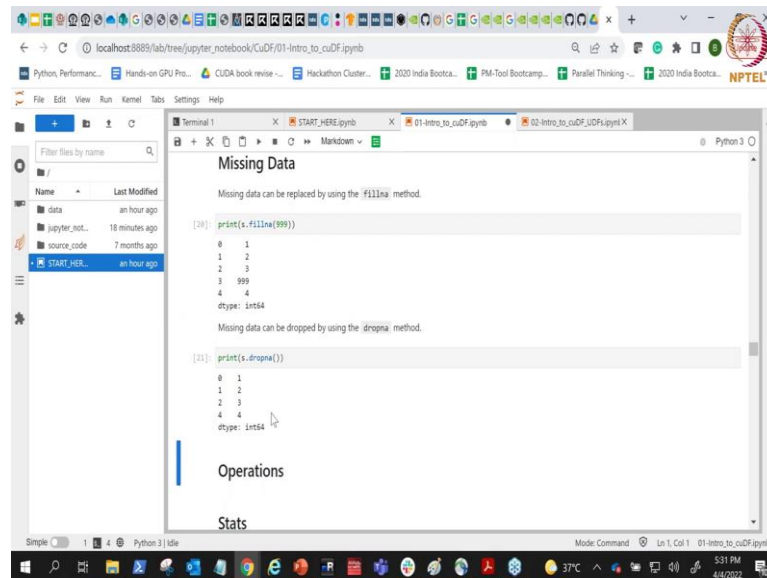


Applied Accelerated Artificial Intelligence
Prof. Bharatkumar Sharma
Department of Computer Science and Engineering
Indian Institute of Technology, Palakkad

Lecture - 47
Accelerated Data Analytics Part 4

(Refer Slide Time: 00:15)



The screenshot shows a Jupyter Notebook interface with a file browser on the left and a code editor on the right. The code editor contains the following text:

```
Missing Data
```

Missing data can be replaced by using the `fillna` method.

```
[10]: print(s.fillna(999))
```

```
0    1
1    2
2    3
3    999
4    4
dtype: int64
```

Missing data can be dropped by using the `dropna` method.

```
[11]: print(s.dropna())
```

```
0    1
1    2
2    3
4    4
dtype: int64
```

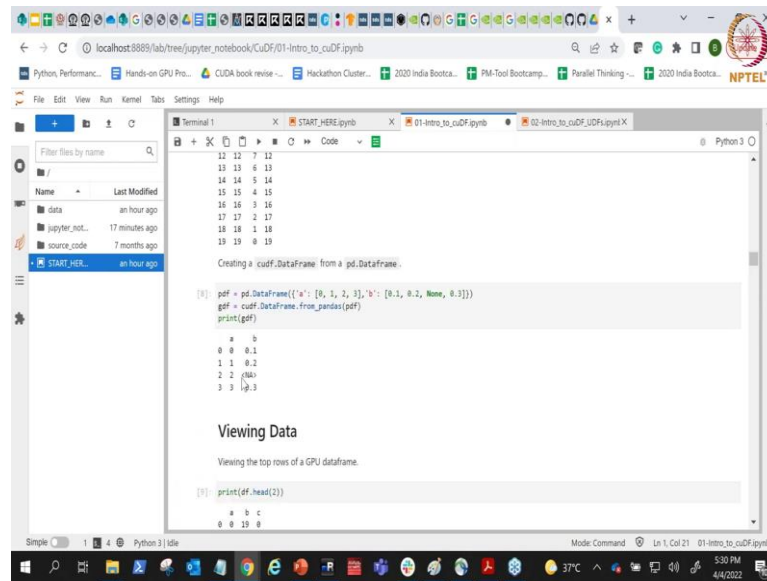
Operations

Stats

So, let us let get started and let us see so the missing data can be replaced by using the fillna method. So, you can see here that there might be a case where some of the data is missing, right. So, if you see in the thing below above there was a column which did not have any data it had not applicable or none.

So, missing data is one of the most key feature which you will see in the real world. In fact, missing data is one of the most critical component which exist, right. So, here in this particular case you would see that there was a missing data here which had which was shown as none previously.

(Refer Slide Time: 01:10)



The screenshot shows a Jupyter Notebook interface with a file browser on the left and a code editor on the right. The code editor contains the following Python code:

```
12 12 7 12
13 13 6 13
14 14 5 14
15 15 4 15
16 16 3 16
17 17 2 17
18 18 1 18
19 19 0 19

Creating a cudf.DataFrame from a pd.DataFrame.

[1]: pdf = pd.DataFrame({'a': [0, 1, 2, 3], 'b': [0.1, 0.2, None, 0.3]})
      gdf = cudf.DataFrame.from_pandas(pdf)
      print(gdf)

a      b
0  0  0.1
1  1  0.2
2  2  (na)
3  3  0.3

Viewing Data

Viewing the top rows of a GPU dataframe.

[2]: print(gdf.head(2))

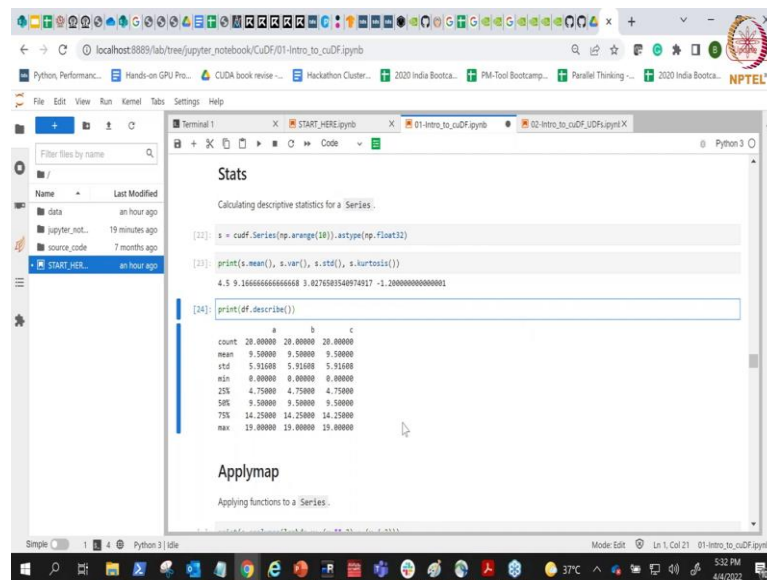
a      b
0  0  0.1
1  1  0.2
```

And let me just show you before you can see here. So, you can see here it was not applicable, because we have put it as none and you can filling those pieces by using the feature which is fill which we just showed it to you fillna.

So, wherever there is not applicable you can fill those values with certain. You can in fact, fill those values with say average of that particular column or average of that particular row or median right; these kinds of techniques are very heavily used by the data scientist or you can drop that column altogether.

So, if you find that particular column you do not need to use, because it will add some bias to your overall training then you can drop all of the not applicable columns.

(Refer Slide Time: 01:53)



The screenshot shows a Jupyter Notebook with the following content:

```
Calculating descriptive statistics for a Series.
```

```
[22]: s = cudf.Series(np.arange(10).astype(np.float32))
```

```
[23]: print(s.mean(), s.var(), s.std(), s.kurtosis())
```

```
4.5 9.166666666666668 3.027683540974917 -1.2000000000000001
```

```
[24]: print(s.describe())
```

```
count    10.000000    10.000000    10.000000
mean      4.500000     9.166667     3.027684
std       1.750000     1.750000     1.750000
min       0.000000     0.000000     0.000000
25%       1.250000     1.250000     1.250000
50%       4.500000     4.500000     4.500000
75%      14.250000    14.250000    14.250000
max      19.000000    19.000000    19.000000
```

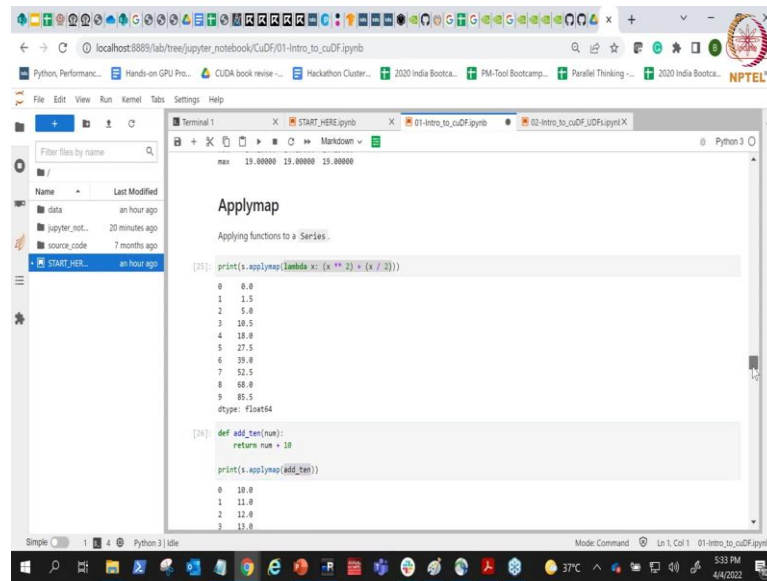
Below the output, the 'Applymap' section is visible, with the text 'Applying functions to a Series.'

So, from a stats perspective again you can calculate different kinds of statistics for a series, like you can see here I have a np arrange and it up to 10 and it is a type float and I am creating a new columnar set by calculating the mean variation standard deviation and you can add many many more functionalities like that.

So, you can see here that if I had a series of data and so I can create the mean of it, the variation standard deviation and all and in fact, if you have a particular column like this you can basically just say describe and that describe basically gives you all the characteristics of about your columnar data like, how many values are there right for a, b and c, what is the mean? What is the standard deviation? What is the minimum value? Right.

And where is 25 percent of the data 50 percent 75 percent what is the maximum value as well. So, you can use the describe feature where it will give you a lot of statistics about the overall table that you have.

(Refer Slide Time: 03:06)



```
max 19.00000 19.00000 19.00000

Applymap

Applying functions to a Series.

[25]: print(s.applymap(lambda x: (x**2) * (x/2)))

0 0.0
1 1.5
2 5.0
3 18.0
4 18.0
5 27.5
6 39.0
7 52.5
8 68.0
9 85.5
dtype: float64

[26]: def add_ten(num):
      return num + 10

      print(s.applymap(add_ten))

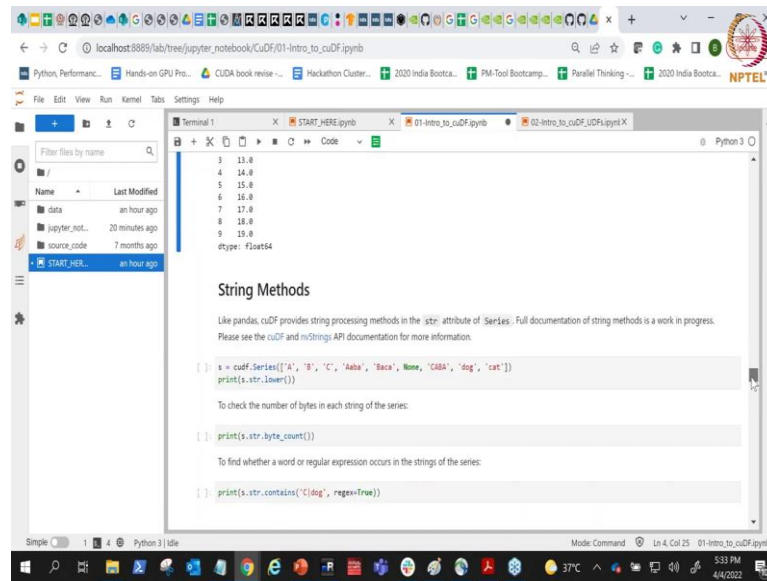
0 10.0
1 11.0
2 12.0
3 13.0
```

In fact, to all of them you can use a particular feature which is `s.applymap`. This `applymap` will basically apply this particular formula which is if you are a programmer in Python you would know how to write lambda functions.

So, what it is doing is that, it is basically doing this operation where I am applying to each and every row the this particular calculation. So, if I just apply the `map`. So, it will you will see that it has taken and it has basically applied this which is $x^2 + x/2$.

So, it takes that value and apply those element to them and then you can see here you in fact rather than writing a lambda function, you can define your own function like here saying that I want to add `n` to all of the numbers and I can say `s.applymap` which means I am going to apply this to every row a particular function which is `add 10`.

(Refer Slide Time: 04:09)



The screenshot shows a Jupyter Notebook interface with a file explorer on the left and a code editor on the right. The file explorer shows a directory structure with files like 'data', 'jupyter_notebook', 'source_code', and 'START_HERE.ipynb'. The code editor displays a terminal window with the following output:

```
3 13.0
4 14.0
5 15.0
6 16.0
7 17.0
8 18.0
9 19.0
dtype: float64
```

The code editor also shows a section titled 'String Methods' with a description: 'Like pandas, cuDF provides string processing methods in the `str` attribute of `Series`. Full documentation of string methods is a work in progress. Please see the `cuDF` and `mStrings` API documentation for more information.'

The code in the editor is:

```
[ ] s = cudf.Series(['A', 'B', 'C', 'Aaba', 'Baba', None, 'CABA', 'dog', 'cat'])
print(s.str.lower())
```

Below the code, there are instructions for other string methods:

To check the number of bytes in each string of the series:

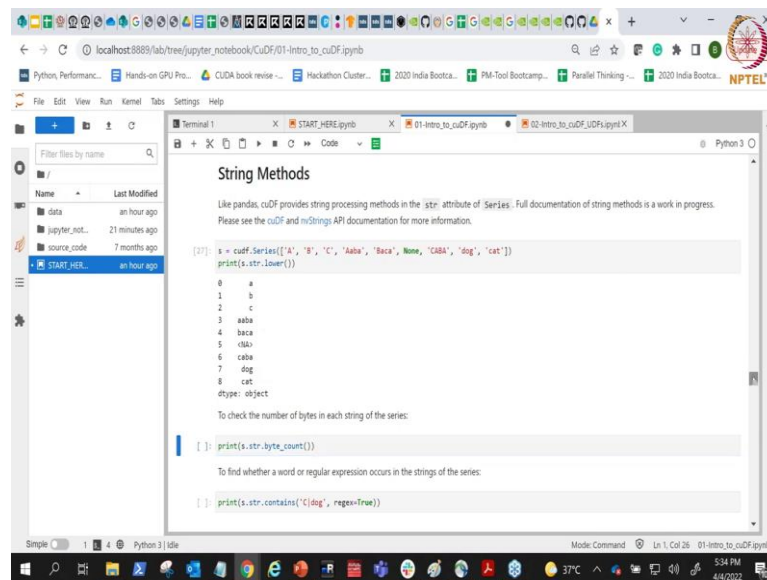
```
[ ] print(s.str.byte_count())
```

To find whether a word or regular expression occurs in the strings of the series:

```
[ ] print(s.str.contains('Cdog', regex=True))
```

And then, it will apply those and you can see here that for every element it has basically added 10 to the original data frame.

(Refer Slide Time: 04:21)



The screenshot shows the same Jupyter Notebook interface as before, but the code editor now displays the following output:

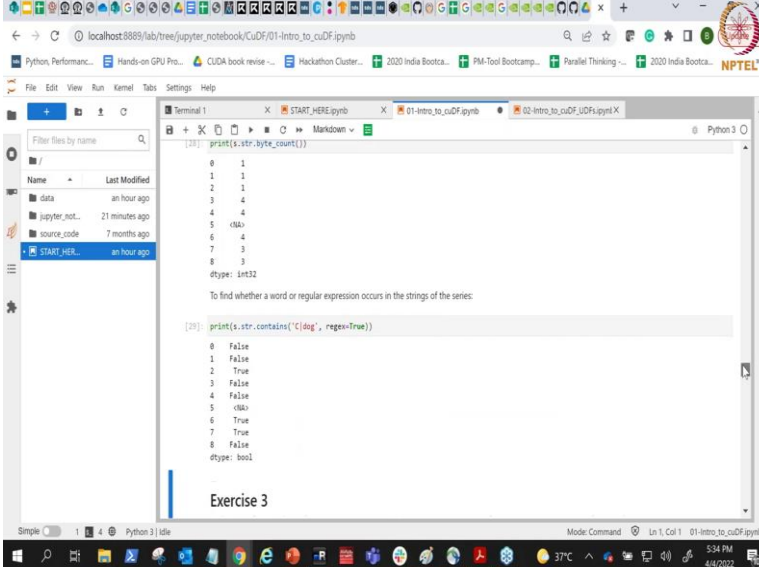
```
[27] s = cudf.Series(['A', 'B', 'C', 'Aaba', 'Baba', None, 'CABA', 'dog', 'cat'])
print(s.str.lower())
0 a
1 b
2 c
3 aaba
4 baba
5 caba
6 caba
7 dog
8 cat
dtype: object
```

The code editor also shows the same 'String Methods' section as before, with the same code and instructions.

So, like pandas cuDF provides string processing methods also and we are going to look to certain extent the string part you can see here cudf series A B C certain elements again there is a none here and if I just run this you can see here you have print s dot str lower.

So, what it does is kind of already known that we are taking the values and we are converting them into lower strings right or if you want to count the bytes of each and every string sometimes you may go out of bound in terms of the memory.

(Refer Slide Time: 04:52)



The screenshot shows a Jupyter Notebook with a file browser on the left and a code editor on the right. The code editor contains the following Python code:

```
[1]: print(s.str.byte_count())  
0    1  
1    1  
2    1  
3    4  
4    4  
5    <NA>  
6    4  
7    3  
8    3  
dtype: int32  
  
To find whether a word or regular expression occurs in the strings of the series:  
  
[2]: print(s.str.contains('Cdog', regex=True))  
0    False  
1    False  
2     True  
3    False  
4    False  
5     <NA>  
6     True  
7     True  
8    False  
dtype: bool
```

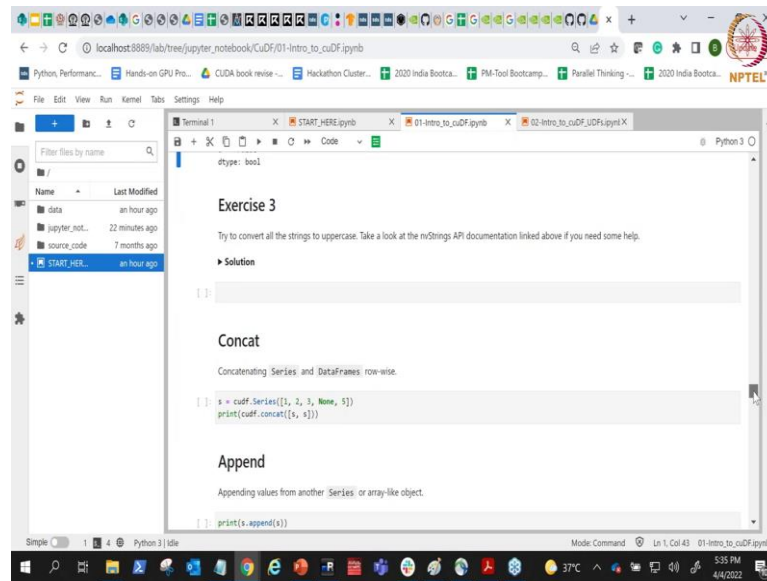
Below the code, the text "Exercise 3" is visible. The Jupyter Notebook interface includes a menu bar (File, Edit, View, Run, Kernel, Help) and a status bar at the bottom showing the current mode (Simple, Command) and the file name (01-Intro_to_cuDf.ipynb).

So, you can in fact count the number of bytes which were used for storing that particular string if required. You can write different kinds of regular expressions, like you can say see if my string contains this particular regex or not.

So, it should either contain this or that and it will only print the values where that particular thing is said. So, you can see here in the first code it is saying false, because there is neither C nor dog and you will see that for the 3rd one it is basically true because, there is a C there, right. So, it consist of C and then there is a dog at the 7th location which is also being set to true.

So, so if the dog value is also set to true which is the 7th row while there is a C as well in the 6th column and that is why even 6 has been marked as true. So, you can do different kinds of regular expressions using the data frame there, ok.

(Refer Slide Time: 05:56)



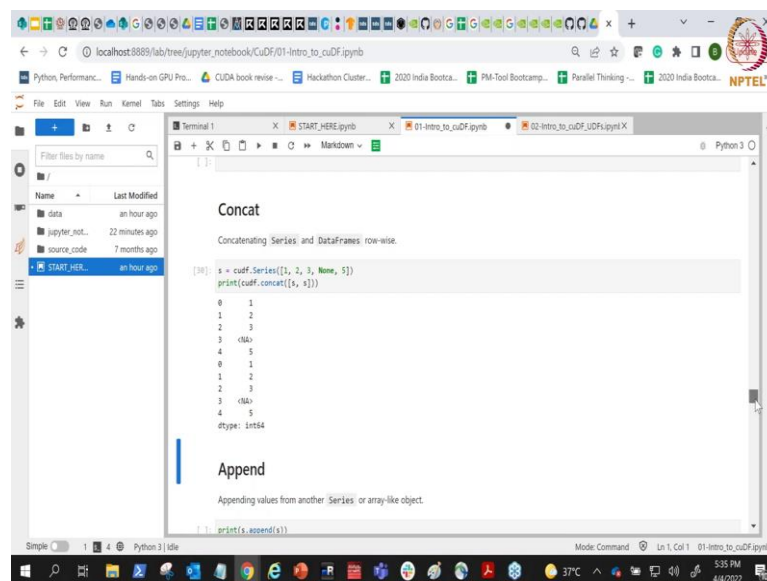
The screenshot shows a Jupyter Notebook titled '01-Intro_to_cuDF.ipynb'. The left sidebar displays a file explorer with folders 'data', 'jupyter_note...', 'source_code', and a file 'START_HERE...'. The main area shows 'Exercise 3' with the instruction: 'Try to convert all the strings to uppercase. Take a look at the mStrings API documentation linked above if you need some help.' Below this, the 'Concat' section explains 'Concatenating Series and DataFrames row-wise' and provides a code snippet:

```
s = cudf.Series([1, 2, 3, None, 5])
print(cudf.concat([s, s]))
```

 The 'Append' section explains 'Appending values from another Series or array-like object' and provides a code snippet:

```
print(s.append(s))
```

(Refer Slide Time: 06:01)



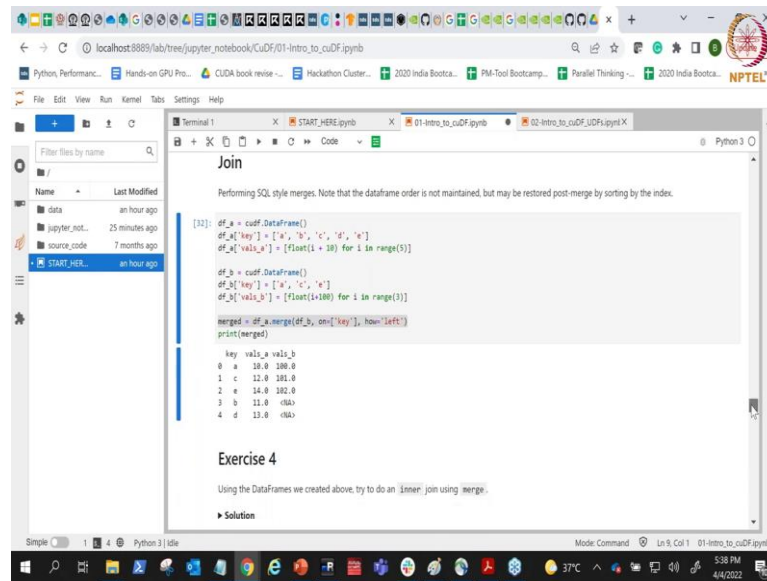
This screenshot shows the same Jupyter Notebook after execution. The 'Concat' section now displays the output of the code:

```
[0] 1
1 2
2 3
3 <NA>
4 5
0 1
1 2
2 3
3 <NA>
4 5
dtype: int64
```

 The 'Append' section remains visible below.

So, this is kind of easy one. So, I am going to skip that part and obviously, you can do things like concatenation. You can concatenate, so here you can see here I have a cudf series 1 2 3 and then a not applicable and 5 and then I am just concatenating it again with itself.

(Refer Slide Time: 06:26)



```
Join

Performing SQL style merges. Note that the dataframe order is not maintained, but may be restored post-merge by sorting by the index.

In [3]: df_a = cudf.DataFrame()
df_a['key'] = ['a', 'b', 'c', 'd', 'e']
df_a['val_a'] = [float(i + 10) for i in range(5)]

df_b = cudf.DataFrame()
df_b['key'] = ['a', 'c', 'e']
df_b['val_b'] = [float(i+100) for i in range(3)]

merged = df_a.merge(df_b, on='key', how='left')
print(merged)

key  val_a  val_b
0  a    10.0   100.0
1  c    12.0   102.0
2  e    14.0   104.0
3  b    11.0    NaN
4  d    13.0    NaN

Exercise 4

Using the DataFrames we created above, try to do an 'inner' join using 'merge'.

Solution
```

So, you can see here 1 2 3 4 5 and again 1 2 3 4 5. So, you can concatenate to it is self or you can append in the end if you would like to which is almost the same that we did later on as well. So, this is another example where we are trying to do a join operation so, it is more of like a SQL syntax. So, if you are a database person, you would understand the SQL like queries and but the main thing is because everything runs in parallel the order is not maintained.

So, you do not get guarantee like if you do anything in the SQL database or your columns the rows are kind of the order is maintained. But, here whenever you fire a SQL like query the orders are not maintained because, they are basically they how should I say they are basically happening in parallel on the GPU, but if you want you can restore them post also.

So, here you can see here we have created a data frame you have a key and value pair and then similarly you have another data frame b we have again a key and value pair and then, we are trying to tell to merge them. So, we are saying df a dot merge.

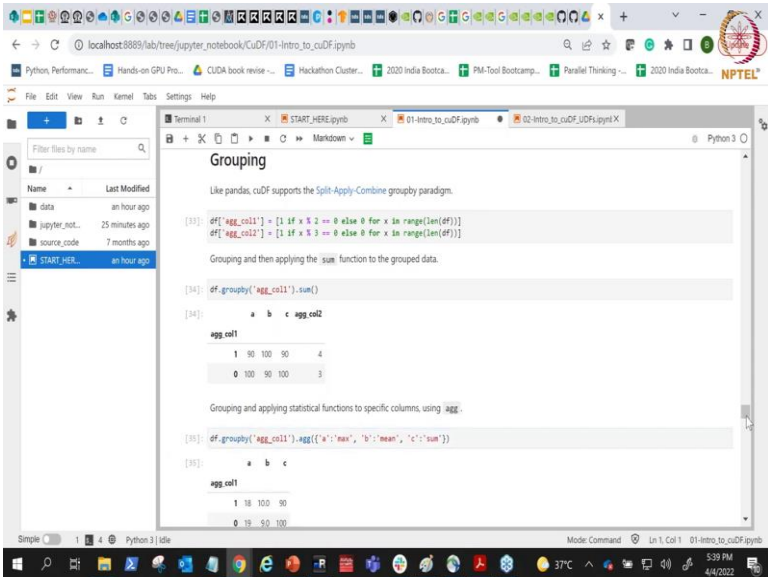
So, merge is one of the most common operation that you find in your database and we are seeing it how to merge. So, we are defining the strategy how to merge it. Whenever there is a necessity of merging should I add it to left or should I do a left merge or should I do a right merge.

So, if you are familiar to SQL database you would understand some of the points which I am trying to mention here you can see here you have a here and you have a here as well. So, you have the value 10 and you have the value 100 here.

So, we have done a left merge based on the key value and that is how it will work and you can see here if there is no value of certain components here, like there was no value for b here common and you will see that b have been populated with not applicable.

So, only at the places where you got the values is kind of in the beginning and you can see here that the order was not maintained. In the beginning the order was a b c d e, but now you see a c e b d which is slightly a different order ah, but it is done something called as a left merge for you.

(Refer Slide Time: 08:51)



```
Like pandas, cuDF supports the Split-Apply-Combine groupby paradigm.

[33]: df['agg_col1'] = [1 if x % 2 == 0 else 0 for x in range(len(df))]
      df['agg_col2'] = [1 if x % 3 == 0 else 0 for x in range(len(df))]

Grouping and then applying the .sum() function to the grouped data.

[34]: df.groupby('agg_col1').sum()

[34]:
```

| | a | b | c | agg_col2 |
|----------|-----|-----|-----|----------|
| agg_col1 | | | | |
| 1 | 90 | 100 | 90 | 4 |
| 0 | 100 | 90 | 100 | 3 |

```
Grouping and applying statistical functions to specific columns, using .agg().

[35]: df.groupby('agg_col1').agg(['a': 'max', 'b': 'mean', 'c': 'sum'])

[35]:
```

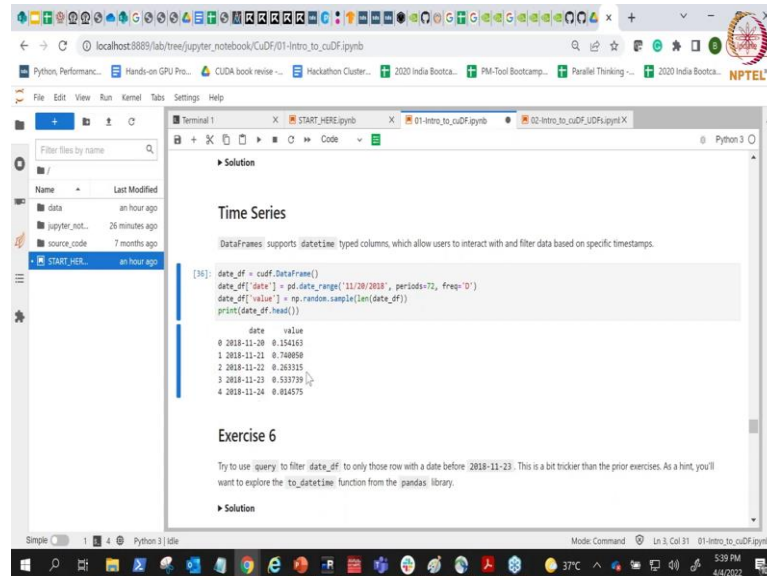
| | a | b | c |
|----------|----|-----|-----|
| agg_col1 | | | |
| 1 | 18 | 100 | 90 |
| 0 | 18 | 50 | 100 |

So, moving to the next one maybe so like pandas cuDF support split so this is another example of grouping. Hence, here you have two columns here aggregate column 1, aggregate column 2 and we are saying set it to 1 if $x \% 2 = 0$, which means if it is divisible by 2 then set it to 1 else set it to 0, right. So, that is what this particular logic actually means if it is divisible by 2 set it to 1 and set it to 0 and if you want you can do a summation of it.

So, if so ideally yes for all of them you will get a summation of how many values are aggregated column how many values are 1 and how many values are 0 here, right. So,

there are different kinds of exercises that you can do this is another example of time series data.

(Refer Slide Time: 09:40)



The screenshot shows a Jupyter Notebook interface with a file explorer on the left and a code editor on the right. The file explorer shows a directory structure with files like 'data', 'jupyter_notebook', 'source_code', and 'START_HERE'. The code editor displays a notebook titled 'Time Series' with a solution section. The solution section contains a code cell with the following code:

```
[36]: data_df = cudf.DataFrame()
      date_df['date'] = pd.date_range('11/20/2018', periods=72, freq='D')
      date_df['value'] = np.random.sample(len(date_df))
      print(date_df.head())
```

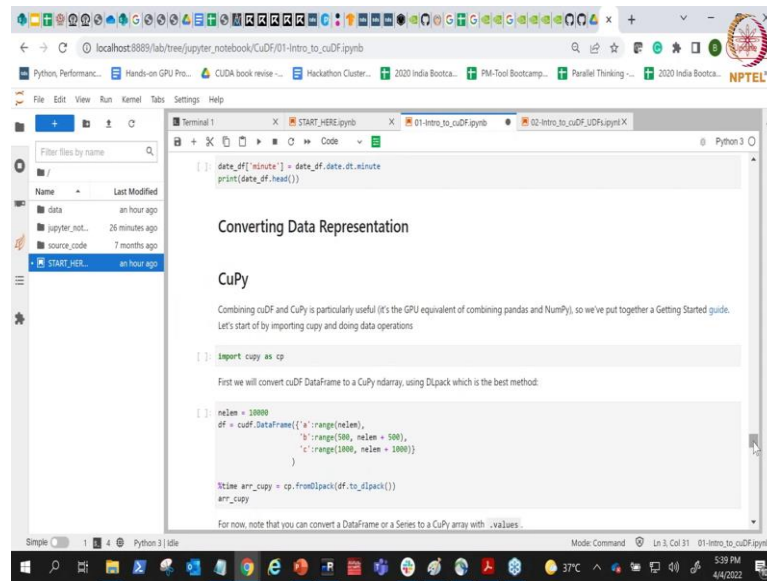
The output of the code is a DataFrame with two columns: 'date' and 'value'. The 'date' column contains dates from 2018-11-20 to 2018-11-24, and the 'value' column contains random values. Below the code cell, there is an 'Exercise 6' section with a hint to use the 'query' method to filter the data.

| | date | value |
|---|------------|----------|
| 0 | 2018-11-20 | 0.154163 |
| 1 | 2018-11-21 | 0.748608 |
| 2 | 2018-11-22 | 0.203335 |
| 3 | 2018-11-23 | 0.533739 |
| 4 | 2018-11-24 | 0.814575 |

So, here you can see a data frame supports datetime. So, datetime is another very common component used in the data sets and you can see here we are using `pd.date_range`.

So, this is the data range. I am setting starting from this particular date; the period is kind of fine the frequency is defined and if I print the value you will see it. So, this is the date and you can see here we defined the range with a period of 72 and it keeps on moving forward here. So, you can define different kinds of dates as well, ok.

(Refer Slide Time: 10:24)



```
[ ]: date_diff["minute"] = date_diff.dt.minute
print(date_diff.head())
```

Converting Data Representation

CuPy

Combining cuDF and CuPy is particularly useful (it's the GPU equivalent of combining pandas and NumPy), so we've put together a Getting Started guide. Let's start by importing cupy and doing data operations

```
[ ]: import cupy as cp

First we will convert cuDF DataFrame to a CuPy ndarray, using DLPack which is the best method.
```

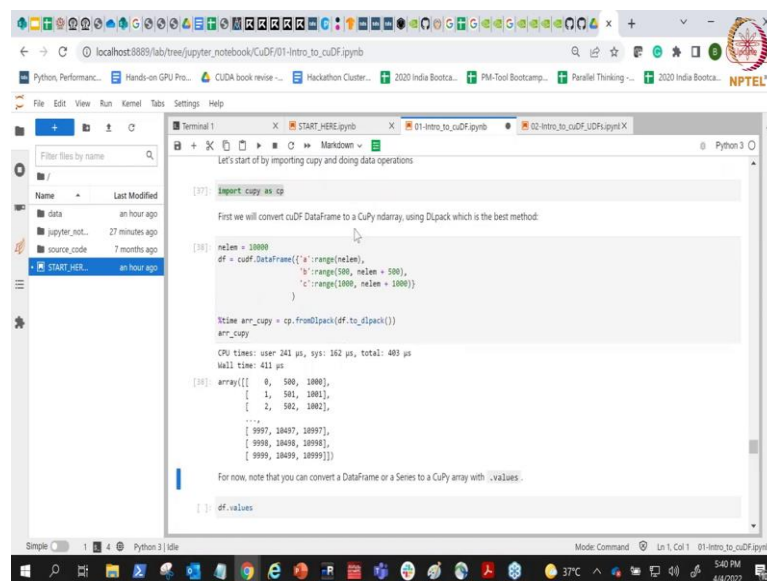
```
[ ]: nsize = 10000
df = cudf.DataFrame({'a':range(nsize),
                      'b':range(500, nsize + 500),
                      'c':range(1000, nsize + 1000)})

%time arr_cupy = cp.fromDLPack(df.to_dlpack())
arr_cupy
```

For now, note that you can convert a DataFrame or a Series to a CuPy array with `.values`.

So, I think I kind of covered a large portion of what all functionalities are provided here.

(Refer Slide Time: 10:28)



```
[ ]: import cupy as cp

First we will convert cuDF DataFrame to a CuPy ndarray, using DLPack which is the best method.
```

```
[ ]: nsize = 10000
df = cudf.DataFrame({'a':range(nsize),
                      'b':range(500, nsize + 500),
                      'c':range(1000, nsize + 1000)})

%time arr_cupy = cp.fromDLPack(df.to_dlpack())
arr_cupy
```

CPU times: user 241 µs, sys: 162 µs, total: 403 µs
Wall time: 411 µs

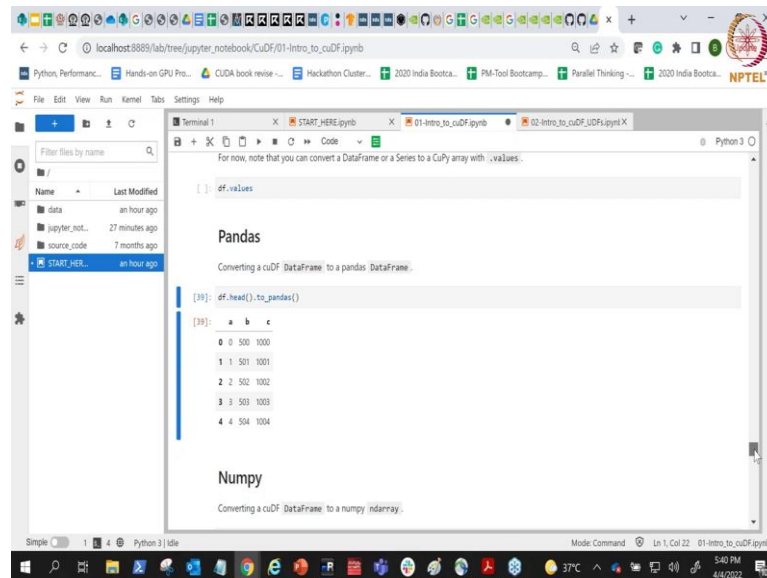
```
[ ]: array([[ 0, 500, 1000],
         [ 1, 501, 1001],
         [ 2, 502, 1002],
         ...,
         [9997, 10497, 10997],
         [9998, 10498, 10998],
         [9999, 10499, 10999]])
```

For now, note that you can convert a DataFrame or a Series to a CuPy array with `.values`.

```
[ ]: df.values
```

As I told you there are other kinds of features also which exist like you can if you we had covered to certain extent prior something called as CuPy. So, in case you are writing your own GPU code using a package called as CuPy, you can basically convert your frames between CuPy to a data frame from so, CuPy has this concept of ndarray n dimensional array.

(Refer Slide Time: 11:02)



The screenshot shows a Jupyter Notebook with a file explorer on the left and a code editor on the right. The code editor contains the following code:

```
df.values
```

Below the code, there is a section titled "Pandas" with the text "Converting a cuDF DataFrame to a pandas DataFrame." and a code cell with the following code:

```
[38]: df.head().to_pandas()
```

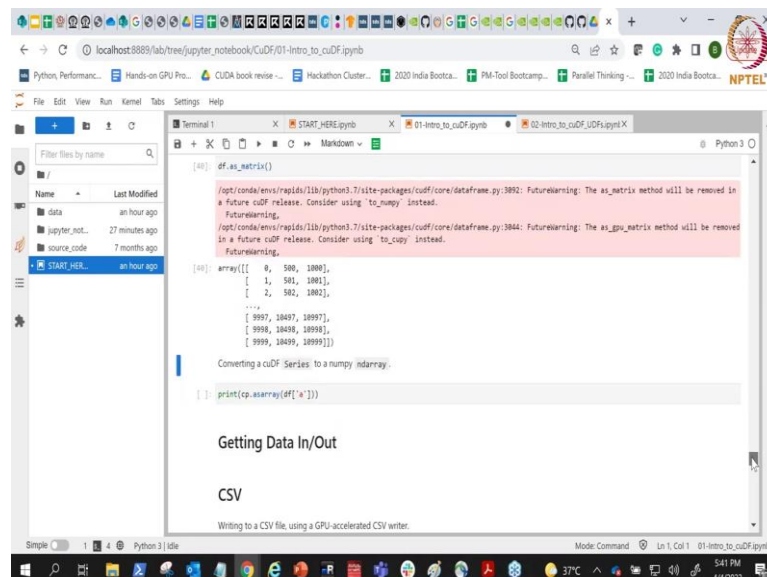
The output of the code is a pandas DataFrame with 5 rows and 3 columns:

| | a | b | c |
|---|---|-----|------|
| 0 | 0 | 500 | 1000 |
| 1 | 1 | 501 | 1001 |
| 2 | 2 | 502 | 1002 |
| 3 | 3 | 503 | 1003 |
| 4 | 4 | 504 | 1004 |

Below the output, there is a section titled "Numpy" with the text "Converting a cuDF DataFrame to a numpy ndarray."

So, you can do those kinds of conversions if it is required or from pandas we have already told you about it. So, you can convert from pandas to cuDF or from cuDF we can convert it back to pandas. So, that you can utilize it on the CPU, say for different purpose like writing it to a file or something.

(Refer Slide Time: 11:20)



The screenshot shows a Jupyter Notebook with a file explorer on the left and a code editor on the right. The code editor contains the following code:

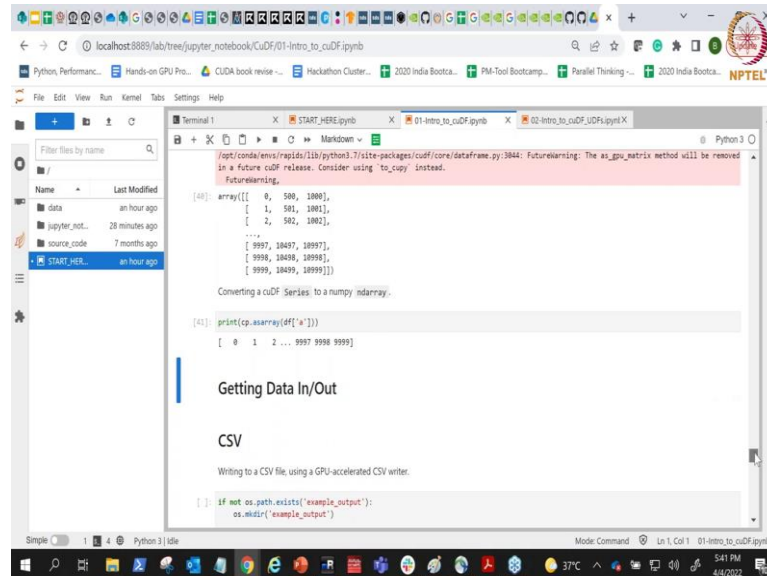
```
df.as_matrix()
```

Below the code, there is a section titled "Getting Data In/Out" with a sub-section titled "CSV" and the text "Writing to a CSV file, using a GPU-accelerated CSV writer."

Same thing exist for numpy in case you want to convert a cuDF dataframe to a numpy arrays. You can say here you can say df dot as matrix and it will basically do the same

thing. So, there might be certain deprecated warning but it is fine, but you can convert them as an array.

(Refer Slide Time: 11:39)



The screenshot shows a Jupyter Notebook with a file browser on the left and a code editor on the right. The code in the editor is as follows:

```
FutureWarning: The as_gpu_matrix method will be removed in a future cuDF release. Consider using 'to_cupy' instead.

array([[ 0, 500, 1000],
       [ 1, 501, 1001],
       [ 2, 502, 1002],
       [9997, 10497, 10997],
       [9998, 10498, 10998],
       [9999, 10499, 10999]])

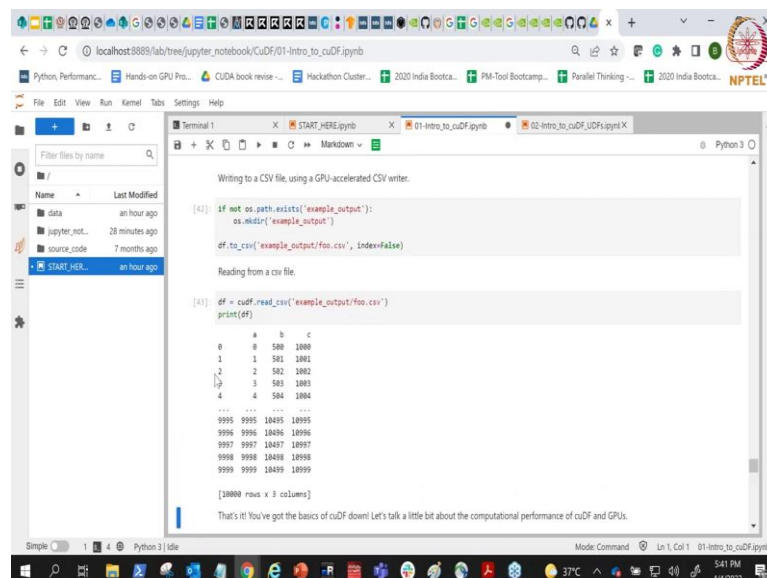
Converting a cuDF Series to a numpy ndarray.

[41]: print(cp.asarray(df['a']))
[ 0  1  2 ... 9997 9998 9999]
```

Below the code, there is a section titled "Getting Data In/Out" with a sub-section "CSV" that says "Writing to a CSV file, using a GPU-accelerated CSV writer." and shows a code snippet for writing to a CSV file.

So, you can see here you can convert them as an array from the data frame as well. So, that kind of a functionality is also provided.

(Refer Slide Time: 11:49)



The screenshot shows a Jupyter Notebook with a file browser on the left and a code editor on the right. The code in the editor is as follows:

```
Writing to a CSV file, using a GPU-accelerated CSV writer.

[42]: if not os.path.exists('example_output'):
      os.mkdir('example_output')

      df.to_csv('example_output/foo.csv', index=False)

Reading from a csv file.

[43]: df = cudf.read_csv('example_output/foo.csv')
      print(df)

   a    b    c
0  0  500 1000
1  1  501 1001
2  2  502 1002
3  3  503 1003
4  4  504 1004
...
9995 9995 10495 10995
9996 9996 10496 10996
9997 9997 10497 10997
9998 9998 10498 10998
9999 9999 10499 10999

[10000 rows x 3 columns]

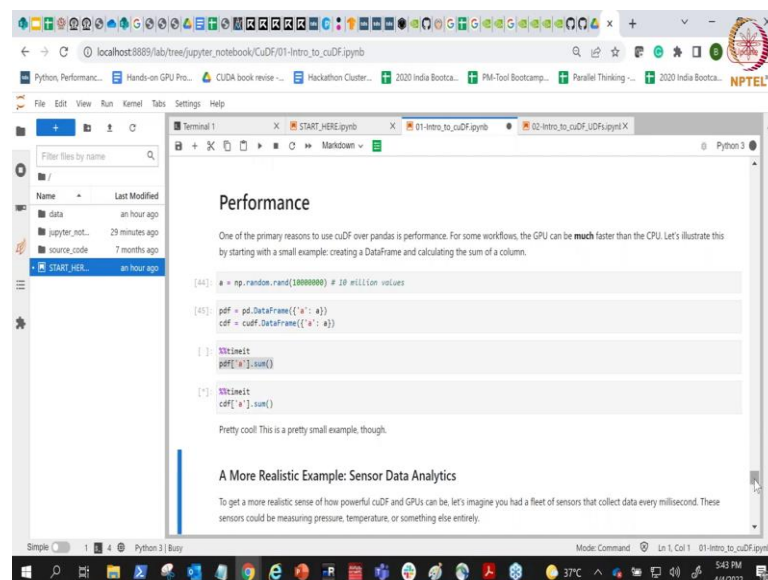
That's it! You've got the basics of cuDF down! Let's talk a little bit about the computational performance of cuDF and GPUs.
```

We also talked about reading or writing from a file you can see here that we are reading from a CSV file here. So, it is quite easy if it exist. So, we are saying data frame dot to CSV. So, here we are actually writing to a file.

So, we are converting it and writing into a file or if you want again whatever we have written in form of a file the data frame we can read it back and it should ideally print the values that we had on our data frame of a, b and c. We can see here it is just printing the top values and then you see dot dot dot.

Because, there are so many values it will not print all the values. So, the print will not print all the values, it will only print certain head values and in the end whatever values are there, but it will print the overall things it says. You have 10000 rows inside it out of which I am kind of presenting you only a few of them.

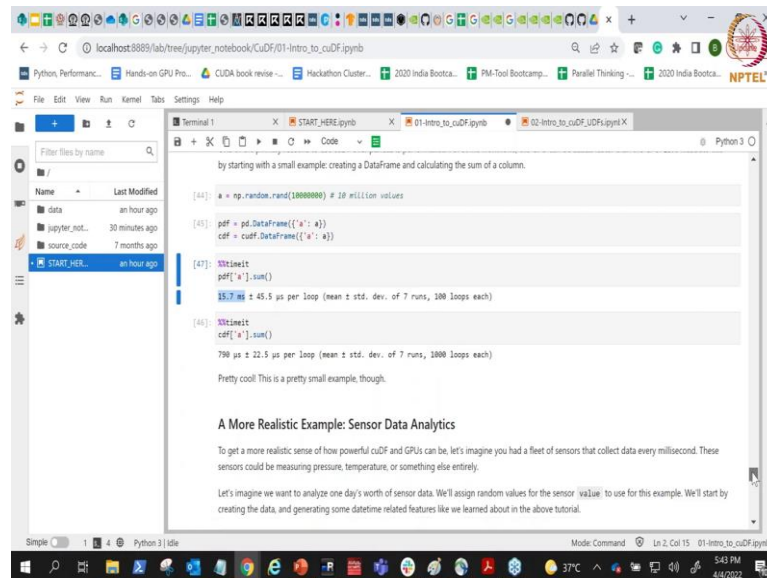
(Refer Slide Time: 12:45)



But, one of the critical reasons why I told so far I have been telling you about how to use cuDF, but what is missing so far is the part of a performance. I just told you that it is used for performance; so far we were reading 1000 rows 10000 rows and all. But, cuDF is all about performance and if only if you are doing certain complicated operations on it or if you have a large data that is where it will make sense.

And here as you can see here first we are going to run a numpy based thing pandas based thing. So, so we have created so many 10 million values and those 10 million values we are populating in the pandas data frame as well as in the CUDA Data Frame and we are going to do a some operation on pandas as well as on cudf and we are printing the time for it. So, let us see how much time it takes.

(Refer Slide Time: 13:43)



The screenshot shows a Jupyter Notebook interface with a file explorer on the left and a code editor on the right. The code in the editor is as follows:

```
by starting with a small example: creating a DataFrame and calculating the sum of a column.

[44]: a = np.random.rand(10000000) # 10 million values
[45]: pdf = pd.DataFrame({'v': a})
[46]: cdf = cuf.DataFrame({'v': a})

[47]: %timeit
pdf['v'].sum()
15.7 ms ± 45.5 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)

[48]: %timeit
cdf['v'].sum()
790 µs ± 22.5 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)

Pretty cool! This is a pretty small example, though.

A More Realistic Example: Sensor Data Analytics

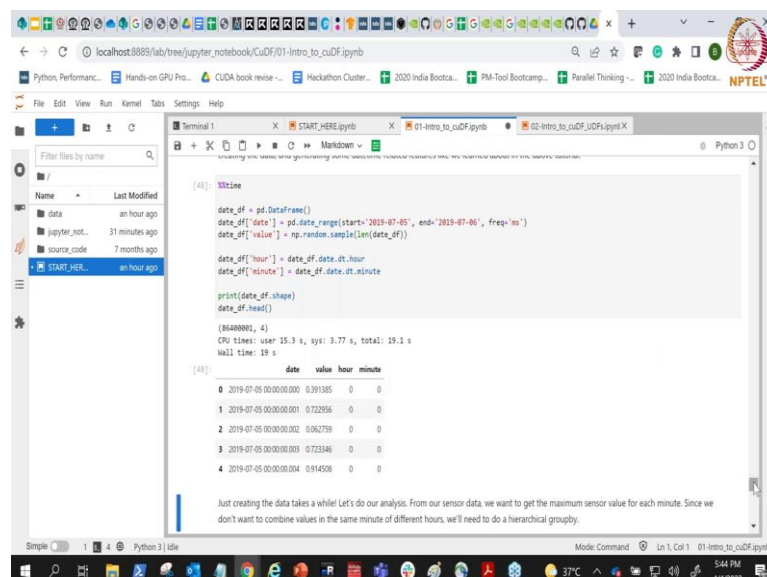
To get a more realistic sense of how powerful cuDF and GPUs can be, let's imagine you had a fleet of sensors that collect data every millisecond. These sensors could be measuring pressure, temperature, or something else entirely.

Let's imagine we want to analyze one day's worth of sensor data. We'll assign random values for the sensor 'value' to use for this example. We'll start by creating the data, and generating some datetime related features like we learned about in the above tutorial.
```

The output shows that the CPU sum took 15.7 ms and the GPU sum took 790 µs.

So, you can see here it has taken 790 microseconds on the GPU and on the CPU let us see how much time it takes for calculating a sum of 10 million elements. So, 790 microseconds on the GPU and it has taken 15 milliseconds on the CPU. So, you can already see the amount of timing difference when I try to add only just one column, it is it is really good right but, it is still a simple example.

(Refer Slide Time: 14:23)



The screenshot shows a Jupyter Notebook interface with a file explorer on the left and a code editor on the right. The code in the editor is as follows:

```
[40]: %time
date_df = pd.DataFrame()
date_df['date'] = pd.date_range(start='2019-07-05', end='2019-07-06', freq='ms')
date_df['value'] = np.random.sample(len(date_df))

date_df['hour'] = date_df.date.dt.hour
date_df['minute'] = date_df.date.dt.minute

print(date_df.shape)
date_df.head()

(86400000, 4)
CPU times: user 15.3 s, sys: 3.77 s, total: 19.1 s
Wall time: 19 s

[41]:
```

| | date | value | hour | minute |
|---|-------------------------|----------|------|--------|
| 0 | 2019-07-05 00:00:00.000 | 0.391385 | 0 | 0 |
| 1 | 2019-07-05 00:00:00.001 | 0.722956 | 0 | 0 |
| 2 | 2019-07-05 00:00:00.002 | 0.062759 | 0 | 0 |
| 3 | 2019-07-05 00:00:00.003 | 0.723346 | 0 | 0 |
| 4 | 2019-07-05 00:00:00.004 | 0.914508 | 0 | 0 |

Just creating the data takes a while! Let's do our analysis. From our sensor data, we want to get the maximum sensor value for each minute. Since we don't want to combine values in the same minute of different hours, we'll need to do a hierarchical groupby.

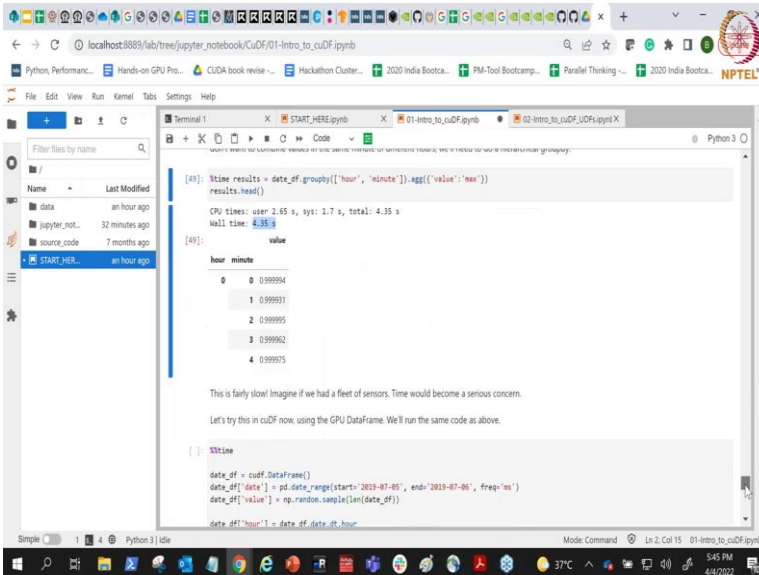
So, let me now show you a more realistic example of sensor data analytics. So, what we are trying to do here is that we are trying to mimic as if you are getting data from various

sensors and it is getting from a series of sensors at every milliseconds and this sensors are measuring say temperature pressure or something, right.

So, what we are doing is, we have a pandas data frame we are creating a date range it starts at this particular date it ends at this date and the frequency is in milliseconds, right. So, this is going to create a date range and the same we are creating the date the values it is kind of a sample values and then, we are going to populate in hours and minutes also.

So, let us create them in the pandas side of it and then, what we are saying is that we are going to just do certain analysis on it like here in this case we are just saying I want to group by the based on hours and minutes I want to do a aggregation and the aggregation type is of max. So, I want to find the max, right.

(Refer Slide Time: 15:29)



The screenshot shows a Jupyter Notebook interface with a file browser on the left and a code editor on the right. The code editor contains the following Python code:

```
[49]: time results = date_df.groupby(['hour', 'minute']).agg({'value': 'max'})
      results.head()
```

The output of the code is displayed below the code cell:

```
CPU times: user 2.65 s, sys: 1.7 s, total: 4.35 s
Wall time: 4.35 s
```

| hour | minute | value |
|------|--------|----------|
| 0 | 0 | 0.999994 |
| 1 | 0 | 0.999993 |
| 2 | 0 | 0.999995 |
| 3 | 0 | 0.999962 |
| 4 | 0 | 0.999975 |

Below the output, there is a text block that reads:

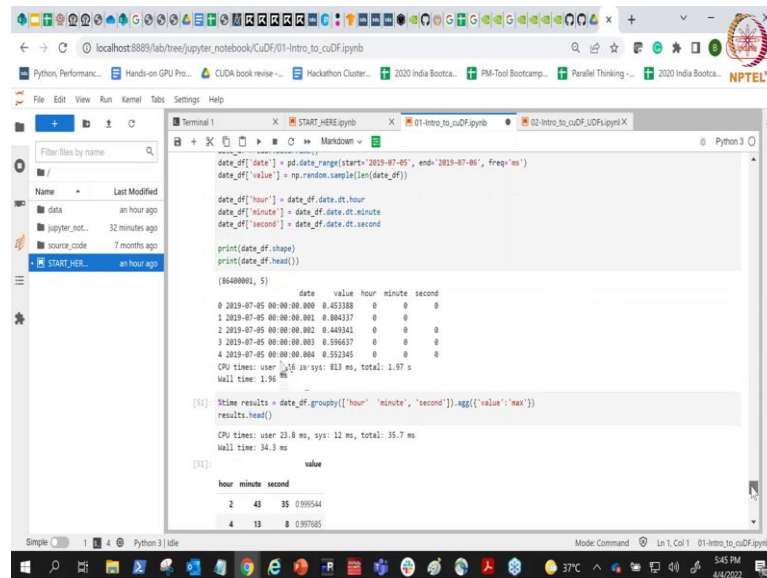
This is fairly slow! Imagine if we had a fleet of sensors. Time would become a serious concern. Let's try this in cuDF now, using the GPU DataFrame. We'll run the same code as above.

The code cell is followed by a new code cell that starts with the following code:

```
[50]: %time
      date_df = cudf.DataFrame()
      date_df['date'] = pd.date_range(start='2019-07-05', end='2019-07-06', freq='ms')
      date_df['value'] = np.random.sample(len(date_df))
      date_df['hour'] = date_df.date.dt.hour
```

So, I want to do aggregation and I want to do a max value of it. So, you can see here creation took around 19 seconds and then I am going to do a performer operation on it. So, I am going to do a group by on hours and minutes and do aggregation based on the max value here and it is going to take some time to finish it, right. So, if the wall clock time is took around 4 seconds to do that aggregation that we just talked about, right.

(Refer Slide Time: 16:01)



```
date_df['date'] = pd.date_range(start='2019-07-05', end='2019-07-06', freq='ms')
date_df['value'] = np.random.sample(len(date_df))

date_df['hour'] = date_df.date.dt.hour
date_df['minute'] = date_df.date.dt.minute
date_df['second'] = date_df.date.dt.second

print(date_df.shape)
print(date_df.head())

(86400000, 5)

   date      value  hour  minute  second
0 2019-07-05 00:00:00.000 0.453380 0 0 0
1 2019-07-05 00:00:00.001 0.884337 0 0 0
2 2019-07-05 00:00:00.002 0.449341 0 0 0
3 2019-07-05 00:00:00.003 0.596637 0 0 0
4 2019-07-05 00:00:00.004 0.552345 0 0 0
CPU times: user 1.95 ms, sys: 83.3 ms, total: 1.97 s
Wall time: 1.96 ms

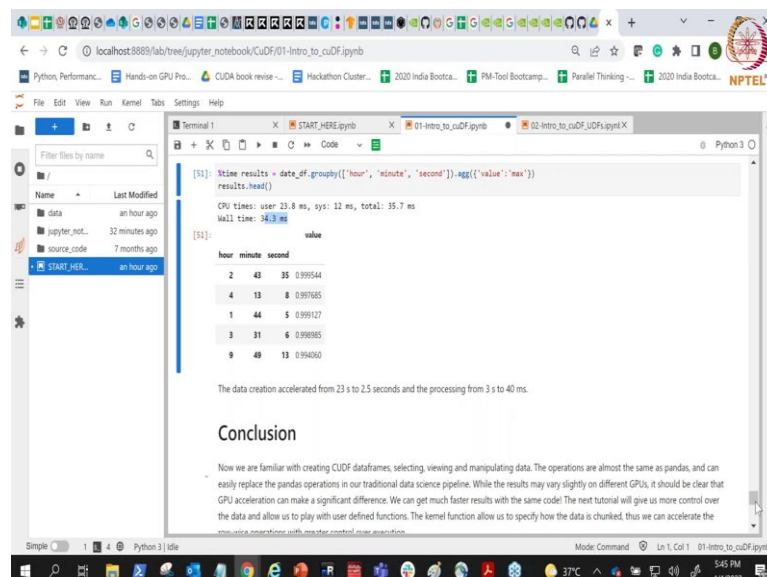
[51]: %time results = date_df.groupby(['hour', 'minute', 'second']).agg({'value': 'max'})
results.head()

CPU times: user 23.8 ms, sys: 12 ms, total: 35.7 ms
Wall time: 34.3 ms

[52]:
   hour  minute  second      value
2    43     35  0.999544
4    13      8  0.997685
```

So, let us do the same thing on the GPU and see. So, you can see here on the GPU it has already finished and it is not even in seconds it is actually 23 milliseconds versus 4 seconds.

(Refer Slide Time: 16:09)



```
[51]: %time results = date_df.groupby(['hour', 'minute', 'second']).agg({'value': 'max'})
results.head()

CPU times: user 23.8 ms, sys: 12 ms, total: 35.7 ms
Wall time: 34.3 ms

[52]:
   hour  minute  second      value
2    43     35  0.999544
4    13      8  0.997685
1    44      5  0.999127
3    31      6  0.998885
9    49     13  0.994090

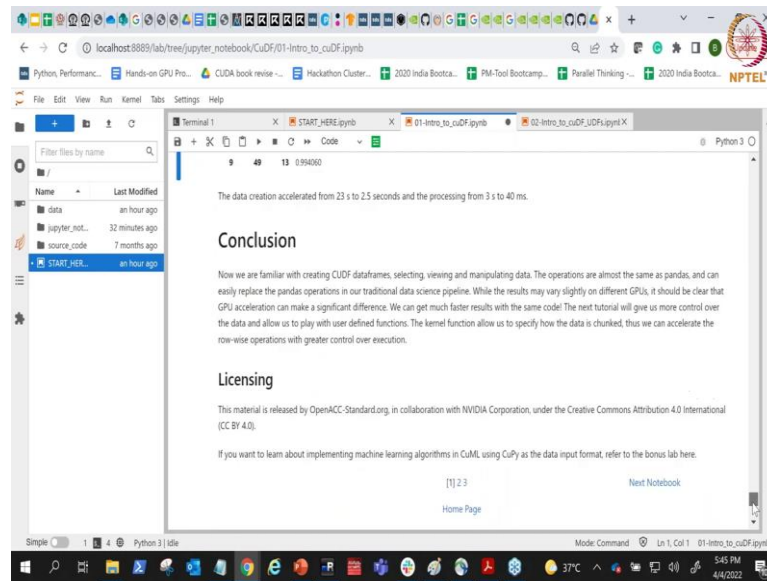
The data creation accelerated from 23 s to 2.5 seconds and the processing from 3 s to 40 ms.

Conclusion

Now we are familiar with creating CUDF dataframes, selecting, viewing and manipulating data. The operations are almost the same as pandas, and can easily replace the pandas operations in our traditional data science pipeline. While the results may vary slightly on different GPUs, it should be clear that GPU acceleration can make a significant difference. We can get much faster results with the same code! The next tutorial will give us more control over the data and allow us to play with user defined functions. The kernel function allow us to specify how the data is chunked, thus we can accelerate the operations compared to the pandas control user application.
```

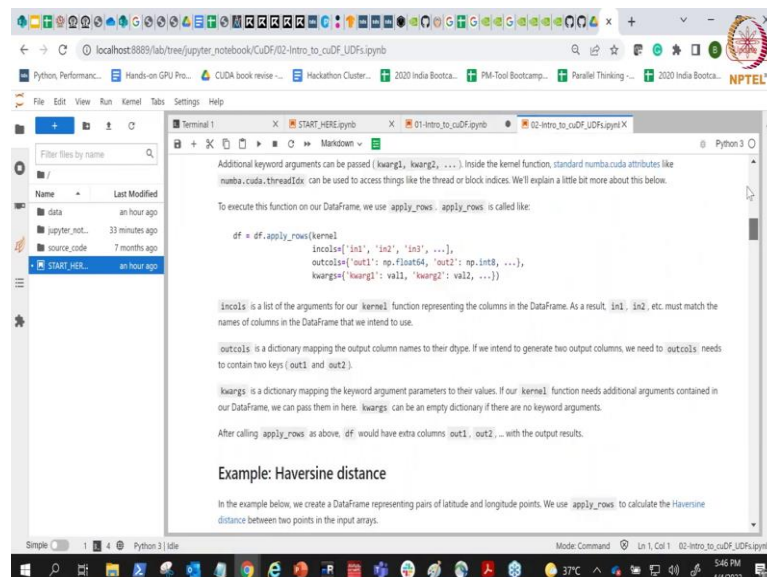
So, the group by of this particular case took around 4 seconds and on the GPU it took around 34 milliseconds right and this is one of the part which I wanted to show was that even though there are so many functionalities.

(Refer Slide Time: 16:31)

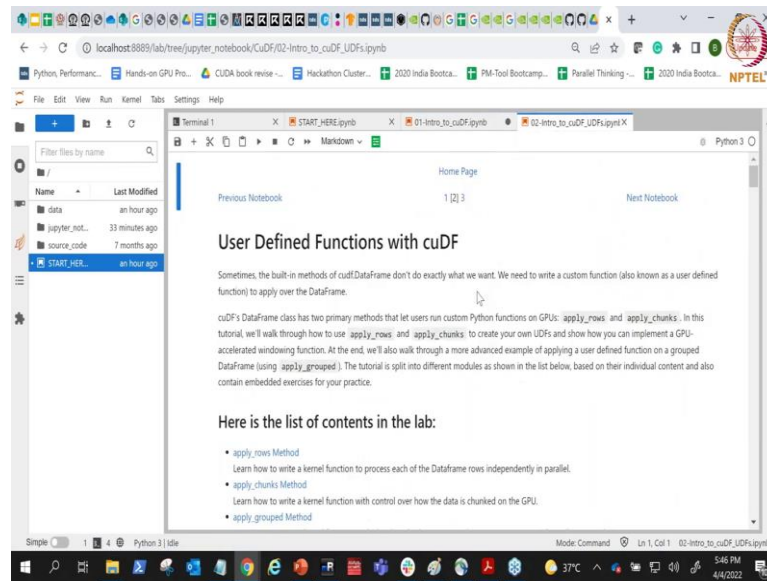


But one of the most critical thing also to understand is the part that, the primary reason where you will use it is when you want to do certain complicated operations or you have large data set which may take a few hours to few days also to do these kinds of operation and that is why cuDF makes it is impact.

(Refer Slide Time: 17:00)

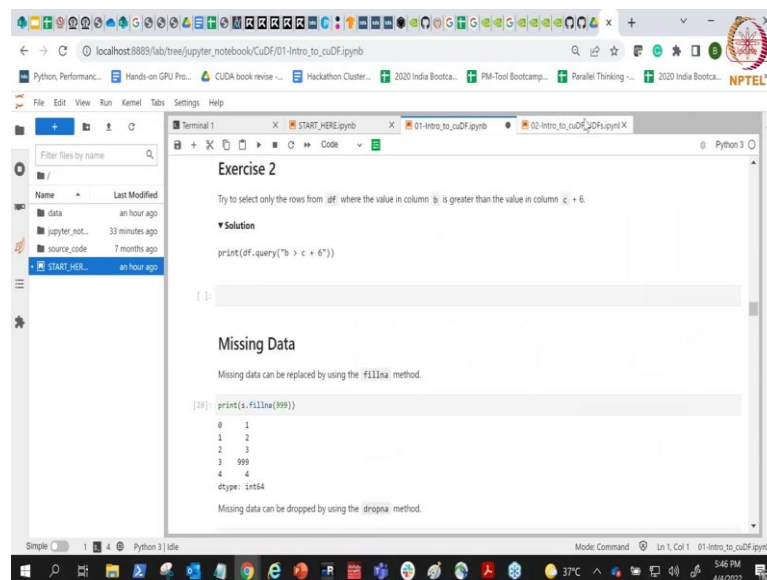


(Refer Slide Time: 17:02)



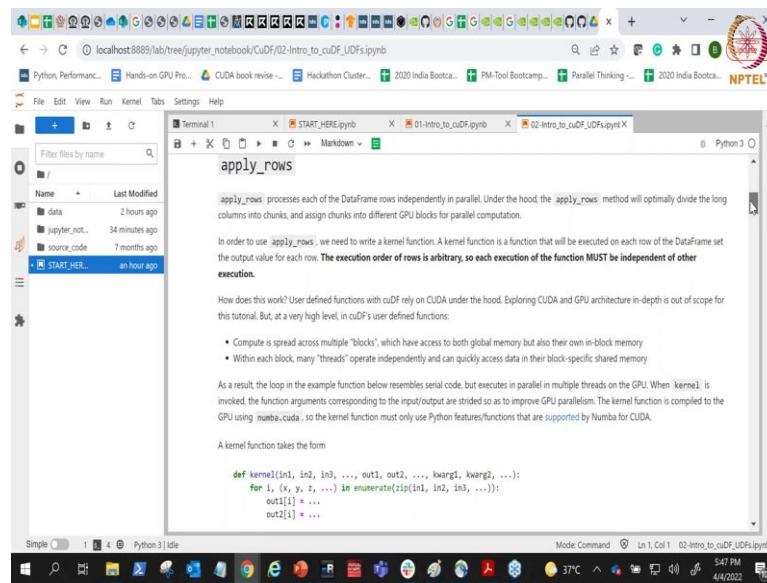
Now, what if now what if this particular functionality which I just talked to you about is not present, like I showed you so many functions here string functions this function that function.

(Refer Slide Time: 17:11)



What if you had a custom operation that you wanted to do on the data which exist. In that case, you can also define something called as user defined functions and you can define your own complicated functions and you can use one of the features which is either `apply_rows` or `apply_chunks`.

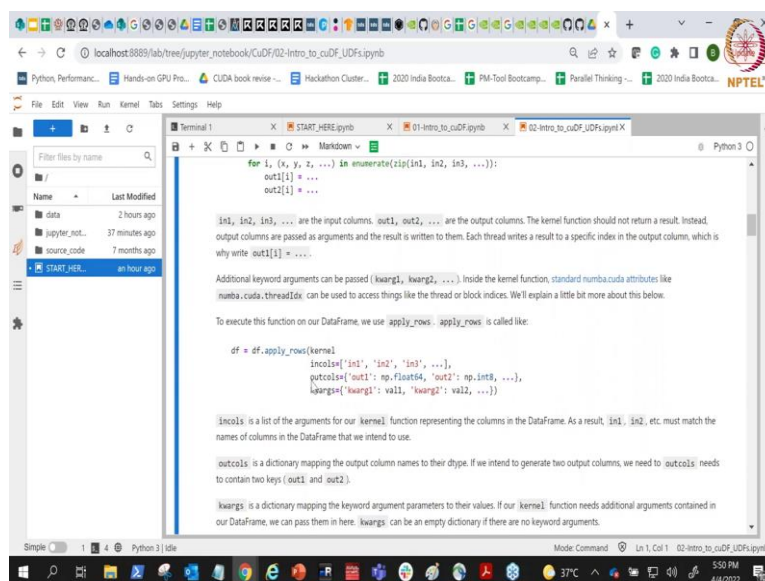
(Refer Slide Time: 17:29)



So, either we will use apply row or apply chunks and I am going to show you very quickly on what that means. So, apply row basically processes each of the row independently in parallel. So, if you remember GPU computing is all about being in parallel, running things in parallel, right.

So, if I call apply row to a particular data frame and if I define what is going to happen inside that apply row that method that row calculation will be independently happening in parallel on the GPU, right. And it will optimally divide the columns into chunks and assign those chunks to different GPU blocks.

(Refer Slide Time: 18:21)



The screenshot shows a Jupyter Notebook with a file browser on the left and a code editor on the right. The code editor contains a kernel function and its application to a DataFrame. The kernel function is defined as follows:

```
for i, (x, y, z, ...) in enumerate(zip(in1, in2, in3, ...)):
    out1[i] = ...
    out2[i] = ...
```

The code also includes a docstring explaining the function's purpose and usage. The application of the kernel function to a DataFrame is shown as follows:

```
df = df.apply_rows(kernel,
                    incols=['in1', 'in2', 'in3', ...],
                    outcols={'out1': np.float64, 'out2': np.int8, ...},
                    kwargs={'val1': val1, 'val2': val2, ...})
```

The code also includes a docstring explaining the function's purpose and usage. The application of the kernel function to a DataFrame is shown as follows:

And what are these blocks and all I will talk about that in a bit, but one of the most critical thing that I have to say and again repeat that I told in the last thing also is that that the execution order of row is arbitrary.

So, each execution of the function must be independent of each other which means, that you cannot be saying that I am dependent on a value on row 1 and only once I have that value then I can change the row 5 or so.

So, if you have a dependency among the rows you cannot use that. So, cuDF under the hood as I said would depend on CUDA behind the scene and it is going to run it on the CUDA architecture and what happens behind the scene it is run across multiple blocks and threads I will quickly say what a block and thread concept is in a bit for now you can assume that something runs in parallel in terms of blocks and threads and then it makes use of a package called a numba.

So, numba is the one which what it will do is it will actually do a JIT compilation just in 10 compilation of your of your function which is supposed to run on GPU which you have defined to be used in apply in the apply rows and it compiles it to lower level CUDA code which finally is given to the GPU.

And how does the function look like, it will basically look like this. Whenever you are defining a function that needs to run on GPU in parallel, you will have certain input

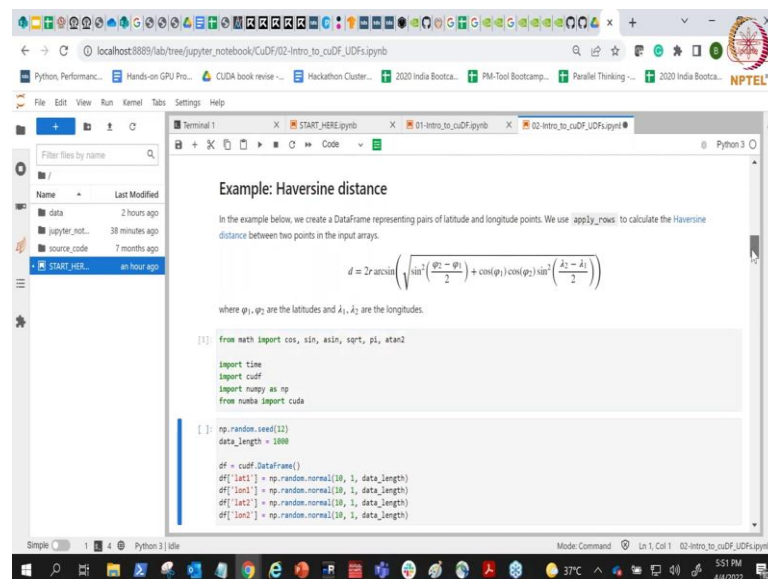
columns like this, you will have certain output columns and then you will have certain additional arguments that you want to pass through your function that needs to run in parallel.

And then, you will have for i and then you will do the operation that you want to do in parallel across all the rows. So and how do you call it? So, you will call it in this fashion you will say apply rows you will define the name of the function which in this case is kernel and then, you will say what are the input columns.

So, you will create an array of it. So, you will say in columns in 1 in 2 in 3 the number of columns that you want, then you will define your out columns and you can see here for the out columns you are also defining the data type.

Like you are saying that the output 1 is of type float 64 and the output 2 is of type integer 8 bit and you will also define the input additional argument that you would like it to have, right. So, the outputs are not returned back outputs are also passed as an argument in case of CUDA world and that is why even to apply rows you pass it as an as a argument here, right.

(Refer Slide Time: 21:05)



The screenshot shows a Jupyter Notebook interface with a file explorer on the left and a code editor on the right. The notebook is titled '02-Intro_to_cuda_UDFs.ipynb'. The code in the notebook is as follows:

```
Example: Haversine distance

In the example below, we create a DataFrame representing pairs of latitude and longitude points. We use apply_rows to calculate the Haversine distance between two points in the input arrays.


$$d = 2r \arcsin \left( \sqrt{\sin^2 \left( \frac{\phi_2 - \phi_1}{2} \right) + \cos(\phi_1) \cos(\phi_2) \sin^2 \left( \frac{\lambda_2 - \lambda_1}{2} \right)} \right)$$


where  $\phi_1, \phi_2$  are the latitudes and  $\lambda_1, \lambda_2$  are the longitudes.

[1]: from math import cos, sin, asin, sqrt, pi, atan2

import time
import cudf
import numpy as np
from numba import cuda

[ ]: np.random.seed(12)
data_length = 1000

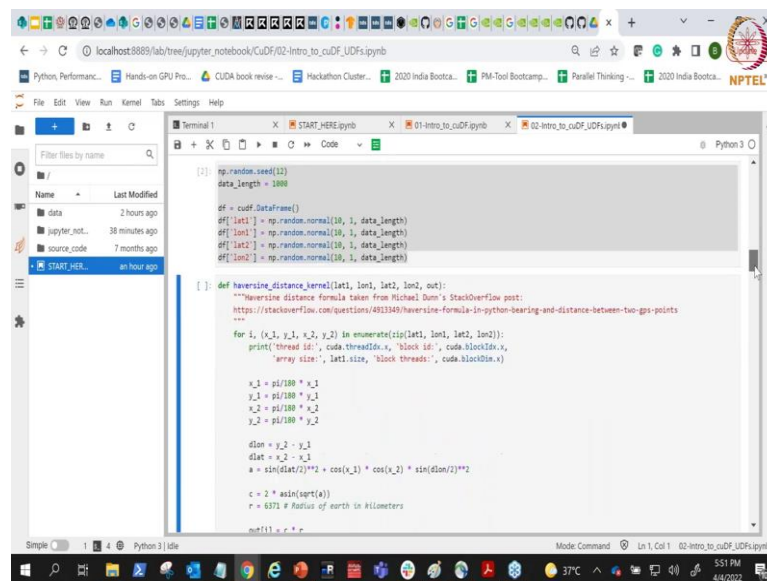
df = cudf.DataFrame()
df['lat1'] = np.random.normal(38, 1, data_length)
df['lon1'] = np.random.normal(10, 1, data_length)
df['lat2'] = np.random.normal(38, 1, data_length)
df['lon2'] = np.random.normal(10, 1, data_length)
```

So, this is an example of in case you for a data frame if you wanted to represent the pairs of longitude, latitude and longitude here and you wanted to calculate the haversine

distance. So, you can if you click on this it will take you to the Wikipedia definition of it, but you can see here basically what we are trying to do for the latitude and longitude.

So, we are trying to do here certain sine and cosine functionality here where we have these variables as latitudes and we have the gamma variables as the longitudes here. So, you can see here from math we are importing cos, sin and different functionalities here.

(Refer Slide Time: 21:47)



```
[2]: np.random.seed(12)
data_length = 1000

df = cudf.DataFrame()
df['lat1'] = np.random.normal(18, 1, data_length)
df['lon1'] = np.random.normal(18, 1, data_length)
df['lat2'] = np.random.normal(18, 1, data_length)
df['lon2'] = np.random.normal(18, 1, data_length)

[ ]: def haversine_distance_kernel(lat1, lon1, lat2, lon2, out):
    """Haversine distance formula taken from Michael Dum's StackOverflow post:
    https://stackoverflow.com/questions/4913349/haversine-formula-in-python-bearing-and-distance-between-two-gps-points
    """
    for i, (x1, y1, x2, y2) in enumerate(zip(lat1, lon1, lat2, lon2)):
        print("thread id:", cuda.threadIdx.x, "block id:", cuda.blockIdx.x,
              "array size:", lat1.size, "block threads:", cuda.blockDim.x)

        x1 = pi/180 * x1
        y1 = pi/180 * y1
        x2 = pi/180 * x2
        y2 = pi/180 * y2

        dlon = y2 - y1
        dlat = x2 - x1
        a = sin(dlat/2)**2 + cos(x1) * cos(x2) * sin(dlon/2)**2

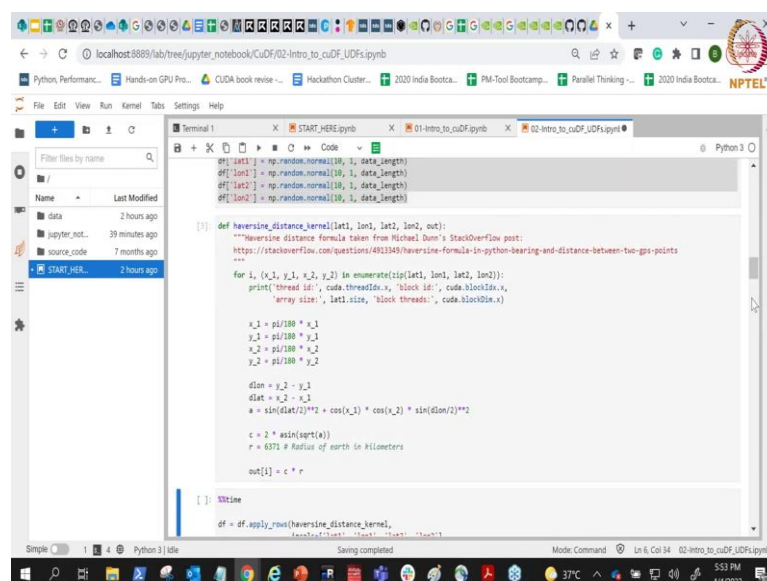
        c = 2 * asin(sqrt(a))
        r = 6371 # Radius of earth in kilometers

        out[i] = c * r

    return out
```

And then, we are creating those latitudes and longitudes and then we are defining our function which I just told you about.

(Refer Slide Time: 21:55)



```
[ ]: df = df.apply_rows(haversine_distance_kernel,
                        scalarArgs={'lat1': lat1, 'lon1': lon1, 'lat2': lat2, 'lon2': lon2})

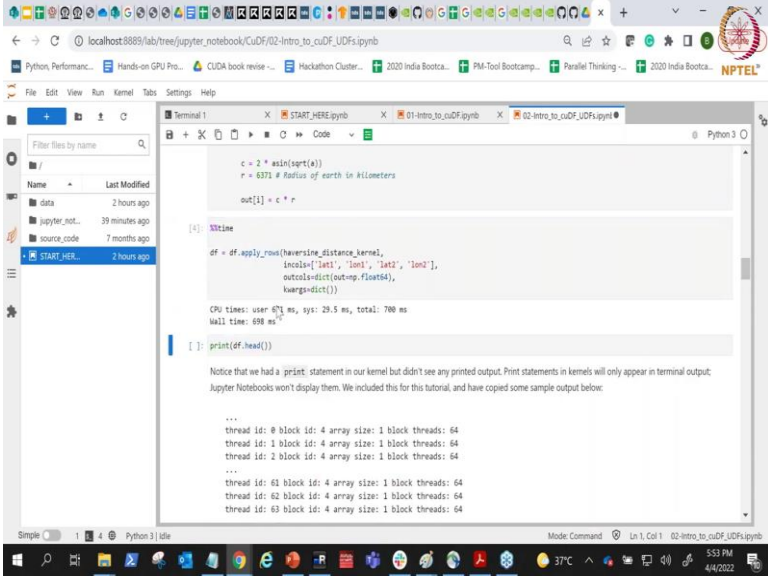
[ ]: %time
```

So, we are doing the we want to do haversine distance function and you can see here we have couple of inputs latitude 1, longitude 1, latitude 2, longitude 2 and we are trying to find the haversine distance between them and this is where the output or the or the haversine distance is actually put and given as an written back as a output.

And you can see here we are saying for the x1, y1, x2, y2 which is latitude 1, longitude 1, latitude 2, longitude 2 and I am doing certain additional calculation here I will go through certain details of CUDA tomorrow.

So, that you get an idea of what I am trying to say here what is it is thread id and block id maybe I will show it to you tomorrow in the not in the tomorrow session, but the next session which is happening on Wednesday; and what I am doing is I am getting this value x1, y1, x2, y2 calculating those just as the sine and the cosine transforms and finally, storing the value here, right. So, what I am saying is that, I want to take this function and I want to calculate this haversine distance in parallel for which I am doing something called as apply rows.

(Refer Slide Time: 23:12)



The screenshot shows a Jupyter Notebook interface with a file explorer on the left and a code editor on the right. The code editor contains the following code:

```
c = 2 * asin(sqrt(a))
r = 6371 # Radius of earth in kilometers
out[1] = c * r
```

Below the code, the output of the `df.apply_rows` function is displayed, showing the execution time and the output of the kernel. The output is a dictionary with the following structure:

```
df = df.apply_rows(haversine_distance_kernel,
                  incols=['lat1', 'lon1', 'lat2', 'lon2'],
                  outcols=dict(out=np.float64),
                  kwargs=dict())
```

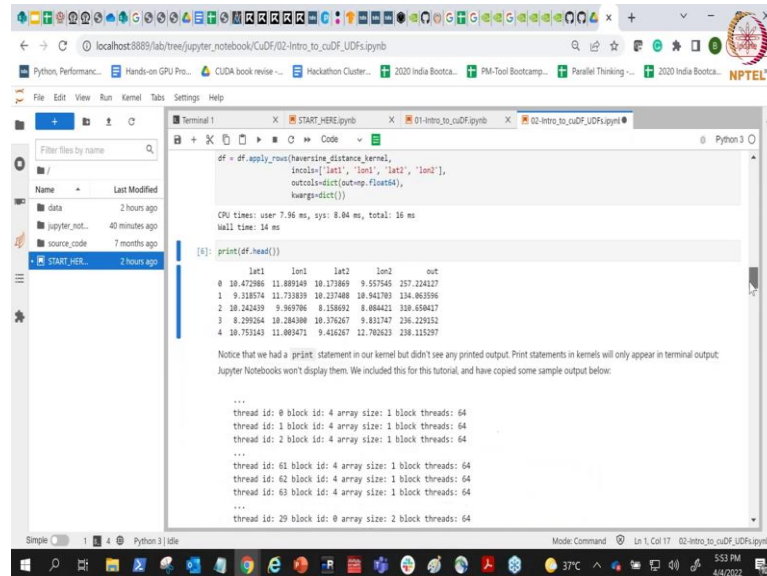
The output shows the execution time: CPU times: user 471 ms, sys: 23.5 ms, total: 700 ms; Wall time: 698 ms. The output of the kernel is a dictionary with the following structure:

```
thread id: 0 block id: 4 array size: 1 block threads: 64
thread id: 1 block id: 4 array size: 1 block threads: 64
thread id: 2 block id: 4 array size: 1 block threads: 64
...
thread id: 61 block id: 4 array size: 1 block threads: 64
thread id: 62 block id: 4 array size: 1 block threads: 64
thread id: 63 block id: 4 array size: 1 block threads: 64
```

When I do the apply row, I am giving it the input columns which I just had I am telling it for all the columns in parallel I am going to calculate this haversine distance. The output is of type dictionary and it is having a data type float 64 and this is the part which I am just passing to it and I just need to run this and I am going to again run this and it is going to do it in parallel you can see here they just calculated in 698 milliseconds and in

fact the second time it becomes much faster 14 milliseconds and it is basically going to print out.

(Refer Slide Time: 23:51)



```
df = df.apply_rows(haversine_distance_kernel,
                  iscols=['lat1', 'lon1', 'lat2', 'lon2'],
                  outcols=dict(out=np.float64),
                  kwargs=dict())

CPU times: user 7.96 ms, sys: 8.84 ms, total: 16 ms
Wall time: 14 ms

[5]: print(df.head())

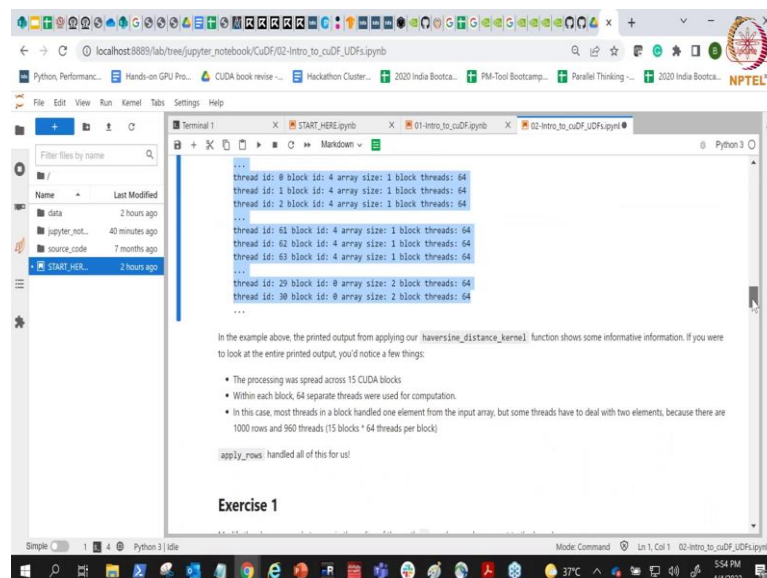
   lat1  lon1  lat2  lon2  out
0  10.472886  11.889149  10.173869  9.557545  257.224227
1  9.318574  11.733839  10.237488  10.941703  134.063596
2  10.242439  9.369706  8.158692  8.684211  130.658427
3  8.292554  10.284389  10.376267  9.832147  136.229522
4  10.753243  11.889471  9.436267  12.782623  238.115297

Notice that we had a print statement in our kernel but didn't see any printed output. Print statements in kernels will only appear in terminal output; Jupyter Notebooks won't display them. We included this for the tutorial, and have copied some sample output below:

...
thread id: 0 block id: 4 array size: 1 block threads: 64
thread id: 1 block id: 4 array size: 1 block threads: 64
thread id: 2 block id: 4 array size: 1 block threads: 64
...
thread id: 61 block id: 4 array size: 1 block threads: 64
thread id: 62 block id: 4 array size: 1 block threads: 64
thread id: 63 block id: 4 array size: 1 block threads: 64
...
thread id: 29 block id: 0 array size: 2 block threads: 64
```

So, if this is your latitude 1, longitude 1, latitude 2, longitude 2 this is the haversine distance which is calculated as the output here.

(Refer Slide Time: 24:10)



```
...
thread id: 0 block id: 4 array size: 1 block threads: 64
thread id: 1 block id: 4 array size: 1 block threads: 64
thread id: 2 block id: 4 array size: 1 block threads: 64
...
thread id: 61 block id: 4 array size: 1 block threads: 64
thread id: 62 block id: 4 array size: 1 block threads: 64
thread id: 63 block id: 4 array size: 1 block threads: 64
...
thread id: 29 block id: 0 array size: 2 block threads: 64
thread id: 30 block id: 0 array size: 2 block threads: 64
...

In the example above, the printed output from applying our haversine_distance_kernel function shows some informative information. If you were to look at the entire printed output, you'd notice a few things:

• The processing was spread across 15 CUDA blocks
• Within each block, 64 separate threads were used for computation.
• In this case, most threads in a block handled one element from the input array, but some threads have to deal with two elements, because there are 1000 rows and 960 threads (15 blocks * 64 threads per block)

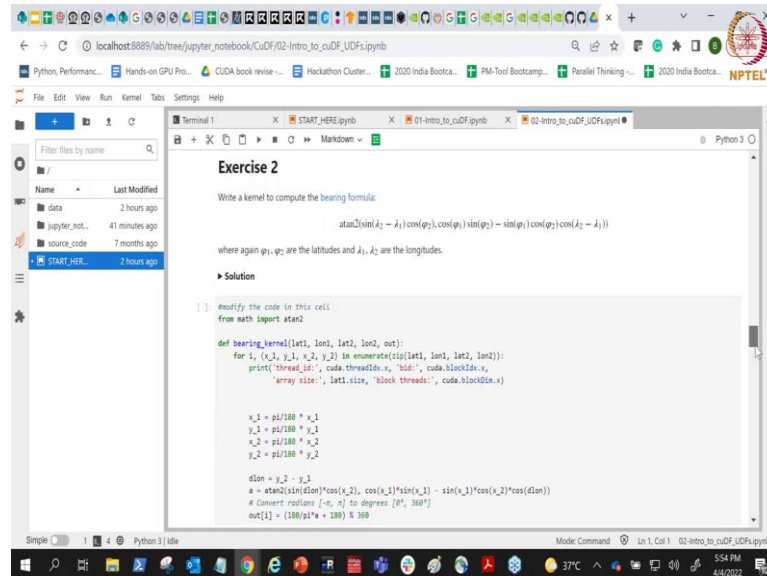
apply_rows handled all of this for us!

Exercise 1
```

So, we have done certain print up statements also and I am not putting those print up statements here at this point of time there are certain outputs it would not display them in the Jupyter notebook because, the print up statement will only appear in the terminal

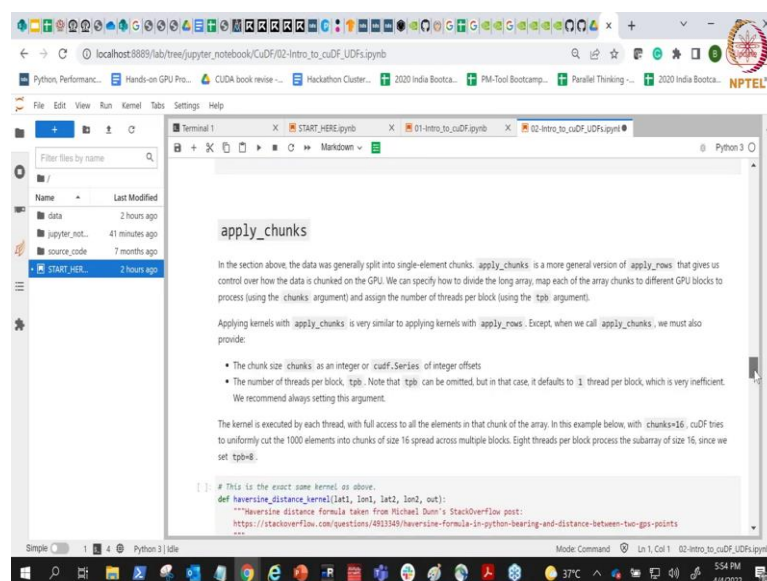
output. So, only if you run it on the terminal these values are going to be output it does not get output on the on the Jupyter notebook.

(Refer Slide Time: 24:39)



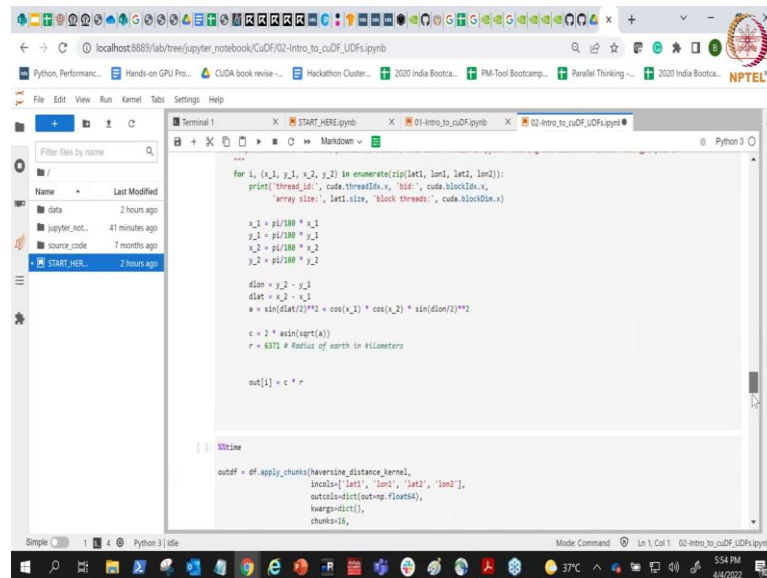
But, I am going to talk about it later on, but once you run this you would understand that it is running something called as CUDA blocks within each block it was running 64 threads and how many rows and number of threads were basically done. So, this is this is the part of the 1st exercise.

(Refer Slide Time: 24:57)



The 2nd part is if you want to take more control on the chunks values and how you want to so, in the previous case we left it up to the numba and up to the apply rows to basically define at what chunk level the data gets divided ah. So, tomorrow what we are going to do is, we are going to first understand how to data gets divided among the CUDA blocks or CUDA threads.

(Refer Slide Time: 25:27)



```
for i, (x1, y1, x2, y2) in enumerate(zip(lat1, lon1, lat2, lon2)):
    print("thread_id:", cuda.threadIdx.x, "bid:", cuda.blockIdx.x,
          "array size:", lat1.size, "block threads:", cuda.blockDim.x)

    x1 = pi/180 * x1
    y1 = pi/180 * y1
    x2 = pi/180 * x2
    y2 = pi/180 * y2

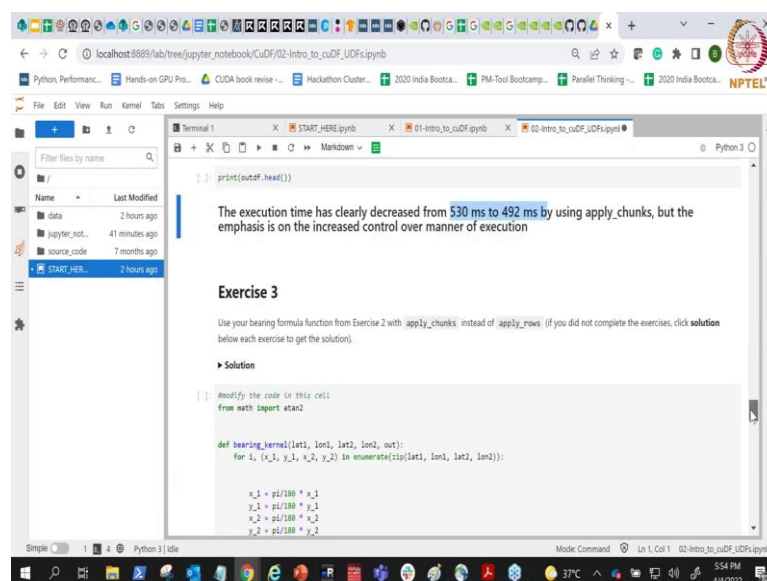
    dlon = y2 - y1
    dlat = x2 - x1
    a = sin(dlat/2)**2 + cos(x1) * cos(x2) * sin(dlon/2)**2
    c = 2 * asin(sqrt(a))
    r = 6371 # Radius of earth in kilometers

    out[i] = c * r

@jit
def outdf = df.apply_chunks(haversine_distance_kernel,
                           incols=['lat1', 'lon1', 'lat2', 'lon2'],
                           outcols=dict(out=np.float64),
                           kwargs=dict(),
                           chunks=16,
```

And then, we will look at the applied chunks button.

(Refer Slide Time: 25:28)



The execution time has clearly decreased from 530 ms to 492 ms by using apply_chunks, but the emphasis is on the increased control over manner of execution

Exercise 3

Use your bearing formula function from Exercise 2 with `apply_chunks` instead of `apply_rows` (if you did not complete the exercises, click [solution](#) below each exercise to get the solution).

Solution

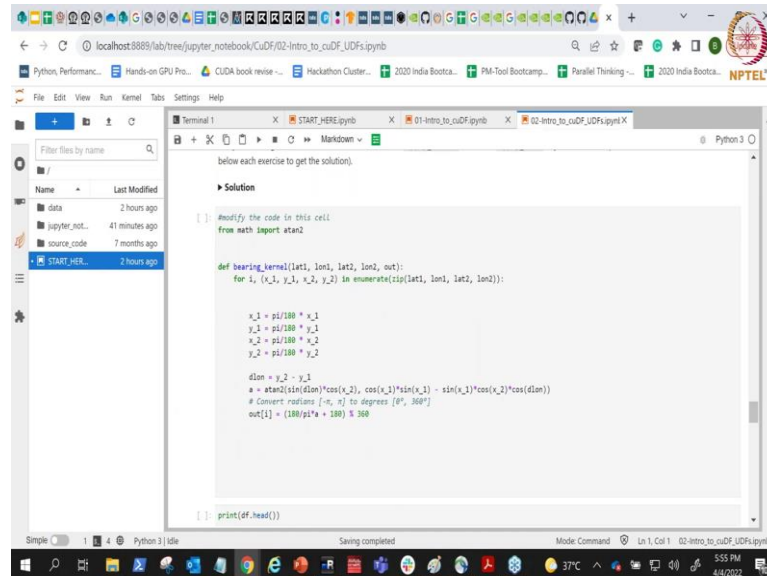
```
#modify the code in this cell
from math import atan2

def bearing_kernel(lat1, lon1, lat2, lon2, out):
    for i, (x1, y1, x2, y2) in enumerate(zip(lat1, lon1, lat2, lon2)):

        x1 = pi/180 * x1
        y1 = pi/180 * y1
        x2 = pi/180 * x2
        y2 = pi/180 * y2
```

Which is which is much more which will give you more performance as compared to blindly just calling apply rows.

(Refer Slide Time: 25:35)



The screenshot shows a Jupyter Notebook interface with a file browser on the left and a code editor in the center. The code editor contains a Python function named `bearing_kernel` that takes four arguments: `lat1`, `lon1`, `lat2`, and `lon2`. The function uses `atan2` to calculate the bearing between two points. The code is as follows:

```
below each exercise to get the solution.

# modify the code in this cell
from math import atan2

def bearing_kernel(lat1, lon1, lat2, lon2, out):
    for i, (x1, y1, x2, y2) in enumerate(zip(lat1, lon1, lat2, lon2)):

        x1 = pi/180 * x1
        y1 = pi/180 * y1
        x2 = pi/180 * x2
        y2 = pi/180 * y2

        dlon = y2 - y1
        a = atan2(sin(dlon)*cos(x2), cos(x1)*sin(x1) - sin(x1)*cos(x2)*cos(dlon))
        # Convert radians [-pi, pi] to degrees [0, 360]
        out[i] = (180/pi*a + 180) % 360

print(df.head())
```

So, with that we are kind of done with the first part and I will look at if there are.