Python for Data Sciences

Parallelizing: Nvidia Libraries

- Armando Camerlingo

IR0000032 – ITINERIS, Italian Integrated Environmental Research Infrastructures System
(D.D. n. 130/2022 - CUP B53C22002150006) Funded by EU - Next Generation EU PNRR-Mission 4 “Education and Research” - Component 2: “From research to business” - Investment 3.1: “Fund for the realisation of an integrated system of research and innovation infrastructures”

Finanziato dall'Unione europea NextGenerationEU

Ministero dell’Università e della Ricerca

Italiadomani

Nvidia Libraries

CuPy

RAPIDS

cuML
cuDF

NVIDIA
CUDA

Compute Infrastructure

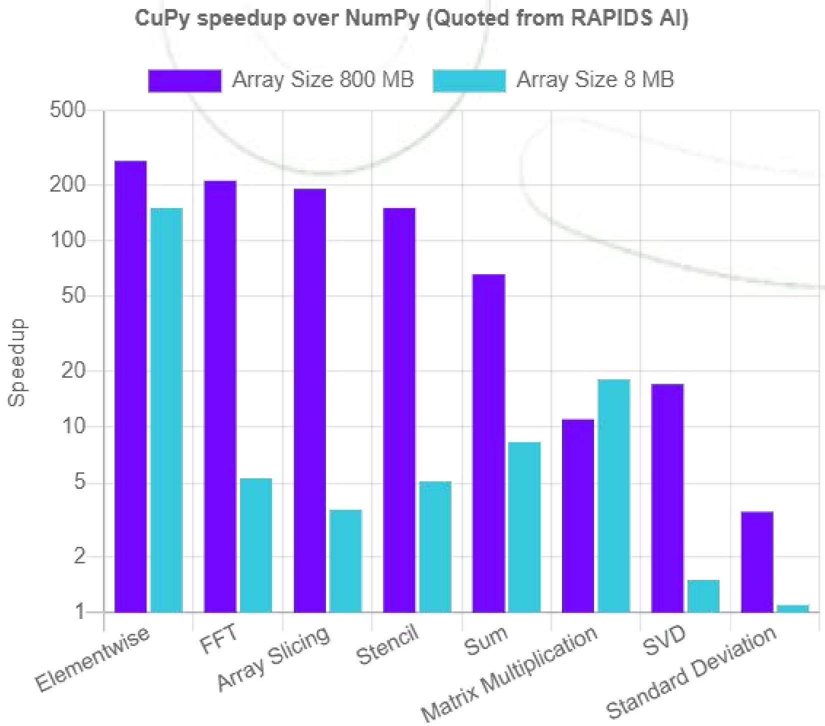
Cloud | Data Center | Desktop | Laptop

Nvidia RAPIDS

Nvidia RAPIDS is a collection of library and resource to accelerate Data Science with Nvidia GPUs. Based on CUDA C++ code, it is implented in order to work with all the major frameworks used in ML, DL and AI.

Nvidia CuPy

CuPy is an open-source library for creating and managing arrays on Nvidia GPUs. It implements different CUDA mathematical libraries that speeds up a lot of operations commonly encountered during scientific computing.



CuPy: creating arrays

```
import numpy as np
import cupy as cp
cp.cuda.Stream.null.synchronize()
```

Let's compare numpy and cupy on the creation of a 2GB array

```
%%timeit -r 1 -n 10
global x_cpu
x_cpu = np.ones((1000, 500, 500))
```

```
448 ms ± 0 ns per loop (mean ± std. dev. of 1 run, 10 loops each)
```

```
%%timeit -n 10
global x_gpu
x_gpu = cp.ones((1000, 500, 500))

cp.cuda.Stream.null.synchronize()
```

```
17.1 ms ± 21.1 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

CuPy: basic operations

```
%%time
x_cpu *= 5
```

```
CPU times: user 189 ms, sys: 0 ns, total: 189 ms
Wall time: 189 ms
```

```
%%time
x_gpu *= 5

cp.cuda.Stream.null.synchronize()
```

```
CPU times: user 15.9 ms, sys: 0 ns, total: 15.9 ms
Wall time: 15.9 ms
```

CuPy: basic operations

```
%%time
x_cpu *= 5
x_cpu *= x_cpu
x_cpu += x_cpu
```

```
CPU times: user 587 ms, sys: 0 ns, total: 587 ms
Wall time: 588 ms
```

```
%%time
x_gpu *= 5
x_gpu *= x_gpu
x_gpu += x_gpu

cp.cuda.Stream.null.synchronize()
```

```
CPU times: user 47.4 ms, sys: 0 ns, total: 47.4 ms
Wall time: 47.3 ms
```

CuPy: basic operations

```
%%time
x_cpu *= 5
x_cpu *= x_cpu
x_cpu += x_cpu
```

```
CPU times: user 587 ms, sys: 0 ns, total: 587 ms
Wall time: 588 ms
```

```
%%time
x_gpu *= 5
x_gpu *= x_gpu
x_gpu += x_gpu

cp.cuda.Stream.null.synchronize()
```

```
CPU times: user 47.4 ms, sys: 0 ns, total: 47.4 ms
Wall time: 47.3 ms
```

CuPy: (not so) basic operations

```
%%time
x_cpu = np.random.random((1000, 1000))
u, s, v = np.linalg.svd(x_cpu)
```

```
CPU times: user 2.49 s, sys: 23.9 ms, total: 2.51 s
Wall time: 2.33 s
```

```
%%time
x_gpu = cp.random.random((1000, 1000))
u, s, v = np.linalg.svd(x_gpu)  # Note the `np` used here

cp.cuda.Stream.null.synchronize()
```

```
CPU times: user 423 ms, sys: 212 ms, total: 634 ms
Wall time: 634 ms
```

CuPy: (not so) basic operations

```
c = np.random.rand(1_000, 1_000)
c.shape
```

```
(1000, 1000)
```

```
d = np.random.rand(1_000, 1_000)
d.shape
```

```
(1000, 1000)
```

```
%timeit np.matmul(c,d)
```

```
54.7 ms ± 12 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

```
c_gpu = cp.asarray(c)
d_gpu = cp.asarray(d)
```

```
%timeit cp.matmul(c_gpu,d_gpu)
```

```
156 µs ± 72.5 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

CuPy: (not so) basic operations

```
e_np = np.random.rand(10_000,10_000)
f_np = np.random.rand(10_000,10_000)
```

```
e_cp = cp.asarray(e_np)
f_cp = cp.asarray(f_np)
```

```
import dask.array as da
e_dask = da.from_array(e_np, chunks = "auto")
f_dask = da.from_array(f_np, chunks = "auto")
```

```
%timeit -n 1 -r 1 np.matmul(e_np, f_np)
```

```
52.9 s ± 0 ns per loop (mean ± std. dev. of 1 run, 1 loop each)
```

```
%timeit -n 1 -r 1 cp.matmul(e_cp, f_cp)
```

```
437 µs ± 0 ns per loop (mean ± std. dev. of 1 run, 1 loop each)
```

```
%timeit -n 1 -r 1 da.matmul(e_dask, f_dask).compute()
```

```
56.1 s ± 0 ns per loop (mean ± std. dev. of 1 run, 1 loop each)
```

CuPy: (not so) basic operations

What about Numba?

```
!uv pip install -q --system numba-cuda==0.4.0
```

```
from numba import config
config.CUDA_ENABLE_PYNVJITLINK = 1
```

```
from numba import vectorize
@vectorize(['float64(float64, float64)'], target='cuda')
def add_func(x,y):
    return x+y
```

```
%timeit -n 7 -r 1 np.add(e_np, f_np)
```

```
309 ms ± 36.5 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

```
%timeit -n 7 -r 1 cp.add(e_cp, f_cp)
```

```
115 µs ± 0 ns per loop (mean ± std. dev. of 1 run, 7 loops each)
```

```
%timeit -n 7 -r 1 da.add(e_dask, f_dask).compute()
```

```
578 ms ± 0 ns per loop (mean ± std. dev. of 1 run, 7 loops each)
```

CuPy: (not so) basic operations

What about Numba?

```
%timeit -n 7 -r 1 add_func(e_np,f_np)
```

```
684 ms ± 0 ns per loop (mean ± std. dev. of 1 run, 7 loops each)
```

Remember we have to manage memory:

```
from numba import cuda
e_cuda = cuda.to_device(e_np)
f_cuda = cuda.to_device(f_np)
out_cuda = cuda.device_array(shape=(10_000,10_000), dtype=np.float64)
```

```
%%timeit
add_func(e_cuda,f_cuda, out=out_cuda)
out_host = out_cuda.copy_to_host()
out_host
```

```
314 ms ± 18.7 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

CuDF

CuDF, parts of RAPIDS API, can create and manipulate GPU-accelerated dataframes. It uses a syntax very similar to pandas, so everything sould look familiar.

```
import os
import cudf
import cupy as cp
import pandas as pd
import numpy as np
```

```
from pathlib import Path
from zipfile import ZipFile
import requests
data_dir = Path("/content/sample_data/") # replace this with a directory of your choice
dest = data_dir / "nycflights.zip"
with ZipFile(dest) as zf:
  print(zf.filelist[0].filename)
  zf.extractall(path=data_dir)
```

CuDF: read csv

The csv will be loaded directly into the GPU

```
from glob import glob
filenames = sorted(glob(os.path.join('data', 'nycflights', '*.csv')))
```

```
filepath = glob("./data/nycflights/*.csv")
```

Pandas

```
pdf = pd.concat((
                  pd.read_csv(f) for f in
                  filepath
                 ),
                 ignore_index=True)
```

```
pdf.head(1)
```

	Year	Month	DayofMonth	DayOfWeek	DepTime	CRSDepTime	ArrTime	CRSArrTime
0	1993	1	29	5	1055.0	1055	1228.0	1212

CuDF

```
cdf = cudf.concat((
                  pd.read_csv(f) for f in
                  filepath
                 ),
                 ignore_index=True)
```

```
cdf.head(1)
```

	Year	Month	DayofMonth	DayOfWeek	DepTime	CRSDepTime	ArrTime	CRSArrTime
0	1993	1	29	5	1055.0	1055	1228.0	1212

CuDF: data exploration

You can use the same functions as pandas:

```
cdf.head(3)
```

```
cdf.tail(5)
```

```
cdf.columns
```

```
cdf.dtypes
```

CuDF: basic operations

Converting data types

Pandas

```
%time pdf["Cancelled"] =
pdf["Cancelled"].astype("bool"))
```

```
CPU times: user 3.27 ms, sys: 38 µs, total:
3.31 ms
Wall time: 3.62 ms
```

CuDF

```
%time cdf["Cancelled"] =
cdf["Cancelled"].astype("bool")
```

```
CPU times: user 601 µs, sys: 0 ns, total: 601
µs
Wall time: 607 µs
```

CuDF: basic operations

Column-wise aggregation

Pandas

```
%time pdf[~pdf.Cancelled]["DepDelay"].mean()
```

```
CPU times: user 168 ms, sys: 46 ms, total: 214
ms
Wall time: 217 ms
np.float64(8.50948876162038)
```

CuDF

```
%time cdf[~cdf.Cancelled]["DepDelay"].mean()
```

```
CPU times: user 10.5 ms, sys: 11.2 ms, total:
21.7 ms
Wall time: 22 ms
np.float64(8.50948876162038)
```

CuDF: basic operations

String operations

Pandas

```
%time pdf["Dest"] = pdf["Dest"].str.title()
```

```
CPU times: user 250 ms, sys: 24.9 ms, total: 274 ms
Wall time: 305 ms
```

```
pdf.head(5)["Dest"]
```

	Dest
0	Buf
1	Buf
2	Buf
3	Syr
4	Syr

CuDF

```
%time cdf["Dest"] = cdf["Dest"].str.title()
```

```
CPU times: user 3.22 ms, sys: 869 µs, total: 4.09 ms
Wall time: 3.62 ms
```

```
cdf.head(5)["Dest"]
```

```
0    Buf
1    Buf
2    Buf
3    Syr
4    Syr
Name: Dest, dtype: object
```

CuDF: basic operations

Data Selection

Pandas

```
pdf.loc[100:105]
```

```
pdf.iloc[100:105]
```

```
%timeit origin_j_pd =
pdf.loc[pdf["Origin"].str.startswith("J")]
```

```
376 ms ± 165 ms per loop (mean ± std. dev. of 7
runs, 1 loop each)
```

```
%time dest_od =
pdf.loc[np.logical_and(pdf["Dest"].str.startswith("O"),
pdf["Dest"].str.endswith("d"))]
```

```
CPU times: user 605 ms, sys: 723 µs, total: 606 ms
Wall time: 605 ms
```

CuDF

```
cdf.loc[100:105]
```

```
cdf.iloc[100:105]
```

```
%timeit origin_j =
cdf.loc[cdf["Origin"].str.startswith("J")]
```

```
6.45 ms ± 782 µs per loop (mean ± std. dev.
of 7 runs, 100 loops each)
```

```
%time dest_od =
cdf.loc[cp.logical_and(cdf["Dest"].str.startswith("
O"), cdf["Dest"].str.endswith("d"))]
```

```
CPU times: user 4.74 ms, sys: 186 µs, total: 4.92
ms
Wall time: 4.81 ms
```

Exercise

1. Modify the data type of the “Diverted” column to boolean;
2. Change the “Origin” column from UPPERCASE to Titlecase;
3. Check the mean arrival delay of all non-diverted flights;

Solution

1.
```
cdf["Diverted"] = cdf["Diverted"].astype("bool")
cdf.dtypes
```

2.
```
%time cdf["Origin"] = cdf["Origin"].str.title()
```

3.
```
cdf[~cdf.Diverted]["ArrDelay"].mean()
```

CuDF: basic operations

Group Operations

Pandas

```
%%time
departure_delays =
pdf["DepDelay"].groupby(pdf["Origin"])
avg_departure_delay = departure_delays.mean()
avg_departure_delay
```

```
CPU times: user 79.2 ms, sys: 1.77 ms, total:
81 ms
Wall time: 87.7 ms
```

	DepDelay
Origin	
EWR	9.308481
JFK	10.118569
LGA	6.939973

dtype: float64

CuDF

```
%%time
departure_delays =
pdf["DepDelay"].groupby(pdf["Origin"])
avg_departure_delay = departure_delays.mean()
avg_departure_delay
```

```
CPU times: user 15.7 ms, sys: 4.17 ms, total:
19.8 ms
Wall time: 95.7 ms
Origin
LGA     6.939973
EWR     9.308481
JFK    10.118569
Name: DepDelay, dtype: float64
```

CuDF: basic operations

Sorting

Pandas

```
%time pdf_dest = pdf["Dest"].sort_values()
print(pdf_dest[:3])
print(pdf_dest[-3:])
```

```
CPU times: user 1.24 s, sys: 0 ns, total: 1.24
s
Wall time: 1.27 s
285131    ABE
264042    ABE
264043    ABE
Name: Dest, dtype: object
445313    TYS
400479    TYS
400493    TYS
Name: Dest, dtype: object
```

CuDF

```
%time cdf_dest = cdf["Dest"].sort_values()
print(cdf_dest[:3])
print(cdf_dest[-3:])
```

```
CPU times: user 27.6 ms, sys: 1.91 ms, total:
29.5 ms
Wall time: 69.6 ms
200736    ABE
221958    ABE
221959    ABE
Name: Dest, dtype: object
445313    TYS
445314    TYS
445315    TYS
Name: Dest, dtype: object
```

Exercise

Using groupby and sort_values, find which destinations are associated with the lowest arrival delay.

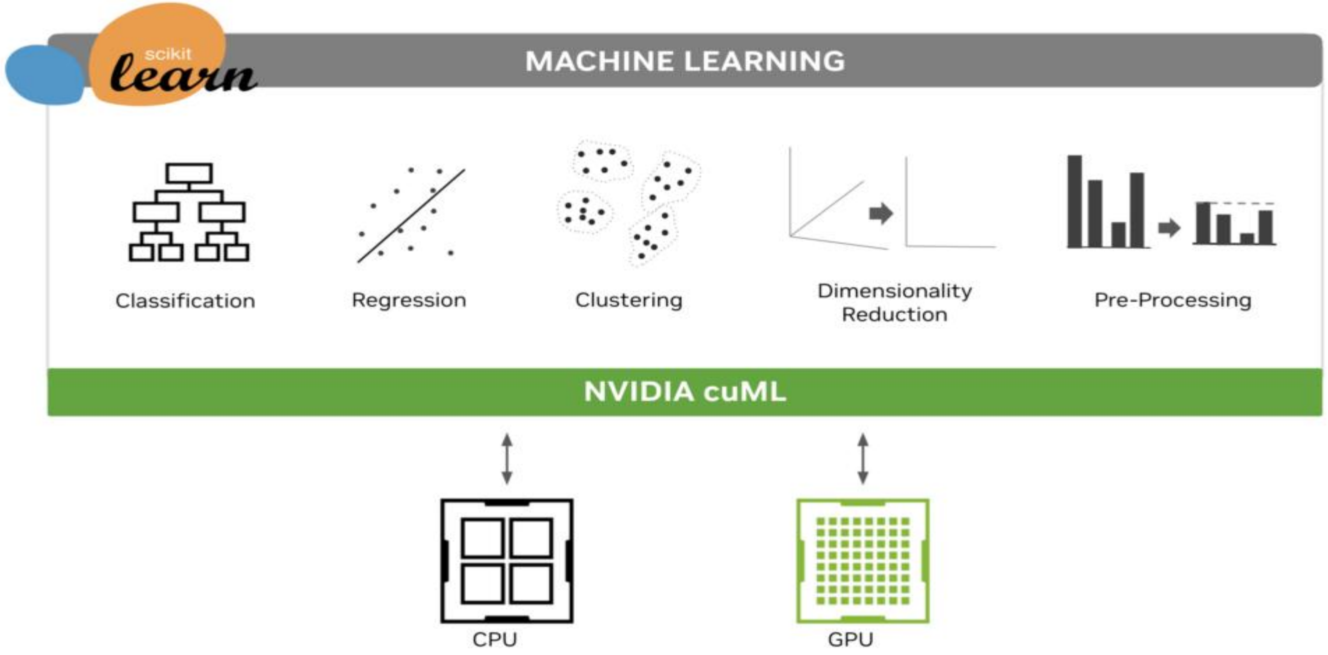
Solution

```python
mean_arrival_delays =
cdf["ArrDelay"].groupby(cdf["Dest"]).mean().sort_values()
mean_arrival_delays
```

```
Dest
SNA     -6.318841
SJC     -5.082927
OMA     -4.802281
PDX     -2.245902
SAV     -2.214112
          ...
ACK     24.636364
EWR     25.750000
ISP     27.714286
ICT     35.315789
JFK    161.250000
Name: ArrDelay, Length: 86, dtype: float64
```

Nvidia CuML

CuML is an open-source library that, using the same syntax of Scikit-Learn, enables the developers to use similar scripts but leveraging the power of Nvidia GPUs.



Nvidia CuML

```
import cudf
df = cudf.read_csv("/content/sample_data/diabetes.csv")
df.head()
```

	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	BMI	DiabetesPedigreeFunction	Age
0	6	148	72	35	0	33.6	0.627	50
1	1	85	66	29	0	26.6	0.351	31
2	8	183	64	0	0	23.3	0.672	32
3	1	89	66	23	94	28.1	0.167	21
4	0	137	40	35	168	43.1	2.288	33

Nvidia CuML

```
X = df.drop(columns=["Outcome"])
X.head()
```

	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	BMI	DiabetesPedigreeFunction	Age
0	6	148	72	35	0	33.6	0.627	50
1	1	85	66	29	0	26.6	0.351	31
2	8	183	64	0	0	23.3	0.672	32
3	1	89	66	23	94	28.1	0.167	21
4	0	137	40	35	168	43.1	2.288	33

```
y = df["Outcome"].values
y[0:5]
```

```
array([1, 0, 1, 0, 1])
```

Nvidia CuML

```
from cuml.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=1,
stratify=y)
```

```
from cuml.neighbors import KNeighborsClassifier

knn = KNeighborsClassifier(n_neighbors = 3)

knn.fit(X_train,y_train)
```

```
knn.predict(X_test)[0:5]
```

```
251    0
325    1
233    0
527    0
464    0
dtype: int64
```

positive

```
knn.predict(X_test)[0:5]
```

```
0.6928104575163399
```

THANKS!

IR0000032 – ITINERIS, Italian Integrated Environmental Research Infrastructures System
(D.D. n. 130/2022 - CUP B53C22002150006) Funded by EU - Next Generation EU PNRR-Mission 4 “Education and Research” - Component 2: “From research to business” - Investment 3.1: “Fund for the realisation of an integrated system of research and innovation infrastructures”

Finanziato dall'Unione europea NextGenerationEU
Ministero dell’Università e della Ricerca
Italiadomani