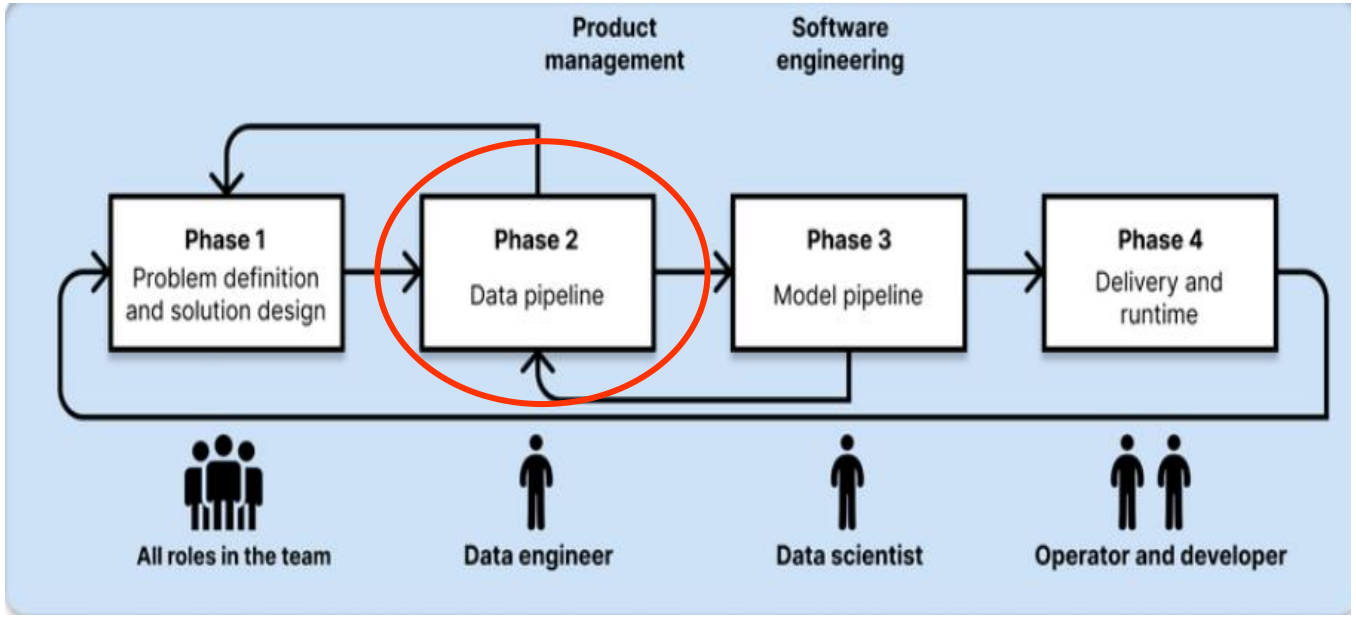
Python for Data Sciences

Data Managing: Polars, Parallelizing: Numba

- 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



- How we receive the data?
- Is the data clean?
- Is the data all relevant to our study case?
- Is the data ready to be ingested by our model?

Data Managing

ITINERIS

Numpy

Pandas

Polars

Polars

ITINERIS

Polars is an open-source library for data manipulation, known for being one of the fastest data processing solutions on a single machine. It features a well-structured, typed API that is both expressive and easy to use.

Polars: advantages

ITINERIS

The goal of Polars is to provide a lightning fast DataFrame library that:

- Utilizes all available cores on your machine.
- Optimizes queries to reduce unneeded work/memory allocations.
- Handles datasets much larger than your available RAM.
- Has an API that is consistent and predictable.
- Has a strict schema (data-types should be known before running the query).

Polars vs Pandas

ITINERIS

Let’s how we perform the same operations on Pandas and Polars. We will work with data on flight delay:

```
from pathlib import Path
from zipfile import ZipFile
import requests

data_dir = Path("./sample_data") # replace this with a directory of your choice
dest = data_dir / "flights.csv.zip"

if not dest.exists():
    r = requests.get(

"https://transtats.bts.gov/PREZIP/On_Time_Reporting_Carrier_On_Time_Performance_1987_present_2022_1.zip",
        verify=False,
        stream=True,
    )
    data_dir.mkdir(exist_ok=True)
    with dest.open("wb") as f:
        for chunk in r.iter_content(chunk_size=102400):
            if chunk:
                f.write(chunk)

    with ZipFile(dest) as zf:
        zf.extract(zf.filelist[0].filename, path=data_dir)

extracted = data_dir / "On_Time_Reporting_Carrier_On_Time_Performance_(1987_present)_2022_1.csv "
```

Pl vs Pd: read csv files

ITINERIS

Pandas

```
pd.options.display.max_rows = 5
%timeit -n 1 -r 1 df_pd =
pd.read_csv(extracted)
df_pd
```

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

	Year	Quarter	Month	DayofMonth	DayOfWeek	FlightDate	Reporting_Airline	DOT_ID_Reporting_
0	2022	1	1	14	5	2022-01-14	YX	
1	2022	1	1	15	6	2022-01-15	YX	
...	...	...	...	...	...	...	...	
537900	2022	1	1	6	4	2022-01-06	DL	
537901	2022	1	1	6	4	2022-01-06	DL	

537902 rows × 110 columns

Polars

```
pl.Config.set_tbl_rows(5) # don't print too
many rows in the book
%timeit -n 1 -r 1 pl.read_csv(extracted,
truncate_ragged_lines=True)
df_pl = pl.read_csv(extracted,
truncate_ragged_lines=True)
df_pl
```

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

shape: (537_902, 110)

Year	Quarter	Month	DayofMonth	DayOfWeek	FlightDate	Reporting_Airline	DOT_ID_Reporting_Airline
i64	i64	i64	i64	i64	str	str	i64
2022	1	1	14	5	"2022-01-14"	"YX"	20452
2022	1	1	15	6	"2022-01-15"	"YX"	20452
2022	1	1	16	7	"2022-01-16"	"YX"	20452
...	...	...	...	...	...	...	...
2022	1	1	6	4	"2022-01-06"	"DL"	19790
2022	1	1	6	4	"2022-01-06"	"DL"	19790

Pl vs Pd: indexing

ITINERIS

Pandas

Pandas uses a special index type that is quite powerful for selecting rows and columns but is also very complicated.

```
%timeit -n 1 -r 1 df_pd.loc[12:15, ["Dest", "Tail_Number"]]
df_pd.loc[12:15, ["Dest", "Tail_Number"]]
```

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

	Dest	Tail_Number
12	DCA	N132HQ
13	DCA	N109HQ
14	DCA	N421YX
15	DCA	N137HQ

Polars

Polars avoids this complexity by simply not having an index. It just has ordinary methods like .select, .filter and .head for accessing a subset of rows or columns.

```
%timeit -n 1 -r 1 df_pl.select(["Dest", "Tail_Number"]).head(16).tail(4)
df_pl.select(["Dest", "Tail_Number"]).head(16).tail(4)
```

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

Polars supports also the square bracket syntax (the way Pd does it)

shape: (4, 2)

Dest	Tail_Number
str	str
"DCA"	"N132HQ"
"DCA"	"N109HQ"
"DCA"	"N421YX"
"DCA"	"N137HQ"

Pl vs Pd: index filtering

Pandas

We set a column as index and search for specific values

```
%timeit -n 1 -r 1
df_pd.set_index("IATA_CODE_Reporting_Airline").l
oc[['AA', 'DL'], ["Dest", "Tail_Number"]]
(

df_pd.set_index("IATA_CODE_Reporting_Airline")
    .loc[['AA', 'DL'], ["Dest", "Tail_Number"]]
)
```

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

IATA_CODE_Reporting_Airline	Dest	Tail_Number
AA	LAX	N106NN
AA	LAX	N112AN
...	...	...
DL	ATL	N989AT
DL	PDX	N815DN

138363 rows × 2 columns

Polars

We just use .filter

```
%timeit -n 1 -r 1 df_pl.select(["Dest",
"Tail_Number"]).head(16).tail(4)
df_pl.select(["Dest",
"Tail_Number"]).head(16).tail(4)
```

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

shape: (138_363, 3)

IATA_CODE_Reporting_Airline	Dest	Tail_Number
str	str	str
"DL"	"LGA"	"N315DN"
"DL"	"FLL"	"N545US"
"DL"	"ATL"	"N545US"
...	...	...
"DL"	"ATL"	"N989AT"
"DL"	"PDX"	"N815DN"

Pl vs Pd: filtering and modifying

Pandas

Here we do it with loc, but there are multiple ways to do it

```
f = pd.DataFrame({'a': [1,2,3,4,5], 'b':
[10,20,30,40,50]})
%timeit -n 1 -r 1 f.loc[f['a'] <= 3, "b"]
= f['b'] // 10
f
```

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

	a	b
0	1	1
1	2	2
2	3	3
3	4	40
4	5	50

Polars

```
f = pl.DataFrame({'a': [1,2,3,4,5], 'b':
[10,20,30,40,50]})

%timeit -n 1 -r 1 f.with_columns(pl.when(pl.col("a")
<= 3).then(pl.col("b") // 10).otherwise(pl.col("b")))

f.with_columns(
    pl.when(pl.col("a") <= 3)
    .then(pl.col("b") // 10)
    .otherwise(pl.col("b"))
)
```

Similar to an if block

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

shape: (5, 2)

a	b
i64	i64
1	1
2	2
3	3
4	40
5	50

Pl vs Pd: method chaining

ITINERIS

Method chaining is the capability to link multiple methods and/or functions by only specifying them one after the other, without nesting them. For example:

```
thing.min().abs().str()
```

instead of

```
str(abs(min(thing)))
```

Let’s how Polars take advantage of this approach in respect to Pandas by applying it to “clean” our data.

Pl vs Pd: method chaining

Pandas

```
def extract_city_name_pd(df: pd.DataFrame) ->
pd.DataFrame:
    """
    Chicago, IL -> Chicago for OriginCityName and
DestCityName
    """
    cols = ["OriginCityName", "DestCityName"]
    return df.assign(**{col: df[col].str.split(",",
regex=False).str[0] for col in cols})
```

Polars

```
def extract_city_name_pl() -> pl.Expr:
    """
    Chicago, IL -> Chicago for
OriginCityName and DestCityName
    """
    cols = ["OriginCityName",
"DestCityName"]
    return
pl.col(cols).str.split(",").list.get(0)
```

pl.Expr are objects that are more convenient to process

Note how much easier is in Polars:

- Not creating new columns (expressions);
- Passing all the columns at once (no for)
- Supports lists (no str[0])

Pl vs Pd: method chaining

Pandas

```
def time_col_pd(col: str, df: pd.DataFrame) -> pd.Series:
    timepart = df[col].replace("2400", "0000")
    return pd.to_datetime(df["FlightDate"] + ' ' +
                          timepart.str.slice(0, 2) + ':' +
                          timepart.str.slice(2, 4), #time formatting to HH:MM
                          errors='coerce' # when a time isnt present, insert NaT (not a Time)
                          )
```

Polars

pl.Time is a dtype not present in Pandas

```
def time_col_pl(col: str) -> pl.Expr:
    col_expr = pl.col(col)
    return (
        pl.when(col_expr == "2400")
        .then(pl.lit("0000"))
        .otherwise(col_expr)
        .str.strptime(pl.Time, "%H%M", strict=True) # convert the string into a pl.Time var
        .alias(col) # replace columns
    )
```

Pl vs Pd: method chaining

ITINERIS

Pandas

```
def time_to_datetime_pd(df: pd.DataFrame, columns: list[str]) -> pd.DataFrame:
    '''
    Combine all time items into datetimes.

    2014-01-01,0914 -> 2014-01-01 09:14:00
    '''
    return df.assign(**{col: time_col_pd(col, df) for col in columns})
```

Polars

```
def time_to_datetime_pl(columns: list[str]) -> list[pl.Expr]:
    """
    Combine all time items into datetimes.
    2014-01-01,0914 -> 2014-01-01 09:14:00
    """
    date_col = pl.col("FlightDate") # it will be a pl.Date variable
    return [
        date_col
        .dt.combine(time_col_pl(col)) # combine a pl.Date variable with a pl.Time var
        .alias(col)
        for col in columns
    ]
```

pl.Date is a dtype not present in Pandas

Pl vs Pd: method chaining

Now let’s start to put it all together. First we define some useful variables for both Pandas and Polars case:

```
category_cols = [
    "Dest",
    "Tail_Number",
    "IATA_CODE_Reporting_Airline",
    "CancellationCode",
]
time_cols = ["DepTime", "ArrTime", "CRSArrTime", "CRSDepTime"]
cols = (
    category_cols
    + time_cols
    + [
        "FlightDate",
        "Flight_Number_Reporting_Airline",
        "OriginCityName",
        "DestCityName",
        "Origin",
        "DepDelay",
    ]
)
```

Pl vs Pd: method chaining

ITINERIS

We extract the data from the csv and apply our functions:

Pandas

```
dtypes_pd = (
    {col: pd.CategoricalDtype() for col
in category_cols}
    | {col: pd.StringDtype() for col in
time_cols}
)
df_pd = (
    pd.read_csv(extracted,
dtype=dtypes_pd, usecols=cols,
na_values="")
    .pipe(extract_city_name_pd)
    .pipe(time_to_datetime_pd,
time_cols)
    .assign(FlightDate=lambda df:
pd.to_datetime(df["FlightDate"]))
)
df_pd[cols].head()
```

Polars

```
dtypes_pl = (
    {col: pl.Categorical for col in
category_cols}
    | {"FlightDate": pl.Date}
    | {col: pl.Utf8 for col in
time_cols}
)
df_pl = (
    pl.scan_csv(extracted,
schema_overrides=dtypes_pl,
null_values="")
    .select(cols)
    .with_columns([extract_city_name_pl
(), *time_to_datetime_pl(time_cols)])
    .collect()
)
df_pl.head()
```

Pl vs Pd: method chaining and plots

Pandas

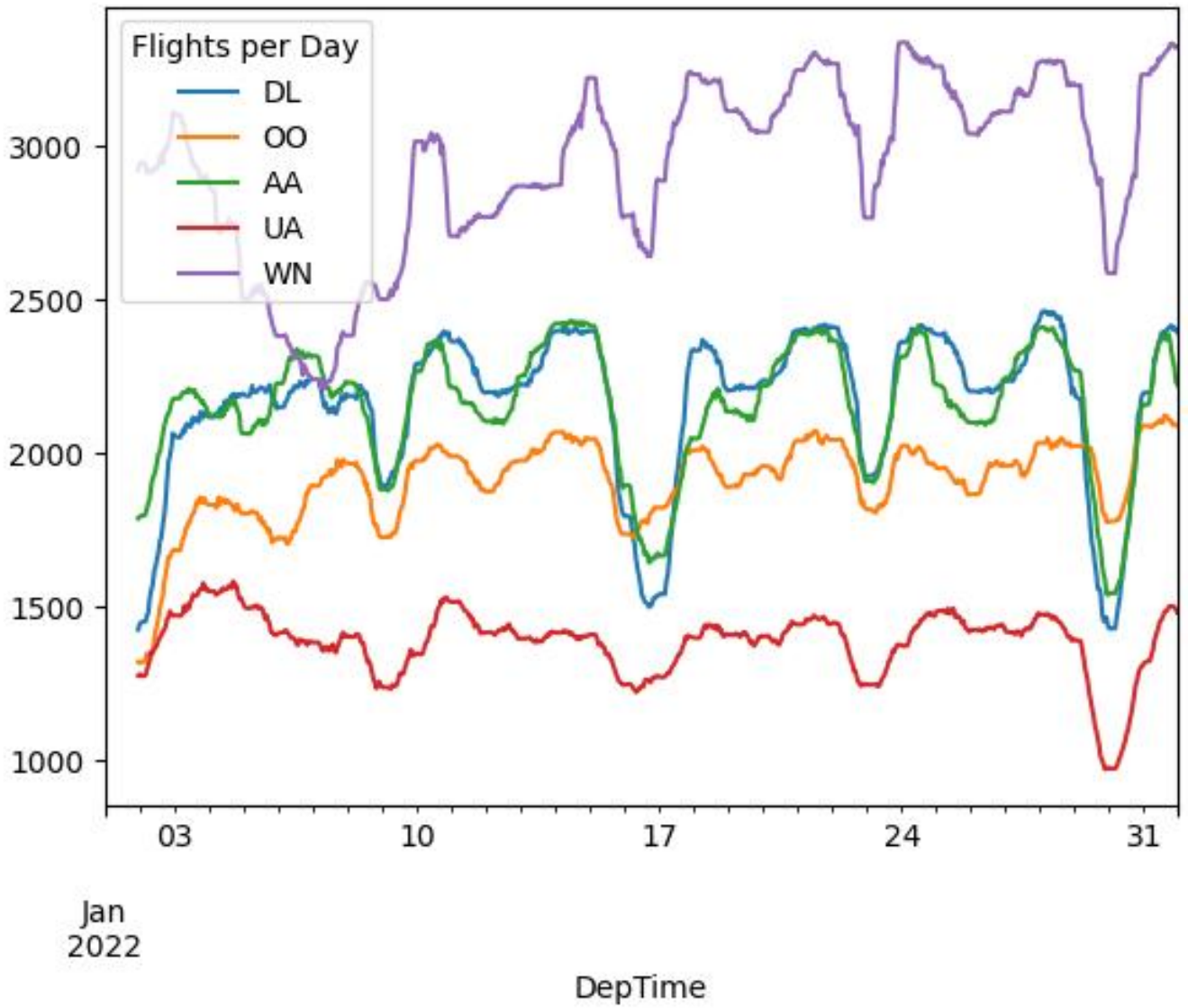
```
(
    df_pd
    .dropna(subset=["DepTime", "IATA_CODE_Reporting_Airline"]) #dropping any not a value in the df
    # filter for the busiest airlines and take the first 5
    .loc[
        lambda x:
x["IATA_CODE_Reporting_Airline"].isin(x["IATA_CODE_Reporting_Airline"].value_counts().index[:5])
    ]
    .assign(
        IATA_CODE_Reporting_Airline=lambda x:
x[ "IATA_CODE_Reporting_Airline"].cat.remove_unused_categories() )
    .set_index("DepTime")
    # TimeGrouper to resample & groupby at once
    .groupby(["IATA_CODE_Reporting_Airline", pd.Grouper(freq="h")])[ #group at hour level
        "Flight_Number_Reporting_Airline" # select the column
    ]
    .count()
    # the .pivot takes care of this in the Polars code.
    .unstack(0)
    .fillna(0)
    .rolling(24).sum()          # )  #take the 24 hour values and sum it
    .rename_axis("Flights per Day", axis=1)
    .plot()
)
```

Pl vs Pd: method chaining and plots

Polars

```
# filter for the busiest airlines
filter_expr =
pl.col("IATA_CODE_Reporting_Airline").is_in(pl.col("IATA_CODE_Reporting_Airline".value_counts(sort=True)
    .struct.field("IATA_CODE_Reporting_Airline") #value_counts return a Struct Type Variable
    .head(5)
)
(
    df_pl .drop_nulls(subset=["DepTime", "IATA_CODE_Reporting_Airline"]) #dropping null values
    .filter(filter_expr) #applying filter expression
    .sort("DepTime")
    .group_by_dynamic("DepTime",every="1h",group_by="IATA_CODE_Reporting_Airline") #grouping by airline at
hour level
    .agg(pl.col("Flight_Number_Reporting_Airline").count()) #aggregation by sum
    .pivot(
        index="DepTime", on="IATA_CODE_Reporting_Airline",values="Flight_Number_Reporting_Airline",
    ) #same as unstuck: DepTime became columns, airlines as rows, and flight number as element
    .sort("DepTime")
    # fill every missing hour with 0 so the plot looks better
    .upsample(time_column="DepTime", every="1h").fill_null(0)
    .select([pl.col("DepTime"), pl.col(pl.UInt32).rolling_sum(24)]) #select alle the columns with numeric
values to be summed
    .to_pandas() #easier to plot
    .set_index("DepTime").rename_axis("Flights per Day", axis=1) .plot()
)
```

Pl vs Pd: method chaining and plots



Exercise

Produce a boxplot (sns.boxplot(use x=<column>,y=<column>,data=<dataframe>) having on the x axis the hour of DepTime and on the y axis the DepDelay (Delay of departure). You can select the Deptime hour with the method dt.hour, for example <your_date>.dt.hour().

Solution

polars

```
import seaborn as sns
delay_pl = (
    df_pl.select(
        pl.col(
            ["DepTime", "DepDelay"],
        )
    )
    .drop_nulls()
    #.filter(pl.col("DepDelay").is_between(5, 600,
closed="none"))
    .with_columns(pl.col("DepTime").dt.hour())
)

sns.boxplot(x="DepTime", y="DepDelay", data=delay_pl)
```

Solution

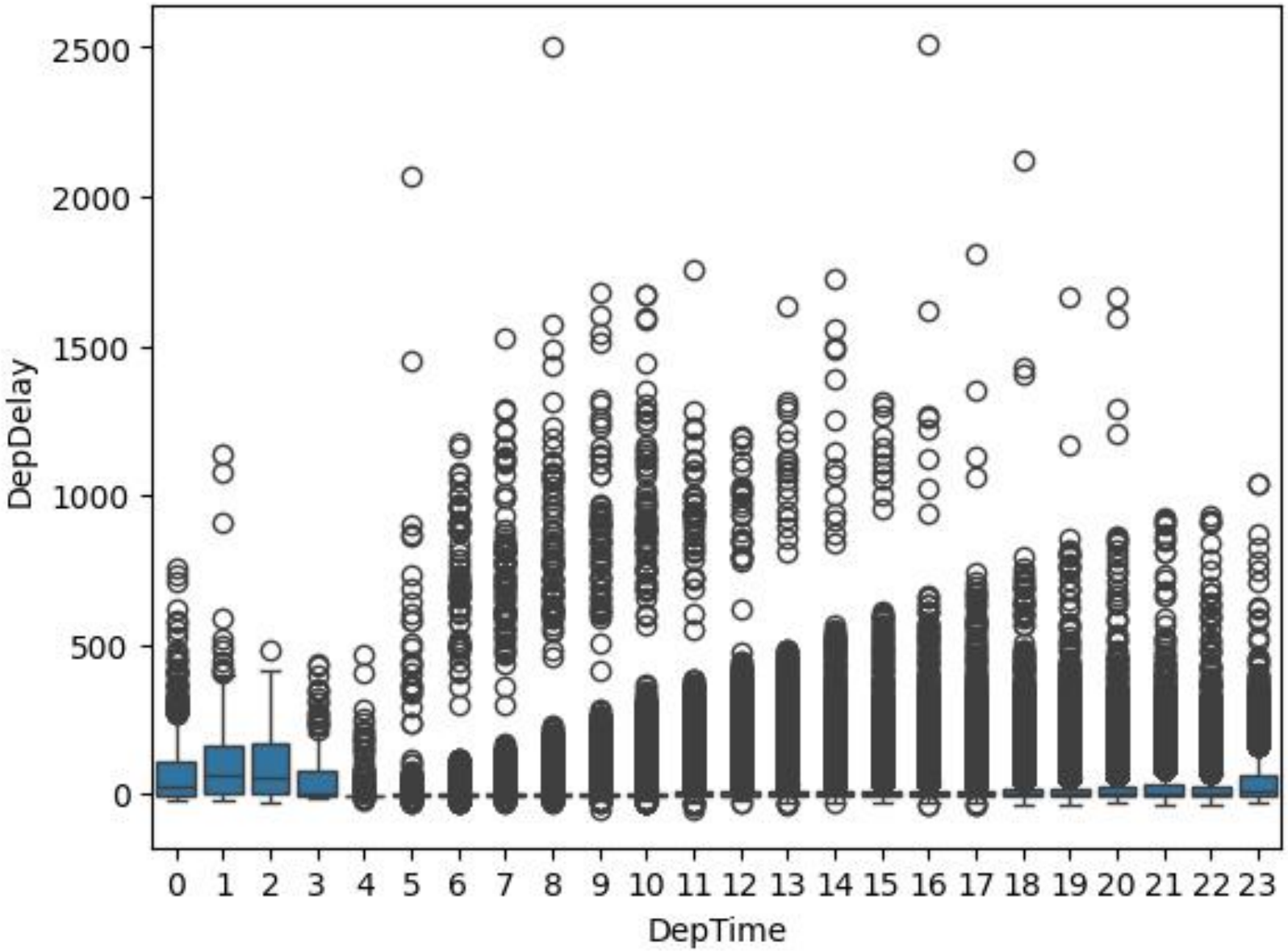
pandas

```
import seaborn as sns

delay_pd = (
    df_pd[[
        "DepTime",
        "DepDelay"
    ]]
    .dropna()
    #.loc[lambda x: x["DepDelay"] < 500]
    .assign(DepTime=lambda df: df["DepTime"].dt.hour)
    )

sns.boxplot(x="DepTime", y="DepDelay", data=delay_pd)
```

Solution


2500
2000
1500
1000
500
0
DepDelay
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23
DepTime

Pl vs Pd: tidying data

Tidy data [1] can be defined as data in which:

- Each variables forms a column;
- Each observation forms a row;
- Each set of observations forms a table

Let’s how to tidy up a dataset with Pandas and Polars

[1] Wickham, H. . (2014). Tidy Data. *Journal of Statistical Software*, *59*(10), 1–23. https://doi.org/10.18637/jss.v059.i10

Pl vs Pd: tidying data

Let’s use a dataset containing NBA games’ information

```
from pathlib import Path

nba_dir = Path("./sample_data/nba/")
column_names = {"Date": "date", "Visitor/Neutral": "away_team", "PTS": "away_points", "Home/Neutral":
"home_team",
"PTS.1": "home_points",
}

if not nba_dir.exists():
    nba_dir.mkdir()
    for month in (
        "october", "november", "december", "january", "february", "march", "april", "may", "june",
    ):
        url = f"http://www.basketball-reference.com/leagues/NBA_2016_games-{month}.html"
        tables = pd.read_html(url)
        raw = (
            pl.from_pandas(tables[0].query("Date !=
'Playoffs'")).rename(column_names).select(column_names.values())
        )
        raw.write_csv(nba_dir / f"{month}.csv")

nba_glob = nba_dir / "*.csv"
pl.scan_csv(nba_glob).head().collect()
```

Pl vs Pd: tidying data

Clean up a bit

Pandas

```
games_pd = (
    pl.read_csv(nba_glob)
    .to_pandas()
    .dropna(how="all")
    .assign(date=lambda x:
pd.to_datetime(x["date"], format="%a,
%b %d, %Y"))
    .sort_values("date")
    .reset_index(drop=True)
    .set_index("date", append=True)
    .rename_axis(["game_id", "date"])
    .sort_index()
)
games_pd.head()
```

Polars

```
games_pl = (
    pl.scan_csv(nba_glob)
    .with_columns(

pl.col("date").str.strptime(pl.Date,
"%a, %b %d, %Y"),
    )
    .sort("date")
    .with_row_index("game_id")
)
games_pl.head().collect()
```

Pl vs Pd: tidying data

We want to calculate the rest days between two games for each team. Data can be prepared to make the job easier:

Pandas

```
tidy_pd = (
    games_pd.reset_index().melt(
        id_vars=["game_id", "date"],
        value_vars=["away_team", "home_team"],
        value_name="team",
    )
    .sort_values("game_id").assign(rest=lambda df: (
            df.sort_values("date").groupby("team")["date"].diff() #difference
between elements in the same group
           .dt.days.sub(1) )
    )
    .dropna(subset=["rest"])
    .astype({"rest": pd.Int8Dtype()})
)
tidy_pd
```

Melt unifies two columns in a single one ordering using the index given in the first argument (inverse of pivot). It also create a column in which it is stored the name of the original columns.

Pl vs Pd: tidying data

ITINERIS

We want to calculate the rest days between two games for each team. Data can be prepared to make the job easier:

Polars

```
tidy_pl = (
    games_pl
    .unpivot(
        index=["game_id", "date"],
        on=["away_team", "home_team"],
        value_name="team",
    )
    .sort("game_id").with_columns((
        pl.col("date")
        .alias("rest")
        .diff().over("team")
        .dt.total_days() - 1).cast(pl.Int8))
    .drop_nulls("rest")
    .collect()
)
tidy_pl
```

Unpivot unifies two columns in a single one ordering using the index given in the first argument (inverse of pivot). It also create a column in which it is stored the name of the original columns.

Pl vs Pd: tidying data

We want to add this new column on the original dataset. We also add columns for the spread between the home team's rest and away team's rest, and a flag for whether the home team won.

Pandas

Pivot create new columns from the values in an existing column using the index given in the first argument and populating them with the argument values.

```
by_game_pd = (
    tidy_pd
.pivot(index=["game_id", "date"], columns="variable", values="rest"),
    .rename(
        columns={"away_team": "away_rest", "home_team": "home_rest"} #rename the
column correctly
    )
)
# join the original dataframe with the pivoted one using game_id and date
joined_pd = by_game_pd.join(games_pd).assign(
    home_win=lambda df: df["home_points"] > df["away_points"],
    rest_spread=lambda df: df["home_rest"] - df["away_rest"],
)
joined_pd
```

Pl vs Pd: tidying data

We want to add this new column on the original dataset. We also add columns for the spread between the home team's rest and away team's rest, and a flag for whether the home team won.

Polars

Pivot create new columns from the values in an existing column using the index given in the first argument and populating them with the argument values.

```
by_game_pl = (
    tidy_pl
    .pivot(index=["game_id", "date"], on="variable“, values="rest")
    .rename({"away_team": "away_rest", "home_team": "home_rest"})
)
# join the original dataframe with the pivoted one using game_id and date
joined_pl = (
    by_game_pl
    .join(games_pl.collect(), on=["game_id", "date"])
    .with_columns([
        pl.col("home_points").alias("home_win") > pl.col("away_points"),
        pl.col("home_rest").alias("rest_spread") - pl.col("away_rest"),
    ])
)
joined_pl
```

Parallelization

ITINERIS

Numba

Ray

Dask

Numba

Numba is a **just-in-time**, **type-specializing**,
Python **function compiler** that exposes a simple interface
for **accelerating numerically-focused** Python functions.

Numba

- **Function compiler**: Numba compiles Python functions in order to make them faster;
- **Type-specializing**: Numba needs specification of the data on which is working to generate a fast implementation;
- **Just-in-time**: Numba process functions the first time they are called;
- **Numerically-focused**: Numba is focused on numeric data types, making it useful also when working with GPUs. Usually numpy arrays are used;

Numba: CPU compilation

The Numba compiler is applied to functions using a **decorator.** Decorators are function modifiers that use a simple syntax. Let's look at how it is done in Numba:

```
from numba import jit
import math

# This is the function decorator syntax and is equivalent to `hypot =
jit(hypot)`.
# The Numba compiler is just a function you can call whenever you want!
@jit
def hypot(x, y):
    x = abs(x);
    y = abs(y);
    t = min(x, y);
    x = max(x, y);
    t = t / x;
    return x * math.sqrt(1+t*t)
```

Numba: CPU compilation

When we call this function as it follows, Numba will compile it for float variables:

```
hypot(3.0, 4.0)
```

```
5.0
```

The original Python implementation is saved in the attribute `py_func`, in order to be able to check if the Numba result is correct:

```
hypot.py_func(3.0, 4.0)
```

```
5.0
```

Numba: Performance

Simple Python function

```
%timeit hypot.py_func(3.0, 4.0)
```

```
684 ns ± 203 ns per loop (mean ± std. dev. of 7 runs, 1000000 loops each)
```

Numba

```
%timeit hypot(3.0, 4.0)
```

```
413 ns ± 93.9 ns per loop (mean ± std. dev. of 7 runs, 1000000 loops each)
```

Math function

```
%timeit math.hypot(3.0, 4.0)
```

```
118 ns ± 24 ns per loop (mean ± std. dev. of 7 runs, 10000000 loops each)
```

Exercise

Here is a function that calculates the value of pi using the Monte Carlo method. Modify it in order to use Numba and compare the performances.

```
import random

def monte_carlo_pi(nsamples):
    acc = 0
    for i in range(nsamples):
        x = random.random()
        y = random.random()
        if (x**2 + y**2) < 1.0:
            acc += 1
    return 4.0 * acc / nsamples
```

Solution

```
import random
from numba import jit

@jit
def monte_carlo_pi(nsamples):
    acc = 0
    for i in range(nsamples):
        x = random.random()
        y = random.random()
        if (x**2 + y**2) < 1.0:
            acc += 1
    return 4.0 * acc / nsamples
```

```
%timeit monte_carlo_pi(nsamples)
```

```
15.3 ms ± 3.32 ms per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

```
%timeit monte_carlo_pi.py_func(nsamples)
```

```
443 ms ± 101 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

Numba: Parallelization

When setting the argument parallel=True, Numba tries to perform the following actions:

- Tries to parallelize code, such as for loops and numerical computations;
- The parallelization is automatic;
- If the results are not quite good, we can perform the parallelization explicitly;
- Parallelization can lead to great improvement in performance, especially on large data and/or intensive computations;

Numba: Parallelization

```
@jit(parallel=True)
def monte_carlo_pi(nsamples):
    acc = 0
    for i in range(nsamples):
        x = random.random()
        y = random.random()
        if (x**2 + y**2) < 1.0:
            acc += 1
    return 4.0 * acc / nsamples
```

```
%timeit monte_carlo_pi(nsamples)
```

```
15.1 ms ± 3.11 ms per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

No difference. We get a warning about Numba not being able to parallelize effectively the code. Let’s do it explicitly.

Numba: Parallelization

```
from numba import njit, prange

@njit(parallel=True)
def monte_carlo_pi(nsamples):
    acc = 0
    for i in prange(nsamples):
        x = random.random()
        y = random.random()
        if (x**2 + y**2) < 1.0:
            acc += 1
    return 4.0 * acc / nsamples
```

```
%timeit monte_carlo_pi(nsamples)
```

```
12.2 ms ± 451 µs per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

We gained 3 ms, so we have a little improvement

Numba: GPU

As a first use of GPU, let’s how Numba compile Numpy ufuncs to use them with GPUs (Nvidia in our case). Ufuncs are a good candidate to use GPUs because they perform the same operation on all elements of the array (data parallel).
Let’s start by installing and miporting the needed packages.

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

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

Numba: GPU

We will use the `vectorize` decorator to tell to Numba to find the best way to perform the same operation on all the input elements

```
import numpy as np
from numba import vectorize
```

```
@vectorize
def add_ten(num):
    return num + 10 # This scalar operation will be performed on each
element
```

```
nums = np.arange(10)
add_ten(nums) # pass the whole array into the ufunc, it performs the
operation on each element
```

```
array([10, 11, 12, 13, 14, 15, 16, 17, 18, 19])
```

Numba: GPU

In order to use the GPU, we have to specify the type of variables we are passing to the ufunc and the type of the output it will produce. Moreover, we have to select the device on which the ufunc will run.

```
# [‘<output_type>(<inputs_type>)’], target=‘<device>’
@vectorize(['int64(int64, int64)'], target='cuda’)
def add_ufunc(x, y):
    return x + y
```

```
a = np.array([1, 2, 3, 4])
b = np.array([10, 20, 30, 40])

c = add_ufunc(a, b)
print("a + b = ", c)
```

```
a + b =  [11 22 33 44]
```

Numba: GPU

For us it was just a one-line difference, but Numba performed all these operations:

- Compiled a CUDA kernel function;
- Allocated GPU memory;
- Copied the input to GPU;
- Executed the kernel;
- Copied the data back to CPU;
- Produced the result as a Numpy array;

Numba on GPU: performance

Let’s compare our GPU kernel with a standard numpy CPU execution:

```
import time

@vectorize(['int64(int64, int64)'],
target='cuda')
def add_ufunc(x, y):
    return x + y

a = np.array([1, 2, 3, 4])
b = np.array([10, 20, 30, 40])

t0 = time.time()
c = add_ufunc(a, b)
t1 = time.time()
t_diff_01 = t1-t0
print("Numba: a + b = ", c, "\n
numba-timing: ", t_diff_01)
```

```
Numba: a + b =  [11 22 33 44]
 numba-timing:  0.05368733406066894
```

```
t2 = time.time()
d = np.add(a, b)
t3 = time.time()
t_diff_23 = t3-t2
print("Numpy: a + b = ", c, "\n
numpy-timing: ", t_diff_23)
```

```
Numpy: a + b =  [11 22 33 44]
 numpy-timing:  0.00012516975402832031
```

Numpy is faster!

Numba on GPU: performance

Why Numpy on CPU is faster? Simply we did not use well the GPU:

- Small inputs;
- Simple calculations;
- I/O overhead (CPU <-> GPU) ;
- Incorrect data type (too large);

Let’s try with another example

Numba on GPU: performance

CPU

```
import math
import numpy as np
from numba import vectorize
import time

SQRT_2PI = np.float32((2*math.pi)**0.5)  # Precompute this constant as a float32.  Numba will inline
it at compile time.

@vectorize(['float32(float32, float32, float32)'], target='cpu')
def gaussian_pdf_cpu(x, mean, sigma):
    '''Compute the value of a Gaussian probability density function at x with given mean and
sigma.'''
    return math.exp(-0.5 * ((x - mean) / sigma)**2) / (sigma * SQRT_2PI)

# Evaluate the Gaussian 100 million times!
x = np.random.uniform(-3, 3, size=1000000).astype(np.float32)
mean = np.float32(0.0)
sigma = np.float32(1.0)

t0 = time.time()
pd = gaussian_pdf_cpu(x, mean, sigma)
t1 = time.time()

t_diff = t1-t0
print("Gaussian probability density function at x: ", pd,"\ntimeing: ", t_diff)
```

```
Gaussian probability density function at x:
[0.00961851 0.37785605 0.1849463  ... 0.25389126
0.25591663 0.0700173 ]
timeing:  1.4956402778625488
```

Numba on GPU: performance

GPU

```
import math
import numpy as np
from numba import vectorize
import time

SQRT_2PI = np.float32((2*math.pi)**0.5)  # Precompute this constant as a float32.  Numba will inline
it at compile time.

@vectorize(['float32(float32, float32, float32)'], target='cuda')
def gaussian_pdf(x, mean, sigma):
    '''Compute the value of a Gaussian probability density function at x with given mean and
sigma.'''
    return math.exp(-0.5 * ((x - mean) / sigma)**2) / (sigma * SQRT_2PI)

# Evaluate the Gaussian 100 million times!
x = np.random.uniform(-3, 3, size=1000000).astype(np.float32)
mean = np.float32(0.0)
sigma = np.float32(1.0)

t0 = time.time()
pd = gaussian_pdf(x, mean, sigma)
t1 = time.time()

t_diff = t1-t0
print("Gaussian probability density function at x: ", pd,"\ntimeing: ", t_diff)
```

```
Gaussian probability density function at x:
[0.02292864... 0.33435765]
Timeing: 0.3975145816802978 5
```

Numba on GPU: Device Functions

In all the previous examples we used ufuncs. If we want to make our custom functions that are not inherently vectorized, we have to use a decorator:

```
@cuda.jit(device=True)
def polar_to_cartesian(rho, theta):
    x = rho * math.cos(theta)
    y = rho * math.sin(theta)
    return x, y

@vectorize(['float32(float32, float32, float32, float32)'], target='cuda')
def polar_distance(rho1, theta1, rho2, theta2):
    x1, y1 = polar_to_cartesian(rho1, theta1)
    x2, y2 = polar_to_cartesian(rho2, theta2)

    return ((x1 - x2)**2 + (y1 - y2)**2)**0.5
```

@cuda.jit indicates that the function is used inside a vectorized CUDA kernel. The device=True flag indicates that it can only be called from other functions running on GPUs.

Numba on GPU: Device Functions

```
n = 1000000
rho1 = np.random.uniform(0.5, 1.5, size=n).astype(np.float32)
theta1 = np.random.uniform(-np.pi, np.pi, size=n).astype(np.float32)
rho2 = np.random.uniform(0.5, 1.5, size=n).astype(np.float32)
theta2 = np.random.uniform(-np.pi, np.pi, size=n).astype(np.float32)

pol_dist = polar_distance(rho1, theta1, rho2, theta2)

print("polar_distance = ", pol_dist)
```

```
polar_distance =  [2.0788662  0.7716167  2.0064583  ... 0.80170226 1.4970344  1.5913861 ]
```

Exercise

Here is a function that removes “noise” (low-level frequencies) from waveforms.

```
def zero_suppress(waveform_value, threshold):
    if waveform_value < threshold:
        result = 0
    else:
        result = waveform_value
    return result
```

Decorate the function to run on CUDA and run it on the following data with a threshold of 15:

```
from matplotlib import pyplot as plt

n = 100000
noise = np.random.normal(size=n) * 3
pulses = np.maximum(np.sin(np.arange(n) / (n / 23)) - 0.3, 0.0)
waveform = ((pulses * 300) + noise).astype(np.int16)
plt.plot(waveform)
```

Solution

```
n = 100000
noise = np.random.normal(size=n) * 3
pulses = np.maximum(np.sin(np.arange(n) / (n / 23)) - 0.3, 0.0)
waveform = ((pulses * 300) + noise).astype(np.int16)

@vectorize(['float32(int16, float32)'], target='cuda')
def zero_suppress(waveform_value, threshold):
    if waveform_value < threshold:
        result = 0
    else:
        result = waveform_value
    return result

result = zero_suppress(waveform, 15)
print(result)
plt.plot(result)
```

Numba on GPU: Manage Memory

Until now, we leave the task to pass array to and from GPU. Each time numba performs an operation, the data is rebounding between CPU and GPU. This is very time expensive and most of the times we want to perform multiple operations on GPU, so there’s no need to bring the data back to GPU. It’ s so important that is also reported in the CUDA best practices:

> **Note**
>
> **High Priority:** Minimize data transfer between the host and the device, even if it means running some kernels on the device that do not show performance gains when compared with running them on the host CPU.

CUDA device arrays solve this problem. Device arrays will not be automatically transfered back to the host after processing, and can be reused as we wish on the device before ultimately, and only if necessary, sending them, or parts of them, back to the host.

Numba on GPU: Manage Memory

Let’s use our previous example:

```
@vectorize(['int64(int64, int64)'], target='cuda’)
def add_ufunc(x, y):
    return x + y
```

```
n = 100000
x = np.arange(n).astype(np.float32)
y = 2 * x
```

```
%timeit add_ufunc(x, y)
```

```
1.53 ms ± 479 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

This will be our time baseline

Numba on GPU: cuda module

The cuda module from Numba contains a method to move arrays from one device to another:

```
from numba import cuda

x_device = cuda.to_device(x)
y_device = cuda.to_device(y)

print(x_device)
print(x_device.shape)
print(x_device.dtype)
```

If you have multiple GPUs you can select which one you want to use by using cuda.select_device(0)

```
<numba.cuda.cudadrv.devicearray.DeviceNDArray object at 0x7e6a650be490>
(100000,)
float32
```

```
%timeit add_ufunc(x_device, y_device)
```

```
421 µs ± 103 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```

Faster! baseline = 1.53 ms

Numba on GPU: cuda module

We can make it faster. We still save the output first on a device array and then copying to the host (even if we don't save the result in a variable!). Let's create a device array directly on the GPU to save the output:

```
out_device = cuda.device_array(shape=(n,), dtype=np.float32)
```

we use the out argument to tell the ufunc where to save the output

```
%timeit add_ufunc(x_device, y_device, out=out_device)
```

```
190 µs ± 29.4 µs per loop (mean ± std. dev. of 7 runs, 10000 loops each)
```

Faster! before was 421 µs

now we can pass it back to host if we want

```
out_host = out_device.copy_to_host()
print(out_host[:10])
```

```
[ 0.  3.  6.  9. 12. 15. 18. 21. 24. 27.]
```

Exercise

Implement the FizzBuzz function to run on GPU using Numba. The FizzBuzz is a function that takes a number as input and returns “Fizz” if the number is a multiple of 3, “Buzz” if is a multiple of 5, “FizzBuzz” if the number is a multiple of both, it returns “None” in all other cases. You can substitute the strings with numbers and then print the string with a if cycle after you call the function. You have to use cuda.jit.
Run it on an array from 1 to 50’000’000 and print the highest instance of Fizz.

```
def fizz_buzz(input):
    ...
```

Solution

```
@vectorize(['int16(int16)'], target='cuda')
def fizz_buzz(input):
    if input % 15 == 0:
        result = 3 # Fizz Buzz
    elif input % 5 == 0:
        result = 2 # Buzz
    elif input % 3 == 0:
        result = 1 #Fizz
    else:
        result = 0 # None
    return result
```

```
data = np.arange(1, 50_000_000).astype(np.int16)
data_device = cuda.to_device(data)
out_device = cuda.device_array(shape=(len(data),), dtype=np.int16)
```

```
%%time
fizz_buzz(data, out=out_device)
```

```
CPU times: user 78.2 ms, sys: 11.4 ms, total: 89.6 ms
Wall time: 149 ms
```

Solution

```
out_host = out_device.copy_to_host()
print(out_host[:10])
```

```
[0 0 1 0 2 1 0 0 1 2]
```

```
out_string = np.empty(len(out_host), dtype="object")
for i in range(1,len(data)):
  if out_host[i] == 3:
    out_string[i] = "FizzBuzz"
  if out_host[i] == 2:
    out_string[i] = "Buzz"
  if out_host[i] == 1:
    out_string[i] = "Fizz"
  if out_host[i] == 0:
    out_string[i] = "None"
```

```
for i in range(1,len(data)):
  if out_string[-i] == "Fizz":
    print(len(data)-i)
    print(out_string[-i])
    break
```

```
49999998
Fizz
```

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