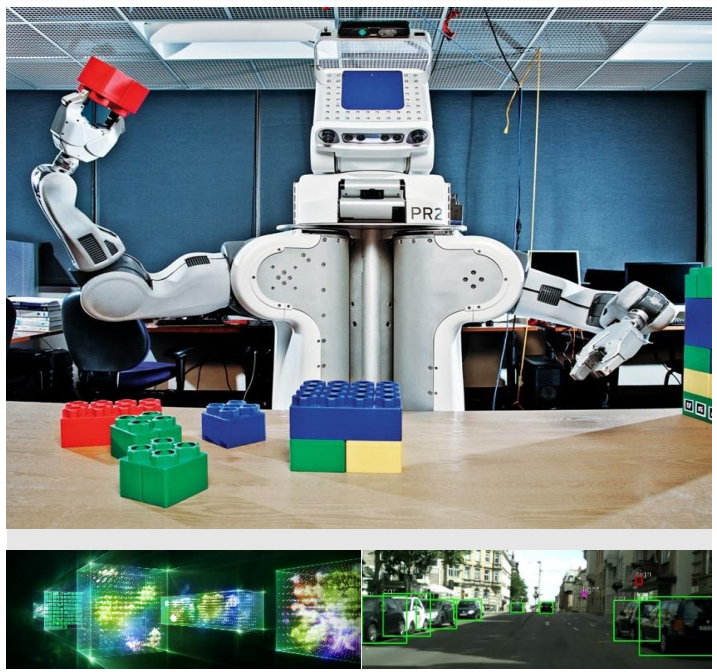
GPUS FOR DATA SCIENCE (RAPIDS)

Sangmoon Lee, sml@nvidia.com

NVIDIA
“THE AI COMPUTING COMPANY”

GPU Computing

Computer Graphics



Artificial Intelligence

What is RAPIDS

RAPIDS

Opens GPU for Data Science

RAPIDS is a set of open source libraries for GPU accelerating **data preparation** and **machine learning**.

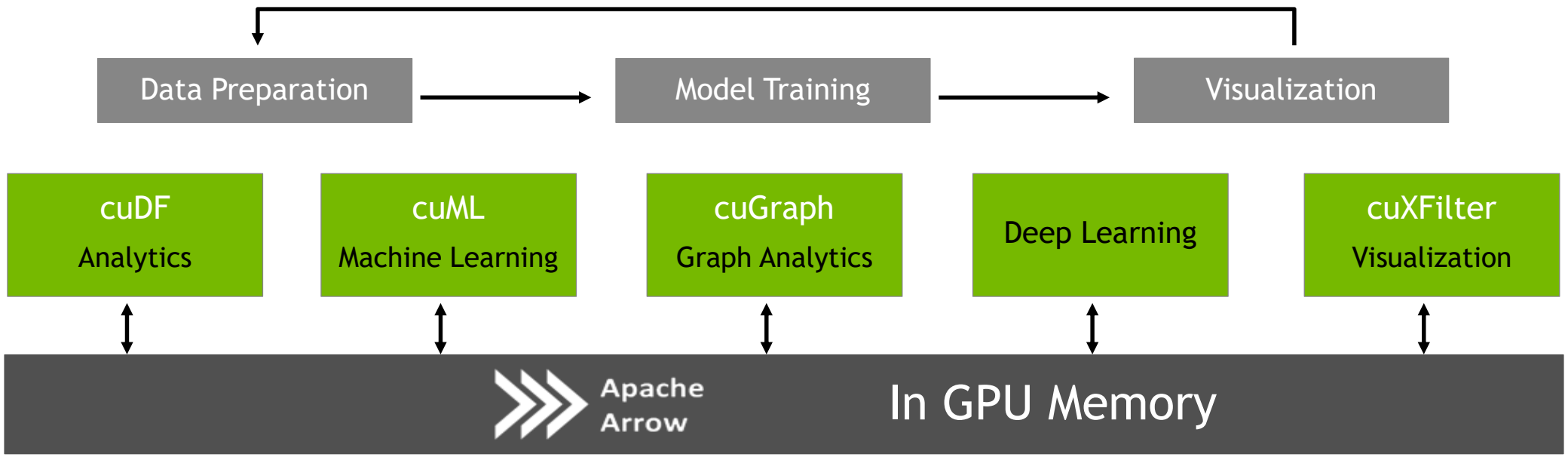
OSS website: http://www.rapids.ai/

RAPIDS: **R**apid **A**ccelerate **P**latform **I**ntegrated for **D**ata **S**cience

RAPIDS

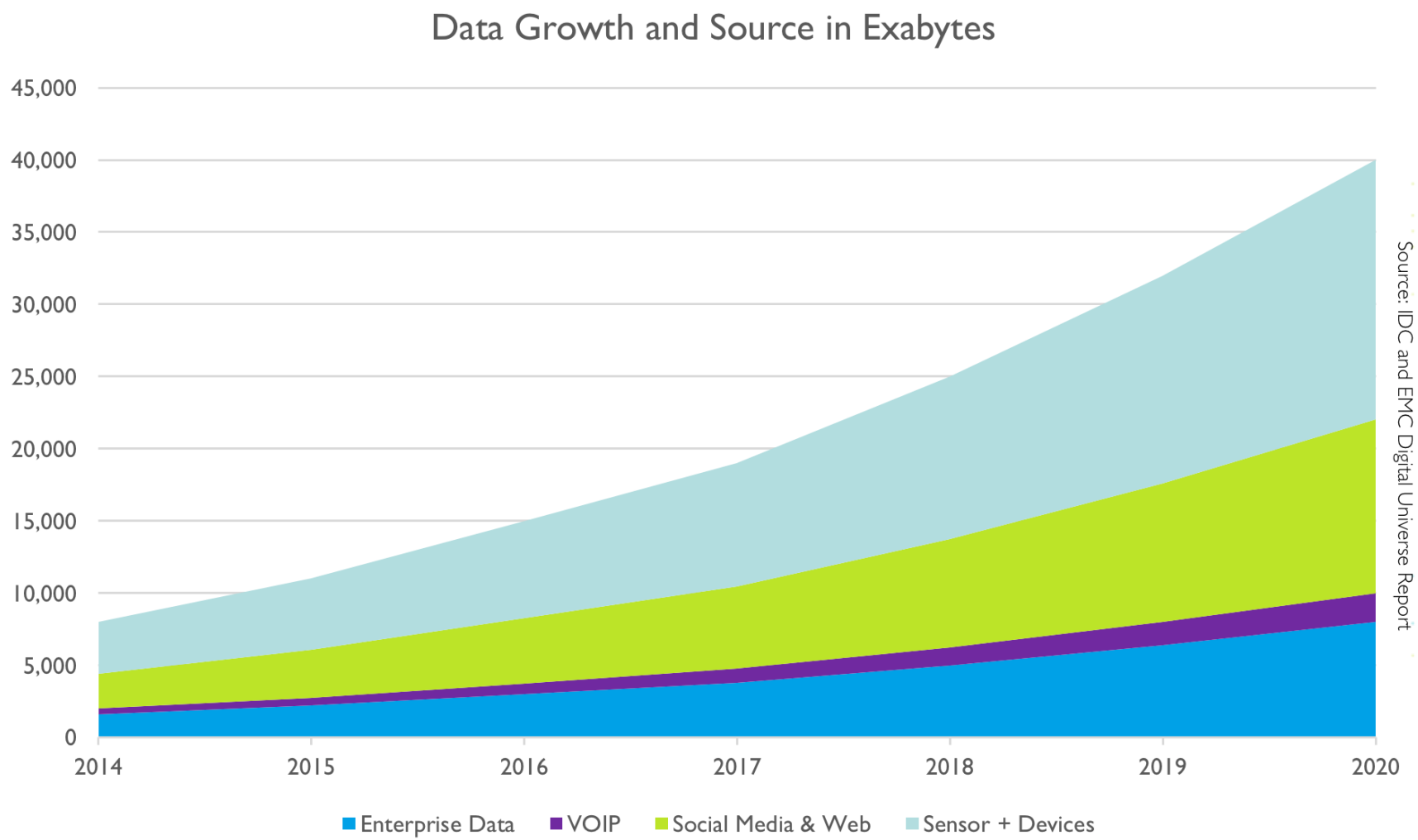
GPU Accelerated End-to-End Data Science pipeline

www.rapids.ai

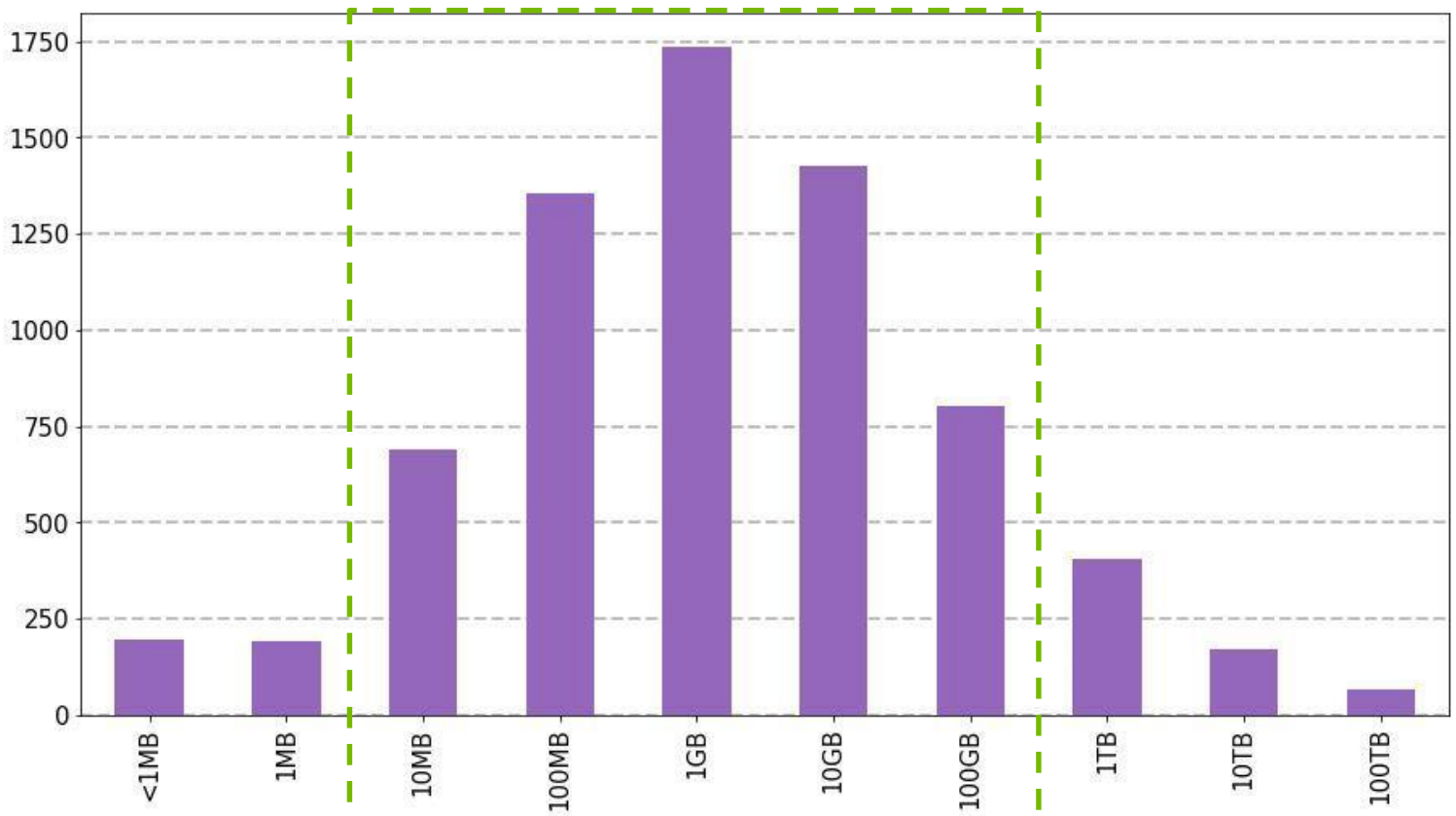


Why RAPIDS

REALITIES OF DATA



WHAT IS TYPICAL DATASET SIZE

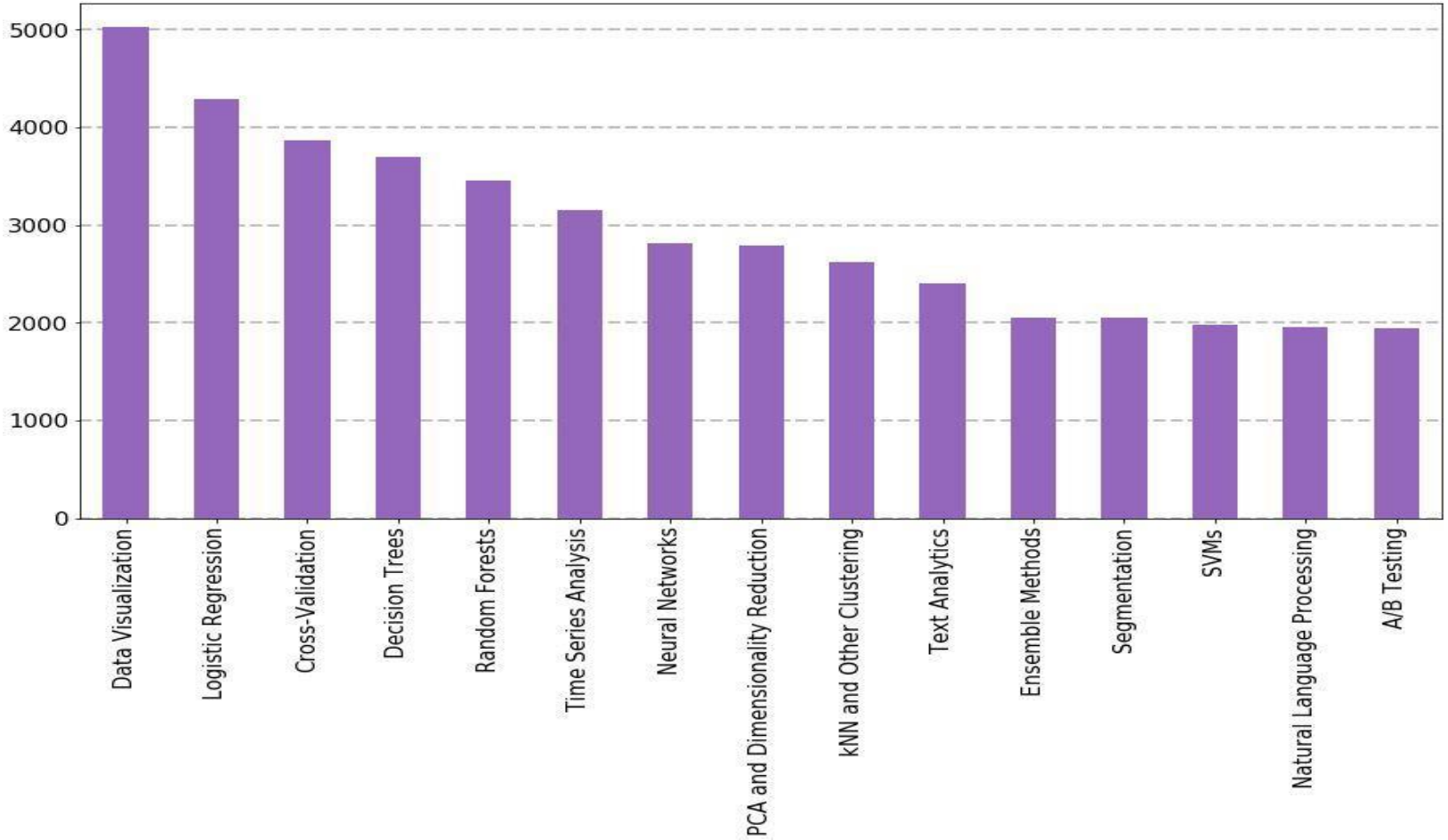


Faster, Repeat analysis

WHAT % OF TIME IS DEVOTED TO:

1. Data Visualization
2. Logistic Regression
3. Cross Validation
4. Decision Trees
5. Random Forests

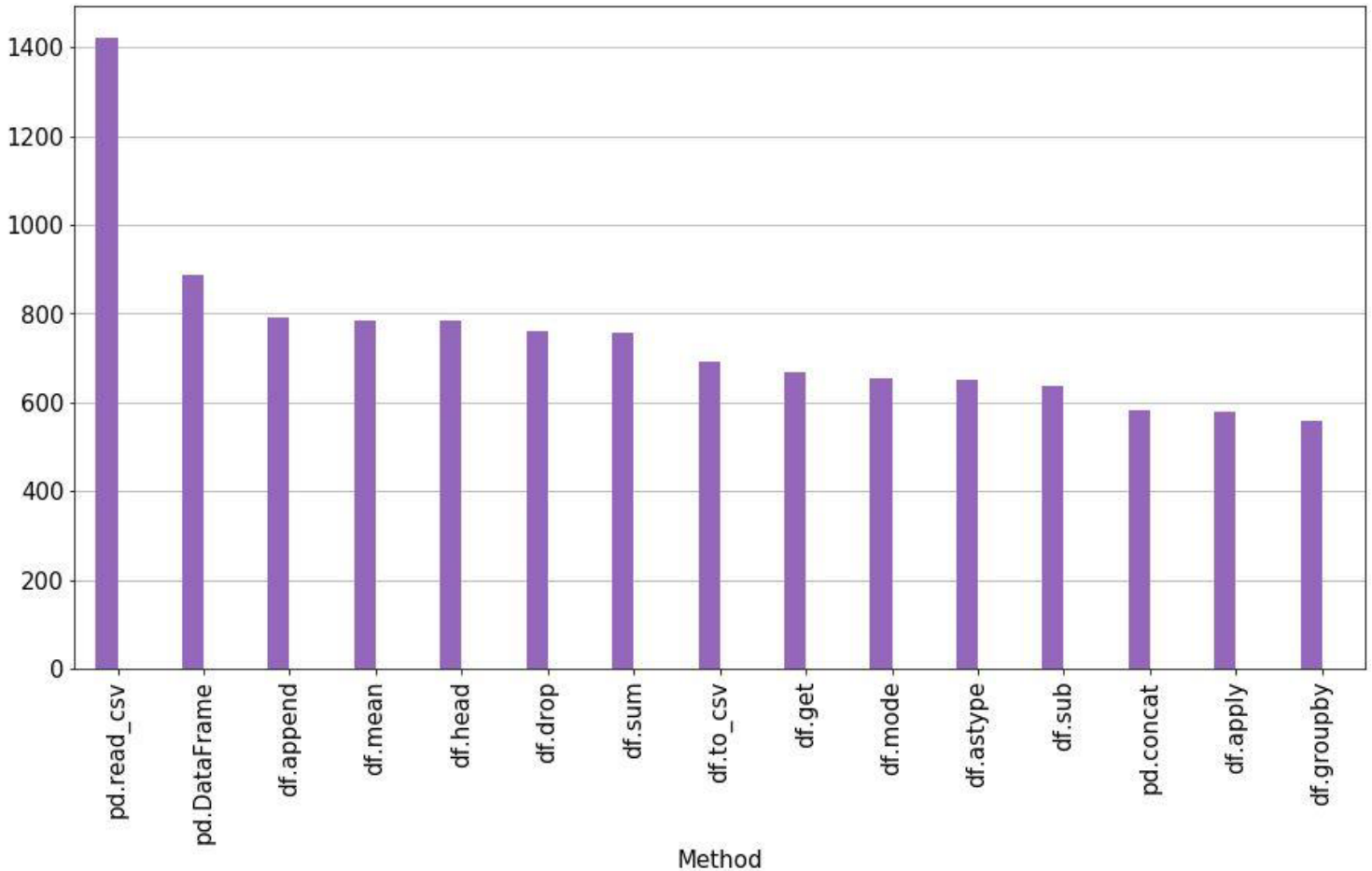
…



MOST USED DATA-FRAME METHODS

1. pd.read_csv
2. pd.DataFrame
3. df.append
4. df.mean
5. df.head
6. df.drop
7. df.sum
8. df.to_csv
9. df.get

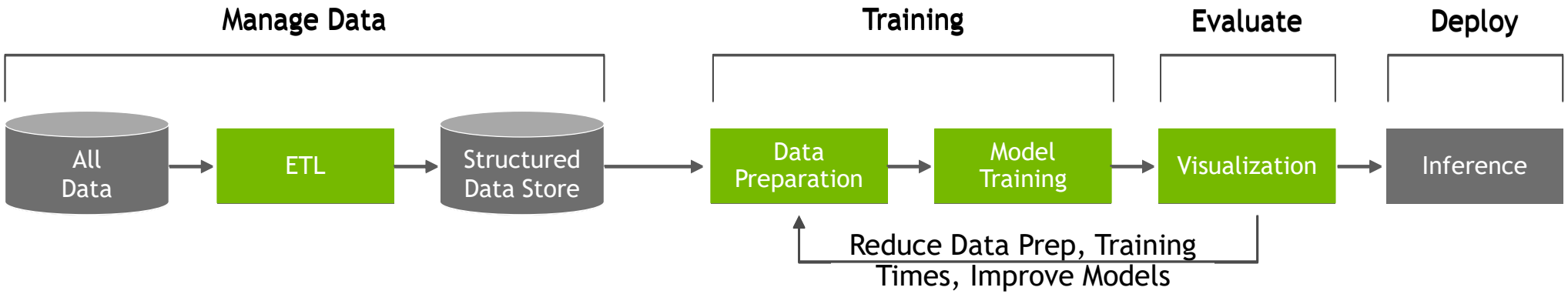
…

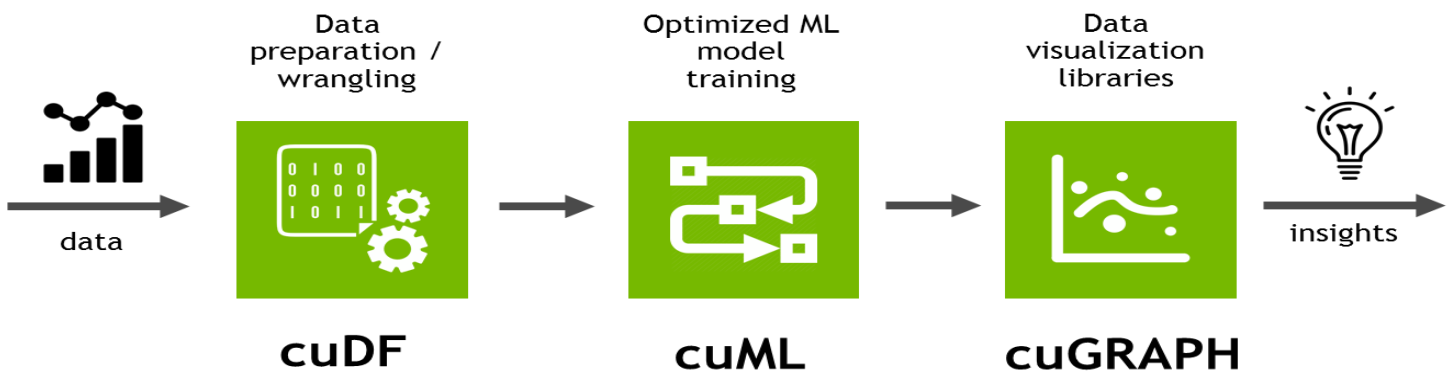


RAPIDS Benefits

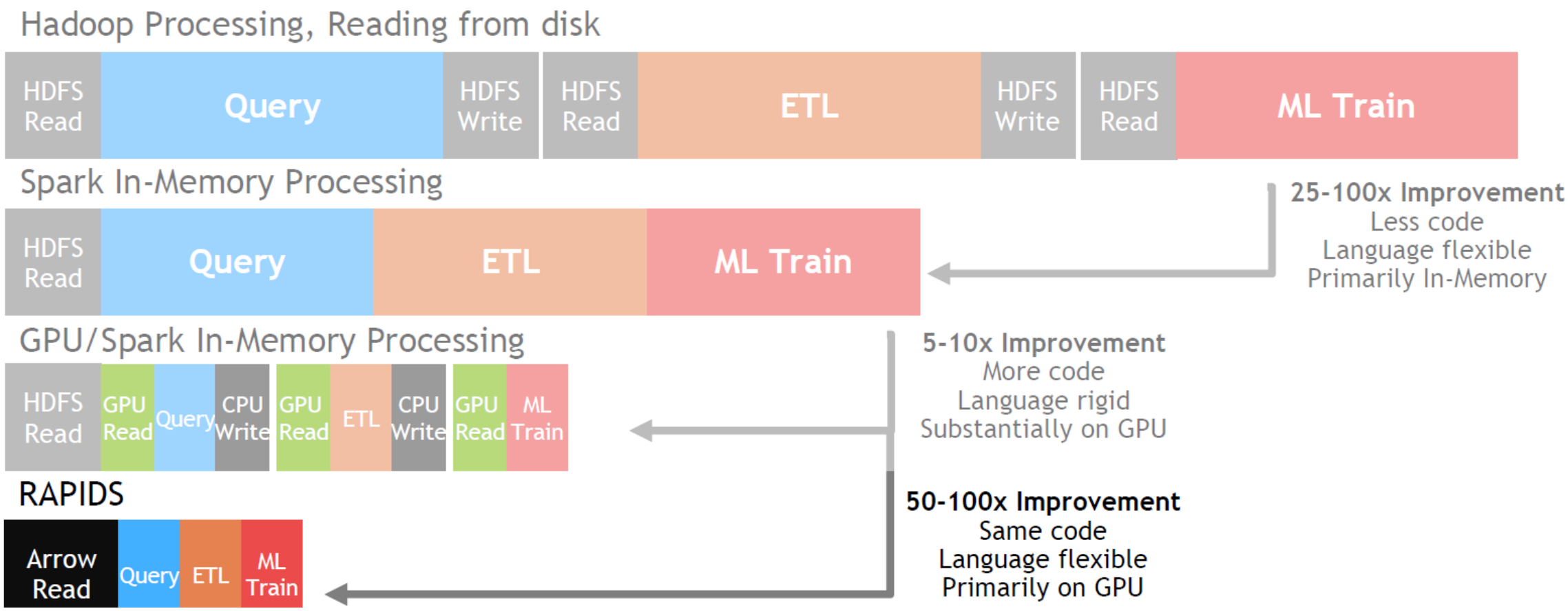
FASTER DATA SCIENCE WORKFLOW

Open Source, GPU-accelerated End-to-End Data Science



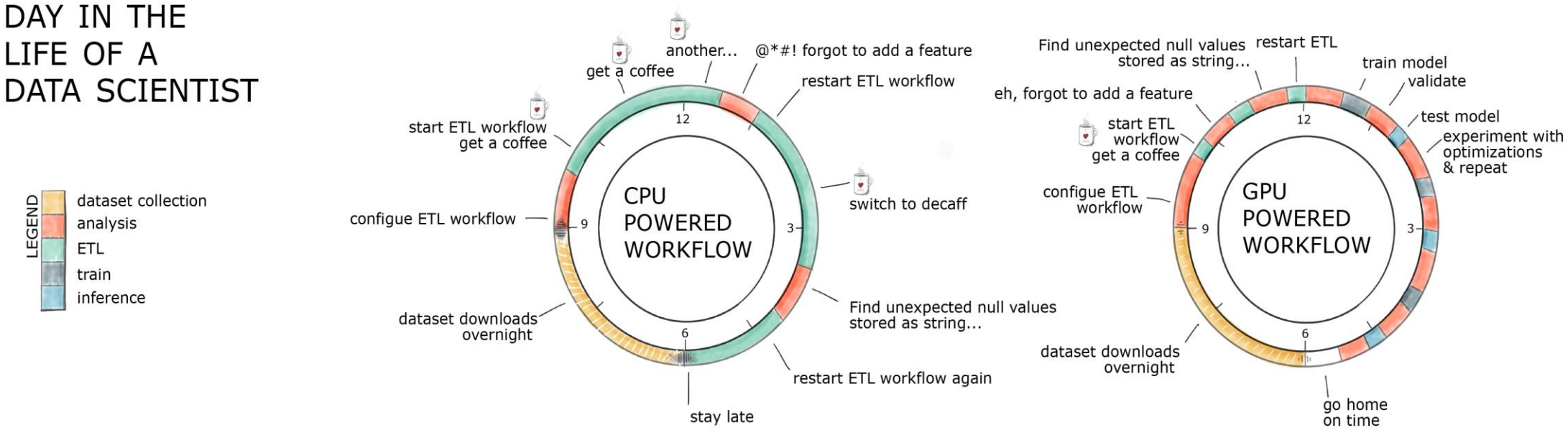


DATA PROCESSING EVOLUTION



DAY IN THE LIFE OF A DATA SCIENTIST

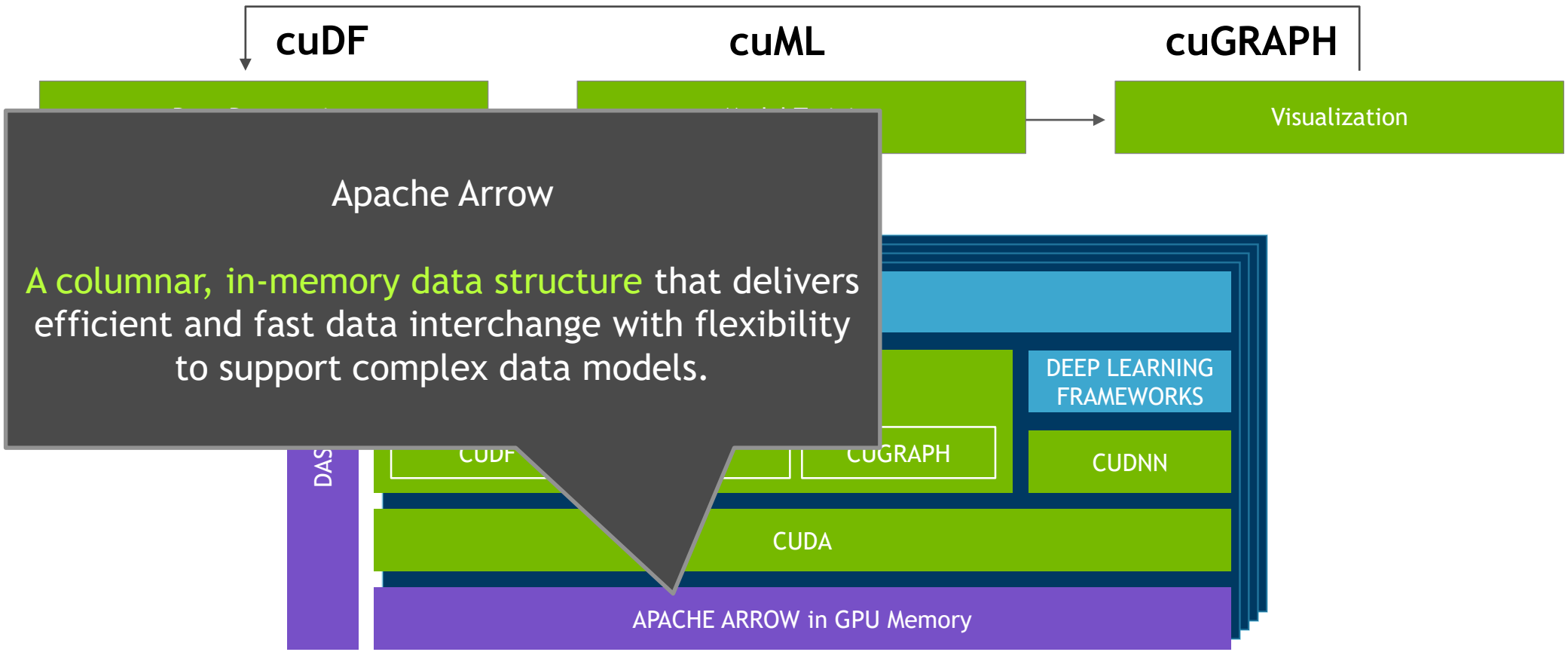
Faster Data Access with Less Data Movement



RAPIDS SW Stack

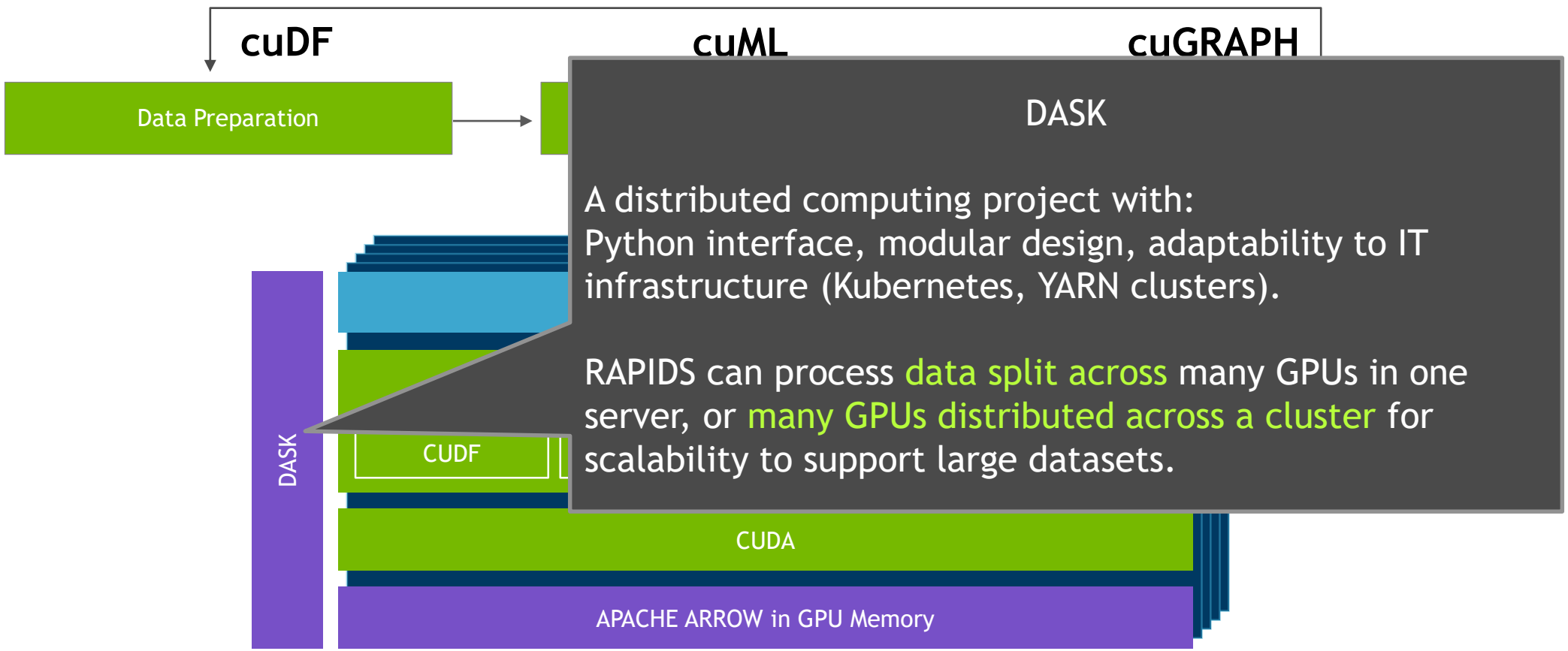
RAPIDS - SOFTWARE STACK

Execute end-to end data science and analytics pipelines on GPUs



RAPIDS - SOFTWARE STACK

Execute end-to end data science and analytics pipelines on GPUs



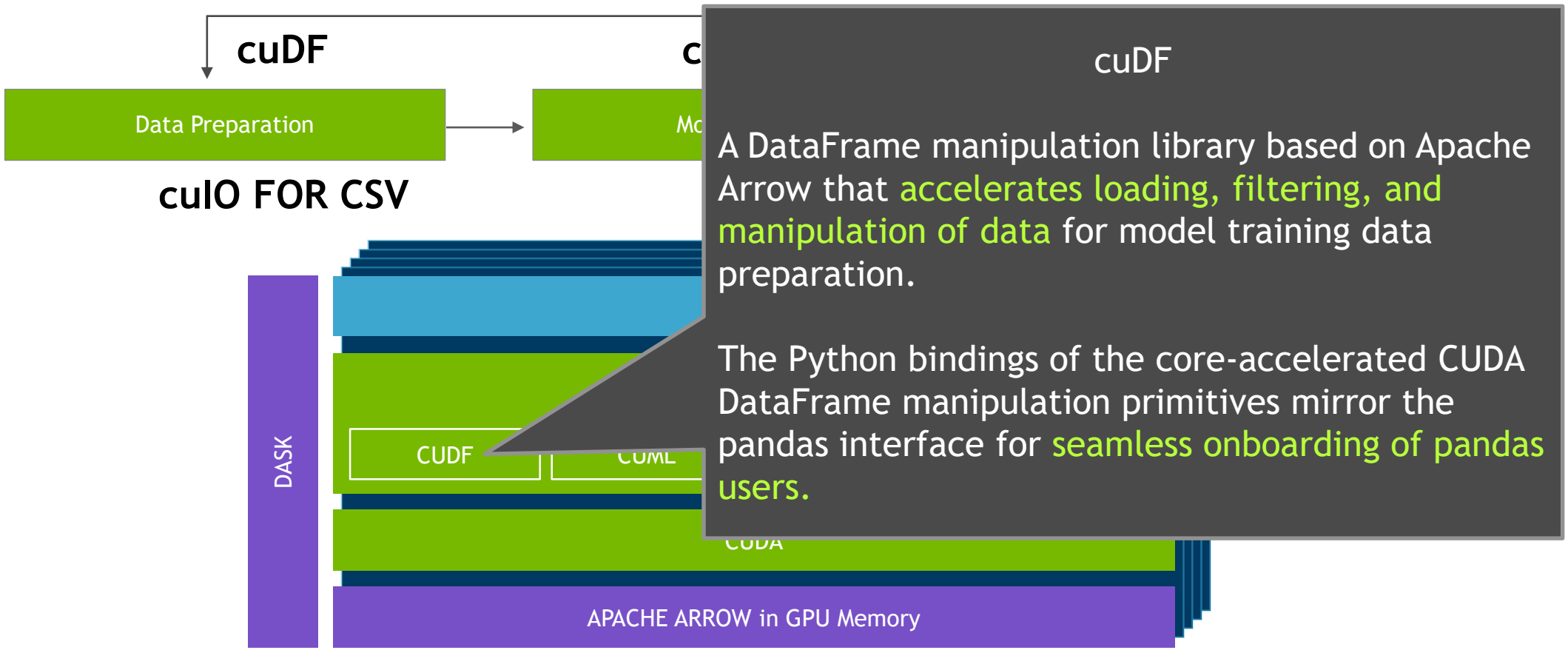
DASK

A distributed computing project with:
Python interface, modular design, adaptability to IT infrastructure (Kubernetes, YARN clusters).

RAPIDS can process data split across many GPUs in one server, or many GPUs distributed across a cluster for scalability to support large datasets.

RAPIDS - SOFTWARE STACK

Execute end-to end data science and analytics pipelines on GPUs



cuDF

A DataFrame manipulation library based on Apache Arrow that accelerates loading, filtering, and manipulation of data for model training data preparation.

The Python bindings of the core-accelerated CUDA DataFrame manipulation primitives mirror the pandas interface for seamless onboarding of pandas users.

RAPIDS - SOFTWARE STACK

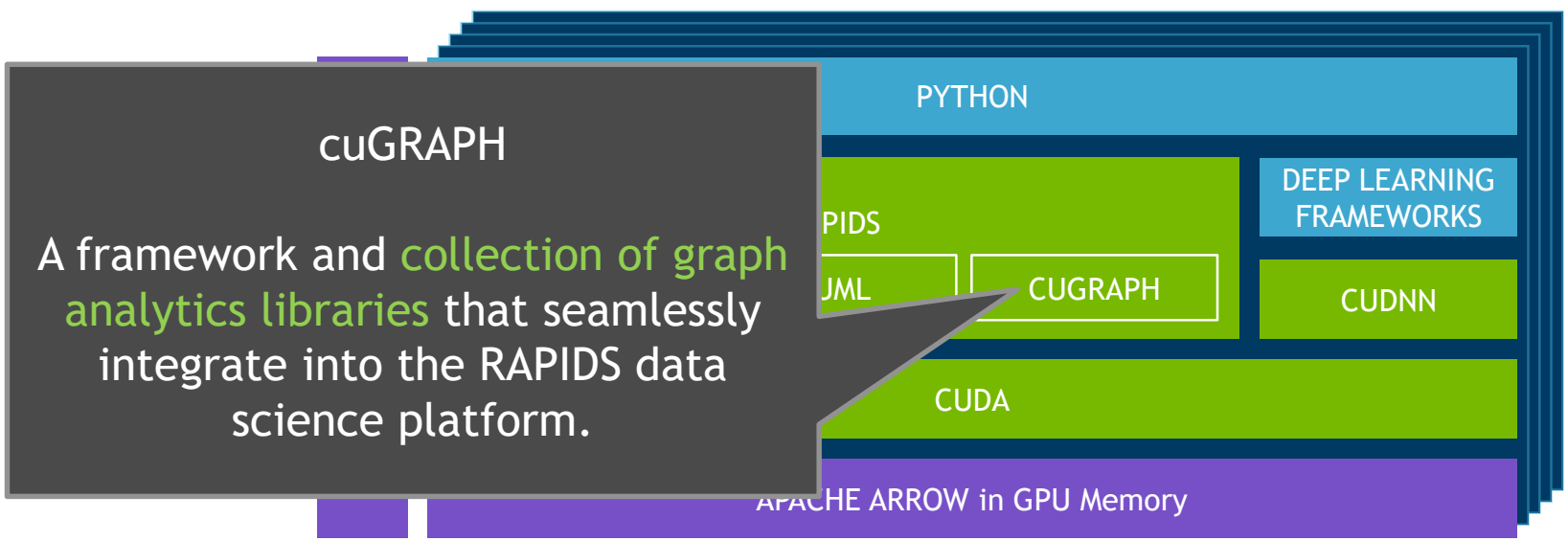
Execute end-to end data science and analytics pipelines on GPUs

cuML

A collection of GPU-accelerated machine learning libraries that will provide GPU versions of all machine learning algorithms available in scikit-learn..

RAPIDS - SOFTWARE STACK

Execute end-to end data science and analytics pipelines on GPUs



cuGRAPH

A framework and collection of graph analytics libraries that seamlessly integrate into the RAPIDS data science platform.

cuDF CODE EXAMPLES

PANDAS (CPU) VS CUDF (GPU) DATAFRAME

Code comparison

```python
import pandas as pd; print('Pandas Version:', pd.__version__)


# here we create a Pandas DataFrame with
# two columns named "key" and "value"
df = pd.DataFrame()
df['key'] = [0, 0, 2, 2, 3]
df['value'] = [float(i + 10) for i in range(5)]
print(df)
```

```
Pandas Version: 0.24.2
   key  value
0    0   10.0
1    0   11.0
2    2   12.0
3    2   13.0
4    3   14.0
```

```python
aggregation = df['value'].sum()
print(aggregation)
```

```
60.0
```

```python
import cudf; print('cuDF Version:', cudf.__version__)


# here we create a cuDF DataFrame with
# two columns named "key" and "value"
df = cudf.DataFrame() ## GPU
df['key'] = [0, 0, 2, 2, 3]
df['value'] = [float(i + 10) for i in range(5)]
print(df)
```

```
cuDF Version: 0.8.0a1+201.gcd2a144.dirty
   key  value
0    0   10.0
1    0   11.0
2    2   12.0
3    2   13.0
4    3   14.0
```

```python
aggregation = df['value'].sum()
print(aggregation)
```

```
60.0
```

DATA LOADING FROM FILE

cuDF code example

CSV files

```python
import nvstrings, nvcategory, cudf
import io, requests

# download CSV file from GitHub
url="https://github.com/plotly/datasets/raw/master/tips.csv"
content = requests.get(url).content.decode('utf-8')

# read CSV from memory
tips_df = cudf.read_csv(io.StringIO(content))
tips_df['tip_percentage'] = tips_df['tip']/tips_df['total_bill']*100

# display average tip by dining party size
print(tips_df.groupby('size').tip_percentage.mean())
```

```
size
1     21.72920154872781
2     16.57191917348289
3    15.215685473711835
4    14.594900639351334
5    14.149548965142023
6    15.622920072028379
Name: tip_percentage, dtype: float64
```

LOADING DATA INTO A GPU DATAFRAME

cuDF code examples

Create an empty DataFrame, and add a column.

```
import cudf

gdf = cudf.DataFrame()
gdf['my_column'] = [6, 7, 8]
print(gdf)
```

Create a DataFrame with two columns.

```
import cudf

gdf = cudf.DataFrame({'a': [3, 4, 5], 'b': [6, 7, 9]})

print(gdf)
```

Load a CSV file into a GPU DataFrame.

```
import cudf

path = './apartments.csv'

names = ['city', 'zipcode', 'price_per_m2', 'year_built',
         'population', 'median_income', 'date']

gdf = cudf.read_csv(path, names=names, delimiter=';', skipfooter=1)
```

Use Pandas to load a CSV file, and copy its content into a GPU DF.

```
import pandas as pd
import cudf

# Load a CSV file using pandas.
pdf = pd.read_csv(path, delimiter=';')

# Convert data types to ones supported by cudf.
pdf['city'] = pdf['city'].astype('category')
pdf['date'] = pdf['date'].astype("datetime64[ms]")

# Create a cudf dataframe from a pandas dataframe.
gdf = cudf.DataFrame.from_pandas(pdf)
```

WORKING WITH GPU DATAFRAMES

cuDF code examples

Return the first three rows as a new DataFrame.

```
print(gdf.head(3))
```

```
 city zipcode price_per_m2 year_built population median_income                    date
espoo    2100  5444.022222       1985       4332         26167 2018-09-06T00:00:00.000
espoo    2130       3768.0       1972       5983         29579 2018-08-20T00:00:00.000
espoo    2140       2770.0       1977       3689         29447 2018-12-19T00:00:00.000
```

Find the mean and standard deviation of a column.

```
print(gdf['population'].mean())
print(gdf['population'].std())
```

```
8014.397849462365
4373.122998945762
```

Transform column values with a custom function.

```
def double_income(median_income):
    return 2*median_income

gdf['median_income'] = gdf['median_income'].applymap(double_income)

print(gdf.head(2))
```

```
   city zipcode price_per_m2 year_built population median_income                    date
0 espoo    2100  5444.022222       1985       4332       52334.0 2018-09-06T00:00:00.000
1 espoo    2130       3768.0       1972       5983       59158.0 2018-08-20T00:00:00.000
```

Row slicing with column selection.

```
print(gdf.loc[2:3, ['zipcode', 'year_built']])
```

```
  zipcode year_built
2    2140       1977
3    2160       1990
Median income dtype is now: float64
```

Count number of occurrences per value, and unique values.

```
print(gdf['city'].value_counts())
print(gdf['city'].unique_count()) # nunique() in pandas.
```

```
helsinki   65
espoo   28
2
```

Change the data type of a column.

```
import numpy as np

print('Median income dtype used to be:', gdf['median_income'].dtype)
gdf['median_income'] = gdf['median_income'].astype(np.float64)
print('Median income dtype is now:', gdf['median_income'].dtype)
```

```
Median income dtype used to be: int64
Median income dtype is now: float64
```

QUERY, SORT, GROUP, JOIN, …

cuDF code examples

Query the columns of a DataFrame with a boolean expression.

```
query = gdf.query("year_built < 1930")
print(query.head(3))
```

```
      city zipcode       price_per_m2 year_built population median_income                     date
30 helsinki     130             7916.0       1911       1536       56220.0 2019-02-17T00:00:00.000
31 helsinki     140  7416.905659999999       1925       7817       55194.0 2018-10-09T00:00:00.000
32 helsinki     150 7727.5714290000005       1907       9299       49734.0 2018-09-29T00:00:00.000
```

Return the first 'n' rows ordered by 'columns' in ascending order.

```
three_smallest = gdf.nsmallest(n=2, columns=['population'])
print(three_smallest)
```

```
      city zipcode price_per_m2 year_built population median_income                     date
41 helsinki     310       3971.0       1972        896       46688.0 2018-10-05T00:00:00.000
30 helsinki     130       7916.0       1911       1536       56220.0 2019-02-17T00:00:00.000
```

Join columns with other DataFrame on index.

```
left = grouped
right = cudf.DataFrame({'zipcode': [28, 65], 'feature1': [1,2]})

# join() uses the index.
join_left = left.set_index('count_zipcode')
join_right = right.set_index('zipcode')

# Different join styles are supported.
joined = join_left.join(join_right, how='right')
```

Sort a column by its values.

```
gdf = gdf.sort_values(by='population', ascending=False)
print(gdf.head(3))
```

```
      city zipcode       price_per_m2 year_built population median_income                     date
89 helsinki     940        1982.028571       1967      25817       38172.0 2019-02-10T00:00:00.000
 8    espoo    2230           4035.075       1992      20397       46148.0 2018-12-09T00:00:00.000
58 helsinki     530        5090.853659       1944      18663       42582.0 2018-10-03T00:00:00.000
```

Group by column with aggregate function.

```
# Differences to pandas:
#    - aggr. column names are prefixed with the aggr. function name.
#    - 'city' becomes index in pandas but not in cudf.
grouped = gdf.groupby(['city']).agg({'zipcode': 'count'})
```

Merge two DataFrames.

```
# Only inner join is supported currently.
merged = left.merge(right, on=['zipcode'])
```

One-hot encoding

```
gdf['city_codes'] = gdf.city.cat.codes
codes = gdf.city_codes.unique()

# get_dummies() in pandas.
encoded = gdf.one_hot_encoding(column='city_codes', cats=codes,
                               prefix='city_codes_dummy', dtype='int8')
```

RAPIDS - SCIKIT_LEARN

Regression: CPU vs GPU coding

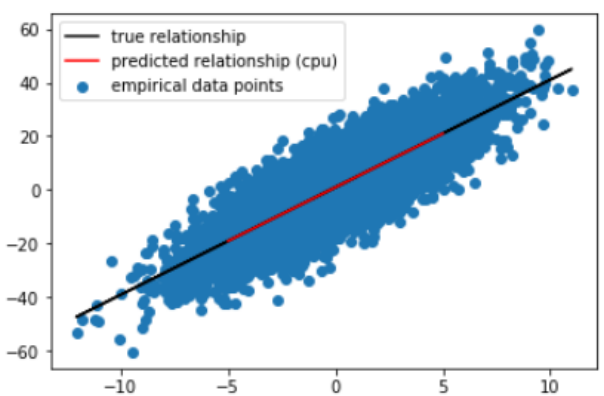
import cudf;
df = cudf.DataFrame({‘x’: x, ‘y’: y_noisy})

```
[ ]
import sklearn; print('Scikit-Learn Version:', sklearn.__version__)
from sklearn.linear_model import LinearRegression


# instantiate and fit model
linear_regression = LinearRegression()
linear_regression.fit(np.expand_dims(x, 1), y)

# create new data and perform inference
inputs = np.linspace(start=-5, stop=5, num=1000)
outputs = linear_regression.predict(np.expand_dims(inputs, 1))
```

```
Scikit-Learn Version: 0.21.1
```

```
[ ]
plt.scatter(x, y_noisy, label='empirical data points')
plt.plot(x, y, color='black', label='true relationship')
plt.plot(inputs, outputs, color='red', label='predicted relationship (cpu)')
plt.legend()
```

```
<matplotlib.legend.Legend at 0x7fe45fb4def0>
```



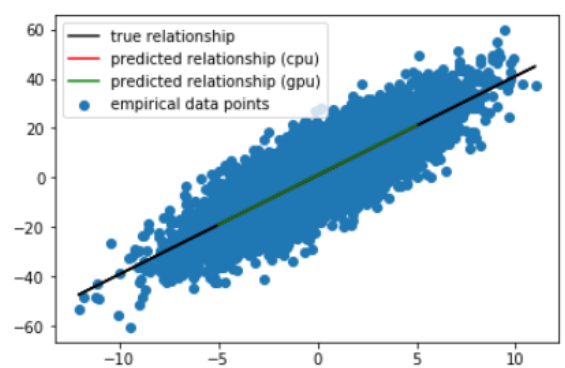

```
[ ]
import cuml; print('cuML Version:', cuml.__version__)
from cuml.linear_model import LinearRegression as LinearRegressionGPU

# instantiate and fit model
linear_regression_gpu = LinearRegressionGPU()
linear_regression_gpu.fit(df[['x']], df['y'])
```

```
cuML Version: 0.8.0a+975.g75d2f78.dirty
<cuml.linear_model.linear_regression.LinearRegression at 0x7fe45fb13940>
```

```
[ ]
# create new data and perform inference
new_data_df = cudf.DataFrame({'inputs': inputs})
outputs_gpu = linear_regression_gpu.predict(new_data_df[['inputs']])
```

```
plt.scatter(x, y_noisy, label='empirical data points')
plt.plot(x, y, color='black', label='true relationship')
plt.plot(inputs, outputs, color='red', label='predicted relationship (cpu)')
plt.plot(inputs, outputs_gpu.to_array(), color='green', label='predicted relationship (gpu)')
plt.legend()
```

```
<matplotlib.legend.Legend at 0x7fe45fac2cf8>
```



CLUSTERING

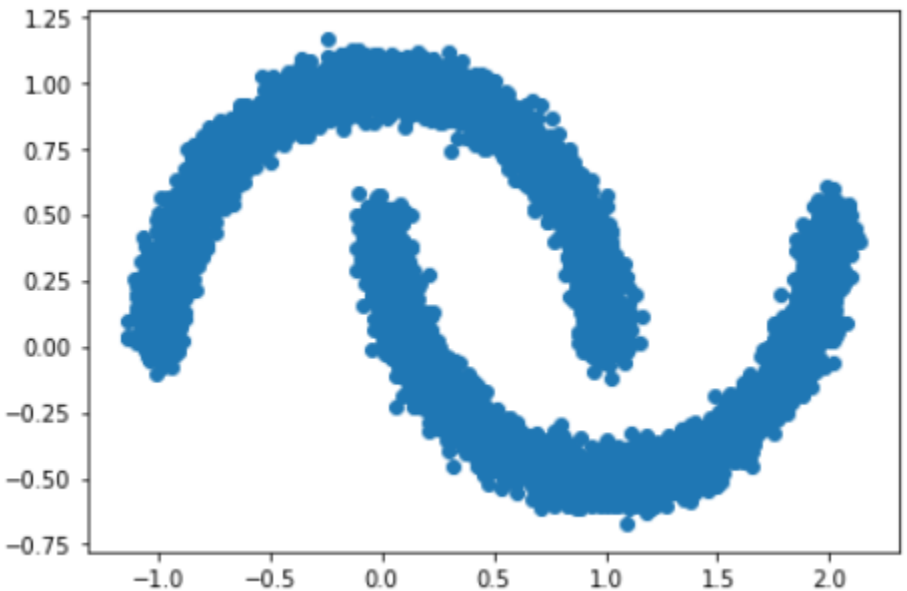
Example

```
[ ]
from matplotlib.colors import ListedColormap
import matplotlib.pyplot as plt


%matplotlib inline
```

```
[ ]
from sklearn.datasets import make_moons

X, y = make_moons(n_samples=int(1e4), noise=0.05, random_state=0)
print(X.shape)
```

```
(10000, 2)
```

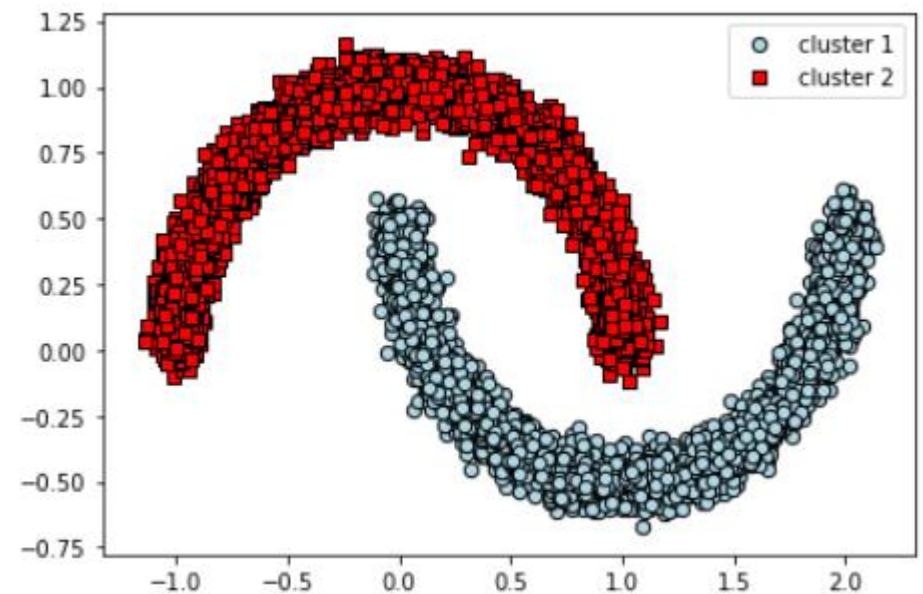
```
[ ]
plt.scatter(X[:, 0], X[:, 1])
plt.tight_layout()
plt.show()
```



DBSCAN

```
%%time
from sklearn.cluster import DBSCAN

db = DBSCAN(eps=0.2, min_samples=2)
y_db = db.fit_predict(X)
```

```
CPU times: user 196 ms, sys: 6.7 ms, total: 203 ms
Wall time: 211 ms
```



RAPIDS

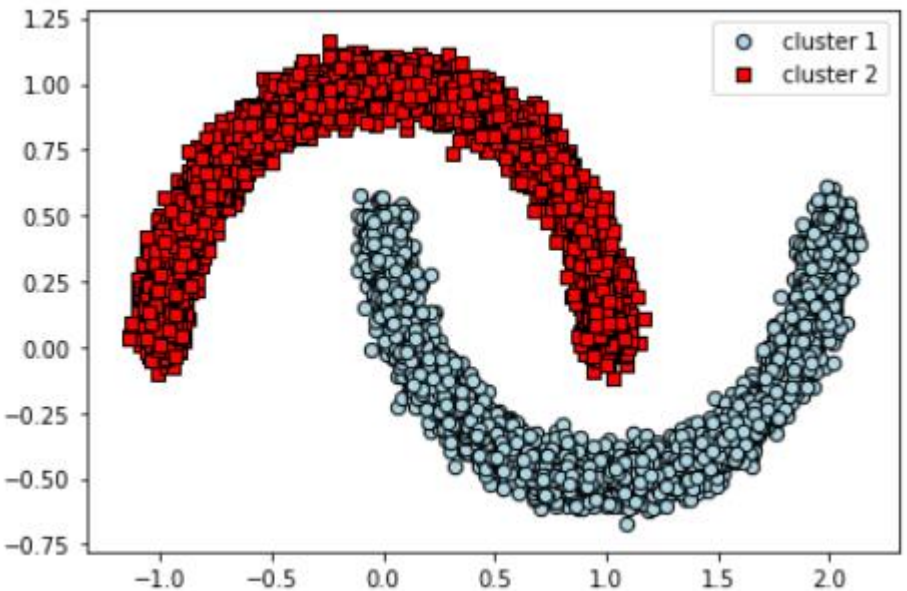
cuDF

```
[ ]
import pandas as pd
import cudf

X_df = pd.DataFrame({'fea%d'%i: X[:, i] for i in range(X.shape[1])})
X_gpu = cudf.DataFrame.from_pandas(X_df)
```

cuML

```
[ ]
%%time
from cuml import DBSCAN as cumlDBSCAN

db_gpu = cumlDBSCAN(eps=0.2, min_samples=2)
y_db_gpu = db_gpu.fit_predict(X_gpu)
```

```
CPU times: user 213 ms, sys: 33.3 ms, total: 247 ms
Wall time: 252 ms
```



COMPARE CPU VS GPU

```
[ ]
import numpy as np

n_rows, n_cols = 10000, 128
X = np.random.rand(n_rows, n_cols)
print(X.shape)
```

```
(10000, 128)
```

```
[ ]
db = DBSCAN(eps=3, min_samples=2)
```

```
[ ]
%%time
y_db = db.fit_predict(X)
```

```
CPU times: user 34 s, sys: 6.85 ms, total: 34 s
Wall time: 34 s
```

```
[ ]
X_df = pd.DataFrame({'fea%d'%i: X[:, i] for i in range(X.shape[1])})
X_gpu = cudf.DataFrame.from_pandas(X_df)
```

option with eps=3 and min_samples=2

```
[ ]
db_gpu = cumlDBSCAN(eps=3, min_samples=2)
```

```
[ ]
%%time
y_db_gpu = db_gpu.fit_predict(X_gpu)
```

```
CPU times: user 441 ms, sys: 199 ms, total: 640 ms
Wall time: 644 ms
```

CPU vs GPU

PCA

Training results:

CPU: 57.1 seconds
GPU: 4.28 seconds

System: AWS p3.8xlarge
CPUs: Intel(R) Xeon(R) CPU E5-2686 v4 @ 2.30GHz, 32 vCPU cores, 244 GB RAM
GPU: Tesla V100 SXM2 16GB
Dataset: https://github.com/rapidsai/cuml/tree/master/python/notebooks/data

Principal Component Analysis (PCA)

Before...

Specific: Import CPU algorithm

[1]:
```
from sklearn.decomposition import PCA
```

Common: Helper functions

[2]:
```
# Timer, load_data...
from helper import *
```

Common: Data loading and algo params

[3]:
```
# Data loading
nrows = 2**22
ncols = 400

X = load_data(nrows, ncols)
print('data', X.shape)

# Algorithm parameters
n_components = 8
whiten = False
random_state = 42
svd_solver = "full"
```

```
use mortgage data
data (4194304, 400)
```

Common: Training

[4]:
```
%%time
pca = PCA(n_components=n_components,svd_solver=svd_solver,
          whiten=whiten, random_state=random_state)
_ = pca.fit_transform(X)
```

```
CPU times: user 9min 19s, sys: 2min 12s, total: 11min 32s
Wall time: 57.1 s
```

...Now!

Specific: Import GPU algorithm

[1]:
```
from cuml import PCA
```

Common: Helper functions

[2]:
```
# Timer, load_data...
from helper import *
```

Common: Data loading and algo params

[3]:
```
# Data loading
nrows = 2**22
ncols = 400

X = load_data(nrows, ncols)
print('data', X.shape)

# Algorithm parameters
n_components = 10
whiten = False
random_state = 42
svd_solver = "full"
```

```
use mortgage data
data (4194304, 400)
```

Specific: DataFrame from Pandas to cuDF

[4]:
```
%%time
import cudf
X = cudf.DataFrame.from_pandas(X)
```

```
CPU times: user 4.46 s, sys: 4.68 s, total: 9.14 s
Wall time: 9.36 s
```

Common: Training

[5]:
```
%%time
pca = PCA(n_components=n_components,svd_solver=svd_solver,
          whiten=whiten, random_state=random_state)
_ = pca.fit_transform(X)
```

```
CPU times: user 1.94 s, sys: 512 ms, total: 2.45 s
Wall time: 4.28 s
```

CPU vs GPU

KNN

Training results:

CPU: ~9 minutes
GPU: 1.12 seconds

System: AWS p3.8xlarge
CPUs: Intel(R) Xeon(R) CPU E5-2686 v4 @ 2.30GHz, 32 vCPU cores, 244 GB RAM
GPU: Tesla V100 SXM2 16GB
Dataset: https://github.com/rapidsai/cuml/tree/master/python/notebooks/data

k-Nearest Neighbors (KNN)

Before...

Specific: Import CPU algorithm

[1]:
```
from sklearn.neighbors import KDTree as KNN
```

Common: Helper functions

[2]:
```
# Timer, load_data...
from helper import *
```

Common: Data loading and algo params

[3]:
```
# Data loading
nrows = 2**17
ncols = 40

X = load_data(nrows, ncols)
print('data', X.shape)

# Algorithm parameters
n_neighbors = 10
```

```
use mortgage data
data (131072, 40)
```

Specific: Training

[4]:
```
%%time
knn = KNN(X)
_ = knn.query(X, n_neighbors)
```

```
CPU times: user 9min 2s, sys: 272 ms, total: 9min 2s
Wall time: 8min 59s
```

...Now!

Specific: Import GPU algorithm

[1]:
```
from cuml import KNN
```

Common: Helper functions

[2]:
```
# Timer, load_data...
from helper import *
```

Common: Data loading and algo params

[3]:
```
# Data loading
nrows = 2**17
ncols = 40

X = load_data(nrows, ncols)
print('data', X.shape)

# Algorithm parameters
n_neighbors = 10
```

```
use mortgage data
data (131072, 40)
```

Specific: DataFrame from Pandas to cuDF

[4]:
```
%%time
import cudf
X = cudf.DataFrame.from_pandas(X)
```

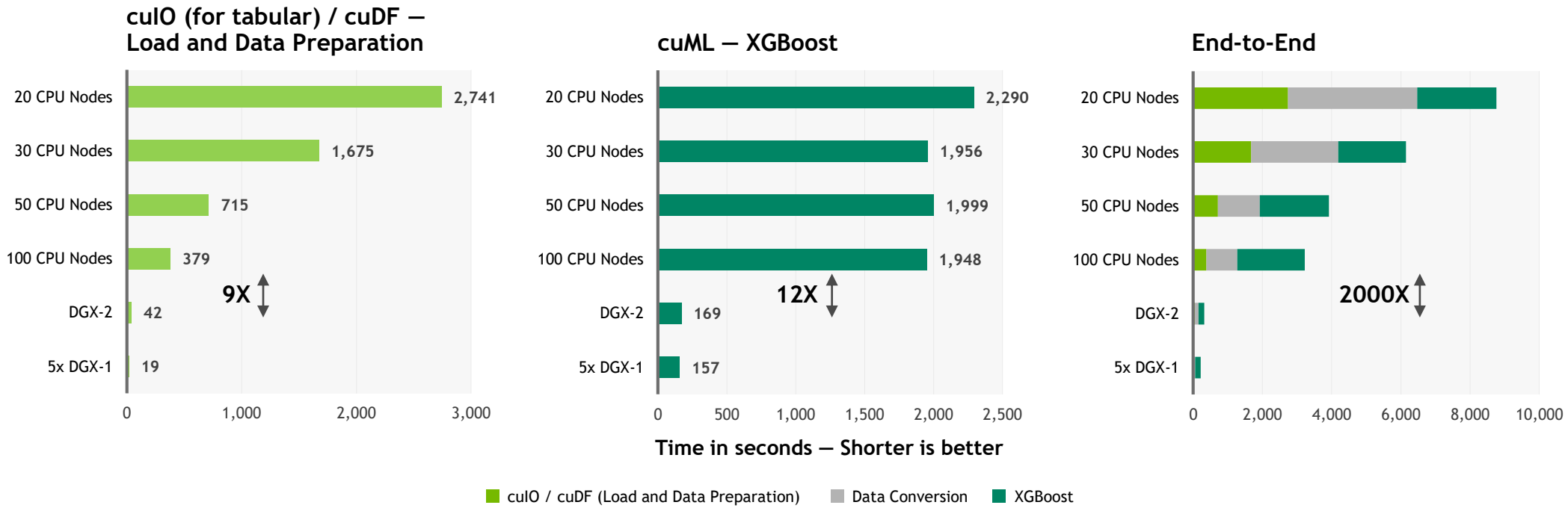
```
CPU times: user 3 s, sys: 552 ms, total: 3.56 s
Wall time: 839 ms
```

Specific: Training

[5]:
```
%%time
knn = KNN(n_gpus=1)
knn.fit(X)
_ = knn.query(X, n_neighbors)
```

```
CPU times: user 692 ms, sys: 428 ms, total: 1.12 s
Wall time: 1.12 s
```

THE RAPIDS (USECASE)



Benchmark

200GB CSV dataset; Data preparation includes joins, variable transformations.

CPU Cluster Configuration

CPU nodes (61 GiB of memory, 8 vCPUs, 64-bit platform), Apache Spark

DGX Cluster Configuration

5x DGX-1 on InfiniBand network

Roadmap

cuDF — ROADMAP

Analytics

Libraries

daskgdf: Distributed Computing pygdf using Dask; Support for multi-GPU, multi-node

pygdf: Python bindings for libgdf (Pandas like API for DataFrame manipulation)

libgdf: CUDA C++ Apache Arrow GPU DataFrame and operators (Join, GroupBy, Sort, etc.)

daskgdf
Distributed Computing

pygdf
Python Bindings

libgdf
CUDA C/C++ helper function

Memory Allocation Requirement

Budget 2-3X dataset size for cuDF working memory

Multi-GPU Multi-node Roadmap

Availability	Multi-GPU	Multi-Node	Peer-to-peer Data Sharing*
Now	Yes	Yes	Yes

Note: No peer-to-peer data sharing means computation performed via map/reduce style programming in Dask

TODAY - RAPIDS 0.6

GTC San Jose 2019

cuML	SG	MG	MGMN
Gradient Boosted Decision Trees (GBDT)	■	■	■
GLM	■	■	
Logistic Regression	■		
Random Forest (regression)	■	■	■
K-Means	■	■	
K-NN	■	■	
DBSCAN	■		
UMAP	■		
ARIMA			
Kalman Filter	■		
Holts-Winters			
Principal Components	■		
Singular Value Decomposition	■	■	

cuGraph	SG	MG	MGMN
Jaccard	■		
Weighted Jaccard	■		
PageRank	■		
Louvain	■		
SSSP	■		
BFS	■		
SSWP			
Triangle Counting	■		
Subgraph Extraction			

SG Single GPU
MG Multi-GPU
MGMN Multi-GPU Multi-Node

ROAD TO 1.0

Q4 2019 – RAPIDS 0.12

cuML	SG	MG	MGMN
Gradient Boosted Decision Trees (GBDT)	■	■	■
GLM	■	■	■
Logistic Regression	■	■	■
Random Forest (regression)	■	■	■
K-Means	■	■	■
K-NN	■	■	■
DBSCAN	■	■	■
UMAP	■	■	■
ARIMA	■	■	
Kalman Filter	■	■	
Holts-Winters	■	■	
Principal Components	■	■	■
Singular Value Decomposition	■	■	■

cuGraph	SG	MG	MGMN
Jaccard	■		
Weighted Jaccard	■		
PageRank	■	■	■
Louvain	■		
SSSP	■		
BFS	■	■	
SSWP	■	■	
Triangle Counting	■		
Subgraph Extraction	■		

SG Single GPU
MG Multi-GPU
MGMN Multi-GPU Multi-Node

XGBOOST SUPPORT SPARK NOW

Scaling using Apache Spark

Apache Spark

Easy to use through Scala/Java API using an existing Spark cluster – Python API coming soon

YARN for managing resources

Dask

Simple to use Python API that is infrastructure agnostic

Kubernetes for managing resources

Both Apache Spark and Dask support loading CSV, Parquet, ORC, and other file formats from local disk, HDFS, S3, GS, Azure Storage

Near identical performance for both Apache Spark and Dask

RAPIDS RESOURCE

http://www.rapids.ai/

cuDF Documentation: https://rapidsai.github.io/projects/cudf/en/latest/

cuML Documentation: https://rapidsai.github.io/projects/cuml/en/latest/

cuGraph Documentation: https://rapidsai.github.io/projects/cugraph/en/0.6.0/index.html

Dask Documentation: https://docs.dask.org/en/latest/

Arrow Documentation: https://arrow.apache.org/docs/

GitHub: https://github.com/RAPIDSai

THANK YOU

Sangmoon Lee, sml@nvidia.com