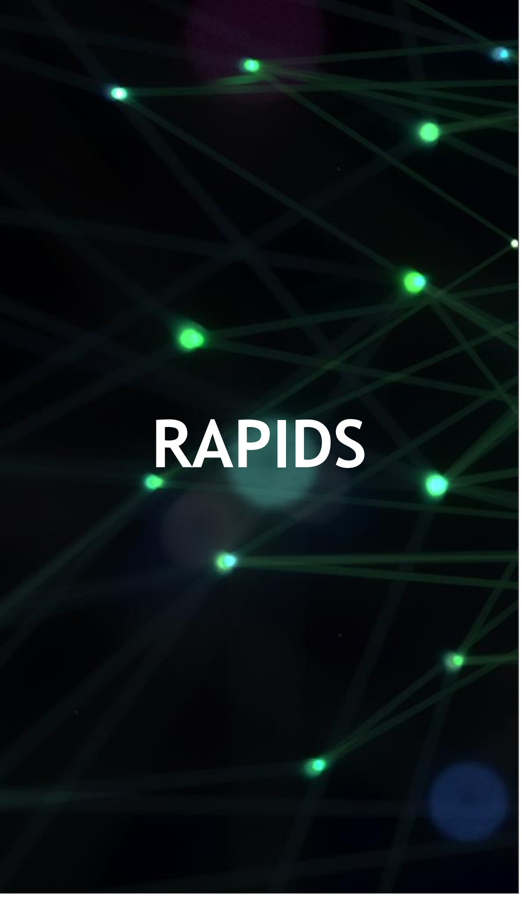
RAPIDS

GPU POWERED MACHINE LEARNING

Miguel Martínez

WHAT IS RAPIDS

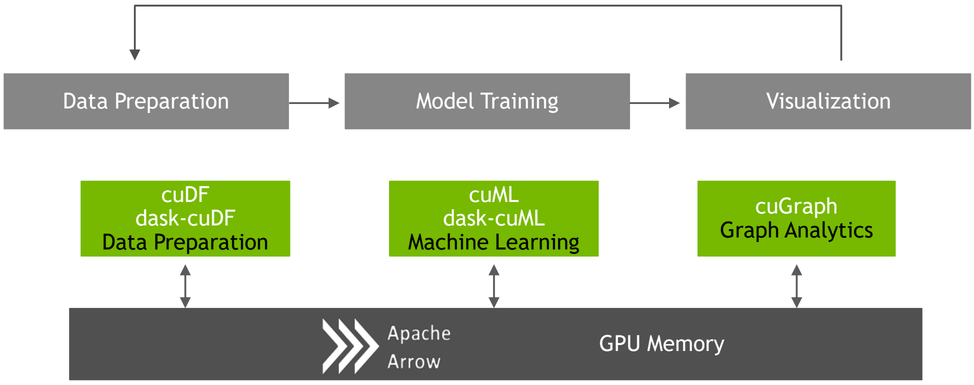


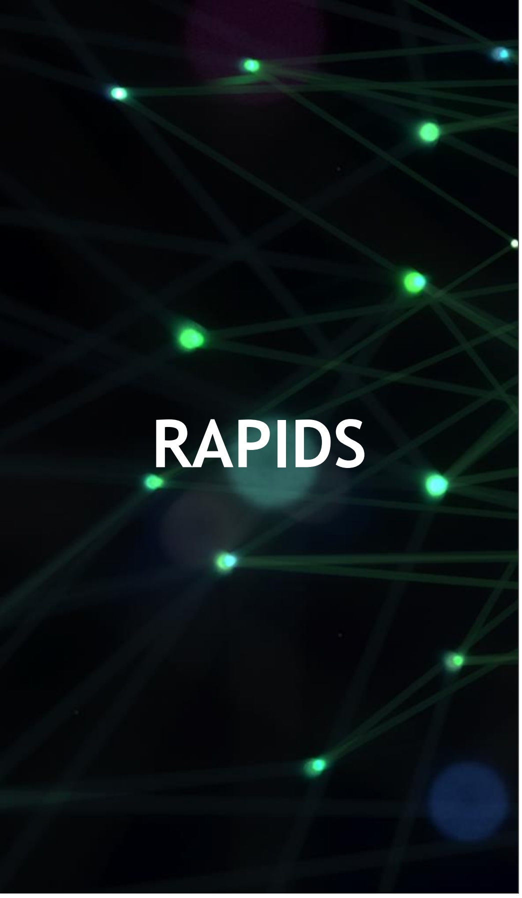
RAPIDS

GPU Accelerated End-to-End Data Science

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

OSS website: rapids.ai





RAPIDS LIBRARIES

cuDF

- GPU-accelerated lightweight in-GPU memory database used for data preparation
- Accelerates loading, filtering, and manipulation of data for model training data preparation
- Python drop-in Pandas replacement built on CUDA C++

cuML

- GPU accelerated traditional machine learning libraries
- XGBoost, PCA, Kalman, K-means, k-NN, DBScan, tSVD …

cuGRAPH

- Collection of graph analytics libraries.

HOW TO SETUP AND START USING RAPIDS

HOW? DOWNLOAD AND DEPLOY

Source available on GitHub | Container available on NGC and Docker Hub | Conda and PIP

https://github.com/rapidsai

https://ngc.nvidia.com
https://hub.docker.com/u/rapidsai

https://anaconda.org/rapidsai
https://pypi.org/project/cudf/
https://pypi.org/project/cuml/

On-premises

Cloud

Pascal GPU architecture or better
CUDA 9.2 or 10.0
Ubuntu 16.04 or 18.04

RUNNING RAPIDS CONTAINER IN THE CLOUD

A step-by-step installation guide (MS Azure)

1. Create a *NC6s_v2* virtual machine instance on *Microsoft Azure Portal* using *NVIDIA GPU Cloud Image for Deep Learning and HPC* as image.

2. Start the virtual machine.

3. Connect to the virtual machine using the following command:

   ```
   $ ssh -L 8080:localhost:8888 \
         -L 8787:localhost:8787 \
         username@public_ip_address
   ```

4. Pull the *RAPIDS container* from *NGC*. Run it.

   ```
   $ docker pull nvcr.io/nvidia/rapidsai/rapidsai:cuda10.0-runtime-ubuntu18.04
   $ docker run --runtime=nvidia \
                --rm -it \
                -p 8888:8888 \
                -p 8787:8787 \
                -p 8786:8786 \
                nvcr.io/nvidia/rapidsai/rapidsai:cuda10.0-runtime-ubuntu18.04
   ```

5. Run JupyterLab:

   ```
   (rapids)$ bash /rapids/notebooks/utils/start-jupyter.sh
   ```

6. Open your browser, and navigate to *http://localhost:8080*.

7. Navigate to:
   - *cuml* folder for cuML IPython examples.
   - *mortgage* folder for XGBoost IPython examples.

8. Enjoy!

RUNNING RAPIDS CONTAINER IN THE CLOUD

A step-by-step installation guide (AWS)

1. Create a *p3.8xlarge* machine instance on Amazon Web Services using *NVIDIA Volta Deep Learning AMI* as image.

2. Start the virtual machine.

3. Connect to the virtual machine using the following command:

```
$ ssh -L 8080:localhost:8888 \
      -L 8787:localhost:8787 \
      ubuntu@public_ip_address
```

4. Pull the *RAPIDS container* from *NGC*. Run it.

```
$ docker pull nvcr.io/nvidia/rapidsai/rapidsai:cuda10.0-runtime-ubuntu18.04
$ docker run --runtime=nvidia \
             --rm -it \
             -p 8888:8888 \
             -p 8787:8787 \
             -p 8786:8786 \
             nvcr.io/nvidia/rapidsai/rapidsai:cuda10.0-runtime-ubuntu18.04
```

5. Run JupyterLab:

```
(rapids)$ bash /rapids/notebooks/utils/start-jupyter.sh
```

6. Open your browser, and navigate to *http://localhost:8080*.

7. Navigate to:
   - *cuml* folder for cuML IPython examples.
   - *mortgage* folder for XGBoost IPython examples.

8. Enjoy!

HOW TO PORT EXISTING CODE

CPU vs GPU

PORTING EXISTING CODE
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

PORTING EXISTING CODE 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
```

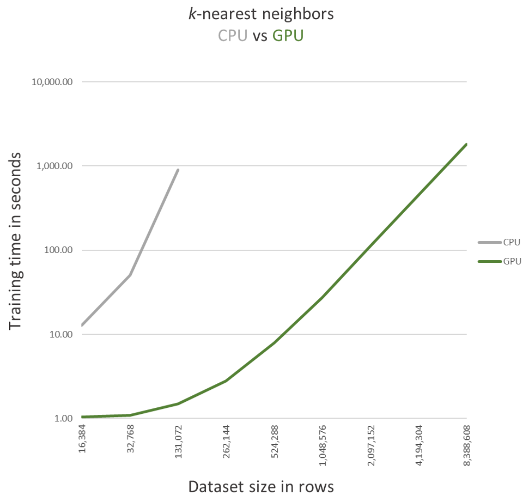
Specific: Training

```
[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
```

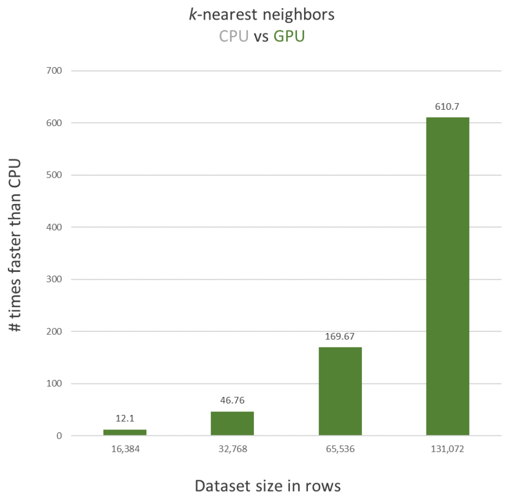
TRAINING TIME COMPARISON

CPU vs GPU



Dataset size trained in 15 minutes.
CPU: ~130.000 rows.
GPU: ~5.900.000 rows.

Specs	NC6s_vs
Cores (Broadwell 2.6Ghz)	6
GPU	1 x P100
Memory	112 GB
Local Disk	~700 GB SSD
Network	Azure Network



The bigger the dataset is, the higher the training performance difference is between CPU and GPU.

WHAT IS XGBOOST

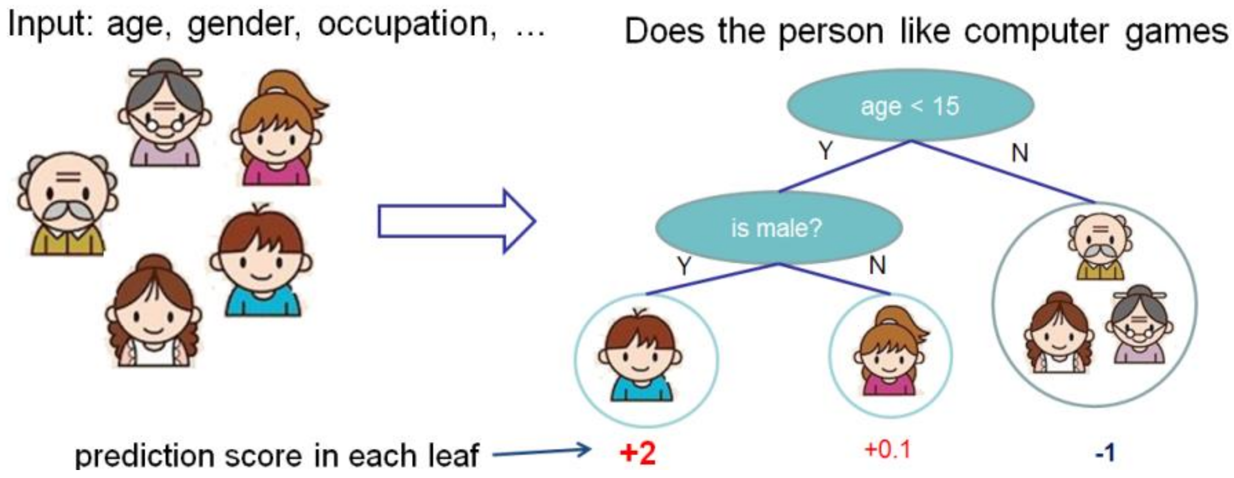
Definition

XGBoost is an implementation of gradient boosted decision trees designed for speed and performance.

It is a powerful tool for solving classification and regression problems in a supervised learning setting.

PREDICT: WHO ENJOYS COMPUTER GAMES

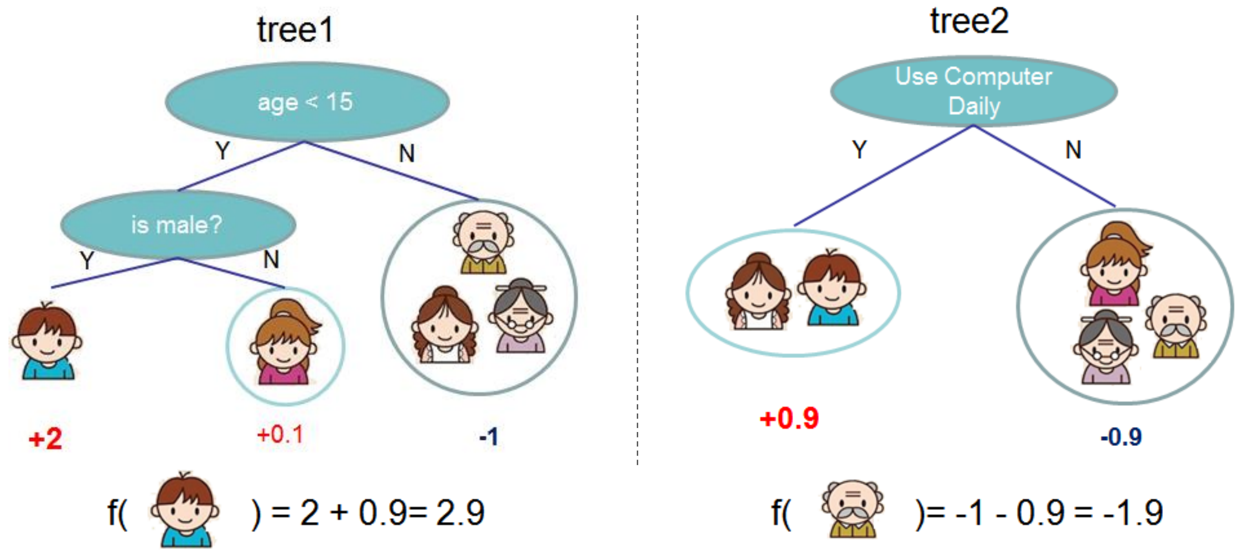
Example of Decision Tree



Source: https://goo.gl/C6WKiF

COMBINE TREES FOR STRONGER PREDICTIONS

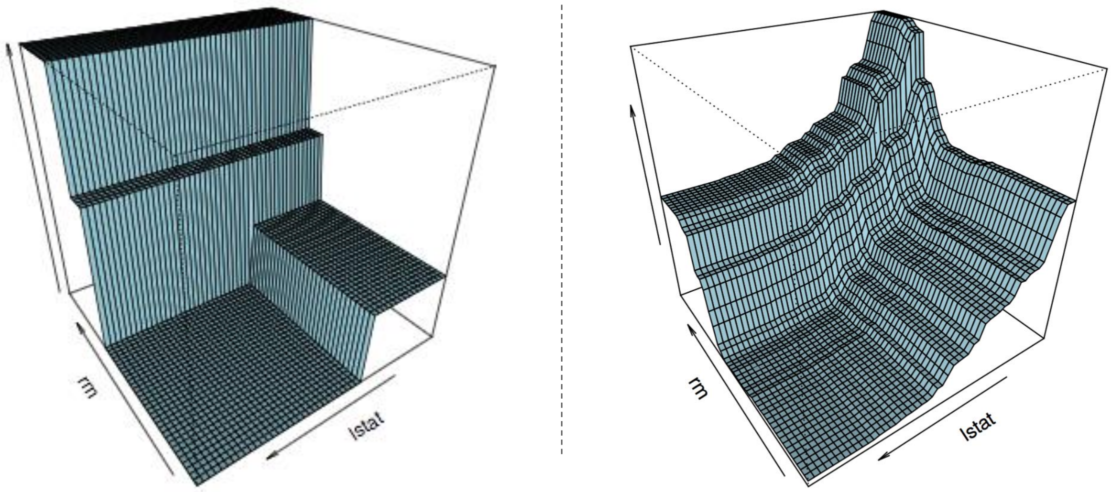
Example of Using Ensembled Decision Trees



Source: https://goo.gl/C6WKiF

TRAINED MODELS VISUALIZATION

Single Decision Tree vs Ensembled Decision Trees



Models fit to the *Boston Housing* Dataset.

Source: https://goo.gl/GWNdEm

WHY XGBoost

A STRONG HISTORY OF SUCCESS

On a Wide Range of Problems

Winner of Caterpiller Kaggle Contest 2015

- Machinery component pricing

Winner of CERN Large Hadron Collider Kaggle Contest 2015

- Classification of rare particle decay phenomena

Winner of KDD Cup 2016

- Research institutions' impact on the acceptance of submitted academic papers

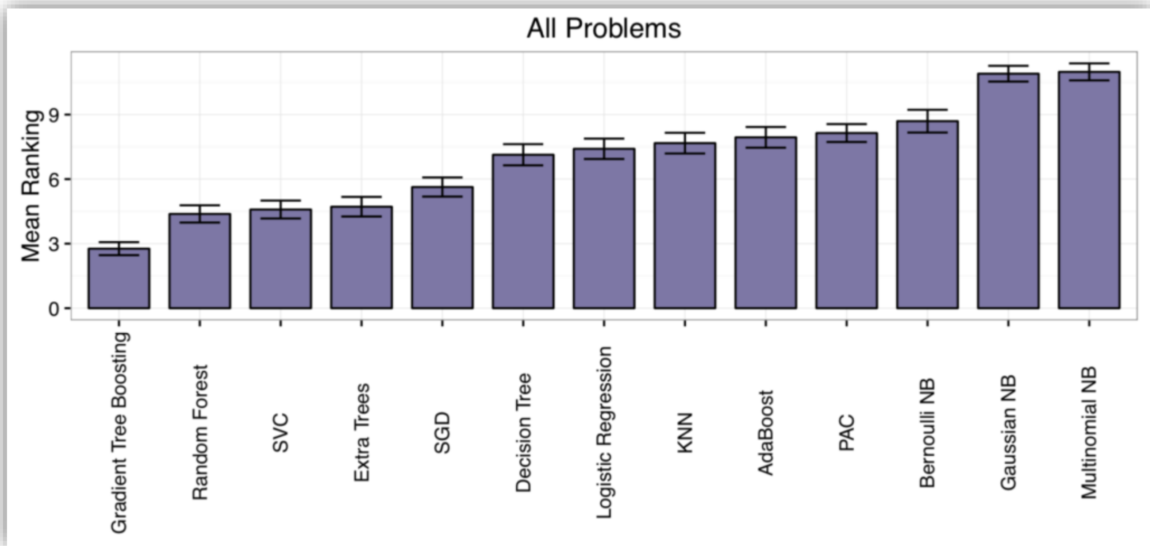
Winner of ACM RecSys Challenge 2017

- Job posting recommendation

WHICH ML ALGORITHM PERFORMS BEST

Average rank across 165 ML datasets

Lower Is Better



Source: https://goo.gl/aztMh2

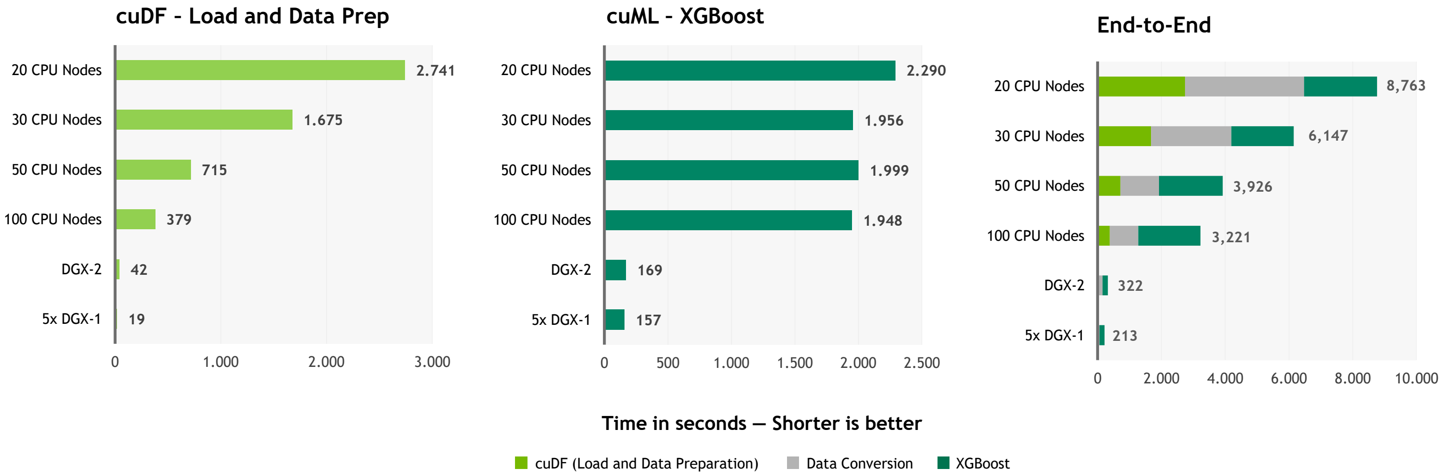
WHY XGBOOST + RAPIDS

WHY RAPIDS WITH XGBOOST

Multi-GPU, Multi-Node, Scalability

- XGBoost:
  - Algorithm tuned for eXtreme performance and high efficiency
  - Multi-GPU and Multi-Node Support
- RAPIDS:
  - End-to-end data science & analytics pipeline entirely on GPU
  - User-friendly Python interfaces
  - Faster results helps hyperparameter tuning
  - Relies on CUDA primitives, exposes parallelism and high-memory bandwidth

BENCHMARKS



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

cuML ROADMAP

cuML Algorithms	Available Now		Q2-2019
XGBoost GBDT	MGMN		
XGBoost Random Forest			MGMN
K-Means Clustering	MG		
K-Nearest Neighbors (KNN)	MG		
Principal Component Analysis (PCA)	SG		
Density-based Spatial Clustering of Applications with Noise (DBSCAN)	SG		
Truncated Singular Value Decomposition (tSVD)	SG		
Uniform Manifold Aproximation and Projection (UMAP)	SG		MG
Kalman Filters (KF)	SG		
Ordinary Least Squares Linear Regression (OLS)	SG		
Stochastic Gradient Descent (SGD)	SG		
Generalized Linear Model, including Logistic (GLM)			MG
Time Series (Holts-Winters)			SG
Autoregressive Integrated Moving Average (ARIMA)			SG

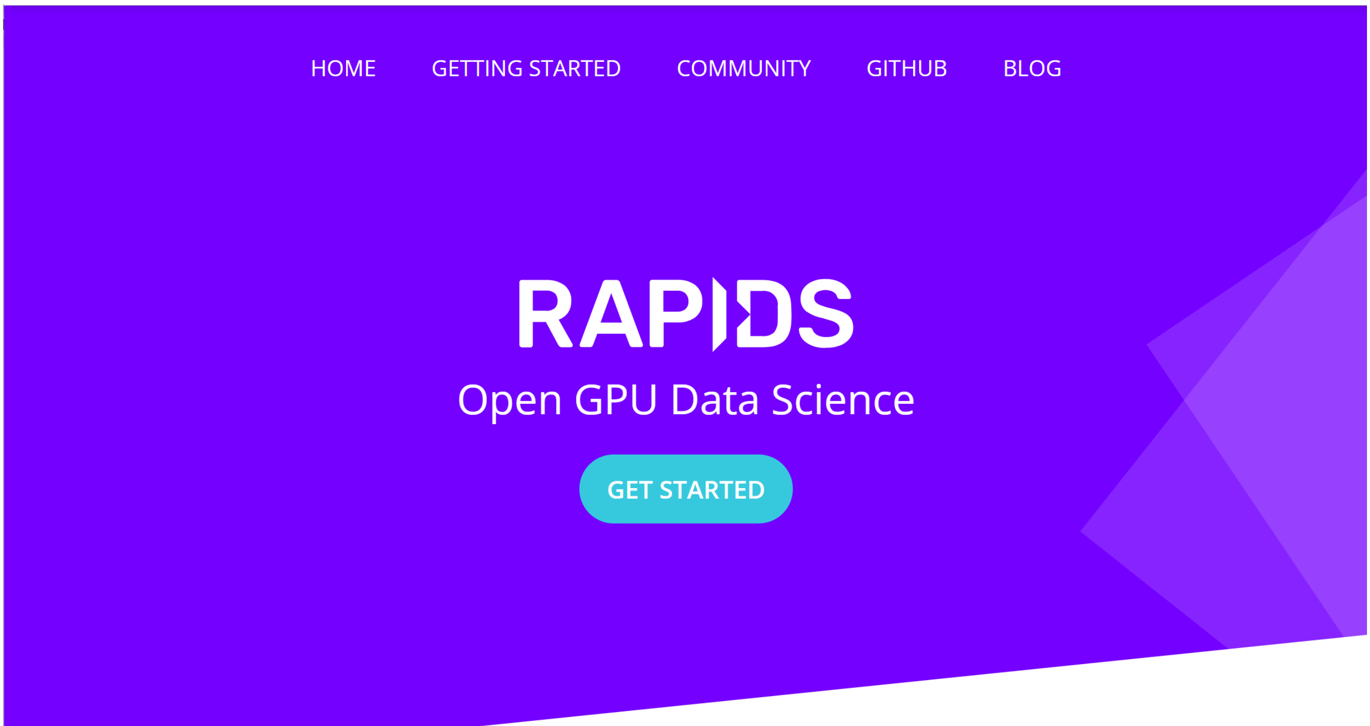
SG
Single GPU

MG
Multi-GPU

MGMN
Multi-GPU Multi-Node

Last updated 29.03.19

LEARN MORE ABOUT RAPIDS



https://rapids.ai

CUDF
CODE SAMPLES

LOADING DATA INTO A GPU DATAFRAME

Create an empty DataFrame, and add a column.

[1]:
```
import cudf

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

```
  my_column
0         6
1         7
2         8
```

Create a DataFrame with two columns.

[2]:
```
import cudf

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

print(gdf)
```

```
     b    a
0    6    3
1    7    4
2    9    5
```

Load a CSV file into a GPU DataFrame.

[3]:
```
import cudf

path = './apartments.csv'

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

# Note: dtype detection is not yet supported.
dtypes = ['category', 'int64', 'float64', 'float64',
          'int64', 'int64', 'date']

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

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

[4]:
```
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

Return the first three rows as a new DataFrame.

[5]:
```
print(gdf.head(3))
```

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

Find the mean and standard deviation of a column.

[7]:
```
print(gdf['population'].mean())
print(gdf['population'].std())
```

```
8014.397849462365
4373.122998945762
```

Change the data type of a column.

[9]:
```
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
```

Row slicing with column selection.

[6]:
```
print(gdf.loc[2:5, ['zipcode', 'year_built']])
```

```
  zipcode year_built
2    2140       1977
3    2160       1990
4    2170       1972
5    2180       1986
```

Count number of occurrences per value, and number of unique values.

[8]:
```
print(gdf['city'].value_counts())
print(gdf['city'].unique_count()) # nunique() in pandas.
```

```
helsinki   65
espoo   28
2
```

Transform column values with a custom function.

[10]:
```
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
```

QUERY, SORT, GROUP, JOIN, MERGE, ONE-HOT ENCODING

Query the columns of a DataFrame with a boolean expression.

[11]:
```
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.

[13]:
```
three_smallest = gdf.nsmallest(n=3, 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
44 helsinki     340  4497.333333       1973       1654       64768.0 2018-11-20T00:00:00.000
```

Join columns with other DataFrame on index.

[15]:
```
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.

[12]:
```
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.

[14]:
```
# Differences to pandas:
#     - aggregated column names are prefixed with the
#       aggregated function name.
#     - 'city' becomes index in pandas but not in cudf.
grouped = gdf.groupby(['city']).agg({'zipcode': 'count'})
```

Merge two DataFrames.

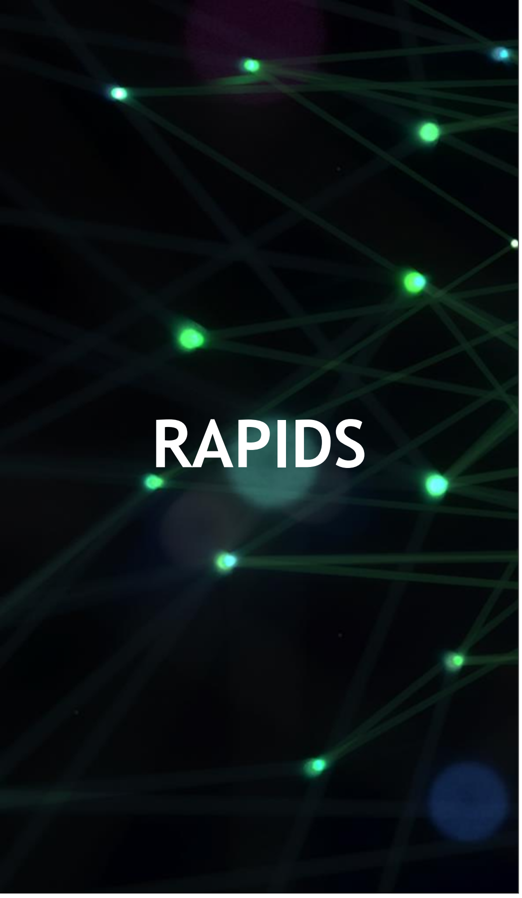
[16]:
```
# Only inner join is supported currently.
merged = left.merge(right, on=['zipcode'])
```

One-hot encoding.

[17]:
```
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')
```

SUMMARY



GPU Accelerated Data Science

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

Visit www.rapids.ai

ONE MORE THING

MESSAGE TO DATA SCIENTISTS

FIND A NEW ARGUMENT


NVIDIA