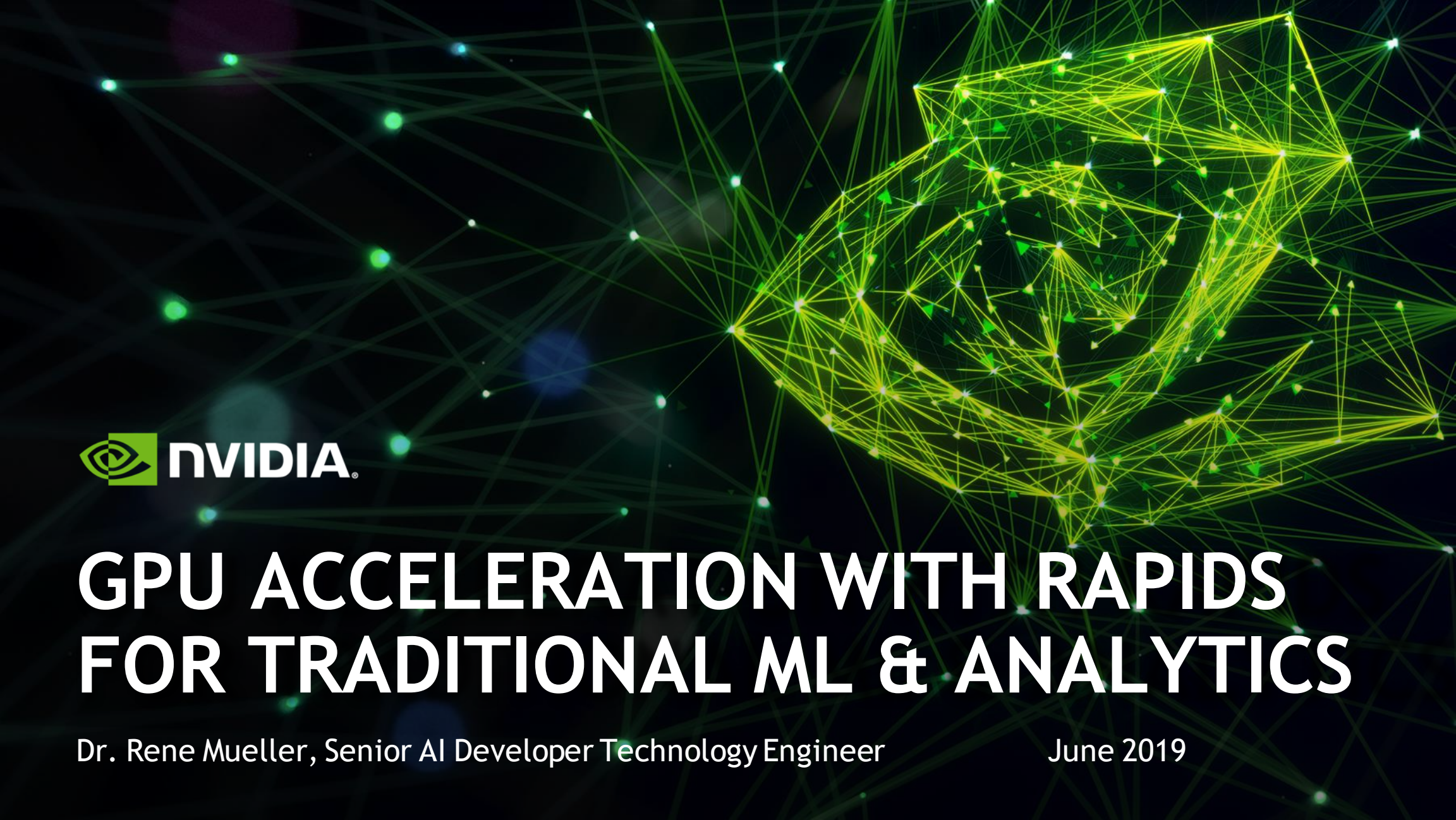


GPU ACCELERATION WITH RAPIDS FOR TRADITIONAL ML & ANALYTICS

Dr. Rene Mueller, Senior AI Developer Technology Engineer

June 2019

CHALLENGES AFFECTING DATA SCIENCE TODAY

What Needs to be Solved to Empower Data Scientists

INCREASING DATA ONSLAUGHT

Data sets are continuing to dramatically increase in size

Multitude of sources

Different formats, varying quality

SLOW CPU PROCESSING

End of Moore's law, CPUs aren't getting faster

Many popular data science tools have been CPU-only

Can only throw so many CPUs at a job

COMPLEX INSTALLATION & MANAGEMENT

Time consuming to install software

Nearly impossible to manage all version conflicts

Updates often break other software

NVIDIA DATA SCIENCE DISTRIBUTION

GPU-Accelerated Data Science Software built on CUDA-X AI and XGBoost

DELIVERY

PYTHON PIP
NGC CONTAINERS
ANACONDA CONDA

DEVELOPMENT

PYTHON
NOTEBOOKS
VISUALIZATION

WORKFLOWS

KUBEFLOW PIPELINES
KUBERNETES

CORE FRAMEWORKS AND LIBRARIES

Chainer · CuPy · DASK · OpenCV · PyTorch · dmlc XGBoost · RAPIDS · TensorFlow

CUDA-X AI

DATA PROCESSING	MACHINE LEARNING	DEEP LEARNING
cuDF, DALI	cuML, cuGRAPH	cuDNN, cuBLAS, NCCL, TensorRT

DEPLOY ON WORKSTATION, IN DATA CENTER, OR ON CLOUD

NVIDIA-powered data science workstations

Local development

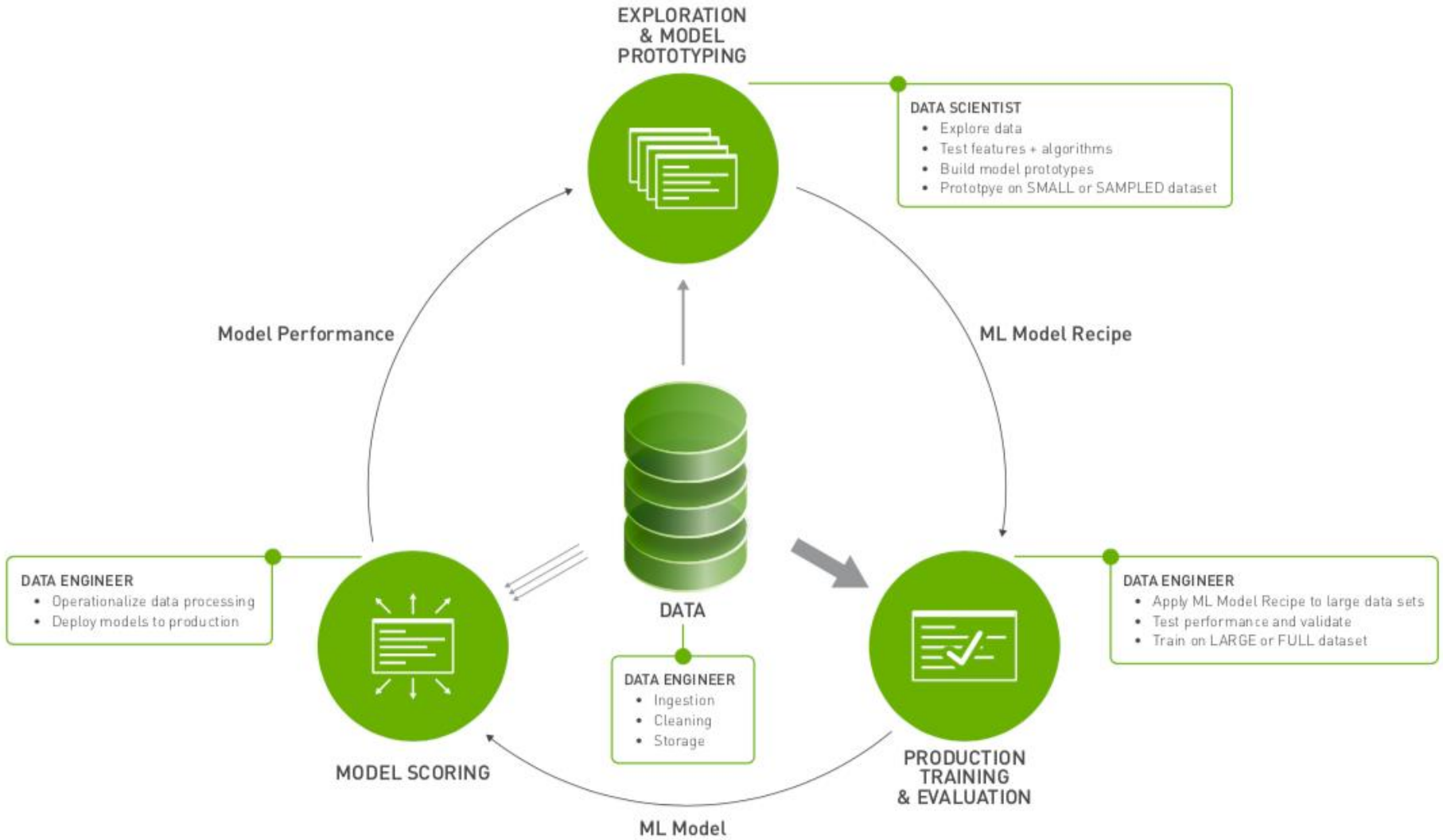
NVIDIA-powered servers

Jupyter notebooks and scale out training

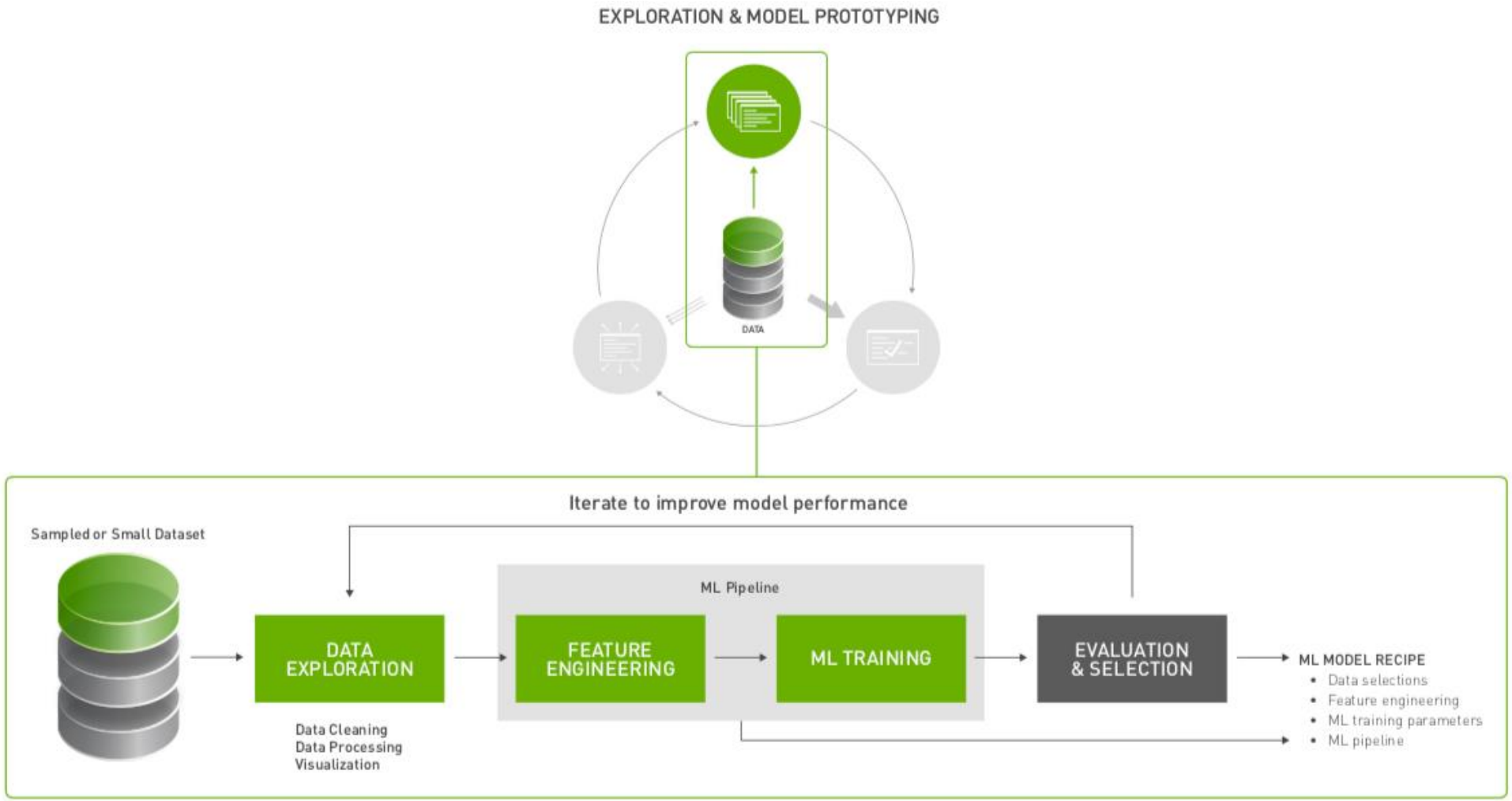
aws · Alibaba Group · Google Cloud · Azure

Jupyter notebooks, scale out training, and inference

DATA SCIENCE DEVELOPMENT LIFECYCLE

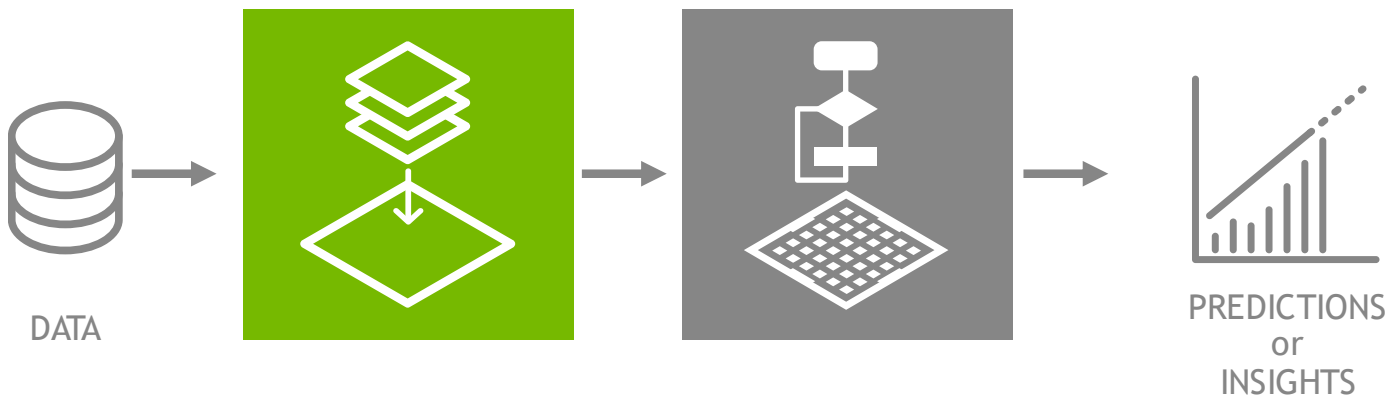


EXPLORATION AND MODEL PROTOTYPING



GPU-ACCELERATED DATA SCIENCE WORKFLOW WITH RAPIDS

Built on CUDA-X AI



Today, RAPIDS cuDF: **DATA PREPARATION**

GPU-accelerated compute for in-memory data preparation

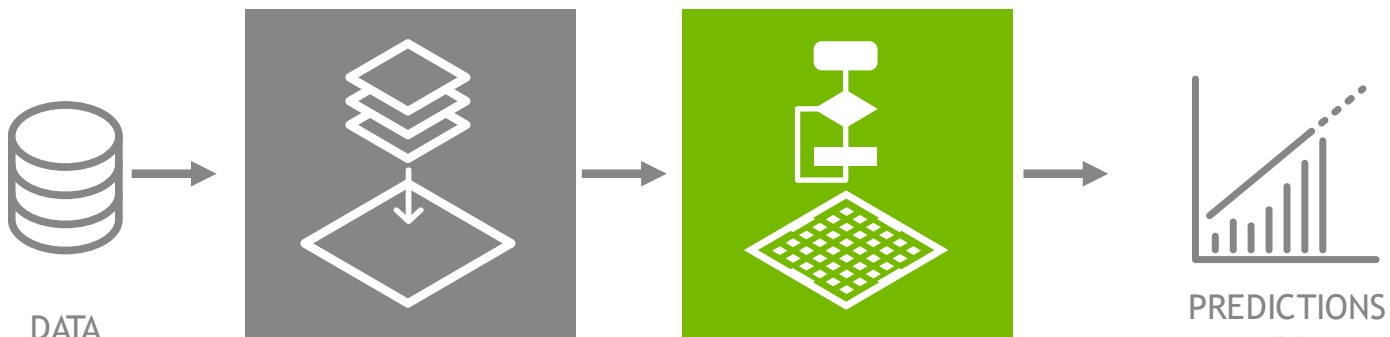
Simplified implementation using familiar data science tools

GPU-accelerated **pandas**-like **cuDF** built on CUDA C++

GPU-accelerated Spark (in development)

GPU-ACCELERATED DATA SCIENCE WORKFLOW WITH RAPIDS

Built on CUDA-X AI



Today, RAPIDS cuML: MACHINE LEARNING

GPU-acceleration of today's most popular ML algorithms

Easy-to-adopt, **scikit-learn** like interface

Besides cuML, RAPIDS also allows easy integration dmlc **XGBoost**

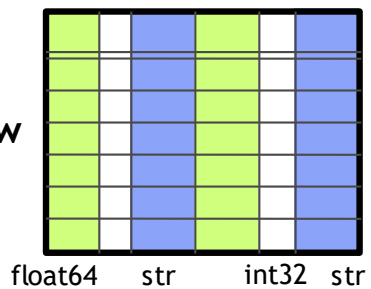
DATAFRAMES

Pandas and cuDF GPU Dataframes

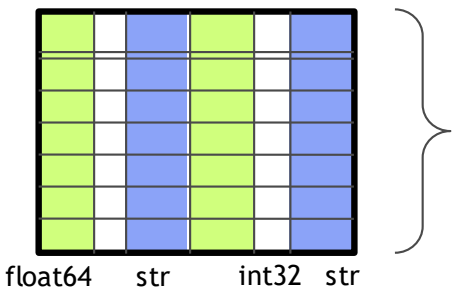
Dataframes store tabular data like spreadsheets or SQL tables:

Pandas Dataframe

User's View

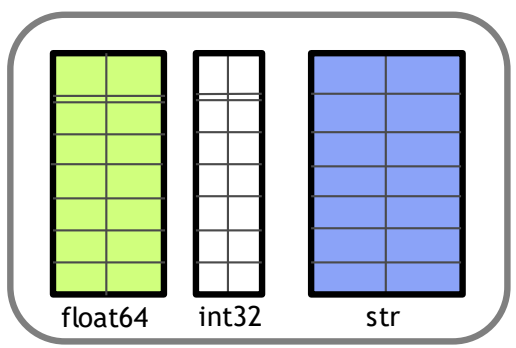


cuDF Dataframe



GPU Dataframe (GDF)

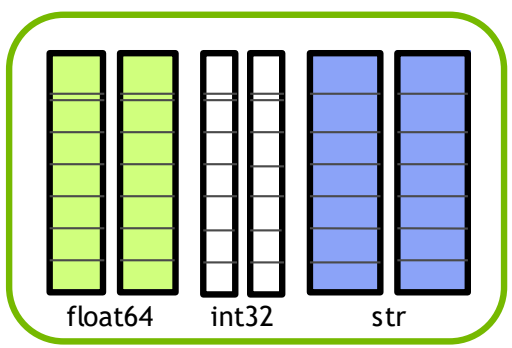
Layout in Memory



Main Memory

cuDF Dataframes are stored in GPU memory in the open-source **Apache Arrow** format (columnar layout).

APACHE ARROW



GPU Device Memory

PANDAS VS. RAPIDS CUDF

Step 1: Loading the CSV Data

Pandas

```
import pandas as pd

pdf = pd.read_csv('kdd99cup.csv')  ← Load as Pandas Dataframe

print('number of rows:', len(pdf))
```

```
number of rows: 4898431
```

```
print(pdf.dtypes.head(10))
```

```
duration            int64
protocol_type      object
service            object
flag               object
src_bytes           int64
dst_bytes           int64
land                int64
wrong_fragment      int64
urgent              int64
hot                 int64
dtype: object
```

cuDF

```
import cudf

gdf = cudf.io.read_csv('kdd99cup.csv')  ← Load as cuDF GPU Dataframe

print('number of rows:', len(gdf))
```

```
number of rows: 4898431
```

```
print(gdf.dtypes.head(10))
```

```
duration            int64
protocol_type      object
service            object
flag               object
src_bytes           int64
dst_bytes           int64
land                int64
wrong_fragment      int64
urgent              int64
hot                 int64
dtype: object
```

PANDAS VS. RAPIDS CUDF

Step 1: Loading the CSV Data

Pandas

```
import pandas as pd

pdf = pd.read_csv('kdd99cup.csv')    ← Load as Pandas Dataframe

print('number of rows:', len(pdf))

number of rows: 4898431

print(pdf.dtypes.head(10))

duration            int64
protocol_type      object
service            object
flag               object
src_bytes           int64
dst_bytes           int64
land                int64
wrong_fragment      int64
urgent              int64
hot                 int64
dtype: object
```

Conversion between Pandas and cuML GPU Dataframe:

```
gdf = cudf.from_pandas(pdf)
pdf = gdf.to_pandas()
```

cuDF

```
import cudf

gdf = cudf.io.read_csv('kdd99cup.csv')    ← Load as cuDF GPU Dataframe

print('number of rows:', len(gdf))

number of rows: 4898431

print(gdf.dtypes.head(10))

duration            int64
protocol_type      object
service            object
flag               object
src_bytes           int64
dst_bytes           int64
land                int64
wrong_fragment      int64
urgent              int64
hot                 int64
dtype: object
```

PANDAS VS. RAPIDS CUDF

Step 2a: Data Preparation and Analysis

Pandas

```
pdf['total_bytes'] = pdf.src_bytes + pdf.dst_bytes
```

Column expressions and assignments

```
pdf[pdf.total_bytes > 1e6].duration.describe()
```

```
count      539.000000
mean      1249.135436
std       4549.945721
min          0.000000
25%          2.000000
50%         12.000000
75%         59.000000
max      39930.000000
Name: duration, dtype: float64
```

Boolean Indexing

cuDF

```
gdf['total_bytes'] = gdf.src_bytes + gdf.dst_bytes
```

```
print(gdf[gdf.total_bytes > 1e6].duration.describe())
```

```
   stats     duration
0  count        539.0
1   mean  1249.135436
2    std  4549.945721
3    min          0.0
4    25%          2.0
5    50%         12.0
6    75%         59.0
7    max      39930.0
```

Joins

Dataframe from Lists

```
pdf_services = pd.DataFrame()
pdf_services['service'] = ['http', 'ftp', 'ssh', 'telnet']
pdf_services['port'] = [80, 21, 22, 23]
pdf = pdf.merge(pdf_services)
print(pdf.head(20))
```

Joins

```
gdf_services = cudf.DataFrame()
gdf_services['service'] = ['http', 'ftp', 'ssh', 'telnet']
gdf_services['port'] = [80, 21, 22, 23]
gdf = gdf.merge(gdf_services)
print(gdf.head(20))
```

Natural Inner-Join (here on 'service' column)

PANDAS VS. RAPIDS CUDF

Step 2b: Select only traffic on port 80 (http) and subset of columns

Pandas

```
c_names = ['duration', 'src_bytes', 'dst_bytes', 'count', 'srv_count', 's
           'srv_serror_rate', 'rerror_rate', 'srv_rerror_rate', 'same_srv
           'dst_host_count', 'dst_host_srv_count', 'dst_host_same_srv_rat
           'dst_host_same_src_port_rate', 'dst_host_srv_diff_host_rate',
            'dst_host_rerror_rate', 'dst_host_srv_rerror_rate']
```

```
pdf_http = pdf[pdf.port == 80][c_names]
```

```
len(pdf_http)
```

```
623091
```

cuDF

```
c_names = ['duration', 'src_bytes', 'dst_bytes', 'count', 'srv_count', 's
           'srv_serror_rate', 'rerror_rate', 'srv_rerror_rate', 'same_srv
           'dst_host_count', 'dst_host_srv_count', 'dst_host_same_srv_rat
           'dst_host_same_src_port_rate', 'dst_host_srv_diff_host_rate',
            'dst_host_rerror_rate', 'dst_host_srv_rerror_rate']
```

```
gdf_http = gdf[gdf.port == 80][c_names]
```

```
len(gdf_http)
```

```
623091
```

PANDAS VS. RAPIDS CUDF

Step 3: Feature Engineering

Pandas

Take the first 20,000 rows and standardize to zero-mean and unit variance.

```
import numpy as np
pdf_http = pdf_http.head(20000)
for col in pdf_http.columns:
    series = pdf_http[col].astype('float32')
    mean = series.mean()
    var = series.var()
    pdf_http[col] = (series - mean) / (1 if var == 0 else np.sqrt(var))
```

cuDF

Take the first 20,000 rows and standardize to zero-mean and unit variance.

```
import numpy as np
gdf_http = gdf_http.head(20000)
for col in gdf_http:
    series = gdf_http[col].astype('float32')
    mean, var = series.mean_var()
    gdf_http[col] = (series - mean) / (1 if var == 0 else np.sqrt(var))
```

Simple Feature Engineering

Convert all columns to the same NumPy data type (float32)

Standardize columns to zero-mean and unit variance (`sklearn.preprocessing.StandardScaler`)

SCIKIT-LEARN VS. RAPIDS CUML

Step 4: Build ML Model

scikit-learn

```
from sklearn.cluster import DBSCAN
clustering = DBSCAN(eps=4, min_samples=100, n_jobs=-1)
```

use all logical CPU cores

```
%%time
clustering.fit(pdf_http)
```

```
CPU times: user 7min 7s, sys: 3.55 s, total: 7min 10s
Wall time: 8.8 s
```

```
pd.Series(clustering.labels_).value_counts()
```

```
 0    19196
 1      576
-1      228
dtype: int64
```

cuML

```
import cuml
clustering = cuml.DBSCAN(eps=4, min_samples=100)
```

```
%%time
clustering.fit(gdf_http)
```

```
CPU times: user 204 ms, sys: 40 ms, total: 244 ms
Wall time: 242 ms
<cuml.cluster.dbscan.DBSCAN at 0x7f4f7c03d978>
```

```
print(clustering.labels_.value_counts())
```

```
0    19563
1      303
-1     134
dtype: int64
```

Density-based Clustering with DBSCAN:

cuML Speedup 36x
on NVIDIA DGX-1
- 2x Intel E5-2698v4 (40 cores)
- 8x Tesla V100 GPUs (only 1 used)

Variance in results due to different algorithms. cuML implementation based on [1] and [2]

[1] Andrade et al. *G-DBSCAN: A GPU Accelerated Algorithm for Density-based Clustering*. In ICCS 2013.
[2] Hawick et al. *Parallel graph component labelling with GPUs and CUDA*. In Parallel Computing 36 (2010).

RAPIDS CUML

Accelerated Algorithms available in v0.7

RAPIDS + XGBoost Integration

Dmlc XGBoost

Algorithm	Scale
DBSCAN (density-based clustering)	Single GPU
K-Means Clustering	Single GPU
K-Nearest Neighbors (uses Facebook Faiss)	Multi-GPU
Linear Regression (ordinary LSQ)	Single-GPU, Multi-GPU with conda cuda10 and dask-cuml
Linear Regression with Lasso, Ridge, and Elastic-Net	Single-GPU
Linear Regression with Elastic-Net	Single-GPU
Linear Kalman Filter	Single-GPU
Principal Component Analysis	Single-GPU
Truncated Singular Value Decomposition (tSVD)	Single-GPU, Multi-GPU with conda cuda10
UMAP: Uniform Manifold Approximation and Projection	Single-GPU
Coordinate Descent	Single-GPU
Stochastic Gradient Descent	Single-GPU

DISTRIBUTION WITH DASK

https://dask.org

Single-Node Multi-GPU

Dask is a parallel compute platform in the Python ecosystem.

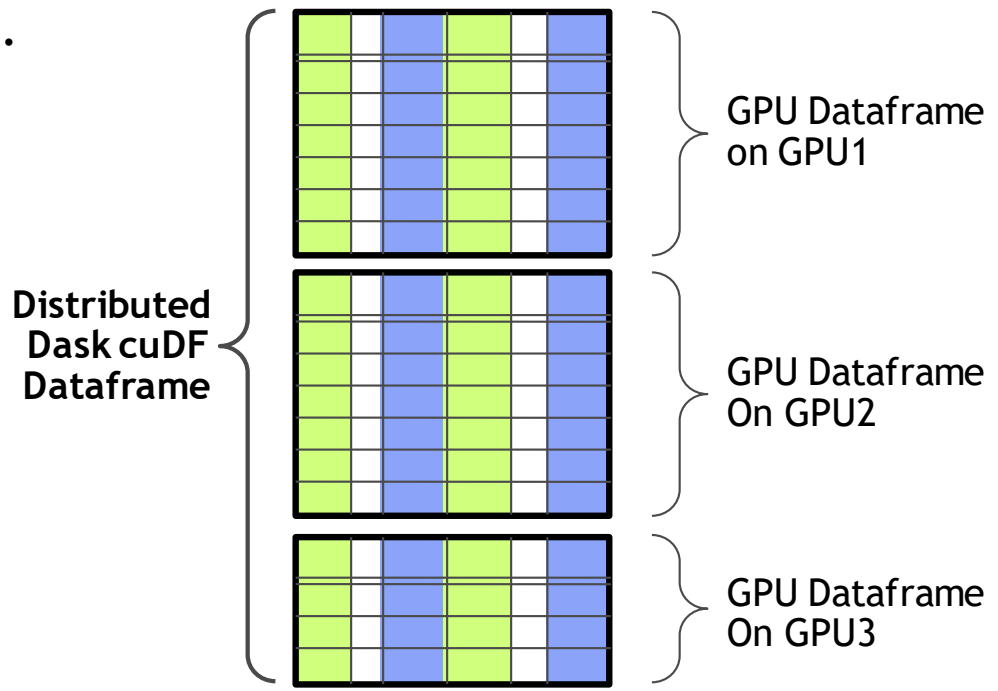
It supports distributed Dataframes and NumPy arrays.

dask-cudf supports partitioned cuDF Dataframes

dask-cuml provides multi-GPU ML algorithms

Currently supported:

- Nearest Neighbors
- Linear Regression



GPU-ACCELERATED DATA SCIENCE PLATFORMS

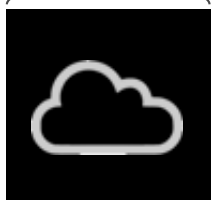
Unparalleled Performance and Productivity

ML in the Cloud
All the top CSPs

ML Enthusiast
High-end PCs

Enterprise Desktop
Individual Workstations

Enterprise Data Center
Shared Infrastructure for Data Science Teams



Max Flexibility **Max Performance**

	NVIDIA GPUs in the Cloud	GeForce	TITAN RTX	NVIDIA-Powered Data Science Workstations	T4 Enterprise Servers	DGX Station, DGX-1 / HGX-1	DGX-2 / HGX-2
Benefit	Ease of getting started, low/no barrier to entry, elasticity of resources	Enthusiast PC solution, easy to acquire, low cost, great performance	The ultimate PC GPU for data scientists. Easy to acquire, deploy and get started experimenting.	Enterprise workstation for experienced data scientists	Standard GPU-accelerated data center infrastructures with the world's leading servers	Enterprise server, proven 4 or 8-way configuration, modular approach for scale-up, fastest multi-GPU & multi-node training	Largest compute and memory capacity in a single node, fastest training solution
Typical GPU Memory (system dependent)	varies depending on offering	22GB	48GB	96GB	64 GB (4 x 16 GB)	128GB-256GB	512GB
GPU Fabric	varies depending on offering	2-way NVLink	2-way NVLink	2-way NVLink	PCIe 3.0	4- and 8-way NVLink	16-way NVSwitch

GETTING STARTED WITH RAPIDS

https://rapids.ai/start.html#get-rapids

Install on personal Linux Workstation/Laptop (with Pascal GPU or later):

```
conda install -c defaults -c nvidia -c rapidsai -c pytorch \
  -c numba -c conda-forge cudf=0.7 cuml=0.7 python=3.7 cudatoolkit=10.0
```

Run in Jupyter Notebook from RAPIDS Docker container (requires Docker CE v18+ and NVIDIA-docker v2+)

- DockerHub: `docker pull rapidsai/rapidsai`
- ngc.nvidia.com: NVIDIA GPU Cloud, GPU-optimized software hub

Run Jupyter Notebook in the Cloud

- Azure Machine Learning Service
- Google Cloud Platform (experimental version in *Deep Learning Virtual Machine* images)
- Google Collaboratory


NVIDIA