RAPIDS
End-to-End Accelerated GPU Data Science

Data Preparation ➔ Model Training ➔ Visualization

- cuDF
cuIO
Analytics
- cuML
Machine Learning
- cuGraph
Graph Analytics
- PyTorch Chainer MxNet
Deep Learning
- cuxfilter <> pyViz
Visualization

GPU Memory ➔ Apache Arrow

Dask
Data Processing Evolution
Faster data access, less data movement

Hadoop Processing, Reading from disk

Spark In-Memory Processing

Traditional GPU Processing

25-100x Improvement
Less code
Language flexible
Primarily In-Memory

5-10x Improvement
More code
Language rigid
Substantially on GPU
Data Movement and Transformation

The bane of productivity and performance
Data Movement and Transformation

What if we could keep data on the GPU?
Learning from Apache Arrow

- Each system has its own internal memory format
- 70-80% computation wasted on serialization and deserialization
- Similar functionality implemented in multiple projects

- All systems utilize the same memory format
- No overhead for cross-system communication
- Projects can share functionality (e.g., Parquet-to-Arrow reader)

From Apache Arrow Home Page - https://arrow.apache.org/
# Data Processing Evolution

**Faster data access, less data movement**

### Hadoop Processing, Reading from disk

<table>
<thead>
<tr>
<th></th>
<th>HDFS Read</th>
<th>HDFS Write</th>
<th>HDFS Read</th>
<th>ETL</th>
<th>HDFS Write</th>
<th>HDFS Read</th>
<th>ML Train</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Query</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Spark In-Memory Processing

<table>
<thead>
<tr>
<th></th>
<th>HDFS Read</th>
<th>Query</th>
<th>ETL</th>
<th>ML Train</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Query</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Traditional GPU Processing

<table>
<thead>
<tr>
<th></th>
<th>HDFS Read</th>
<th>GPU Read</th>
<th>CPU Write</th>
<th>Query</th>
<th>GPU Read</th>
<th>ETL</th>
<th>CPU Write</th>
<th>GPU Read</th>
<th>ML Train</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Query</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### RAPIDS

<table>
<thead>
<tr>
<th></th>
<th>Arrow Read</th>
<th>Query</th>
<th>ETL</th>
<th>ML Train</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Query</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **25-100x Improvement**
  - Less code
  - Language flexible
  - Primarily In-Memory

- **5-10x Improvement**
  - More code
  - Language rigid
  - Substantially on GPU

- **50-100x Improvement**
  - Same code
  - Language flexible
  - Primarily on GPU
Faster Speeds, Real-World Benefits

**cuIO/cuDF - Load and Data Preparation**
- 20 CPU Nodes: 2741 seconds
- 30 CPU Nodes: 1675 seconds
- 50 CPU Nodes: 715 seconds
- 100 CPU Nodes: 379 seconds
- DGX-2: 42 seconds
- 5x DGX-1: 19 seconds

**XGBoost Machine Learning**
- 20 CPU Nodes: 2290 seconds
- 30 CPU Nodes: 1956 seconds
- 50 CPU Nodes: 1999 seconds
- 100 CPU Nodes: 1948 seconds
- DGX-2: 169 seconds
- 5x DGX-1: 157 seconds

**End-to-End**
- 20 CPU Nodes: 8762 seconds
- 30 CPU Nodes: 6148 seconds
- 50 CPU Nodes: 3925 seconds
- 100 CPU Nodes: 3221 seconds
- DGX-2: 322 seconds
- 5x DGX-1: 213 seconds

**Time in seconds (shorter is better)**
- cuIO/cuDF (Load and Data Prep)
- Data Conversion
- XGBoost

**Benchmark**
- 200GB CSV dataset; Data prep includes joins, variable transformations

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

**DGX Cluster Configuration**
- 5x DGX-1 on InfiniBand network
Faster Speeds, Real-World Benefits

Improving Over Time

**cuIO/cuDF - Load and Data Preparation**

<table>
<thead>
<tr>
<th>Dataset</th>
<th>cuIO</th>
<th>cuDF v0.2</th>
<th>cuDF v0.10</th>
<th>cuDF v0.11</th>
</tr>
</thead>
<tbody>
<tr>
<td>DGX-2</td>
<td>42</td>
<td>37</td>
<td>19</td>
<td>17</td>
</tr>
</tbody>
</table>

**XGBoost Machine Learning**

<table>
<thead>
<tr>
<th>Dataset</th>
<th>cuIO</th>
<th>cuDF v0.2</th>
<th>cuDF v0.10</th>
<th>cuDF v0.11</th>
</tr>
</thead>
<tbody>
<tr>
<td>DGX-2</td>
<td>169</td>
<td>147</td>
<td>157</td>
<td>137</td>
</tr>
</tbody>
</table>

**End-to-End**

<table>
<thead>
<tr>
<th>Dataset</th>
<th>cuIO</th>
<th>cuDF v0.2</th>
<th>cuDF v0.10</th>
<th>cuDF v0.11</th>
</tr>
</thead>
<tbody>
<tr>
<td>DGX-2</td>
<td>322</td>
<td>209</td>
<td>213</td>
<td>164</td>
</tr>
</tbody>
</table>

**Benchmark**

200GB CSV dataset; Data prep includes joins, variable transformations

**CPU Cluster Configuration**

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

**DGX Cluster Configuration**

5x DGX-1 on InfiniBand network
Speed, UX, and Iteration
The Way to Win at Data Science

Winners are those who went through “more iterations” of the “loop of progress” -- going from an idea, to its implementation, to actionable results. So the winning teams are simply those able to run through this loop “faster”.

And this is were Keras gives you an edge.

François Cholet @fcholet - Apr 3
We often talk about how following UX best practices for API design makes Keras more accessible and easier to use, and how this helps beginners.

But those who stand to benefit most from good UX “aren’t” the beginners. It’s actually the very best practitioners in the world.

François Cholet @fcholet - Apr 3
Because good UX reduces the overhead (development overhead & cognitive overhead) to setting up new experiments. It means you will be able to iterate faster. You will be able to try more ideas.

And ultimately, that’s how you win competitions or get papers published.

François Cholet @fcholet - Apr 3
So I don’t think it’s mere personal preference if Kaggle champions are overwhelmingly using Keras.

Using Keras means you’re more likely to win, and inversely, those who practice the sort of fast experimentation strategy that sets them up to win are more likely to prefer Keras.

Joshua Patterson @datametrician - Apr 3
Replying to @fcholet:
This is the fundamental belief that drives RAPIDS. @nvidia GPU infrastructure is fast, people need to iterate quickly, people want a known #python interface. Combine them and you’re off to the races!

All (primary + auxiliary) ML software tools used by top-5 Kaggle teams in each competition (n=120)

- Scikit-Learn
- Keras
- LightGBM
- WildNet
- PyTorch
- TensorFlow (onNVIDIA)
- Caffe
- MxNet
- Fastai
- Caffe2
- CatBoost
- R Random Forests

Deep
Classic
RAPIDS Core
Open Source Data Science Ecosystem
Familiar Python APIs
RAPIDS
End-to-End Accelerated GPU Data Science

Data Preparation → Model Training → Visualization

Dask

cuDF cuIO Analytics → cuML Machine Learning → cuGraph Graph Analytics → PyTorch Chainer MxNet Deep Learning → cuxfilter <-> pyViz Visualization

GPU Memory

Apache Arrow
RAPIDS
Scaling RAPIDS with Dask

Data Preparation → Model Training → Visualization

Dask

cuDF cuIO
Analytics

cuML
Machine Learning

cuGraph
Graph Analytics

PyTorch Chainer MxNet
Deep Learning

cuxfilter <-> pyViz
Visualization

GPU Memory → Apache Arrow
Why Dask?

**PyData Native**
- **Easy Migration**: Built on top of NumPy, Pandas, Scikit-Learn, etc.
- **Easy Training**: With the same APIs
- **Trusted**: With the same developer community

**Deployable**
- **HPC**: SLURM, PBS, LSF, SGE
- **Cloud**: Kubernetes
- **Hadoop/Spark**: Yarn

**Easy Scalability**
- Easy to install and use on a laptop
- Scales out to thousand-node clusters

**Popular**
- Most common parallelism framework today in the PyData and SciPy community
Why OpenUCX?

Bringing hardware accelerated communications to Dask

• TCP sockets are slow!

• UCX provides uniform access to transports (TCP, InfiniBand, shared memory, NVLink)

• Alpha Python bindings for UCX (ucx-py)

• Will provide best communication performance, to Dask based on available hardware on nodes/cluster

conda install -c conda-forge -c rapidsai \ cudatoolkit=<CUDA version> ucx-proc=*=gpu ucx ucx-py
Benchmarks: Distributed cuDF Random Merge

cuDF v0.11, UCX-PY 0.11

Running on NVIDIA DGX-2:
GPU: NVIDIA Tesla V100 32GB
CPU: Intel(R) Xeon(R) CPU 8168 @ 2.70GHz

Benchmark Setup:
DataFrames: Left/Right 1x int64 column key column, 1x int64 value columns
Merge: inner
30% of matching data balanced across each partition
Scale up with RAPIDS

RAPIDS and Others
Accelerated on single GPU
NumPy -> CuPy/PyTorch/..
Pandas -> cuDF
Scikit-Learn -> cuML
Numba -> Numba

PyData
NumPy, Pandas, Scikit-Learn, Numba and many more
Single CPU core
In-memory data
Scale out with RAPIDS + Dask with OpenUCX

RAPIDS and Others
Accelerated on single GPU
NumPy -> CuPy/PyTorch/...
Pandas -> cuDF
Scikit-Learn -> cuML
Numba -> Numba

RAPIDS + Dask with OpenUCX
Multi-GPU
On single Node (DGX)
Or across a cluster

PyData
NumPy, Pandas, Scikit-Learn, Numba and many more
Single CPU core
In-memory data

Dask
Multi-core and Distributed PyData
NumPy -> Dask Array
Pandas -> Dask DataFrame
Scikit-Learn -> Dask-ML
... -> Dask Futures
RAPIDS
GPU Accelerated data wrangling and feature engineering

Data Preparation -> Model Training -> Visualization

Dask

cuDF cuIO Analytics

Dask

cuML Machine Learning

cuGraph Graph Analytics

PyTorch Chainer MxNet
Deep Learning

cuxfilter <-> pyViz Visualization

GPU Memory

Apache Arrow
GPU-Accelerated ETL

The average data scientist spends 90+% of their time in ETL as opposed to training models.
ETL: the Backbone of Data Science

libcuDF is...

CUDA C++ Library

- Table (dataframe) and column types and algorithms
- CUDA kernels for sorting, join, groupby, reductions, partitioning, elementwise operations, etc.
- Optimized GPU implementations for strings, timestamps, numeric types (more coming)
- Primitives for scalable distributed ETL

```cpp
std::unique_ptr<table> gather(table_view const& input, column_view const& gather_map, ...)
{
    // return a new table containing
    // rows from input indexed by
    // gather_map
}
```
ETL: the Backbone of Data Science

cuDf is...

Python Library

- A Python library for manipulating GPU DataFrames following the Pandas API
- Python interface to CUDA C++ library with additional functionality
- Creating GPU DataFrames from Numpy arrays, Pandas DataFrames, and PyArrow Tables
- JIT compilation of User-Defined Functions (UDFs) using Numba
Benchmarks: single-GPU Speedup vs. Pandas

cuDF v0.10, Pandas 0.24.2

Running on NVIDIA DGX-1:

GPU: NVIDIA Tesla V100 32GB
CPU: Intel(R) Xeon(R) CPU E5-2698 v4 @ 2.20GHz

Benchmark Setup:

DataFrames: 2x int32 columns key columns, 3x int32 value columns

Merge: inner

GroupBy: count, sum, min, max calculated for each value column
ETL: the Backbone of Data Science

cuDF is not the end of the story

Data Preparation → Model Training → Visualization

Dask

cuDF cuIO
cuML
Machine Learning

cuGraph
Graph Analytics

PyTorch Chainer MxNet
Deep Learning

cuxfilter <-> pyViz
Visualization

GPU Memory → Apache Arrow
ETL: the Backbone of Data Science

String Support

Current v0.12 String Support

• Regular Expressions
• Element-wise operations
  • Split, Find, Extract, Cat, Typecasting, etc...
• String GroupBys, Joins, Sorting, etc.
• Categorical columns fully on GPU
• Native String type in libcudf C++

Future v0.13+ String Support

• Extensive performance optimization
• More Pandas String API compatibility
• JIT-compiled String UDFs
Extraction is the Cornerstone

cuDf I/O for Faster Data Loading

- Follow Pandas APIs and provide >10x speedup
- CSV Reader - v0.2, CSV Writer v0.8
- Parquet Reader - v0.7, Parquet Writer v0.12
- ORC Reader - v0.7, ORC Writer v0.10
- JSON Reader - v0.8
- Avro Reader - v0.9

- GPU Direct Storage integration in progress for bypassing PCIe bottlenecks!

- Key is GPU-accelerating both parsing and decompression wherever possible

```python
import pandas, cudf
cpu_time = time.time()

cpu_read = pandas.read_csv('data/nyc/yellow_tripdata_2015-01.csv')

cpu_time = time.time()

cuda_read = cudf.read_csv('data/nyc/yellow_tripdata_2015-01.csv')

cpu_time = time.time()
du -hs data/nyc/yellow_tripdata_2015-01.csv
```

Source: Apache Crail blog: SQL Performance: Part 1 - Input File Formats
ETL is not just DataFrames!
RAPIDS
Building bridges into the array ecosystem

Data Preparation → Model Training → Visualization

Dask

cuDF cuIO Analytics → cuML Machine Learning

cuGraph Graph Analytics → PyTorch Chainer MxNet Deep Learning

cuxfilter <-> pyViz Visualization

GPU Memory

Apache Arrow
Interoperability for the Win
DLPack and __cuda_array_interface__

PYTORCH  mpi4py

mxnet

Chainer  CuPy

Numba
Interoperability for the Win
DLPack and __cuda_array_interface__

PYTORCH

mpi4py

Numba

RAPIDS
Open GPU Data Science

mxnet

Chainer

CuPy
ETL: Arrays and DataFrames

Dask and CUDA Python arrays

- Scales NumPy to distributed clusters
- Used in climate science, imaging, HPC analysis up to 100TB size
- Now seamlessly accelerated with GPUs
Benchmark: single-GPU CuPy vs NumPy

SVD Benchmark
Dask and CuPy Doing Complex Workflows
Petabyte Scale Analytics with Dask and CuPy

<table>
<thead>
<tr>
<th>Architecture</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single CPU Core</td>
<td>2hr 39min</td>
</tr>
<tr>
<td>Forty CPU Cores</td>
<td>11min 30s</td>
</tr>
<tr>
<td>One GPU</td>
<td>1min 37s</td>
</tr>
<tr>
<td>Eight GPUs</td>
<td>19s</td>
</tr>
</tbody>
</table>

3.2 PETABYTES IN LESS THAN 1 HOUR
Distributed GPU array | parallel reduction | using 76x GPUs

Cluster configuration: 20x GCP instances, each instance has:
- CPU: 1 VM socket (Intel Xeon CPU @ 2.30GHz), 2-core, 2 threads/core, 132GB mem, GbE ethernet, 950 GB disk
- GPU: 4x NVIDIA Tesla P100-16GB-PCIe (total GPU DRAM across nodes 1.22 TB)

Software: Ubuntu 18.04, RAPIDS 0.5.1, Dask=1.1.1, Dask-Distributed=1.1.1, CuPY=5.2.0, CUDA 10.0.130

https://blog.dask.org/2019/01/03/dask-array-gpus-first-steps
ETL: Arrays and DataFrames
More Dask Awesomeness from RAPIDS

https://youtu.be/gV0cykgsTPM

https://youtu.be/R5CiXti_MWo
Machine Learning
More models more problems
Problem
Data sizes continue to grow

Massive Dataset

Histories / Distributions
Dimension Reduction
Feature Selection
Remove Outliers
Sampling

Better to start with as much data as possible and explore / preprocess to scale to performance needs.

Time Increases

Hours? Days?

Iterate. Cross Validate & Grid Search. Iterate some more.

Meet reasonable speed vs accuracy tradeoff
ML Technology Stack

- Python
- Cython
- cuML Algorithms
- cuML Prims
- CUDA Libraries
- CUDA

- Dask cuML
- Dask cuDF
- cuDF
- Numpy
- Thrust
- Cub
- cuSolver
- nvGraph
- CUTLASS
- cuSparse
- cuRand
- cuBlas
Algorithms

GPU-accelerated Scikit-Learn

Classification / Regression

Inference

Clustering

Decomposition & Dimensionality Reduction

Time Series

Decision Trees / Random Forests
Linear Regression
Logistic Regression
K-Nearest Neighbors
Support Vector Machines

Random forest / GBDT inference

K-Means
DBSCAN
Spectral Clustering

Principal Components
Singular Value Decomposition
UMAP
Spectral Embedding
T-SNE

Holt-Winters
Kalman Filtering
ARIMA

More to come!

Key:
- Preexisting
- NEW or enhanced for 0.12
RAPIDS matches common Python APIs

CPU-Based Clustering

```python
from sklearn.datasets import make_moons
import pandas

X, y = make_moons(n_samples=int(1e2),
                  noise=0.05, random_state=0)
X = pandas.DataFrame({'fea%d' % i: X[:, i]
                      for i in range(X.shape[1])})
```

```python
from sklearn.cluster import DBSCAN
dbscan = DBSCAN(eps = 0.3, min_samples = 5)
dbscan.fit(X)
y_hat = dbscan.predict(X)
```
RAPIDS matches common Python APIs

**GPU-Accelerated Clustering**

```python
from sklearn.datasets import make_moons
import cudf
X, y = make_moons(n_samples=int(1e2), noise=0.05, random_state=0)
X = cudf.DataFrame(
    {'fea%d' % i: X[:, i] for i in range(X.shape[1])}
)

from cuml import DBSCAN
dbscan = DBSCAN(eps = 0.3, min_samples = 5)
dbscan.fit(X)
y_hat = dbscan.predict(X)
```
Benchmarks: single-GPU cuML vs scikit-learn

1x V100 vs. 2x 20 core CPU
cuML’s Forest Inference Library accelerates prediction (inference) for random forests and boosted decision trees:

- Works with existing saved models (XGBoost, LightGBM, scikit-learn RF cuML RF soon)
- Lightweight Python API
- Single V100 GPU can infer up to 34x faster than XGBoost dual-CPU node
- Over 100 million forest inferences per sec (with 1000 trees) on a DGX-1 for large (sparse) or dense models
• RAPIDS works closely with the XGBoost community to accelerate GBDTs on GPU
• The default rapids conda metapackage includes XGBoost
• XGBoost can seamlessly load data from cuDF dataframes
• Dask allows XGBoost to scale to arbitrary numbers of GPUs
• With the *gpu_hist* tree method, a single GPU can outpace 10s to 100s of CPUs
Road to 1.0  
February 2020 - RAPIDS 0.12

<table>
<thead>
<tr>
<th>cuML</th>
<th>Single-GPU</th>
<th>Multi-GPU</th>
<th>Multi-Node-Multi-GPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gradient Boosted Decision Trees (GBDT)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>GLM</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Logistic Regression</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Random Forest</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>K-Means</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>K-NN</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DBSCAN</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>UMAP</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Holt-Winters</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ARIMA</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>t-SNE</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Principal Components</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Singular Value Decomposition</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SVM</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## Road to 1.0

**March 2020 - RAPIDS 0.13**

<table>
<thead>
<tr>
<th>cuML</th>
<th>Single-GPU</th>
<th>Multi-Node-Multi-GPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gradient Boosted Decision Trees (GBDT)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>GLM</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Logistic Regression</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Random Forest</td>
<td></td>
<td></td>
</tr>
<tr>
<td>K-Means</td>
<td></td>
<td></td>
</tr>
<tr>
<td>K-NN</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DBSCAN</td>
<td></td>
<td></td>
</tr>
<tr>
<td>UMAP</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Holt-Winters</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ARIMA</td>
<td></td>
<td></td>
</tr>
<tr>
<td>t-SNE</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Principal Components</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Singular Value Decomposition</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SVM</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
cuGraph
Graph Analytics
More connections more insights

Data Preparation → Model Training → Visualization

Dask

- cuDF cuIO Analytics
- cuML Machine Learning
- cuGraph Graph Analytics
- PyTorch Chainer MxNet Deep Learning
- cuxfilter <> pyViz Visualization

GPU Memory

Apache Arrow
Goals and Benefits of cuGraph
Focus on Features and User Experience

**Breakthrough Performance**
- Up to 500 million edges on a single 32GB GPU
- Multi-GPU support for scaling into the billions of edges

**Seamless Integration with cuDF and cuML**
- Property Graph support via DataFrames

**Multiple APIs**
- **Python**: Familiar NetworkX-like API
- **C/C++**: lower-level granular control for application developers

**Growing Functionality**
- Extensive collection of algorithm, primitive, and utility functions
Graph Technology Stack

nvGRAPH has been Opened Sourced and integrated into cuGraph. A legacy version is available in a RAPIDS GitHub repo

* Gunrock is from UC Davis
Algorithms

GPU-accelerated NetworkX

- Community
- Components
- Link Analysis
- Link Prediction
- Traversal
- Structure
- Query Language
- Multi-GPU
- Utilities

More to come!

- Spectral Clustering
- Balanced-Cut
- Modularity Maximization
- Louvain
- Ensemble Clustering for Graphs
- Subgraph Extraction
- KCore and KCore Number
- Weakly Connected Components
- Strongly Connected Components
- Page Rank (Multi-GPU)
- Personal Page Rank
- Katz
- Jaccard
- Weighted Jaccard
- Overlap Coefficient
- Single Source Shortest Path (SSSP)
- Breadth First Search (BFS)
- Triangle Counting
- COO-to-CSR (Multi-GPU)
- Transpose

Auto-renumbering

Renumbering
Benchmarks: single-GPU cuGraph vs NetworkX

**Performance Speedup: cuGraph vs NetworkX**

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Nodes</th>
<th>Edges</th>
</tr>
</thead>
<tbody>
<tr>
<td>preferentialAttachment</td>
<td>100,000</td>
<td>999,970</td>
</tr>
<tr>
<td>caidaRouterLevel</td>
<td>192,244</td>
<td>1,218,132</td>
</tr>
<tr>
<td>coAuthorsDBLP</td>
<td>299,067</td>
<td>299,067</td>
</tr>
<tr>
<td>dblp-2010</td>
<td>326,186</td>
<td>1,615,400</td>
</tr>
<tr>
<td>citationCiteseer</td>
<td>268,495</td>
<td>2,313,294</td>
</tr>
<tr>
<td>coPapersDBLP</td>
<td>540,486</td>
<td>30,491,458</td>
</tr>
<tr>
<td>coPapersCiteseer</td>
<td>434,102</td>
<td>32,073,440</td>
</tr>
<tr>
<td>as-Skitter</td>
<td>1,696,415</td>
<td>22,190,596</td>
</tr>
</tbody>
</table>
Benchmarks: cyLouvain and SciPy PageRank
## Multi-GPU PageRank Performance

PageRank portion of the HiBench benchmark suite

<table>
<thead>
<tr>
<th>HiBench Scale</th>
<th>Vertices</th>
<th>Edges</th>
<th>CSV File (GB)</th>
<th># of GPUs</th>
<th># of CPU Threads</th>
<th>PageRank for 3 Iterations (secs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Huge</td>
<td>5,000,000</td>
<td>198,000,000</td>
<td>3</td>
<td>1</td>
<td></td>
<td>1.1</td>
</tr>
<tr>
<td>BigData</td>
<td>50,000,000</td>
<td>1,980,000,000</td>
<td>34</td>
<td>3</td>
<td></td>
<td>5.1</td>
</tr>
<tr>
<td>BigData x2</td>
<td>100,000,000</td>
<td>4,000,000,000</td>
<td>69</td>
<td>6</td>
<td></td>
<td>9.0</td>
</tr>
<tr>
<td>BigData x4</td>
<td>200,000,000</td>
<td>8,000,000,000</td>
<td>146</td>
<td>12</td>
<td></td>
<td>18.2</td>
</tr>
<tr>
<td>BigData x8</td>
<td>400,000,000</td>
<td>16,000,000,000</td>
<td>300</td>
<td>16</td>
<td></td>
<td>31.8</td>
</tr>
<tr>
<td>BigData x8</td>
<td>400,000,000</td>
<td>16,000,000,000</td>
<td>300</td>
<td></td>
<td>800*</td>
<td>5760*</td>
</tr>
</tbody>
</table>

*BigData x8, 100x 8-vCPU nodes, Apache Spark GraphX ⇒ 96 mins!
## Road to 1.0
### February 2020 - RAPIDS 0.12

<table>
<thead>
<tr>
<th>cuGraph</th>
<th>Single-GPU</th>
<th>Multi-GPU</th>
<th>Multi-Node-Multi-GPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jaccard and Weighted Jaccard</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Page Rank</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Personal Page Rank</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SSSP</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BFS</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Triangle Counting</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Subgraph Extraction</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Katz Centrality</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Betweenness Centrality</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Connected Components (Weak and Strong)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Louvain and ECG</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Spectral Clustering</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>K-Cores</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
# Road to 1.0
## March 2020 - RAPIDS 0.13

<table>
<thead>
<tr>
<th>cuGraph</th>
<th>Single-GPU</th>
<th>Multi-Node-Multi-GPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jaccard and Weighted Jaccard</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Page Rank</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Personal Page Rank</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SSSP</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BFS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Triangle Counting</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Subgraph Extraction</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Katz Centrality</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Betweenness Centrality</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Connected Components (Weak and Strong)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Louvain and ECG</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Spectral Clustering</td>
<td></td>
<td></td>
</tr>
<tr>
<td>K-Cores</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
cuSpatial
cuSpatial

Breakthrough Performance & Ease of Use
- Up to 1000x faster than CPU spatial libraries
- Python and C++ APIs for maximum usability and integration

Growing Functionality
- Extensive collection of algorithm, primitive, and utility functions for spatial analytics

Seamless Integration into RAPIDS
- cuDF for data loading, cuGraph for routing optimization, and cuML for clustering are just a few examples
cuSpatial Technology Stack

- Python
- Cython
- cuSpatial
- cuDF C++
- Thrust
- CUDA
## cuSpatial

### 0.12 and Beyond

<table>
<thead>
<tr>
<th>Layer</th>
<th>0.12 Functionality</th>
<th>Functionality Roadmap (2020)</th>
</tr>
</thead>
<tbody>
<tr>
<td>High-level Analytics</td>
<td>C++ Library w. Python bindings enabling distance, speed, trajectory similarity, trajectory clustering</td>
<td>C++ Library w. Python bindings for additional spatio-temporal trajectory clustering, acceleration, dwell-time, salient locations, trajectory anomaly detection, origin destination, etc.</td>
</tr>
<tr>
<td>Graph layer</td>
<td>cuGraph</td>
<td>Map matching, Djikstra algorithm, Routing</td>
</tr>
<tr>
<td>Query layer</td>
<td>Spatial Window</td>
<td>Nearest Neighbor,KNN, Spatiotemporal range search and joins</td>
</tr>
<tr>
<td>Index layer</td>
<td></td>
<td>Grid, Quad Tree, R-Tree, Geohash, Voronoi Tessellation</td>
</tr>
<tr>
<td>Geo-operations</td>
<td>Point in polygon (PIP), Haversine distance, Hausdorff distance, lat-lon to xy transformation</td>
<td>Line intersecting polygon, Other distance functions, Polygon intersection, union</td>
</tr>
<tr>
<td>Geo-representation</td>
<td>Shape primitives, points, polylines, polygons</td>
<td>Additional shape primitives</td>
</tr>
</tbody>
</table>
## cuSpatial

### Performance at a Glance

<table>
<thead>
<tr>
<th>cuSpatial Operation</th>
<th>Input data</th>
<th>cuSpatial Runtime</th>
<th>Reference Runtime</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Point-in-Polygon Test</strong></td>
<td>1.3+ million vehicle point locations and 27 Region of Interests</td>
<td>1.11 ms (C++) 1.50 ms (Python)</td>
<td>334 ms (C++, optimized serial)</td>
<td>301X</td>
</tr>
<tr>
<td></td>
<td></td>
<td>[Nvidia Titan V]</td>
<td>130468.2 ms (python Shapely API, serial)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>[Intel i7-7800X]</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>334 ms (C++, optimized serial)</td>
<td>130468.2 ms (python Shapely API, serial)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>[Intel i7-7800X]</td>
<td></td>
</tr>
<tr>
<td><strong>Haversine Distance Computation</strong></td>
<td>13+ million Monthly NYC taxi trip pickup and drop-off locations</td>
<td>7.61 ms (Python)</td>
<td>416.9 ms (Numba)</td>
<td>54.7X</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>[Nvidia T4]</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>[Nvidia T4]</td>
<td></td>
</tr>
<tr>
<td><strong>Hausdorff Distance Computation (for clustering)</strong></td>
<td>52,800 trajectories with 1.3+ million points</td>
<td>13.5s [Quadro V100]</td>
<td>19227.5s (Python SciPy API, serial)</td>
<td>1,400X</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>[Intel i7-6700K]</td>
<td></td>
</tr>
</tbody>
</table>
Community
Ecosystem Partners

CONTRIBUTORS

ADOPTERS

OPEN SOURCE
Building on top of RAPIDS
A bigger, better, stronger ecosystem for all

nuclio

High-Performance Serverless event and data processing that utilizes RAPIDS for GPU Acceleration

blazingSQL

GPU accelerated SQL engine built on top of RAPIDS

Streamz

Distributed stream processing using RAPIDS and Dask
BlazingSQL

RAPIDS

AI

blazingSQL
SQL Queries

cuDF
Data Preparation

cuML
Machine Learning

cuGRAPH
Graph Analytics

Apache Arrow on GPU

TPC-H SF1000 Query Times - GCS Storage
from blazingsql import BlazingContext
import cudf

bc = BlazingContext()

bc.s('bsql', bucket_name='bsql', access_key_id='<access_key>', secret_key='<secret_key>')

bc.create_table('orders', s3://bsql/orders/)

gdf = bc.sql('select * from orders').get()
RAPIDS + Nuclio

Serverless meets GPUs

https://towardsdatascience.com/python-pandas-at-extreme-performance-912912b1047c
Deploy RAPIDS Everywhere

Focused on robust functionality, deployment, and user experience

Integration with major cloud providers
Both containers and cloud specific machine instances
Support for Enterprise and HPC Orchestration Layers
5 Steps to getting started with RAPIDS

1. **Install** RAPIDS on using [Docker](https://docker.com), [Conda](https://conda.io), or [Colab](https://colab.research.google.com/)

2. **Explore** our [walk through videos](https://www.youtube.com), [blog content](https://blog.google), our [github](https://github.com), the [tutorial notebooks](https://github.com), and our [examples workflows](https://github.com),

3. **Build** your own data science workflows.

4. **Join** our community conversations on [Slack](https://www.slack.com), [Google](https://www.google.com), and [Twitter](https://twitter.com)

5. **Contribute** back. Don’t forget to ask and answer questions on [Stack Overflow](https://stackoverflow.com)
Easy Installation
Interactive Installation Guide

**METHOD**
- Conda
- Docker + Examples
- Docker + Dev Env
- Source

**RELEASE**
- Stable (0.12)
- Nightly (0.13a)

**PACKAGES**
- All Packages
- cuDF
- cuML
- cuGraph
- cuSpatial
- cuxfilter
- CLX

**LINUX**
- Ubuntu 16.04
- Ubuntu 18.04
- CentOS 7

**PYTHON**
- Python 3.6
- Python 3.7

**CUDA**
- CUDA 9.2
- CUDA 10.0
- CUDA 10.1.2
- CUDA 10.2

⚠️ **NOTE:** cuxfilter is not installed with rapids, it must be installed separately. Click on "cuxfilter" above to install.

**COMMAND**
```
conda install -c rapidsai -c nvidia -c conda-forge \
-c defaults rapids=0.12 python=3.6
```
Explore: RAPIDS Github

https://github.com/rapidsai
Explore: RAPIDS Docs
Improved and easier to use!

https://docs.rapids.ai
Explore: RAPIDS Code and Blogs

Check out our code and how we use it

**RAPIDS cuDF - GPU DataFrames**

- **NOTE:** For the latest stable README.md ensure you are on the `master` branch.

Built based on the Apache Arrow columnar memory format, cuDF is a GPU DataFrame library for loading, joining, aggregating, filtering, and otherwise manipulating data.

cuDF provides a pandas-like API that will be familiar to data engineers & data scientists, so they can use it to easily accelerate their workflows without going into details of CUDA programming.

For example, the following snippet downloads a CSV, then uses the GPU to parse it into rows and columns and run calculations:

```python
import cudf, io, requests
from io import StringIO

url = 'https://github.com/plotly/datasets/raw/master/tips.csv'
content = requests.get(url).content.decode('utf-8')

tips_df = cudf.read_csv(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())
```

**Output:**

```
size
```

**https://github.com/rapidsai**

**https://medium.com/rapids-ai**

---

**RAPIDS Release 0.8: Same Community New Freedoms**

Making more friends and building more bridges to more ecosystems, it's now easier than ever to get started with RAPIDS.

**Josh Patterson**

**gQuant—GPU Accelerated examples for Quantitative Analyst Tasks**

A simple trading strategy backtest for 5000 stocks using GPUs and getting 20X speedup

**Yi-Dong**

**Financial data modeling with RAPIDS.**

See how RAPIDS was used to place 17th in the Banca Santander Kaggle Competition

**Joe Li**

**NVIDIA GPUs and Apache Spark, One Step Closer**

RAPIDS XGBoost4j-Spark Package Now Available

**Karthikeyan Rajendran**

**When Less is More: A brief story about XGBoost feature engineering**

A glimpse into how a Data Scientist makes decisions about featuring engineering on XGBoost machine

**Nightly News: CI produces latest packages**

Release code early and often. Stay current on latest features with our nightly conda and container releases.
Explore: Notebooks Contrib

Tutorials, examples, and various E2E demos available, with Youtube explanations, code walkthroughs and use cases

| intro_tutorials | 05_Introduction_to_Dask_cuDF | This notebook shows how to work with cuDF DataFrames distributed across multiple GPUs using Dask. |
| intro_tutorials | 06_Introduction_to_Supervised_Learning | This notebook shows how to do GPU accelerated Supervised Learning in RAPIDS. |
| intro_tutorials | 07_Introduction_to_XGBoost | This notebook shows how to work with GPU accelerated XGBoost in RAPIDS. |
| intro_tutorials | 08_Introduction_to_Dask_XGBoost | This notebook shows how to work with Dask XGBoost in RAPIDS. |
| intro_tutorials | 09_Introduction_to_Dimensionality_Reduction | This notebook shows how to do GPU accelerated Dimensionality Reduction in RAPIDS. |
| intro_tutorials | 10_Introduction_to_Clustering | This notebook shows how to do GPU accelerated Clustering in RAPIDS. |

Intermediate Notebooks:

<table>
<thead>
<tr>
<th>Folder</th>
<th>Notebook Title</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>examples</td>
<td>DBSCAN_Demo_FULL</td>
<td>This notebook shows how to use DBSCAN algorithm and its GPU accelerated implementation present in RAPIDS.</td>
</tr>
<tr>
<td>examples</td>
<td>Dask_with_cuDF_and_XGBoost</td>
<td>In this notebook we show how to quickly setup Dask and train an XGBoost model using cuDF.</td>
</tr>
</tbody>
</table>
Join the Conversation

Google Groups
Docker Hub
Slack Channel
Stack Overflow
Contribute Back

Issues, feature requests, PRs, Blogs, Tutorials, Videos, QA...bring your best!

Getting Started with cuDF (RAPIDS)
Getting Started
RAPIDS Docs
New, improved, and easier to use

https://docs.rapids.ai
RAPIDS Docs

Easier than ever to get started with cuDF

10 Minutes to cuDF

Modeled after 10 Minutes to Pandas, this is a short introduction to cuDF, geared mainly for new users.

```python
(1):
import os
import numpy as np
import pandas as pd
import cudf
np.random.seed(12)

### Portions of this were borrowed from the
### cuDF cheat sheet, existing cuDF documentation,
### 10 Minutes to Pandas.
### Created November, 2018.

Object Creation

Creating a `Series`.

```python
(2):
s = cudf.Series([1,2,3,4,4])
print(s)
```

Creating a `DataFrame` by specifying values for each column.
RAPIDS

How do I get the software?

- https://github.com/rapidsai

- https://anaconda.org/rapidsai/


- https://hub.docker.com/r/rapidsai/rapidsai/
Join the Movement

Everyone can help!

Integrations, feedback, documentation support, pull requests, new issues, or code donations welcomed!
THANK YOU

Joshua Patterson
@datametrician
joshuap@nvidia.com