RAPIDS
The Platform Inside and Out
Release 0.13

Joshua Patterson - Director, RAPIDS Engineering
Data Processing Evolution

Faster data access, less data movement

Hadoop Processing, Reading from disk

Spark In-Memory Processing

Traditional GPU Processing

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

25-100x Improvement
Less code
Language flexible
Primarily In-Memory
Data Movement and Transformation

The bane of productivity and performance

CPU

APP B

Copy & Convert

APP A

GPU

APP B

Copy & Convert

GPU Data

APP A

Copy & Convert

GPU Data

APP A

Load Data
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

Spark In-Memory Processing

Traditional GPU Processing

RAPIDS

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**

<table>
<thead>
<tr>
<th>CPU Nodes</th>
<th>Time (seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>20 CPU Nodes</td>
<td>2741</td>
</tr>
<tr>
<td>30 CPU Nodes</td>
<td>1675</td>
</tr>
<tr>
<td>50 CPU Nodes</td>
<td>715</td>
</tr>
<tr>
<td>100 CPU Nodes</td>
<td>379</td>
</tr>
<tr>
<td>DGX-2</td>
<td>42</td>
</tr>
<tr>
<td>5x DGX-1</td>
<td>19</td>
</tr>
</tbody>
</table>

**XGBoost Machine Learning**

<table>
<thead>
<tr>
<th>CPU Nodes</th>
<th>Time (seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>20 CPU Nodes</td>
<td>2290</td>
</tr>
<tr>
<td>30 CPU Nodes</td>
<td>1956</td>
</tr>
<tr>
<td>50 CPU Nodes</td>
<td>1999</td>
</tr>
<tr>
<td>100 CPU Nodes</td>
<td>1948</td>
</tr>
<tr>
<td>DGX-2</td>
<td>169</td>
</tr>
<tr>
<td>5x DGX-1</td>
<td>157</td>
</tr>
</tbody>
</table>

**End-to-End**

<table>
<thead>
<tr>
<th>CPU Nodes</th>
<th>Time (seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>20 CPU Nodes</td>
<td>8762</td>
</tr>
<tr>
<td>30 CPU Nodes</td>
<td>6148</td>
</tr>
<tr>
<td>50 CPU Nodes</td>
<td>3925</td>
</tr>
<tr>
<td>100 CPU Nodes</td>
<td>3221</td>
</tr>
<tr>
<td>DGX-2</td>
<td>322</td>
</tr>
<tr>
<td>5x DGX-1</td>
<td>213</td>
</tr>
</tbody>
</table>

**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

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, Ease of Use, and Iteration

The Way to Win at Data Science
RAPIDS 0.13 Release Summary
What’s New in Release 0.13?

- **cuDF** Dataframes library python code base now ported to use the libcudf++ API, adds expanded groupby aggregations, join methods, concatenate optimizations, distributed multi column sorting and multi column hash repartitioning
- **cuML** machine learning library adds new Dask API to XGBoost 1.0, new configurable types for estimators, and multi-node, multi-GPU support for linear models (PCA, tSVD, OLS, Ridge)
- **cuGraph** graph analytics library adds betweenness centrality, K-Truss community detection, and code refactoring
- **UCX-Py** focused on resolving Multi-Node Multi-GPU InfiniBand tests
- **BlazingSQL** adds ROUND(), CASE with Strings, and for distributed queries AVG() support
- **cuSignal** adds conda install support, polyphase resampler, and new acoustics module
- **cuSpatial** adds batched cubic spline interpolation for trajectories and major documentation improvements
RAPIDS Core
Open Source Data Science Ecosystem

Familiar Python APIs

Data Preparation → Model Training → Visualization

- Pandas Analytics
- Scikit-Learn Machine Learning
- NetworkX Graph Analytics
- PyTorch Chainer MxNet Deep Learning
- Matplotlib/Plotly Visualization

Dask

CPU Memory
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
Dask
RAPIDS
Scaling RAPIDS with Dask

17

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 v0.13, UCX-PY 0.13
- Running on NVIDIA DGX-1 (8GPUs):
  - 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
cuDF
RAPIDS
GPU Accelerated data wrangling and feature engineering

Data Preparation → Model Training → Visualization

Dask

cuDF cuIO Analytics → cuML Machine Learning → cuGraph Graph Analytics

GPU Memory

PyTorch Chainer MxNet Deep Learning → cuxfilter <-> pyViz Visualization

Apache Arrow
GPU-Accelerated ETL

The average data scientist spends 90+% of their time in ETL as opposed to training models.
ETL Technology Stack

- Python
- Cython
- cuDF C++
- CUDA Libraries
- CUDA

- Dask cuDF
- cuDF
- Pandas

- Thrust
- Cub
- Jitify
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

Running on NVIDIA DGX-1:

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

Benchmark Setup:

RMM Pool Allocator Enabled

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
ETL: the Backbone of Data Science

String Support

Current v0.13 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 v0.14+ String Support

• Further performance optimization
• 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

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__
Interoperability for the Win

DLPack and \texttt{\_\_cuda\_array\_interface\_\_}
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
cuML
Machine Learning
More models more problems

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
Problem
Data sizes continue to grow

Massive Dataset

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

Histograms / Distributions

Dimension Reduction
Feature Selection

Remove Outliers

Sampling

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

Inference
- Random forest / GBDT inference

Clustering
- K-Means
- DBSCAN
- Spectral Clustering

Decomposition & Dimensionality Reduction
- Principal Components
- Singular Value Decomposition
- UMAP
- Spectral Embedding
- T-SNE

Time Series
- Holt-Winters
- Seasonal ARIMA

Cross Validation

Hyper-parameter Tuning

Key:
- Preexisting
- NEW or enhanced for 0.13

More to come!
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])})

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({f'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 and cuPy arrays
• 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
• Version 1.0 of XGBoost launched with the RAPIDS 0.13 stack.
## Road to 1.0
### March 2020 - RAPIDS 0.13

<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>Linear Regression</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

## 2020 - RAPIDS 1.0

<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>Linear Regression (regularized)</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
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

Graph Classes
- Structure
- Multi-GPU
- Utilities

Renumbering
- Auto-renumbering

Community
Components
Link Analysis
Link Prediction
Traversal
Centrality

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

More to come!
Benchmarks: single-GPU cuGraph vs NetworkX
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 V100 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>800*</td>
<td>5760*</td>
<td></td>
</tr>
</tbody>
</table>

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

**March 2020 - RAPIDS 1.0**

<table>
<thead>
<tr>
<th>cuGraph</th>
<th>Single-GPU</th>
<th>Multi-Node-Multi-GPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>PageRank</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Personal Page Rank</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Katz</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Betweenness Centrality</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Spectral Clustering</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Louvain</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ensemble Clustering for Graphs</td>
<td></td>
<td></td>
</tr>
<tr>
<td>K-Core</td>
<td></td>
<td></td>
</tr>
<tr>
<td>K-Truss</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Triangle Counting</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Connected Components (Weak and Strong)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Jaccard</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Overlap Coefficient</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Single Source Shortest Path (SSSP)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Breadth First Search (BFS)</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

## 0.13 and Beyond

<table>
<thead>
<tr>
<th>Layer</th>
<th>0.13 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, spline interpolation</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>Point-in-Polygon Test</td>
<td>1.3+ million vehicle point locations and 27 Region of Interests</td>
<td>1.11 ms (C++) 1.50 ms (Python) [Nvidia Titan V]</td>
<td>334 ms (C++, optimized serial) 130468.2 ms (python Shapely API, serial) [Intel i7-7800X]</td>
<td>301X (C++) 86,978X (Python)</td>
</tr>
<tr>
<td>Haversine Distance Computation</td>
<td>13+ million Monthly NYC taxi trip pickup and drop-off locations</td>
<td>7.61 ms (Python) [Nvidia T4]</td>
<td>416.9 ms (Numba) [Nvidia T4]</td>
<td>54.7X (Python)</td>
</tr>
<tr>
<td>Hausdorff Distance Computation (for clustering)</td>
<td>52,800 trajectories with 1.3+ million points</td>
<td>13.5s [Quadro V100]</td>
<td>19227.5s (Python SciPy API, serial) [Intel i7-6700K]</td>
<td>1,400X (Python)</td>
</tr>
</tbody>
</table>
Unlike other RAPIDS libraries, cuSignal is purely developed in Python with custom CUDA Kernels written with Numba and CuPy (notice no Cython layer).
cuSignal - Selected Algorithms

GPU-accelerated SciPy Signal

Convolve/Correlate
FFT Convolve
Convolve/Correlate 2D

Resampling - Polyphase, Upfirdn, Resample
Hilbert/Hilbert 2D
Wiener
Firwin

Chirp
Square
Gaussian Pulse

Kaiser
Blackman
Hamming
Hanning

Periodogram
Welch
Spectrogram

Convolutions
Filtering and Filter Design
Waveform Generation
Window Functions
Spectral Analysis
Wavelets
Peak Finding

More to come!
### Speed of Light Performance - V100

*timeit* (7 runs) rather than *time*. Benchmarked with ~1e8 sample signals on a DGX Station

<table>
<thead>
<tr>
<th>Method</th>
<th>Scipy Signal (ms)</th>
<th>cuSignal (ms)</th>
<th>Speedup (xN)</th>
</tr>
</thead>
<tbody>
<tr>
<td>fftconvolve</td>
<td>28400</td>
<td>92.2</td>
<td>308.0</td>
</tr>
<tr>
<td>correlate</td>
<td>16800</td>
<td>48.4</td>
<td>347.1</td>
</tr>
<tr>
<td>resample</td>
<td>14700</td>
<td>51.1</td>
<td>287.7</td>
</tr>
<tr>
<td>resample_poly</td>
<td>3110</td>
<td>13.7</td>
<td>227.0</td>
</tr>
<tr>
<td>welch</td>
<td>4620</td>
<td>53.7</td>
<td>86.0</td>
</tr>
<tr>
<td>spectrogram</td>
<td>2520</td>
<td>28</td>
<td>90.0</td>
</tr>
<tr>
<td>cwt</td>
<td>46700</td>
<td>277</td>
<td>168.6</td>
</tr>
</tbody>
</table>

Learn more about cuSignal functionality and performance by browsing the [notebooks](#). 
Efficient Memory Handling

Seamless Data Handoff from cuSignal to PyTorch >=1.4

Leveraging the \_\_cuda\_array\_interface\_\_ for Speed of Light End-to-End Performance

```python
from numba import cuda
import cupy as cp
import torch
from cusignal import resample_poly

# Create CuPy Array on GPU
gpu_arr = cp.random.randn(100_000_000, dtype=cp.float32)

# Polyphase Resample
resamp_arr = resample_poly(gpu_arr, up=2, down=3, window=('kaiser', 0.5))

# Zero Copy to PyTorch
torch_arr = torch.as_tensor(resamp_arr, device='cuda')

# Confirm Pointers
print('Resample Array: ', resamp_arr.__cuda_array_interface__['data'])
print('Torch Array: ', torch_arr.__cuda_array_interface__['data'])

Resample Array: (140516096213000, False)
Torch Array: (140516096213000, False)
```

Enabling Online Signal Processing with Zero-Copy Memory

CPU <-> GPU Direct Memory Access with Numba’s Mapped Array

```python
import numpy as np
import cupy as cp
from numba import cuda

# Create CPU/GPU Shared Memory, similar to numpy.zeros()
N = 2**18
shared_arr = cusignal.get_shared_mem(N, dtype=np.complex64)
print('CPU Pointer: ', shared_arr.__cuda_array_interface__['data'])
print('GPU Pointer: ', shared_arr.__cuda_array_interface__['data'])

CPU Pointer: (190905170817648, False)
GPU Pointer: (190905170817648, False)

# Load Shared Array with Numpy Array
shared_arr = np.random.randn(N) + 3j*np.random.randn(N)

%timeit
# Perform CPU FFT
cpu_fft = np.fft.fft(shared_arr)
8 ms ± 60.2 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)

%timeit
# Perform GPU FFT
gpu_fft = cp.fft.fft(cp.asarray(shared_arr))
cp.cuda.Device(0).synchronize()
866 µs ± 38 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```
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

Load Time vs Query Execution Time
```python
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://www.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://www.example.com), our [github](https://github.com), the [tutorial notebooks](https://www.example.com/tutorials), and our [examples workflows](https://www.example.com/examples)

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://www.twitter.com)

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

Interactive Installation Guide

RAPIDS RELEASE SELECTOR

RAPIDS is available as conda packages, docker images, and from source builds. Use the tool below to select your preferred method, packages, and environment to install RAPIDS. Certain combinations may not be possible and are dimmed automatically. Be sure you’ve met the required prerequisites above and see the details below.

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

RELEASE
- Stable (0.13)
- Nightly (0.14a)

PACKAGES
- cuDF
- cuML
- cuGraph
- cuSignal
- cuSpatial
- cuXfilter

LINUX
- Ubuntu 16.04
- Ubuntu 18.04
- CentOS 7
- RHEL 7

PYTHON
- Python 3.6
- Python 3.7

CUDA
- CUDA 10.0
- CUDA 10.1.2
- CUDA 10.2

NOTE: Ubuntu 16.04/18.04 & CentOS 7 use the same `conda install` commands.

COMMAND
```
conda install -c rapidsai -c nvidia -c conda-forge
- $ defaults rapid=0.13 python=3.6
```

COPY COMMAND ✓

DETAILS BELOW
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 the 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

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
3d 19 - 7 min read

gQuant—GPU Accelerated Example for Quantitative Analysis Tasks
A simple trading strategy backtest for 5000 stocks using GPUs and getting 20X speedup

Yi Dong
3d 36 - 6 min read

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 an 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.

https://github.com/rapidsai

https://medium.com/rapids-ai
# Explore: Notebooks Contrib

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

<table>
<thead>
<tr>
<th>Folder</th>
<th>Notebook Title</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>intro_tutorials</td>
<td>05_Introduction_to_Dask_cuDF</td>
<td>This notebook shows how to work with cuDF DataFrames distributed across multiple GPUs using Dask.</td>
</tr>
<tr>
<td>intro_tutorials</td>
<td>06_Introduction_to_Supervised_Learning</td>
<td>This notebook shows how to do GPU accelerated Supervised Learning in RAPIDS.</td>
</tr>
<tr>
<td>intro_tutorials</td>
<td>07_Introduction_to_XGBoost</td>
<td>This notebook shows how to work with GPU accelerated XGBoost in RAPIDS.</td>
</tr>
<tr>
<td>intro_tutorials</td>
<td>08_Introduction_to_Dask_XGBoost</td>
<td>This notebook shows how to work with Dask XGBoost in RAPIDS.</td>
</tr>
<tr>
<td>intro_tutorials</td>
<td>09_Introduction_to_Dimensionality_Reduction</td>
<td>This notebook shows how to do GPU accelerated Dimensionality Reduction in RAPIDS.</td>
</tr>
<tr>
<td>intro_tutorials</td>
<td>10_Introduction_to_Clustering</td>
<td>This notebook shows how to do GPU accelerated Clustering in RAPIDS.</td>
</tr>
</tbody>
</table>

### 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!

How GPU Computing literally saved me at work?

Python + GPU = Power, 2 Days to 20 seconds

John Murray
@MurrayData

Comparison CPU vs GPU @rapidsai to project 100 million x,y points to lat/lon to 0.01mm accuracy. CPU 1 core c 65 mins, multicore c 13 mins, GPU #RAPIDSAI 2 seconds. I optimised the code since previous run. Dell T7910 Xeon E5-2640V4x2/NVIDIA Titan Xp cc @NvidiaAI @marc_stampfl
Getting Started
Get Started
Easier than ever to get started with cuDF

https://docs.rapids.ai
RAPIDS

How do I get the software?

- https://github.com/rapidsai
- https://anaconda.org/rapidsai/
- https://ngc.nvidia.com/registry/nvidia-rapidsai
- https://hub.docker.com/r/rapidsai/rapidsai/
Join the Movement
Everyone can help!

APACHE ARROW
https://arrow.apache.org/
@ApacheArrow

RAPIDS
https://rapids.ai
@RAPIDSAI

Dask
https://dask.org
@Dask_dev

GPU Open Analytics Initiative
http://gpuopenanalytics.com/
@GPUOAI

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

Joshua Patterson
joshuap@nvidia.com

@datametrician