RAPIDS

The Platform Inside and Out
Release 0.17

Joshua Patterson - Senior Director, RAPIDS Engineering
Why GPUs?
Numerous hardware advantages

- Thousands of cores with up to ~20 TeraFlops of general purpose compute performance
- Up to 1.5 TB/s of memory bandwidth
- Hardware interconnects for up to 600 GB/s bidirectional GPU <-> GPU bandwidth
- Can scale up to 16x GPUs in a single node

Almost never run out of compute relative to memory bandwidth!
RAPIDS
End-to-End GPU Accelerated Data Science
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

CPU

APP B

Copy & Convert

APP A

Load Data

Read Data

Copy & Convert

GPU

APP B

GPU DATA

APP A

GPU DATA
Data Movement and Transformation

What if We Could Keep Data on the GPU?

CPU

APP B

Copy & Convert

APP A

GPU

APP B

GPU DATA

GPU DATA

APP A

Load Data

Read Data

Copy & Convert

Copy & Convert

X

X

APP B

APP A
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)

Source: From Apache Arrow Home Page - https://arrow.apache.org/
Data Processing Evolution
Faster Data Access, Less Data Movement

Hadoop Processing, Reading from Disk

- HDFS Read
- Query
- HDFS Write
- HDFS Read
- ETL
- HDFS Write
- HDFS Read
- ML Train

Spark In-Memory Processing

- HDFS Read
- Query
- ETL
- ML Train

Traditional GPU Processing

- HDFS Read
- GPU Read
- Query
- CPU Write
- GPU Read
- ETL
- CPU Write
- GPU Read
- ML Train

RAPIDS

- Arrow Read
- Query
- ETL
- ML Train

- 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
Lightning-fast performance on real-world use cases

GPU Big Data Benchmark (GPU-BDB) is a data science benchmark derived from TPCx-BB\(^1\), consisting of 30 end-to-end queries representing real-world ETL and Machine Learning workflows. It involves both structured and unstructured data. The benchmark starts with reading data from disk, performs common analytical and ML techniques (including NLP), then writes results back to disk to simulate a real world workflow.

Results at 10TB scale show RAPIDS’ performance increasing over time, while TCO continues to go down. The recently announced DGX-A100 640GB is perfectly suited to data science workloads, and lets us do more work in almost half as many nodes as the DGX-A100 320GB (6 nodes vs 10) for even better TCO.

Continuous Improvement

- 2.8x performance, almost a third the nodes, and cheaper to boot—in <1 year
- BlazingSQL at 10TB showing 25% improvement compared to Dask over TCP
- Q27 faster and more accurate with hugging Face

---

1: GPU-BDB is derived from the TPCx-BB benchmark and is used for internal performance testing. Results from GPU-BDB are not comparable to TPCx-BB.
Faster Speeds, Real World Benefits
Faster Data Access, Less Data Movement

cuIO/cuDF - Load and Data Preparation

<table>
<thead>
<tr>
<th>Nodes</th>
<th>Time (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>20 CPU</td>
<td>2,740</td>
</tr>
<tr>
<td>30 CPU</td>
<td>1,675</td>
</tr>
<tr>
<td>50 CPU</td>
<td>715</td>
</tr>
<tr>
<td>100 CPU</td>
<td>379</td>
</tr>
<tr>
<td>16x A100</td>
<td>32</td>
</tr>
</tbody>
</table>

XGBoost Machine Learning

<table>
<thead>
<tr>
<th>Nodes</th>
<th>Time (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>20 CPU</td>
<td>2,290</td>
</tr>
<tr>
<td>30 CPU</td>
<td>1,956</td>
</tr>
<tr>
<td>50 CPU</td>
<td>1,999</td>
</tr>
<tr>
<td>100 CPU</td>
<td>1,948</td>
</tr>
<tr>
<td>16x A100</td>
<td>99</td>
</tr>
</tbody>
</table>

End-to-End

<table>
<thead>
<tr>
<th>Nodes</th>
<th>Time (secs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>20 CPU</td>
<td>8,763</td>
</tr>
<tr>
<td>30 CPU</td>
<td>6,147</td>
</tr>
<tr>
<td>50 CPU</td>
<td>3,926</td>
</tr>
<tr>
<td>100 CPU</td>
<td>3,221</td>
</tr>
<tr>
<td>16x A100</td>
<td>146</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

A100 Cluster Configuration
16 A100 GPUs (40GB each)

RAPIDS Version
RAPIDS 0.17
Faster Speeds, Real World Benefits
Getting faster over time--and even better with A100 and RAPIDS 0.17

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

<table>
<thead>
<tr>
<th>System</th>
<th>Time (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>DGX-2 RAPIDS v0.2</td>
<td>42</td>
</tr>
<tr>
<td>DGX-2 RAPIDS v0.10</td>
<td>37</td>
</tr>
<tr>
<td>16x A100 RAPIDS v0.17</td>
<td>32</td>
</tr>
</tbody>
</table>

**XGBoost Machine Learning**

<table>
<thead>
<tr>
<th>System</th>
<th>Time (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>DGX-2 RAPIDS v0.2</td>
<td>169</td>
</tr>
<tr>
<td>DGX-2 RAPIDS v0.10</td>
<td>147</td>
</tr>
<tr>
<td>16x A100 RAPIDS v0.17</td>
<td>75</td>
</tr>
</tbody>
</table>

**End-to-End**

<table>
<thead>
<tr>
<th>System</th>
<th>Time (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>DGX-2 RAPIDS v0.2</td>
<td>209</td>
</tr>
<tr>
<td>DGX-2 RAPIDS v0.10</td>
<td>209</td>
</tr>
<tr>
<td>16x A100 RAPIDS v0.17</td>
<td>122</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

**A100 Cluster Configuration**

16 A100 GPUs (40GB each)
Speed, UX, and Iteration
The Way to Win at Data Science
RAPIDS 0.17 Release Summary
What’s New in Release 0.17?

- **cuDF** adds initial support for decimal types in Python, and nested types support continues to improve. The team also released a number of bug fixes and under-the-hood improvements.

- **cuML** adds sparse data support for kNN and UMAP, an experimental accelerated SHAP model explainability approach, LARS (Least Angle Regression), MNMG Logistic Regression via dask-glm, and major internal refactorings.

- **XGBoost** 1.3.0 ships with 0.17, including GPU-accelerated TreeSHAP explainability, Dask API improvements, and more.

- **cuGraph** adds multi-node multi-GPU version of Katz centrality; single GPU versions of Minimum, and Maximum, Spanning Trees (MST), and the Hungarian algorithm. Expanded compatibility to include SciPy and CuPy as well as better Pandas and Numpy support.

- **UCX-Py** focused on code cleanup and documentation improvements.

- **BlazingSQL** adds new string functions (REPLACE, TRIM, UPPER, and others). Additionally, a new communication layer improving distributed performance and supporting UCX has been added.

- **cuxfilter** adds datetime dtype and the datetime slider widget for time series brushing, in addition to performance improvements.

- **RMM** adds new stream wrapper classes for C++ and various documentation improvements (Python docs now online!).
Exactly as it sounds—our goal is to make RAPIDS as usable and performant as possible wherever data science is done. We will continue to work with more open source projects to further democratize acceleration and efficiency in data science.
RAPIDS Core
Open Source Data Science Ecosystem
Familiar Python APIs

Data Preparation → Model Training → Visualization

Dask

Pandas
Scikit-Learn
NetworkX
PyTorch,
TensorFlow,
MxNet
Matplotlib

Analytics
Machine Learning
Graph Analytics
Deep Learning
Visualization

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, TensorFlow, MxNet Deep Learning

cuxfilter, pyViz, plotly Visualization

GPU Memory

Apache Arrow
Dask
RAPIDS
Scaling RAPIDS with Dask

Data Preparation → Model Training → Visualization

Dask

cuDF cuIO Analytics → cuML Machine Learning → cuGraph Graph Analytics → PyTorch, TensorFlow, MxNet Deep Learning → cuxfilter, pyViz, plotly Visualization

GPU Memory

Apache Arrow
Why Dask?

**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
- Modularly built for acceleration

**PYDATA NATIVE**
- Easy Migration: Built on top of NumPy, Pandas Scikit-Learn, etc
- Easy Training: With the same API

**POPULAR**
- Most Common parallelism framework today in the PyData and SciPy community
- Millions of monthly Downloads and Dozens of Integrations

**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

Scale Out / Parallelize
TCP sockets are slow!

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

Python bindings for UCX (ucx-py)

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

Easy to use!

conda install -c conda-forge -c rapidsai \ cudatoolkit=<CUDA version> ucx-proc=*=gpu ucx ucx-py

cluster = LocalCUDACluster(protocol='ucx' \ enable_infiniband=True, \ enable_nvlink=True)

client = Client(cluster)

Why OpenUCX?
Bringing Hardware Accelerated Communications to Dask

NVIDIA DGX-2 Inner join Benchmark

Comms Type
- DGX2 NV
- IB+NV
- IB
- NV
- TCP-UCX

37.7
17.4
11.7
3.9
1.9

cuDF Merge Bandwidth GB/s
Scale Up with RAPIDS

RAPIDS AND OTHERS
Accelerated on single GPU

- NumPy -> CuPy/PyTorch/..
- Pandas -> cuDF
- Scikit-Learn -> cuML
- NetworkX -> cuGraph
- Numba -> Numba

PYDATA
NumPy, Pandas, Scikit-Learn, NetworkX, 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
- NetworkX -> cuGraph
- Numba -> Numba

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

**RAPIDS + DASK WITH OPENUCX**
Multi-GPU
On single Node (DGX)
Or across a cluster

**DASK**
Multi-core and distributed PyData
- NumPy -> Dask Array
- Pandas -> Dask DataFrame
- Scikit-Learn -> Dask-ML
... -> Dask Futures

**RAPIDS**
Scale Up / Accelerate

**DASK**
Scale Out / Parallelize

**RAPIDS + DASK**
Scale Out with RAPIDS + Dask with OpenUCX
cuDF
RAPIDS
GPU Accelerated Data Wrangling and Feature Engineering

Data Preparation → Model Training → Visualization

Dask

cuDF cuIO
Analytics

cuML
Machine Learning

cuGraph
Graph Analytics

PyTorch, TensorFlow, MXNet
Deep Learning

cuxfilter, pyViz, plotly
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 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

```python
In [2]:
# Read in the data. Notice how it decompresses as it reads the data into memory.
gdf = cudf.read_csv('/raptors/data/black-friday.zip')

In [3]:
# Taking a look at the data. We use `to_pandas()` to get the pretty printing.
gdf.head().to_pandas()

Out[3]:

<table>
<thead>
<tr>
<th>User_ID</th>
<th>Product_ID</th>
<th>Gender</th>
<th>Age</th>
<th>Occupation</th>
<th>City_Category</th>
<th>Stay_In_Current_City_Years</th>
<th>Marital_Status</th>
<th>Product_Cate</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1000001</td>
<td>F</td>
<td>0-17</td>
<td>A</td>
<td></td>
<td>10</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>1000001</td>
<td>F</td>
<td>0-17</td>
<td>A</td>
<td></td>
<td>10</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1000001</td>
<td>F</td>
<td>0-17</td>
<td>A</td>
<td></td>
<td>10</td>
<td>0</td>
<td>12</td>
</tr>
<tr>
<td>3</td>
<td>1000001</td>
<td>F</td>
<td>0-17</td>
<td>A</td>
<td></td>
<td>10</td>
<td>0</td>
<td>12</td>
</tr>
<tr>
<td>4</td>
<td>1000002</td>
<td>M</td>
<td>55+</td>
<td>C</td>
<td></td>
<td>16</td>
<td>4</td>
<td>8</td>
</tr>
</tbody>
</table>

In [4]:
# Grabbing the first character of the years in city string to get rid of plus sign, and converting to int
gdf['city_years'] = gdf.Stay_In_Current_City_Years.str.get(0).stol()

In [5]:
# Here we can see how we can control what the value of our dummies with the replace method and turn strings to int

gdf['City_Region'] = gdf.City_Category.str.replace('A', '1')
gdf['City_Region'] = gdf.City_Category.str.replace('B', '2')
gdf['City_Region'] = gdf.City_Category.str.replace('C', '3')
gdf['City_Region'] = gdf['City_Category'].str.stoi()
```
Benchmarks: Single-GPU Speedup vs. Pandas

cuDF v0.13, Pandas 0.25.3

- 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 V0.17 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++
- NLP Preprocessors
  - Tokenizers, Normalizers, Edit Distance, Porter Stemmer, etc.

FUTURE V0.18+ STRING SUPPORT
- Further performance optimization
- JIT-compiled String UDFs
Extraction is the Cornerstone
cuIO for Faster Data Loading

- Follow Pandas APIs and provide >10x speedup
- Multiple supported formats, including:
  - CSV Reader, CSV Writer
  - Parquet Reader, Parquet Writer
  - ORC Reader, ORC Writer
  - JSON Reader
  - Avro Reader
- GPU Direct Storage integration in progress for bypassing PCIe bottlenecks!
- Key is GPU-accelerating both parsing and decompression
- Benchmark:
  - Dataset: NY Taxi dataset (Jan 2015)
  - GPU: Single 32GB V100
  - RAPIDS Version: 0.17
ETL is Not Just DataFrames!
RAPIDS
Building Bridges into the Array Ecosystem
Interoperability for the Win

- Real-world workflows often need to share data between libraries
- RAPIDS supports device memory sharing between many popular data science and deep learning libraries
- Keeps data on the GPU--avoids costly copying back and forth to host memory
- Any library that supports DLPack or __cuda_array_interface__ will allow for sharing of memory buffers between RAPIDS and supported libraries
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

![Graph showing compute time vs. rows and columns for different configurations of Dask and CuPy.]
cuML
Machine Learning
More Models More Problems

Data Preparation → Model Training → Visualization

Dask

cuDF cuIO Analytics

cuML Machine Learning

cuGraph Graph Analytics

PyTorch, TensorFlow, MxNet Deep Learning

cuxfilter, pyViz, plotly Visualization

GPU Memory

Apache Arrow
Problem
Data Sizes Continue to Grow

MASSIVE DATASET

HISTOGRAMS / 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?

Meet Reasonable Speed vs Accuracy Trade-off

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
```
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)
```
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])})

dbscan = DBSCAN(eps=0.3, min_samples=5)
dbscan.fit(X)
y_hat = dbscan.predict(X)
Algorithms

GPU-accelerated Scikit-Learn

Classification / Regression
- Decision Trees / Random Forests
- Linear/Lasso/Ridge/ElasticNet Regression
- Logistic Regression
- K-Nearest Neighbors
- Support Vector Machine Classification and Regression
- Naive Bayes

Inference
- Random Forest / GBDT Inference (FIL)

Preprocessing
- Text vectorization (TF-IDF / Count)
- Target Encoding
- Cross-validation / splitting

Clustering
- K-Means
- DBSCAN
- Spectral Clustering
- Principal Components
- Singular Value Decomposition
- UMAP
- Spectral Embedding
- T-SNE

Decomposition & Dimensionality Reduction
- Holt-Winters
- Seasonal ARIMA / Auto ARIMA

More to come!
Benchmarks: Single-GPU cuML vs Scikit-learn

1x V100 vs. 2x 20 Core CPUs (DGX-1, RAPIDS 0.15)
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
XGBoost + RAPIDS: Better Together

- RAPIDS comes paired with XGBoost 1.3.0
- XGBoost now builds on the GoAI interface standards to provide zero-copy data import from cuDF, cuPY, Numba, PyTorch and more
- Official Dask API makes it easy to scale to multiple nodes or multiple GPUs
- gpu_hist tree builder delivers huge perf gains
  Memory usage when importing GPU data decreased by 2/3 or more
- Now supports Learning to Rank on GPU

All RAPIDS changes are integrated upstream and provided to all XGBoost users – via pypi or RAPIDS conda

XGBoost speedup on GPUs comparing a single NVIDIA V100 GPU to a dual 20-core Intel Xeon E5-2698 server
Accelerated machine learning models in RAPIDS give you the flexibility to use hyperparameter optimization (HPO) experiments to explore all variants to find the most accurate possible model for your problem.

With GPU acceleration, RAPIDS models can train 25x faster than CPU equivalents, enabling more experimentation in less time.

The RAPIDS team works closely with major cloud providers and OSS solution providers to provide code samples to get started with HPO in minutes.

https://rapids.ai/hpo
HPO Use Case: 100-Job Random Forest Airline Model

Huge speedups translate into >7x TCO reduction

Based on sample Random Forest training code from cloud-ml-examples repository, running on Azure ML. 10 concurrent workers with 100 total runs, 100M rows, 5-fold cross-validation per run.

GPU nodes: 10x Standard_NC6s_v3, 1 V100 16G, vCPU 6 memory 112G, Xeon E5-2690 v4 (Broadwell) - $3.366/hour

CPU nodes: 10x Standard_DS5_v2, vCPU 16 memory 56G, Xeon E5-2673 v3 (Haswell) or v4 (Broadwell) - $1.017/hour
SHAP Explainability

GPUTreeSHAP for XGBoost

- **SHAP** provides a principled way to explain the impact of input features on each prediction or on the model overall - critical for interpretability

- SHAP has often been too computationally-expensive to deploy for large-scale production

- RAPIDS ships with GPU-accelerated SHAP for XGBoost with speedups of 20x or more ([demo code available in the XGBoost repo](#))

- RAPIDS 0.17 includes experimental [Kernel and Permutation explainers](#) for black box models

---

**GPUTreeSHAP Speedups (1x V100 vs. 2x 20 E5-2698)**

<table>
<thead>
<tr>
<th>Dataset and model size</th>
<th>SMALL MODEL</th>
<th>MED MODEL</th>
<th>LARGE MODEL</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADULT</td>
<td>50x</td>
<td>100x</td>
<td>150x</td>
</tr>
<tr>
<td>CAL-HOUSING</td>
<td>10x</td>
<td>20x</td>
<td>30x</td>
</tr>
<tr>
<td>COVTYPE</td>
<td>5x</td>
<td>10x</td>
<td>15x</td>
</tr>
<tr>
<td>FASHION-MNIST</td>
<td>1x</td>
<td>2x</td>
<td>3x</td>
</tr>
</tbody>
</table>

---
## Road to 1.0 - cuML

**RAPIDS 0.17 - December 2020**

<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</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>
## Road to 1.0 - cuML

### 2021 Mid-Year Plan

<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</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, cuI/O, cuML
- cuGraph Graph Analytics
- PyTorch, TensorFlow, MxNet Deep Learning
- cuxfilter, pyViz, plotly 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

Scaling without changing API
- Scale without having to rewrite workflow

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

Compatibility
- Accepts NetworkX graph
- Accepts SciPy and CuPy Sparse Matrix

GROWING FUNCTIONALITY
- Extensive collection of algorithm, primitive, and utility functions
Graph Technology Stack

**Python**

**Cython**

**cuGraph Algorithms**
- Prims
- cuHornet
- Gunrock*

**CUDA Libraries**

**CUDA**

**Dask cuGraph**
**Dask cuDF**
**cuDF**

**Numpy**

**CUDA Libraries**
- Thrust
- Cub
- cuSolver
- cuSparse
- cuRand

* Gunrock is from UC Davis
Benchmarks: Single-GPU cuGraph vs NetworkX

Performance Speedup: cuGraph vs NetworkX

Dataset | Nodes | Edges
--- | --- | ---
preferentialAttachment | 100,000 | 999,970
Dblp-2010 | 326,186 | 1,615,400
coPapersCiteseer | 434,102 | 32,073,440
As-Skitter | 1,696,415 | 22,190,596
NetworkX Compatibility

Use your NetworkX.Graph Objects directly within cuGraph

```python
import networkx as nx
import time
import operator

# create a random graph
G = nx.barabasi_albert_graph(N, M)

... do some NetworkX stuff ..

t1 = time.time()
betweenness = nx.betweenness_centrality(G)
t2 = time.time() - t1

print(t2)
```

**NetworkX**

```python
import networkx as nx
import time
import operator
import cugraph as cnx

# create a random graph
G = nx.barabasi_albert_graph(N, M)

... do some NetworkX stuff ..

t1 = time.time()
betweenness = cnx.betweenness_centrality(G)
t2 = time.time() - t1

print(t2)
```

**NetworkX + RAPIDS cuGraph**

<table>
<thead>
<tr>
<th>Node</th>
<th>Edges</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>1,344</td>
<td>0.45</td>
</tr>
<tr>
<td>200</td>
<td>2,944</td>
<td>1.14</td>
</tr>
<tr>
<td>400</td>
<td>6,144</td>
<td>2.64</td>
</tr>
<tr>
<td>800</td>
<td>12,544</td>
<td>5.26</td>
</tr>
<tr>
<td>1,600</td>
<td>25,344</td>
<td>12.99</td>
</tr>
<tr>
<td>3,200</td>
<td>50,944</td>
<td>26.5</td>
</tr>
<tr>
<td>6,400</td>
<td>102,144</td>
<td>48.62</td>
</tr>
<tr>
<td>12,800</td>
<td>204,544</td>
<td>89.81</td>
</tr>
<tr>
<td>25,600</td>
<td>409,344</td>
<td>180.42</td>
</tr>
<tr>
<td>51,200</td>
<td>818,944</td>
<td>328.05</td>
</tr>
</tbody>
</table>
## 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>800*</td>
<td></td>
<td>5760*</td>
</tr>
</tbody>
</table>

*BigData x8, 100x 8-vCPU nodes, Apache Spark GraphX \( \Rightarrow \) 96 mins!
## Road to 1.0
RAPIDS 0.17 - October 2020

<table>
<thead>
<tr>
<th>cuGRAPH</th>
<th>Single-GPU</th>
<th>Multi-Node-Multi-GPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>Page Rank</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-Truss &amp; K-Core</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Connected Components (Weak &amp; Strong)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Triangle Counting</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>
<tr>
<td>Jaccard &amp; Overlap Coefficient</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Force Atlas 2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Hungarian Algorithm</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Leiden</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## Road to 1.0
### 2021 Mid-Year Plan

<table>
<thead>
<tr>
<th>cuGRAPH</th>
<th>Single-GPU</th>
<th>Multi-Node-Multi-GPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>Page Rank</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-Truss &amp; K-Core</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Connected Components (Weak &amp; Strong)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Triangle Counting</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>
<tr>
<td>Jaccard &amp; Overlap Coefficient</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Force Atlas 2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Hungarian Algorithm</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Leiden</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
cuSpatial
cuSpatial Technology Stack

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

**Current and planned functionality**

<table>
<thead>
<tr>
<th>Layer</th>
<th>0.17 Functionality</th>
<th>Functionality Roadmap</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>High-level Analytics</strong></td>
<td>C++ Library w. Python bindings enabling distance, speed, trajectory similarity, trajectory clustering</td>
<td>Symmetric segment path distance</td>
</tr>
<tr>
<td><strong>Graph Layer</strong></td>
<td>cuGraph</td>
<td>cuGraph</td>
</tr>
<tr>
<td><strong>Query Layer</strong></td>
<td>Spatial Window, Nearest polyline</td>
<td>Nearest Neighbor, Spatiotemporal range search and joins</td>
</tr>
<tr>
<td><strong>Index Layer</strong></td>
<td>Quadtree</td>
<td></td>
</tr>
<tr>
<td><strong>Geo-operations</strong></td>
<td>Point in polygon (PIP), Haversine distance, Hausdorff distance, lat-lon to xy transformation</td>
<td>ST_distance and ST_contains</td>
</tr>
<tr>
<td><strong>Geo-Representation</strong></td>
<td>Shape primitives, points, polylines, polygons</td>
<td>Fiona/Geopandas I/O support and object representations</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) 130468.2 ms (python Shapely API, serial)</td>
<td>301X (C++) 86,978X (Python)</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) 416.9 ms (Numba)</td>
<td>416.9 ms (Numba)</td>
<td>54.7X (Python)</td>
</tr>
<tr>
<td><strong>Hausdorff Distance Computation</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 (Python)</td>
</tr>
</tbody>
</table>
cuSignal
cuSignal Technology Stack

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

- Convolution
  - Convolve/Correlate
  - FFT Convolve
  - Convolve/Correlate 2D
- Filtering and Filter Design
- Waveform Generation
- Window Functions
- Wavelets
- Spectral Analysis
  - Periodogram
  - Welch
  - Spectrogram
- Peak Finding
- Resampling - Polyphase, Upfirdn, Resample
- Hilbert/Hilbert 2D
- Wiener
- Firwin
- Chirp
- Square
- Gaussian Pulse
- Kaiser
- Blackman
- Hamming
- Hanning

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>27300</td>
<td>85.1</td>
<td>320.8</td>
</tr>
<tr>
<td>correlate</td>
<td>4020</td>
<td>47.4</td>
<td>84.8</td>
</tr>
<tr>
<td>resample</td>
<td>14700</td>
<td>45.9</td>
<td>320.2</td>
</tr>
<tr>
<td>resample_poly</td>
<td>2360</td>
<td>8.29</td>
<td>284.6</td>
</tr>
<tr>
<td>welch</td>
<td>4870</td>
<td>45.5</td>
<td>107.0</td>
</tr>
<tr>
<td>spectrogram</td>
<td>2520</td>
<td>23.3</td>
<td>108.1</td>
</tr>
<tr>
<td>convolve2d</td>
<td>8410</td>
<td>9.92</td>
<td>847.7</td>
</tr>
</tbody>
</table>

Learn more about cuSignal functionality and performance by browsing the [notebooks](#).
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: (14051609213088, False)
Torch Array: (14051609213088, 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: (130806170817400, False)
GPU Pointer: (130806170817400, False)

# Load Shared Array with Numpy Array
shared_arr = np.random.randn(N) + 1j*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().synchronize()
866 µs ± 38 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```
Cyber Log Accelerators
Built using RAPIDS and GPU-accelerated platforms

Targeted towards Senior SOC (Security Operations Center) Analysts, InfoSec Data Scientists, Threat Hunters, and Forensic Investigators

Notebooks geared towards info sec and cybersecurity data scientists and data engineer

SIEM integrations that enable easy data import/export and data access

Workflow and I/O components that enable users to instantiate new use cases while

Cyber-specific primitives that provide accelerated functions across cyber data types
CLX Technology Stack

CLX Applications / Use Cases

Python

Cython

RAPIDS

GPU Packages

CUDA Libraries

CUDA

Security Products and SIEMs
CLX Contains Various Use Cases and Connectors
Example Notebooks Demonstrate RAPIDS for Cybersecurity Applications

<table>
<thead>
<tr>
<th>CLX</th>
<th>Type</th>
<th>Proof-of-Concept</th>
<th>Stable</th>
</tr>
</thead>
<tbody>
<tr>
<td>DGA Detection</td>
<td>Use Case</td>
<td></td>
<td>Improved!</td>
</tr>
<tr>
<td>Network Mapping</td>
<td>Use Case</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Asset Classification</td>
<td>Use Case</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Phishing Detection</td>
<td>Use Case</td>
<td></td>
<td>Improved!</td>
</tr>
<tr>
<td>Security Alert Analysis</td>
<td>Use Case</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EDA for Cyber</td>
<td>Use Case</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Splunk Integration</td>
<td>Integration</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CLX Query</td>
<td>Integration</td>
<td></td>
<td></td>
</tr>
<tr>
<td>cyBERT</td>
<td>Log Parsing</td>
<td></td>
<td></td>
</tr>
<tr>
<td>GPU Subword Tokenizer</td>
<td>Pre-Processing</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Accelerated IPv4</td>
<td>Primitive</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Accelerated DNS</td>
<td>Primitive</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
cyBERT
AI Log Parsing for Known, Unknown, and Degraded Cyber Logs

- Provide a flexible method that does not use heuristics/regex to parse cybersecurity logs
- Parsing with micro-F1 and macro-F1 > 0.999 across heterogeneous log types with a validation loss of < 0.0048
- Second version \(~160x\) (min) faster than first version due to creation of the first all-GPU subword tokenizer that supports non-truncation of logs/sentences
- Streaming-ready version now available on the CLX GitHub repo (uses cuStreamz and accelerated Kafka reading)
GPU SubWord Tokenizer
Fully On-GPU Pre-Processing for BERT Training/Inference

- Only wordpiece tokenizer that supports non-truncation of logs/sentences
- Returns encoded tensor, attention mask, and metadata to reform broken logs
- Supports stride/overlap
- Ready for immediate pipelining into PyTorch for inference
- Over 316x faster than Python-based Hugging Face tokenizer
- Nearly 17x faster than new Rust-based Hugging Face tokenizer

SubWord Tokenizer Speedup Comparison

CPU numbers ran on 2x Intel Xeon E5 v4 @ 2.2 GHz. GPU numbers ran on 1x NVIDIA Tesla V100
CLX Query

Query Long-Term Data Store Directly from Splunk
Visualization
Cuxfilter allows you to visually explore your cuDF dataframes through fully cross-filtered dashboards in less than 10 lines of notebook code.

Select from vetted chart libraries, pre-designed layouts, and multiple themes to find the best fit for a use case, at speeds typically 20x faster per chart than Pandas.

Cuxfilter is designed for easy install and use with RAPIDS. Learn more about our approach [here](https://docs.rapids.ai/api/cuxfilter/stable).
Plotly Dash is RAPIDS accelerated, combining the ease-of-use development and deployment of Dash apps with the compute speed of RAPIDS. Work continues to optimize Plotly’s API for even deeper RAPIDS integration.

300 MILLION DATAPoint CENSUS EXAMPLE

Interact with data points of every individual in the United States, in real time, with the 2010 Census visualization. Get the code on GitHub and read about details here.

https://github.com/rapidsai/plotly-dash-rapids-census-demo
pyViz Community
GPU Accelerated Python Visualizations

**HoloViews**
Uses cuDF to easily annotate and interactively explore data with minimal syntax. Work continues to optimize its API for deeper RAPIDS integration.
holoviews.org

**Datashader**
Uses cuDF / dask cuDF for accelerated server-side rendering of extremely high density visualizations. Work continues to optimize more of its features for GPUs.
datashader.org

**hvPlot**
A higher-level plotting API built on HoloViews, work is ongoing to ensure its features can utilize the RAPIDS integration with HoloViews.
hvplot.holoviz.org

**bokeh**
A backbone chart library used throughout the pyViz community, RAPIDS is actively supporting integration development to further its use in GPU accelerated libraries and enhance rendering performance.
bokeh.org

https://datashader.org/
Ecosystem Partners

CONTRIBUTORS
- Anaconda
- CapitalOne
- CuPy
- Chainer
- UNROCK
- Quansight
- Walmart

ADOPTERS
- Booz Allen Hamilton
- CapitalOne
- Databricks
- Graphistry
- H2O.ai
- IBM
- Igusio
- Inria
- Kinetica
- MAPR
- Omnisci
- Preferred Networks
- PyTorch
- Saturn Cloud
- Uber
- USRA Labs
- Walmart Global Tech
- anyScale
- Dask
- GOAI
- Nuclio
- Numba
- scikit-learn
- XGBoost
Building on Top of RAPIDS
A Bigger, Better, Stronger Ecosystem for All

NVTabular
ETL library for recommender systems building off of RAPIDS and Dask

blazingSQL
GPU accelerated SQL engine built on top of RAPIDS

Streamz
Distributed stream processing using RAPIDS and Dask
NVTabular
ETL library for recommender systems building off of RAPIDS and Dask

A HIGH LEVEL API BUILDING UPON DASK-CUDF

NVTabular’s high level API allows users to think about what they want to do with the data, not how they have to do it or how to scale it, for operations common within recommendation workflows.

ACCELERATED GPU DATALOADERS

Using cuIO primitives and cuDF, NVTabular accelerates dataloading for PyTorch & Tensorflow, removing I/O issues common in deep learning based recommender system models.
BlazingSQL
GPU-accelerated SQL engine built with RAPIDS

BLAZING FAST SQL ON RAPIDS
• Incredibly fast distributed SQL engine on GPUs--natively compatible with RAPIDS!
• Allows data scientists to easily connect large-scale data lakes to GPU-accelerated analytics
• Directly query raw file formats such as CSV and Apache Parquet inside Data Lakes like HDFS and AWS S3, and directly pipe the results into GPU memory.

NOW WITH OUT-OF-CORE EXECUTION
• Users no longer limited by available GPU memory
• 10TB workloads on a single Tesla V100 (32GB)!

¬ GPU Big Data Benchmark (GPU-BDB¹) is a data science benchmark consisting of 30 end-to-end queries representing real-world ETL and Machine Learning workflows, involving both structured and unstructured data.
¬ Using BlazingSQL, we saw a 20% increase in performance - - on 6 fewer nodes!

1: GPU-BDB is derived from the TPCx-BB benchmark and is used for internal performance testing. Results from GPU-BDB are not comparable to TPCx-BB.
from blazingsql import BlazingContext
import cudf

bc = BlazingContext()

bc.s('bsql', bucket_name='bsql', access_key_id='<access_key>', secret_key='<secret_key>')</n
c.create_table('orders', s3://bsql/orders/')</n
gdf = bc.sql('select * from orders').get()
ACCELERATED KAFKA CONSUMER

Ingesting Kafka messages to cudf is increased by roughly 3.5 - 4X over standard cpu ingestion. Streaming TCO is lowered by using cheaper VM instance types.

CHECKPOINTING

Streaming job version of “where was I?” Streams can gracefully handle errors by continuing to process a stream at the exact point they were before the error occurred.

### Kafka throughput

<table>
<thead>
<tr>
<th></th>
<th>streamz</th>
<th>cuStreamz</th>
</tr>
</thead>
<tbody>
<tr>
<td>No Checkpointing</td>
<td>134</td>
<td>489</td>
</tr>
<tr>
<td>With Checkpointing</td>
<td>96</td>
<td>386</td>
</tr>
</tbody>
</table>

megabytes
Join the Conversation

GOOGLE GROUPS
https://groups.google.com/forum/#!forum/rapidsai

DOCKER HUB
https://hub.docker.com/r/rapidsai/rapidsai

SLACK CHANNEL
https://rapids-goai.slack.com/join

STACK OVERFLOW
https://stackoverflow.com/tags/rapids
Contribute Back
Issues, Feature Requests, PRs, Blogs, Tutorials, Videos, QA...Bring Your Best!
# RAPIDS Notices

Communicate and document changes to RAPIDS for contributors, core developers, users, and the community

<table>
<thead>
<tr>
<th>Notice</th>
<th>Title</th>
<th>Topic</th>
<th>RAPIDS Version</th>
<th>Updated</th>
</tr>
</thead>
<tbody>
<tr>
<td>RDN 1</td>
<td>‘dask-xgboost’ is deprecated in v0.15 &amp; removed v0.16</td>
<td>Library Deprecation</td>
<td>v0.15</td>
<td>26 August 2020</td>
</tr>
<tr>
<td>RGN 1</td>
<td>Stable/Release Branch Renaming to ‘main’ in v0.15</td>
<td>Git Repo Change</td>
<td>v0.15</td>
<td>26 August 2020</td>
</tr>
<tr>
<td>RGN 2</td>
<td>v0.15 No CUDA 11 Release for ‘ctx’</td>
<td>Release Change</td>
<td>v0.15</td>
<td>26 August 2020</td>
</tr>
<tr>
<td>RGN 4</td>
<td>v0.15 Release Delay for ‘cufftfilter’</td>
<td>Release Change</td>
<td>v0.15</td>
<td>26 August 2020</td>
</tr>
<tr>
<td>RSN 2</td>
<td>EOL Python 3.6 &amp; CUDA 10.0 in v0.14</td>
<td>Platform Support Change</td>
<td>v0.14 &amp; v0.15</td>
<td>13 July 2020</td>
</tr>
<tr>
<td>REN 3</td>
<td>Support for Python 3.8 in v0.15</td>
<td>Platform Support Change</td>
<td>v0.15</td>
<td>17 July 2020</td>
</tr>
<tr>
<td>REN 4</td>
<td>Support for CUDA 11.0 in v0.15</td>
<td>Platform Support Change</td>
<td>v0.15</td>
<td>26 August 2020</td>
</tr>
</tbody>
</table>

[https://docs.rapids.ai/notices](https://docs.rapids.ai/notices)
Getting Started
5 Steps to Getting Started with RAPIDS

1. Install RAPIDS on Docker, Conda, deploy in your favorite cloud instance, or quick start with app.blazingsql.com.

2. Explore our walk through videos, blog content, our github, the tutorial notebooks, and our example workflows.

3. Build your own data science workflows.

4. Join our community conversations on Slack, Google, and Twitter.

5. Contribute back. Don’t forget to ask and answer questions on Stack Overflow.
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.

NOTICES
- RAPIDS repos will rename stable/release branches in v0.15
- Python 3.6 & CUDA 10.0 EOL in v0.14
- Release changes to cdk and cuafilter in v0.15
- RAPIDS dask-xgboost library is deprecated in v0.15

METHOD

Preferred
- Conda 📘
- Docker + Examples 📖
- Docker + Dev Env 🐬
- Source 📁

Advanced

RELEASE

Legacy (0.14) 📘
Stable (0.15) 📖
Nightly (0.16a) 🐬

PACKAGES

All Packages 📘
cuDf 📖
cuML 🐬
cuGraph 📁
cuSignal 📖
cuSpatial 🐬
cufilter 📁

LINUX

Ubuntu 16.04 📘
Ubuntu 18.04 📖
CentOS 7 🐬
RHEL 7 📁

PYTHON

Python 3.6 (0.14 only) 📘
Python 3.7 📖
Python 3.8 (0.15/0.16 only) 🐬

CUDA

CUDA 10.0 (0.14 only) 📘
CUDA 10.1.2 📖
CUDA 10.2 🐬
CUDA 11.0 (0.15/0.16 only) 📁

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

COMMAND

conda install --rapidsai --nvidia --conda-forge
-- defaults rapids=0.15 python=3.7

https://rapids.ai/start.html
RAPIDS Docs

Up to Date and Easy to Use

https://docs.rapids.ai
RAPIDS Docs
Easier than Ever to Get Started with cuDF

https://docs.rapids.ai
Explore: RAPIDS Code and Blogs
Check out our Code and How We Use It

---

https://github.com/rapidsai

---

https://medium.com/rapids-ai

---

RAPIDS cuDF - GPU DataFrames

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_mf['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:

---

https://github.com/rapidsai

---

https://medium.com/rapids-ai
Explore: RAPIDS Github

https://github.com/rapidsai
Explore: RAPIDS Community Notebooks

Community supported notebooks have tutorials, examples, and various E2E demos. RAPIDS Youtube channel has explanations, code walkthroughs and use cases.

https://github.com/rapidsai-community/notebooks-contrib
RAPIDS
How Do I Get the Software?

GITHUB
https://github.com/rapidsai

ANACONDA
https://anaconda.org/rapidsai/

NGC

DOCKER
https://hub.docker.com/r/rapidsai/rapidsai/
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
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