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
Data Processing Evolution

Faster data access, less data movement

Hadoop Processing, Reading from disk

<table>
<thead>
<tr>
<th>HDFS Read</th>
<th>Query</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>
</table>

Spark In-Memory Processing

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

Traditional GPU Processing

<table>
<thead>
<tr>
<th>HDFS Read</th>
<th>GPU Read</th>
<th>Query</th>
<th>GPU Read</th>
<th>ETL</th>
<th>GPU Read</th>
<th>ML Train</th>
</tr>
</thead>
</table>

- **25-100x Improvement**
  - 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 A

Copy & Convert

GPU

APP B

Data

Read Data

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 (eg, 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
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**
- DGX-2 RAPIDS v0.3: 42 seconds
- DGX-2 RAPIDS v0.10: 37 seconds
- 5x DGX-1 RAPIDS v0.2: 19 seconds
- 5x DGX-1 RAPIDS v0.10: 17 seconds

**XGBoost Machine Learning**
- DGX-2 RAPIDS v0.3: 169 seconds
- DGX-2 RAPIDS v0.10: 147 seconds
- 5x DGX-1 RAPIDS v0.2: 157 seconds
- 5x DGX-1 RAPIDS v0.10: 137 seconds

**End-to-End**
- DGX-2 RAPIDS v0.3: 322 seconds
- DGX-2 RAPIDS v0.10: 209 seconds
- 5x DGX-1 RAPIDS v0.2: 213 seconds
- 5x DGX-1 RAPIDS v0.10: 164 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
Speed, Ease of Use, and Iteration

The Way to Win at Data Science
RAPIDS Core
Open Source Data Science Ecosystem

Familiar Python APIs

Data Preparation → Model Training → Visualization

Pandas
Scikit-Learn
NetworkX
PyTorch
Chainer
MxNet

Dash

Familiar Python APIs

Matplotlib/Plotly

CPU Memory
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

Dask

GPU Memory

Apache Arrow
Dask
RAPIDS
Scaling RAPIDS with Dask
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**
cuDF
GPU Accelerated data wrangling and feature engineering

Data Preparation → Model Training → Visualization

cuDF cuIO
Analytics

cuML
Machine Learning

cuGraph
Graph Analytics

PyTorch Chainer
MxNet
Deep Learning

cuxfilter <-> pyViz
Visualization

Dask

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
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 v0.13 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

```python
def main():
    import pandas, cudf
    %time len(pandas.read_csv('data/nyc/yellow_tripdata_2015-01.csv'))
    CPU times: user 25.9 s, sys: 3.26 s, total: 29.2 s
    Wall time: 29.2 s

    %time len(cudf.read_csv('data/nyc/yellow_tripdata_2015-01.csv'))
    CPU times: user 1.59 s, sys: 372 ms, total: 1.96 s
    Wall time: 2.12 s
```

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__
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>

https://blog.dask.org/2019/01/03/dask-array-gpus-first-steps

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

<table>
<thead>
<tr>
<th>Array size</th>
<th>Wall Time (data creation + compute)</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.2 PB (20M x 20M doubles)</td>
<td>54 min 51 s</td>
</tr>
</tbody>
</table>

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
cuML
Machine Learning
More models more problems
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.

Meet reasonable speed vs accuracy tradeoff

Time Increases

Hours? Days?

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

More to come!

Key:
- Preexisting
- NEW or enhanced for 0.13
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'feat{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

**Forest Inference**

Taking models from training to production

**Graph: XGBoost CPU Inference vs. FIL GPU (1K trees, 1M rows)**

- **CPU Time (XGBoost, 40 cores)** vs. **FIL GPU Time (1x V100)**
- **Inference time (ms)**
- **bosch**: 23x
- **airline**: 36x
- **higgs**: 34x
- **epsilon**: 23x

Using FIL_BATCH_REGRE algorithm
• 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
Algorithms

GPU-accelerated NetworkX

- Community
- Components
- Link Analysis
- Link Prediction
- Traversal
- Centrality

Graph Classes
- Structure
- Multi-GPU
- Utilities

Renumbering
Auto-renumbering

More to come!

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
**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>
</tbody>
</table>

**BigData x8** 400,000,000 16,000,000,000 300 800* 5760*

*BigData x8, 100x 8-vCPU nodes, Apache Spark GraphX ⇒ 96 mins!
Community
Ecosystem Partners

CONTRIBUTORS

ADOPTERS

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

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

GPU accelerated SQL engine built on top of RAPIDS

Distributed stream processing using RAPIDS and Dask
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
- cuFilter

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 \
-c defaults rapid=0.13 python=3.6
```

COPY COMMAND

DETAILS BELOW
Contribute Back

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

Getting Started with cuDF (RAPIDS)
Getting Started
RAPIDS Docs
Improved and easier to use!

https://docs.rapids.ai
RAPIDS
How do I get the software?

- [https://github.com/rapidsai](https://github.com/rapidsai)
- [https://anaconda.org/rapidsai](https://anaconda.org/rapidsai/)
- [https://hub.docker.com/r/rapidsai/rapidsai](https://hub.docker.com/r/rapidsai/rapidsai)
THANK YOU

Nick Becker
nicholas@nvidia.com