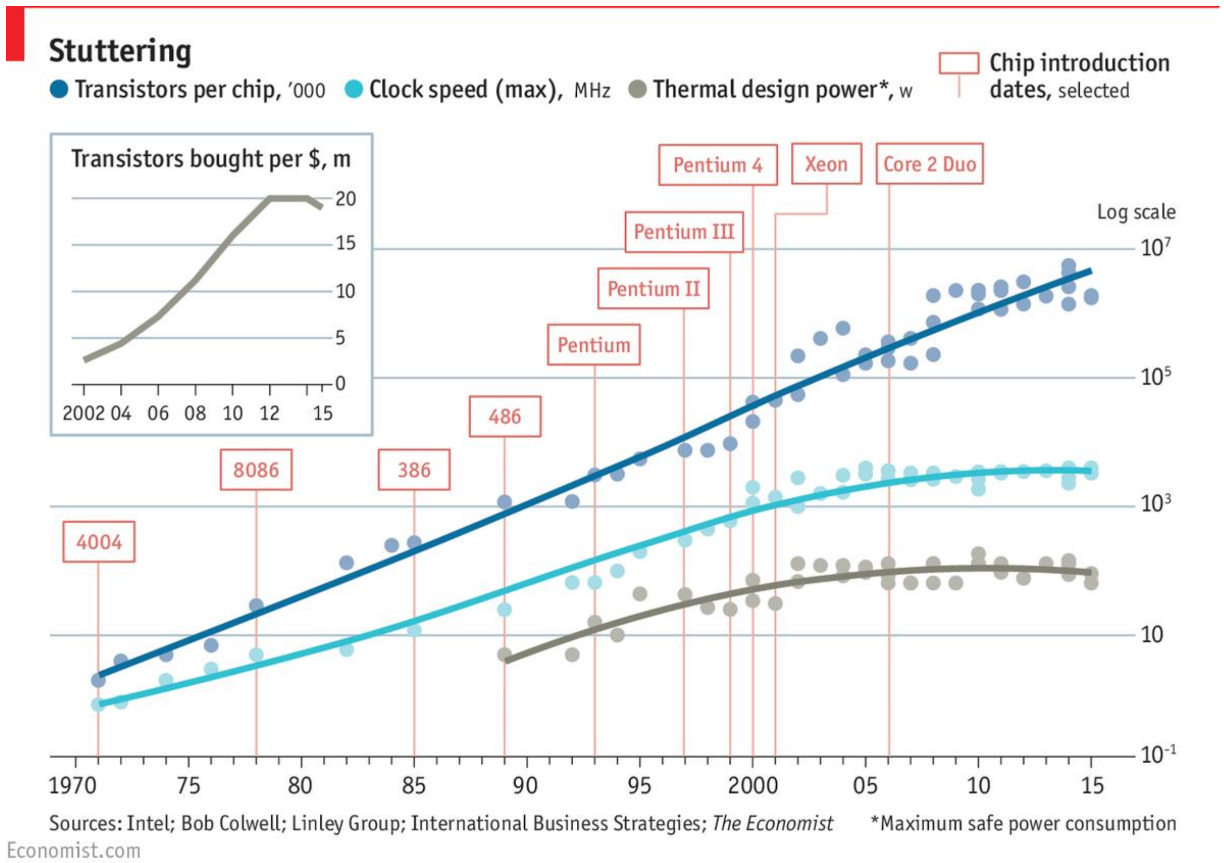
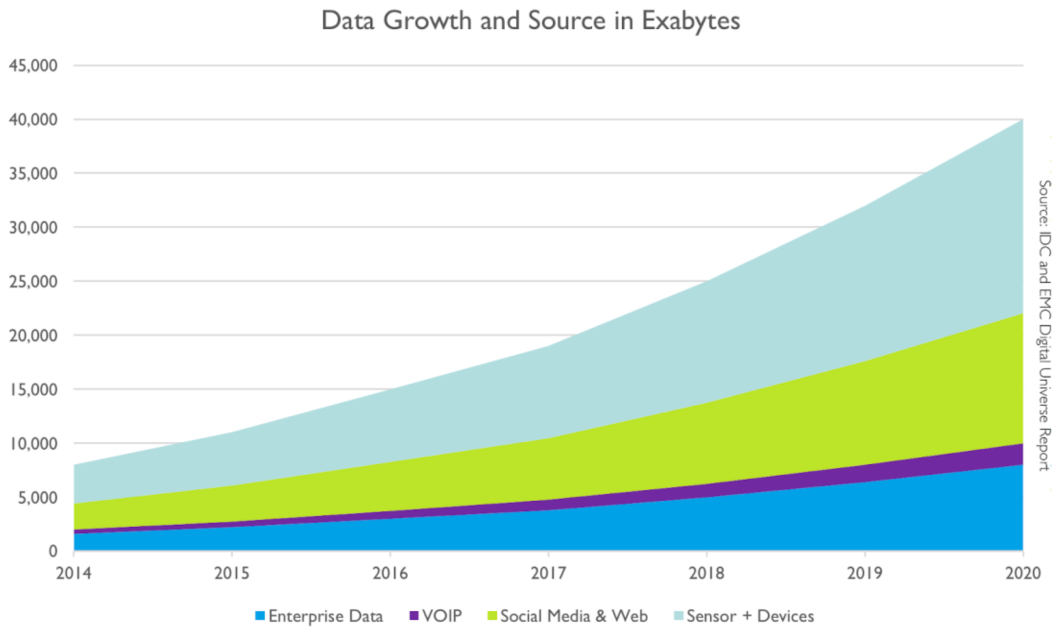
RAPIDS

GPU Accelerated Data Analytics in Python

Mads R. B. Kristensen, NVIDIA


Stuttering
Transistors per chip, '000
Clock speed (max), MHz
Thermal design power*, w
Chip introduction dates, selected
Transistors bought per $, m
20
15
10
5
0
2002 04 06 08 10 12 15
4004
8086
386
486
Pentium
Pentium II
Pentium III
Pentium 4
Xeon
Core 2 Duo
Log scale
10^7
10^5
10^3
10
10^{-1}
1970 75 80 85 90 95 2000 05 10 15
Sources: Intel; Bob Colwell; Linley Group; International Business Strategies; The Economist
*Maximum safe power consumption
Economist.com


Data Growth and Source in Exabytes
45,000
40,000
35,000
30,000
25,000
20,000
15,000
10,000
5,000
0
2014
2015
2016
2017
2018
2019
2020
Enterprise Data
VOIP
Social Media & Web
Sensor + Devices
Source: IDC and EMC Digital Universe Report

Scale up and out with RAPIDS and Dask

RAPIDS and Others

Accelerated on single GPU

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

Dask + RAPIDS

Multi-GPU
On single Node (DGX)
Or across a cluster

PyData

NumPy, Pandas, Scikit-Learn
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 Up / Accelerate

Scale out / Parallelize

Scale up and out with RAPIDS and Dask

Scale Up / Accelerate

RAPIDS and Others

Accelerated on single GPU

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

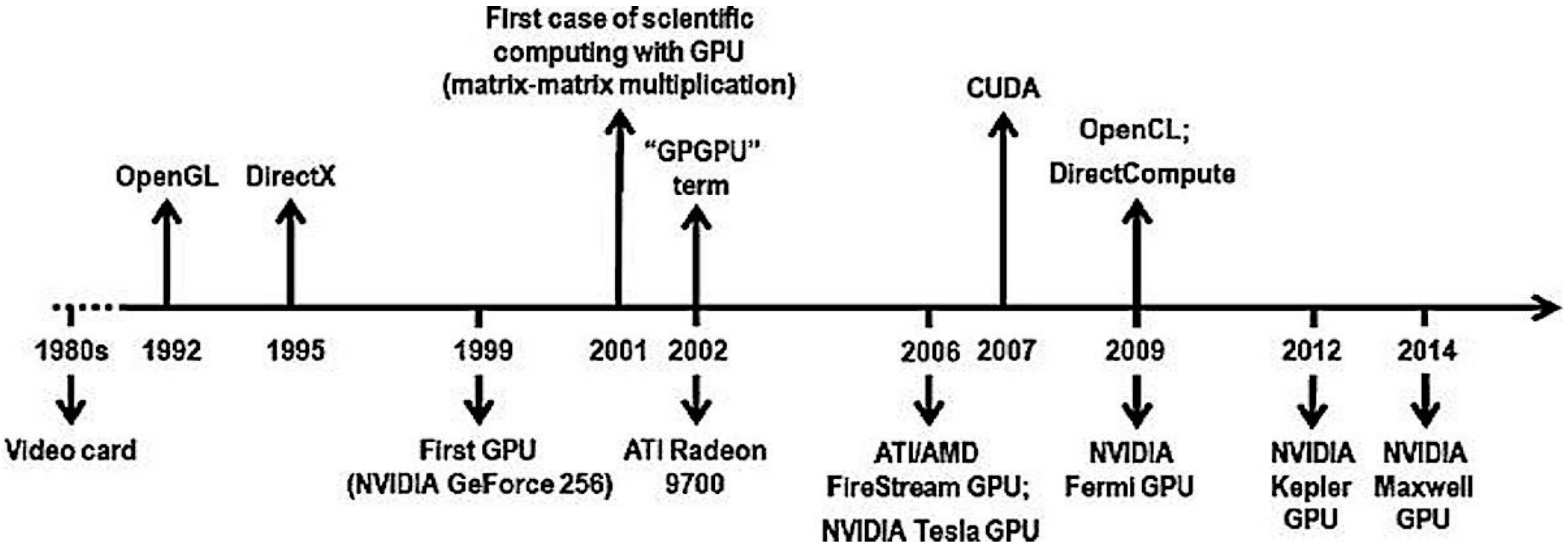
PyData

NumPy, Pandas, Scikit-Learn
and many more

Single CPU core
In-memory data

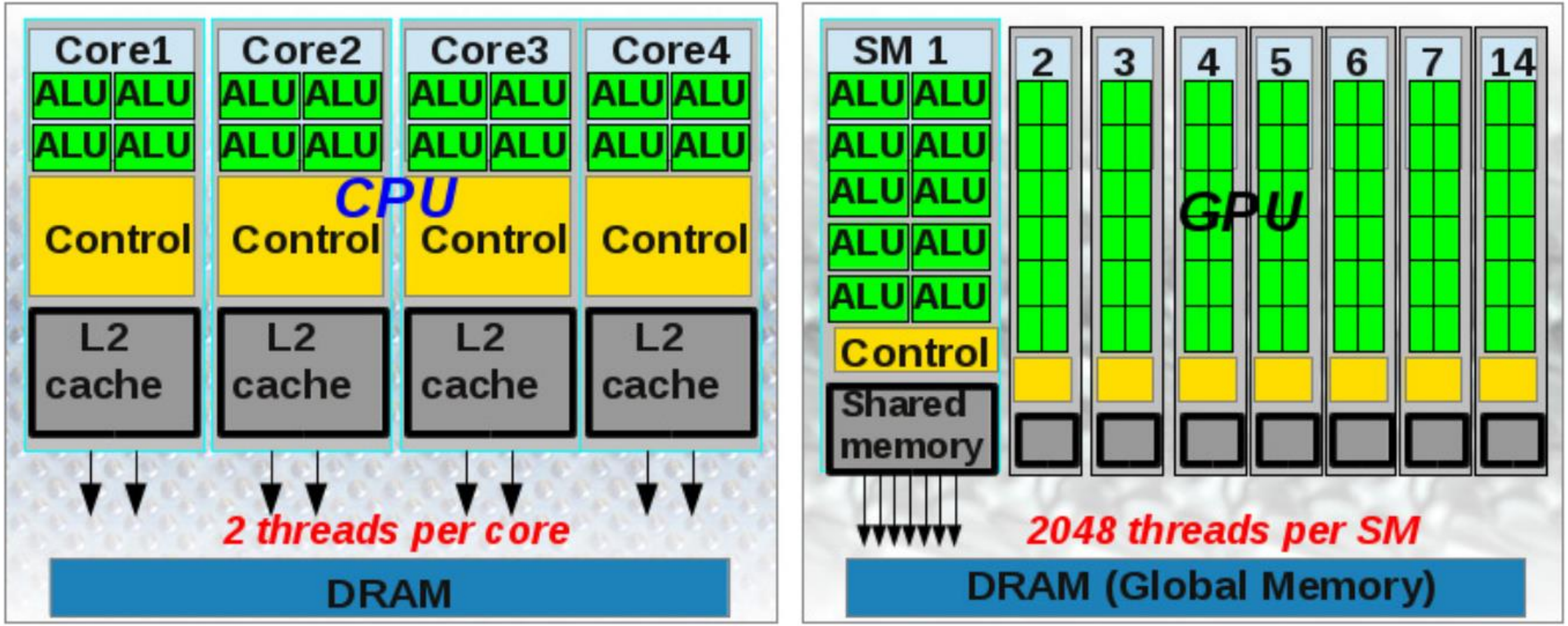
Scale out / Parallelize

History of the GPU



DOI: https://doi.org/10.1007/978-3-319-17885-1_1606

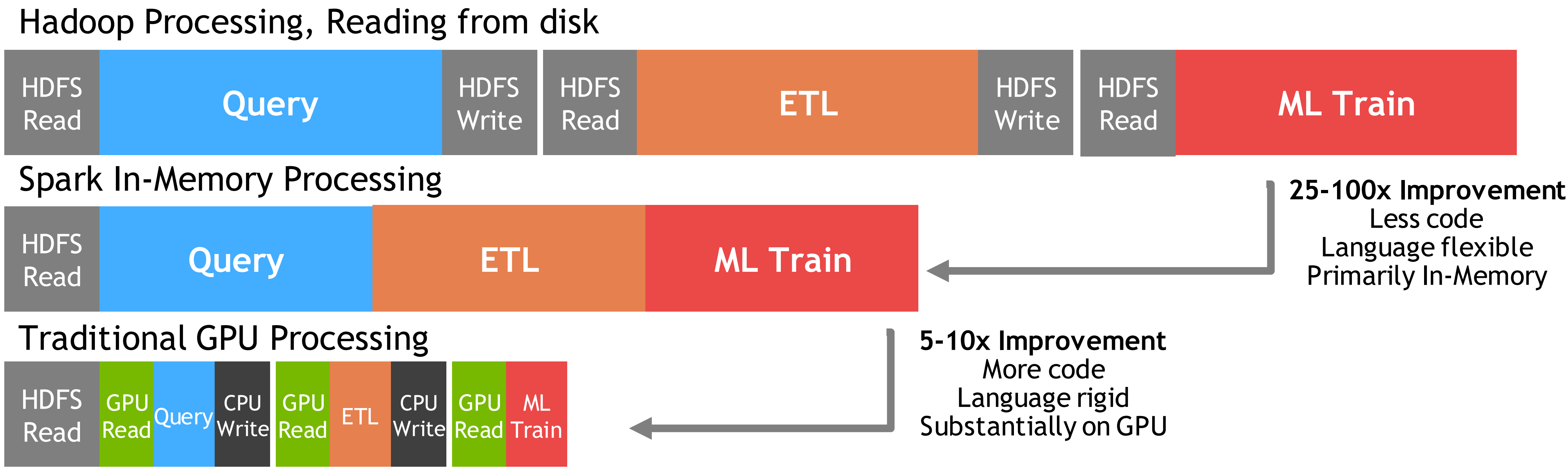
CPU vs GPU



DOI: 10.1016/j.cam.2013.12.032.

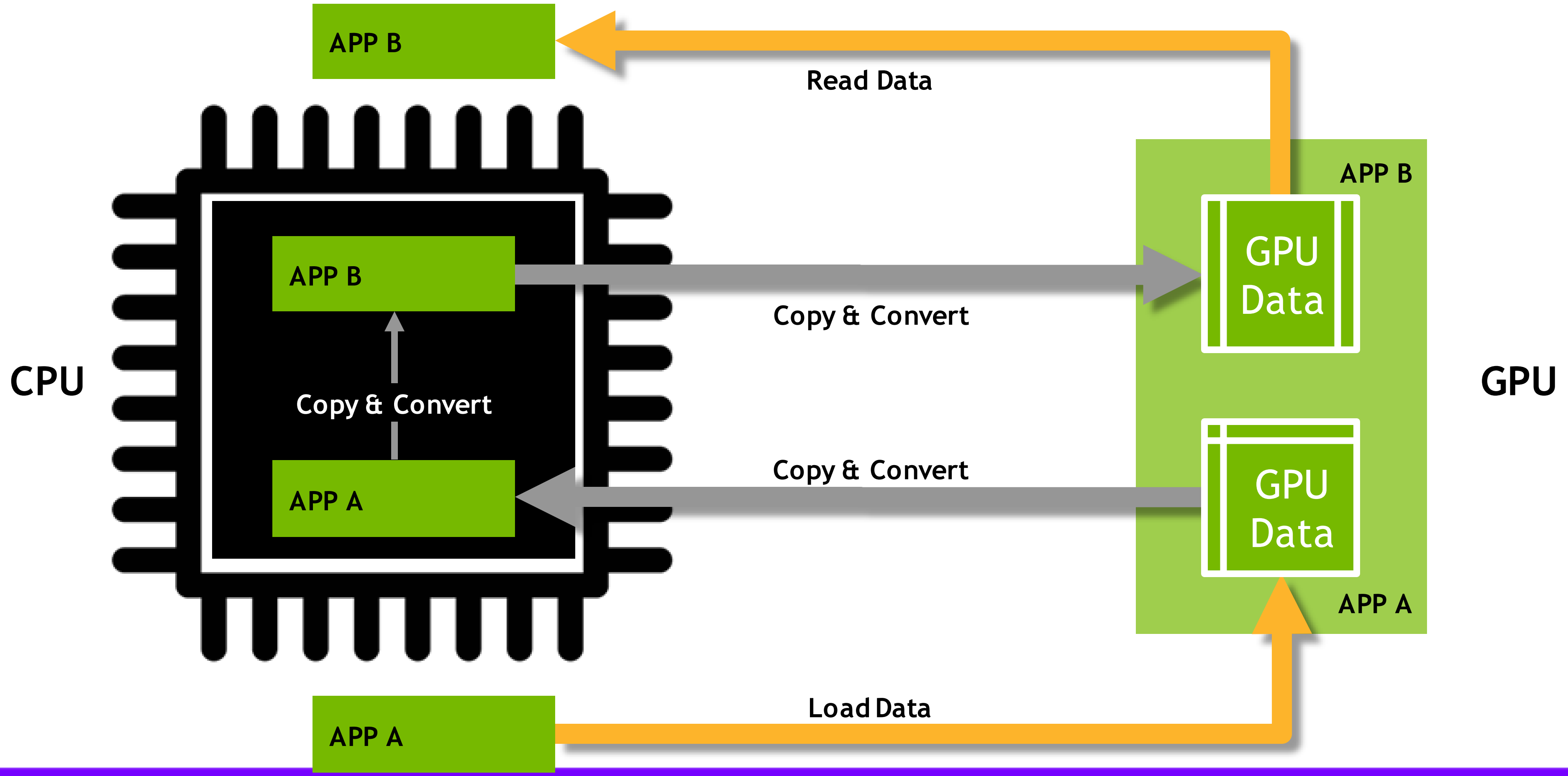
Data Processing Evolution

Faster data access, less data movement



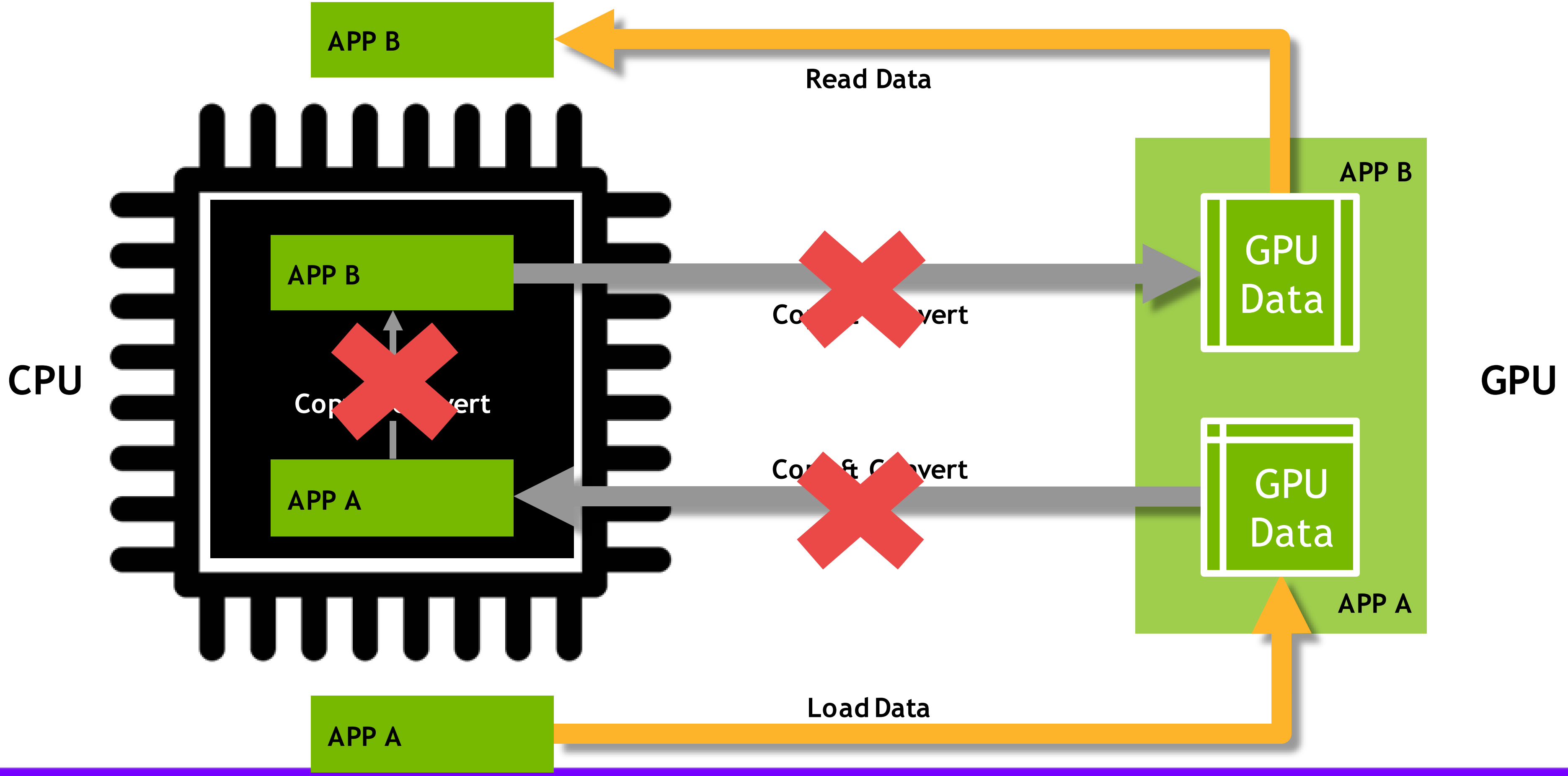
Data Movement and Transformation

What if we could keep data on the GPU?



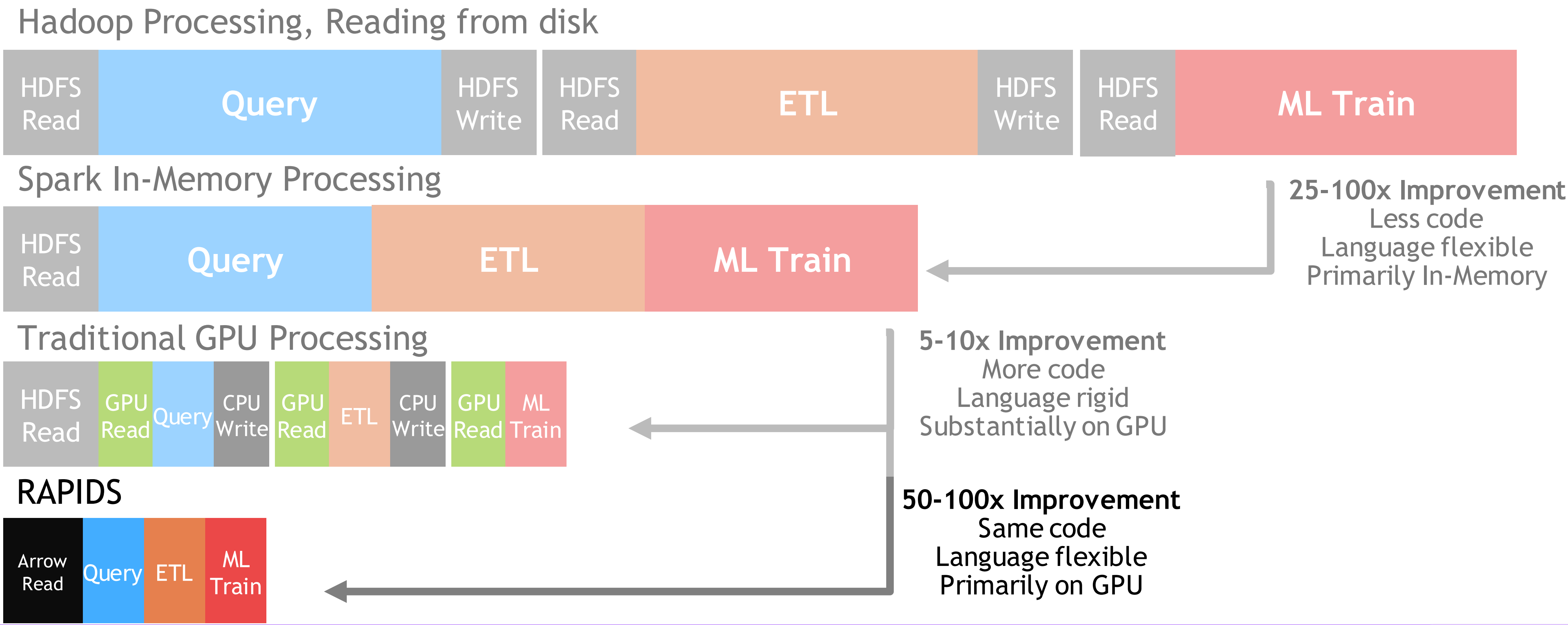
Data Movement and Transformation

What if we could keep data on the GPU?



Data Processing Evolution

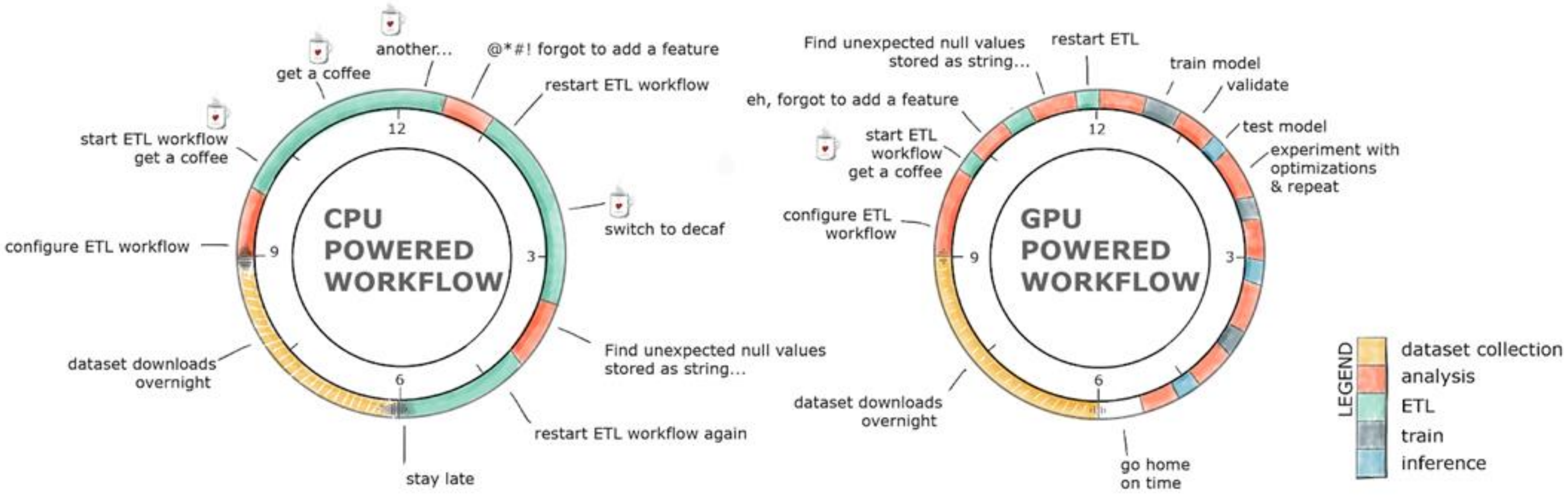
Faster data access, less data movement



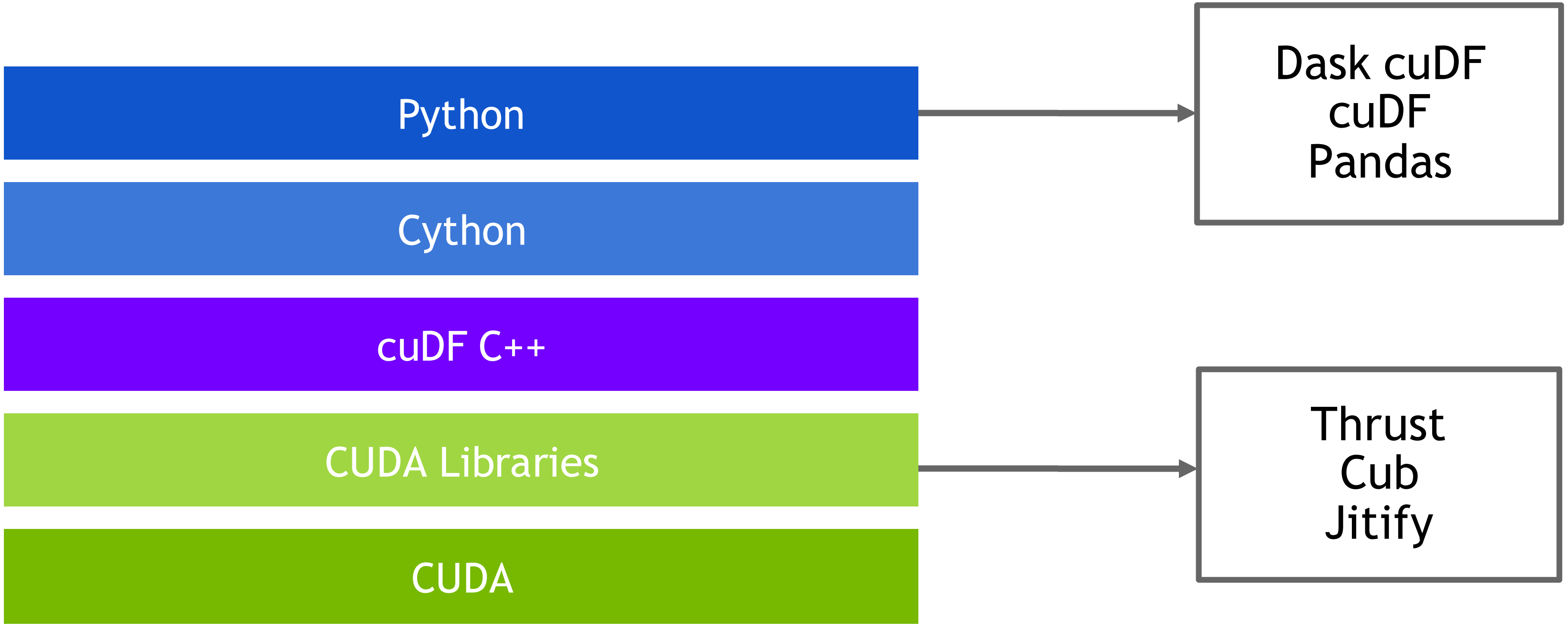
CuDF – Pandas on GPU

GPU-Accelerated ETL

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



ETL Technology Stack



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

```
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…

In [2]:
```
#Read in the data. Notice how it decompresses as it reads the data into memory.
gdf = cudf.read_csv('/rapids/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]:

	User_ID	Product_ID	Gender	Age	Occupation	City_Category	Stay_In_Current_City_Years	Marital_Status	Product_Cat
0	1000001	P00069042	F	0-17	10	A	2	0	3
1	1000001	P00248942	F	0-17	10	A	2	0	1
2	1000001	P00087842	F	0-17	10	A	2	0	12
3	1000001	P00085442	F	0-17	10	A	2	0	12
4	1000002	P00285442	M	55+	16	C	4+	0	8

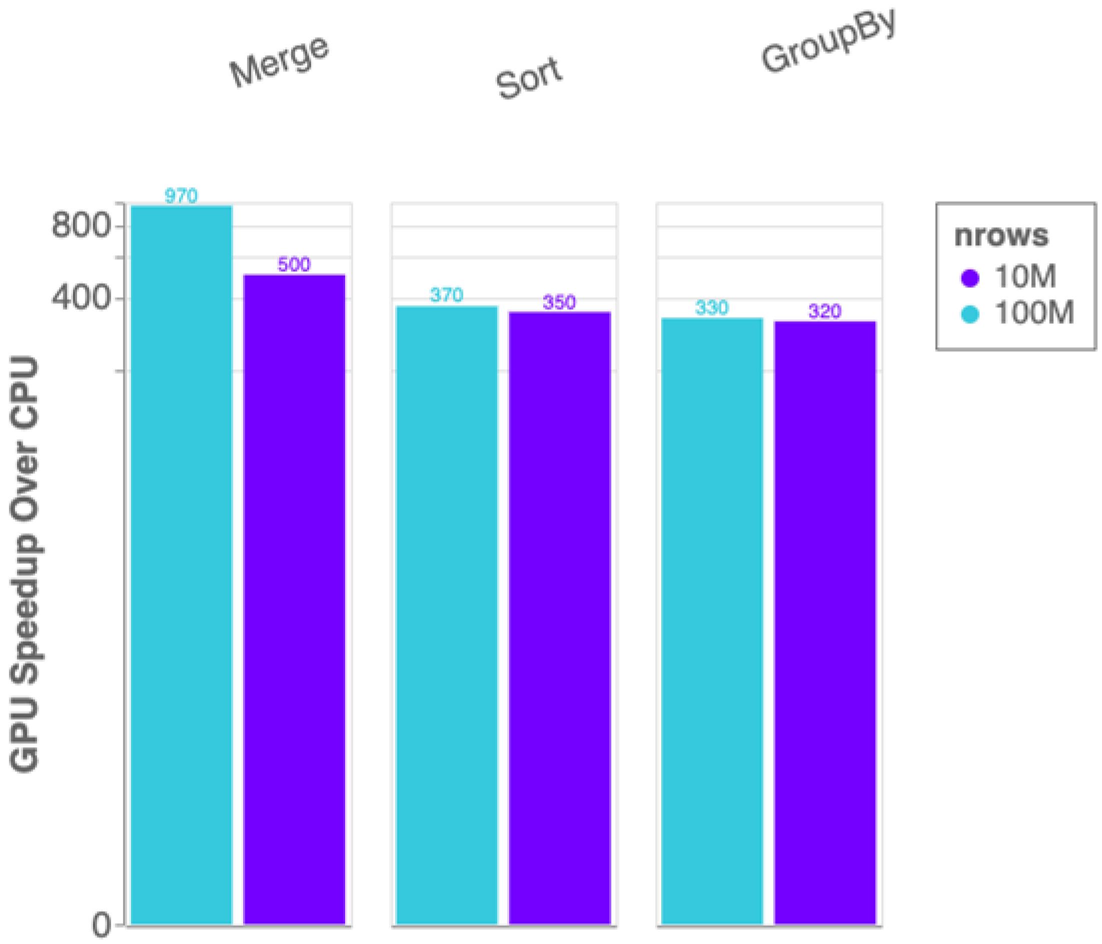
In [6]:
```
#grabbing the first character of the years in city string to get rid of plus sign, and converting
 to int
gdf['city_years'] = gdf.Stay_In_Current_City_Years.str.get(0).stoi()
```

In [7]:
```
#Here we can see how we can control what the value of our dummies with the replace method and turn
strings to ints
gdf['City_Category'] = gdf.City_Category.str.replace('A', '1')
gdf['City_Category'] = gdf.City_Category.str.replace('B', '2')
gdf['City_Category'] = gdf.City_Category.str.replace('C', '3')
gdf['City_Category'] = gdf['City_Category'].str.stoi()
```

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

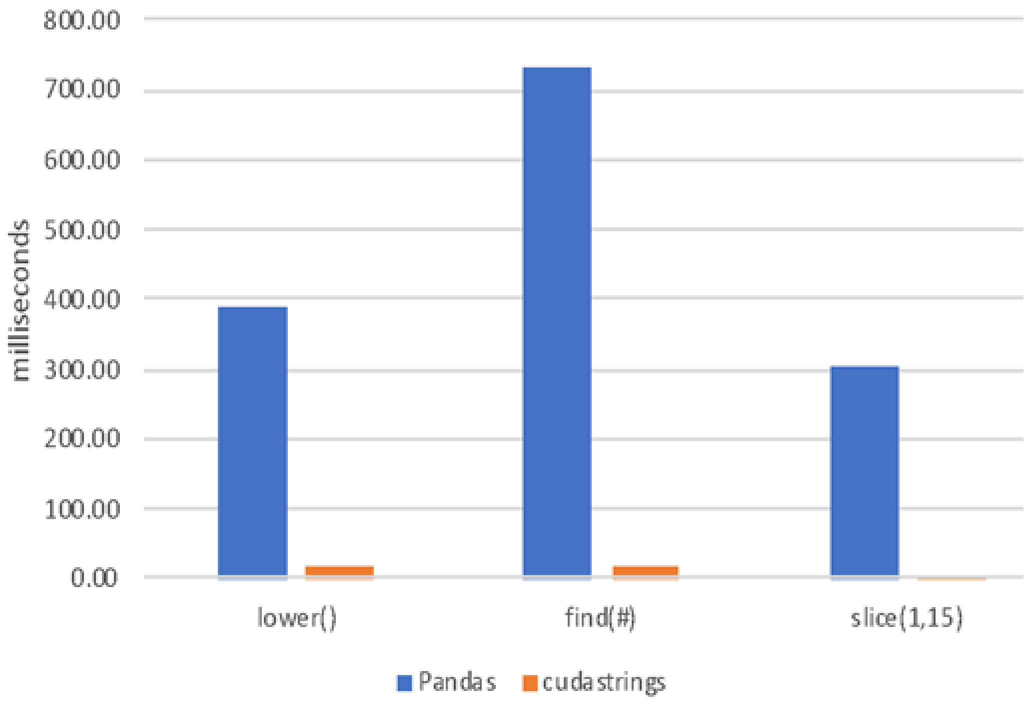
Merge: inner

GroupBy: count, sum, min, max calculated for each value column

ETL: the Backbone of Data Science

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++



- 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
- Parquet Reader
- ORC Reader
- JSON Reader
- Avro Reader

- GPU Direct Storage (GDS) integration for bypassing PCIe bottlenecks!

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

```
1]: import pandas, cudf

2]: %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
2]: 12748986

3]: %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
3]: 12748986

4]: !du -hs data/nyc/yellow_tripdata_2015-01.csv
    1.9G    data/nyc/yellow_tripdata_2015-01.csv
```

Source: *Apache Crail blog: [SQL Performance: Part 1 - Input File Formats]*

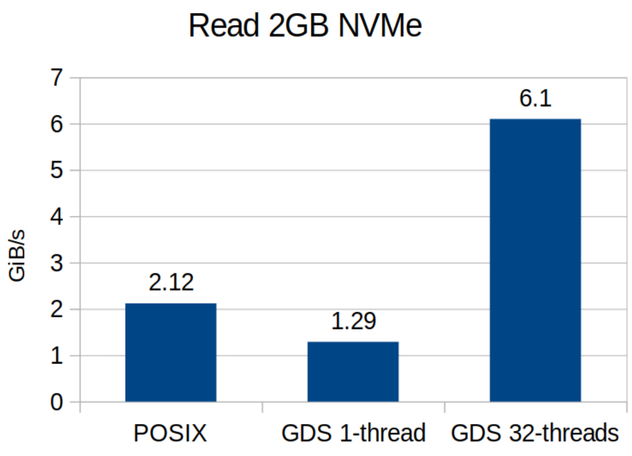
RAPIDS KvikIO

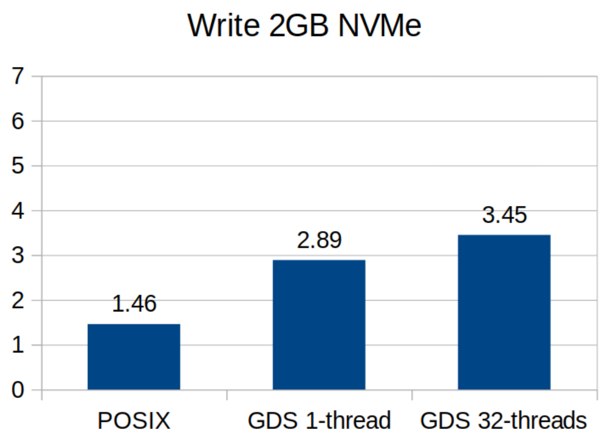
KvikIO is a C++ and Python frontend for cuFile that provide features such as an object-oriented API, exception handling, RAII semantic, multithreading IO, fallback mode, and a Zarr backend.

Using KvikIO should feel natural to C++ and Python developers.

```cpp
#include <cuda_runtime.h>
#include <kvikio/file_handle.hpp>
using namespace std;

int main() {
  void *a = nullptr;
  cudaMalloc(&a, 80);
  // Read file into `a` in parallel using 16 threads
  kvikio::default_thread_pool::reset(16);
  {
    kvikio::FileHandle f("/nvme/input.raw", "r");
    future<size_t> fut = f.pread(a, sizeof(a), 0);
    size_t read = fut.get(); // Blocking
    // Note, `f` closes automatically on destruction.
  }
}
```

Comparing KvikIO's Zarr backend versus manually copying between GPU and host memory before accessing the Zarr array using POSIX





NVIDIA DGX A100 (using one of the GPUs)
2x AMD EPYC 7742 64-Core@3.4GHz (max boost)
1x NVMe Samsung PM1733 SSD (MZWLJ3T8HBLS-00007)

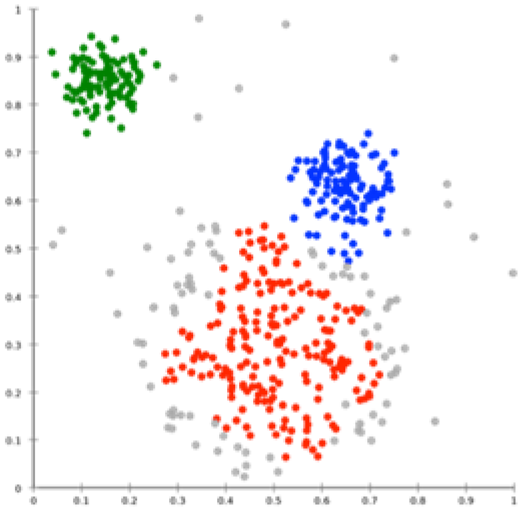
KvikIO: https://github.com/rapidsai/kvikio

```python
# Write CuPy array to disk
import cupy
import kvikio
a = cupy.arange(10)
with kvikio.CuFile("/nvme/input.raw", "w") as f:
    f.write(a)

# Write same CuPy array to a Zarr store
import zarr
from kvikio.zarr import GDSStore
z = zarr.array(a,
    compressor=None,
    store=GDSStore("/nvme/store"),
    meta_array=cupy.empty(()),
)
# We can not access the Zarr array `z` as a
# regular CuPy array.
b = z[:]  # Read from disk to GPU seamlessly
```

CuML – Scikit-Learn on GPU

Algorithms

GPU-accelerated Scikit-Learn



Cross Validation

Hyper-parameter Tuning

More to come!

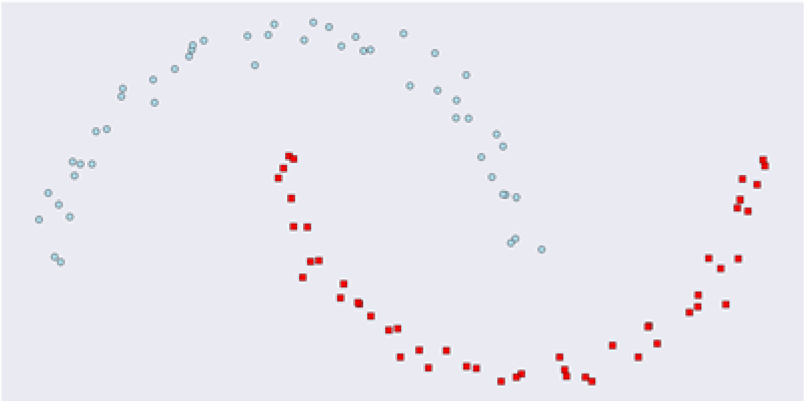
Classification / Regression	Decision Trees / **Random Forests** **Linear Regression** Logistic Regression K-Nearest Neighbors Support Vector Machines
Inference	Random forest / GBDT inference
Clustering	K-Means DBSCAN Spectral Clustering
Decomposition & Dimensionality Reduction	Principal Components Singular Value Decomposition UMAP Spectral Embedding T-SNE
Time Series	Holt-Winters **Seasonal ARIMA**

RAPIDS matches common Python APIs

CPU-Based Clustering

```
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)
```

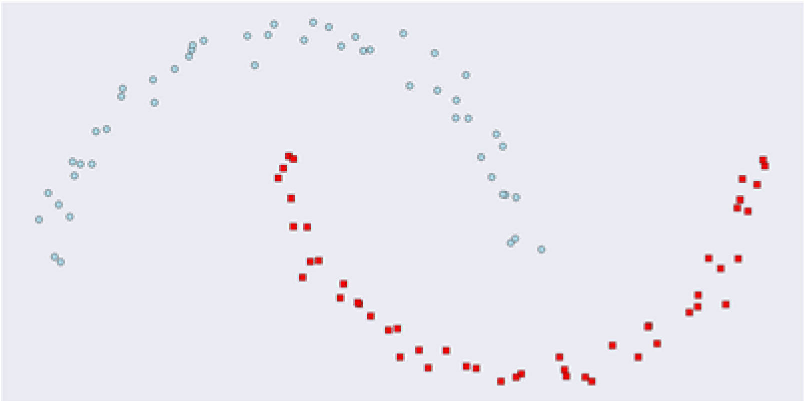


RAPIDS matches common Python APIs

GPU-Accelerated Clustering

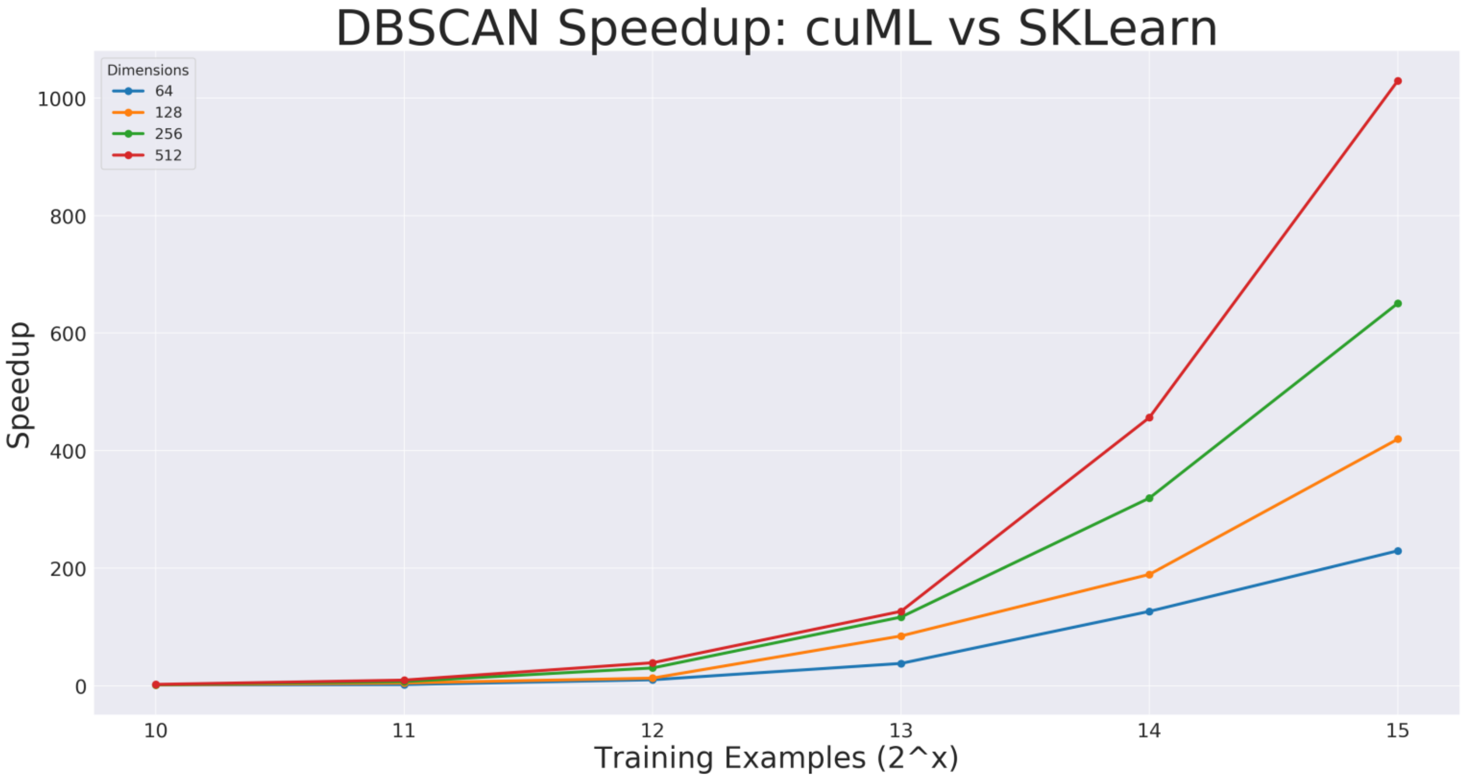
```
from sklearn.datasets import make_moons
import cudf

X, y = make_moons(n_samples=int(1e2),
                  noise=0.05, random_state=0)

X =    cudf.DataFrame({'fea%d'%i: X[:, i]
                     for i in range(X.shape[1])})
```

```
from cuml                import DBSCAN
dbscan = DBSCAN(eps = 0.3, min_samples = 5)

dbscan.fit(X)

y_hat = dbscan.predict(X)
```

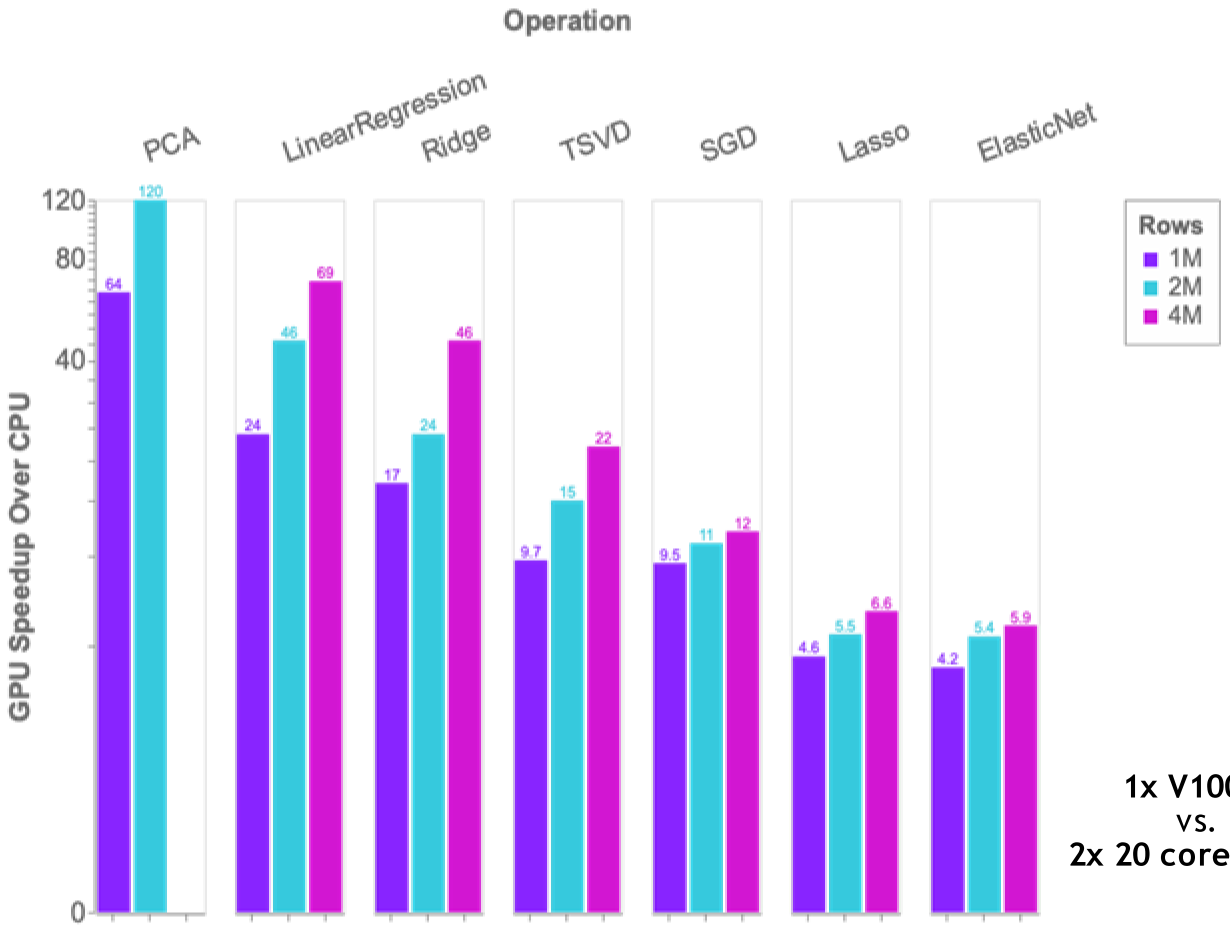


CLUSTERING

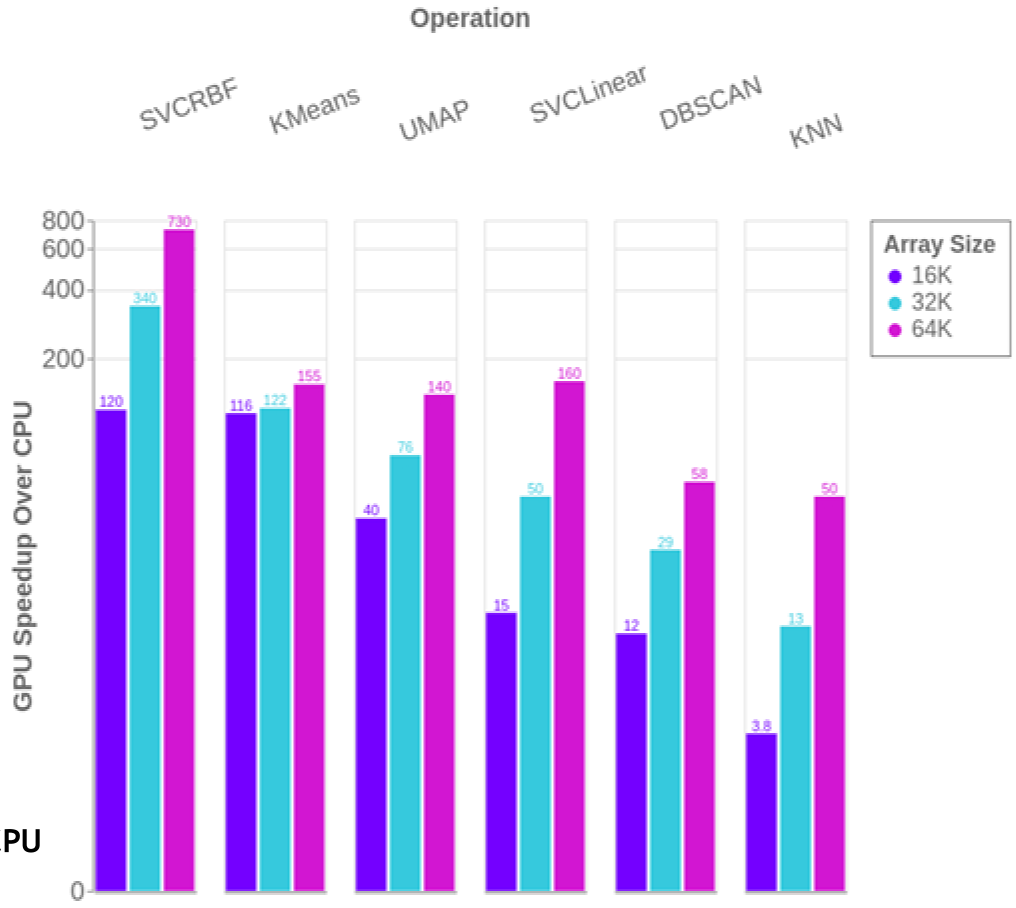
Benchmark



Benchmarks: single-GPU cuML vs scikit-learn



1x V100
vs.
2x 20 core CPU



Scale up and out with RAPIDS and Dask

RAPIDS and Others

Accelerated on single GPU

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

Dask + RAPIDS

Multi-GPU
On single Node (DGX)
Or across a cluster

PyData

NumPy, Pandas, Scikit-Learn 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 Up / Accelerate

Scale out / Parallelize

Dask
Distributing Python Libraries

Scale up and out with RAPIDS and Dask

Scale Up / Accelerate

PyData

NumPy, Pandas, Scikit-Learn
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

Dask Parallelizes

Natively

- **Support existing data science libraries**
  - Built on top of NumPy, Pandas, Scikit-Learn, … (easy to migrate)
  - With the same APIs (easy to train)
- **Scales**
  - Scales out to thousand-node clusters
  - Easy to install and use on a laptop
- **Popular**
  - Most common parallelism framework today at PyData and SciPy conferences
- **Deployable**
  - HPC: SLURM, PBS, LSF, SGE
  - Cloud: Kubernetes
  - Hadoop/Spark: Yarn

Parallel NumPy

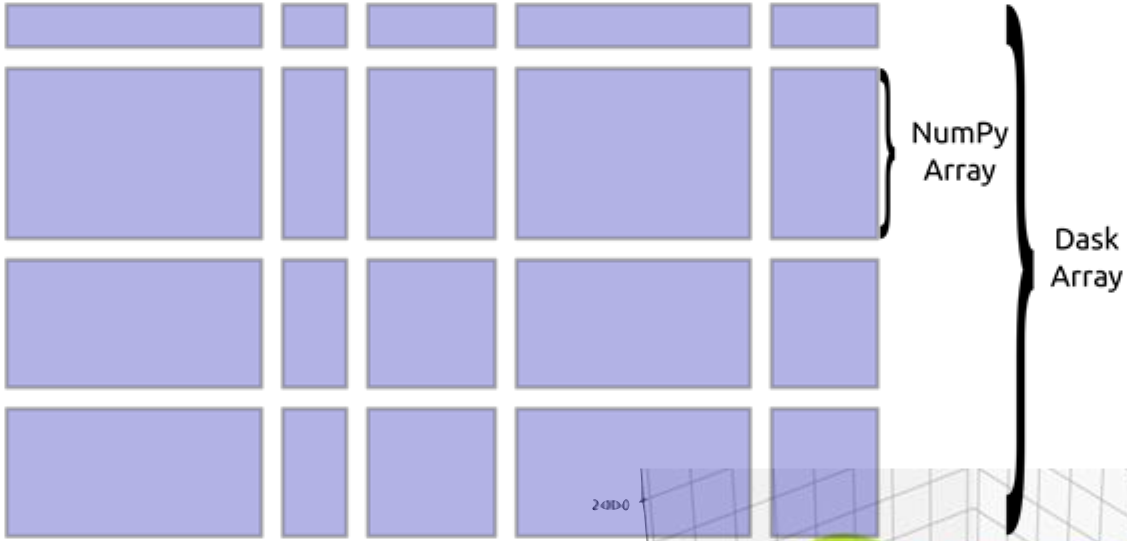
For imaging, simulation analysis, machine learning

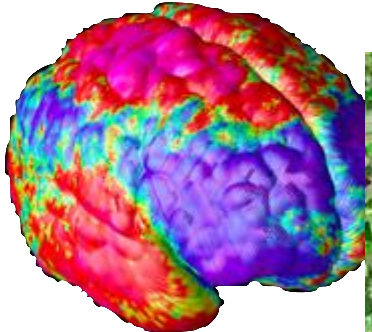
- Same API as NumPy

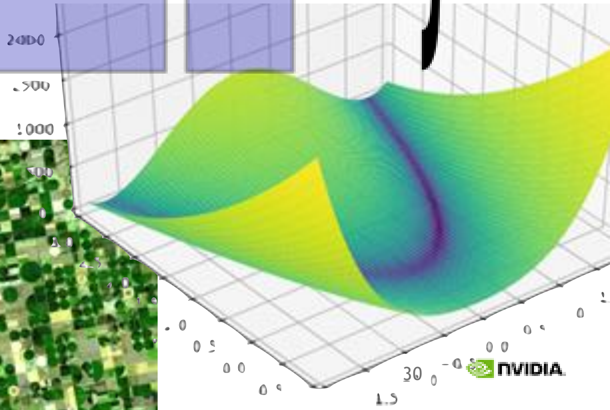
```
import dask.array as da
x = da.from_hdf5(...)
x + x.T - x.mean(axis=0)
```

- One Dask Array is built from many NumPy arrays

  Either lazily fetched from disk
  Or distributed throughout a cluster







Parallel Pandas

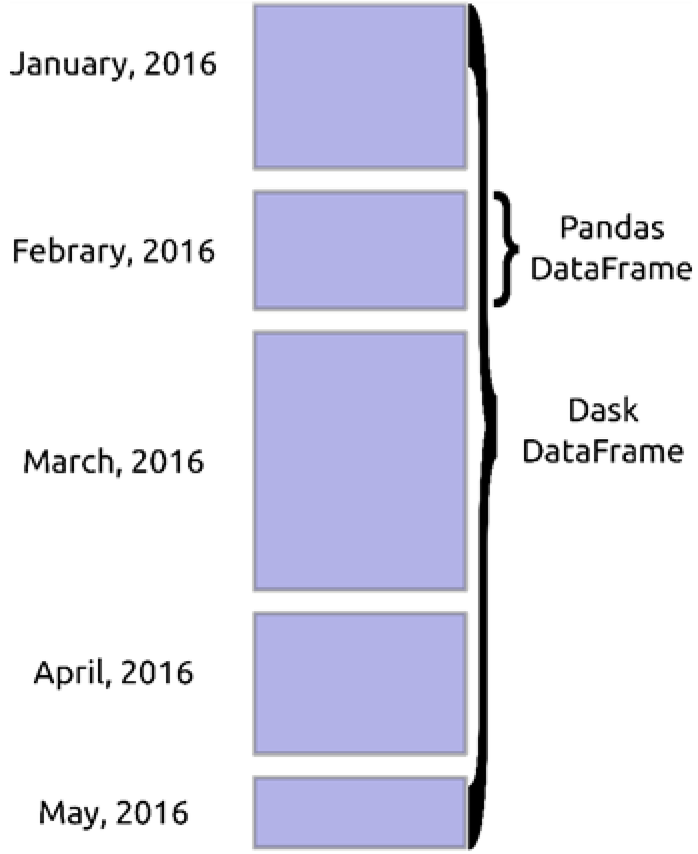
For ETL, time series, data munging

- Same API as Pandas

```
import dask.dataframe as dd
df = dd.read_csv(...)
df.groupby('name').balance.max()
```

- One Dask DataFrame is built from many Pandas DataFrames

  Either lazily fetched from disk
  Or distributed throughout a cluster

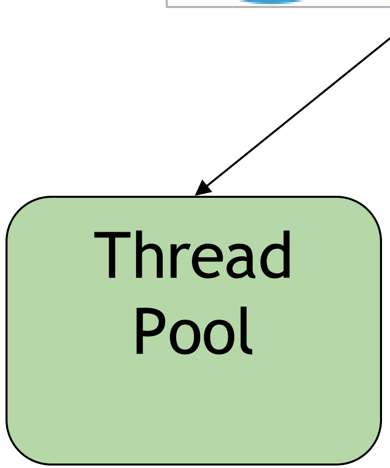


Parallel Scikit-Learn

For Hyper-Parameter Optimization, Random Forests, ...

- Same API

```
estimator = RandomForest()
estimator.fit(data, labels)
```



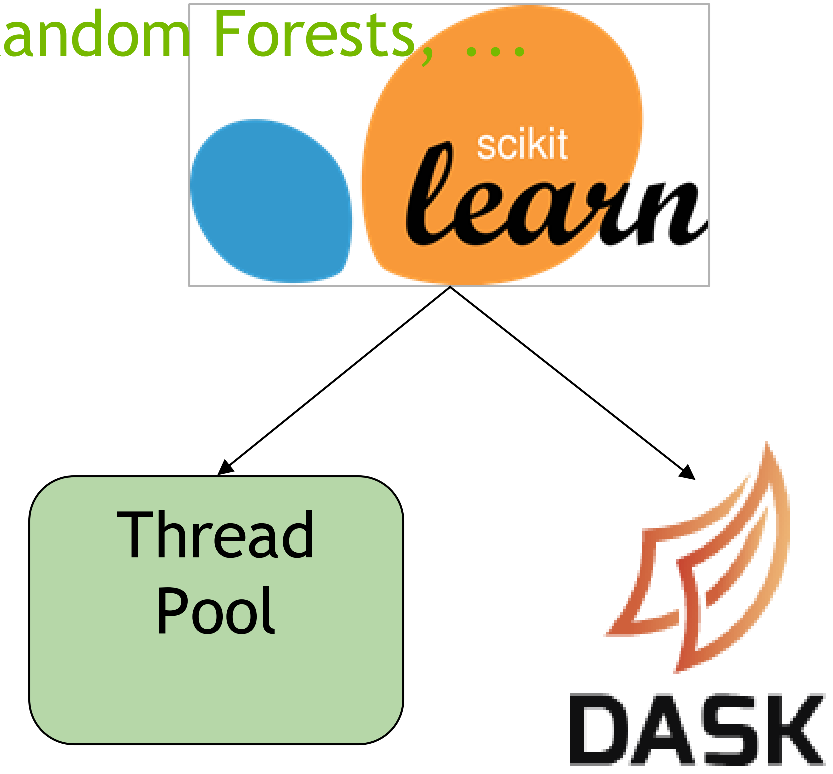
Parallel Scikit-Learn

For Hyper-Parameter Optimization, Random Forests, ...

- Same API

```
from scikit_learn.externals import joblib
with joblib.parallel_backend('dask'):
    estimator = RandomForest()
    estimator.fit(data, labels)
```

- Same exact code, just wrap in a "with" block
- Replaces default threaded execution with Dask
  Allowing scaling onto clusters
- Available in most Scikit-Learn algorithms where joblib is used



Parallel Python

For custom systems, ML algorithms, workflow engines

- Parallelize existing codebases

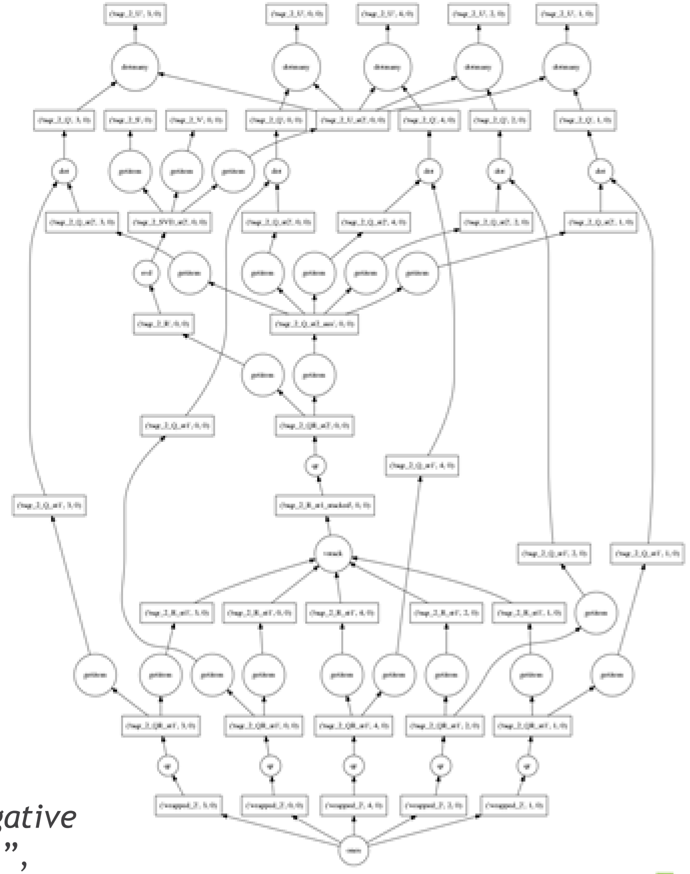
```
results = {}

for x in X:
  for y in Y:
    if x < y:
      result = f(x, y)
    else:
      result = g(x, y)
    results.append(result)
```

Parallel Python

For custom systems, ML algorithms, workflow engines

- Parallelize existing codebases

```
f = dask.delayed(f)
g = dask.delayed(g)

results = {}

for x in X:
  for y in Y:
    if x < y:
      result = f(x, y)
    else:
      result = g(x, y)
    results.append(result)

result = dask.compute(results)
```



M Tepper, G Sapiro "*Compressed nonnegative matrix factorization is fast and accurate*", IEEE Transactions on Signal Processing, 2016

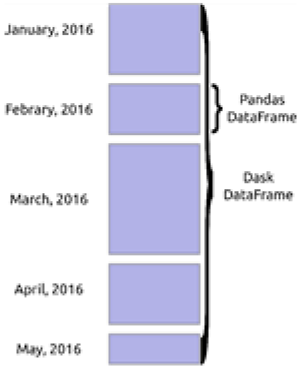
Dask Connects Python users to Hardware

User

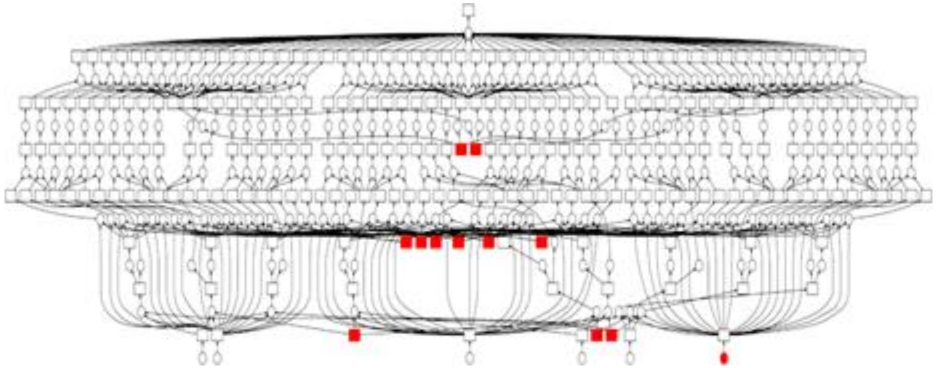
Execute on distributed hardware

Dask Connects Python users to Hardware

User



Writes high level code
(NumPy/Pandas/Scikit-Learn)



Turns into a task graph

Execute on distributed hardware

Scale up and out with RAPIDS and Dask

Scale Up / Accelerate

RAPIDS and Others

Accelerated on single GPU

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

RAPIDS

Dask + RAPIDS

Multi-GPU
On single Node (DGX)
Or across a cluster

RAPIDS
DASK

PyData

NumPy, Pandas, Scikit-Learn
and many more

Single CPU core
In-memory data

PyData
NumPy
scikit learn
pandas

Dask

Multi-core and Distributed PyData

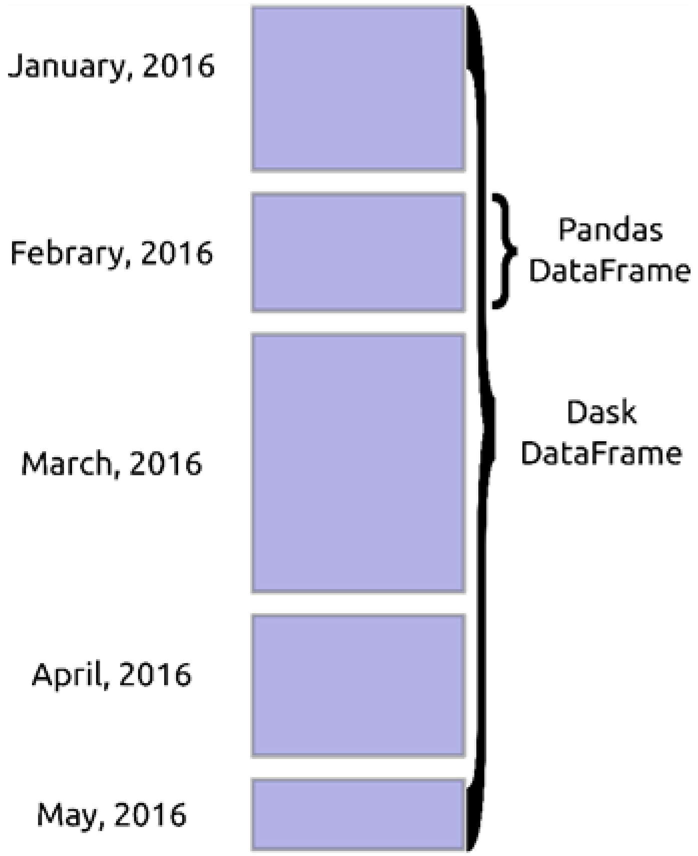
NumPy -> Dask Array
Pandas -> Dask DataFrame
Scikit-Learn -> Dask-ML
… -> Dask Futures

DASK

Scale out / Parallelize

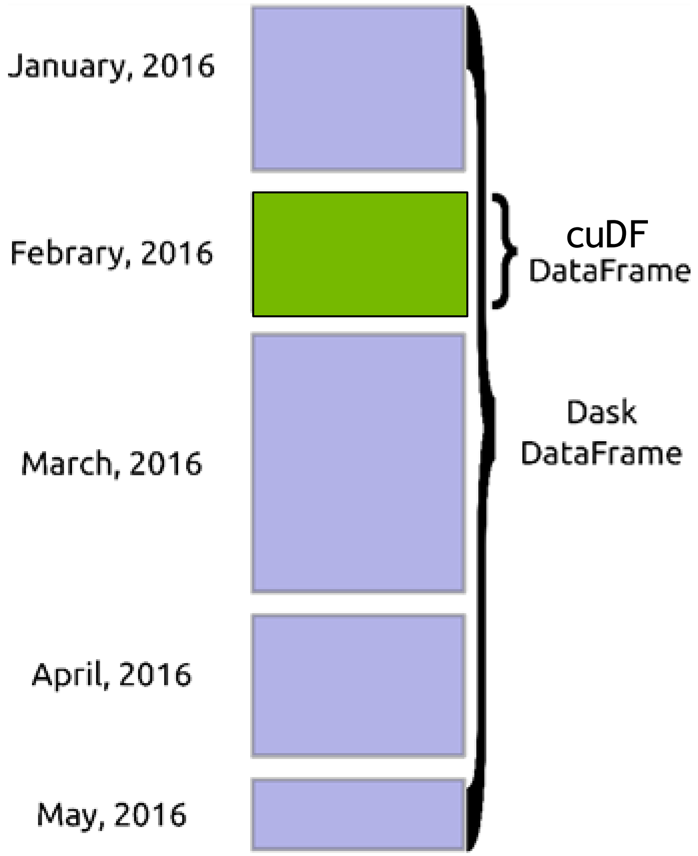
Combine Dask with cuDF

Many GPU DataFrames form a distributed DataFrame

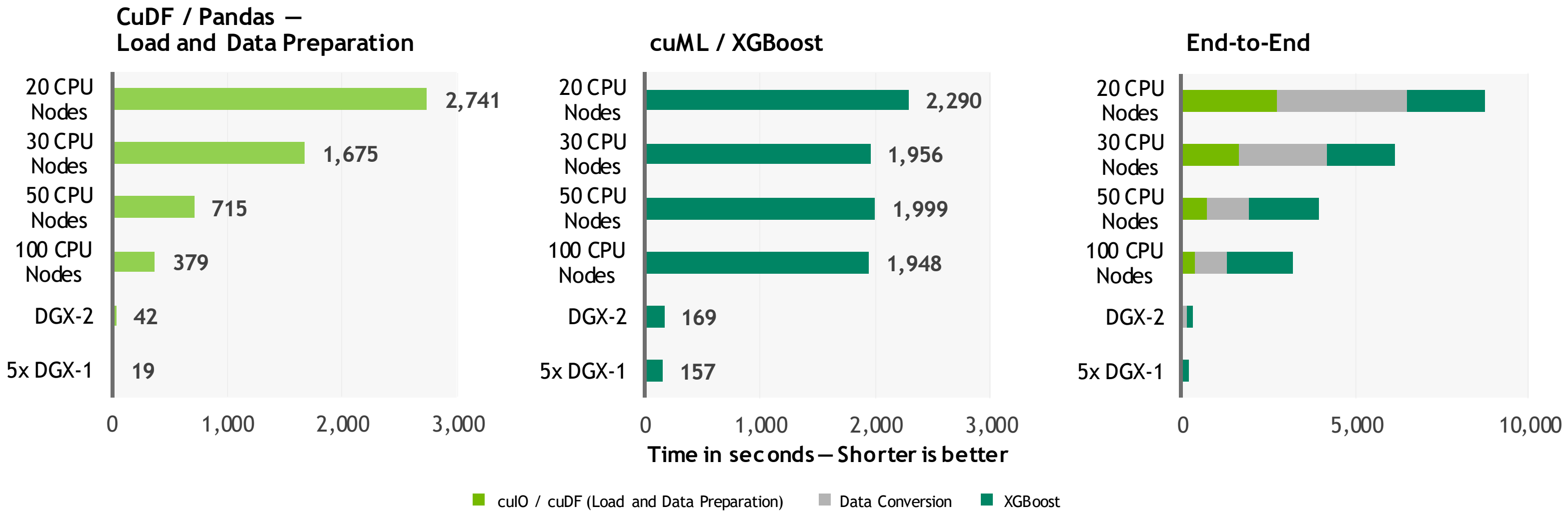


Combine Dask with cuDF

Many GPU DataFrames form a distributed DataFrame



END-TO-END BENCHMARKS



Benchmark

200GB CSV dataset; Data preparation includes joins, variable transformations.

CPU Cluster Configuration

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

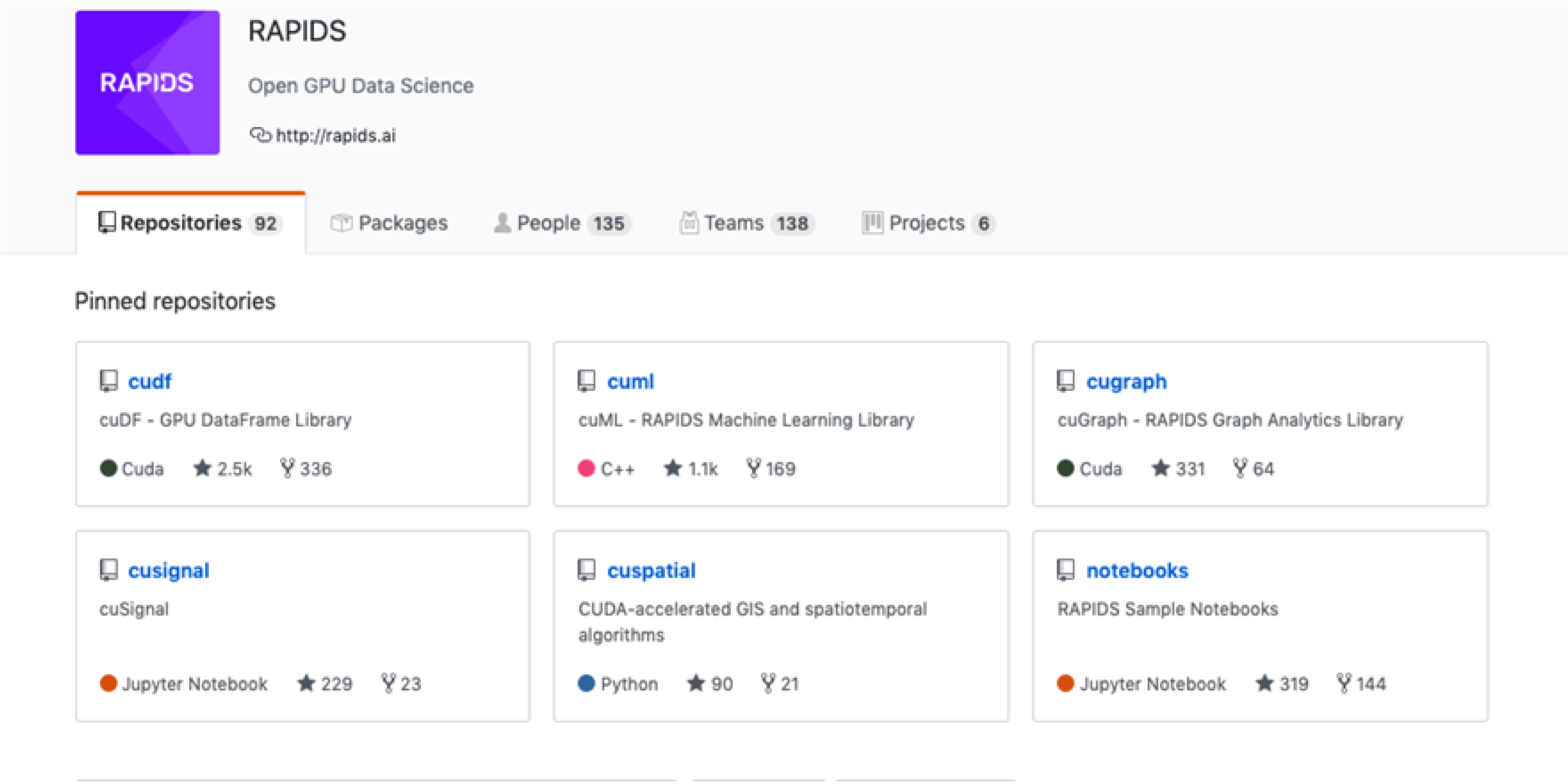
DGX Cluster Configuration

5x DGX-1 on InfiniBand network

Getting Started

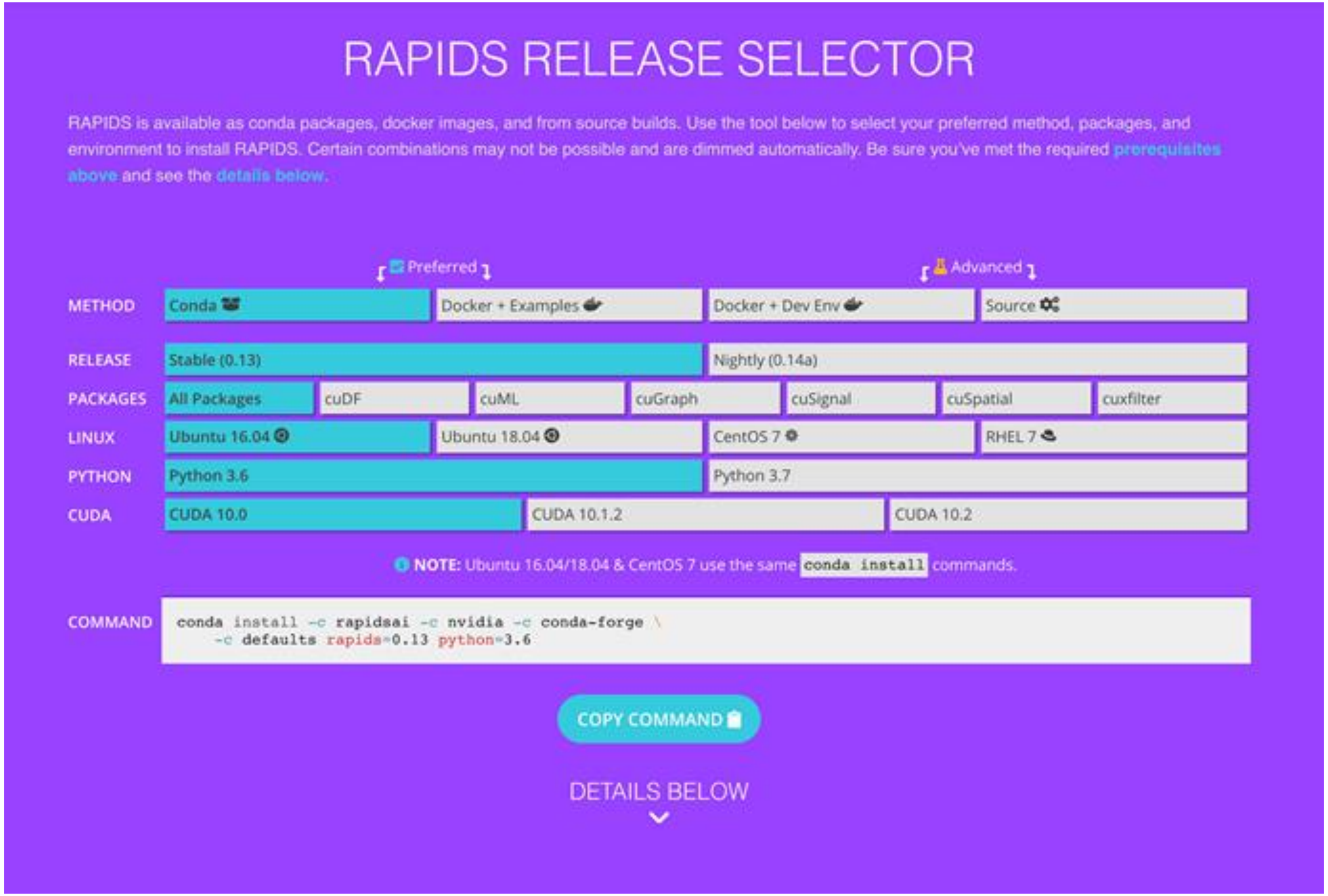
Explore: RAPIDS Github

https://github.com/rapidsai



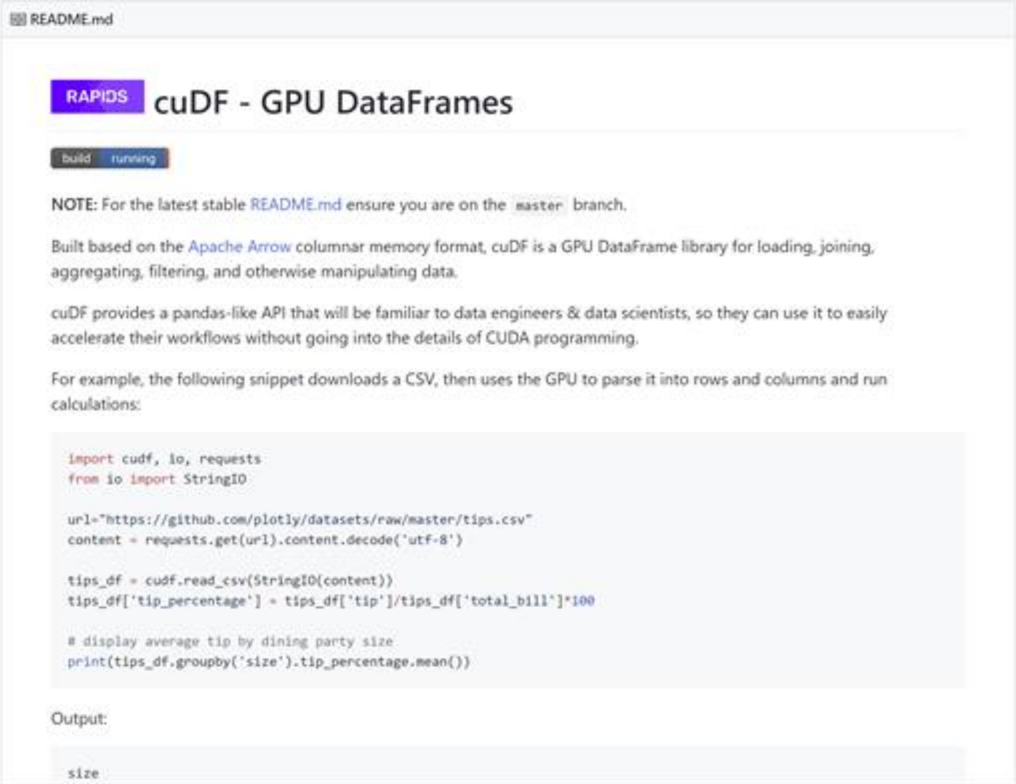
Easy Installation

Interactive Installation Guide



Explore: RAPIDS Code and Blogs

Check out our code and how we use it


README.md

RAPIDS cuDF - GPU DataFrames

build running

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.

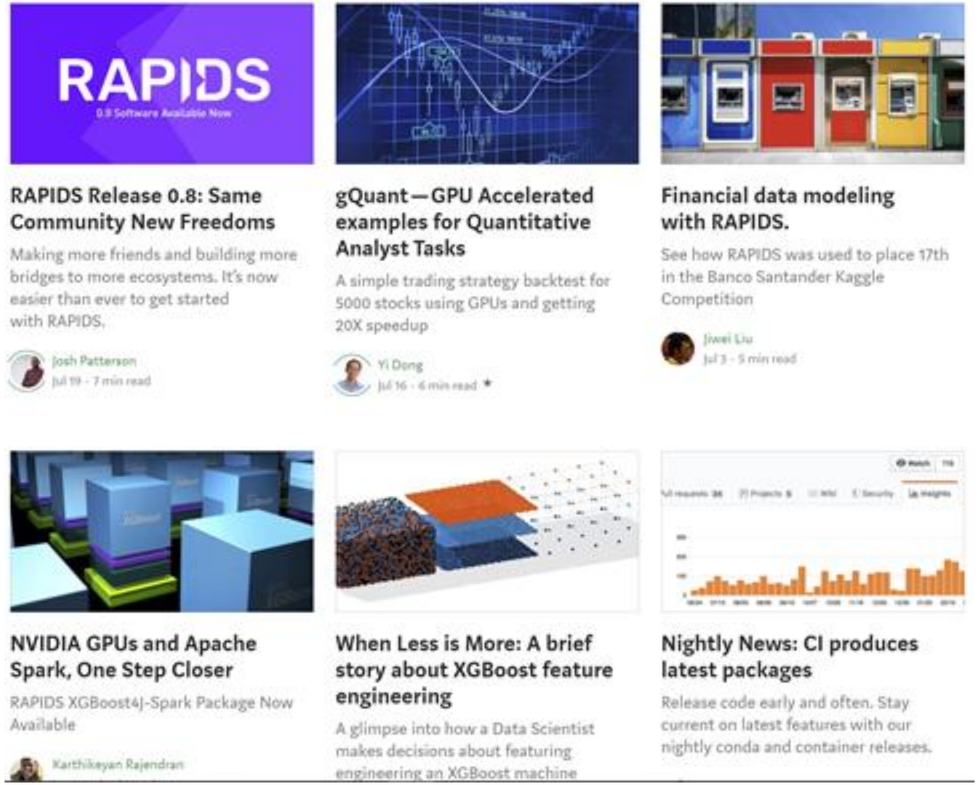
cuDF provides a pandas-like API that will be familiar to data engineers & data scientists, so they can use it to easily accelerate their workflows without going into the details of CUDA programming.

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

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

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

tips_df = cudf.read_csv(StringIO(content))
tips_df['tip_percentage'] = tips_df['tip']/tips_df['total_bill']*100

# display average tip by dining party size
print(tips_df.groupby('size').tip_percentage.mean())
```

Output:

```
size
```



https://github.com/rapidsai

https://medium.com/rapids-ai