RAPIDS: Accelerated Data Science and Data Processing

Mads R. B. Kristensen, NVIDIA

RAPIDS Accelerates Data Science End-to-End

Data Loading & ETL

RAPIDS cuDF
Tabular Data Loading and Processing

RAPIDS for Apache Spark
Apache Spark Accelerator

RAPIDS Dask
Dask Accelerator

Model Training & Analytics

RAPIDS cuGraph
Large Scale Graph Analytics

RAPIDS cuML
Machine Learning Toolkit

Deep Learning
cuGraph-powered GNNs

RAPIDS for Apache Spark
Apache Spark ML Accelerator

Vector Search & Model Inference

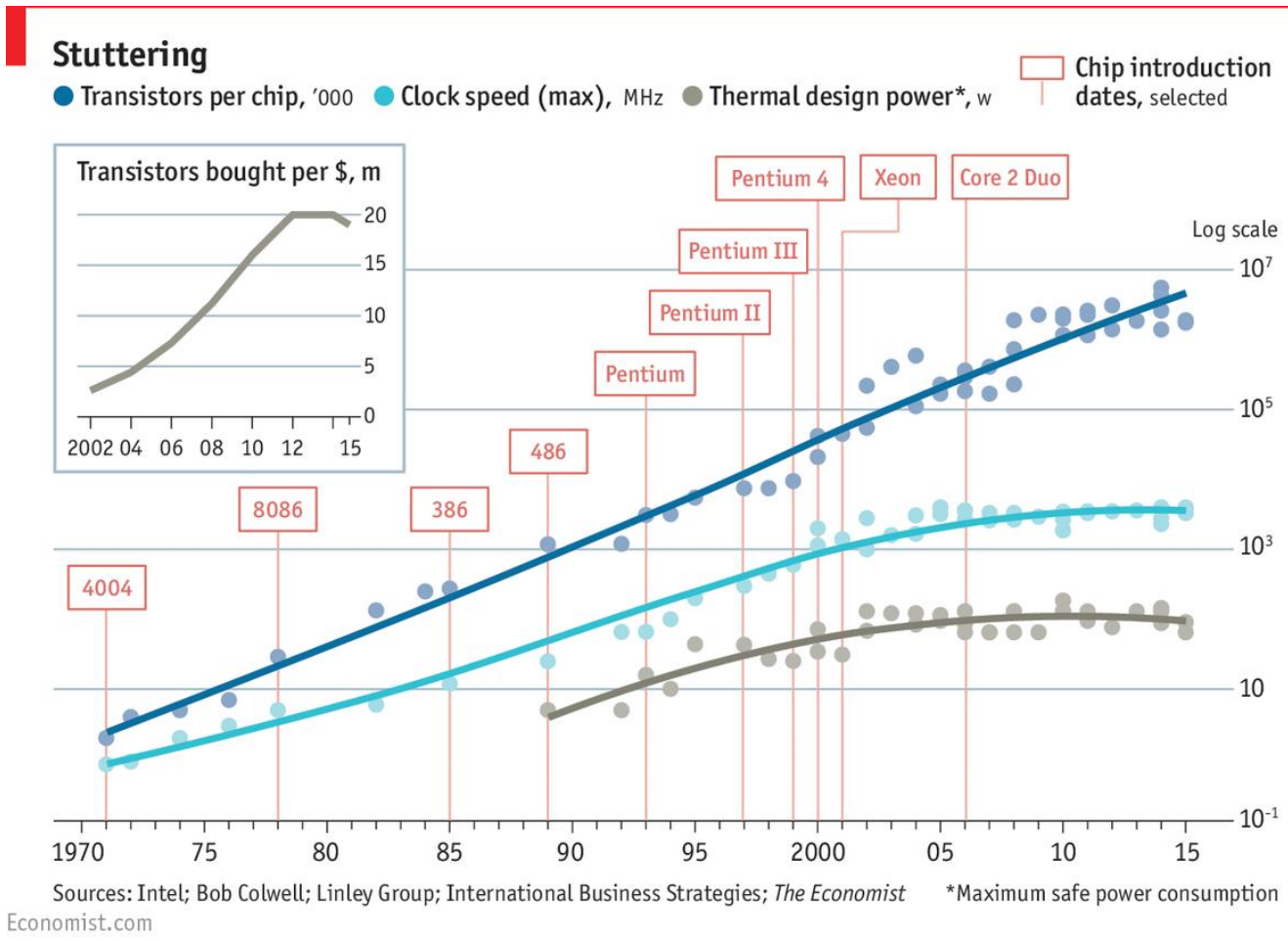
RAPIDS RAFT / cuVS
Vector Search & ML

Triton FIL
Optimized tree model inference

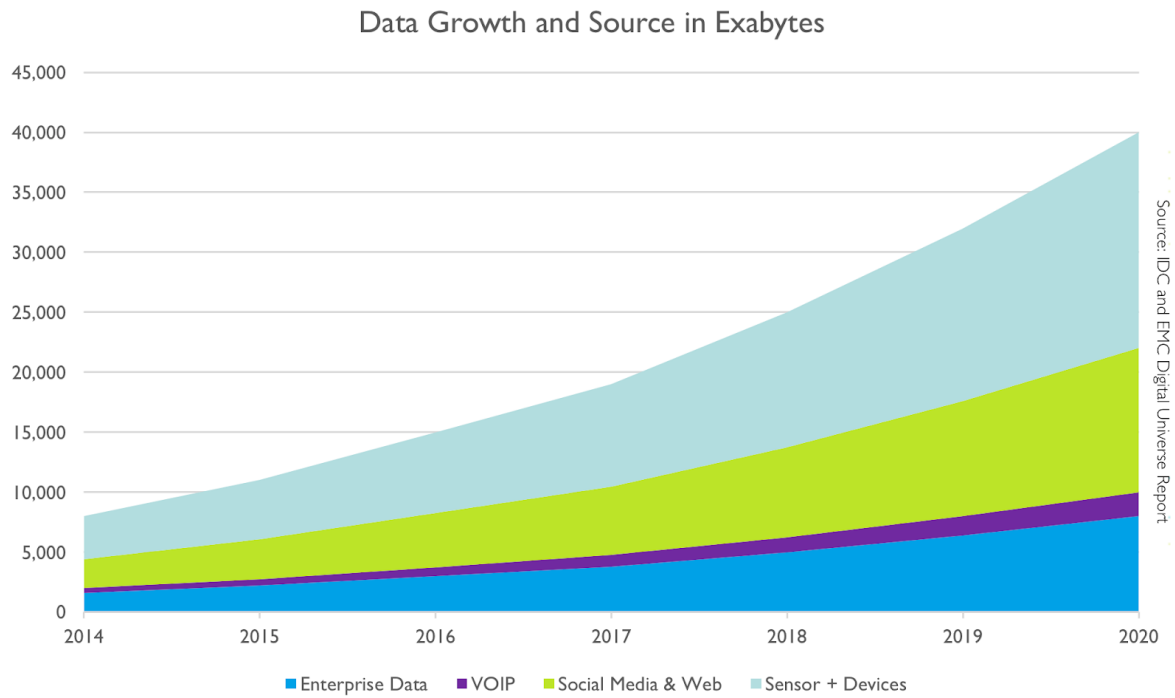
NVIDIA AI Enterprise

Development Tools | Cloud Native Management and Orchestration | Infrastructure Optimization

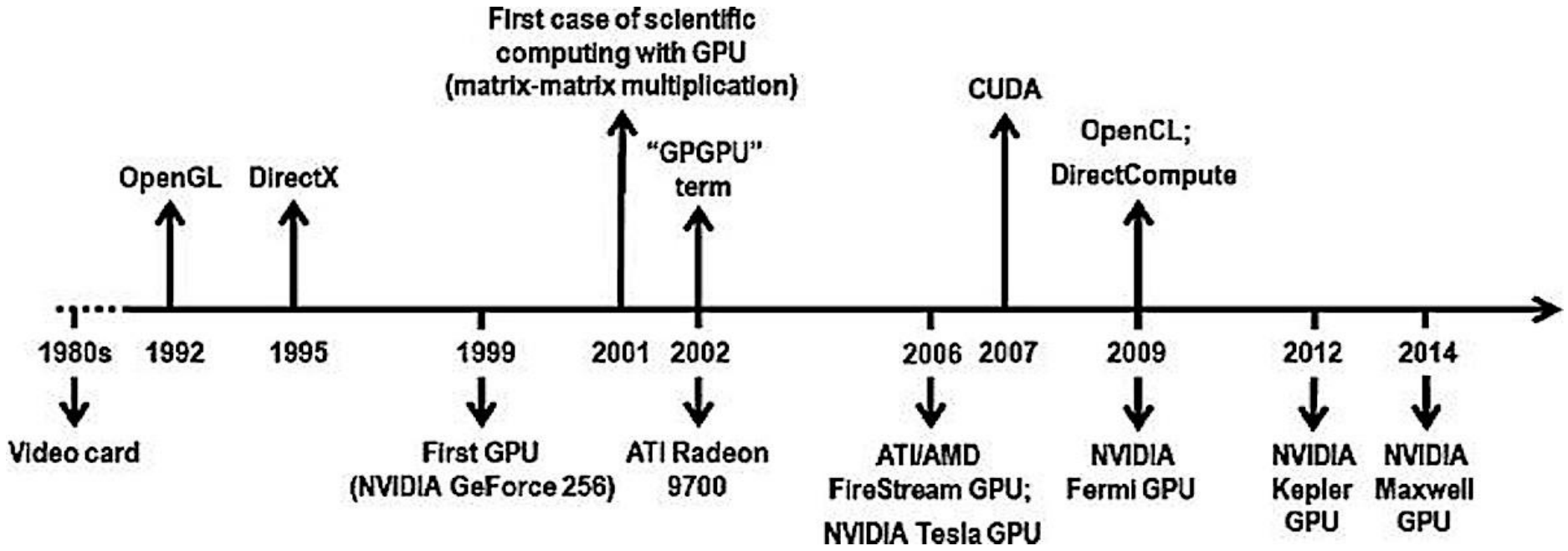
Cloud | Data Center | Edge | RTX Laptop



Moore's law: transistors doubles every two years

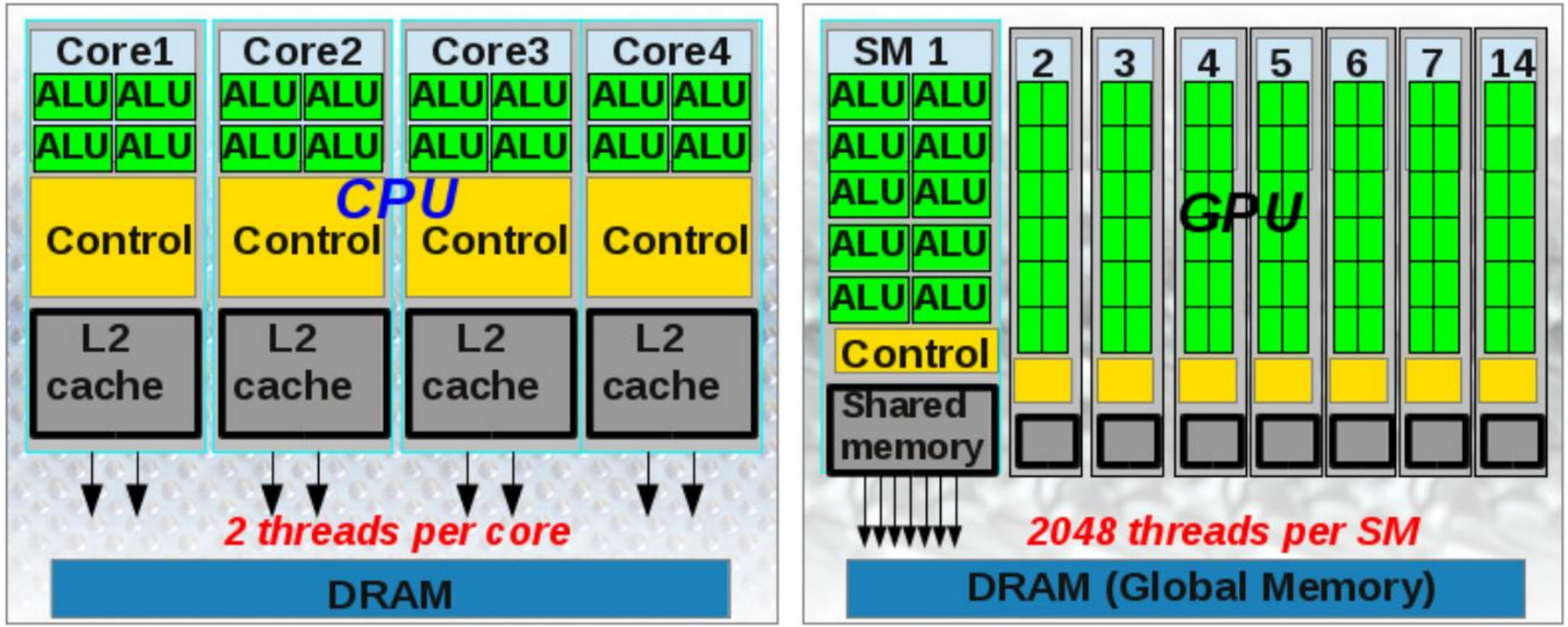


History of the GPU



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

CPU vs GPU

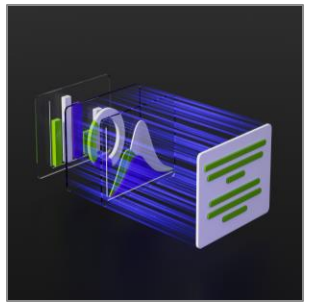


DOI:
10.1016/j.cam.2013.12.032.

Modern Enterprise Applications Need Accelerated Computing

Internet scale data | Massive models | Real-time performance

Recommenders



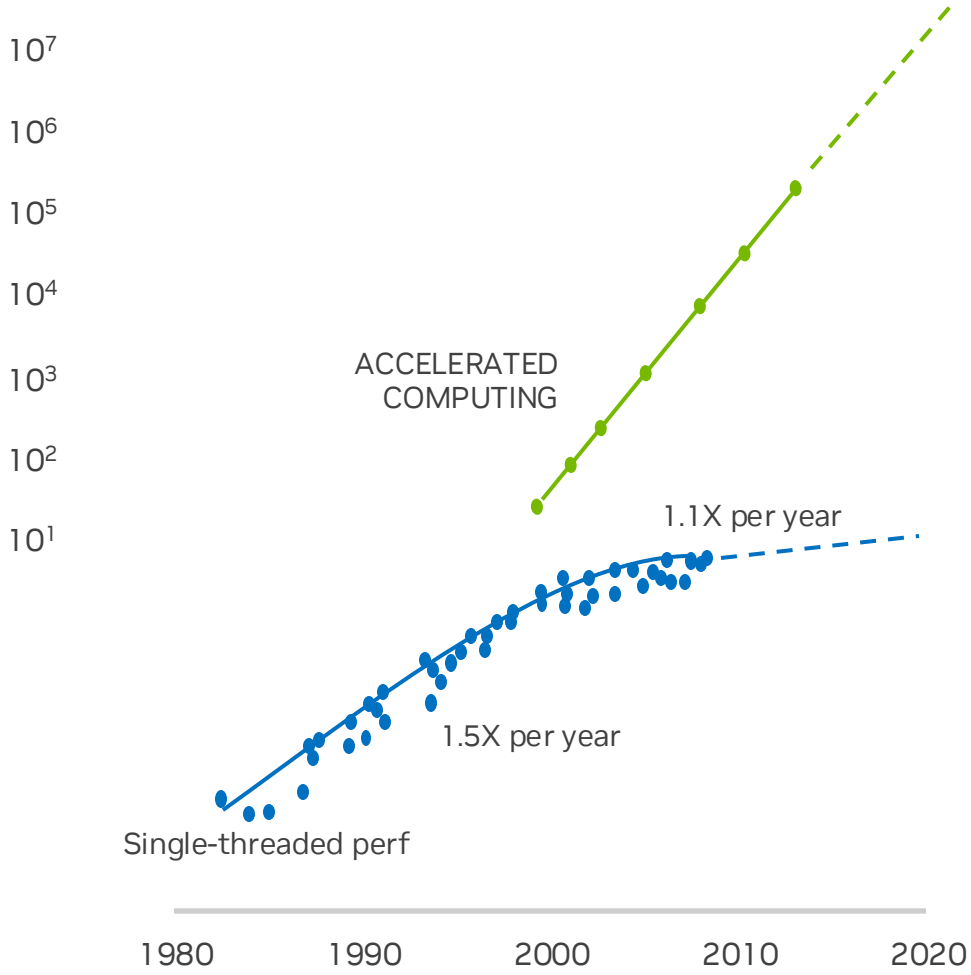
LLMs

Forecasting

Fraud Detection

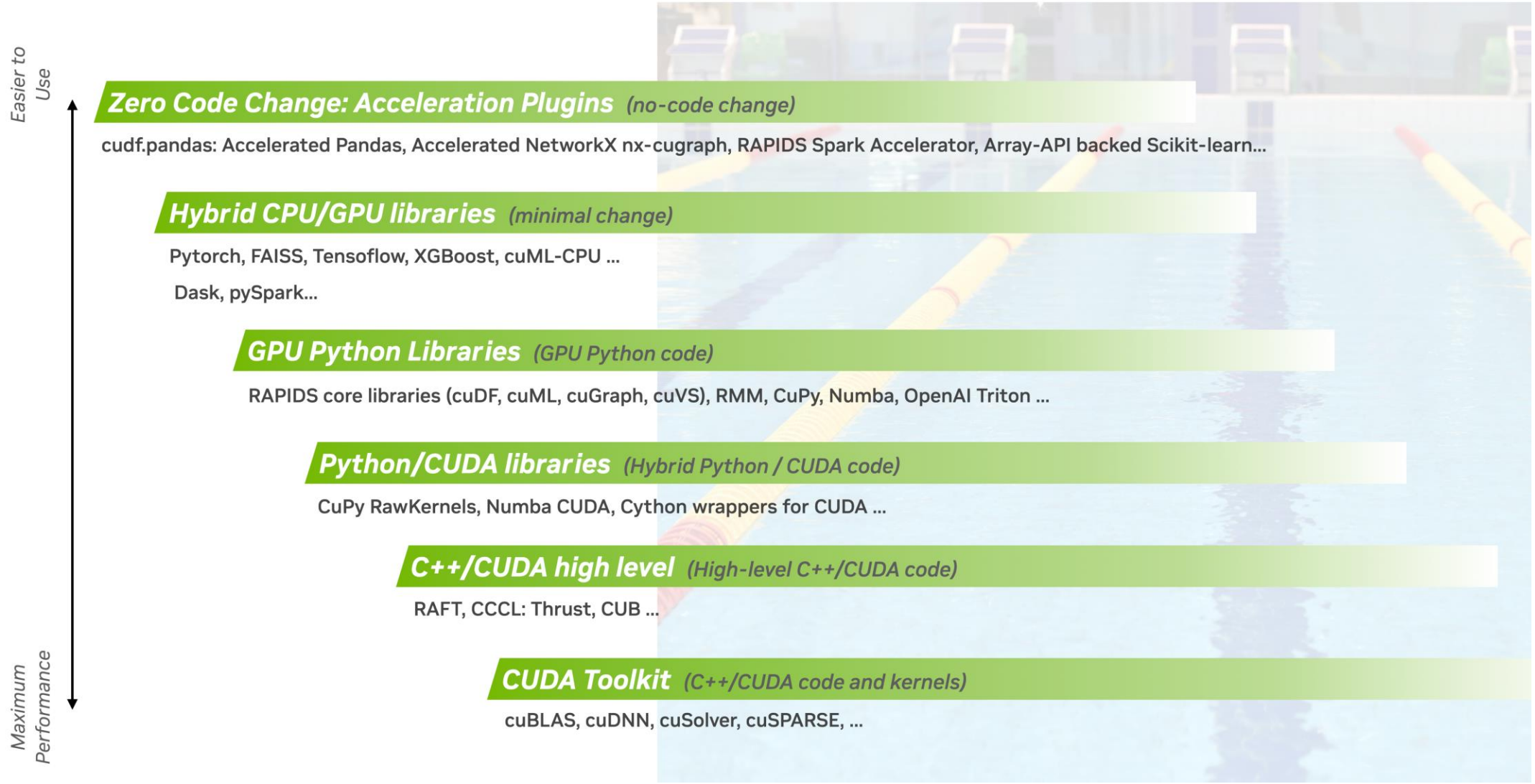
Genomic Analysis

Cybersecurity



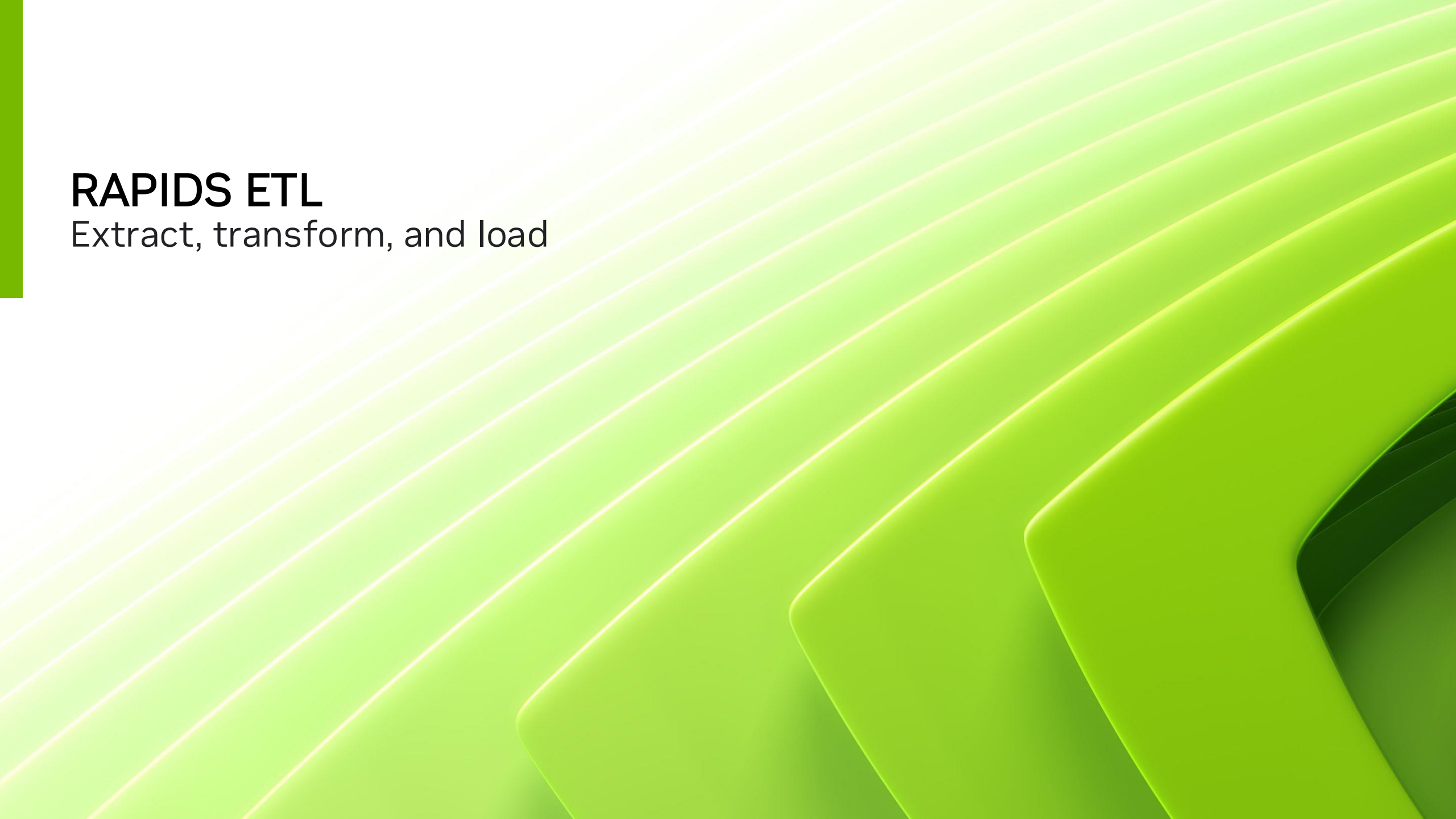
Accelerated Computing Swim Lanes

RAPIDS makes accelerated computing more seamless while enabling specialization for maximum performance



RAPIDS ETL

Extract, transform, and load

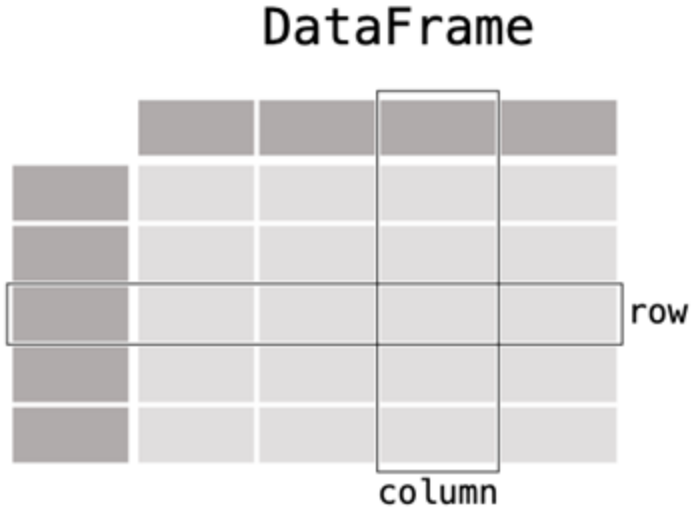


Pandas

Python's Preeminent DataFrame Library

9.5M
Pandas users

65%
Users love using pandas (HuggingFace most loved at 72%)




```python
import pandas as pd

df = (
    pd.read_csv('data.csv')
    .melt(id_vars=['id', 'name'])
    .rename(columns={
        'variable':'var',
        'value':'val'})
    .query('val >= 200')
    .sort_values('val', ascending=False)
)
```

135M+
Monthly downloads

25%
Y/Y downloads growth

cuDF - GPU DataFrames

```
[ ]: %%time
import pandas as pd

df = pd.read_csv("rows.csv",
                 usecols=[
                     "Registration State",
                     "Violation Description",
                     "Vehicle Body Type"
                 ])
(df[["Registration State", "Violation Description"]]
 .value_counts()  # count of offences per state per offence
 .groupby("Registration State")  # grouped by state
 .head(1)  # offence with largest count
 .sort_index()  # sort by state name
 .reset_index()
 .head(5)
)
```

```
[ ]: %%time
import cudf

df = cudf.read_csv("rows.csv",
                   usecols=[
                       "Registration State",
                       "Violation Description",
                       "Vehicle Body Type"
                   ])
(df[["Registration State", "Violation Description"]]
 .value_counts()  # count of offences per state per offence
 .groupby("Registration State")  # grouped by state
 .head(1)  # offence with largest count
 .sort_index()  # sort by state name
 .reset_index()
 .head(5)
)
```

A GPU DataFrame library in Python with a pandas-like API built into the PyData ecosystem

Pandas-like API on the GPU

CPU

pandas

```
>>> import pandas as pd
>>> df = pd.read_csv("filepath")
>>> df.groupby("col").mean()
>>> df.rolling(window=3).sum()
```

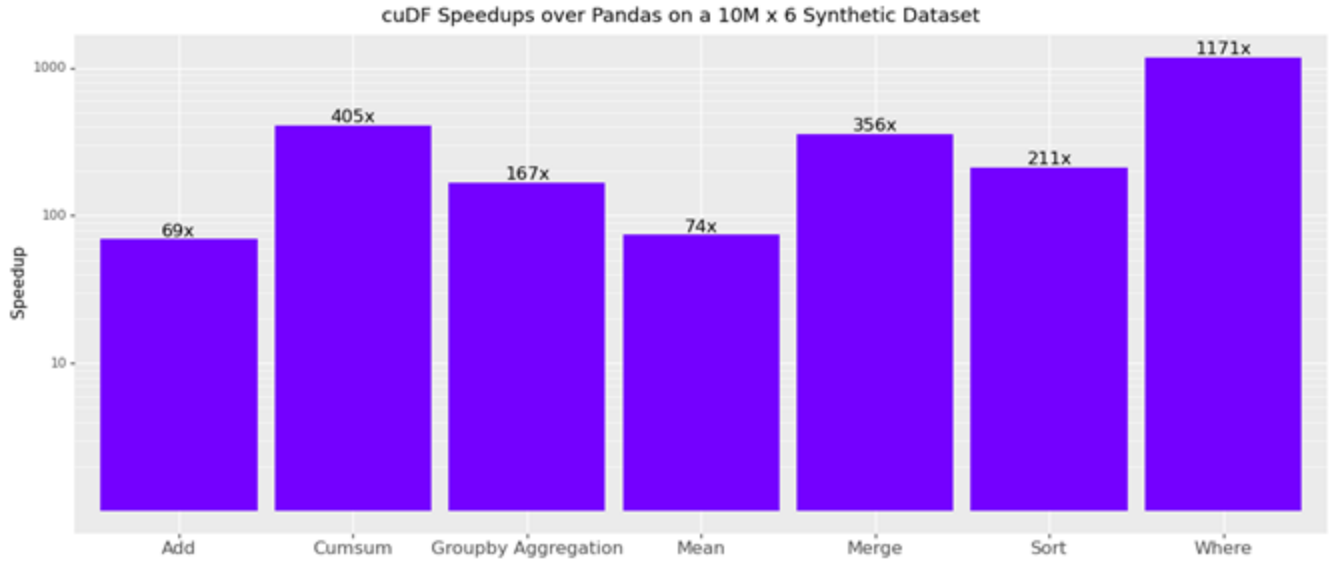
GPU

cuDF

```
>>> import cudf
>>> df = cudf.read_csv("filepath")
>>> df.groupby("col").mean()
>>> df.rolling(window=3).sum()
```

Average Speed-Ups: *10-100x*

Best-in-Class Performance (Benchmark)



Groupby	Strings and Regex	UDFs	Nested Types	Time Series
Indexing	Missing Data	CuPy Interoperability	Rolling Windows	

10 Minutes to cuDF

NVIDIA A100 vs. AMD EPYC 7642 48-Core Processor
cuDF Python vs. Pandas

cuDF Problems

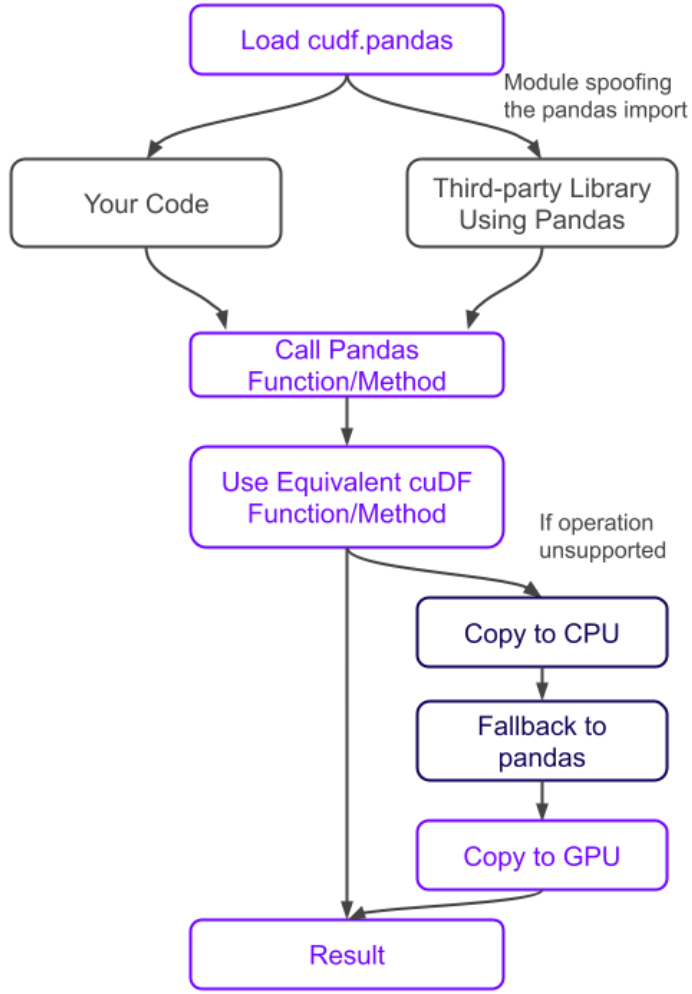
cuDF coverage of the Pandas API (green=implemented, gray=not implemented)

"We just don't have time to rewrite our code in a new paradigm."

Accelerated pandas

cudf.pandas: the zero code change GPU accelerator for pandas built on cuDF

- Requires *no changes* to existing pandas code. Just
  - `%load_ext cudf.pandas`
  - `$ python -m cudf.pandas <script.py>`
- 100% of the pandas API
- Accelerates workflows up to 150x using the GPU
- Compatible with code that uses third-party libraries
- Falls back to using pandas on the CPU for unsupported functions and methods

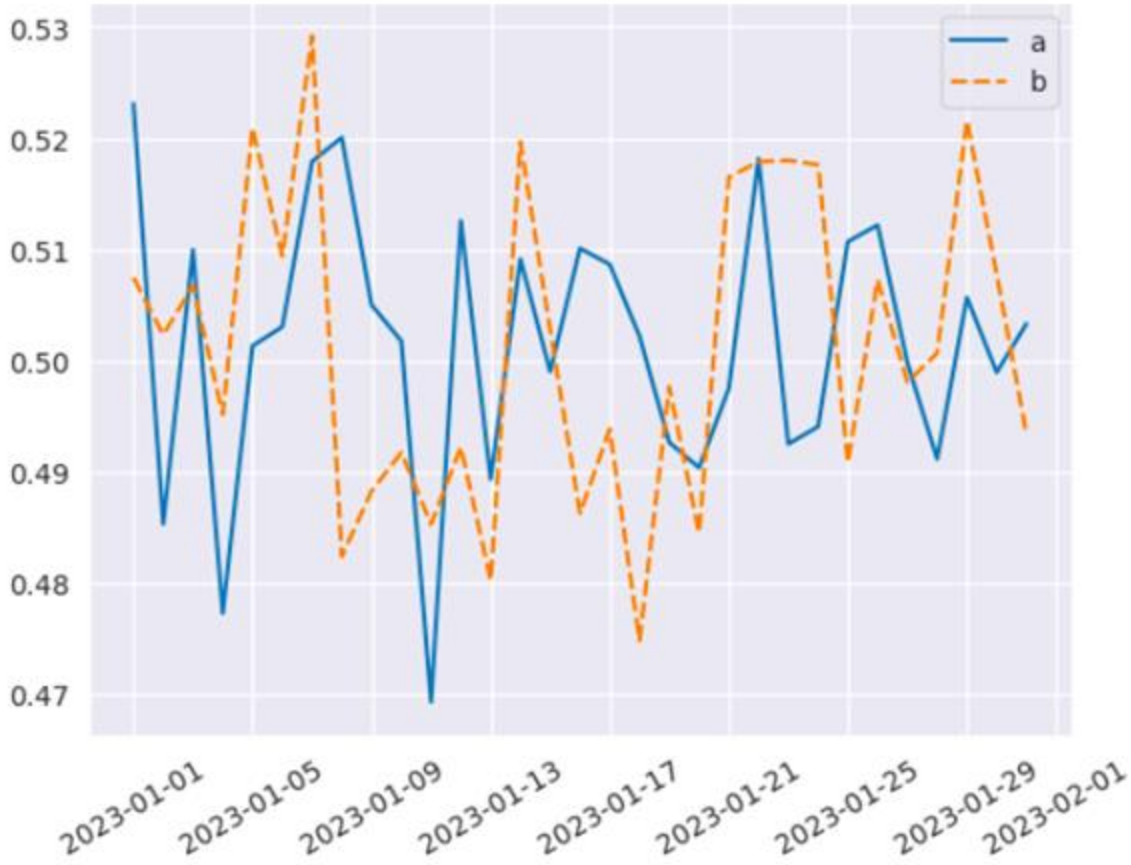



```
[ ]:
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt

data = pd.read_parquet("data.parquet")
subset = data.index.indexer_between_time("09:30", "16:00")
data = data.iloc[subset]
results = data.groupby(pd.Grouper(freq="1D")).mean()

sns.lineplot(results)
plt.xticks(rotation=30)
```

cudf.pandas in Action

A brief example

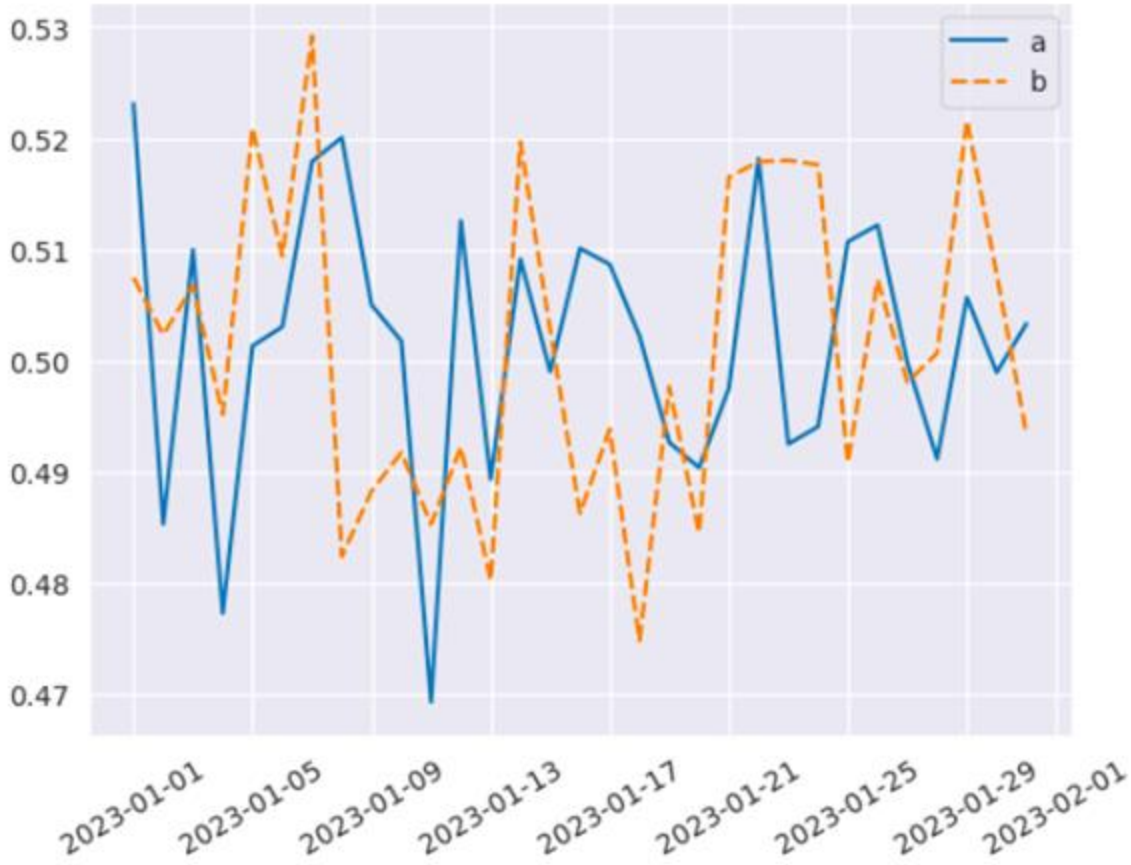
```
%%load_ext cudf.pandas

import pandas as pd
import matplotlib.pyplot as plt
import numpy as np
import seaborn as sns

rng = pd.date_range("2023-01-01", "2023-02-01",  freq="1T")
df = pd.DataFrame({
    "a": np.random.rand(len(rng)),
    "b": np.random.rand(len(rng))
    },
    index=rng
)

df = df.iloc[rng.indexer_between_time("09:30", "16:00")]
results = df.groupby(pd.Grouper(freq="1D")).mean()

_ = sns.lineplot(results)
_ = plt.xticks(rotation=30)
```



cudf.pandas in Action

A brief example

```
%%load_ext cudf.pandas

import pandas as pd
import matplotlib.pyplot as plt
import numpy as np
import seaborn as sns

rng = pd.date_range("2023-01-01", "2023-02-01",  freq="1T")
df = pd.DataFrame({
    "a": np.random.rand(len(rng)),
    "b": np.random.rand(len(rng))
    },
    index=rng
)

df = df.iloc[rng.indexer_between_time("09:30", "16:00")]
results = df.groupby(pd.Grouper(freq="1D")).mean()

_ = sns.lineplot(results)
_ = plt.xticks(rotation=30)
```

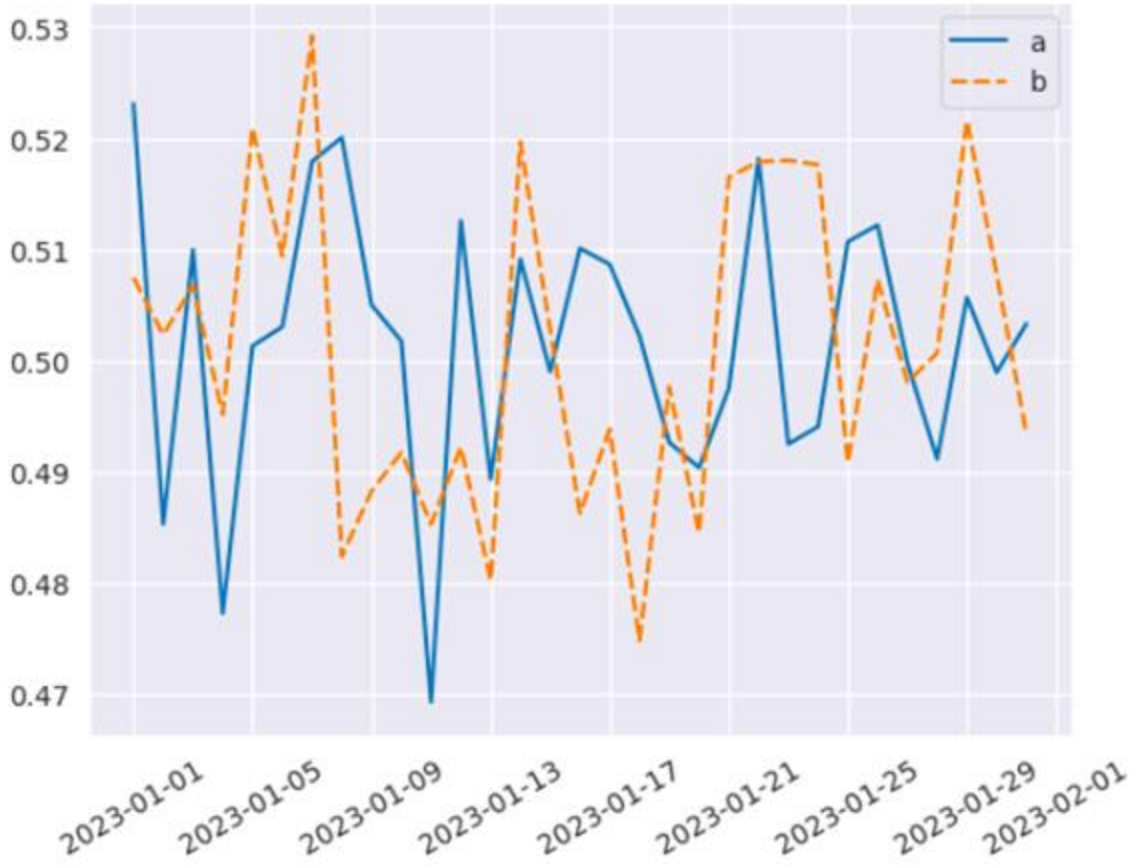


This part runs entirely on the **GPU**
cuDF supports all these operations

cudf.pandas in Action

A brief example

```
%%load_ext cudf.pandas

import pandas as pd
import matplotlib.pyplot as plt
import numpy as np
import seaborn as sns

rng = pd.date_range("2023-01-01", "2023-02-01",  freq="1T")
df = pd.DataFrame({
    "a": np.random.rand(len(rng)),
    "b": np.random.rand(len(rng))
    },
    index=rng
)

df = df.iloc[rng.indexer_between_time("09:30", "16:00")]
results = df.groupby(pd.Grouper(freq="1D")).mean()

_ = sns.lineplot(results)
_ = plt.xticks(rotation=30)
```

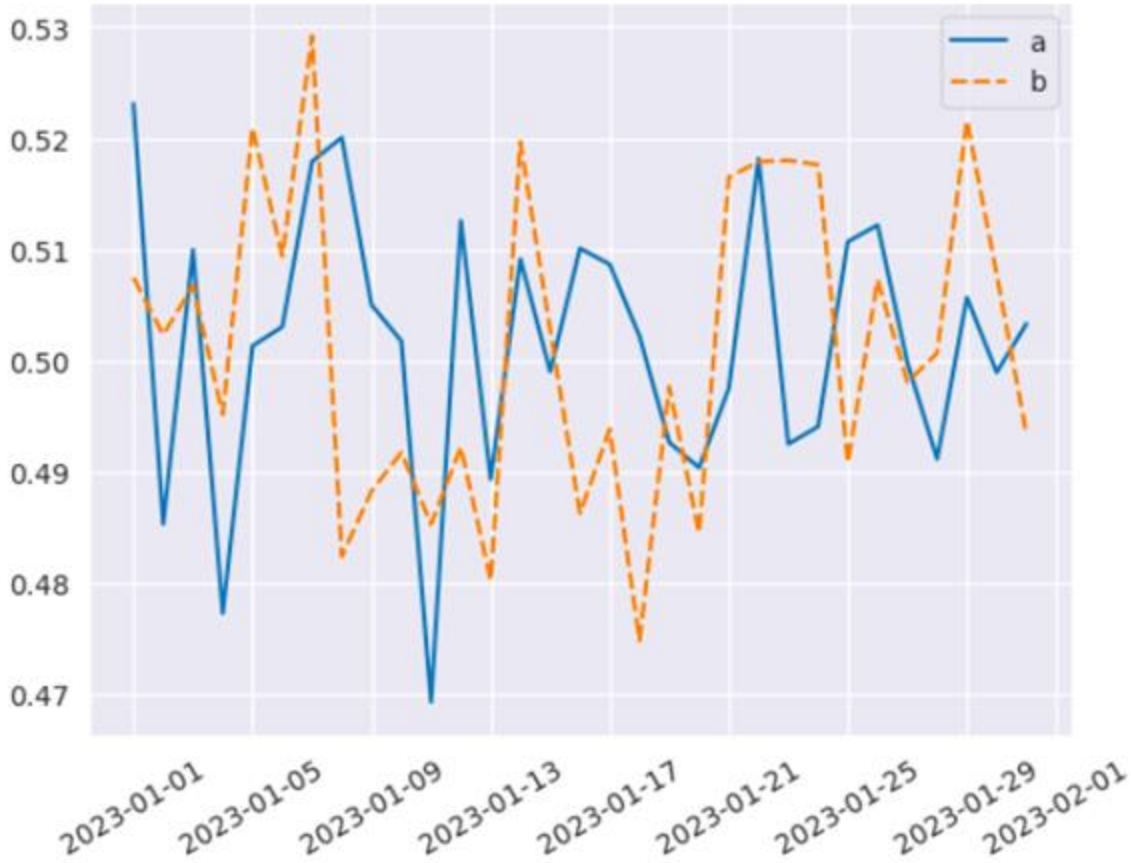


`indexer_between_time` isn't supported on the GPU — so it runs on the CPU

`cudf.pandas` in Action

A brief example

```
%%load_ext cudf.pandas

import pandas as pd
import matplotlib.pyplot as plt
import numpy as np
import seaborn as sns

rng = pd.date_range("2023-01-01", "2023-02-01",  freq="1T")
df = pd.DataFrame({
    "a": np.random.rand(len(rng)),
    "b": np.random.rand(len(rng))
    },
    index=rng
)

df = df.iloc[rng.indexer_between_time("09:30", "16:00")]
results = df.groupby(pd.Grouper(freq="1D")).mean()

_ = sns.lineplot(results)
_ = plt.xticks(rotation=30)
```

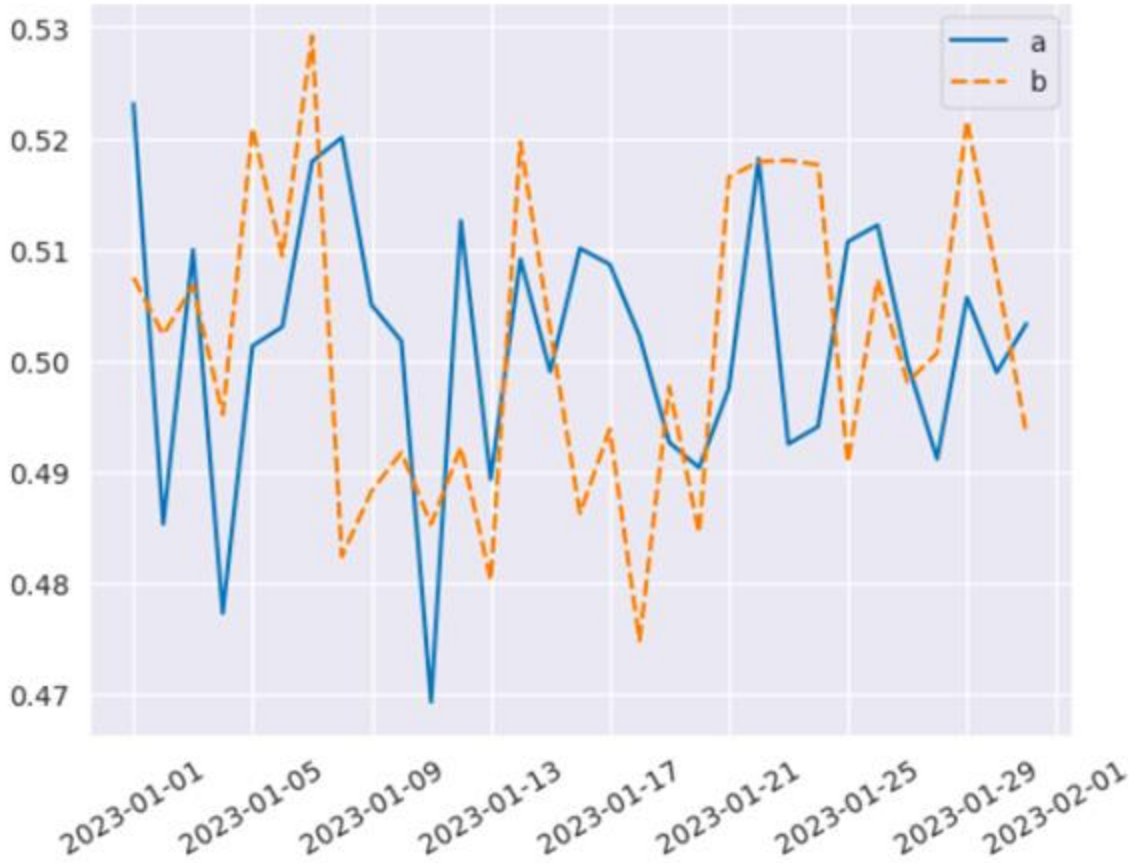


But this part happens on the GPU. The result of `indexer_between_time` is copied back from CPU to GPU

cudf.pandas in Action

A brief example

```
%%load_ext cudf.pandas

import pandas as pd
import matplotlib.pyplot as plt
import numpy as np
import seaborn as sns

rng = pd.date_range("2023-01-01", "2023-02-01",  freq="1T")
df = pd.DataFrame({
    "a": np.random.rand(len(rng)),
    "b": np.random.rand(len(rng))
    },
    index=rng
)

df = df.iloc[rng.indexer_between_time("09:30", "16:00")]
results = df.groupby(pd.Grouper(freq="1D")).mean()

_ = sns.lineplot(results)
_ = plt.xticks(rotation=30)
```

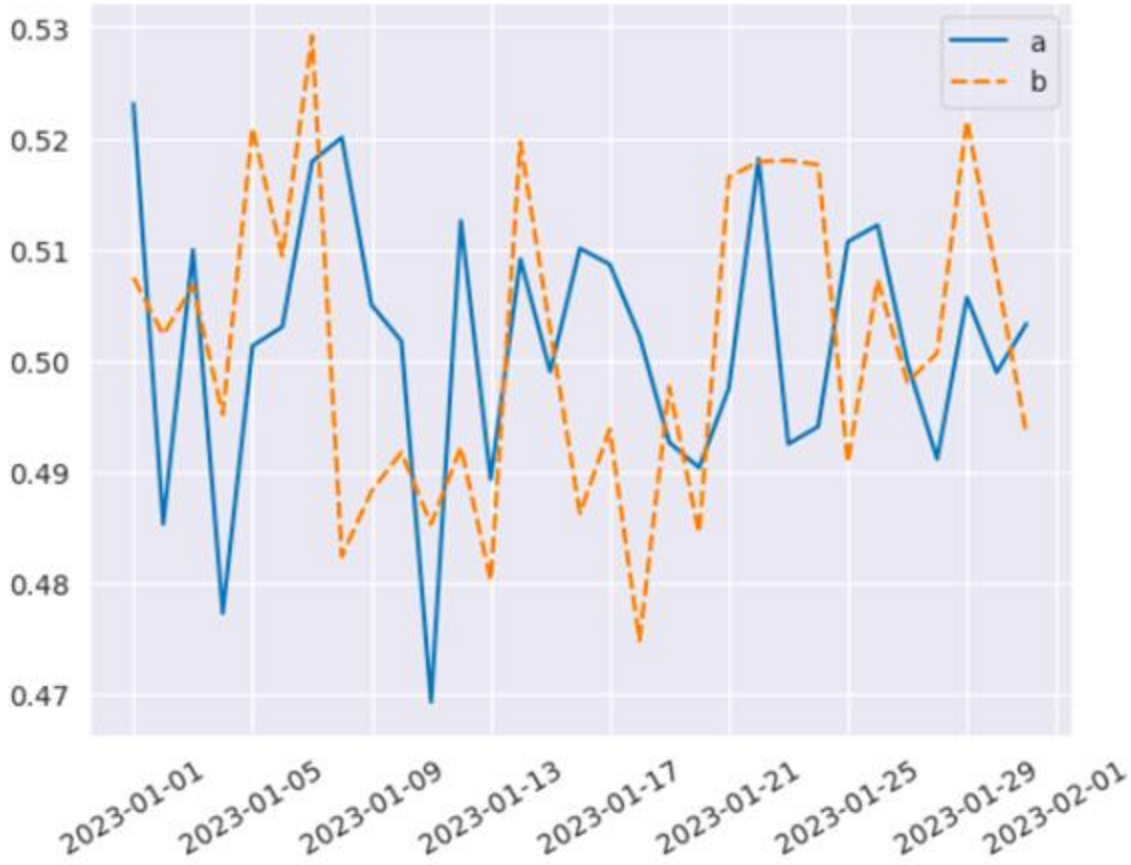


This part runs entirely on the **GPU**

cudf.pandas in Action

A brief example

```
%%load_ext cudf.pandas

import pandas as pd
import matplotlib.pyplot as plt
import numpy as np
import seaborn as sns

rng = pd.date_range("2023-01-01", "2023-02-01",  freq="1T")
df = pd.DataFrame({
    "a": np.random.rand(len(rng)),
    "b": np.random.rand(len(rng))
    },
    index=rng
)

df = df.iloc[rng.indexer_between_time("09:30", "16:00")]
results = df.groupby(pd.Grouper(freq="1D")).mean()

_ = sns.lineplot(results)
_ = plt.xticks(rotation=30)
```



We can seamlessly interoperate with third-party libraries like Seaborn

cudf.pandas summary

Provides **all** of the Pandas API

Uses the GPU (via cuDF) for operations supported by cuDF

Uses the CPU (via Pandas) for operations not supported by cuDF

Data movement is completely hidden from the user

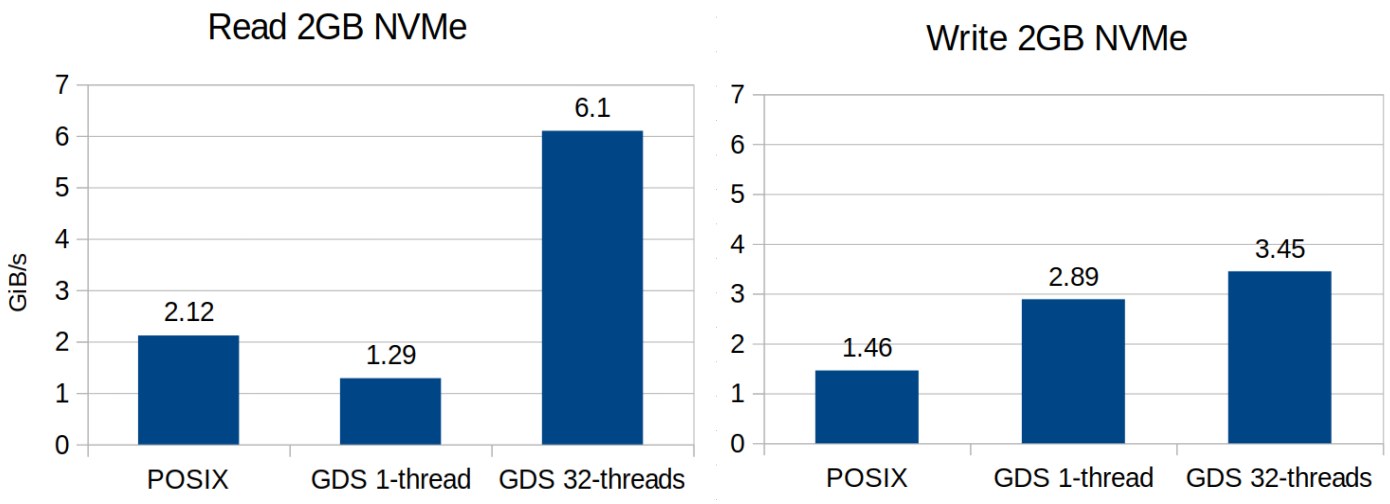
Zero code change: accelerates Pandas “in-place”

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.

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

```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.
  }
}
```

```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
```

RAPIDS ML and Graph Analytics

cuML

Accelerated machine learning with a scikit-learn API

CPU

Scikit-learn

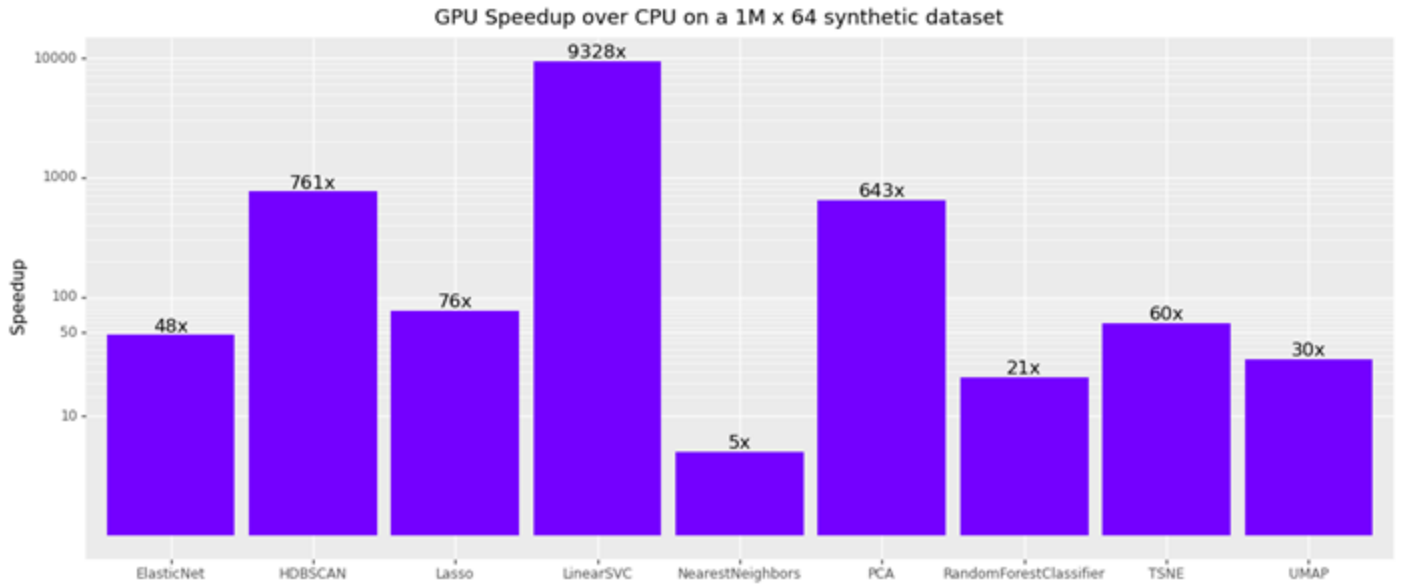
```
>>> from sklearn.ensemble import
RandomForestClassifier
>>> clf = RandomForestClassifier()
>>> clf.fit(x, y)
```

GPU

cuML

```
>>> from cuml.ensemble import
RandomForestClassifier
>>> clf = RandomForestClassifier()
>>> clf.fit(x, y)
```

50+ GPU-Accelerated Algorithms



Time Series	Classification	Regression	Clustering	Preprocessing
Cross Validation	Tree Models	Dimensionality Reduction	Explainability	

A100 GPU vs. AMD EPYC 7642 (96 logical cores)
cuML 23.04, scikit-learn 1.2.2, umap-learn 0.5.3

Accelerated XGBoost

"XGBoost is All You Need" – Bojan Tunguz, 4x Kaggle Grandmaster

CPU

XGBoost

```
>>> from xgboost import XGBClassifier
>>> clf = XGBClassifier()
>>> clf.fit(x, y)
```

GPU

XGBoost

```
>>> from xgboost import XGBClassifier
>>> clf = XGBClassifier(device="cuda")
>>> clf.fit(x, y)
```

Up to 20x Speedups

- One line of code change to unlock up to 20x speedups with GPUs
- Scalable to the world's largest datasets with Dask and PySpark
- Built-in SHAP support for model explainability
- Deployable with Triton for lighting-fast inference in production
- RAPIDS helps maintain the XGBoost project

RAPIDS ❤ dmlc XGBoost

Available Algorithms

K-Means

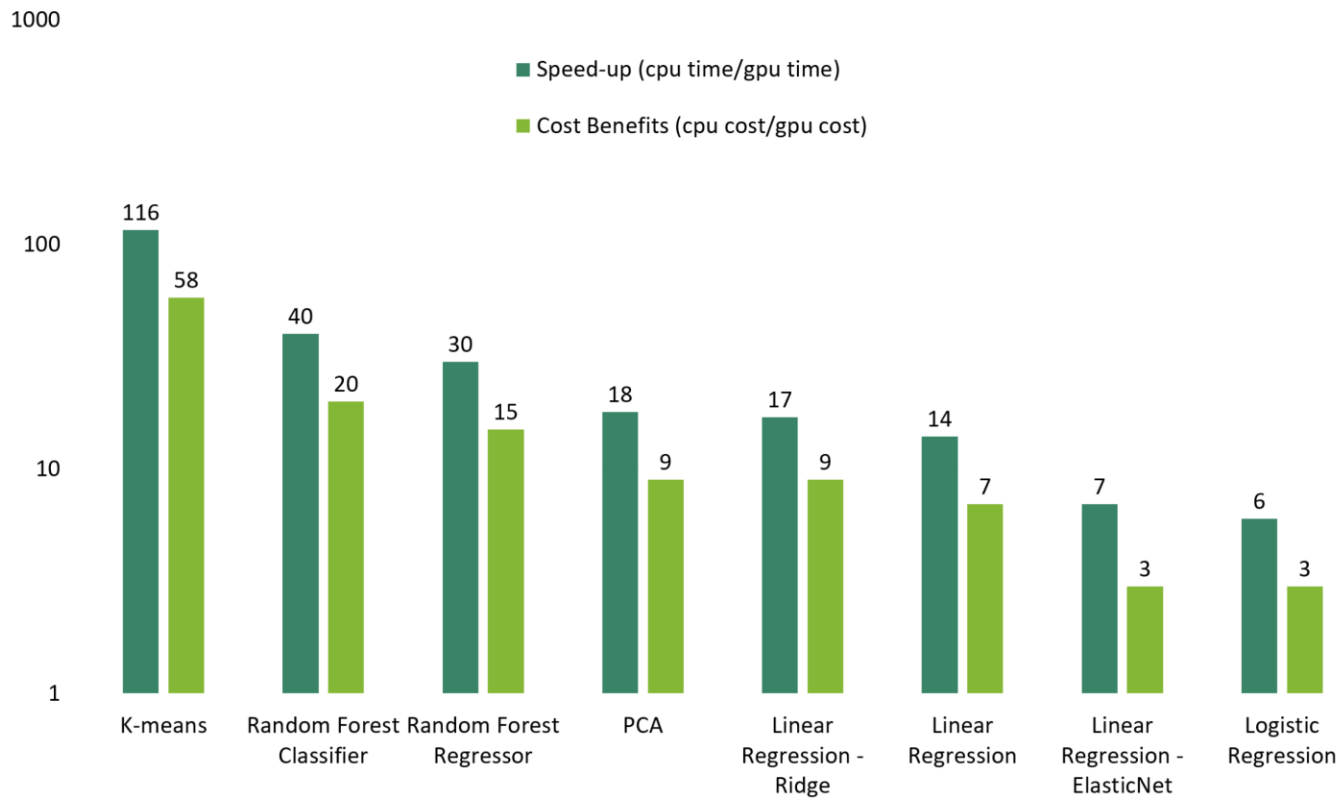
Linear Regression

Logistic Regression[1]

PCA

Random Forest Classifier

Random Forest Regressor

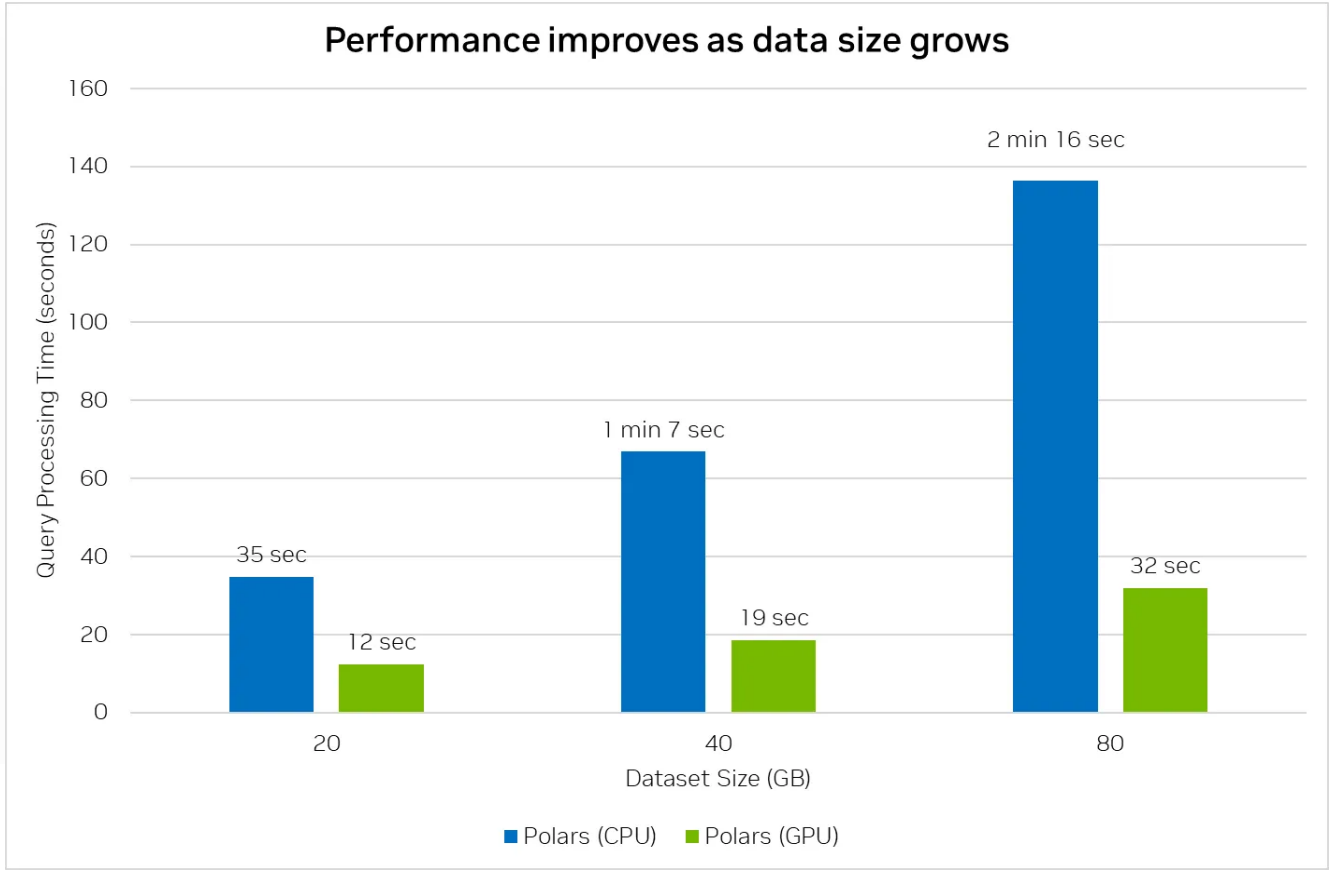


Polars

CuDF - Polars

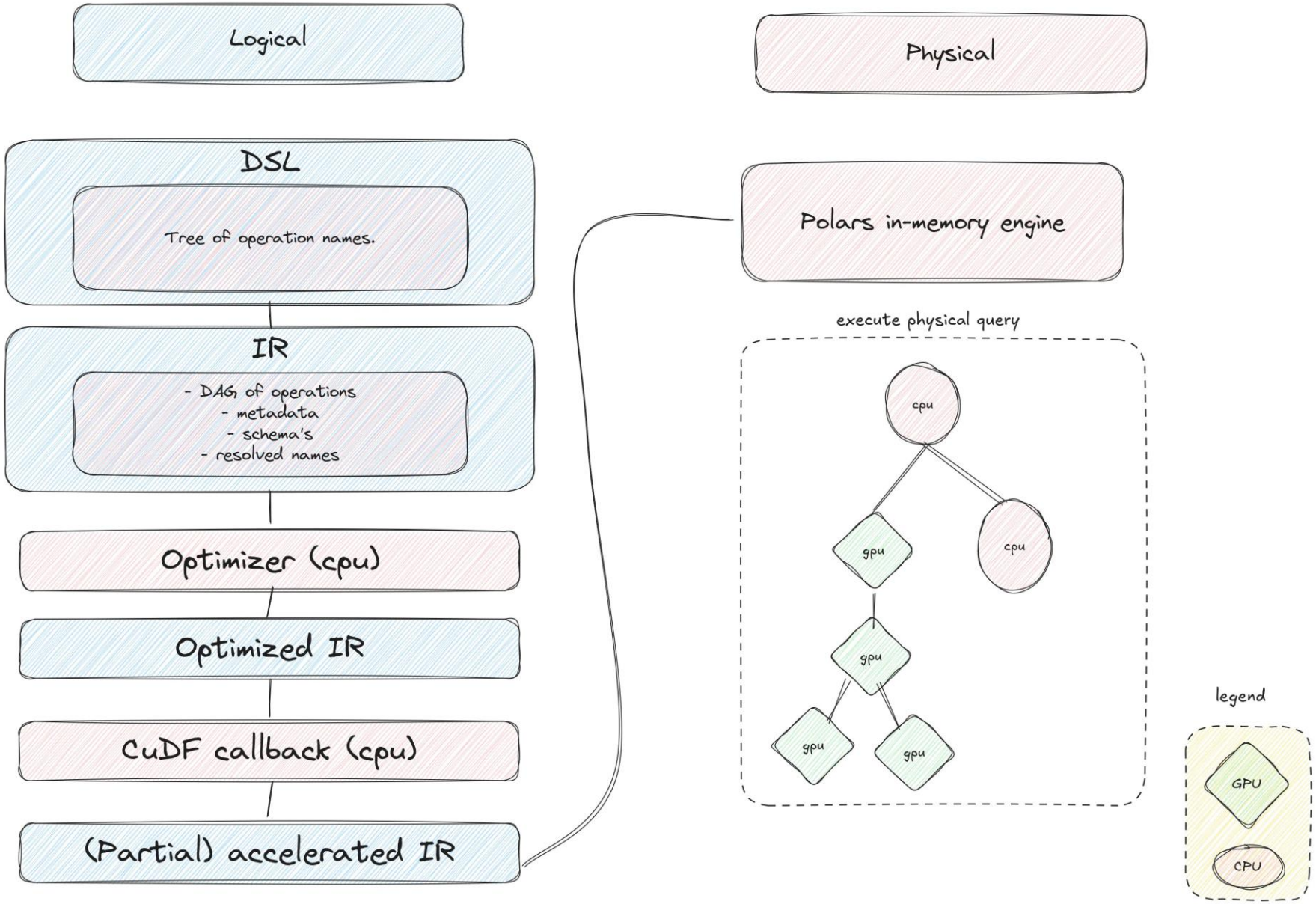
Dataframe library similar to Pandas for Python and Rust

```python
import polars as pl

q = (
    pl.scan_csv("docs/assets/data/iris.csv")
    .filter(pl.col("sepal_length") > 5)
    .group_by("species")
    .agg(pl.all().sum())
)

# Run on the CPU
result_cpu = q.collect()

# Run on the GPU
result_gpu = query.collect(engine="gpu")
```



- H100 PCIe, 80GiB; single socket Intel Xeon W9-3495X (56 physical cores)
- Total runtime for PDS-H benchmark
  - https://github.com/pola-rs/polars-benchmark

Source https://pola.rs/posts/gpu-engine-release/



Source https://pola.rs/posts/gpu-engine-release/

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

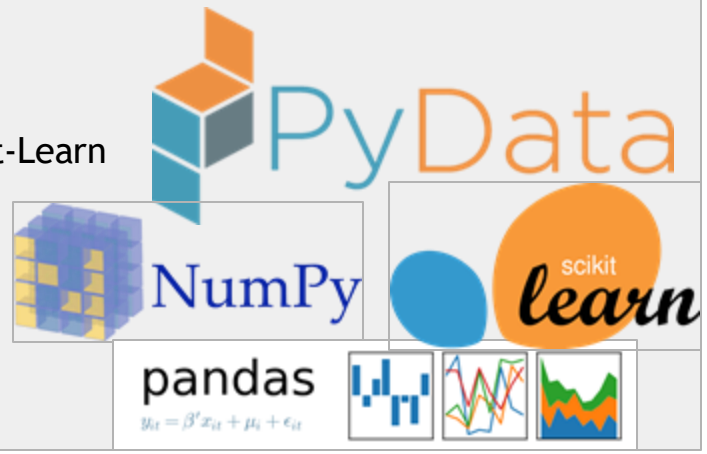
RAPIDS

DASK

PyData

NumPy, Pandas, Scikit-Learn and many more

Single CPU core
In-memory data



Dask

Multi-core and Distributed PyData

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

DASK

Scale out / Parallelize

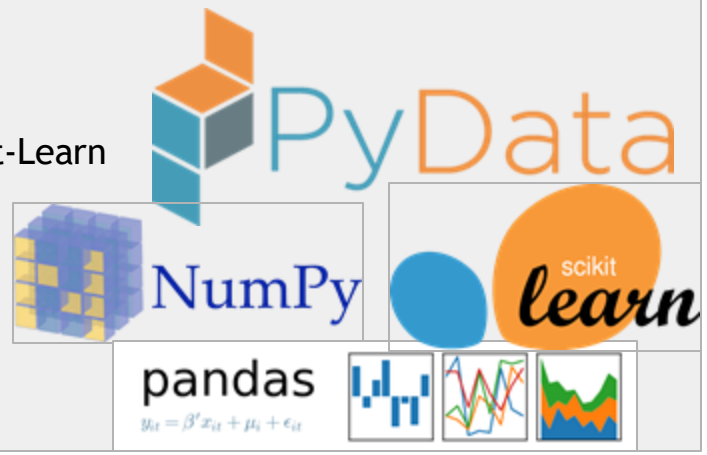
Scale up and out with RAPIDS and Dask

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

DASK

Scale out / Parallelize

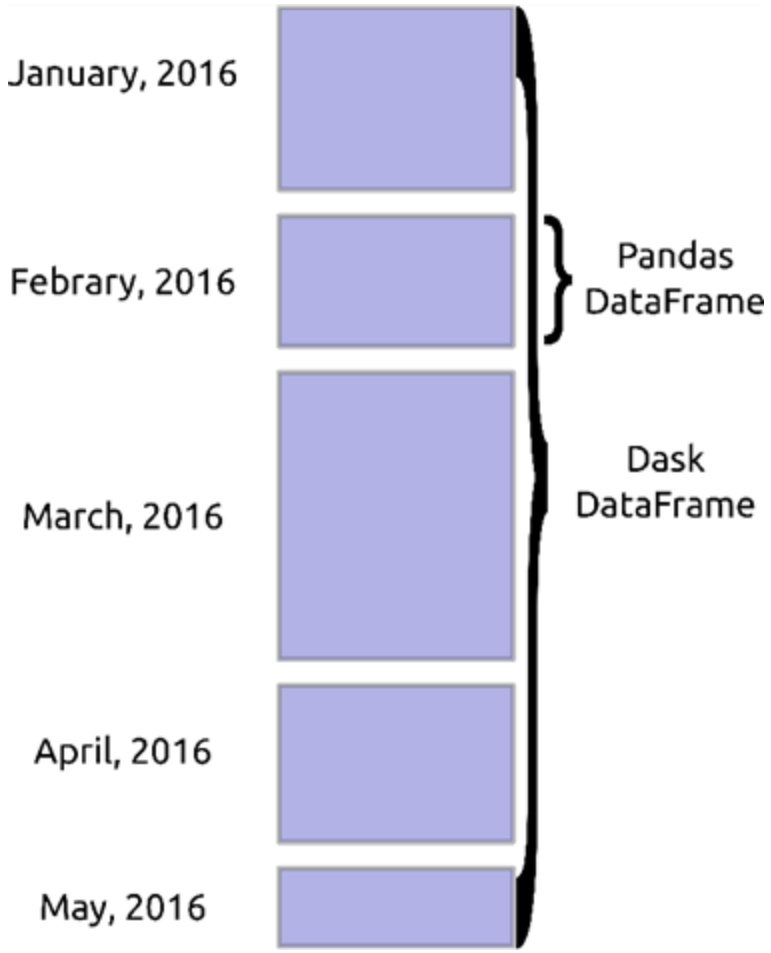
Parallel Pandas
For ETL, time series, data munging

- 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

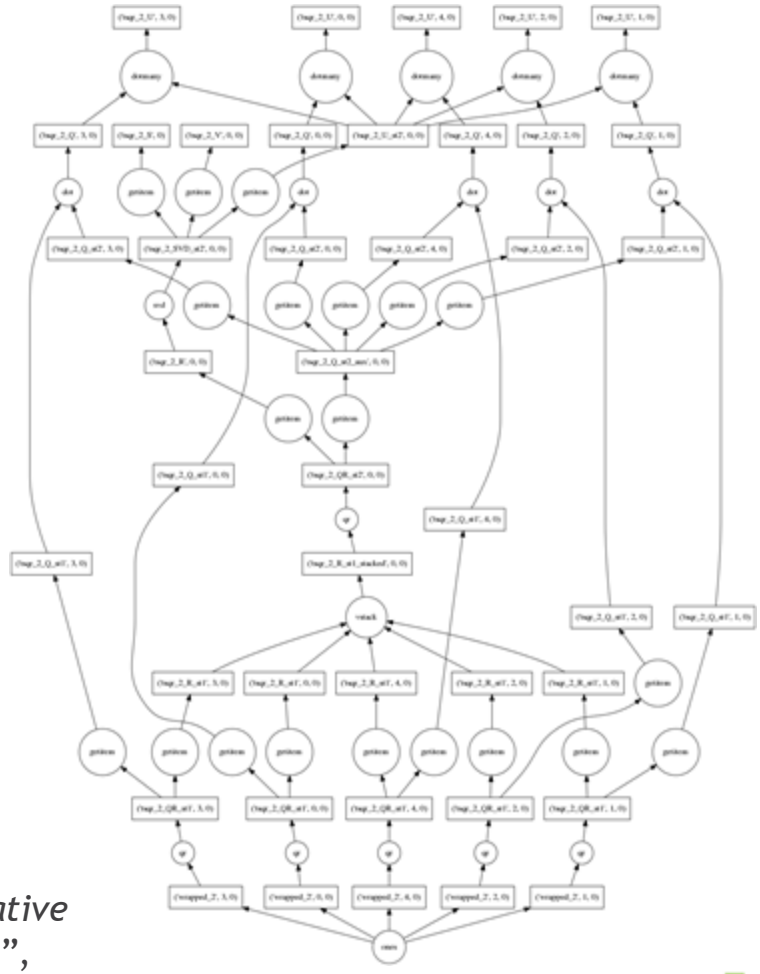
  Either lazily fetched from disk
  Or distributed throughout a cluster



Parallel Python
For custom systems, ML algorithms, workflow engines

- Parallelize existing codebases

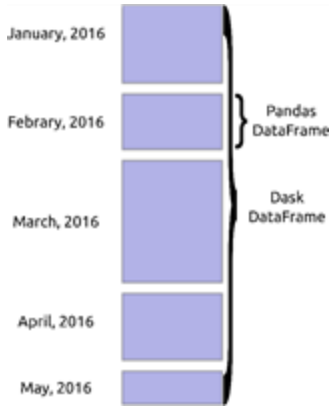
```
f = dask.delayed(f)
g = dask.delayed(g)

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)
```



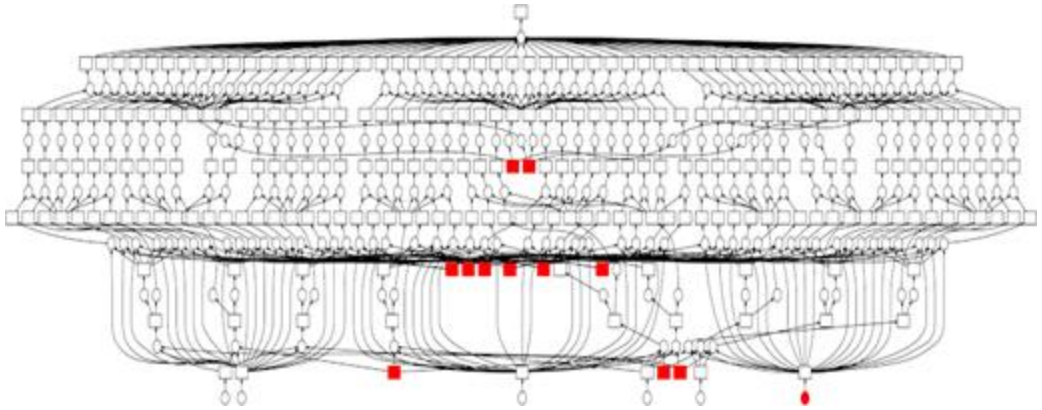
M Tepper, G Sapiro "*Compressed nonnegative matrix factorization is fast and accurate*", IEEE Transactions on Signal Processing, 2016

Dask Connects Python users to Hardware

User



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

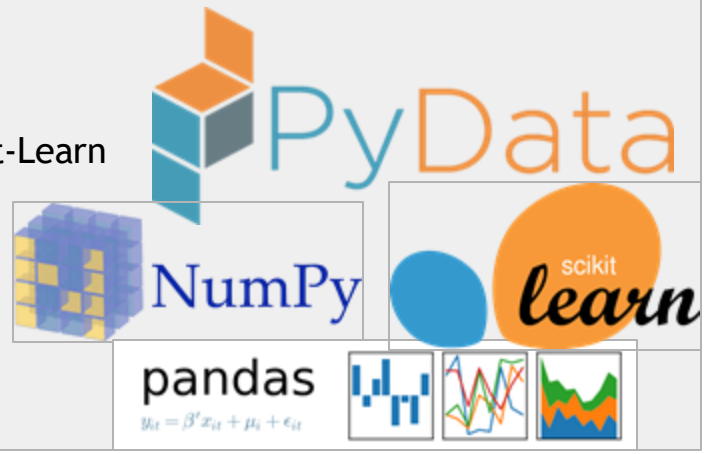
RAPIDS

DASK

PyData

NumPy, Pandas, Scikit-Learn and many more

Single CPU core
In-memory data



Dask

Multi-core and Distributed PyData

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

DASK

Scale out / Parallelize

Accelerated Dask

Just set “cudf” as the backend and use Dask-CUDA Workers

- Configurable Backend and GPU-Aware Workers
- Memory Spilling (GPU->CPU->Disk)
- Optimized Memory Management
- Accelerated RDMA and Networking (UCX)

```
import dask
from dask_cuda import LocalCUDACluster
from dask.distributed import Client
import dask.dataframe as dd

dask.config.set({"dataframe.backend": "cudf"})

cluster = LocalCUDACluster(...)
client = Client(cluster)
```

```
from dask_cuda import LocalCUDACluster
cluster = LocalCUDACluster(...)
cluster
```

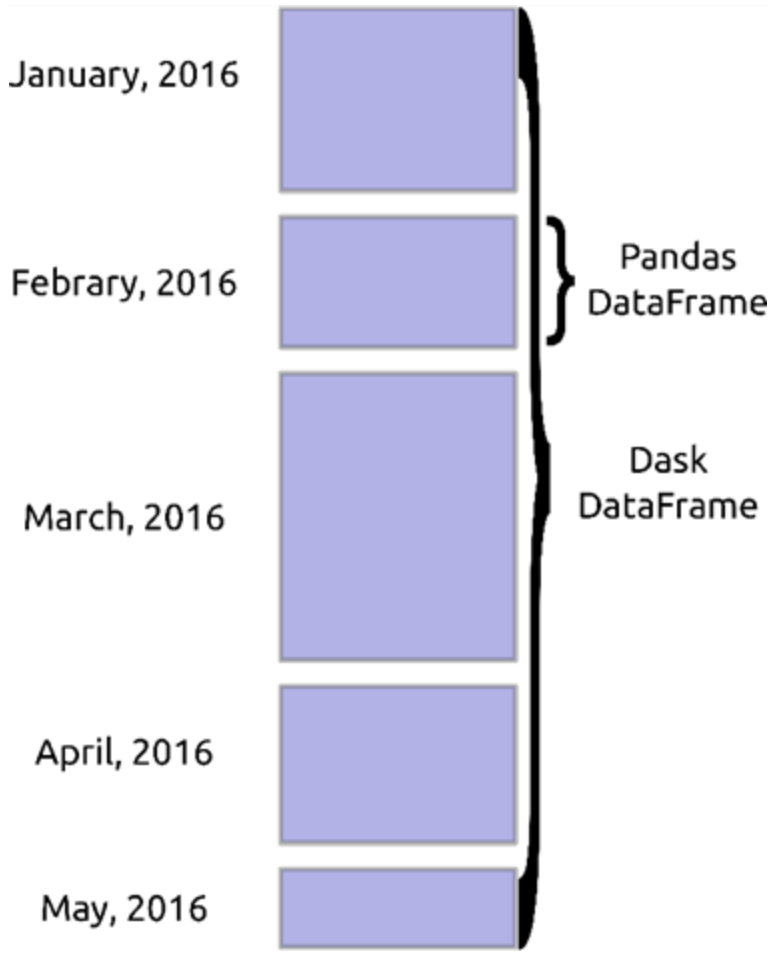
LocalCUDACluster

382454ff

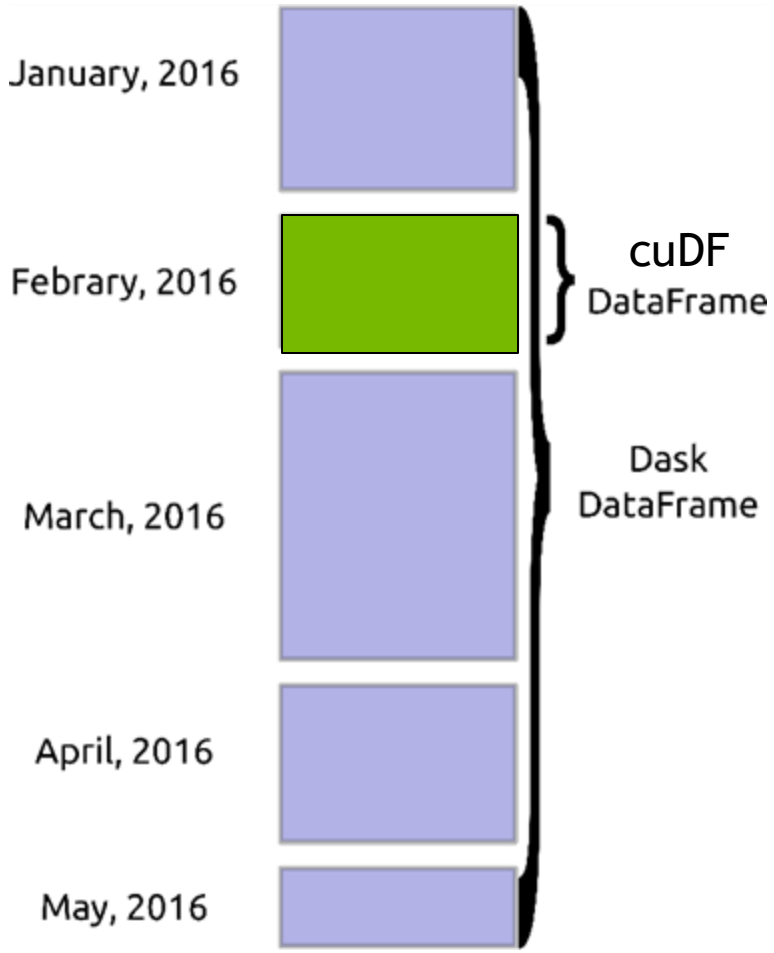
Dashboard: http://127.0.0.1:8787/status	**Workers:** 1
Total threads: 1	**Total memory:** 0.98 TiB
Status: running	**Using processes:** True

▸ Scheduler Info

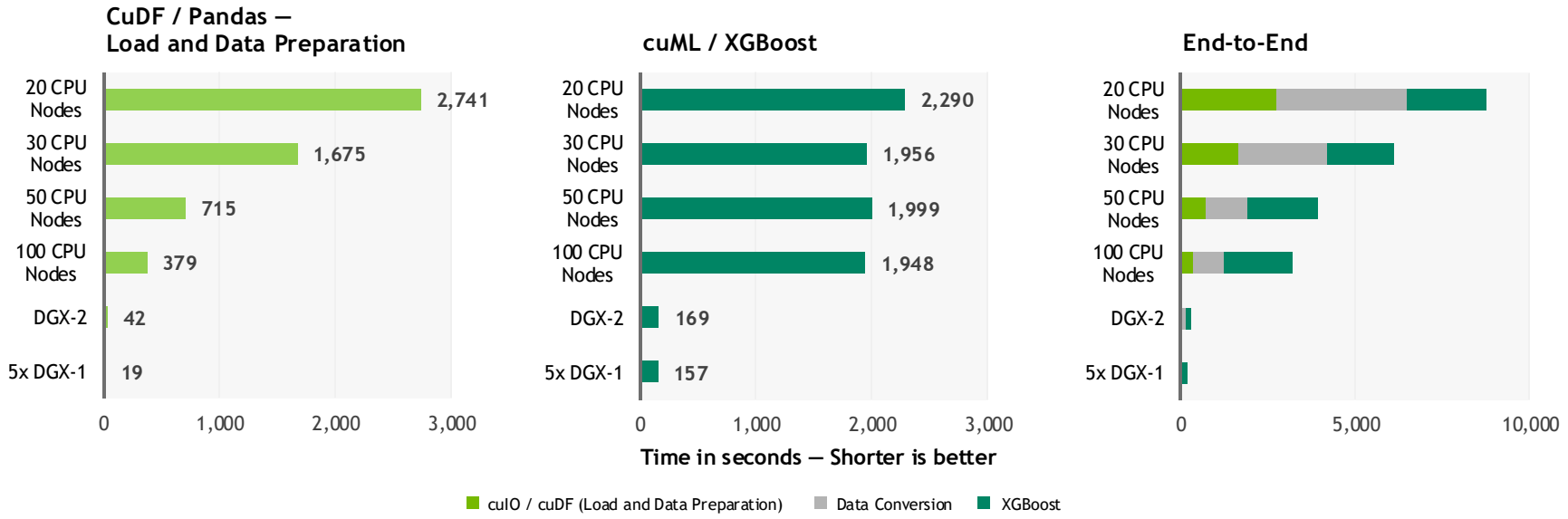
Combine Dask with cuDF
Many GPU DataFrames form a distributed DataFrame



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

Accelerated Apache Spark

Zero code change acceleration for Spark DataFrames and SQL

CPU Spark

```
spark.sql("""
select
    order
    count(*) as order_count
from
    orders"""
)
```

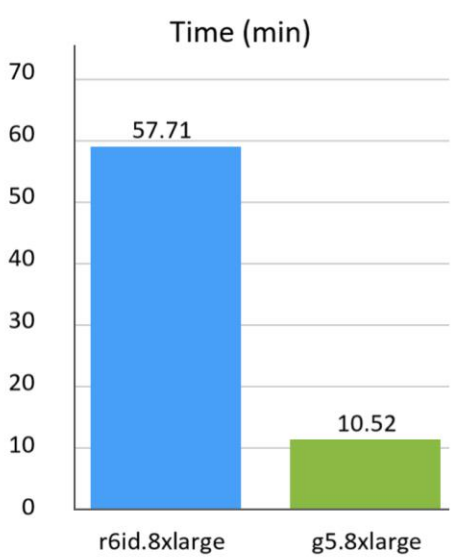
GPU Spark

```
spark.conf.set("spark.plugins",
"com.nvidia.spark.SQLPlugin")

spark.sql("""
select
    order
    count(*) as order_count
from
    orders"""
)
```

Average Speed-Ups: >5x

- RAPIDS operates as a software plugin to the popular Apache Spark platform
- Automatically accelerates supported operations (with CPU fallback if needed)
- Requires no code changes
- Works with Spark standalone, YARN clusters, Kubernetes clusters

NVIDIA Decision Support Benchmark 3TB (Public Cloud)



5.5x faster

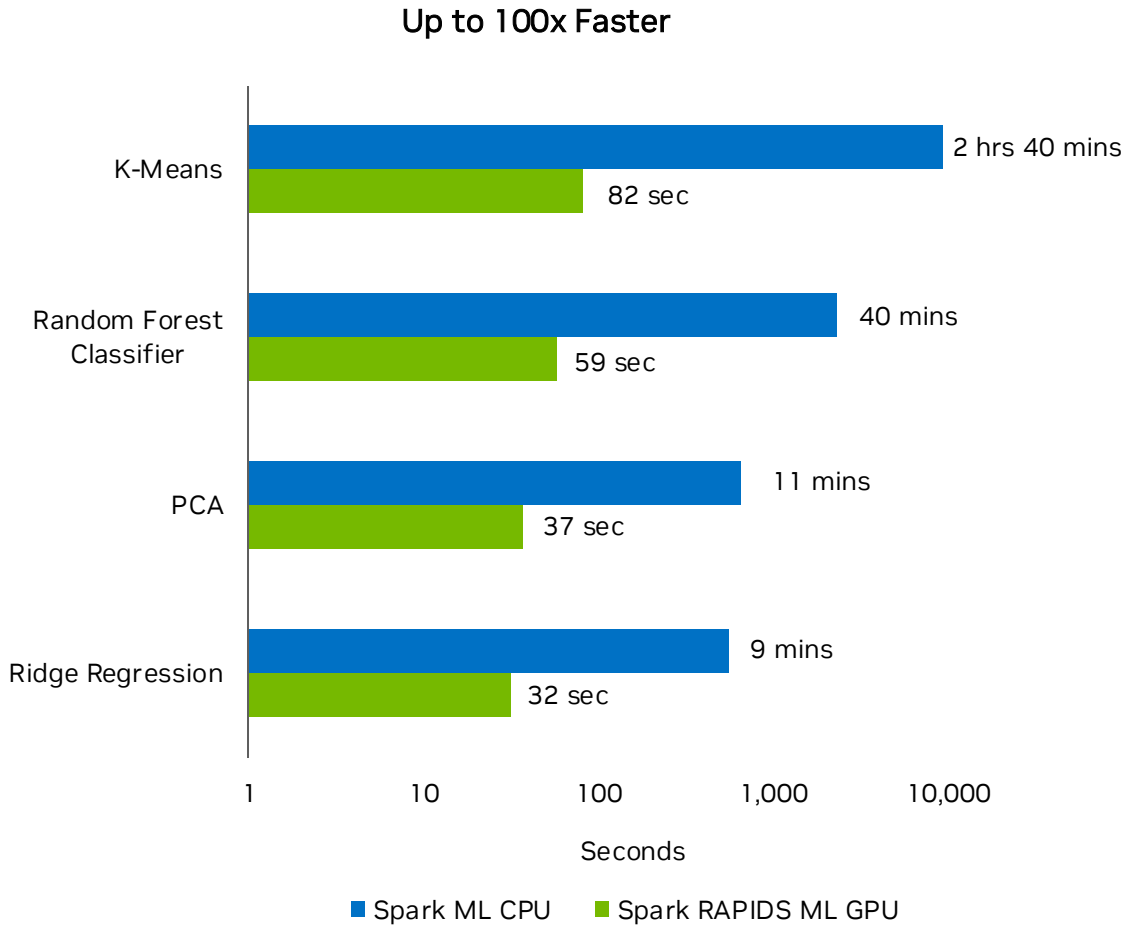
80% cost savings

Apache Spark 3.4.1, RAPIDS Spark release 24.04
See GTC session S62257 for details

Accelerated Apache Spark ML

Bringing GPU-accelerated machine learning to every Apache Spark user

```
from spark_rapids_ml.clustering import Kmeans

kmeans_estm = KMeans()\
.setK(100)\
.setFeaturesCol("features")\
.setMaxIter(30)

kmeans_model = kmeans_estm.fit(pyspark_data_frame)

kmeans_model.write().save("saved-model")

transformed = kmeans_model.transform(pyspark_data_frame)
```



CPU Cluster: Intel Xeon Platinum 8000 series, 32GB RAM
GPU Cluster: NVIDIA A10 24GB, Dataset: 12GB Synthetic Dataset

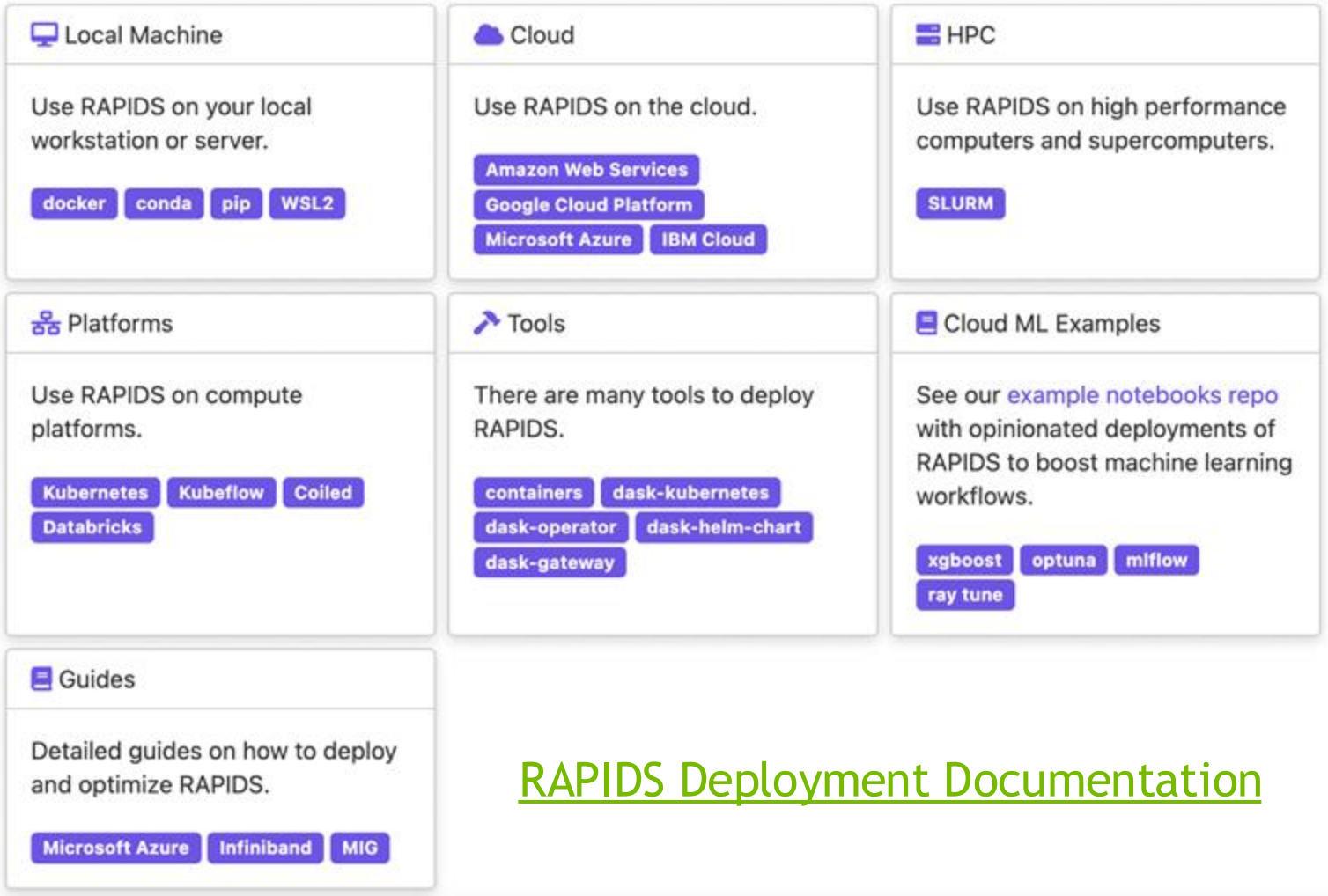
Hybrid CPU/GPU libraries

NVIDIA

Getting Started and Learning More

Deploying RAPIDS

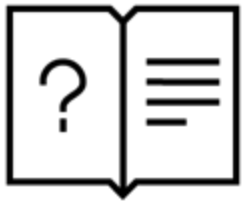
Documentation to get you and up and running RAPIDS anywhere



RAPIDS Deployment Documentation

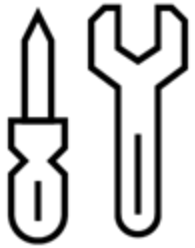
How to Get Started with RAPIDS

A Variety of Ways to Get Up & Running



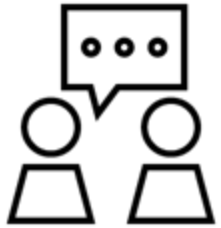
More about RAPIDS

- Learn more at RAPIDS.ai
- Read the API docs
- Check out the RAPIDS blog
- Read the NVIDIA DevBlog



Self-Start Resources

- Get started with RAPIDS
- Deploy on the Cloud today
- Start with Google Colab
- Look at the cheat sheets



Discussion & Support

- Check the RAPIDS GitHub
- Use the NVIDIA Forums
- Reach out on Slack
- Talk to NVIDIA Services

Get Engaged

@RAPIDSai

https://github.com/rapidsai

https://rapids.ai/slack-invite/

https://rapids.ai

NVIDIA