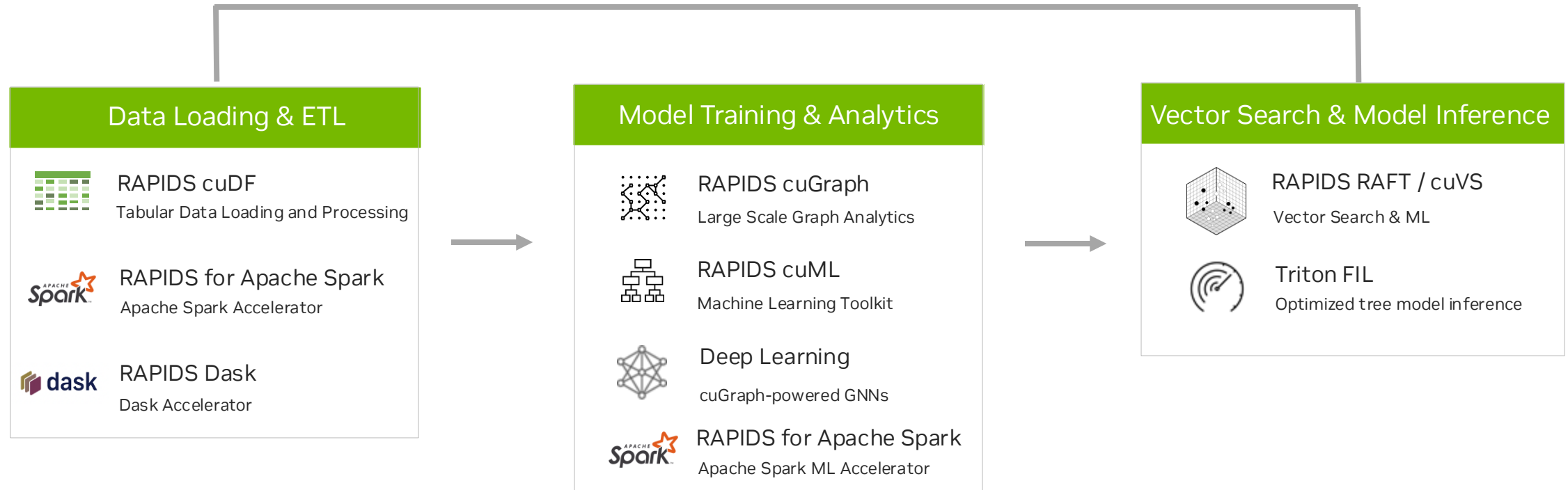




RAPIDS: Accelerated Data Science and Data Processing

Mads R. B. Kristensen, NVIDIA

RAPIDS Accelerates Data Science End-to-End



NVIDIA AI Enterprise

Development Tools | Cloud Native Management and Orchestration | Infrastructure Optimization



Cloud



Data Center

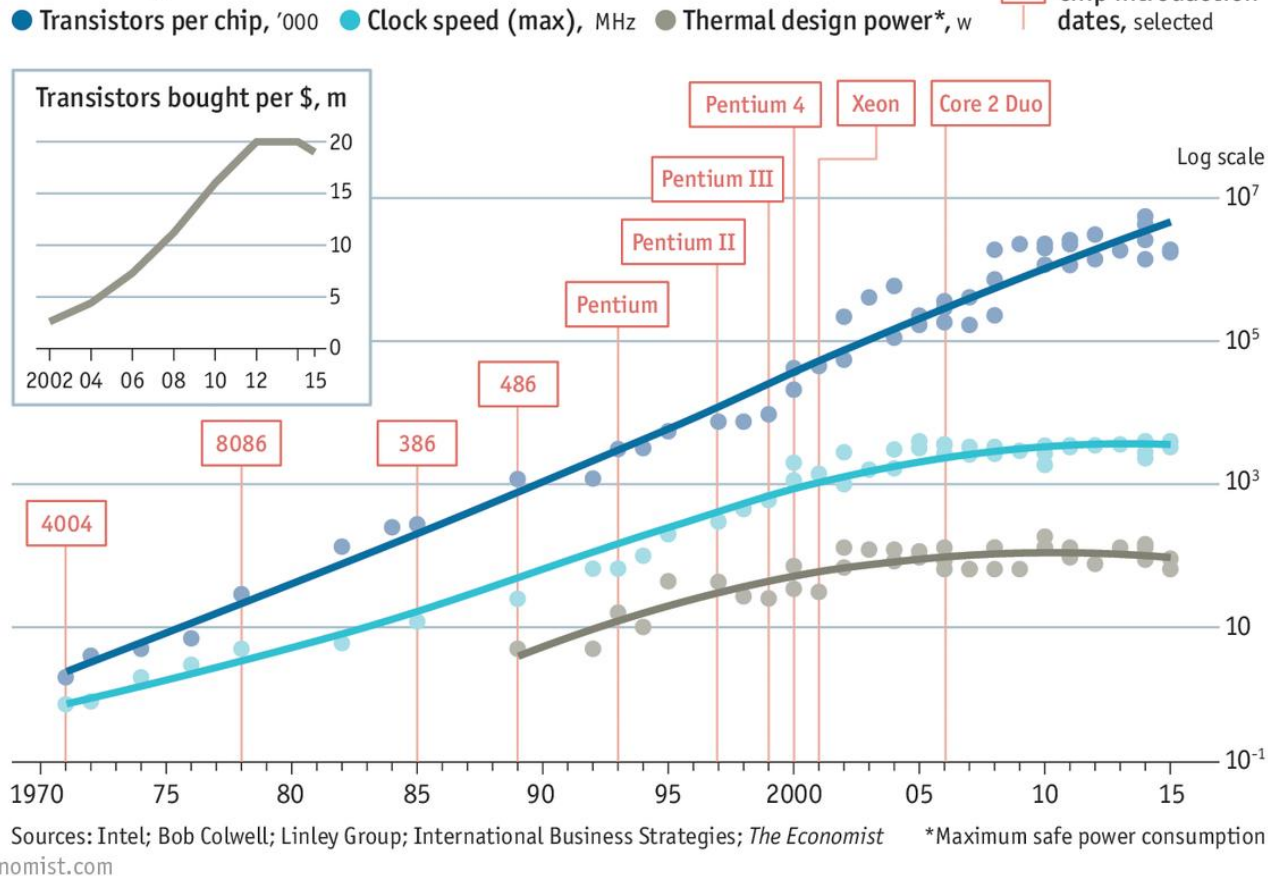


Edge



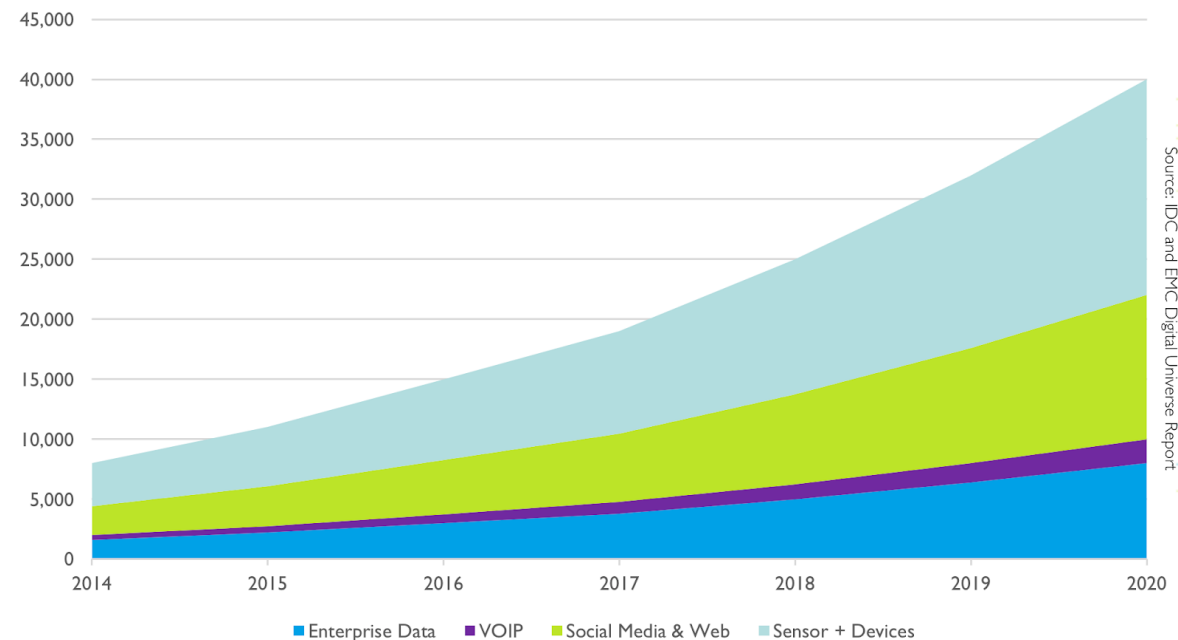
RTX Laptop

Stuttering

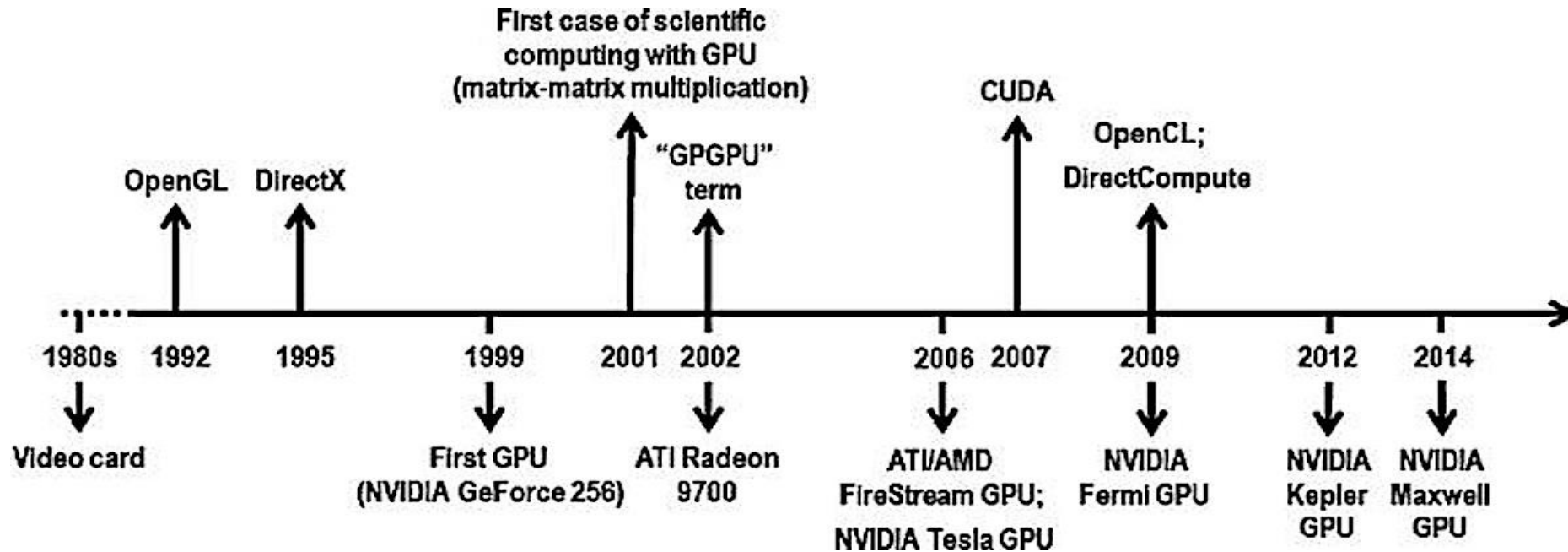


Moore's law: transistors doubles every two years

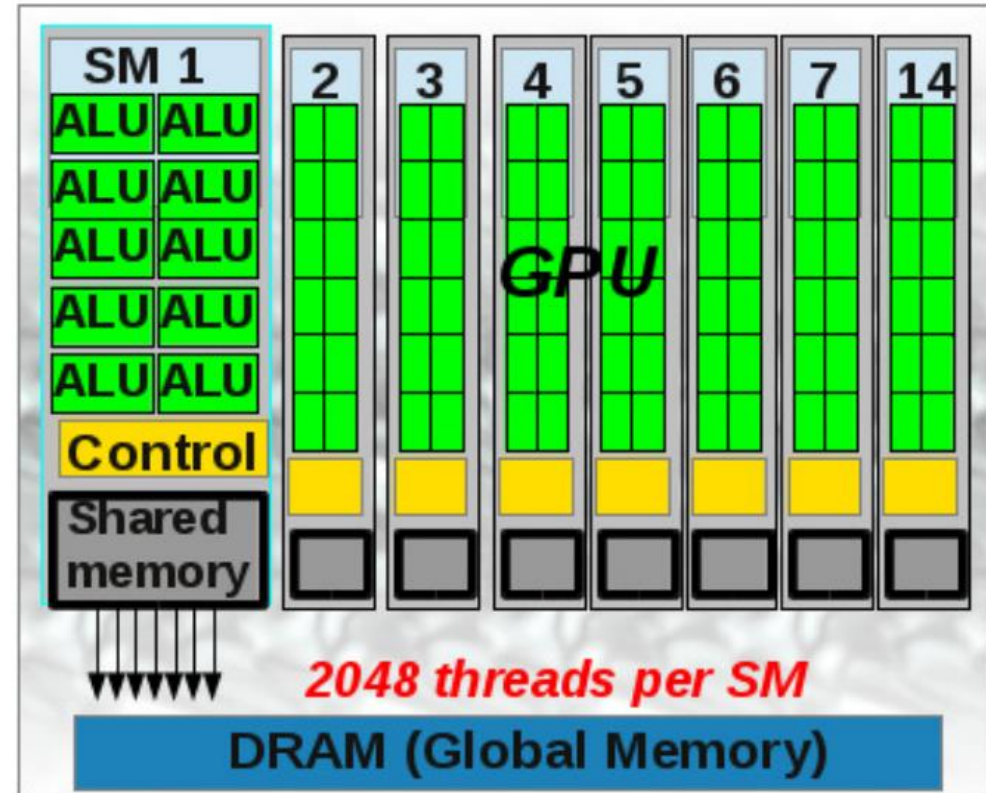
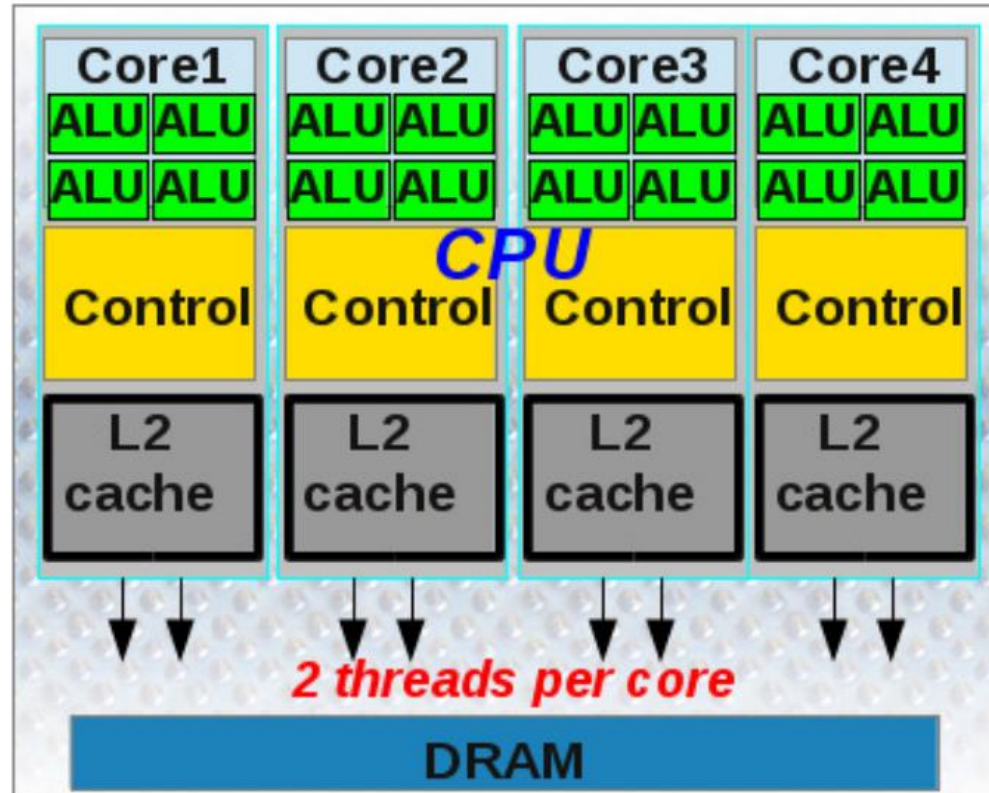
Data Growth and Source in Exabytes



History of the GPU



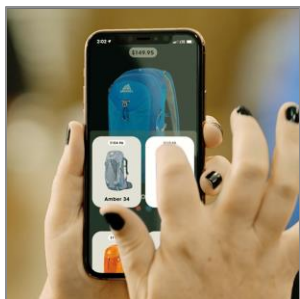
CPU vs GPU



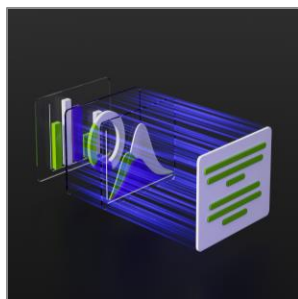
DOI:
10.1016/j.cam.2013.12.032.

Modern Enterprise Applications Need Accelerated Computing

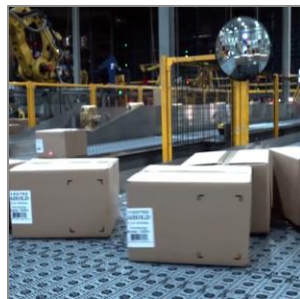
Internet scale data | Massive models | Real-time performance



Recommenders



LLMs



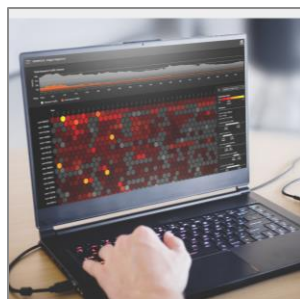
Forecasting



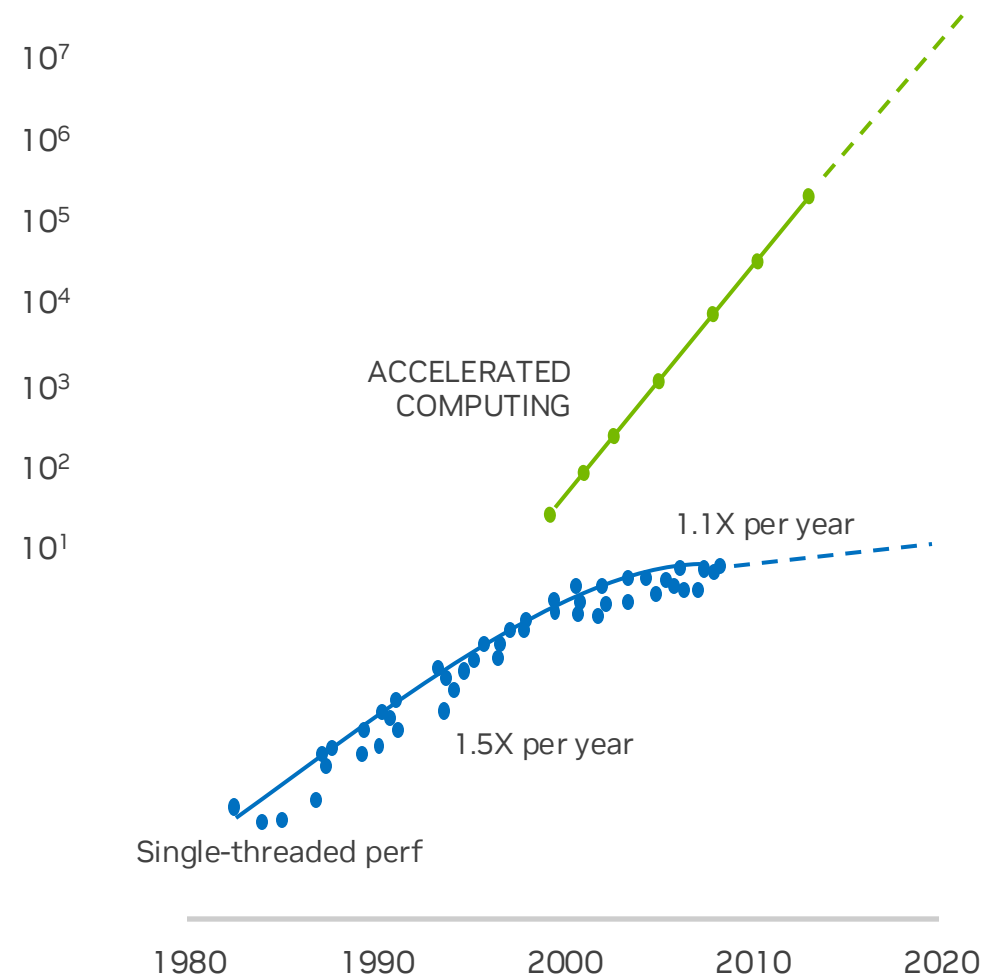
Fraud Detection



Genomic Analysis

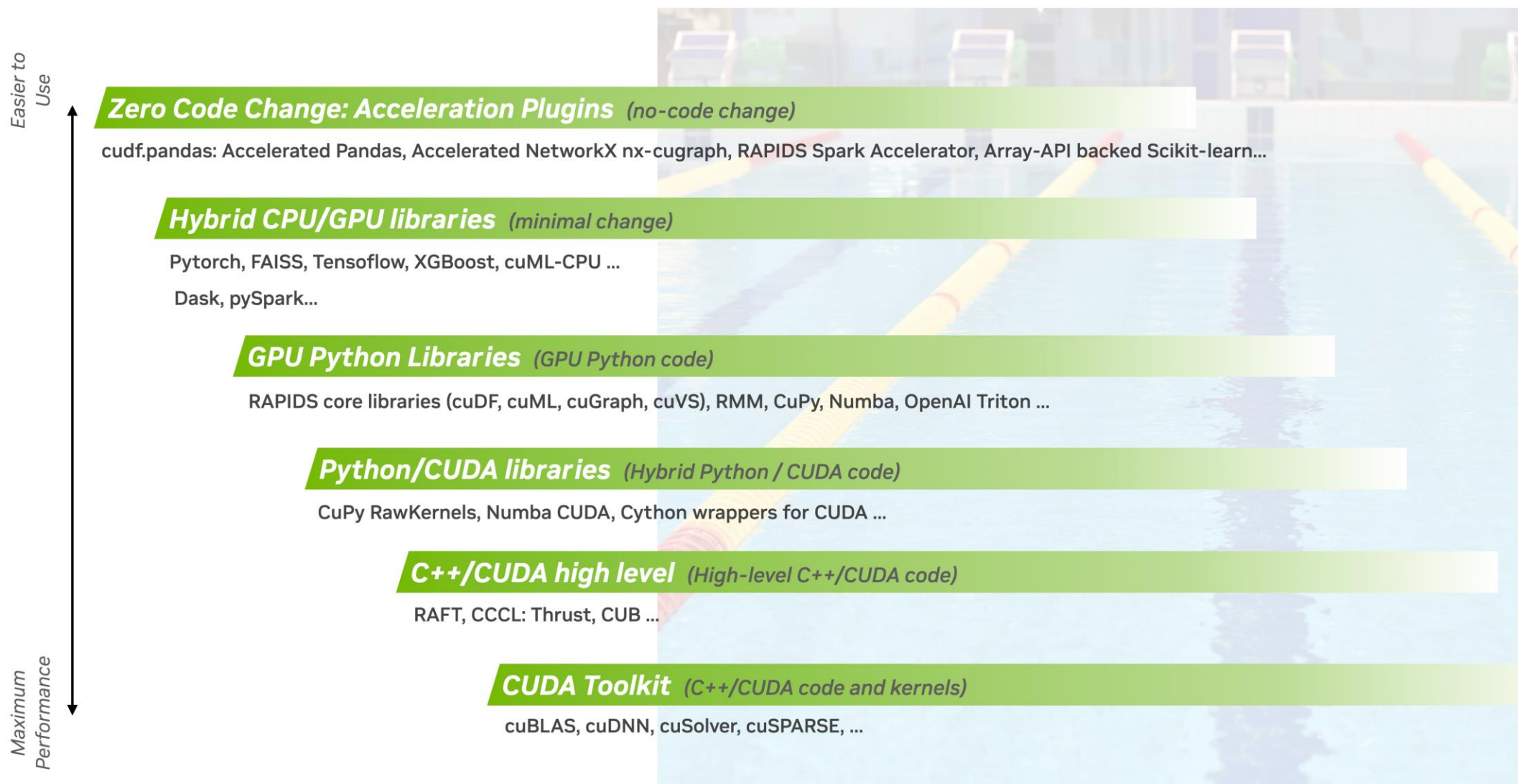


Cybersecurity



Accelerated Computing Swim Lanes

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



RAPIDS ETL

Extract, transform, and load

Pandas

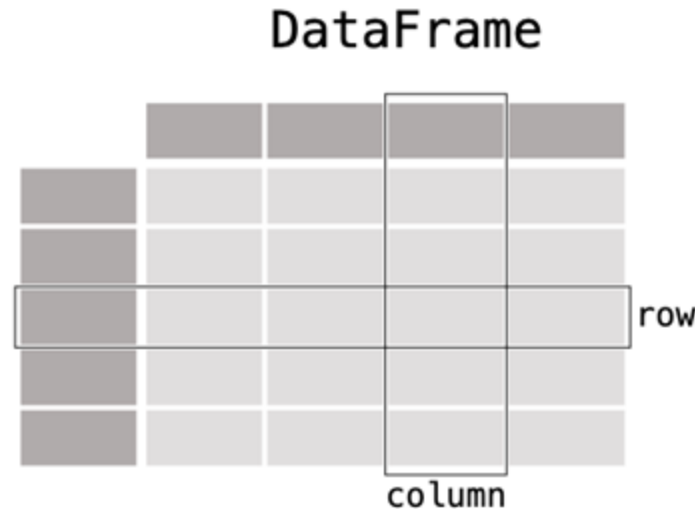
Python's Preeminent DataFrame Library

9.5M

Pandas users

65%

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



```
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

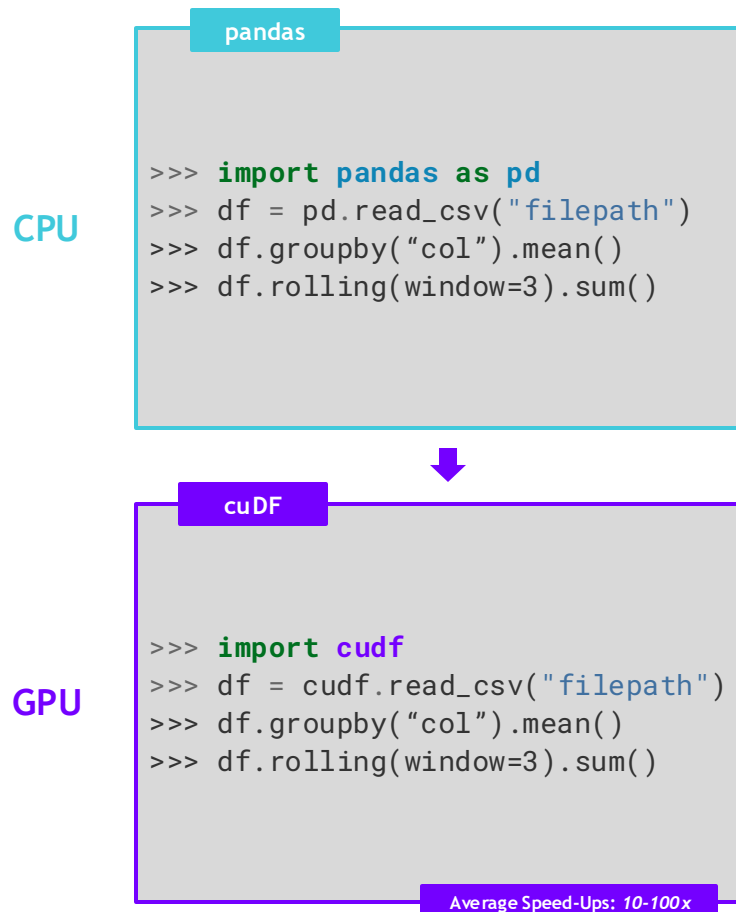
df = pd.read_csv("rows.csv",
                 usecols=[
                     "Registration State",
                     "Violation Description",
                     "Vehicle Body Type"
                 ])
(df[["Registration State", "Violation Description"]]
 .value_counts() # count of offences per state 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

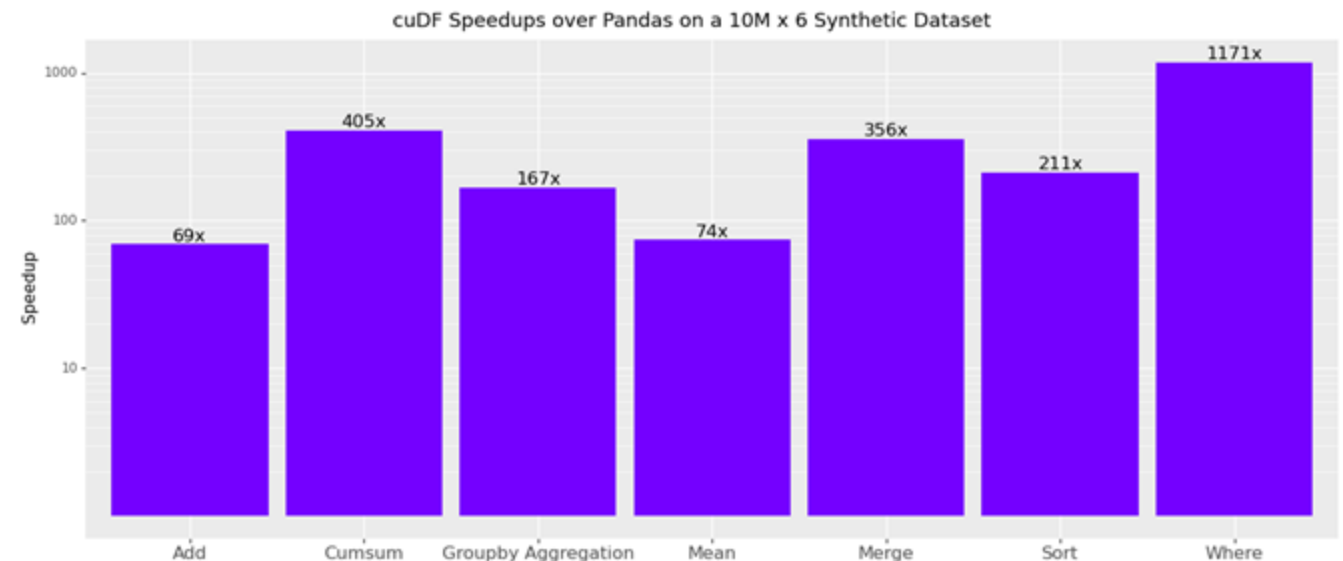
df = cudf.read_csv("rows.csv",
                  usecols=[
                      "Registration State",
                      "Violation Description",
                      "Vehicle Body Type"
                  ])
(df[["Registration State", "Violation Description"]]
 .value_counts() # count of offences 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



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

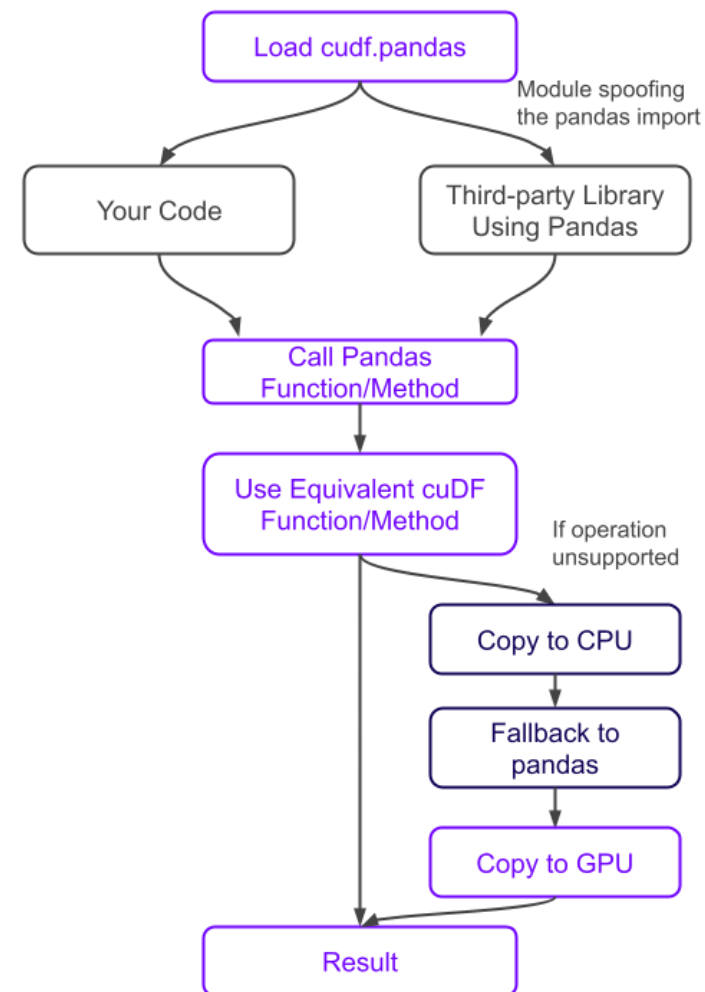
cudf.pandas: the zero code change GPU accelerator for pandas built on cuDF

- 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

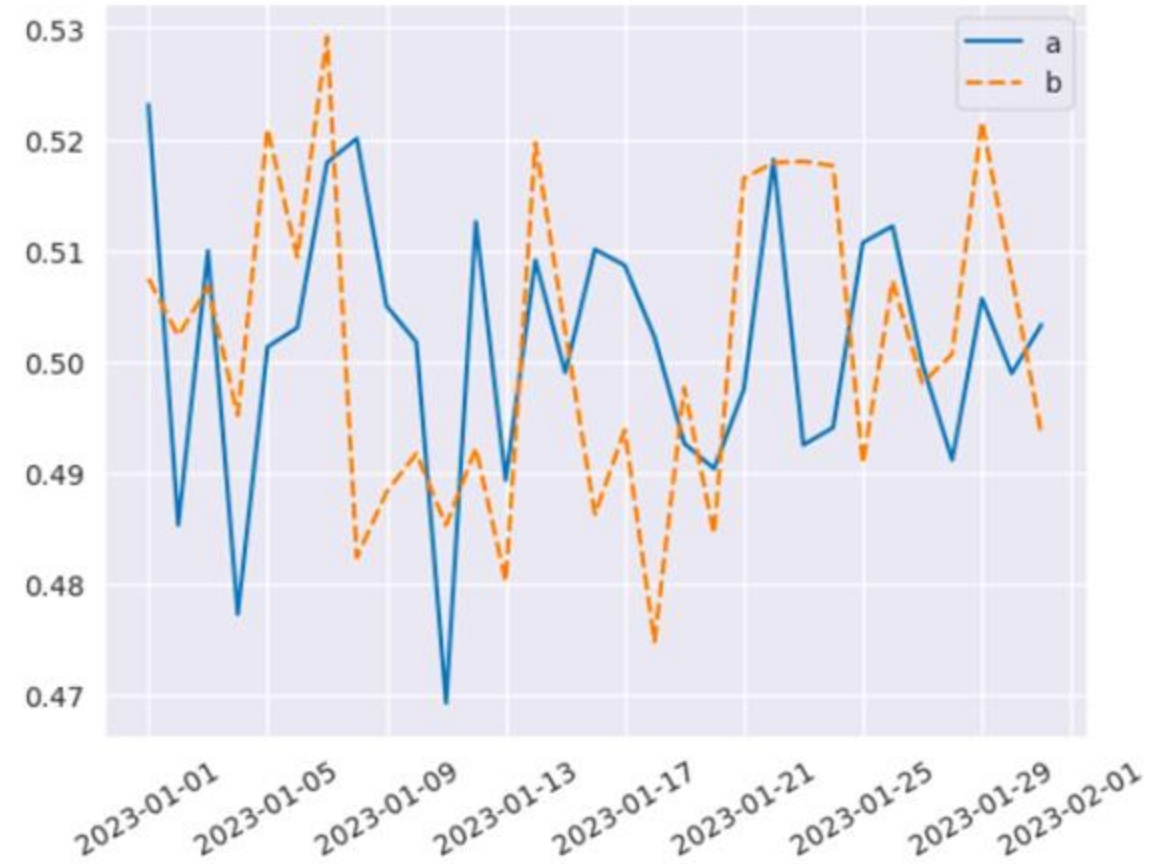
```
%%load_ext cudf.pandas

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

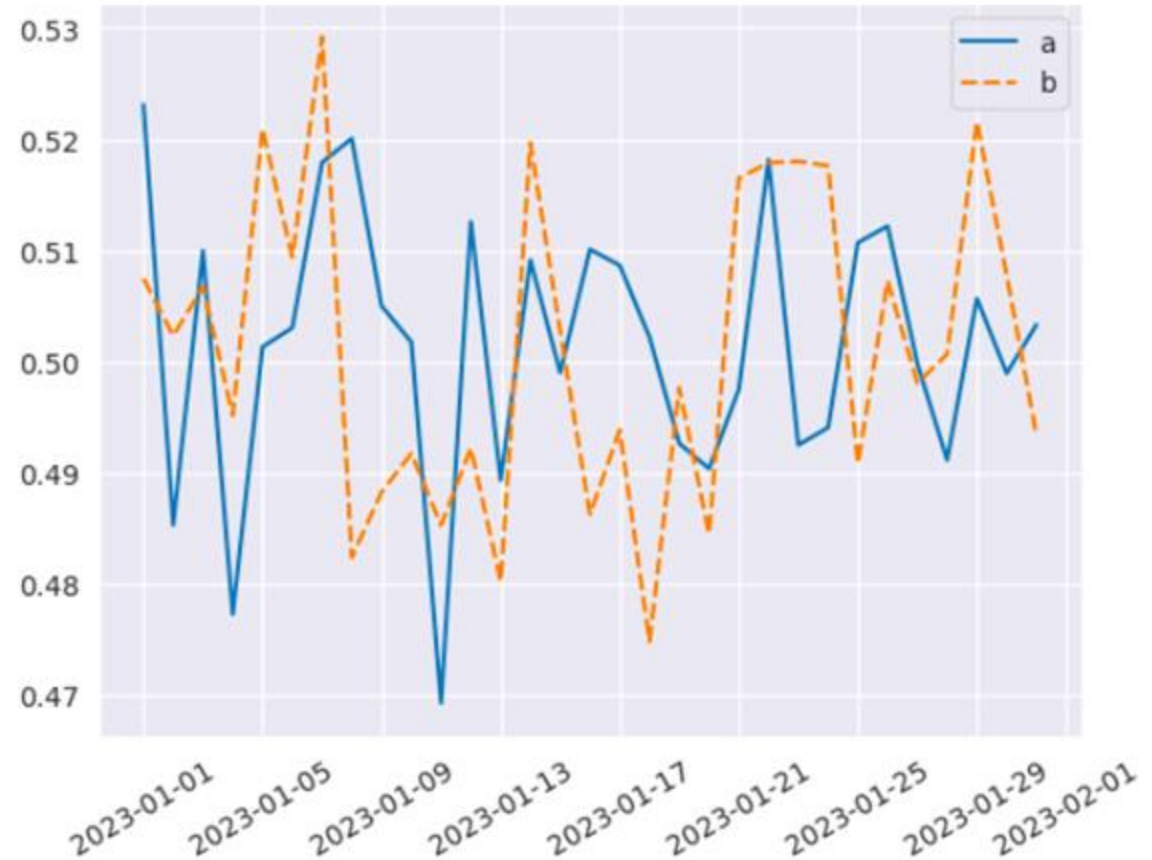
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np
import seaborn as sns

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

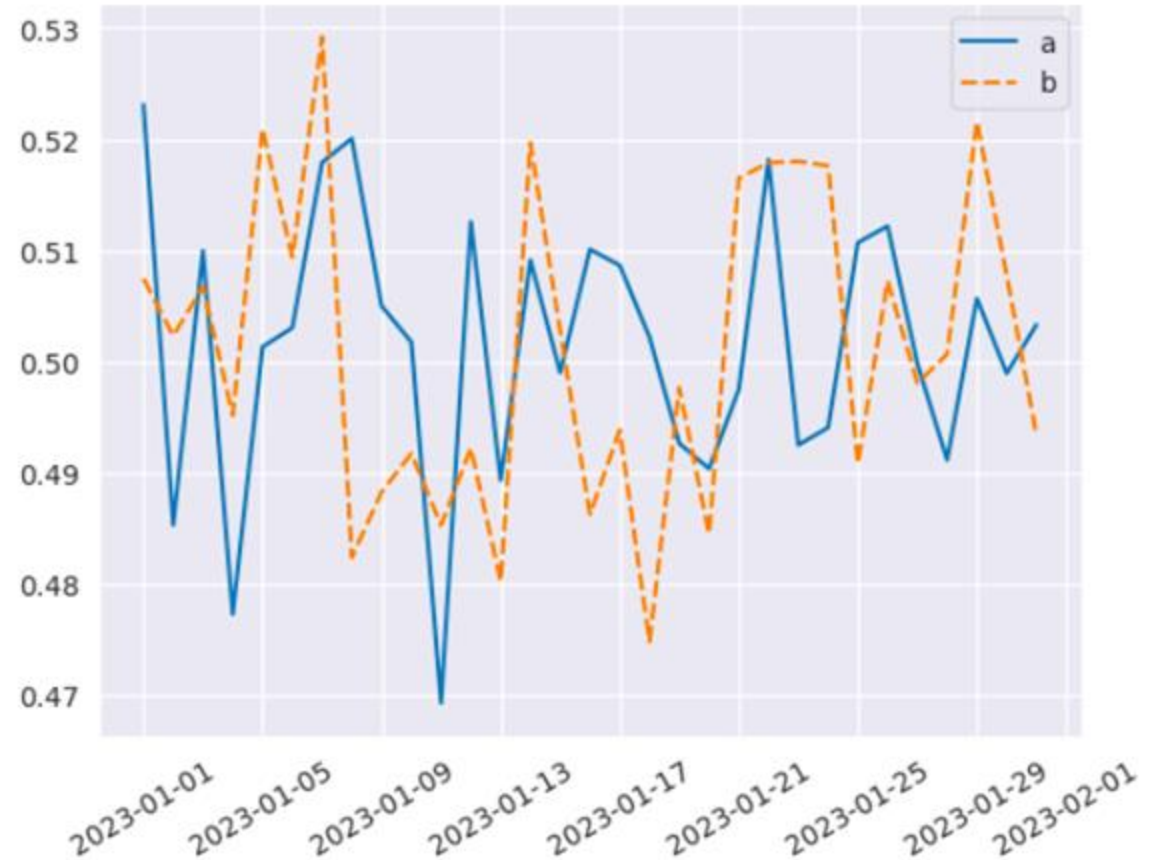
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np
import seaborn as sns

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

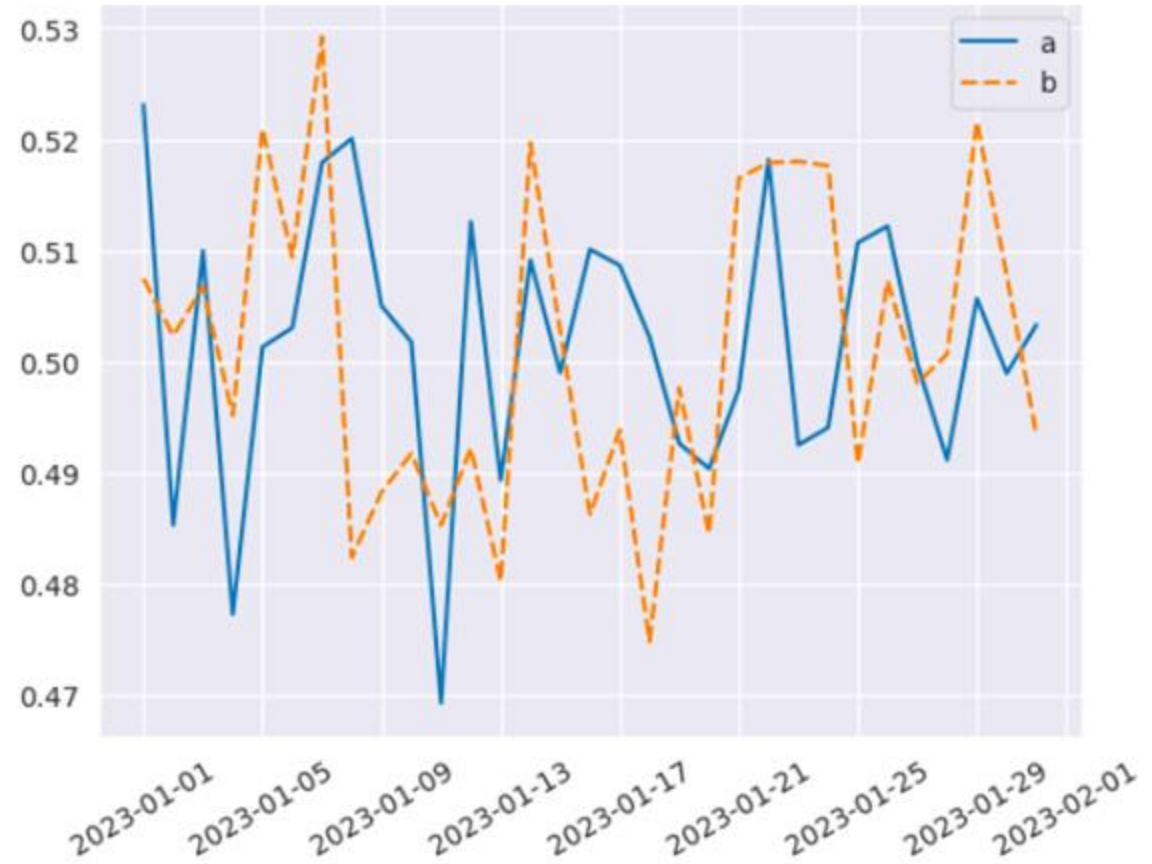
```
%%load_ext cudf.pandas

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

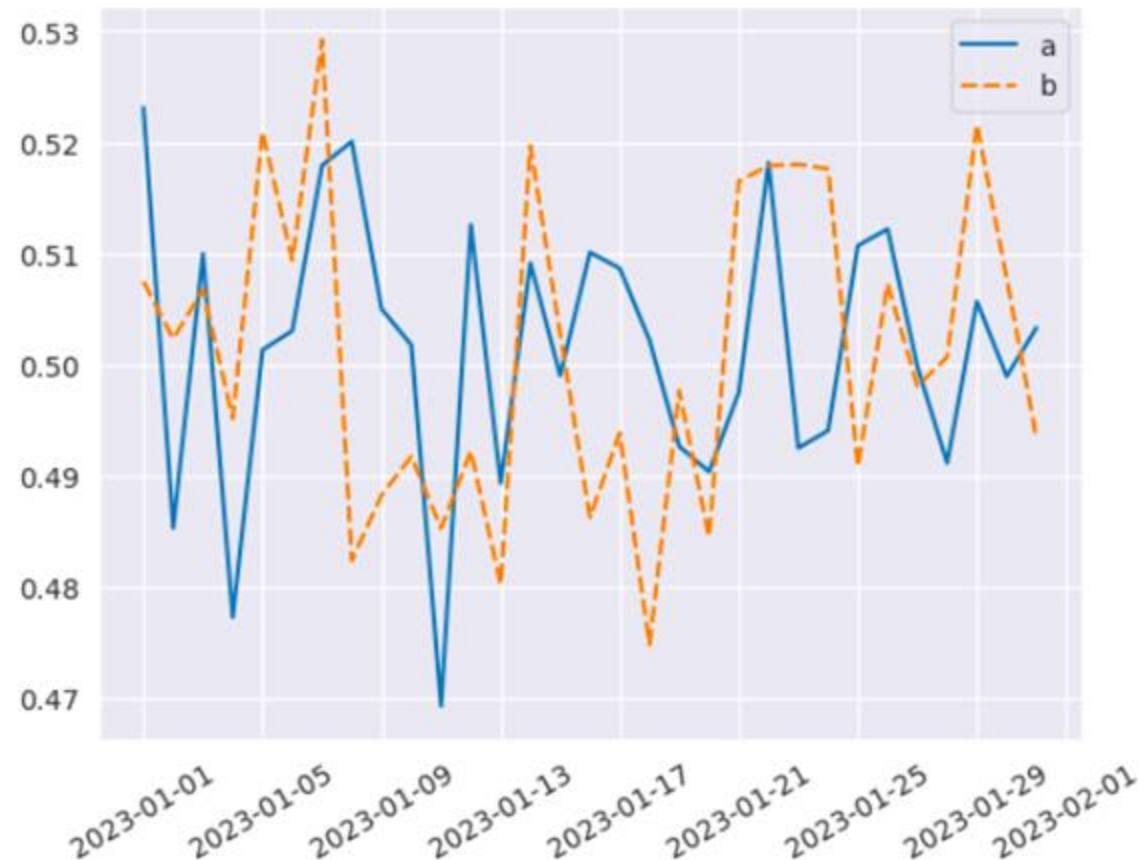
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np
import seaborn as sns

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

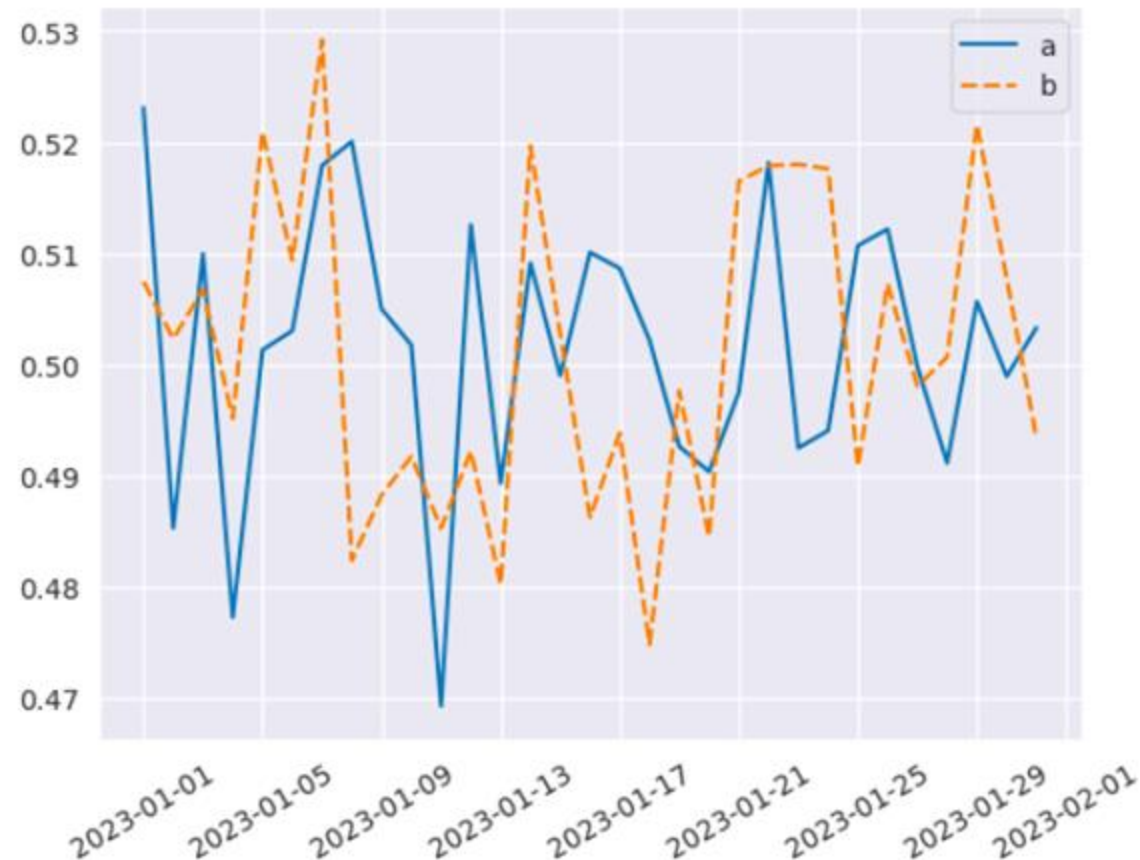
```
%%load_ext cudf.pandas

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

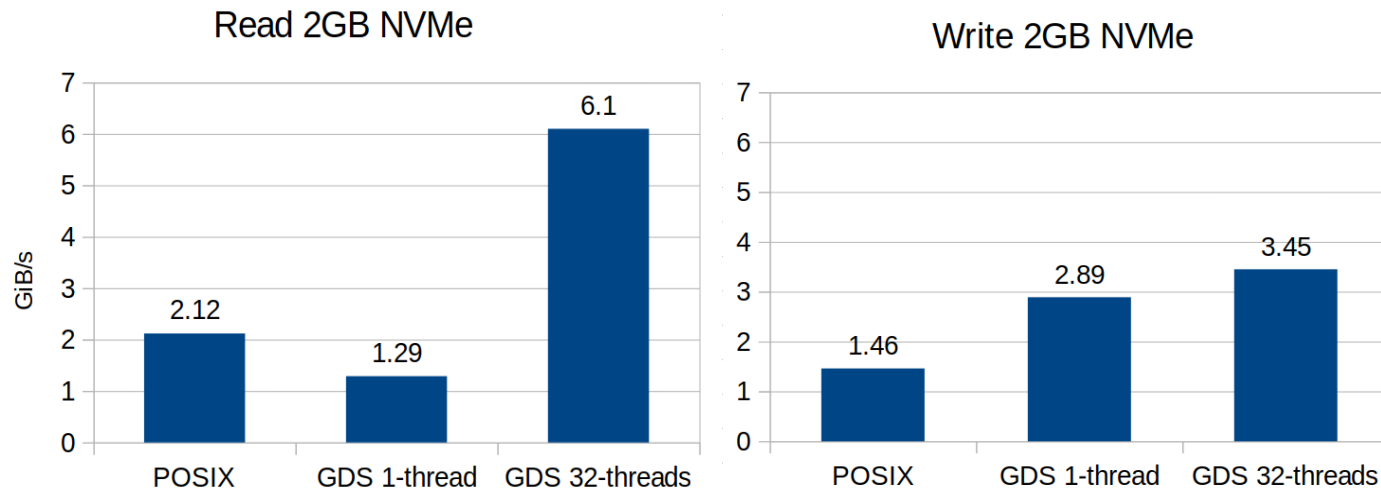
Zero code change: accelerates Pandas “in-place”

RAPIDS KvikIO

KvikIO is a C++ and Python frontend for **cuFile** that provide features such as an object-oriented API, exception handling, RAI 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 (MZWLJ3T8HBL5-00007)

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

```
1 #include <cuda_runtime.h>
2 #include <kvikio/file_handle.hpp>
3 using namespace std;
4
5 int main() {
6     void *a = nullptr;
7     cudaMalloc(&a, 80);
8     // Read file into `a` in parallel using 16 threads
9     kvikio::default_thread_pool::reset(16);
10    {
11        kvikio::FileHandle f("/nvme/input.raw", "r");
12        future<size_t> fut = f.pread(a, sizeof(a), 0);
13        size_t read = fut.get(); // Blocking
14        // Note, `f` closes automatically on destruction.
15    }
16 }
```

```
1 # Write CuPy array to disk
2 import cupy
3 import kvikio
4 a = cupy.arange(10)
5 with kvikio.CuFile("/nvme/input.raw", "w") as f:
6     f.write(a)
7
8 # Write same CuPy array to a Zarr store
9 import zarr
10 from kvikio.zarr import GDSStore
11 z = zarr.array(a,
12               compressor=None,
13               store=GDSStore("/nvme/store"),
14               meta_array=cupy.empty(()),
15             )
16 # We can not access the Zarr array `z` as a
17 # regular CuPy array.
18 b = z[:] # Read from disk to GPU seamlessly
```



RAPIDS ML and Graph Analytics

cuML

Accelerated machine learning with a scikit-learn API

50+ GPU-Accelerated Algorithms

CPU

Scikit-learn

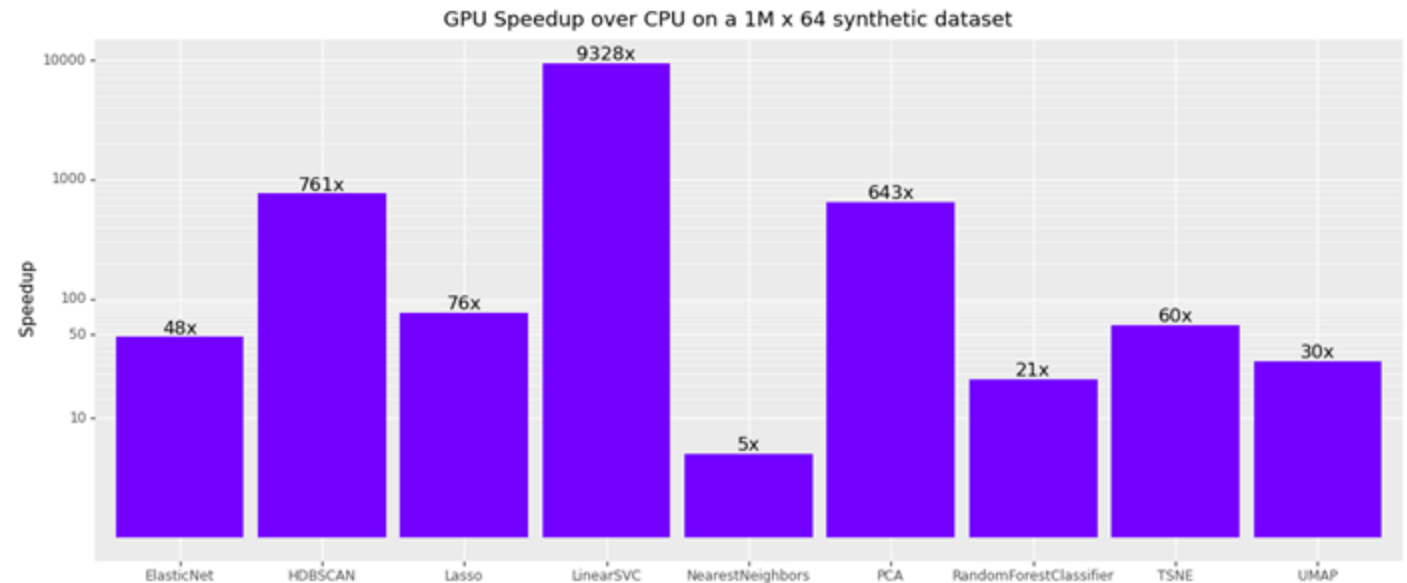
```
>>> from sklearn.ensemble import  
RandomForestClassifier  
>>> clf = RandomForestClassifier()  
>>> clf.fit(x, y)
```



GPU

cuML

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



Time Series

Classification

Regression

Clustering

Preprocessing

Cross Validation

Tree Models

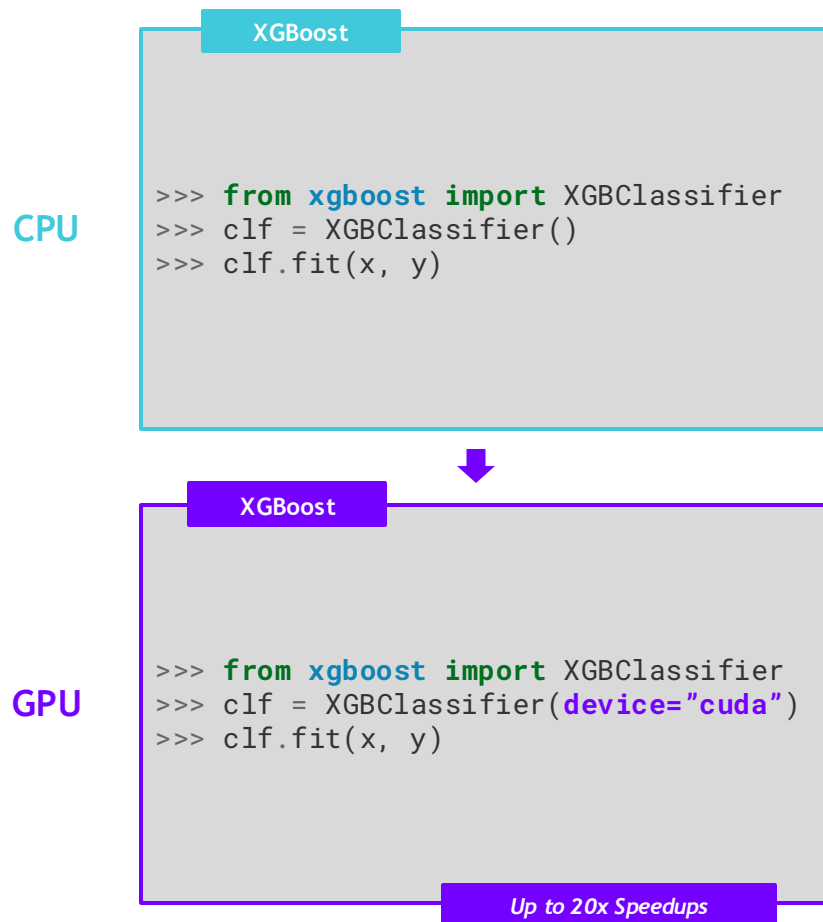
Dimensionality Reduction

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



- 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



Available Algorithms

K-Means

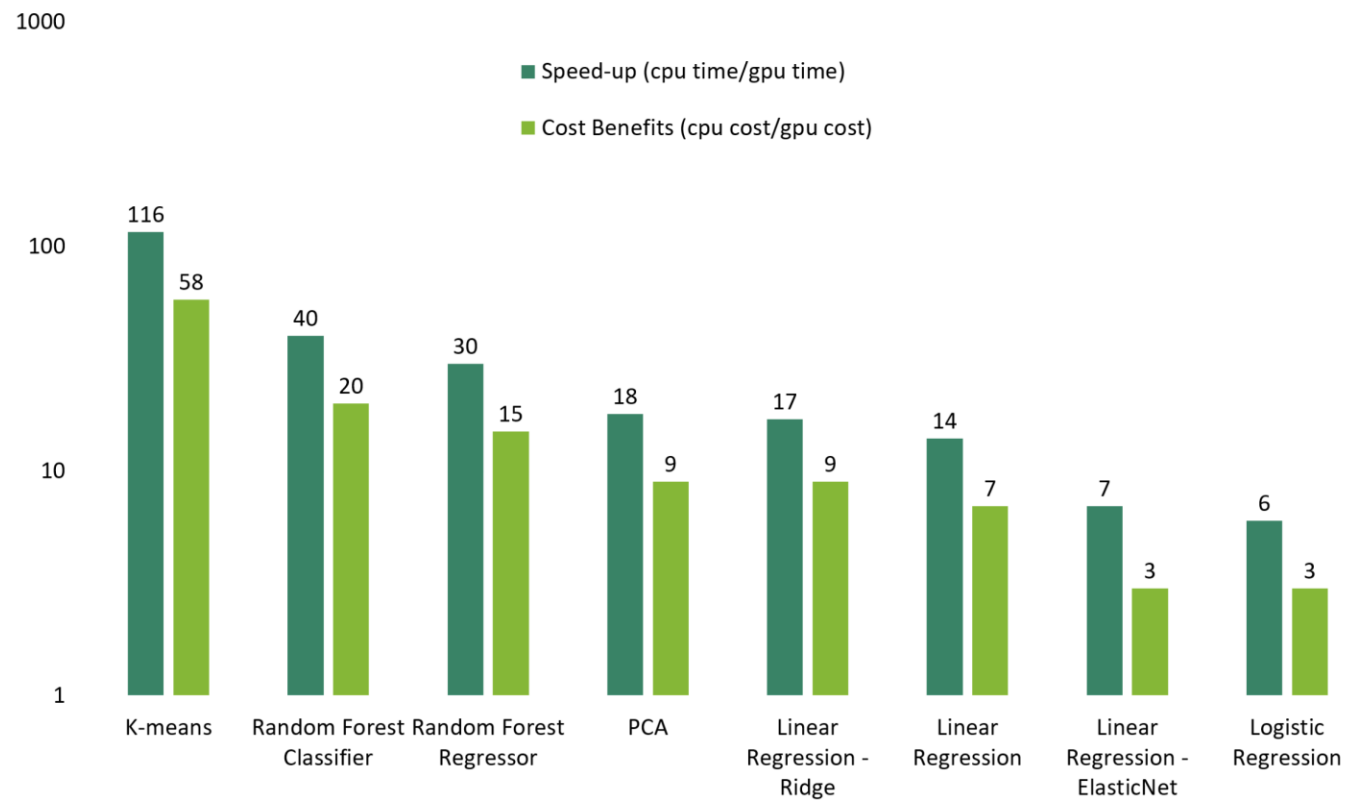
Linear Regression

Logistic Regression¹

PCA

Random Forest Classifier

Random Forest Regressor





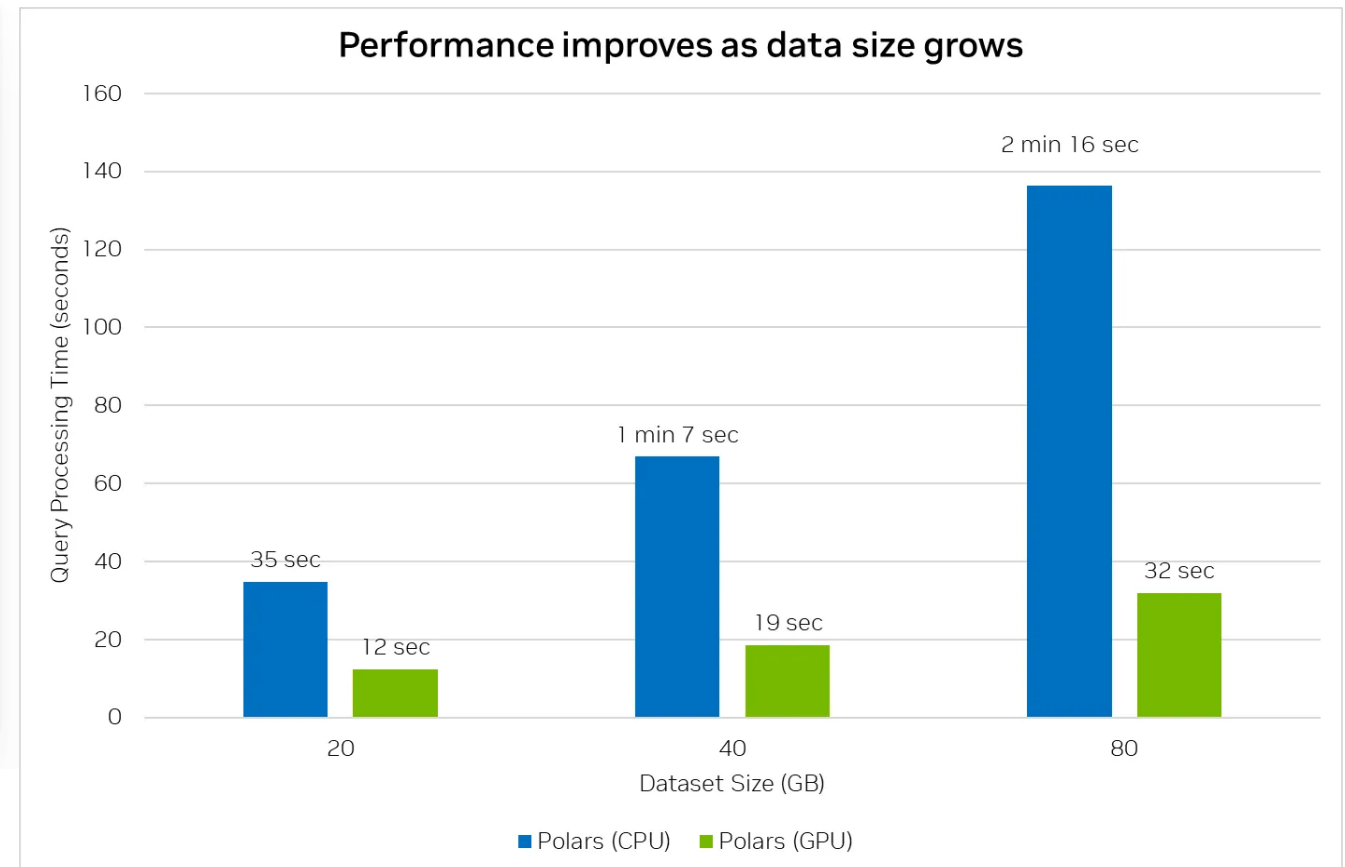
Polars

CuDF - Polars

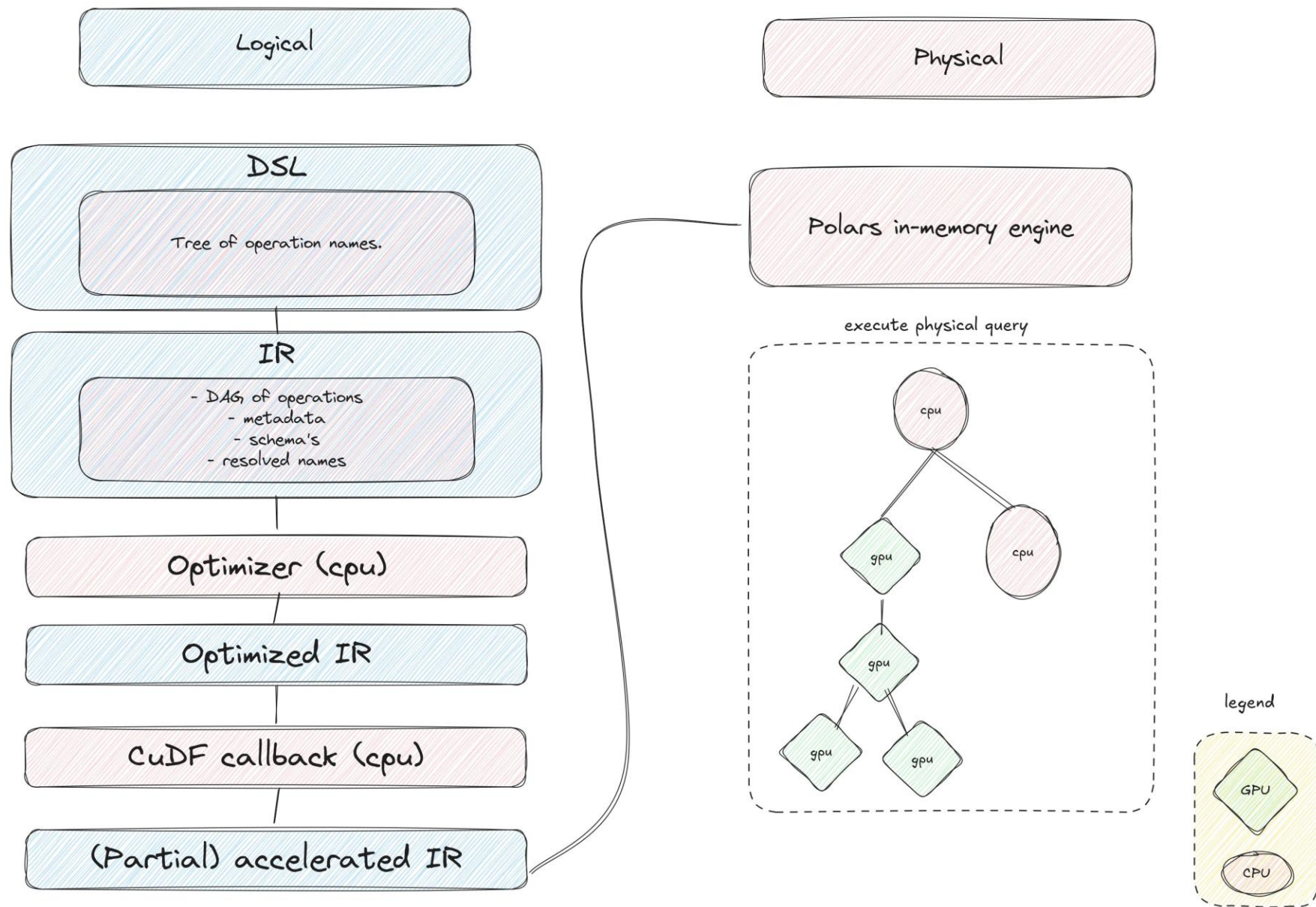
Dataframe library similar to Pandas for Python and Rust

```
1 import polars as pl
2
3 q = (
4     pl.scan_csv("docs/assets/data/iris.csv")
5     .filter(pl.col("sepal_length") > 5)
6     .group_by("species")
7     .agg(pl.all().sum())
8 )
9
10 # Run on the CPU
11 result_cpu = q.collect()
12
13 # Run on the GPU
14 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

RAPIDS and Others

Accelerated on single GPU

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



Dask + RAPIDS

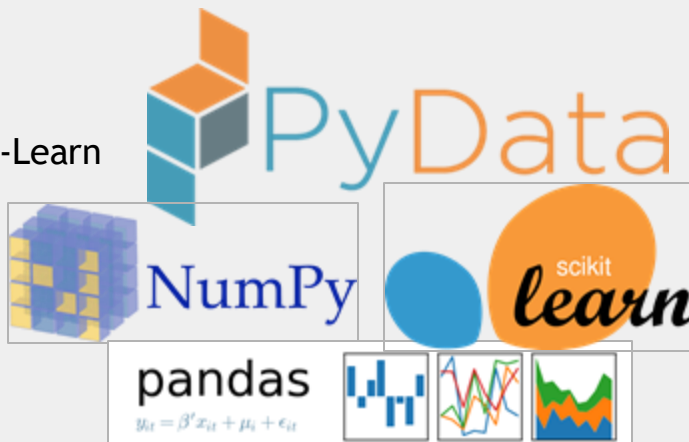
Multi-GPU
On single Node (DGX)
Or across a cluster



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



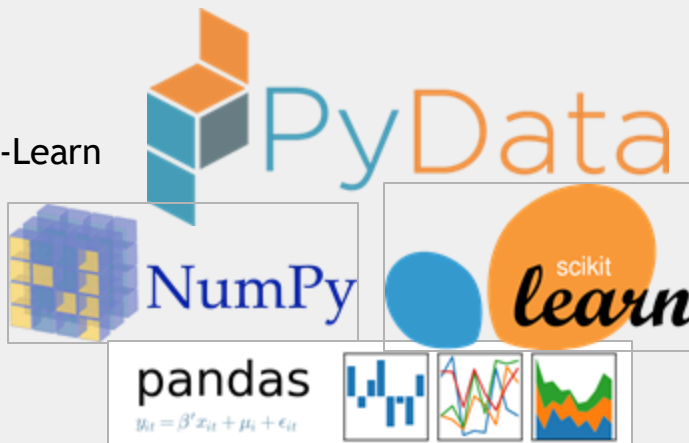
Scale up and out with RAPIDS and Dask

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

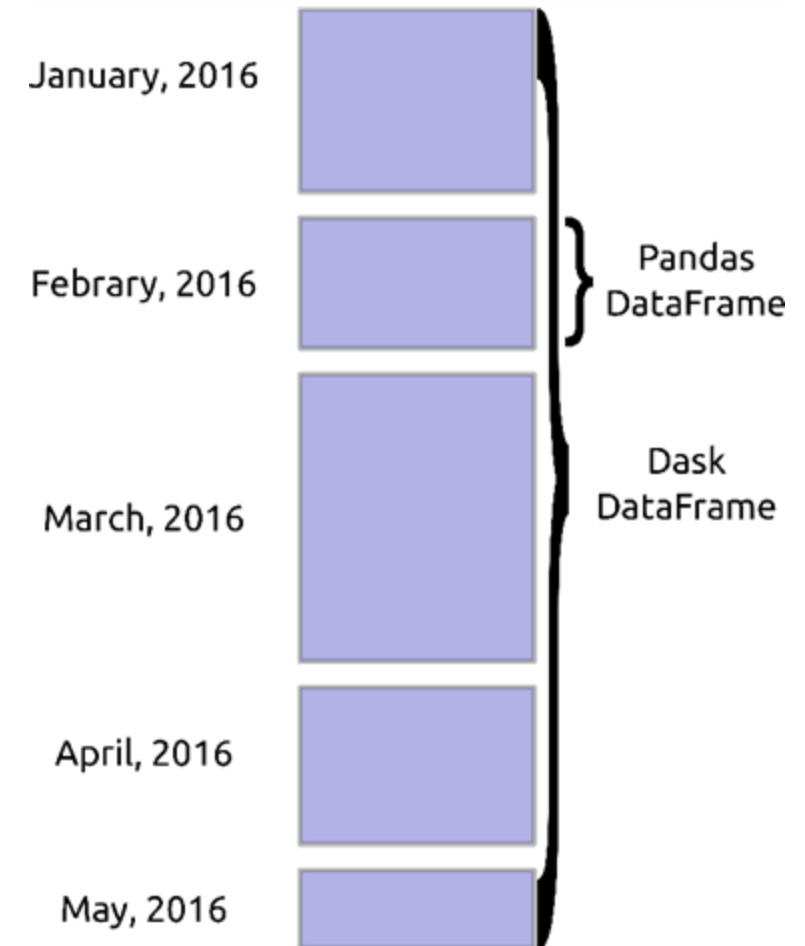
Parallel Pandas
For ETL, time series, data munging

- 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

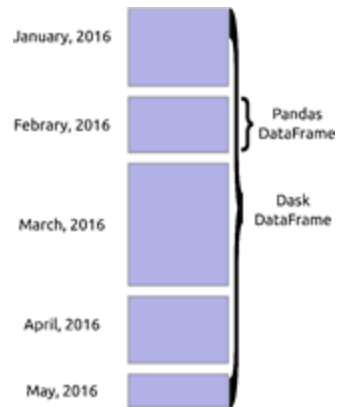
Either lazily fetched from disk
Or distributed throughout a cluster



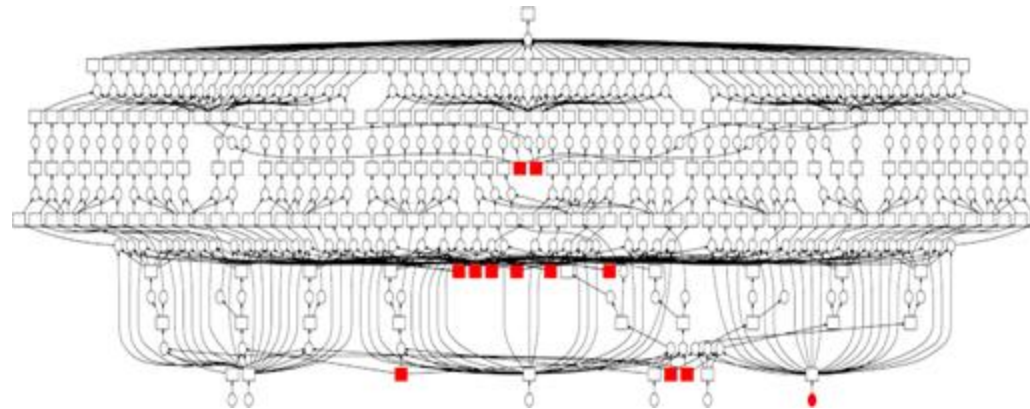
Dask Connects Python users to Hardware



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

RAPIDS and Others

Accelerated on single GPU

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



Dask + RAPIDS

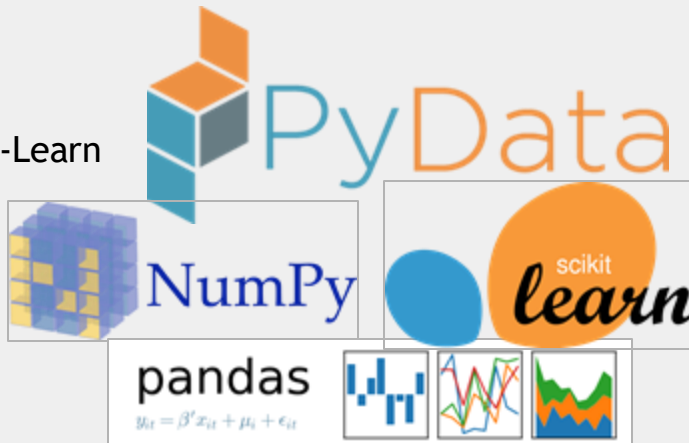
Multi-GPU
On single Node (DGX)
Or across a cluster



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



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

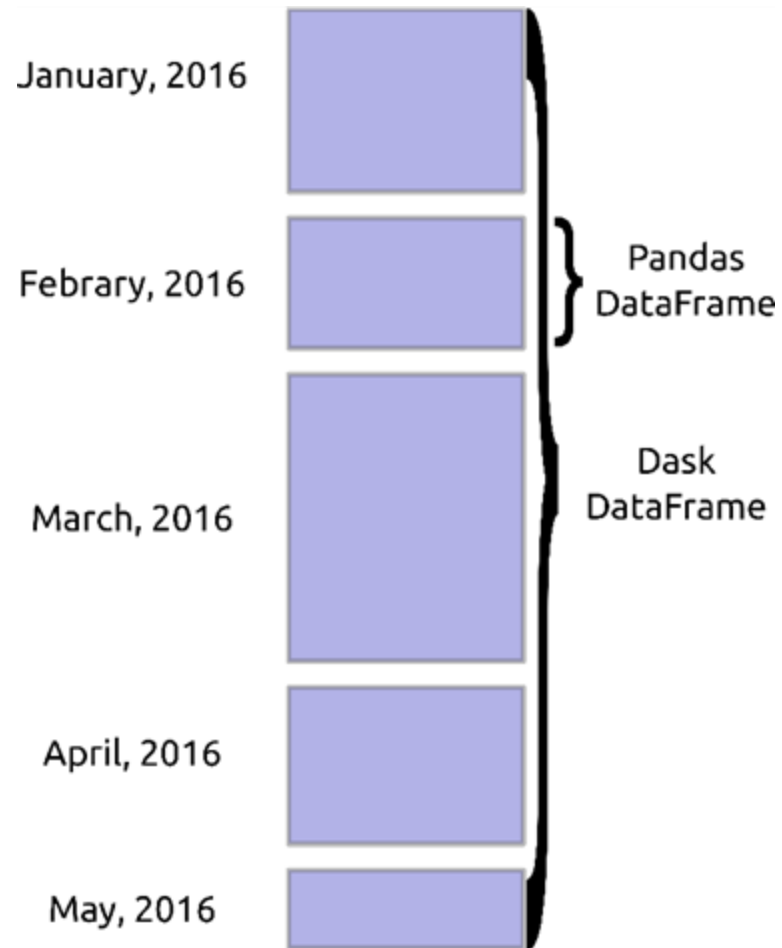
Total memory: 0.98 TiB

Status: running

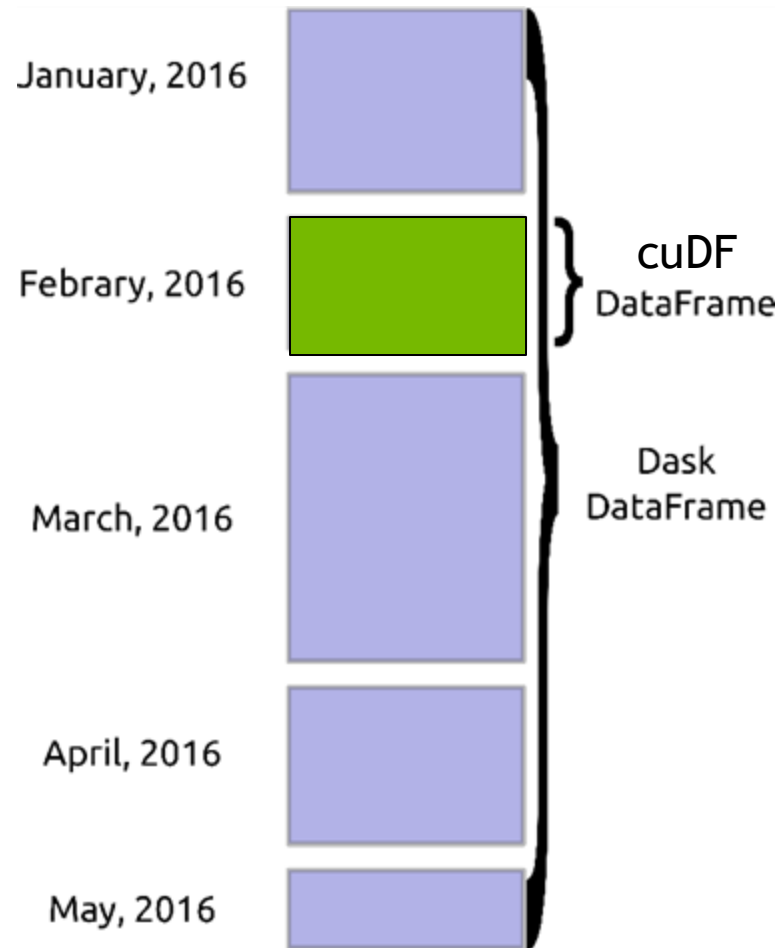
Using processes: True

► Scheduler Info

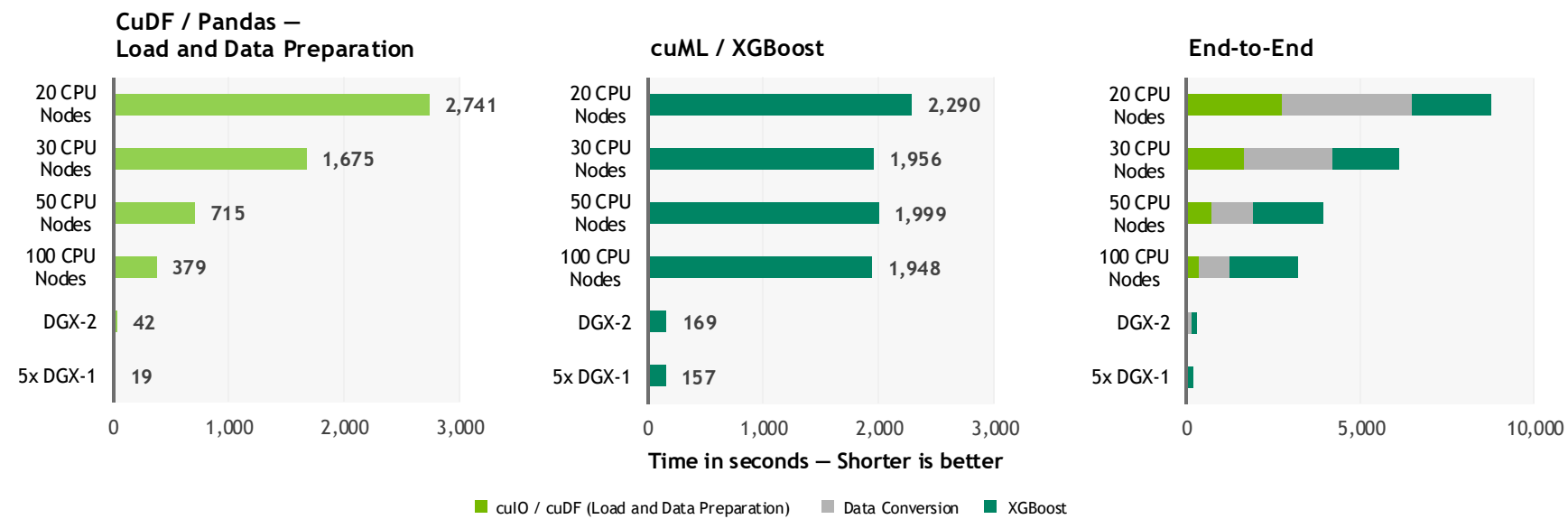
Combine Dask with cuDF
Many GPU DataFrames form a distributed DataFrame



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

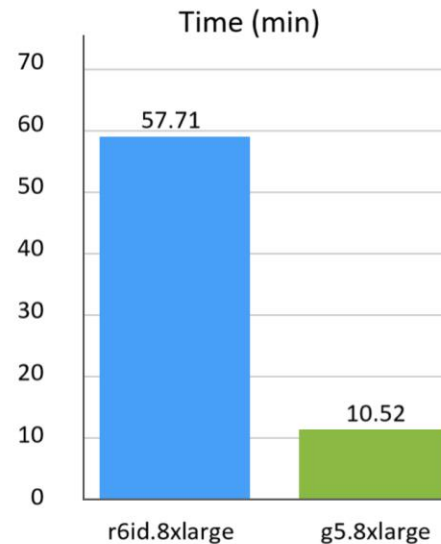
```
spark.conf.set("spark.plugins",
"com.nvidia.spark.SQLPlugin")

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

Accelerated Apache Spark ML

Bringing GPU-accelerated machine learning to every Apache Spark user

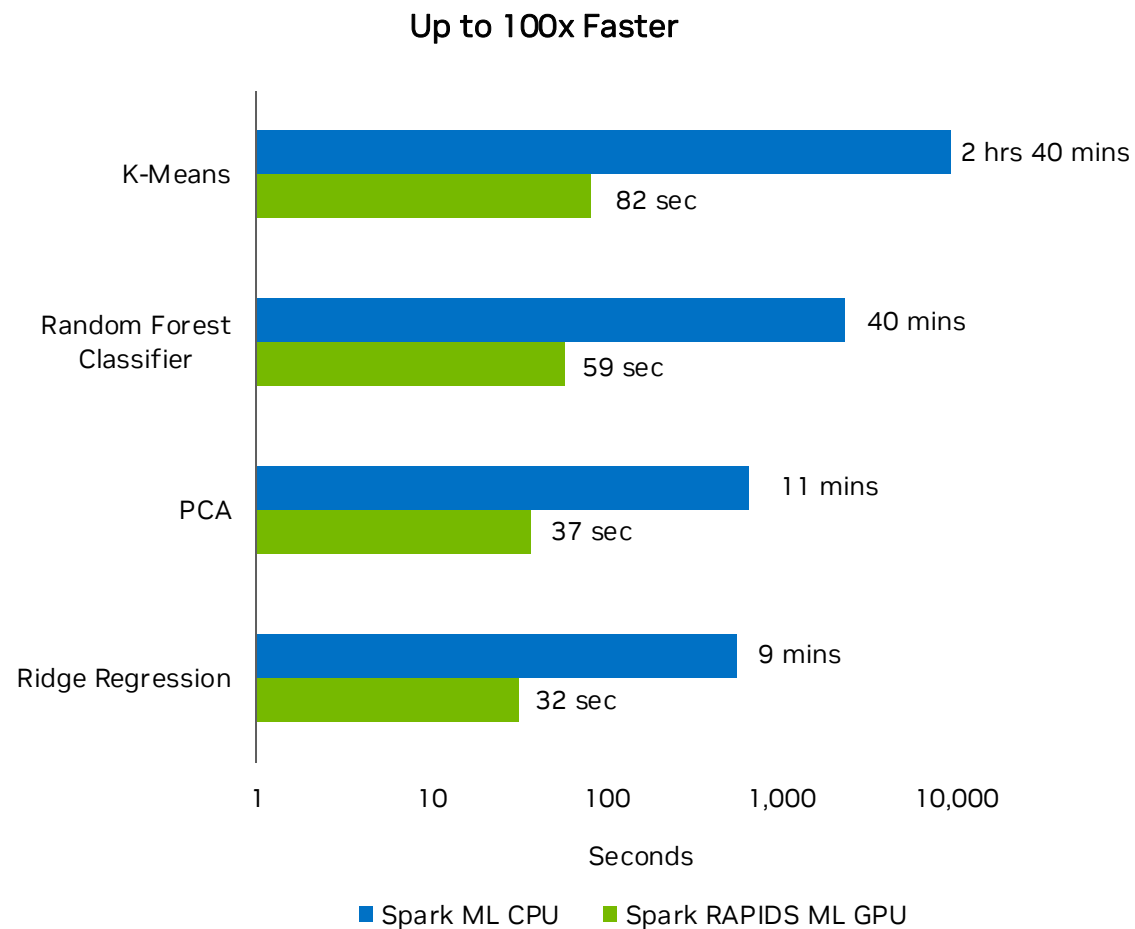
```
from spark_rapids_ml.clustering import Kmeans

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



Getting Started and Learning More

Deploying RAPIDS

Documentation to get you up and running RAPIDS anywhere

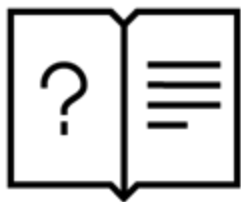
The image displays a grid of eight cards, each representing a different deployment method for RAPIDS. The cards are arranged in three rows: the first row has three cards (Local Machine, Cloud, HPC), the second row has three cards (Platforms, Tools, Cloud ML Examples), and the third row has one card (Guides) on the left. Each card has a title, a description, and a list of associated technologies or tools in purple buttons.

- Local Machine**
Use RAPIDS on your local workstation or server.
docker conda pip WSL2
- Cloud**
Use RAPIDS on the cloud.
Amazon Web Services
Google Cloud Platform
Microsoft Azure IBM Cloud
- HPC**
Use RAPIDS on high performance computers and supercomputers.
SLURM
- Platforms**
Use RAPIDS on compute platforms.
Kubernetes Kubeflow Coiled
Databricks
- Tools**
There are many tools to deploy RAPIDS.
containers dask-kubernetes
dask-operator dask-helm-chart
dask-gateway
- Cloud ML Examples**
See our [example notebooks repo](#) with opinionated deployments of RAPIDS to boost machine learning workflows.
xgboost optuna mlflow
ray tune
- Guides**
Detailed guides on how to deploy and optimize RAPIDS.
Microsoft Azure Infiniband MIG

[RAPIDS Deployment Documentation](#)

How to Get Started with RAPIDS

A Variety of Ways to Get Up & Running



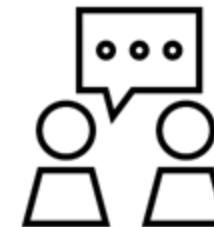
More about RAPIDS

- Learn more at [RAPIDS.ai](https://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/>

RAPIDS

<https://rapids.ai>

