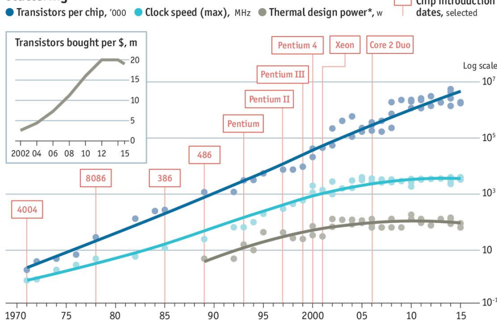
RAPIDS: Accelerated Data Science and Data Processing

Mads R. B. Kristensen, May 13, 2026

A short history of the CPU



Moore's law: transistor counts still double, but clock speed and single-thread perf have plateaued.

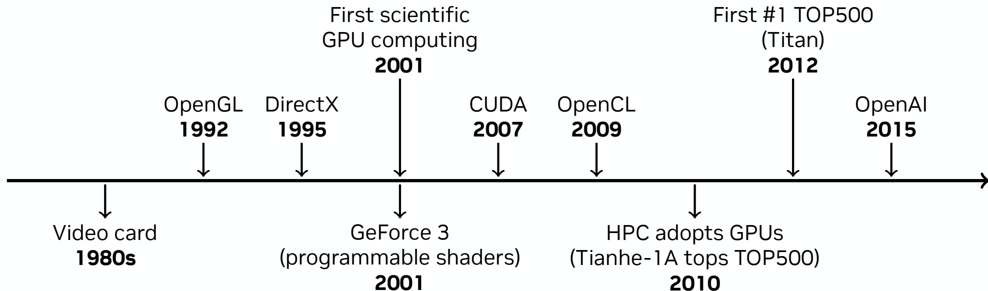
Transistor counts kept increasing, spent on:

- more cache
- deeper pipelines
- branch prediction
- out-of-order execution
- memory prefetching
- speculative execution

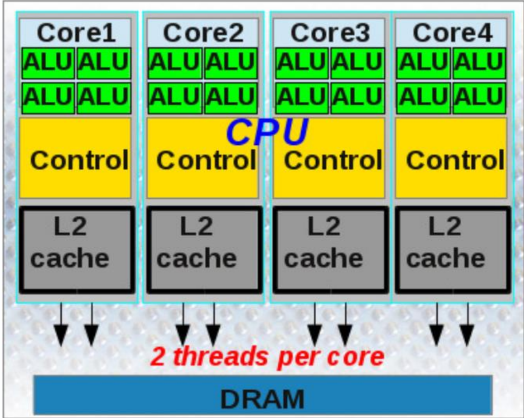
Finally, we had no other choice than to add multiple cores.

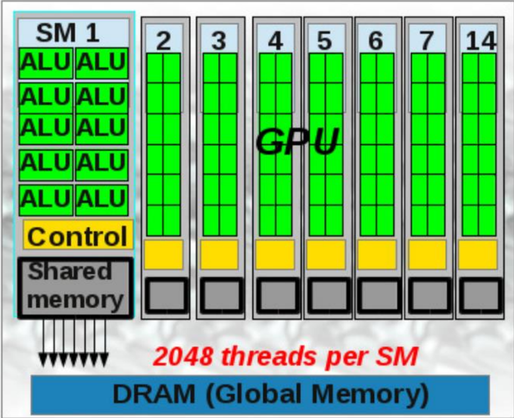
NVIDIA

A short history of the GPU



CPU vs GPU





NVIDIA.

GPUs aren’t just for HPC

The same hardware that runs simulations runs DataFrames

For decades, GPUs have powered:

- Scientific simulations and dense linear algebra
- Computer graphics and ray tracing
- Neural-network training (which is HPC in disguise)

The common thread: the **same** operation applied to **lots** of data in parallel.

Does that apply to structured-data analytics too?

NVIDIA.

Layout and bandwidth close the deal

Two things had to be true for analytics on GPUs to actually work:

1. **Columnar memory layout** (Apache Arrow, cuDF, Polars): contiguous columns let thousands of GPU threads read in lock-step.
2. **Massive memory bandwidth**: analytics is **bandwidth-bound**, not compute-bound.

	Compute	Memory bandwidth
Modern CPU	1–2 TFLOPS	$\sim$400 GB/s
Modern GPU	30–50 TFLOPS	2–3 TB/s

5–10$\times$ faster column scans, exactly what DataFrame work needs.

NVIDIA.

Example: a GROUPBY on the GPU

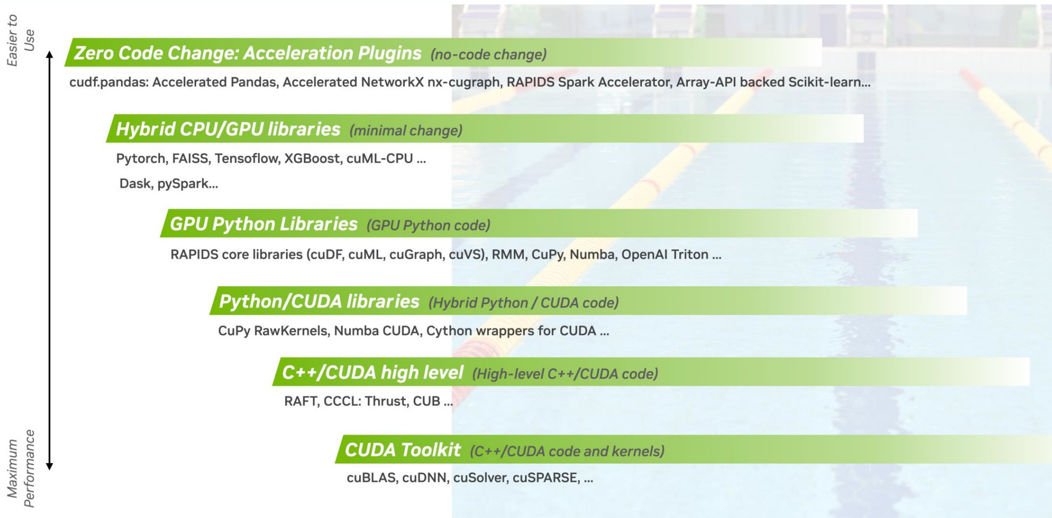
```
df.groupby("category").value.mean()
```

Decomposed:

1. Read both columns in parallel
2. Hash rows into groups
3. Parallel reduction per group
4. Write aggregated results

Thousands of GPU threads cooperate, no for-loop on the host.

The accelerated-computing “swim lanes”



RAPIDS overview

The accelerated data-science stack

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

cuDF: Accelerated Pandas

Pandas dominates DataFrames

- One of the most widely used Python libraries
- 670M+ monthly PyPI downloads
- Core component of the PyData ecosystem
- Used by $\sim 77\%$ of Python data scientists

```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)
)
```

cuDF: a pandas-like API on the GPU

Pandas (CPU):

```
import pandas as pd
df = pd.read_csv("data.csv")
df.groupby("col").mean()
```

cuDF (GPU):

```
import cudf
df = cudf.read_csv("data.csv")
df.groupby("col").mean()
```

cuDF on A100 vs pandas on Xeon 7642

Operation	Speed-up
groupby	405×
sort	211×
where	1171×

NVIDIA.

cuDF has gaps, though

cuDF coverage of the pandas API (green = implemented, grey = not).

NVIDIA

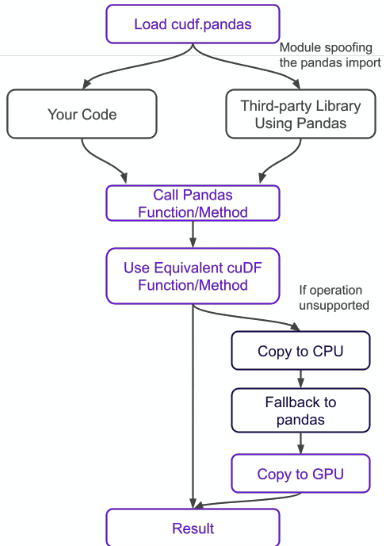
Accelerated Pandas

zero-code-change GPU acceleration

- *No* changes to existing pandas code.
- In a notebook: `%load_ext cudf.pandas`
- From the CLI:

```
python -m cudf.pandas script.py
```

- Falls back to pandas on the CPU for unsupported ops.
- Up to **150×** on the GPU; works with third-party libraries.



NVIDIA

cudf.pandas in action

```python
%load_ext cudf.pandas
import pandas as pd, numpy as np, 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)
idx = rng.indexer_between_time("09:30", "16:00")
df = df.iloc[idx]
results = df.groupby(pd.Grouper(freq="1D")).mean()
sns.lineplot(results)
```

The full plan: one line of setup, then pure pandas.

NVIDIA

cudf.pandas in action

```
%load_ext cudf.pandas
import pandas as pd, numpy as np, 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)
idx = rng.indexer_between_time("09:30", "16:00")
df = df.iloc[idx]
results = df.groupby(pd.Grouper(freq="1D")).mean()
sns.lineplot(results)
```

DataFrame construction runs entirely on the **GPU** (cuDF).

NVIDIA.

cudf.pandas in action

```
%load_ext cudf.pandas
import pandas as pd, numpy as np, 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)
idx = rng.indexer_between_time("09:30", "16:00")
df = df.iloc[idx]
results = df.groupby(pd.Grouper(freq="1D")).mean()
sns.lineplot(results)
```

`indexer_between_time` isn't in cuDF ⇒ falls back to CPU (pandas); result is copied back to the GPU.

NVIDIA.

cudf.pandas in action

```python
%load_ext cudf.pandas
import pandas as pd, numpy as np, 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)
idx = rng.indexer_between_time("09:30", "16:00")
df = df.iloc[idx]
results = df.groupby(pd.Grouper(freq="1D")).mean()
sns.lineplot(results)
```

`groupby().mean()` runs entirely on the **GPU**.

NVIDIA

cudf.pandas in action

```
1   %load_ext cudf.pandas
2   import pandas as pd, numpy as np, seaborn as sns
3
4   rng = pd.date_range("2023-01-01", "2023-02-01", freq="1T")
5   df  = pd.DataFrame({"a": np.random.rand(len(rng)),
6                       "b": np.random.rand(len(rng))}, index=rng)
7   idx = rng.indexer_between_time("09:30", "16:00")
8   df = df.iloc[idx]
9   results = df.groupby(pd.Grouper(freq="1D")).mean()
10  sns.lineplot(results)
```

Seaborn plotting works seamlessly through the proxy.

NVIDIA.

cudf.pandas in one slide

- Provides **all** of the pandas API
- GPU (cuDF) for supported ops
- CPU (pandas) for everything else
- Data movement is hidden from the user
- Zero code change: accelerates pandas in place

RAPIDS ML and Graph Analytics

cuML: scikit-learn-compatible ML on the GPU

scikit-learn:

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

cuML:

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

50+ GPU-accelerated algorithms: classification, regression, clustering, dim. reduction, time series, tree models, explainability.

NVIDIA.

Zero-code-change cuML (cuml.accel)

- Drop-in accelerator: `%load_ext cuml.accel`
- Supports K-means, linear / logistic regression, random forest, ...
- Speed-ups $6\times$ – $116\times$ on common algorithms
- Cost-benefit (cpu$/gpu$) still favourable ($3\times$ – $58\times$)

NVIDIA.

Accelerated XGBoost

```python
from xgboost import XGBClassifier
clf = XGBClassifier(device="cuda")   # the one-line change
clf.fit(x, y)
```

- Up to **20×** speed-up
- Scales to the largest datasets via Dask and PySpark
- Built-in SHAP for explainability
- Triton-deployable for production inference
- RAPIDS team are core maintainers

Polars: a plan optimizer for DataFrames

Eager vs lazy

Eager (pandas, cuDF)

```
df = pd.read_csv("data.csv")
df = (
    df[df["x"] > 0]
      .groupby("category")
      .mean()
)
```

- Each step executes immediately
- No view of the full plan
- Few optimization opportunities

Lazy (Polars)

```
q = (
    pl.scan_csv("data.csv")
      .filter(pl.col("x") > 0)
      .group_by("category")
      .mean()
)
result = q.collect()
```

- Builds a plan
- Optimizes the whole plan
- Executes only at `.collect()`

NVIDIA.

Why lazy matters

With the full plan in hand, Polars can apply optimization like:

- Predicate pushdown
- Projection pruning
- Operator fusion
- Streaming execution
- Parallel scheduling

A short history of the GPU

PDS-H benchmark, 22 queries on a ~10 GB dataset

Engine	Total time	vs pandas
pandas 2.2.3	365.71 s	1×
Polars 1.30.0	3.89 s	94×

Source: pola.rs/posts/benchmarks/, June 2025.

NVIDIA.

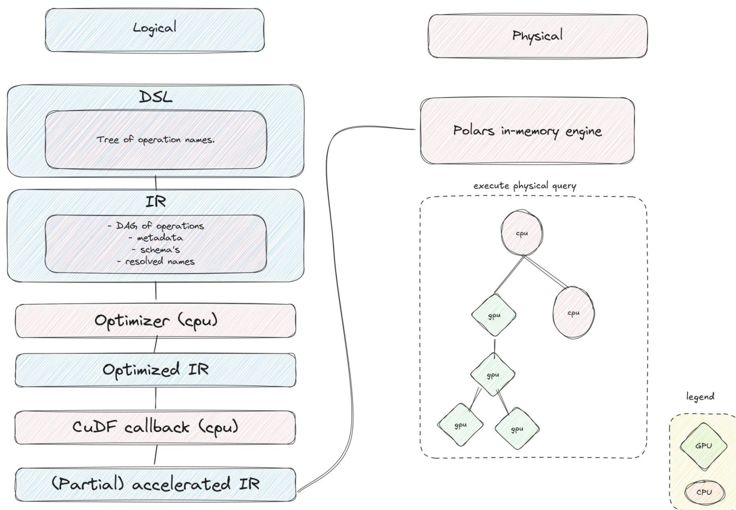
Polars on the GPU

```python
import polars as pl

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

result_cpu = q.collect()                    # CPU
result_gpu = q.collect(engine="gpu")    # GPU via cuDF
```

- Lazy DataFrame library (Python and Rust)
- Single-GPU engine: **one-keyword change**

NVIDIA.

How the Polars GPU engine works



NVIDIA.

Scaling Polars to multiple GPUs

One rank per GPU on a default local Ray cluster

```python
import polars as pl
from cudf_polars.experimental.rapidsmpf.frontend.ray import RayEngine

q = (pl.scan_parquet("ny-taxi/2024/*.parquet")
       .filter(pl.col("total_amount") > 15.0))

with RayEngine() as engine:
    result = q.collect(engine=engine)
```

- Same lazy query, only the engine changes.
- `RayEngine()` spins up a local Ray cluster, one rank per available GPU.
- Built on Ray + RapidsMPF for fast inter-GPU transfer.

NVIDIA

Other multi-GPU engines

Same idea, different backend:

```
from cudf_polars.experimental.rapidsmpf.frontend.dask import DaskEngine
from cudf_polars.experimental.rapidsmpf.frontend.spmd import SPMDEngine
```

- **RayEngine**: runs on a Ray cluster.
- **DaskEngine**: runs on a Dask-CUDA cluster.
- **SPMDEngine**: low-level base for custom multi-process / MPI-style setups.

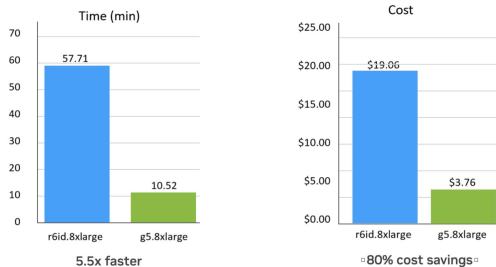
Pick whichever cluster manager you already use; the Polars query plan stays the same.

RAPIDS Accelerator for Apache Spark

RAPIDS as a Spark plugin

```
spark.conf.set(
  "spark.plugins",
  "com.nvidia.spark.SQLPlugin"
)
spark.sql("""
SELECT order, count(*) AS order_count
FROM orders
""")
```

- No code changes to your queries
- Accelerates supported operations; CPU fallback otherwise
- Works with Spark standalone, YARN, Kubernetes



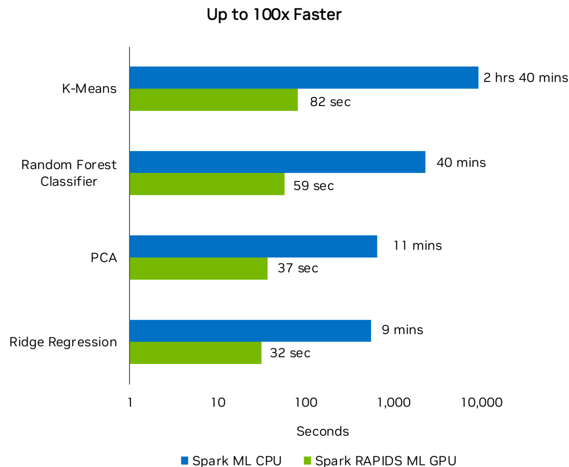
NVIDIA Decision Support Benchmark, 3 TB, 5.5$\times$ faster, $\sim$80% cheaper.

NVIDIA

Spark ML on the GPU

```
from spark_rapids_ml.clustering \
    import KMeans

estm  = (
    KMeans()
        .setK(100)
        .setFeaturesCol("features")
        .setMaxIter(30)
)
model = estm.fit(df)
```

Up to **100× faster** on common Spark ML algorithms (CPU cluster vs A10 GPU, 12 GB synthetic dataset).



Getting Started and Learning More

Where can you run RAPIDS?

Local machine
docker, conda, pip, WSL2

Cloud
AWS, GCP, Azure, IBM Cloud

HPC
SLURM-managed clusters

Platforms
Kubernetes, Kubeflow, Coiled, Databricks

Orchestration
dask-kubernetes, dask-operator, dask-helm-chart, dask-gateway

ML examples
XGBoost + Optuna, MLflow, Ray Tune

Take-aways

- Single-thread scaling is dead; data-parallel hardware wins.
- **RAPIDS** mirrors the PyData stack on the GPU:
    - cuDF $\sim$ pandas
    - cuML $\sim$ scikit-learn
    - cuGraph $\sim$ NetworkX
- **Zero-code-change** accelerators lower the barrier to entry.
- **Multi-GPU Polars** scales the same code to Ray, Dask, or SPMD clusters.
- **Spark** extend the same APIs to clusters, and they compose with RAPIDS.

NVIDIA

Getting started

Project page	rapids.ai
Docs / API	docs.rapids.ai
Try in-browser	Google Colab (free GPU runtime)
Source	github.com/rapidsai
Community	RAPIDS Slack, NVIDIA Developer Forums

Questions?