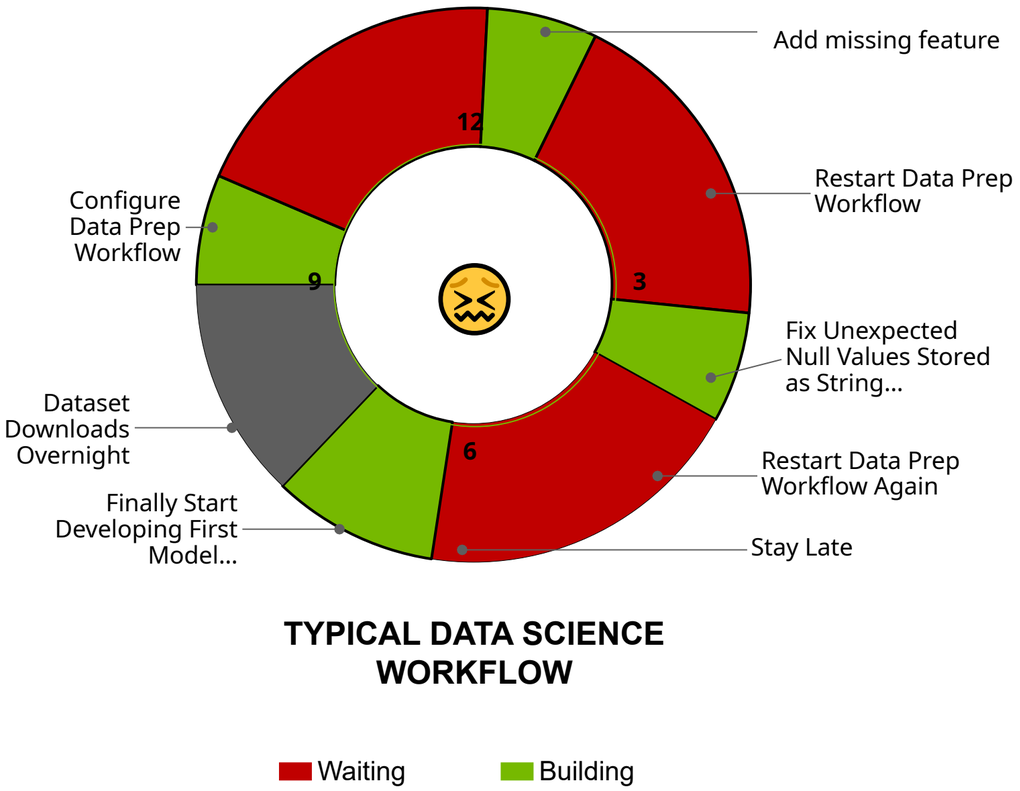
DLI Accelerated Data Science Teaching Kit

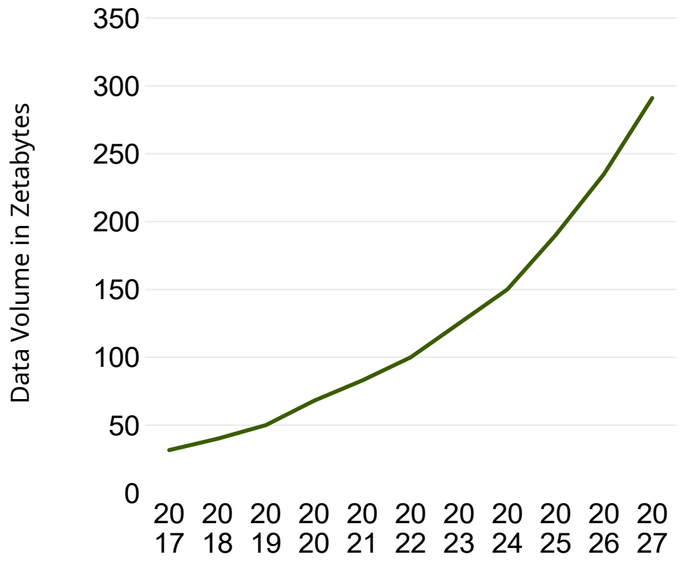
Lecture 21.1 – GPU Accelerating Your Workflows

Challenges with Applying Data Science in Industry Today

Performance bottlenecks limit user productivity



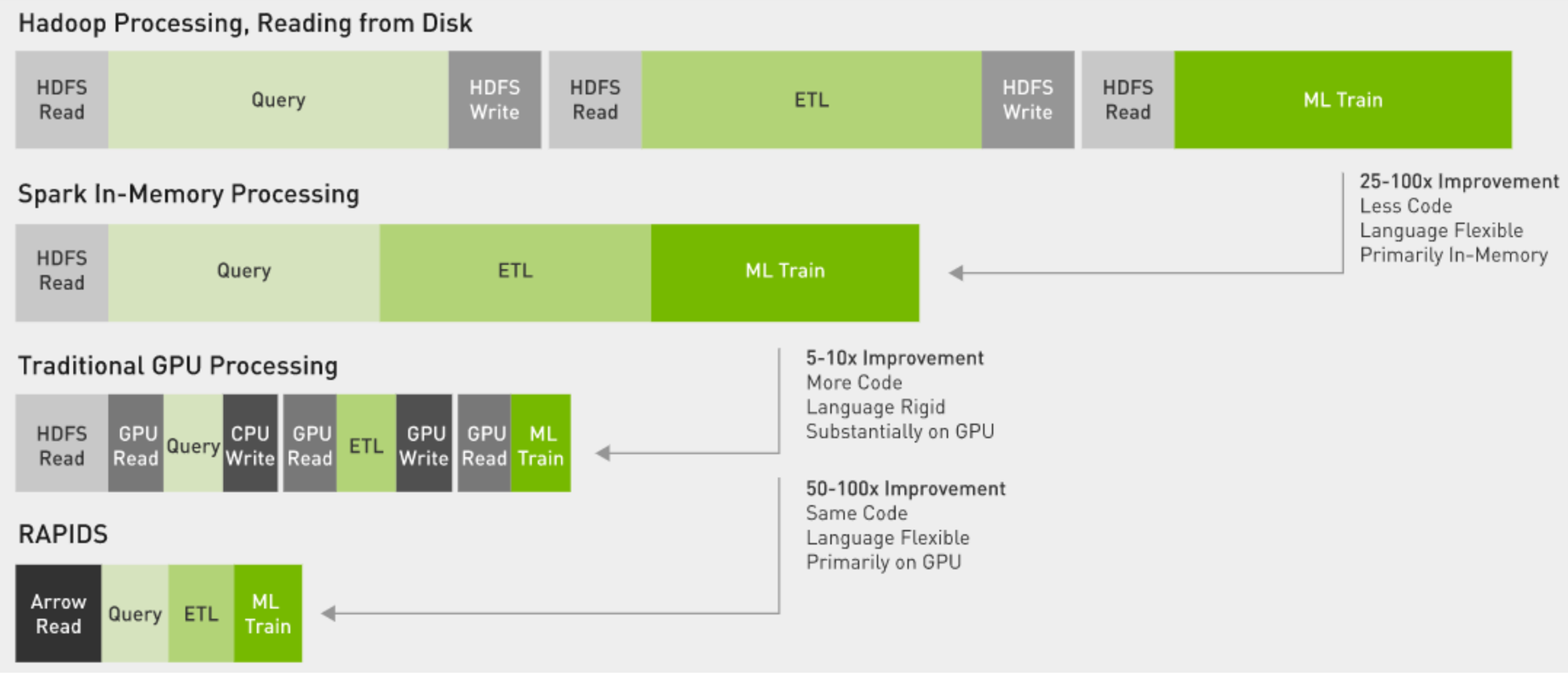
Data continues to explode



Source: IDC, Revelations in the Global DataSphere, US49346123, July 2023

Brief History on Data Processing Tools

Data processing evolution

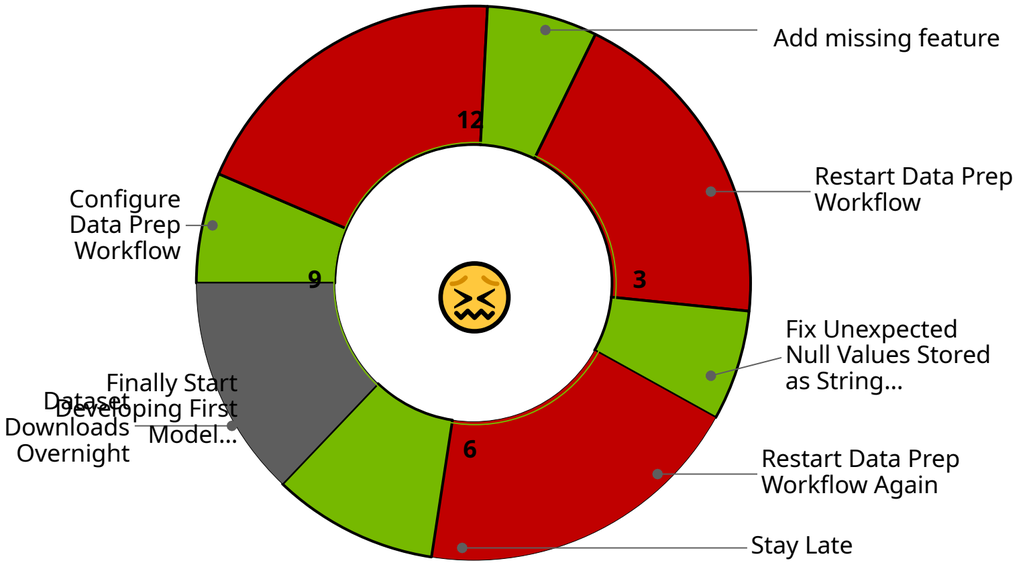


Overview of data processing with different frameworks

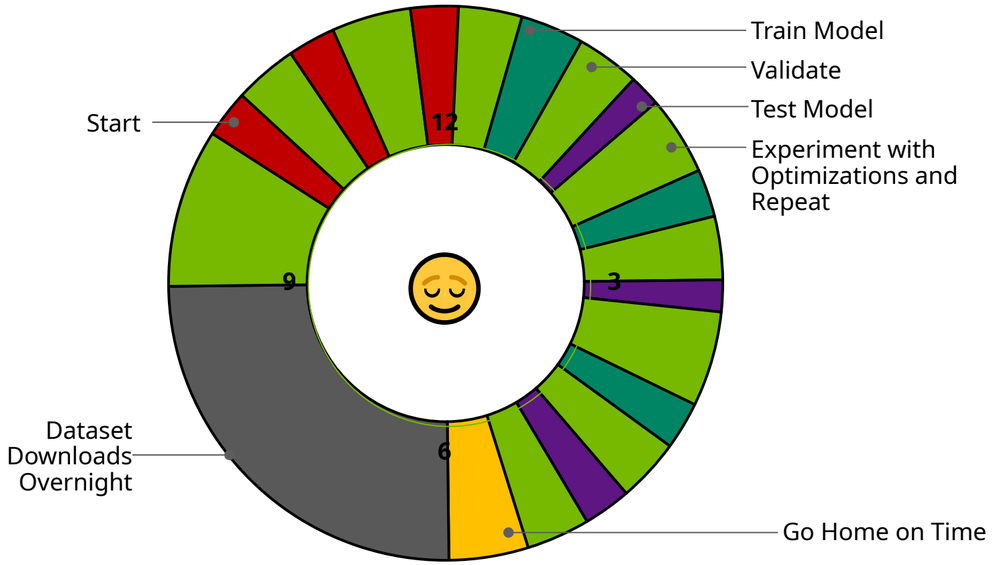
 Image credit: NVIDIA

Accelerated Data Science

Less waiting, more building



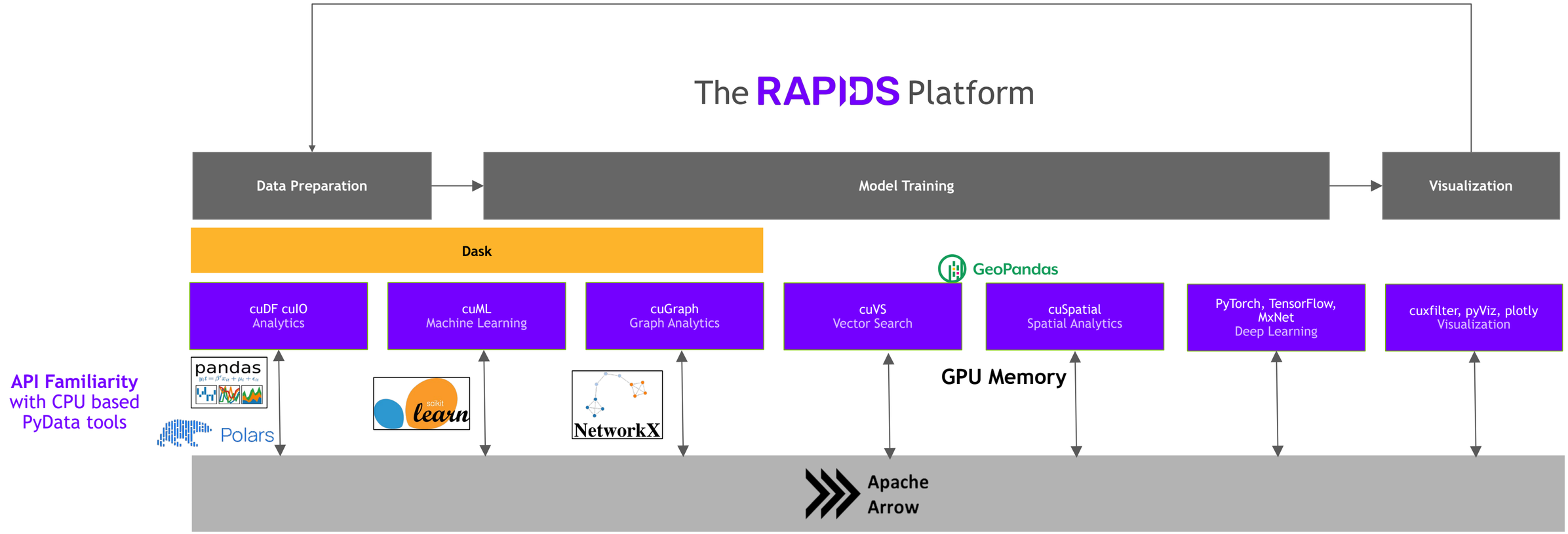
CPU WORKFLOW



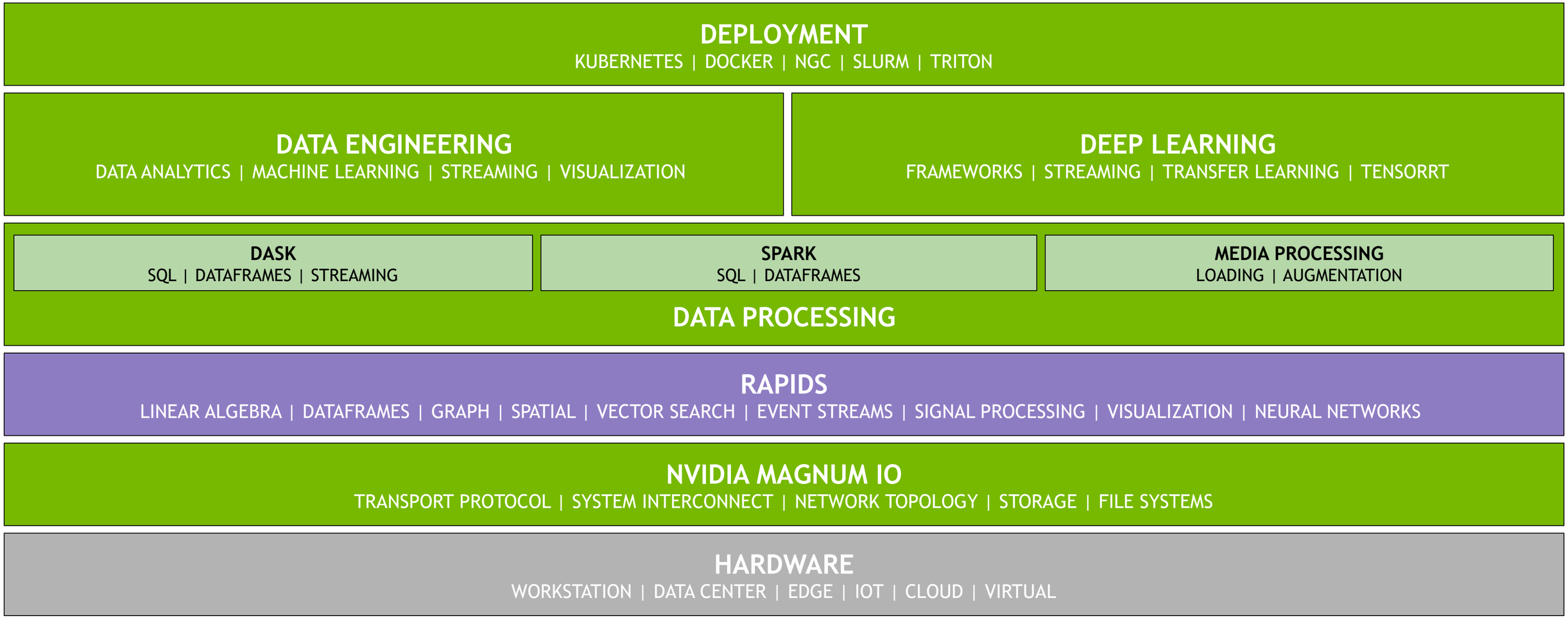
ACCELERATED COMPUTING WORKFLOW

Waiting Building

RAPIDS Libraries Accelerate Data Science Pipelines End-to-End



NVIDIA Accelerated Data Science Stack



Benefits of RAPIDS

Easy integration and familiarity

- Easy end-to-end accelerated analytics using GPUs
- Connecting data practitioners to High Performance Computing
- Familiar syntax for most data scientists

- Integrated with several data science frameworks like Apache Spark, Numba, etc. along with Deep Learning frameworks like Pytorch, Tensorflow, etc.
- Overcomes communication bottlenecks with the use of frameworks like UCX-py

Benefits of RAPIDS

Run anywhere at scale

- Great scalability: a single workstation to multi-GPU servers to multi-node clusters
- Provides a platform to scale up and out with the help of other libraries like Dask
- Run anywhere: Cloud or on-premise environment
- Faster data access with less data movement
- Bridging the gap between compute resources and existing frameworks

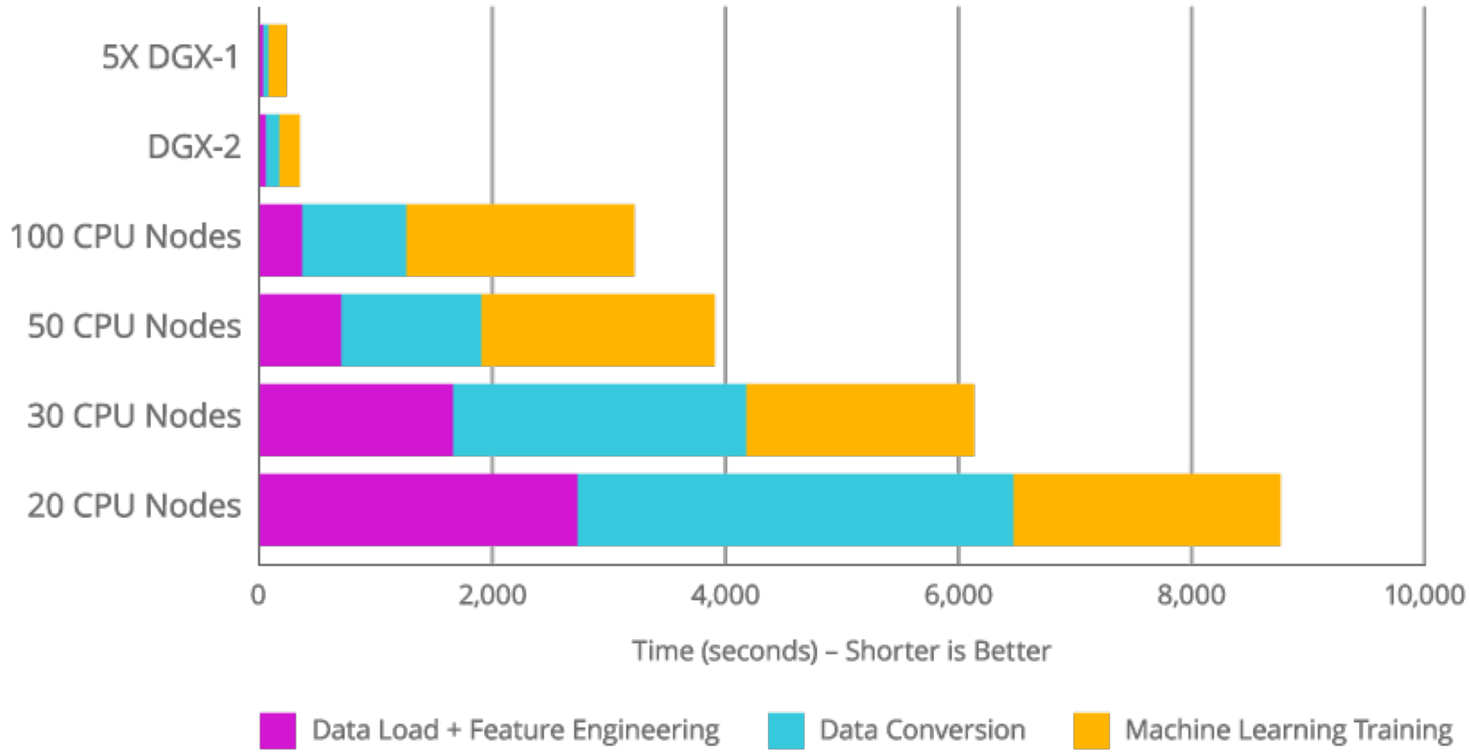
Speed Comparison

Introduction

- Data exploration is a vital part of Data Science Workflows:
  - Understanding, cleaning, manipulating data
  - Consistent data types, formats and filling in gaps
  - Reiterating to add features
- Pandas: Functions developed with vectorized operations for top-speed computations
- But they are CPU-constrained, therefore, tasks are very time consuming
- RAPIDS: Much faster Extract, Transform and Load (ETL) tasks

Speed Comparison

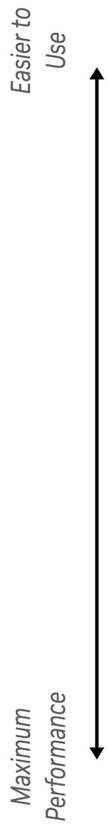
Example: Fannie Mae loan performance Dataset: 400 GB data in memory



Results for a complete ETL (manipulating DataFrames and training a gradient boosted decision tree model on the GPU using XGBoost)

 Image credit: NVIDIA

There are 3 levels to access this acceleration

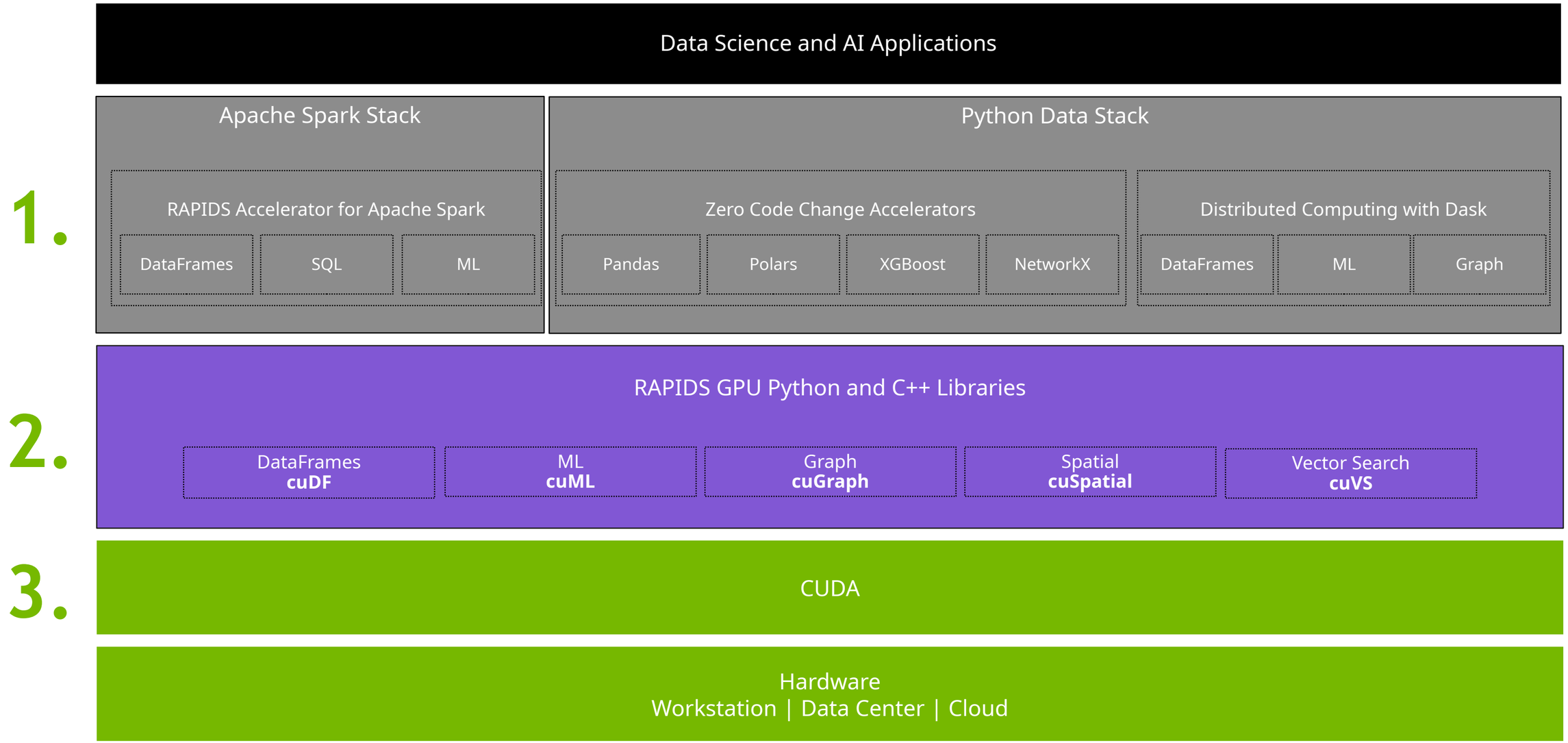


1. ***Zero Code Change: Acceleration Plugins*** *(no-code change)*
   pandas and Polars accelerated with cuDF, NetworkX accelerated by cuGraph, RAPIDS Accelerator for Apache Spark...

2. ***GPU Python Libraries*** *(GPU Python code)*
   RAPIDS core libraries (cuDF, cuML, cuGraph, cuVS), RMM, CuPy, Numba, OpenAI Triton

3. ***C++/CUDA high level*** *(High-level C++/CUDA code)*

The GPU Accelerated Data Science Stack



1. Intro to Zero-Code-Change

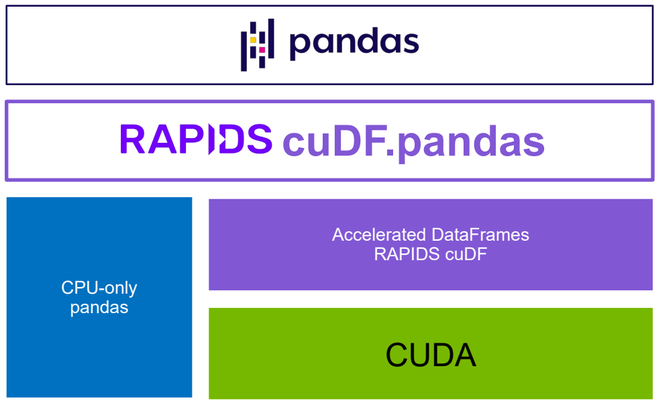
Zero Code Change Acceleration....But Why?

- Popular libraries are hard to switch-
  - Familiarity
  - Availability of tooling
  - Alternatives requiring changes to preexisting code
- Same code for both CPU and GPU env -
  - Developing on local machines with CPU
  - Deploying on cloud (Spot Instances / EC2) with GPU access
- Not the whole code needs to run on GPU-
  - Memory Hungry tasks like I/O on CPU
  - Compute Hungry tasks like feature engineering, model training on GPU

GPU accelerator for pandas powered by RAPIDS cuDF

Up to 50x speedup with zero code change

The cuDF.pandas software stack



- Accelerates workflows up to 50x using the GPU
- Compatible with code that uses third-party libraries
  - Integration tested with SciPy, scikit-learn, XGBoost, Matplotlib, seaborn, HoloViews, PyTorch, TensorFlow, …

Requires *no changes* to existing pandas code. Just:

```
%load_ext cudf.pandas
import pandas as pd
...
```

To accelerate a Python script, use the Python module flag on the command line:

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

Or, explicitly enable cudf.pandas via import if you can't use command line flags:

```
import cudf.pandas
cudf.pandas.install()

import pandas as pd
...
```

GPU accelerator for pandas powered by RAPIDS cuDF

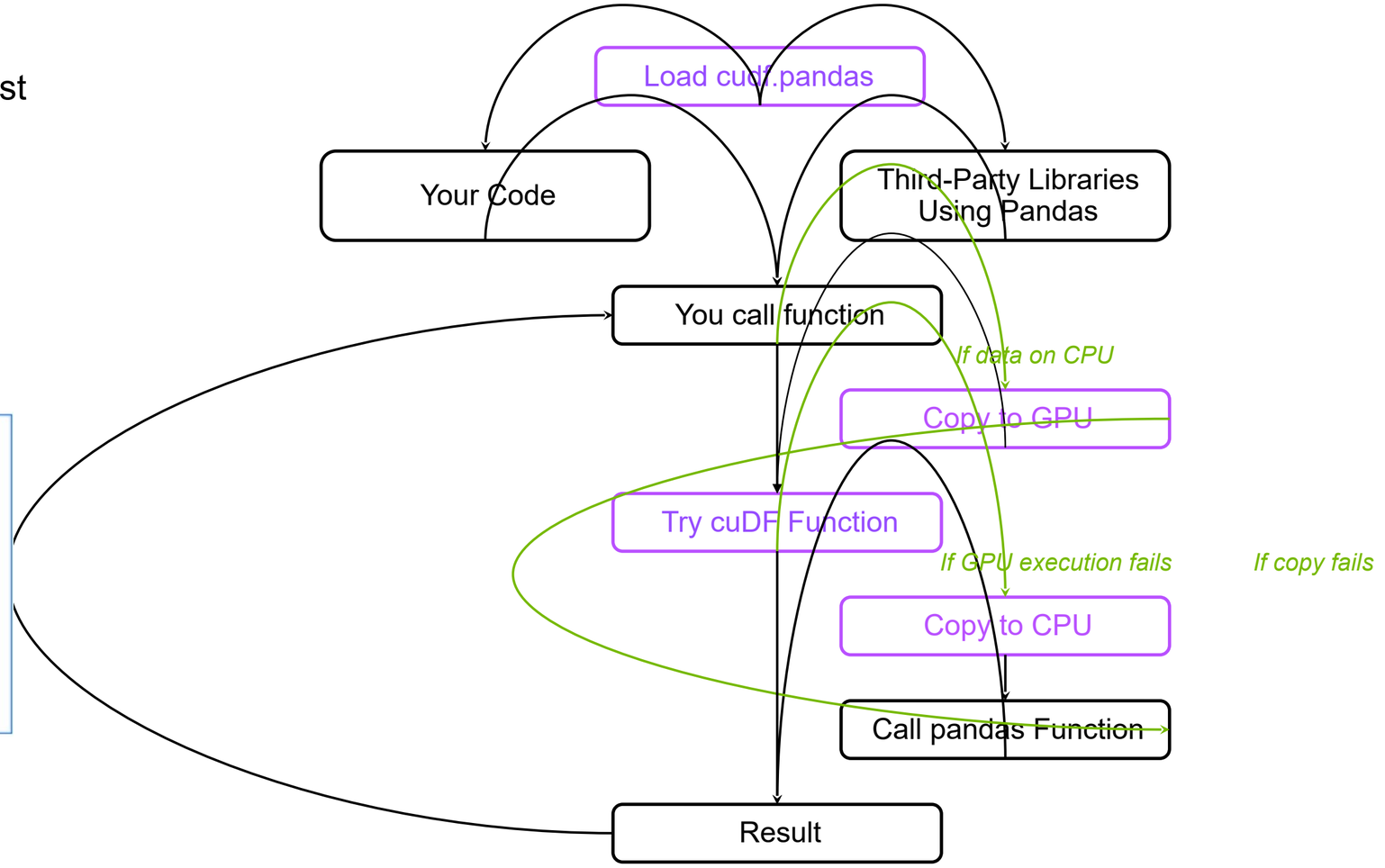
How it works

- Requires ***no changes*** to existing pandas code. Just
  - %load_ext cudf.pandas
  - $ python –m cudf.pandas <script.py>

```
[ ]:
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)
```



Demonstration video

cuDF.Pandas can profile your code to optimize how it runs on both CPU and GPU

- Use the %%cudf.pandas.profile cell magic in Jupyter, or import it directly:

```
from cudf.pandas import Profiler

with Profiler() as p:
    # code goes here

p.print_per_function_stats()
```

- Shows which functions ran on the CPU and which ran on the GPU

```
%%cudf.pandas.profile

rng = pd.date_range("2023-01-01", "2023-02-01", freq="10ms")
data = pd.DataFrame(
    {
        "a": np.random.rand(len(rng)),
        "b": np.random.rand(len(rng))
    },
    index=rng
)
data = data.iloc[rng.indexer_between_time("09:30", "16:00")]
results = data.groupby(pd.Grouper(freq="1D")).mean()
results.head()
```

Total time elapsed: 12.855 seconds
11 GPU function calls in 1.322 seconds
1 CPU function calls in 4.416 seconds

Stats

Function	GPU ncalls	GPU cumtime	GPU percall	CPU ncalls	CPU cumtime	CPU percall
date_range	1	0.008	0.008	0	0.000	0.000
DatetimeIndex.__len__	2	0.000	0.000	0	0.000	0.000
DataFrame	2	0.873	0.436	0	0.000	0.000
DatetimeIndex.indexer_betwee…	0	0.000	0.000	1	4.416	4.416
_DataFrameIlocIndexer.__geti…	1	0.127	0.127	0	0.000	0.000
Grouper	1	0.000	0.000	0	0.000	0.000
DataFrame.groupby	1	0.021	0.021	0	0.000	0.000
DataFrameResampler.mean	1	0.259	0.259	0	0.000	0.000
DataFrame.head	1	0.001	0.001	0	0.000	0.000
DataFrame.__repr__	1	0.033	0.033	0	0.000	0.000

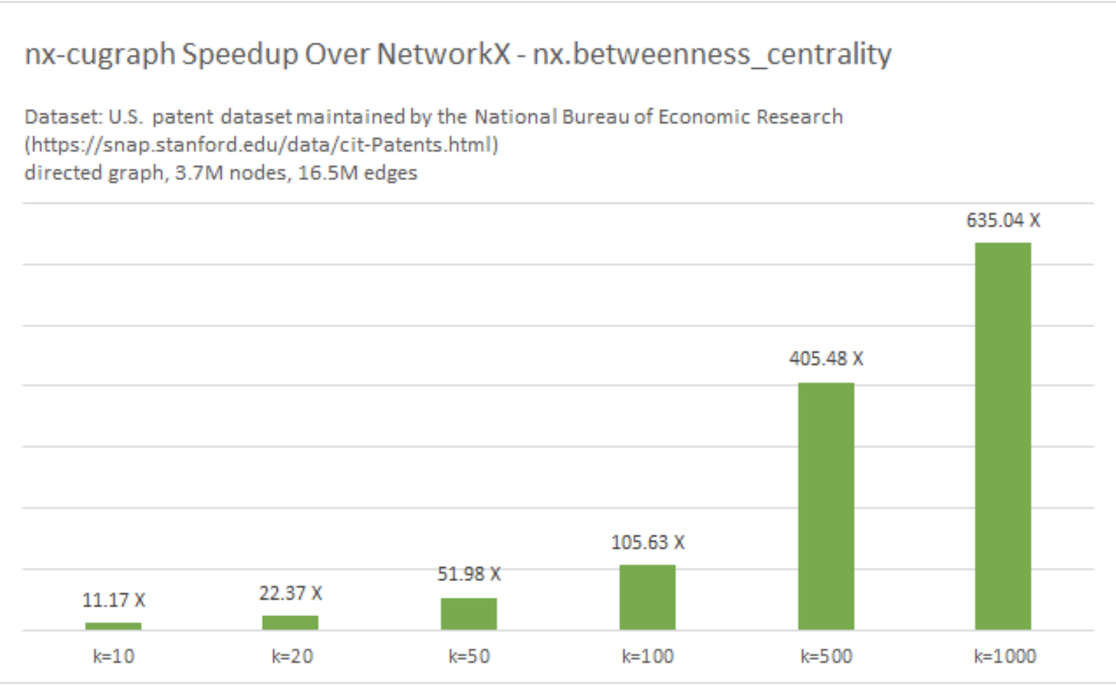
Not all pandas operations ran on the GPU. The following functions required CPU fallback:

- DatetimeIndex.indexer_between_time

To request GPU support for any of these functions, please file a Github issue here:
https://github.com/rapidsai/cudf/issues/new/choose.

NetworkX accelerated by RAPIDS cuGraph

- Zero-code-change GPU-acceleration for NetworkX code
- Accelerates up to 600x depending on algorithm and graph size
- Support for 60 popular graph algorithms and growing
- Fallback to CPU for any unsupported algorithms



NetworkX 3.2, CPU: Intel(R) Xeon(R) Platinum 8480CL 2TB, GPU: NVIDIA H100 80GB

```python
import pandas as pd
import networkx as nx

url = "https://data.rapids.ai/cugraph/datasets/cit-Patents.csv"
df = pd.read_csv(url, sep=" ", names=["src", "dst"], dtype="int32")
G = nx.from_pandas_edgelist(df, source="src", target="dst")

%time result = nx.betweenness_centrality(G, k=10)
```

```
user@machine:/# ipython bc_demo.ipy
CPU times: user 7min 38s, sys: 5.6 s, total: 7min 44s
Wall time: 7min 44s

user@machine:/# NETWORKX_BACKEND_PRIORITY=cugraph ipython bc_demo.ipy
CPU times: user 18.4 s, sys: 1.44 s, total: 19.9 s
Wall time: 20 s
```

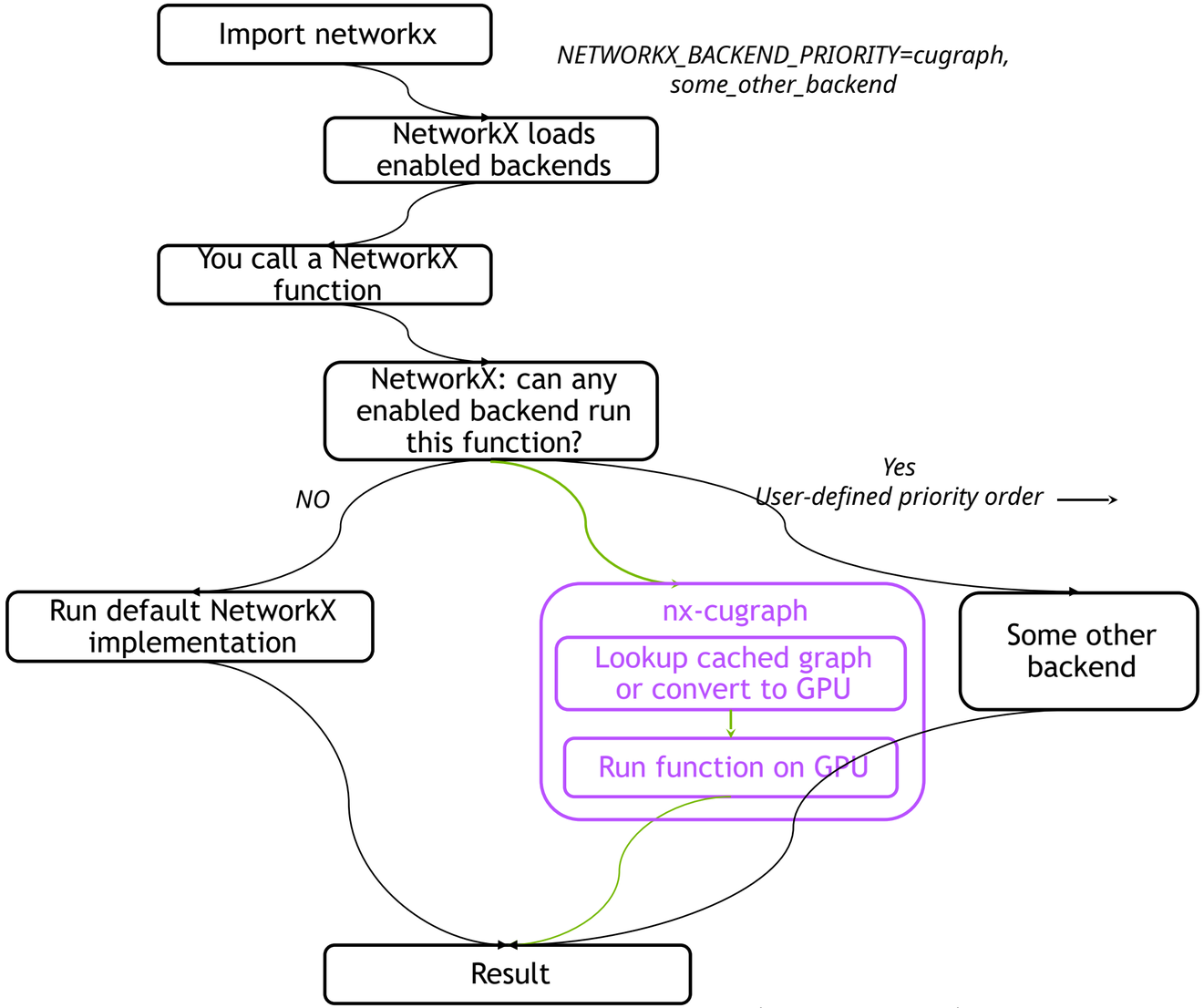
NeworkX accelerated by RAPIDS cuGraph

How it works

- nx-cugraph is a GPU ***backend*** for NetworkX

- What's a NetworkX backend?
  - NetworkX added the ability to ***dispatch*** various function calls to third-party backends, starting in NetworkX 3.0
  - Backends provide an alternate implementation for NetworkX to call
    - Allows users to run implementations optimized for their environment without changing their code – the NetworkX "frontend" remains the same

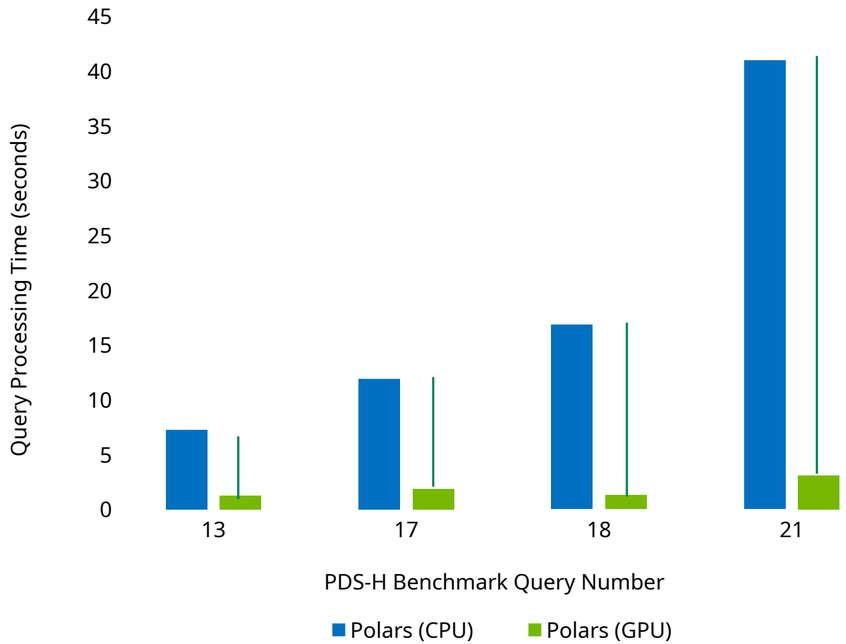
- Multiple backends can be used together:
  - Ex. Access a graph in a remote graph database using a database backend, run algorithms on GPU using the nx-cugraph backend



Polars GPU engine powered by RAPIDS cuDF

Process 100s of millions of rows in seconds

- Delivers fastest performance for Polars applications, 13x on NVIDIA GPUs vs CPUs
- Zero code change – simply set engine="gpu" in the *collect* operation
- Compatible with the ecosystem of tools built for Polars
- Graceful CPU fallback for unsupported queries




```python
import polars as pl

(transactions
 .group_by("CUST_ID").agg(pl.col("AMOUNT").sum())
 .sort(by="AMOUNT", descending=True)
 .head()
 .collect())
```

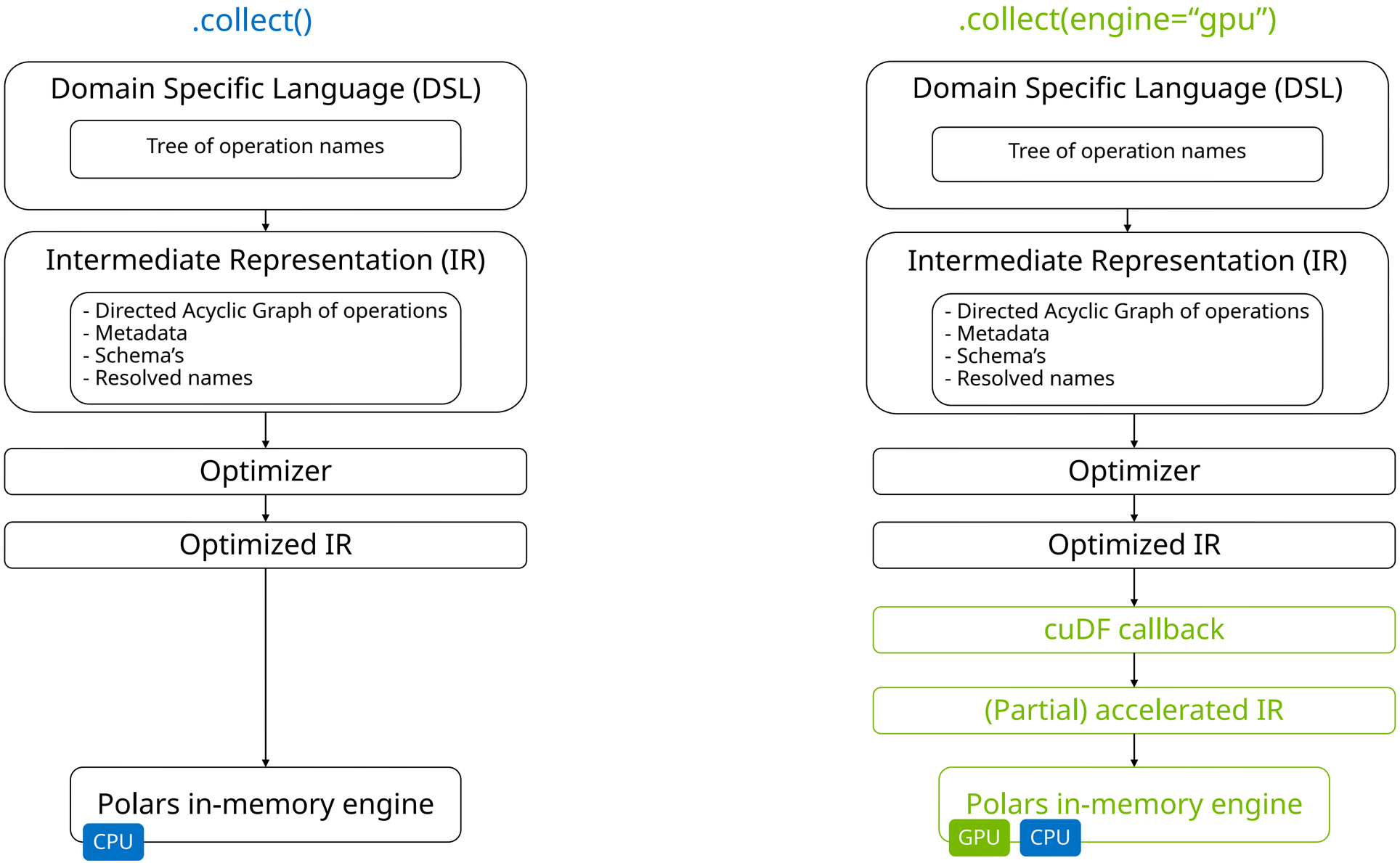
```python
import polars as pl

(transactions
 .group_by("CUST_ID").agg(pl.col("AMOUNT").sum())
 .sort(by="AMOUNT", descending=True)
 .head()
 .collect(engine="gpu"))
```


PDS-H benchmark | GPU: NVIDIA H100 PCIe | CPU: Intel Xeon W9-3495X (Sapphire Rapids) | Storage: Local NVMe

Polars GPU engine powered by RAPIDS cuDF

How it works



DLI Accelerated Data Science Teaching Kit

Thank You