



# ACCELERATED DATA SCIENCE IN MEDICINE WITH CUPY, RAPIDS & NUMBA

HUIWEN JU - [HJU@NVIDIA.COM](mailto:HJU@NVIDIA.COM) - SOLUTIONS ARCHITECT, HIGHER EDUCATION & RESEARCH

# AGENDA

---

Overview of GPU Computing

---

GPU-Accelerated Numerical Computing with *CuPy*

---

GPU-Accelerated Data Science with *RAPIDS*

---

Custom GPU Kernels with *Numba*

---

Frameworks Interoperability - *Data Conversion Bottleneck*

---

ZERO-COPY end-to-end pipeline - try out on *NVIDIA Brev*



# AGENDA

Overview of GPU Computing

---

GPU-Accelerated Numerical Computing with *CuPy*

---

GPU-Accelerated Data Science with *RAPIDS*

---

Custom GPU Kernels with *Numba*

---

Frameworks Interoperability - *Data Conversion Bottleneck*

---

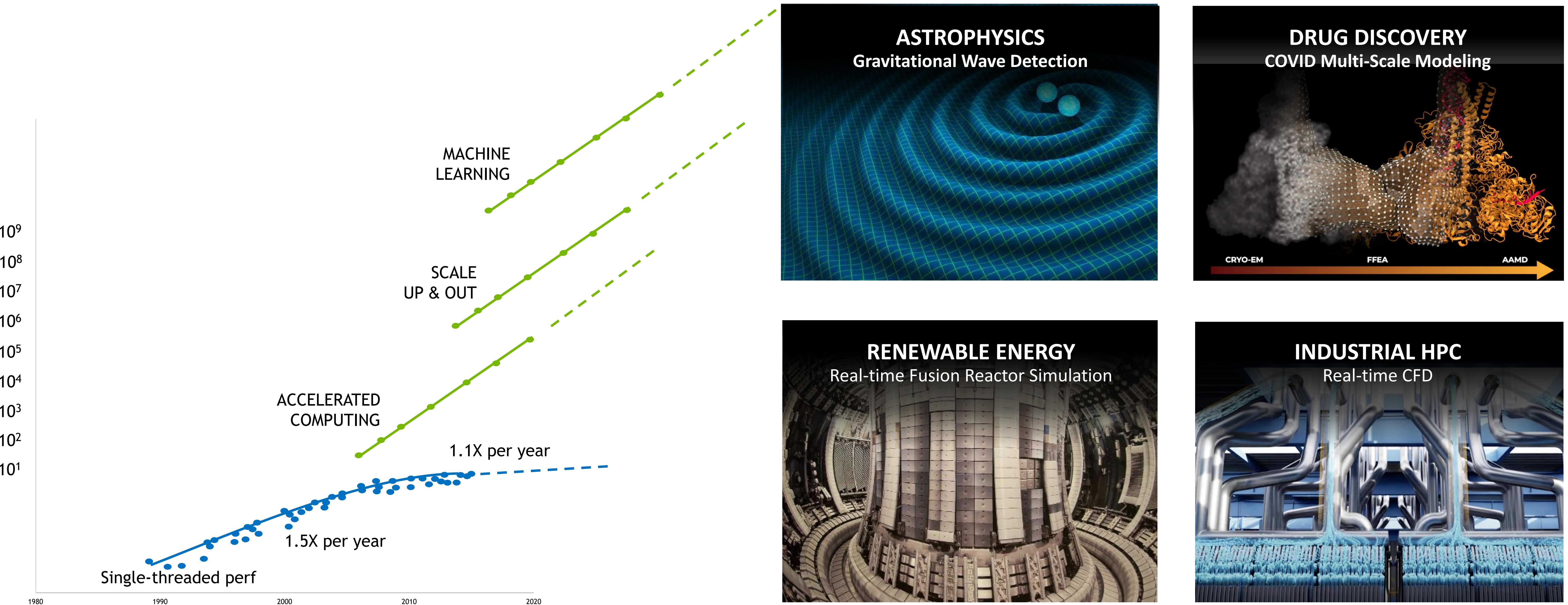
ZERO-COPY end-to-end pipeline - try out on *NVIDIA Brev*

---

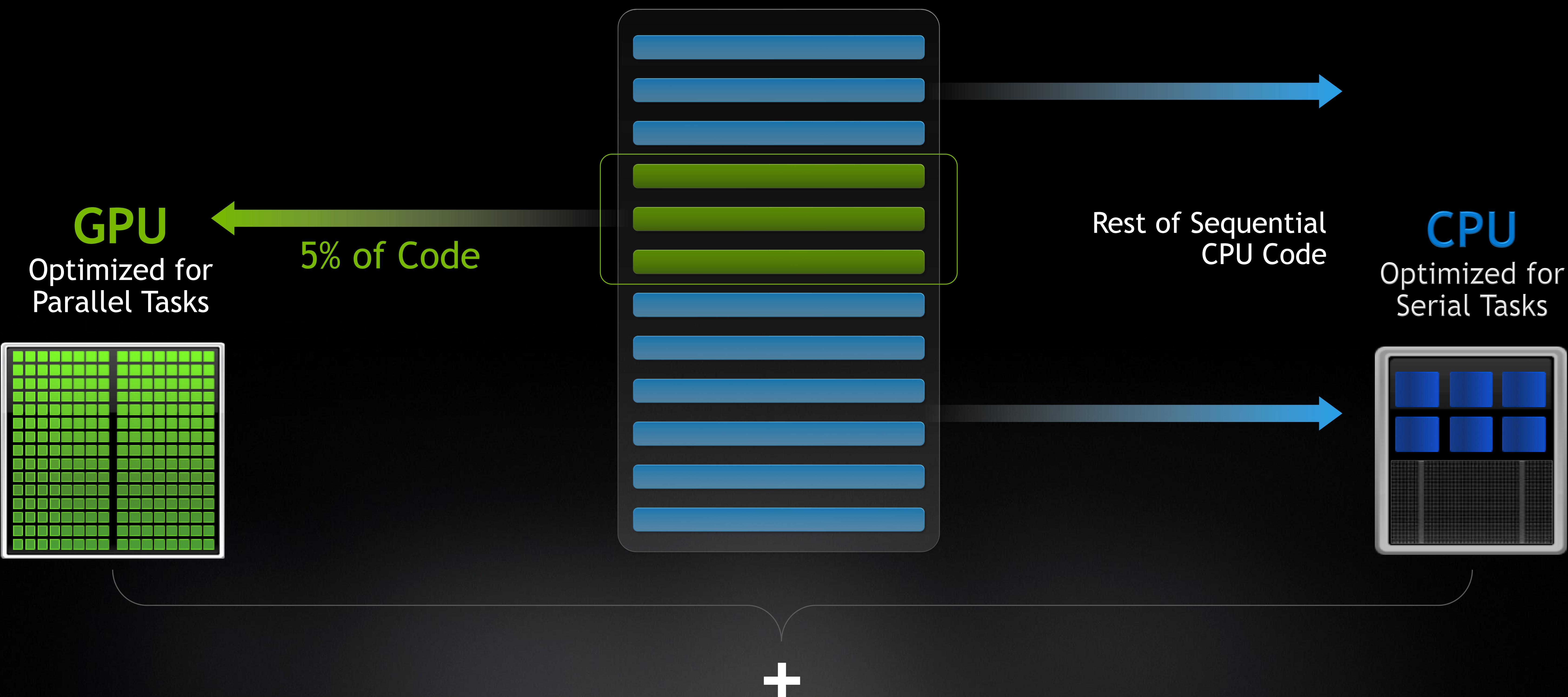


# MILLION-X SPEEDUP FOR INNOVATION AND DISCOVERY

Combination of Accelerated Computing, Data Center Scale and AI



# ACCELERATED COMPUTING WITH GPUS



# A FEW GENERAL TIPS FOR SUCCESSFUL GPU COMPUTING

- **Minimize data movement to and from the GPU**

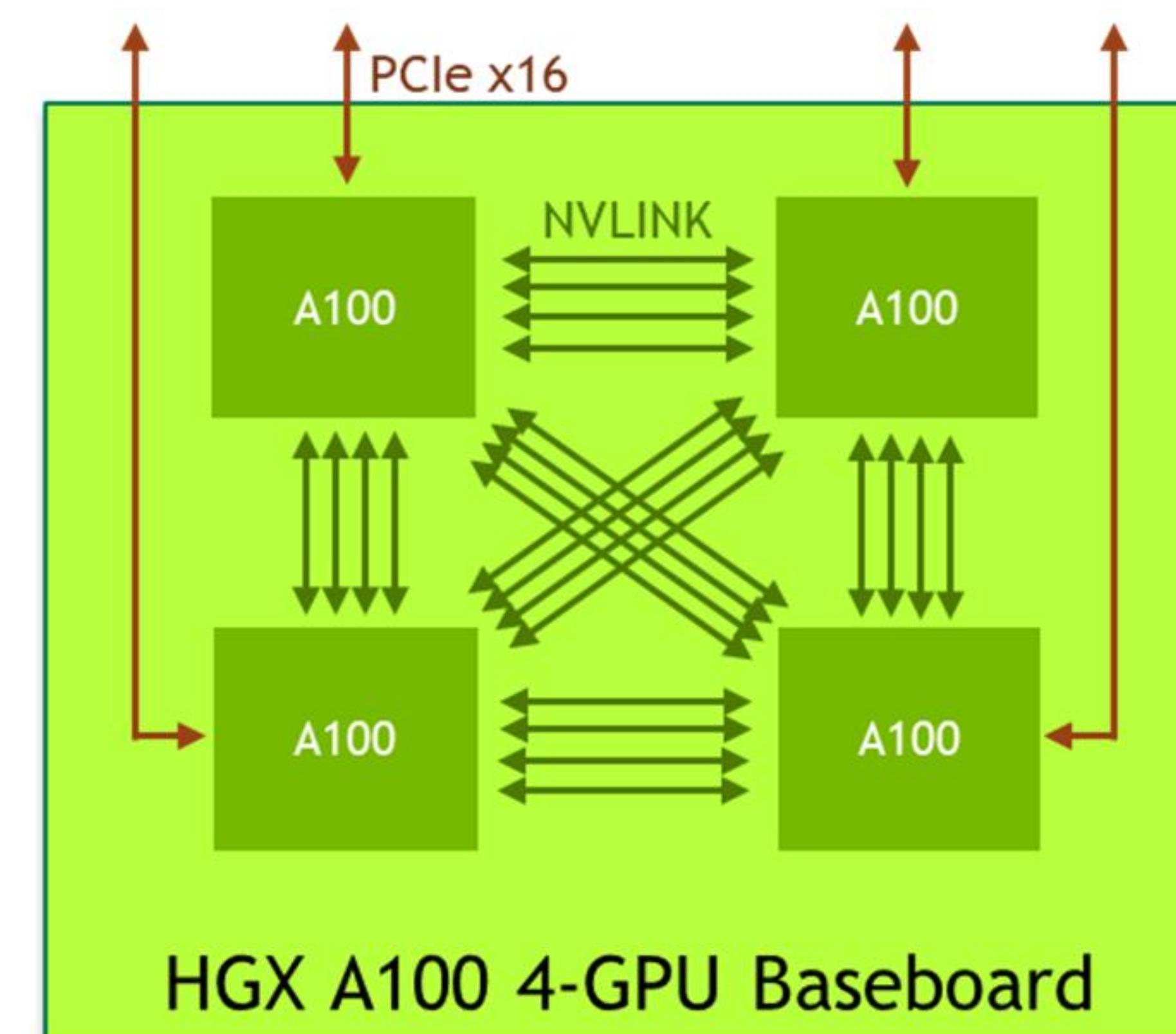
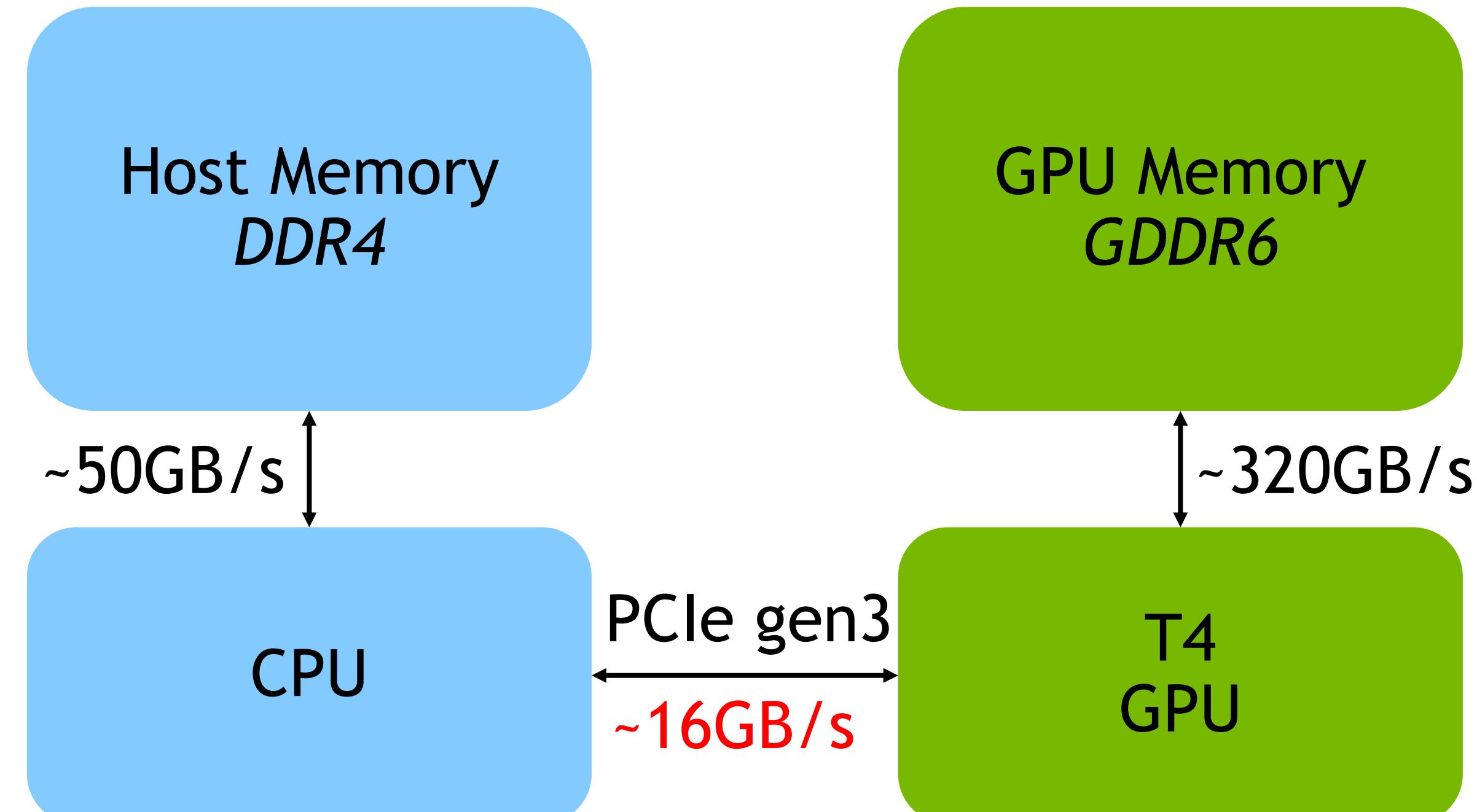
- What happens on the GPU, stays on the GPU!
- PCI express is a bottleneck for data movement
- Try NVLink for GPU peer-to-peer, 600 GB/s!

- **GPUs are parallel processing machines**

- Leave serial operations to the CPU
- Look for high arithmetic intensity, chunky loops, dense linear algebra
- Experiment with reduced precision, mixed-precision iterative refinement
- High memory bandwidth - Fast FFTs.

- **Stand on the Shoulders of Those Before You!**

- There is a rich ecosystem of GPU-accelerated libraries  
<https://developer.nvidia.com/gpu-accelerated-libraries>
- Profiling tools (Nsight) are compatible with Python GPU tools  
*We care about performance - make a relevant test suite!*
- Many applications are already GPU-accelerated
- <https://www.nvidia.com/en-us/gpu-accelerated-applications/>
- <https://ngc.nvidia.com/>



# AGENDA

Overview of GPU Computing

GPU-Accelerated Numerical Computing with *CuPy*

GPU-Accelerated Data Science with *RAPIDS*

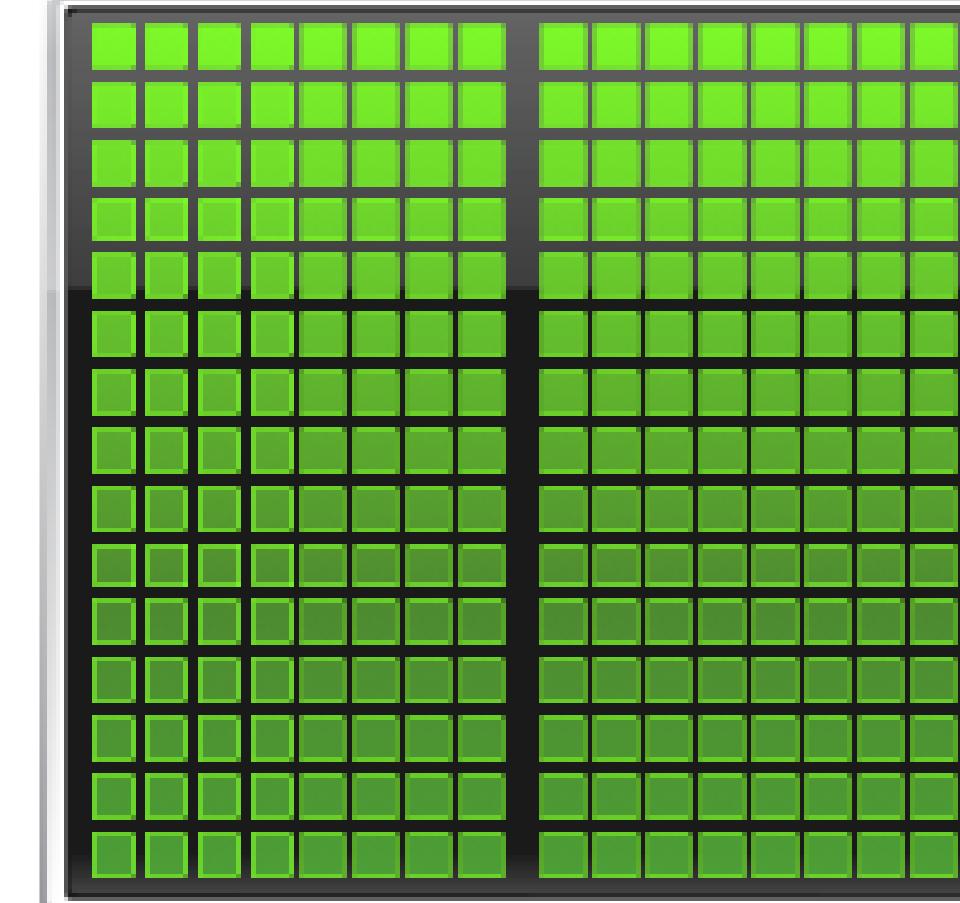
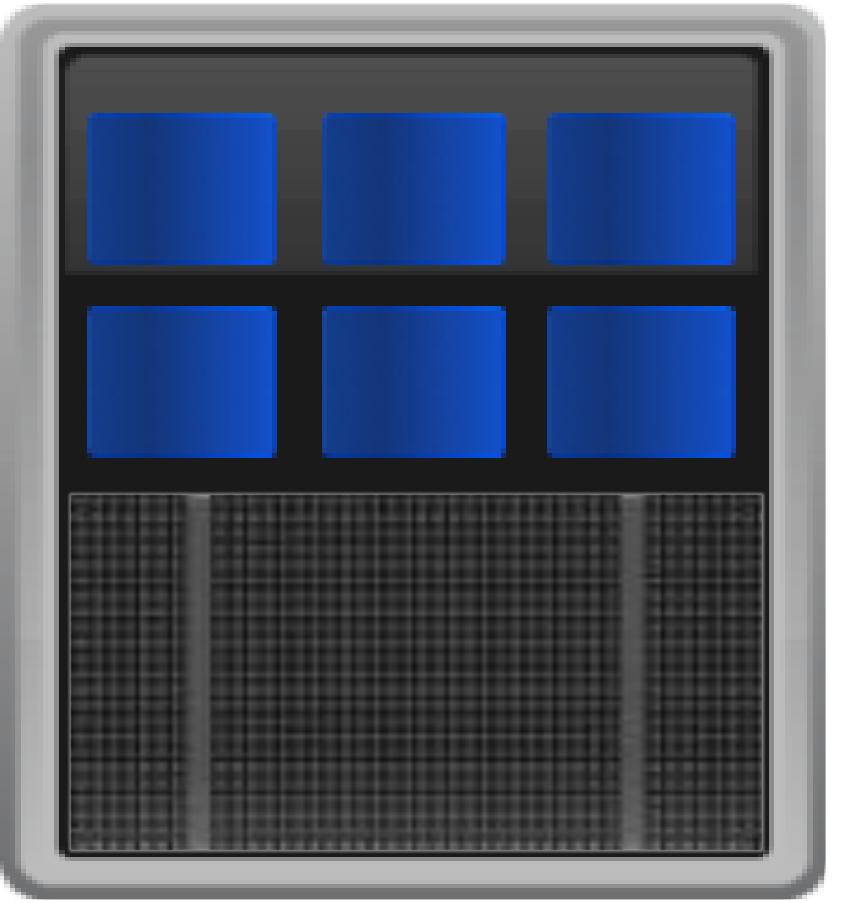
Custom GPU Kernels with *Numba*

Frameworks Interoperability - *Data Conversion Bottleneck*

ZERO-COPY end-to-end pipeline - try out on *NVIDIA Brev*



# NUMERICAL COMPUTING IN PYTHON



- Mathematical focus
- Operates on arrays of data
  - *ndarray*, holds data of same type
- Many years of development
- Highly tuned for CPUs

- NumPy like interface
- Trivially port code to GPU
- Copy data to GPU
  - CuPy *ndarray*
- Data interoperability with DL frameworks, RAPIDS, and Numba
- Uses high tuned NVIDIA libraries
- Can write custom CUDA functions

# CUPY

A NumPy like interface to GPU-acceleration ND-Array operations

**BEFORE**

```
import numpy as np  
  
size = 4096  
A = np.random.randn(size,size)  
  
Q, R = np.linalg.qr(A)
```

**AFTER**

```
import cupy as cp  
  
size = 4096  
A = cp.random.randn(size,size)  
  
Q, R = cp.linalg.qr(A)
```



*52x Speedup!*



CuPy

# AGENDA

Overview of GPU Computing

---

GPU-Accelerated Numerical Computing with *CuPy*

---

GPU-Accelerated Data Science with *RAPIDS*

---

Custom GPU Kernels with *Numba*

---

Frameworks Interoperability - *Data Conversion Bottleneck*

---

ZERO-COPY end-to-end pipeline - try out on *NVIDIA Brev*

---



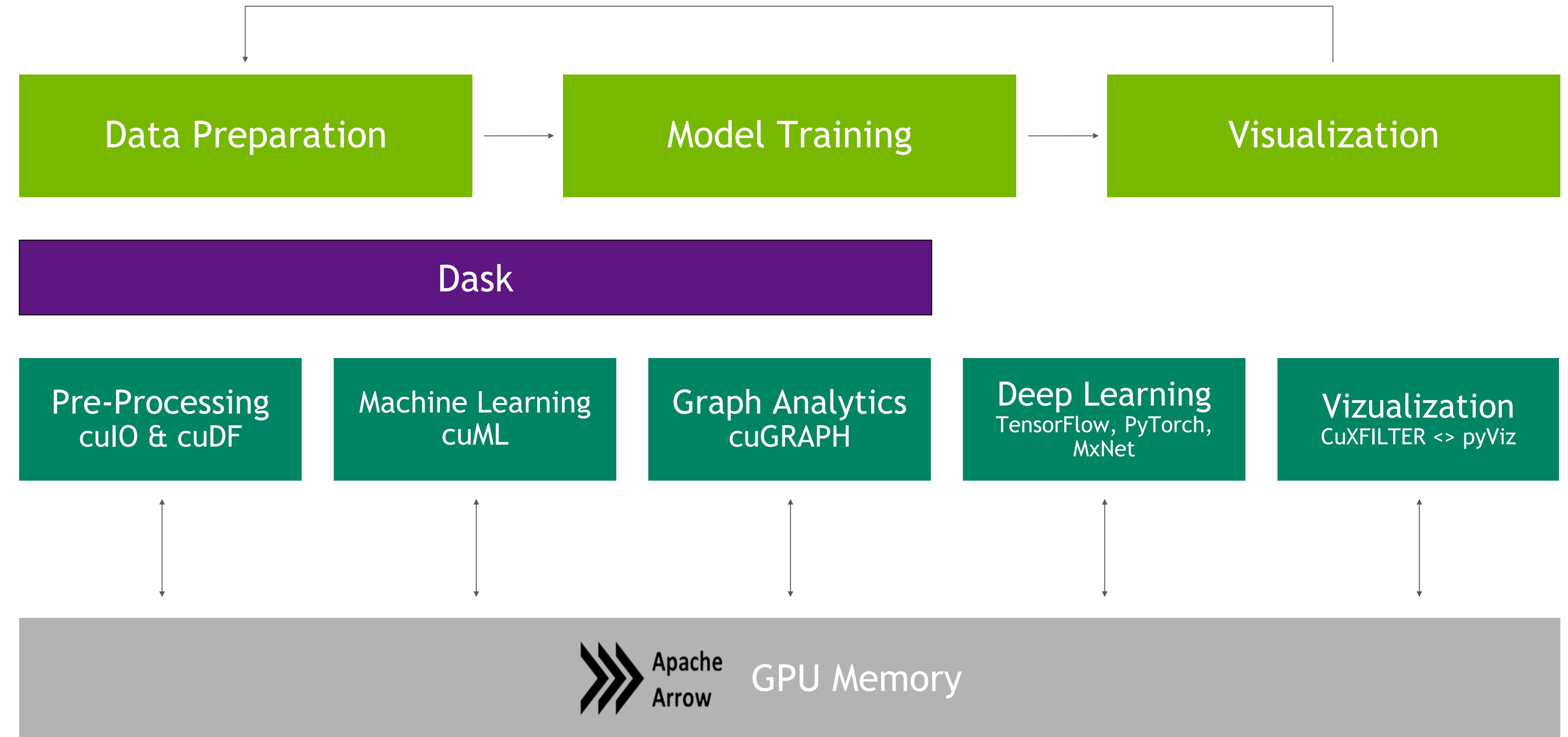
# RAPIDS ACCELERATES POPULAR DATA SCIENCE TOOLS

Delivering enterprise-grade data science solutions in pure python

The [RAPIDS](#) suite of open-source software libraries gives you the freedom to execute end-to-end data science and analytics pipelines entirely on GPUs.

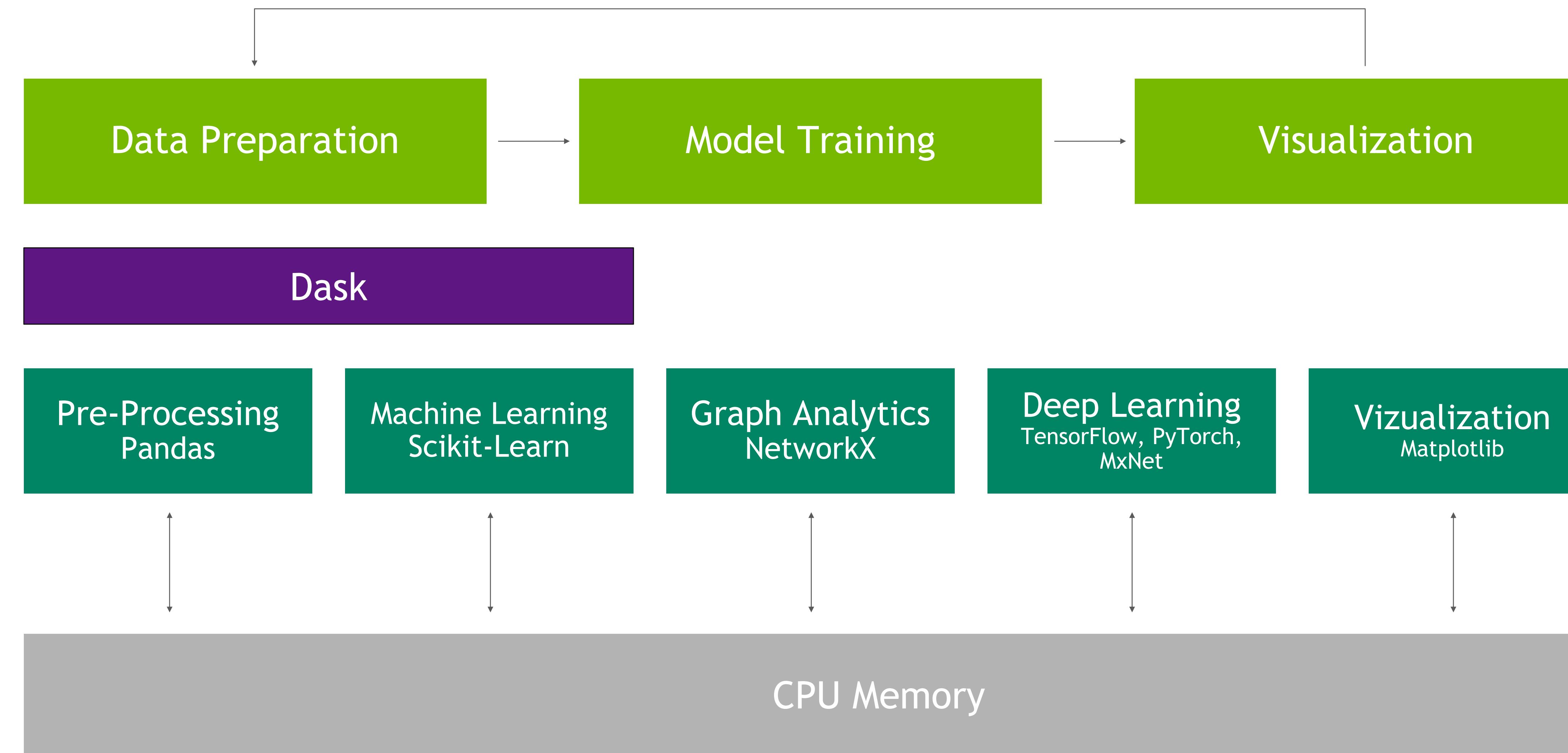
RAPIDS utilizes **NVIDIA CUDA** primitives for low-level compute optimization and exposes GPU parallelism and high-bandwidth memory speed through user-friendly Python interfaces like PyData.

With Dask, RAPIDS can scale out to multi-node, multi-GPU cluster to power through big data processes.

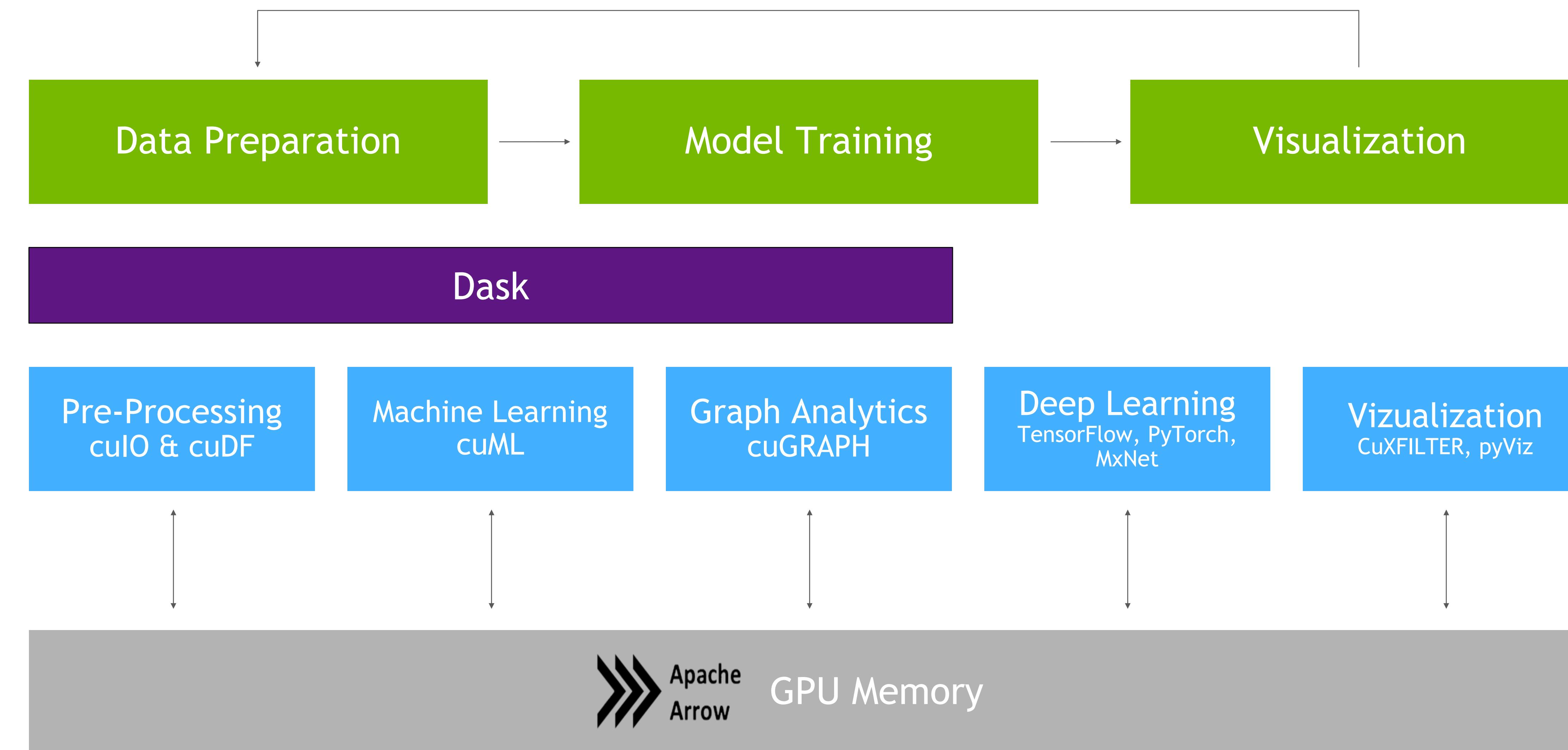


***RAPIDS enables the Python stack with the power of NVIDIA GPUs***

# TRADITIONAL DATA SCIENCE APPLICATIONS



# RAPIDS: GPU-ACCELERATED DATA SCIENCE WITH API ALIGNMENT



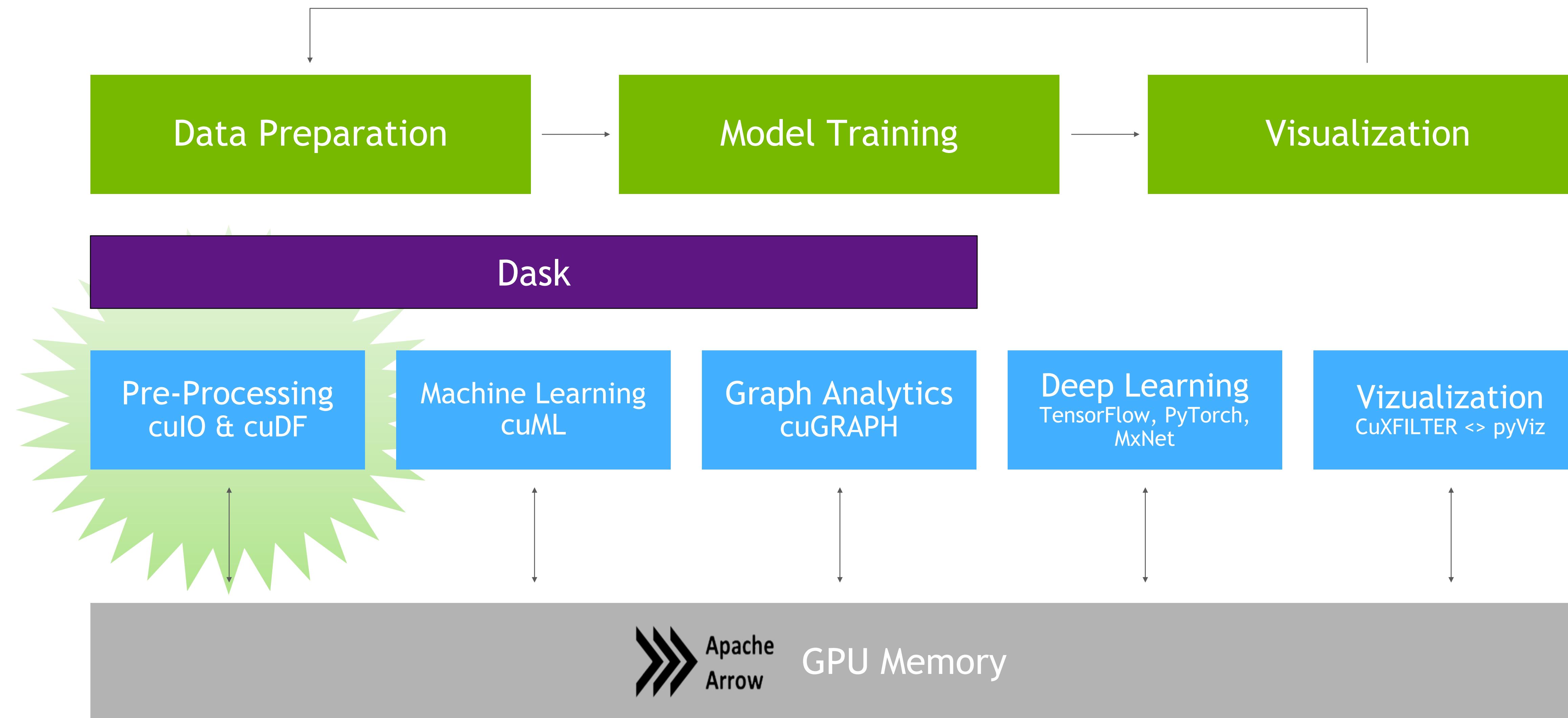
# DATA SCIENCE API ALIGNMENT

Open-source software that accelerates popular data science packages

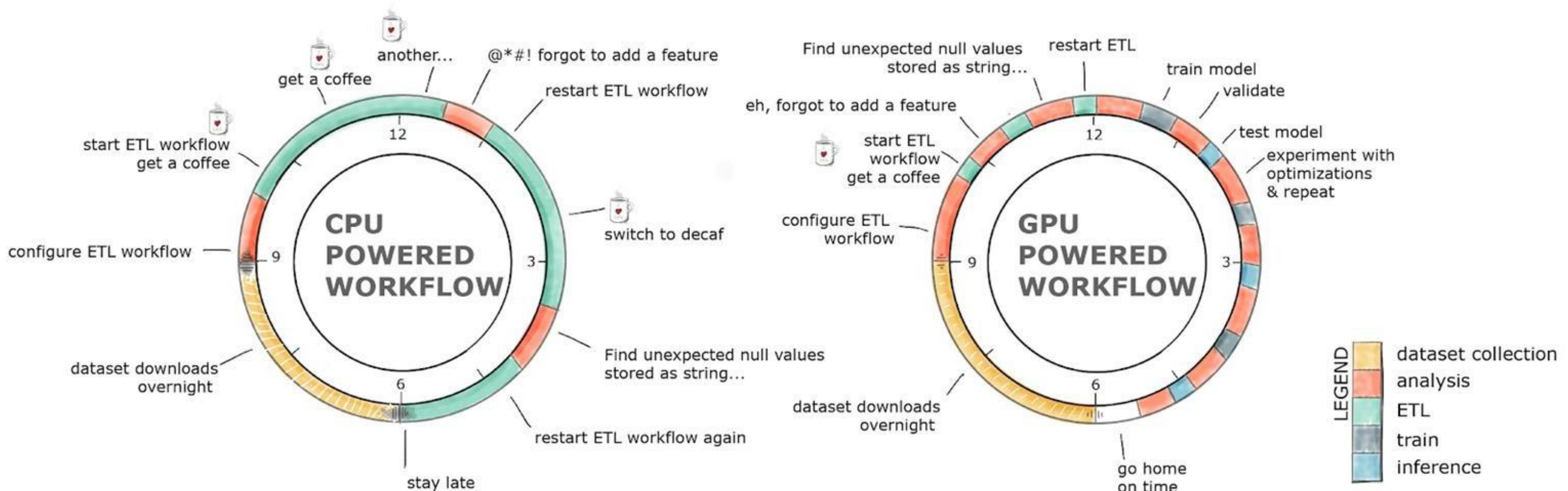
Function	CPU	GPU/RAPIDS
Data handling	pandas	cuDF **
Machine learning	scikit-learn	cuML **
Graph analytics	NetworkX	cuGraph
Geospatial	GeoPandas/SciPy	cuSpatial
Signals	SciPy.signal	cuSignal
Image Processing	scikit-image	cuCIM

The RAPIDS and GPU-accelerated PyData stack bring GPGPU to data scientists at the Python layer providing familiar APIs without the steep curve of learning new programming language or paradigm

# RAPIDS: GPU-ACCELERATED DATA SCIENCE WITH API ALIGNMENT



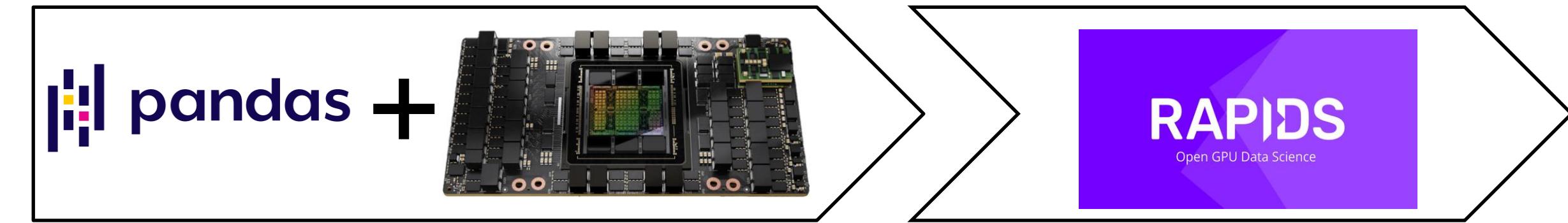
# THE BURDEN OF DATA PROCESSING: EXTRACT, TRANSFORM, LOAD



*The Average Data Scientist Spends 90% of Their Time in ETL as Opposed to Training Models*

# GPU-ACCELERATED PANDAS WITH CUDF

- Use RAPIDS CuDF to accelerate computationally expensive ETL operations
- Manipulate GPU DataFrames following the Pandas API
- Create GPU DataFrames from Numpy arrays, CuPy arrays, Pandas DataFrames, and PyArrow Tables
- Python interface to CUDA C++ library with additional functionality
- Available via pip and conda



```
import cudf as pd
import numpy as np
from time import time

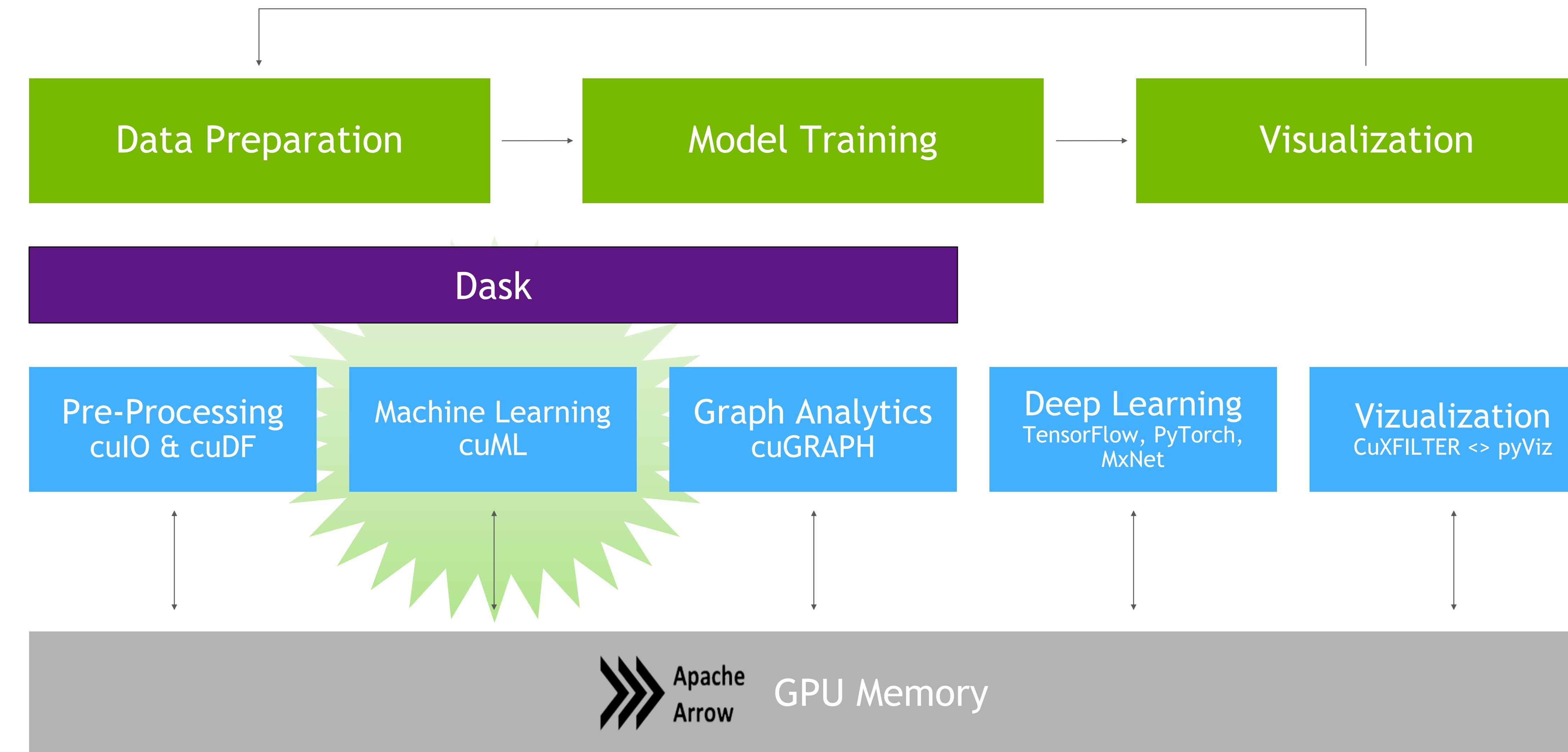
import matplotlib.pyplot as plt
import seaborn as sns

%matplotlib inline

wine_set = pd.read_csv("data/winequality.csv")

wine_set.head(n=5)
wine_set.tail(n=5)
```

# RAPIDS: GPU-ACCELERATED DATA SCIENCE WITH API ALIGNMENT



# DATASET SIZES CONTINUE TO GROW

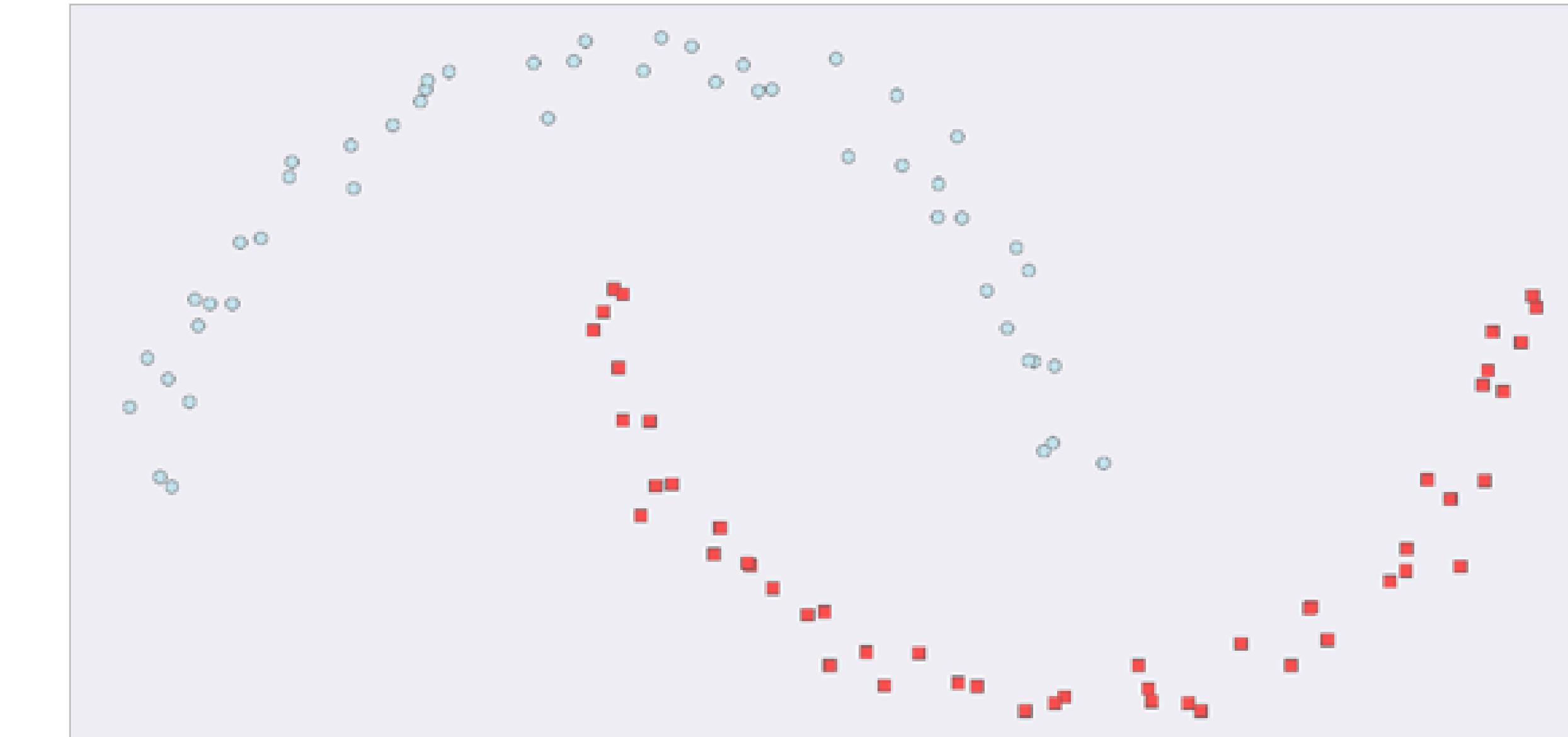
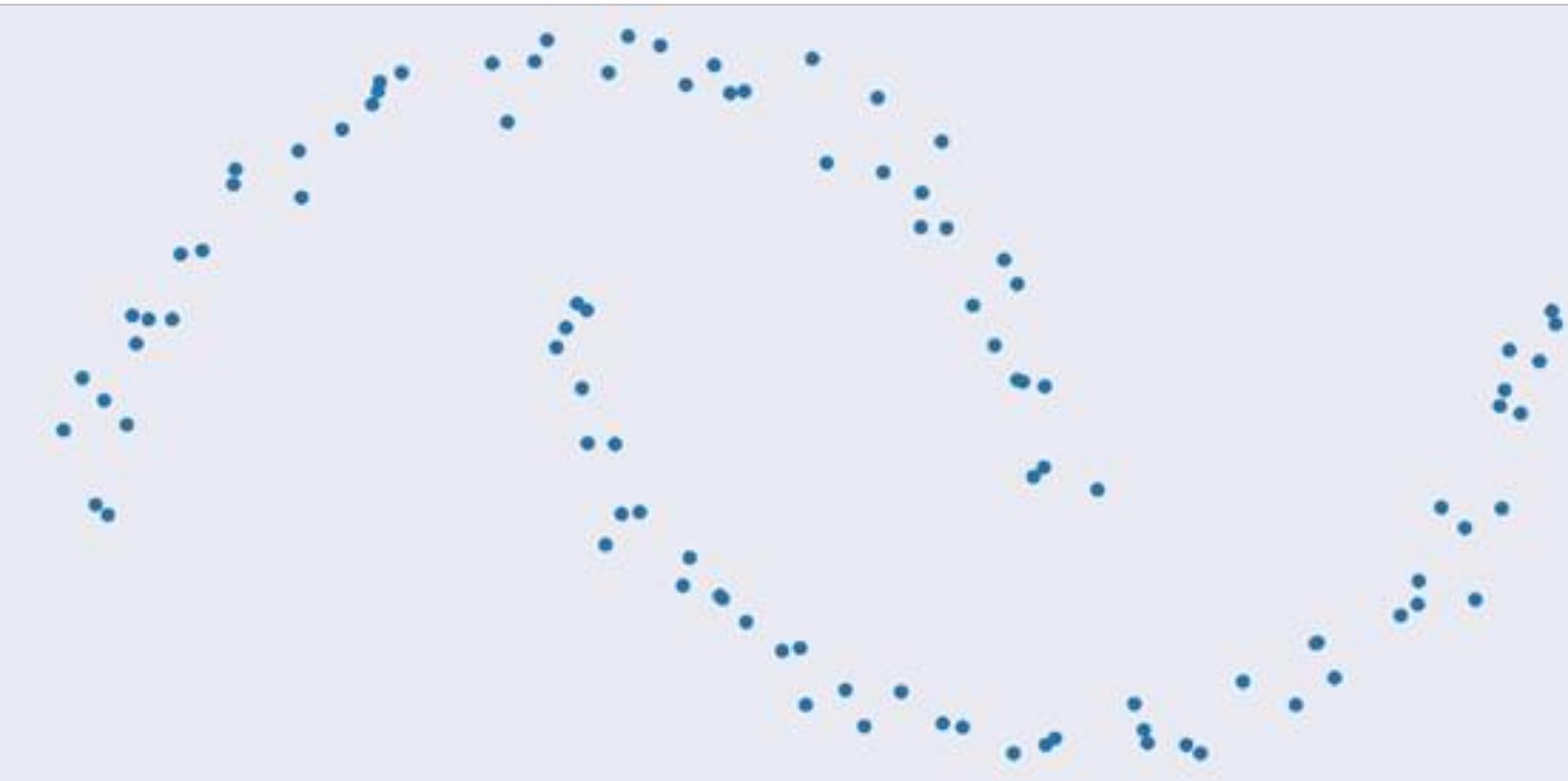
```
from sklearn.datasets import make_moons
import pandas

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

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

```
from sklearn.cluster import DBSCAN
dbscan = DBSCAN(eps = 0.3, min_samples = 5)

y_hat = dbscan.fit_predict(X)
```



# DATASET SIZES CONTINUE TO GROW

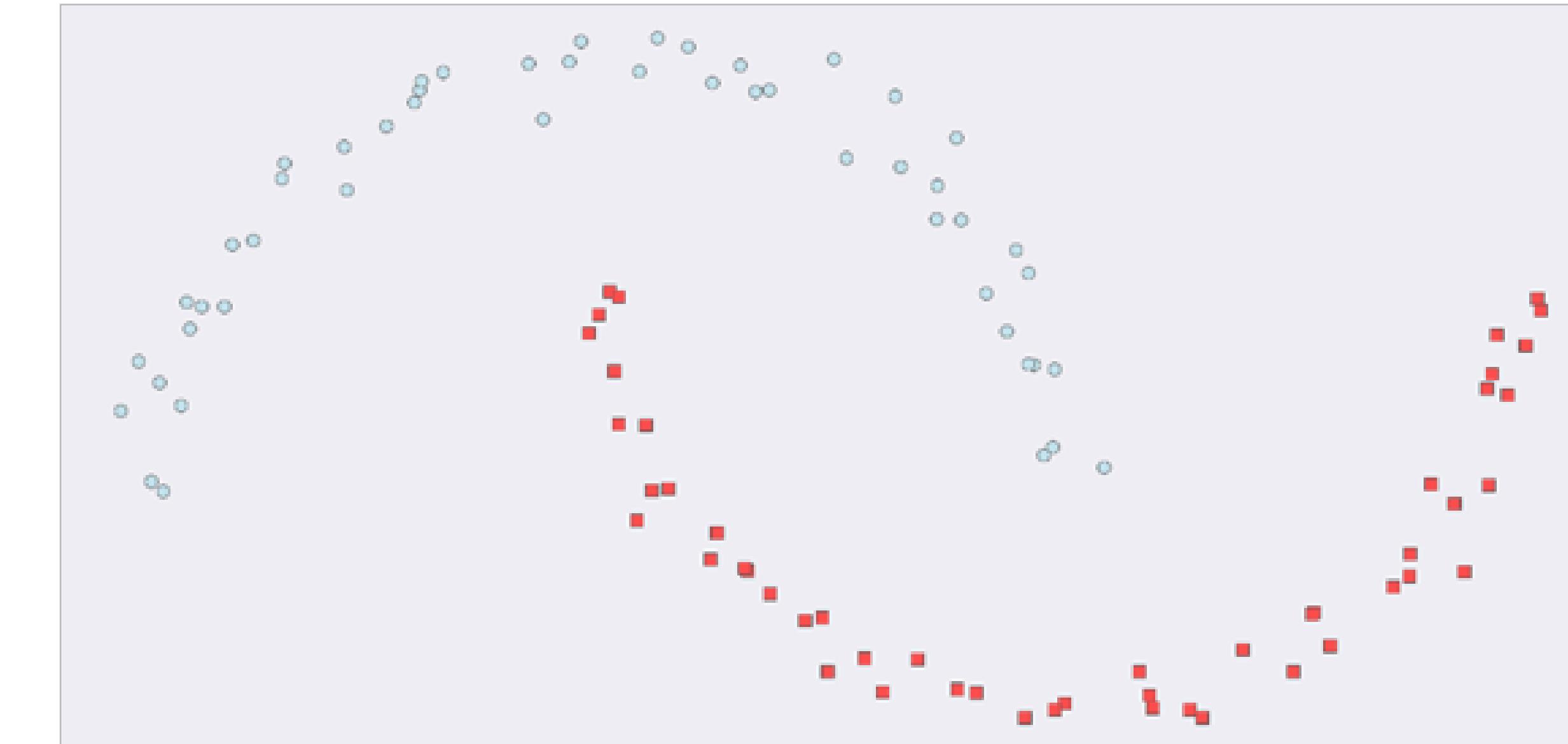
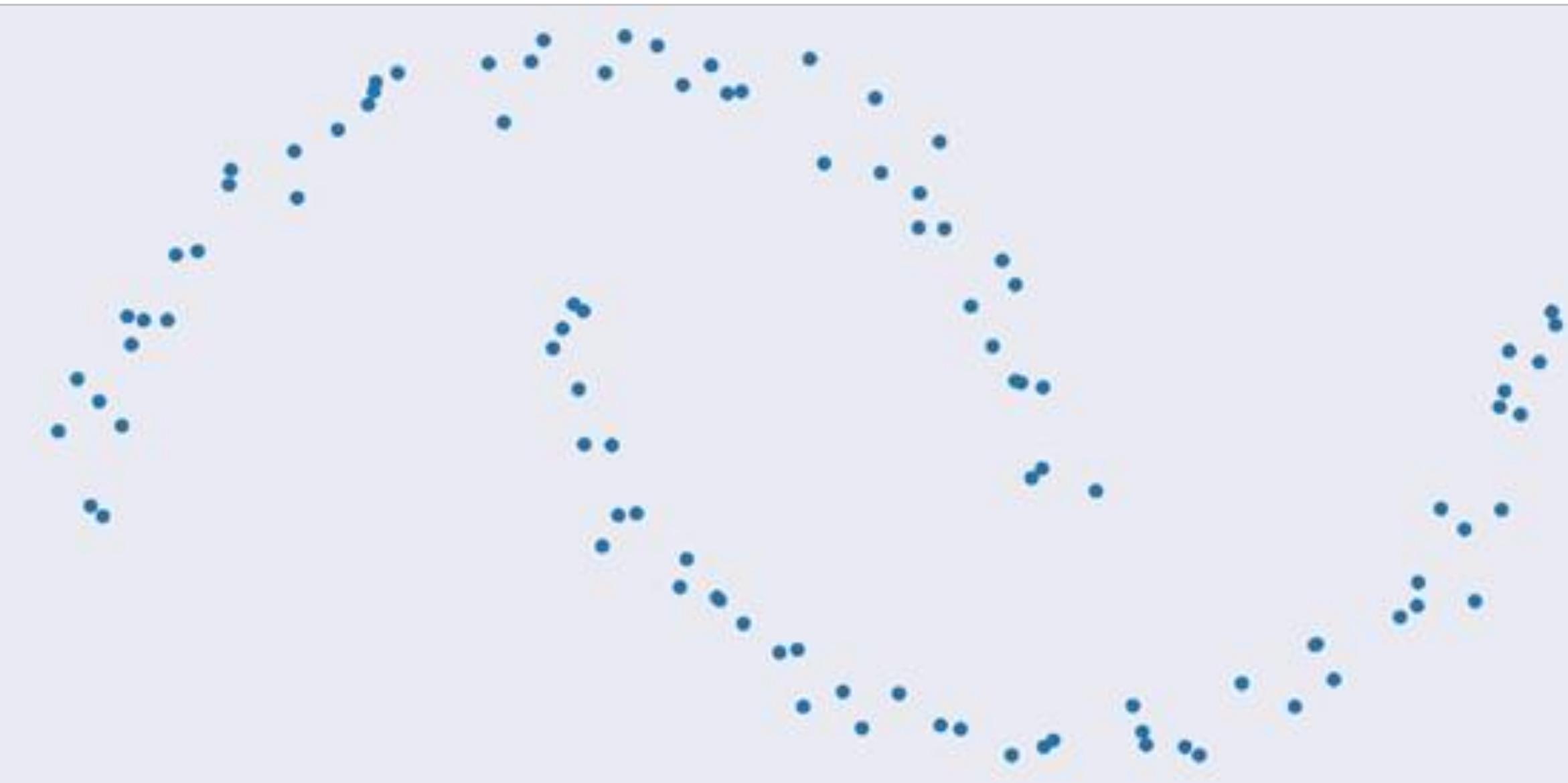
```
from sklearn.datasets import make_moons
import cudf

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

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

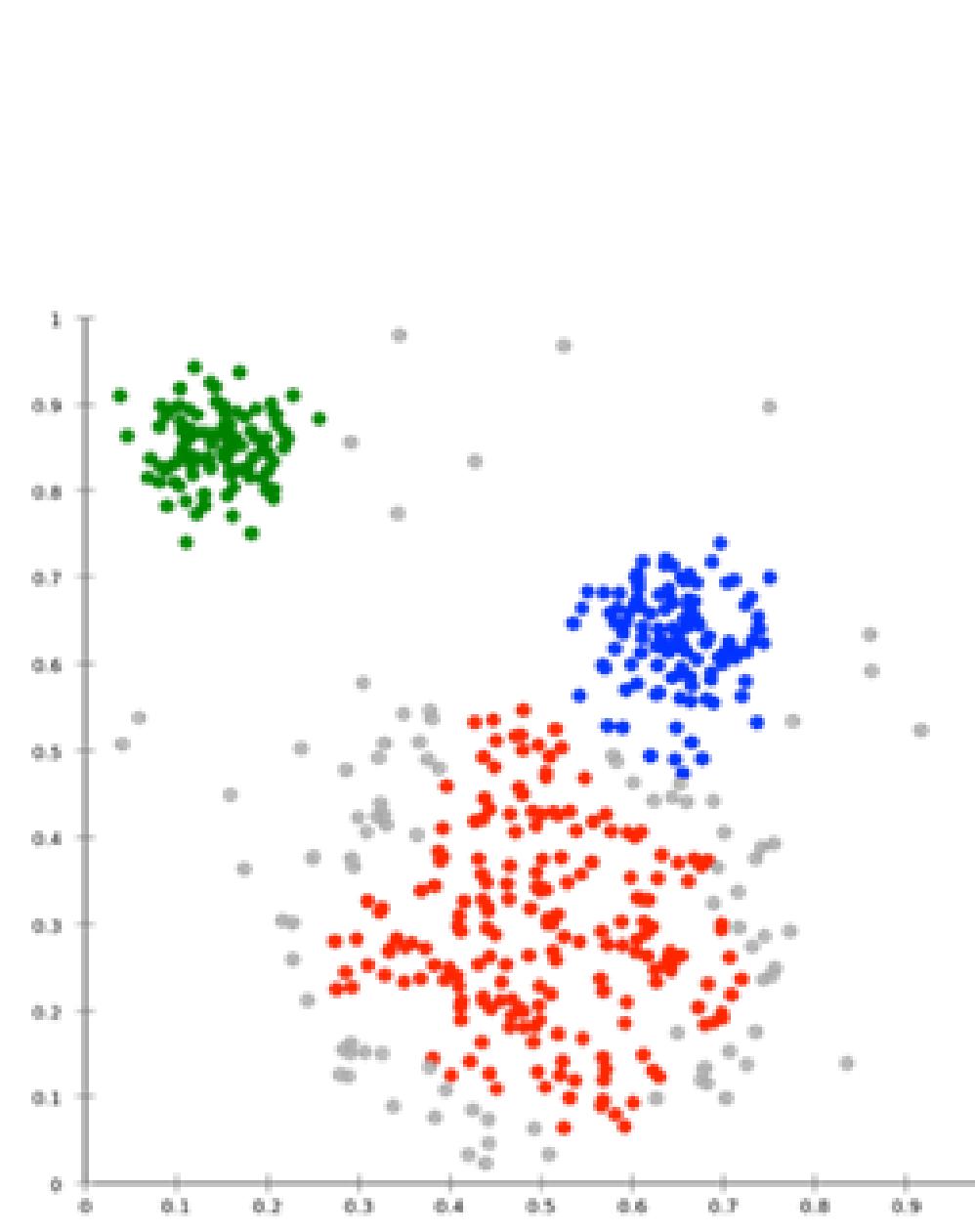
```
from cuml import DBSCAN
dbscan = DBSCAN(eps = 0.3, min_samples = 5)

y_hat = dbscan.fit_predict(X)
```



# CUML ALGORITHMS

Classification / Regression
Decision Trees / <b>Random Forests</b>
Linear/Lasso/Ridge/LARS/ElasticNet Regression
Logistic Regression
K-Nearest Neighbors ( <b>exact or approximate</b> )
Support Vector Machine Classification and Regression
Naive Bayes



Clustering Decomposition Dimensionality Reduction
K-Means
<b>DBSCAN</b>
Spectral Clustering
Principal Components (including iPCA)
Singular Value Decomposition
UMAP
Spectral Embedding T-SNE

Inference
Random Forest / GBDT Inference (FIL)

Time Series
Holt-Winters
Seasonal ARIMA / Auto ARIMA

Preprocessing
Text vectorization (TF-IDF / Count)
Target Encoding
Cross-validation / splitting

Hyper-parameter Tuning
Cross Validation

More to come!

# MONAI Core

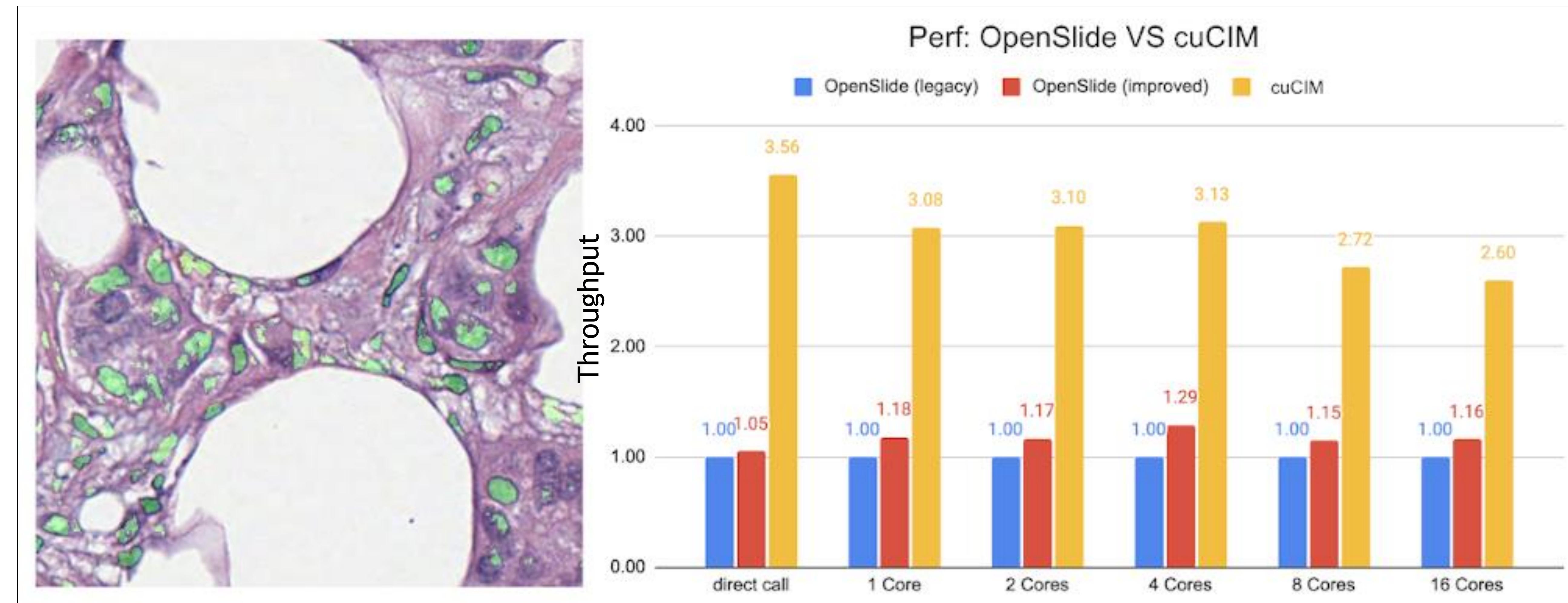
1. Medical imaging specific

2. Superior performance

3. Friendly community

## Optimize data loading

*cuCIM - Whole Slide Imaging (digital pathology)*



cuCIM - a library within [RAPIDS](#)

# MONAI Core

Optimize GPU utilization

*Do transforms on GPU*

1. Medical imaging specif

2. Superior performance

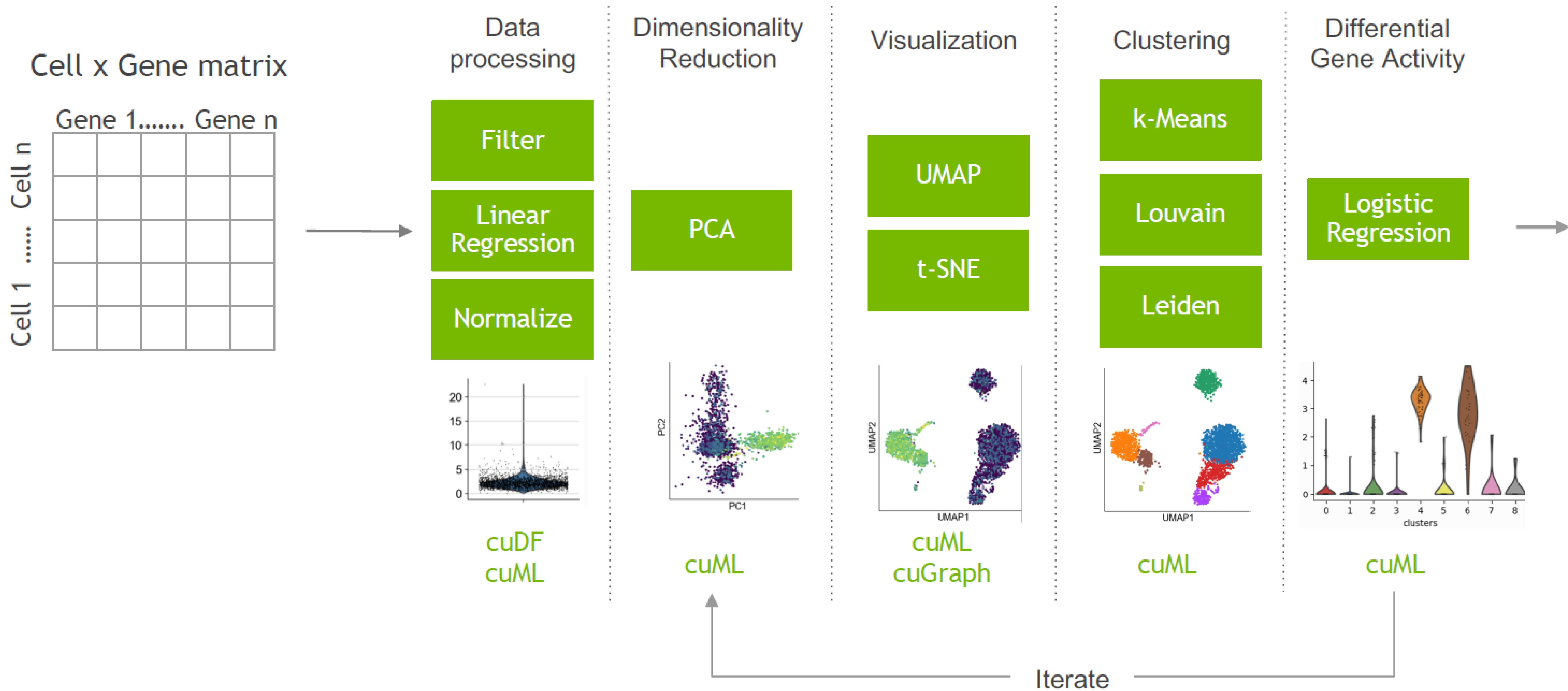
3. Friendly community

***cuCIM -> common transforms in digital pathology***

```
13 from monai.transforms import (
14     Activations,
15     AsDiscrete,
16     CastToType,
17     CastToTyped,
18     Compose,
19     CuCIM,
20     GridSplitd,
21     Lambdad,
22     RandCuCIM,
23     RandFlipd,
24     RandRotate90d,
25     RandZoomd,
26     ScaleIntensityRanged,
27     ToCopy,
28     ToNumpyd,
29     TorchVisiond,
30     ToTensor,
31     ToTensord,
32 )
```

[MONAI Core pathology tutorials](#)

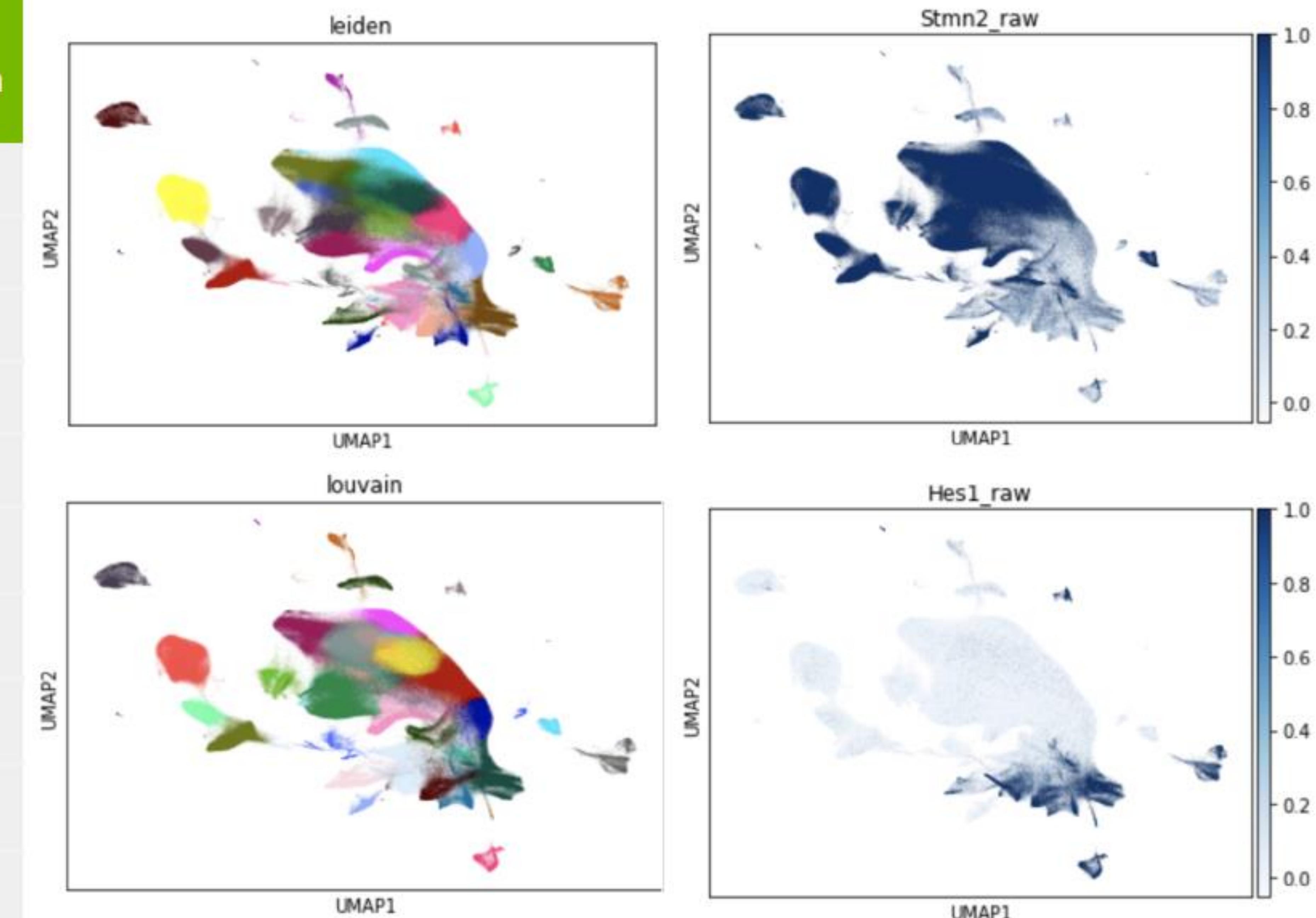
# SINGLE-CELL RNA-SEQ ANALYSIS USING RAPIDS



# GPU ANALYSIS OF 1 MILLION CELLS

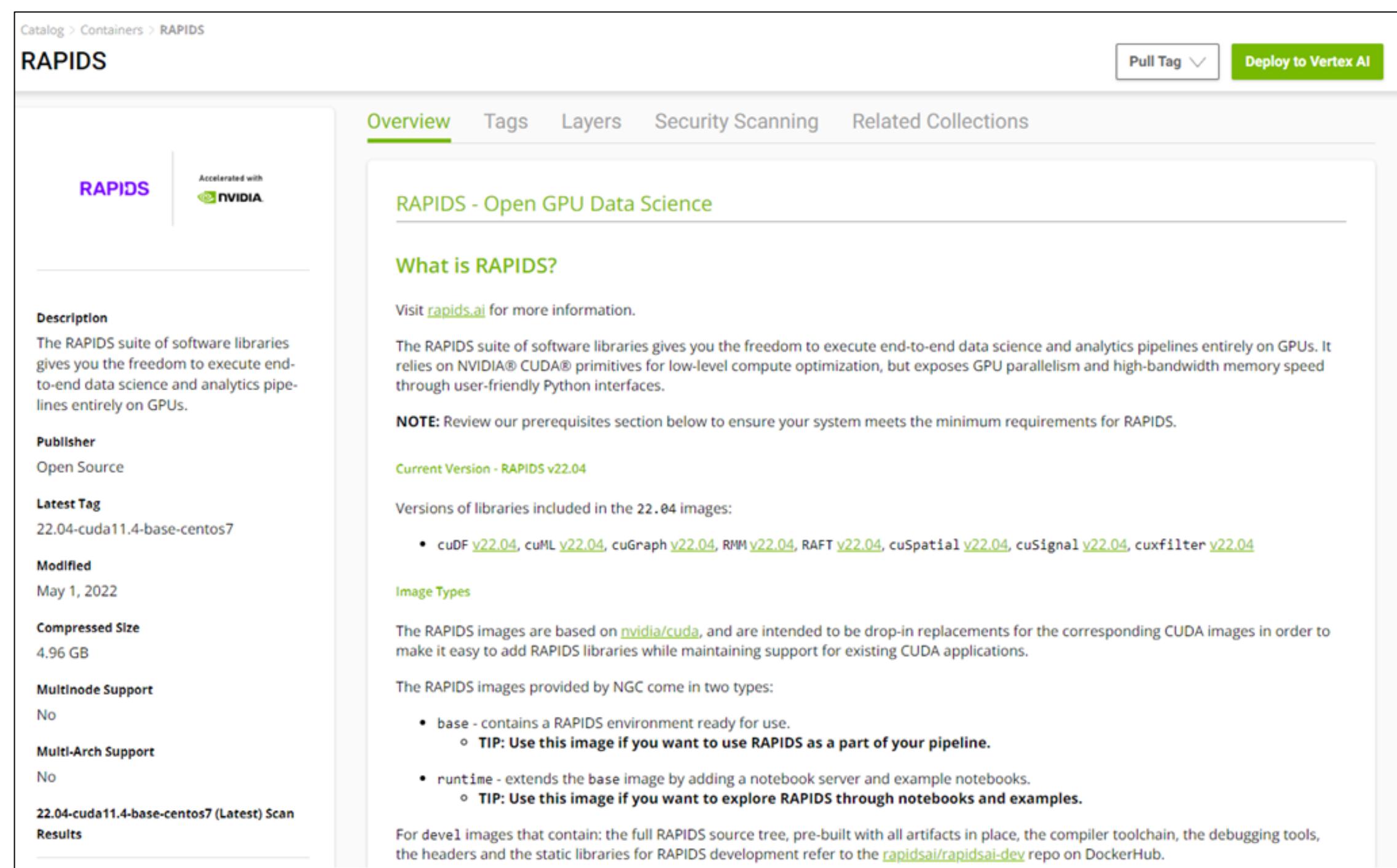
From 3.5 hours to 8 minutes

	CPU Runtime n1-highmem-32 32 vCPUs	GPU runtime a2-highgpu-1g Tesla A100 40GB GPU	GPU acceleration
Preprocessing	28m35s	3m21s	9x
PCA	29.2s	11.4s	2.6x
t-SNE	1hr23m10s	28s	178x
KNN	3m5s	46s	4x
UMAP	21m47s	13.4s	98x
k-means clustering	2m6s	1.9s	66x
Louvain clustering	15m5s	1.9s	476x
Leiden clustering	51m1s	1.4s	2186x
End-to-end runtime	3hr31m48s	8m22s	25x
End-to-end cost	\$6.682	\$0.553	



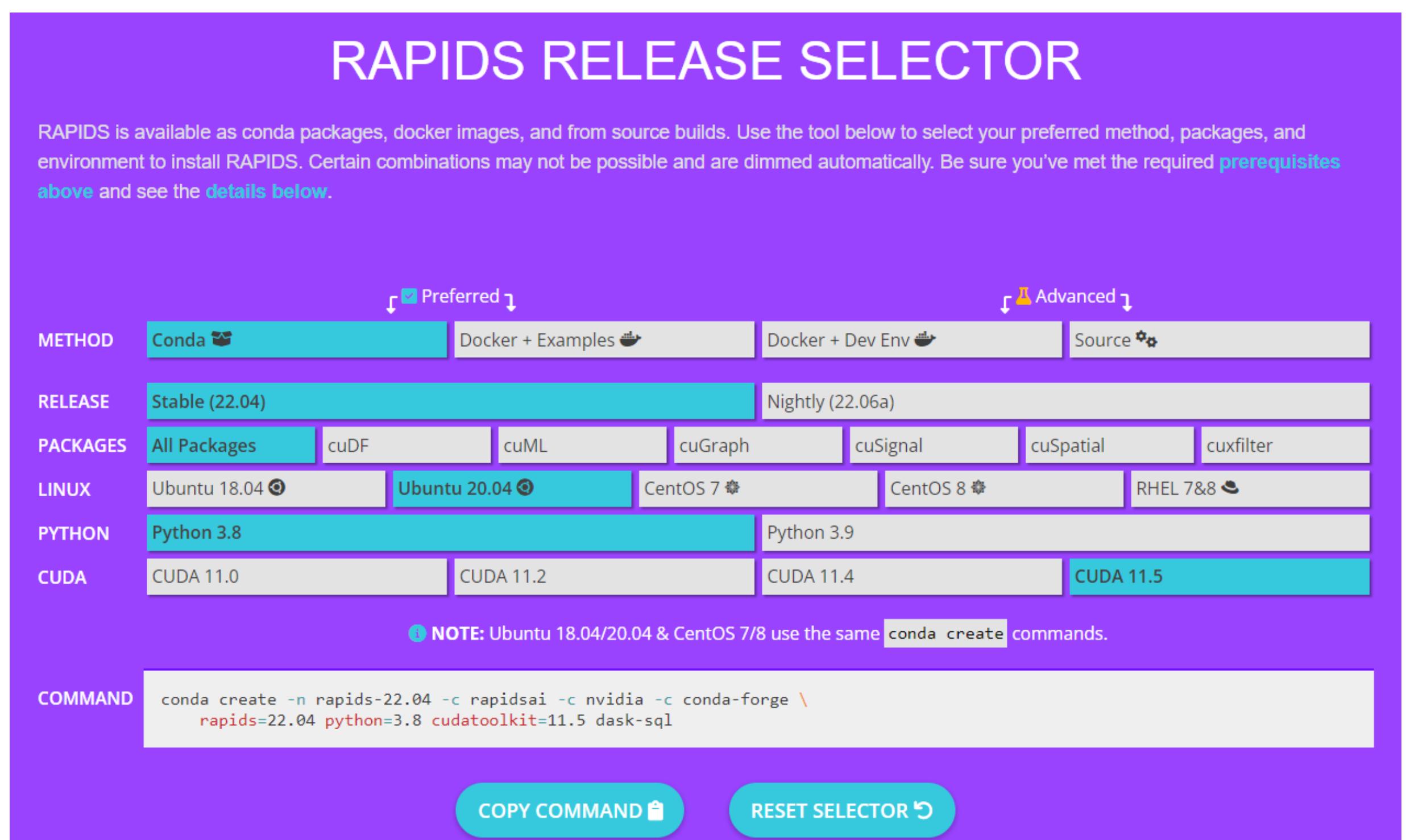
# INSTALLATION

NVIDIA NGC  
RAPIDS Container  
<https://ngc.nvidia.com>



The screenshot shows the NGC Catalog interface for the RAPIDS container. It includes sections for Overview, Tags, Layers, Security Scanning, and Related Collections. The Overview section provides a brief description of RAPIDS as a suite of software libraries for GPU data science, mentions it's accelerated by NVIDIA, and links to the RAPIDS GitHub repository. It also shows the latest tag (22.04-cuda11.4-base-centos7), modified date (May 1, 2022), compressed size (4.96 GB), and multi-node support (No). The Layers section lists components like cuDF, cuML, cuGraph, cuSignal, cuSpatial, and cuXfilter.

RAPIDS Release Selector  
(conda, container, source)  
<https://rapids.ai>



The screenshot shows the RAPIDS Release Selector tool. It has a purple header with the title "RAPIDS RELEASE SELECTOR". Below the header, a note states that RAPIDS is available as conda packages, docker images, and from source builds. It includes a "Preferred" tab and an "Advanced" tab. The "METHOD" section shows Conda as the selected option. The "RELEASE" section shows Stable (22.04) as the selected option. The "PACKAGES" section shows All Packages as the selected option. The "LINUX" section shows Ubuntu 18.04 and Ubuntu 20.04 as selected. The "PYTHON" section shows Python 3.8 as the selected option. The "CUDA" section shows CUDA 11.0 as the selected option. A note at the bottom states: "NOTE: Ubuntu 18.04/20.04 & CentOS 7/8 use the same `conda create` commands." At the bottom are "COPY COMMAND" and "RESET SELECTOR" buttons.

Available on your campus cluster?

# AGENDA

Overview of GPU Computing

GPU-Accelerated Numerical Computing with *CuPy*

GPU-Accelerated Data Science with *RAPIDS*

Custom GPU Kernels with *Numba*

Frameworks Interoperability - *Data Conversion Bottleneck*

ZERO-COPY end-to-end pipeline - try out on *NVIDIA Brev*



# WHAT IS NUMBA? WHEN DO WE USE IT?

Lower-level CUDA kernel development without leaving Python

## Just-in-time compiler

Numba is a JIT compiler for Python functions that you specify. Numba targets both CPU and GPU.

## Opt-in

Numba only compiles functions you specify. You don't need to compile the full program

## PyData ecosystem

While not all functions in python can be compiled with Numba, the PyData ecosystem is well covered.

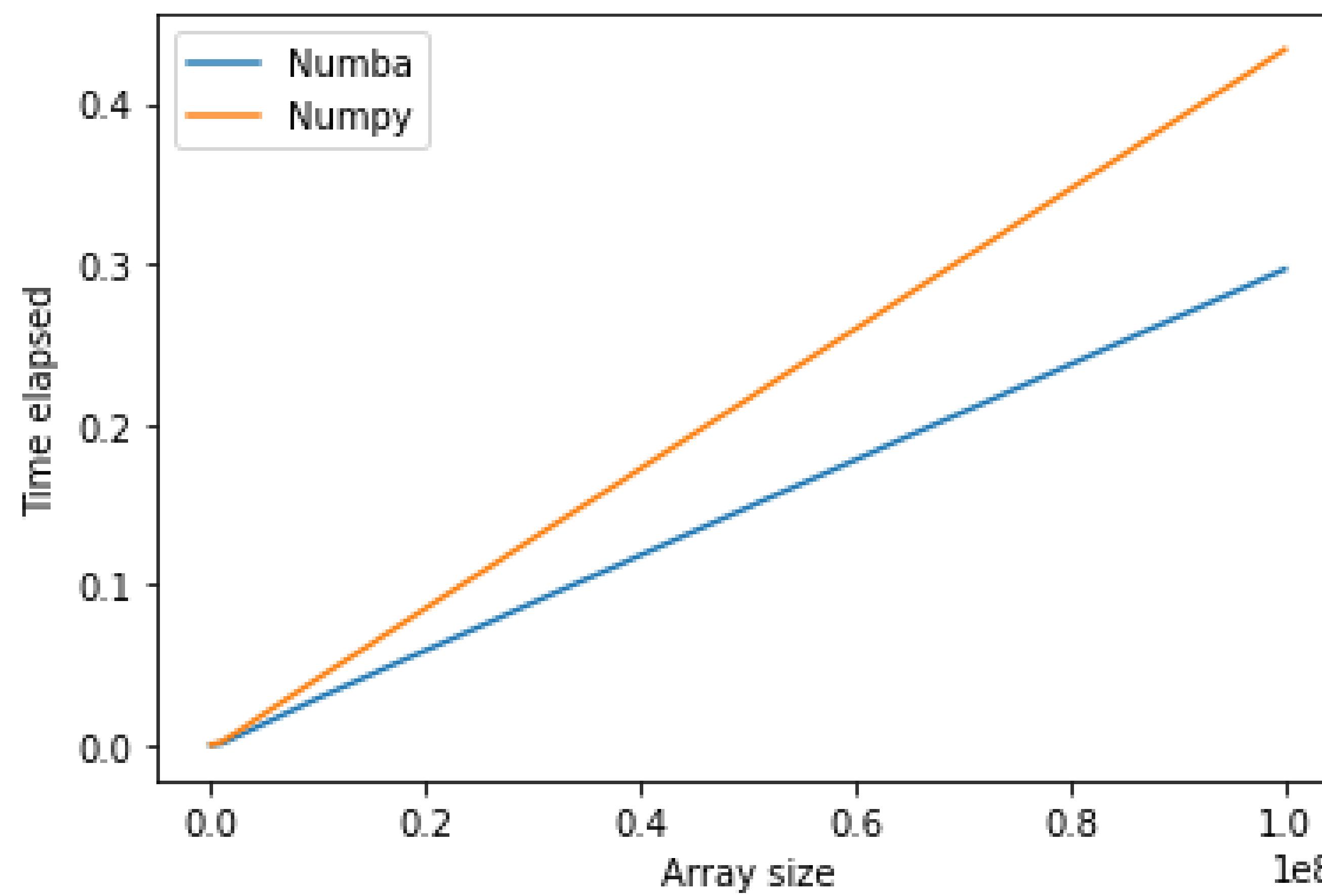
Numba provides the Python programmer a simple way to write customizable GPU accelerated code without needing CUDA C/C++

# NUMBA VECTORIZE

NumPy ufunc operates elementwise on arrays, Numba @vectorize allows us to accelerate those types of operations

```
from numba import vectorize
import numpy as np
import time

@vectorize
def rel_diff(x, y):
    return 2 * (x - y) / (x + y)
```



```
size_list = [1000, 10000, 100000, 1000000, 10000000,
100000000]

numpy_times = []
numba_times = []

for size in size_list:
    x=np.random.randn(size).astype(np.float32) + 1
    y=np.random.randn(size).astype(np.float32) + 1.1

    # Run baseline Numpy implementation
    2 * (x - y) / (x + y)

    # Run our vectorized Numba function
    rel_diff(x, y)

    numpy_times.append(time.time() - start)
    numba_times.append(time.time() - start)
```

With this "vectorized" Numba function we see improved performance as we increase our input size, making this solution ideal for large problem sizes.

# NUMBA CUDA

Lower-level CUDA kernel development without leaving Python

## BEFORE

```
import numba

@jit()
def vector_add(arr1, arr2):

    arr_size = arr1.shape[0]
    result = np.empty(size=(arr_size))

    for i in prange(arr_size):
        result[i] = arr1[i] + arr2[i]

    return result
```

## AFTER

```
import numba

@cuda.jit()
def vector_add(arr1, arr2, result):

    startx = cuda.grid(1)
    stridex = cuda.gridsize(1)

    arr_size = arr1.shape[0]

    for i in range(startx, arr_size, stridex):
        result[i] = arr1[i] + arr2[i]
```

- Initialize data or copy data to GPU
- Lower-level support for custom CUDA kernels without C/C++
- JIT compiled kernels for fast execution
- Move data between DL frameworks, RAPIDS, and Numba



# SUMMARY

Function	CPU	GPU/RAPIDS
Data handling	pandas	cuDF
Machine learning	scikit-learn	cuML
Function	CPU	GPU
Numerical Computing	NumPy	CuPy
JIT Kernels	Numba	Numba

# NVIDIA DEEP LEARNING INSTITUTE

Self-paced courses

Instructor-led workshops

RAPIDS

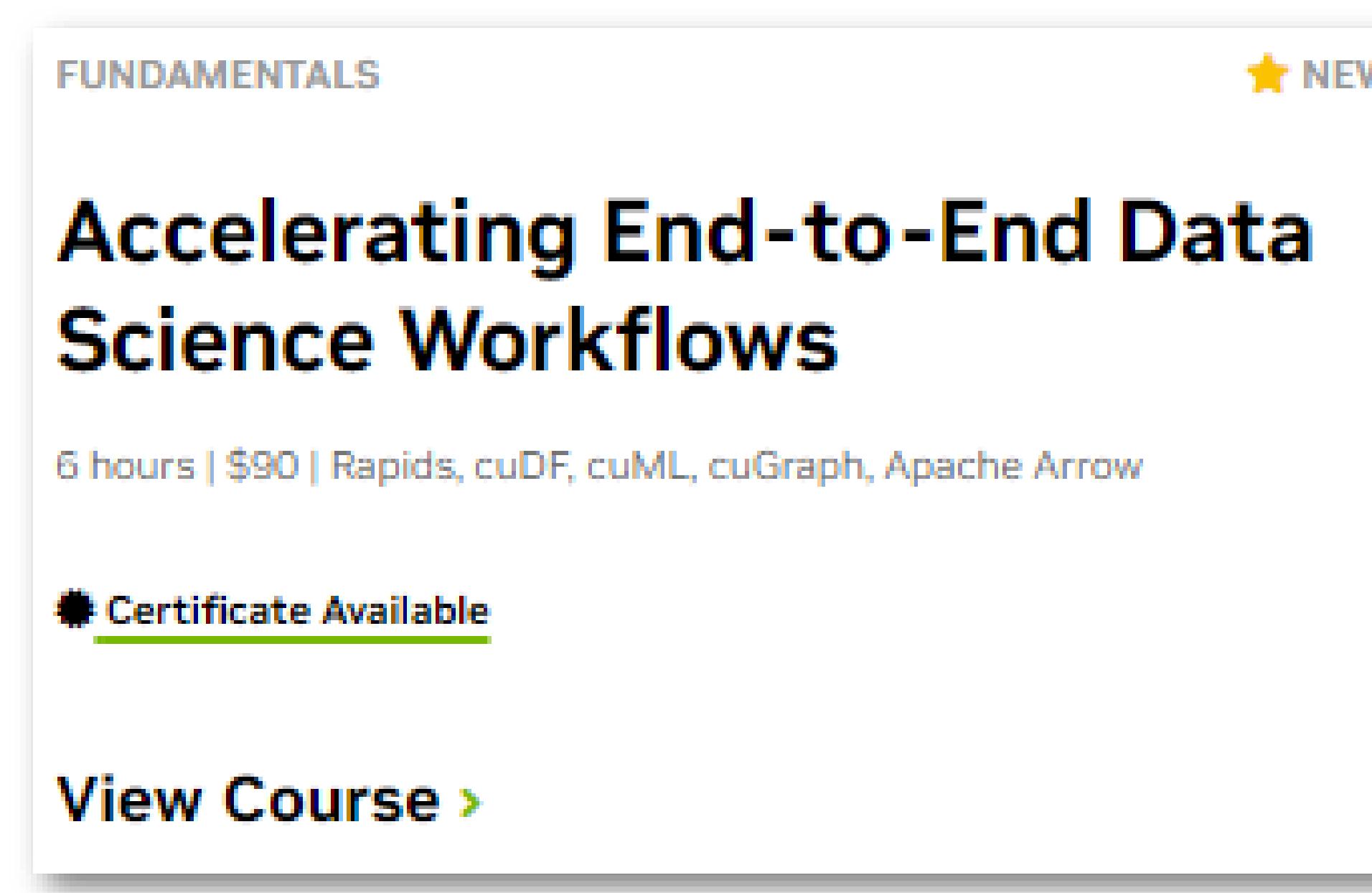
FUNDAMENTALS      ★ NEW

**Accelerating End-to-End Data Science Workflows**

6 hours | \$90 | Rapids, cuDF, cuML, cuGraph, Apache Arrow

● Certificate Available

[View Course >](#)

A course card for "Accelerating End-to-End Data Science Workflows". It features the RAPIDS logo at the top left, followed by the word "FUNDAMENTALS" and a yellow star icon with the text "NEW" next to it. The main title is "Accelerating End-to-End Data Science Workflows" in bold black font. Below the title is a description: "6 hours | \$90 | Rapids, cuDF, cuML, cuGraph, Apache Arrow". A small circular icon with a dot and a line is followed by the text "Certificate Available" underlined. At the bottom is a blue button with white text: "View Course >".

Numba

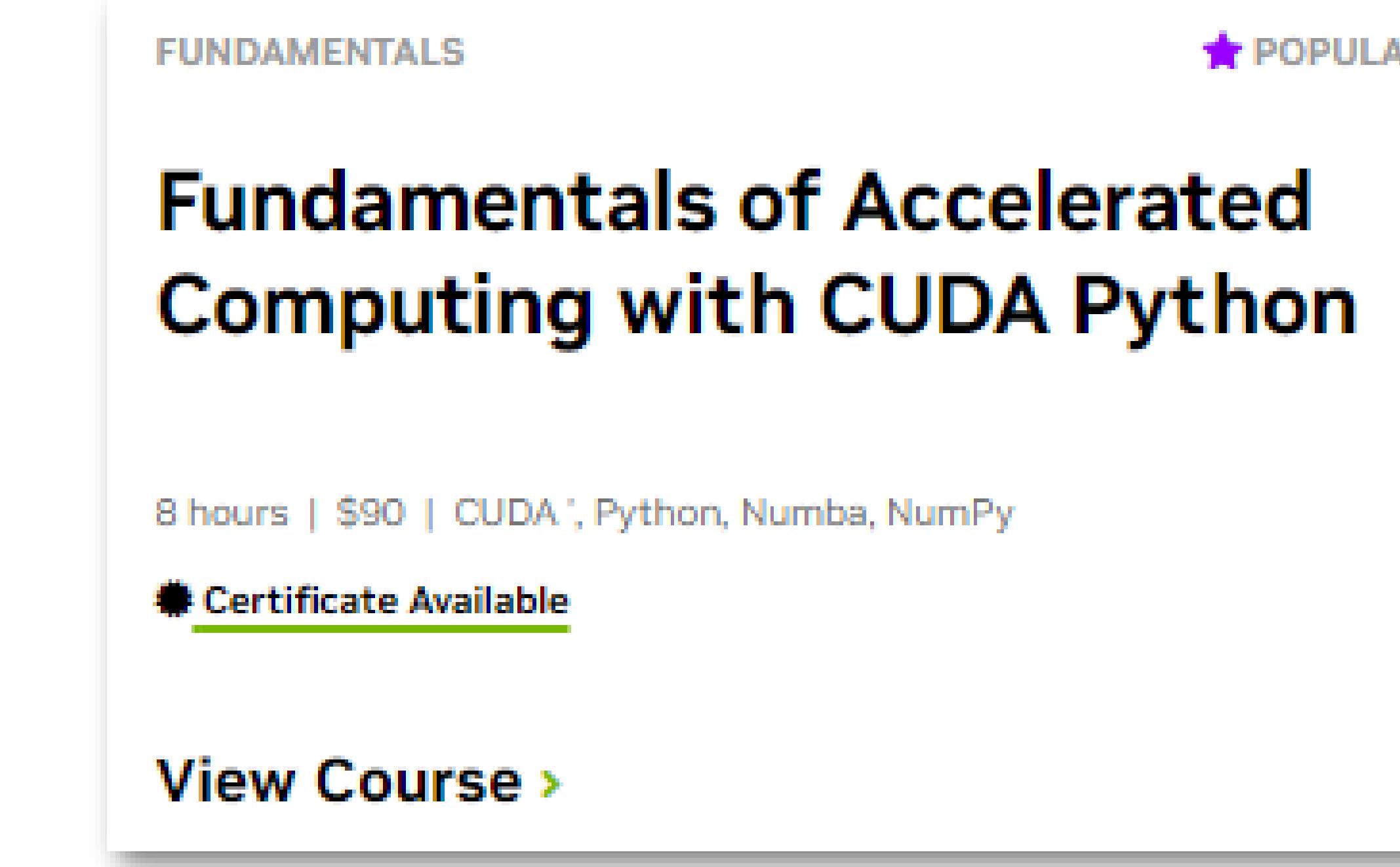
FUNDAMENTALS      ★ POPULAR

**Fundamentals of Accelerated Computing with CUDA Python**

8 hours | \$90 | CUDA®, Python, Numba, NumPy

● Certificate Available

[View Course >](#)

A course card for "Fundamentals of Accelerated Computing with CUDA Python". It features the Numba logo at the top left, followed by the word "FUNDAMENTALS" and a purple star icon with the text "POPULAR" next to it. The main title is "Fundamentals of Accelerated Computing with CUDA Python" in bold black font. Below the title is a description: "8 hours | \$90 | CUDA®, Python, Numba, NumPy". A small circular icon with a dot and a line is followed by the text "Certificate Available" underlined. At the bottom is a blue button with white text: "View Course >".

# SESSIONS AT PREVIOUS GTC

SEARCH ON [NVIDIA ON DEMAND](#)

If you found this content useful, please consider tuning into these sessions too:

- GPU-accelerated Feature Extraction and Image Similarity in Pure Python [S41661]
- Enabling Python User-Defined Functions in Accelerated Applications with Numba [S41056]
- No More Porting: Coding for GPUs with Standard C++, Fortran, and Python [S41496]
- Shifting through the Gears of GPU Programming: Understanding Performance and Portability Trade-offs [S41620]
- Evaluating Your Options for Accelerated Numerical Computing in Pure Python

# GTC 2025

<https://www.nvidia.com/gtc/>

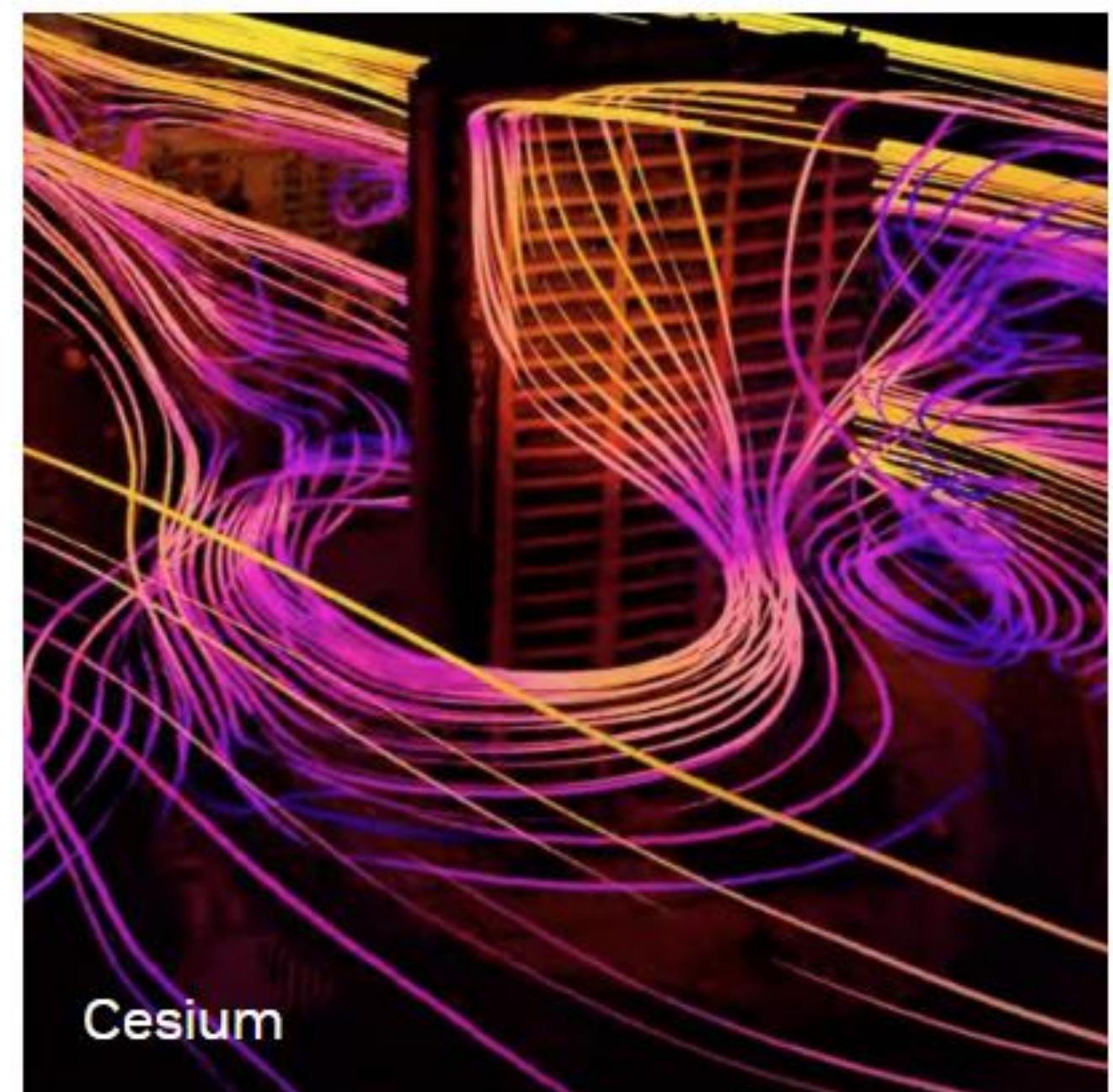
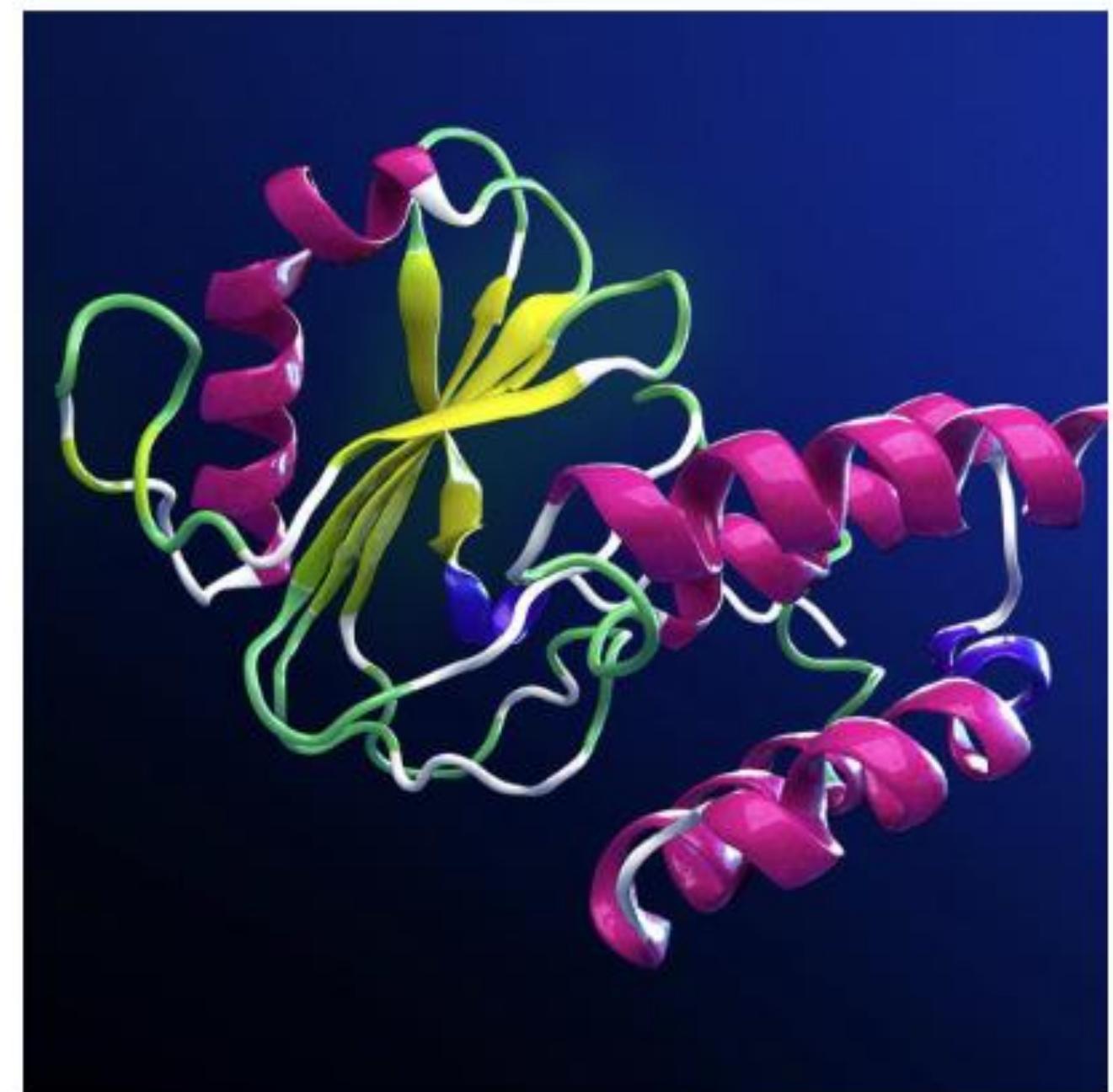
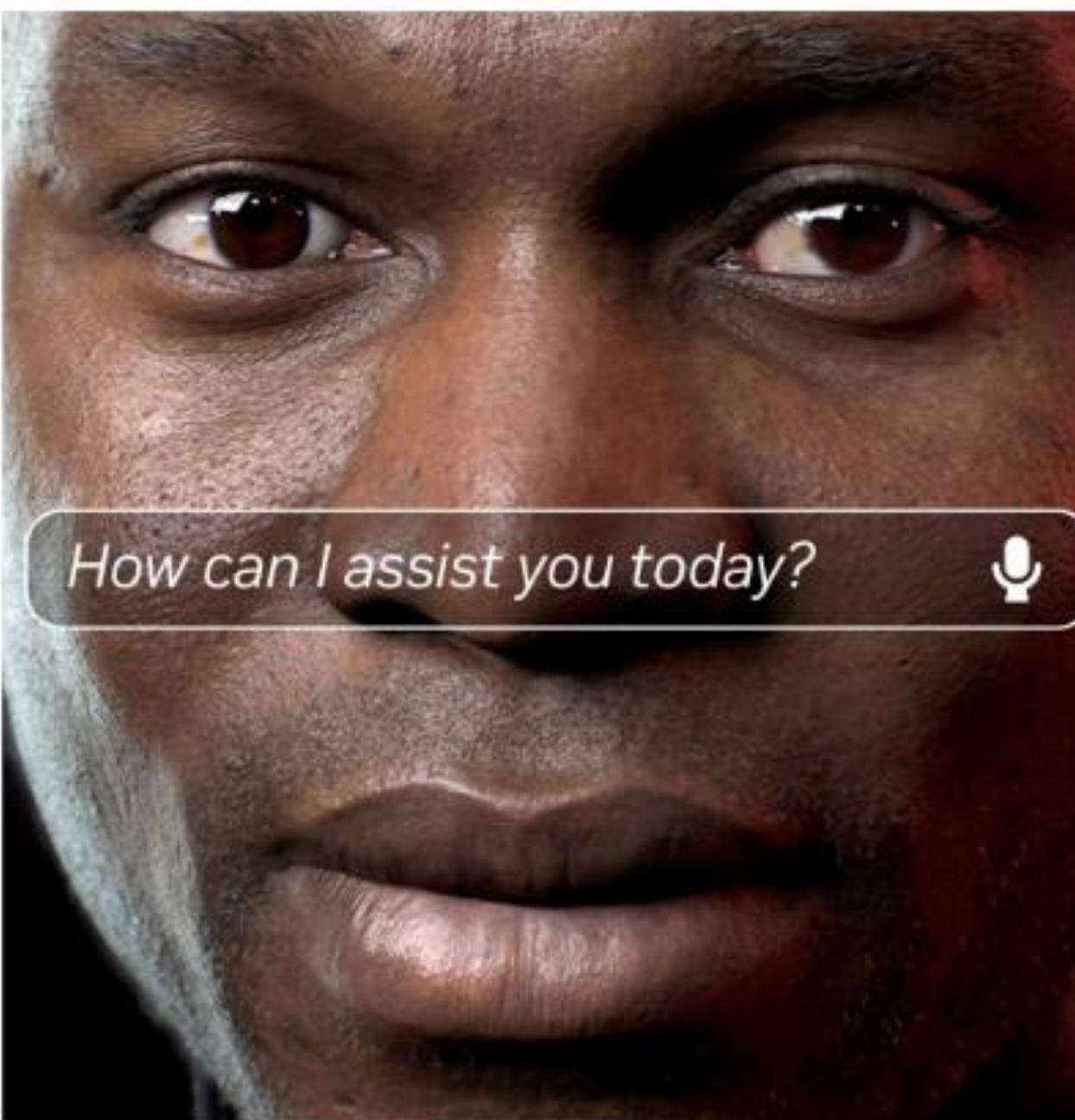
**AI Conference** March 17-21, 2025 | **Keynote** March 18 | **Exhibits** March 18-21 | **Workshops** March 16-20 | **San Jose, CA & Virtual**

**Connect** with thousands of developers, researchers, and trailblazers revolutionizing our world through accelerated computing and AI.

**Experience** the highly anticipated keynote by NVIDIA CEO Jensen Huang at the SAP Center in San Jose and choose from over 500 expert-led sessions tailored to every technical level.

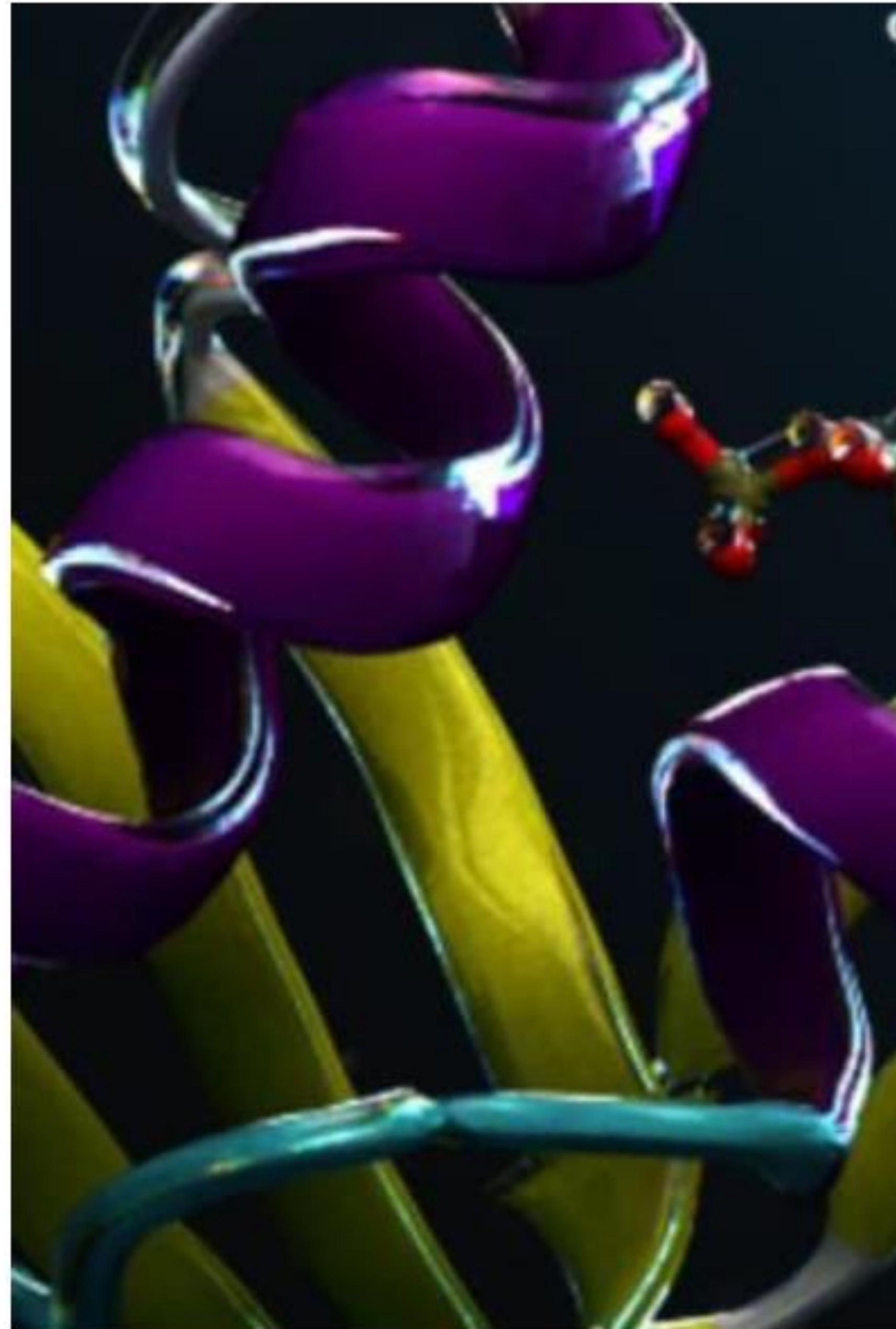
**Explore** 200+ cutting-edge exhibits and enjoy exclusive networking events designed to spark new collaborations.

Register before early-bird pricing ends on February 3 for a chance to win a seat at the keynote venue and see Jensen Huang live.



# NVIDIA Clara for Healthcare and Life Sciences

World's Largest Data Industry | 36% CAGR by 2025



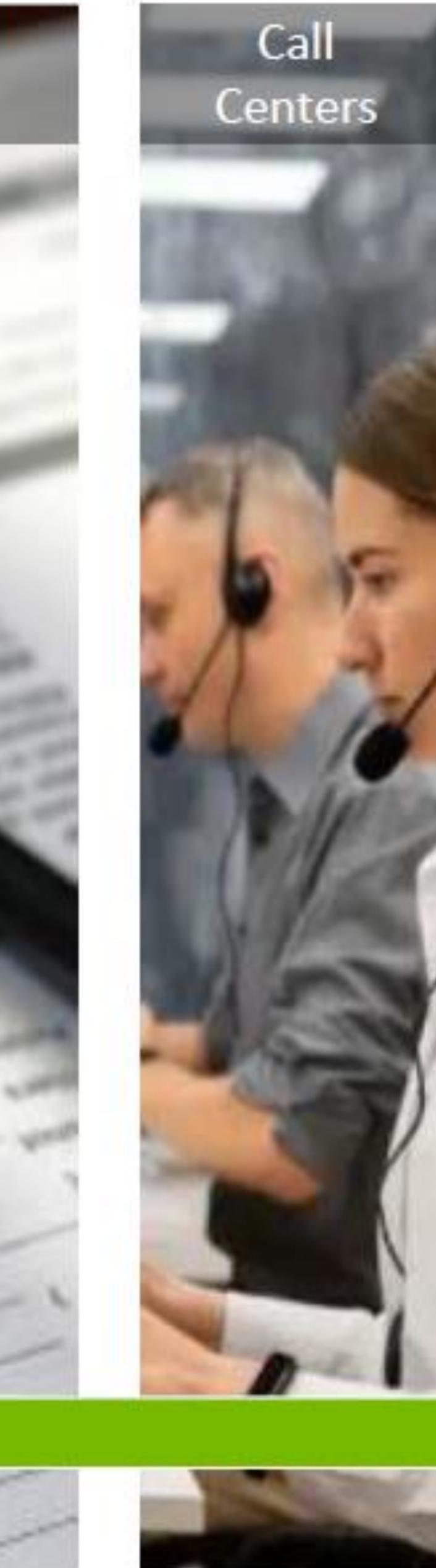
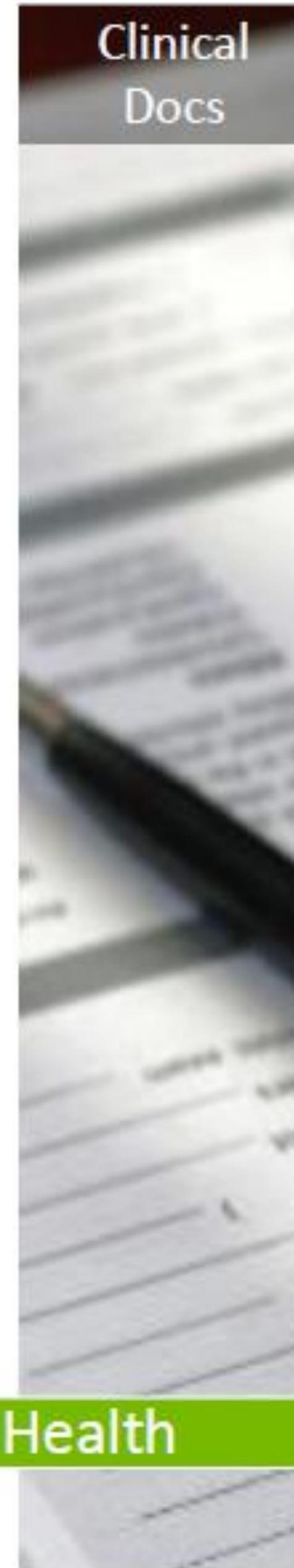
Digital Biology

BioNeMo, Parabricks



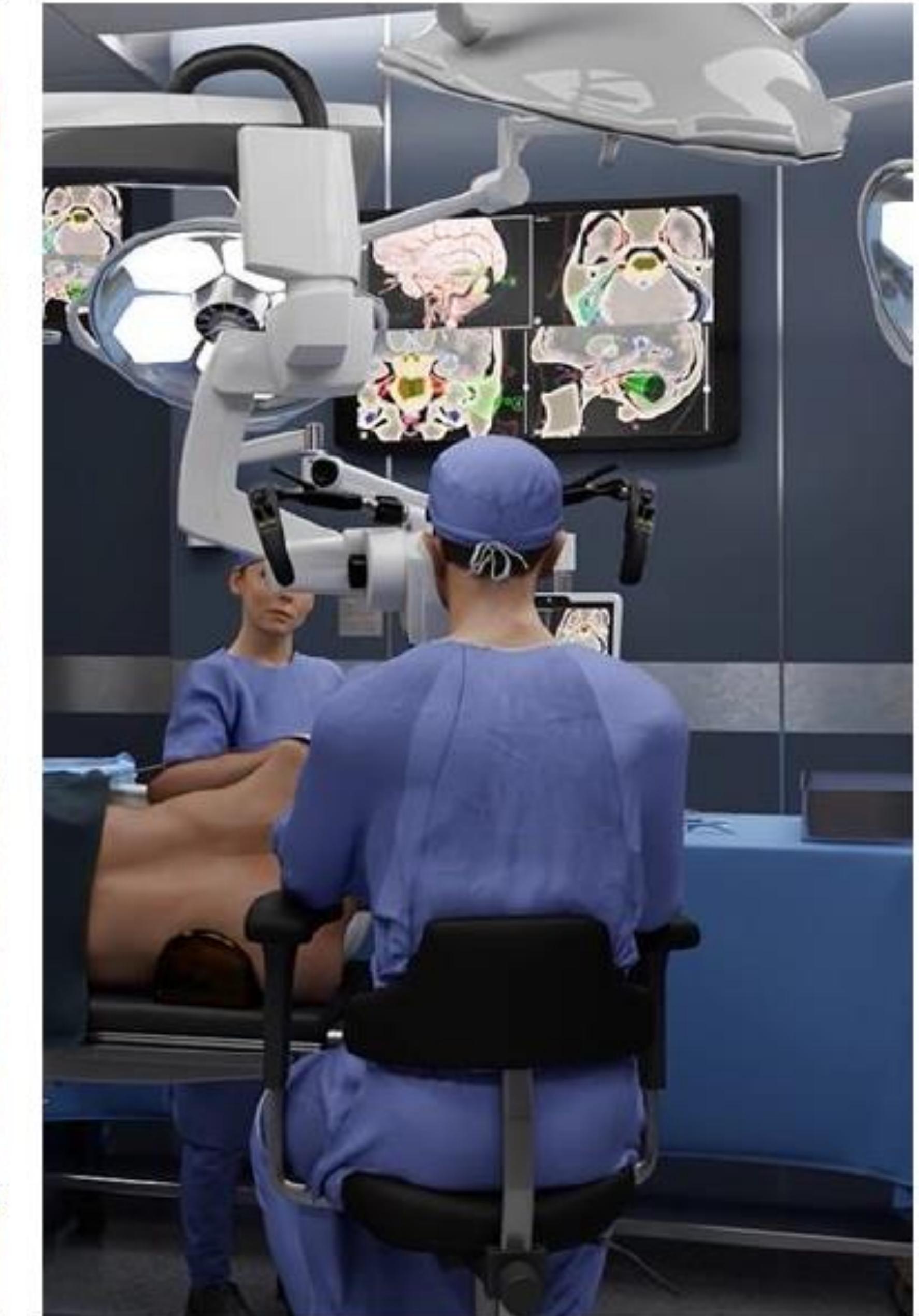
Real-World Data

NeMo



Agentic Interactions

ACE



Digital Devices

Holoscan, Isaac, MONAI

# AGENDA

Overview of GPU Computing

---

GPU-Accelerated Numerical Computing with *CuPy*

---

GPU-Accelerated Data Science with *RAPIDS*

---

Custom GPU Kernels with *Numba*

---

Frameworks Interoperability - *Data Conversion Bottleneck*

---

ZERO-COPY end-to-end pipeline - try out on *NVIDIA Brev*

---



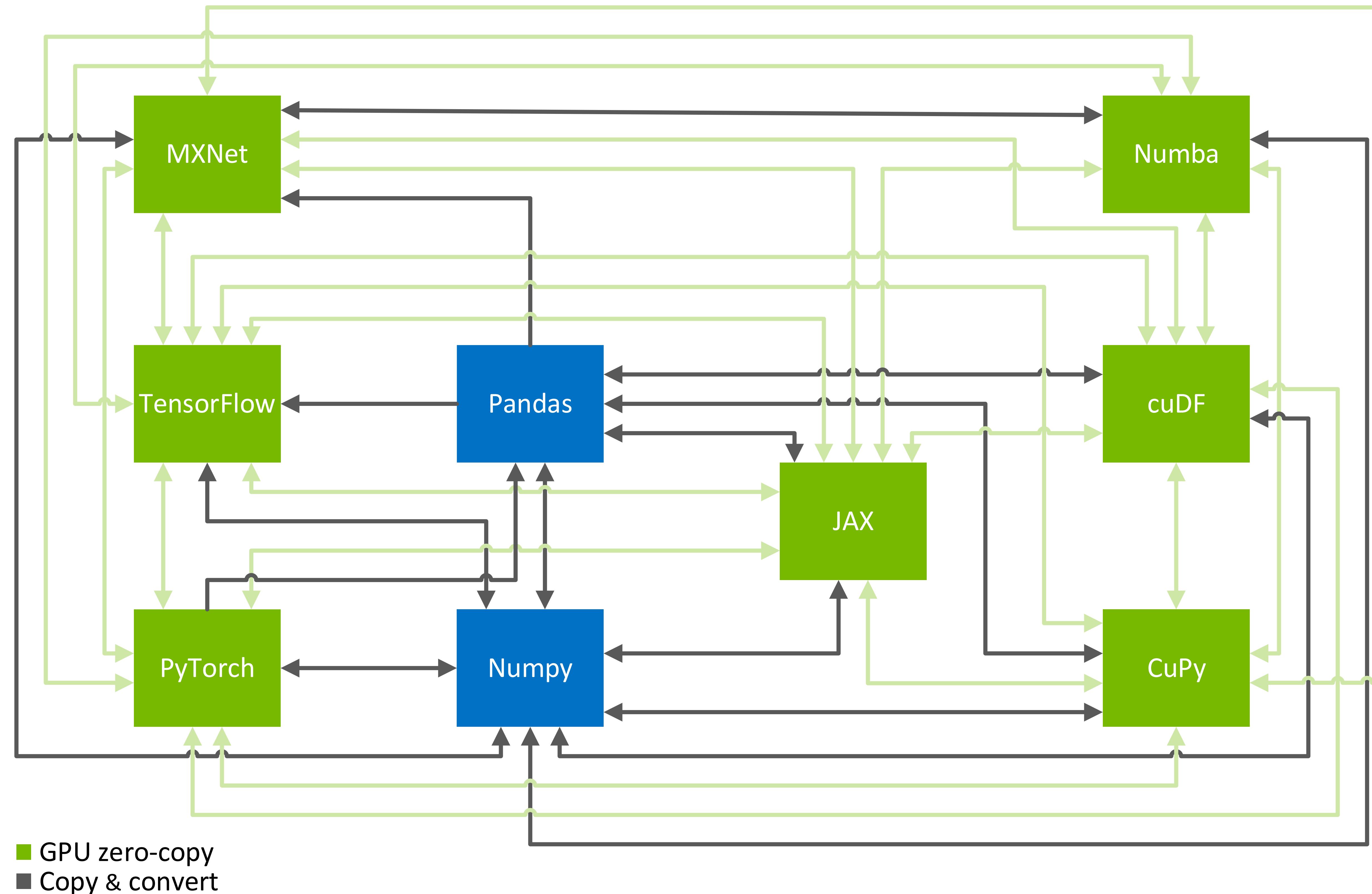
# FRAMEWORK INTEROPERABILITY

When a single framework is not enough



# MIX AND MATCH WORKFLOWS

Use the right tool, for the right job, in the right way



# DLPACK

## Sharing tensors the easiest way

DLPack is an open in-memory tensor structure which enables:

- Easier sharing of tensors and operators between deep learning frameworks.
- Easier wrapping of vendor level operator implementations, allowing collaboration when introducing new devices/ops.
- Quick swapping of backend implementations, like different version of BLAS.
- For final users, this could bring more operators, and possibility of mixing usage between frameworks.

### From cuDF to CuPy

```
# Convert a cuDF DataFrame to a CuPy ndarray
src = cudf.DataFrame({'x': [1, 2], 'y': [3, 4]})
dst = cp.fromDlpack(src.to_dlpack())

print(type(dst), "\n", dst)

<class 'cupy.core.core.ndarray'>
[[1 3]
 [2 4]]
```

### From CuPy to PyTorch

```
# Convert a CuPy ndarray to a PyTorch Tensor
src = cp.array([[1, 2], [3, 4]])
dst = torch.utils.dlpack.from_dlpack(src.toDlpack())

print(type(dst), "\n", dst)

<class 'torch.Tensor'>
tensor([[1, 2],
        [3, 4]], device='cuda:0')
```

# CUDA ARRAY INTERFACE 3.0

## Seamless Ingestion

The `__cuda_array_interface__` attribute returns a dictionary (`dict`) that must contain the following entries:

**shape: (integer, ...)**

A tuple of int (or long) representing the size of each dimension.

**typestr: str**

The type string. This has the same definition as `typestr` in the numpy array interface.

**data: (integer, boolean)**

The data is a 2-tuple. The first element is the data pointer as a Python `int` (or `long`). The data must be device-accessible. For zero-size arrays, use 0 here. The second element is the read-only flag as a Python `bool`.

**version: integer**

An integer for the version of the interface being exported. The current version is 3.



# DLPACK & CUDA ARRAY INTERFACE

	DLPack		NumPy Array Interface	CUDA Array Interface
	CPU	GPU	CPU	GPU
Pandas	X	n/a	✓	n/a
NumPy	X	n/a	✓	n/a
cuDF	n/a	✓	n/a	✓
CuPy	n/a	✓	n/a	✓
JAX	✓	✓	✓	✓
Numba	X	X	✓	✓
TensorFlow	✓	✓	✓	X
PyTorch	✓	✓	✓	✓
MXNet	✓	✓	✓	X

## CUDA Array Interface adopted by:

- Numba
- CuPy
- PyTorch
- PyArrow
- mpi4py
- ArrayViews
- JAX
- PyCUDA
- DALI
- RAPIDS
- cuDF
- cuML
- cuSignal
- RMM

# Summary

Complex workloads make use of multiple libraries.

Interoperability via DLPack and CUDA Array Interface (CAI).

Zero-copy and no data conversion is the goal. Not always possible, yet.

# AGENDA

Overview of GPU Computing

---

GPU-Accelerated Numerical Computing with *CuPy*

---

GPU-Accelerated Data Science with *RAPIDS*

---

Custom GPU Kernels with *Numba*

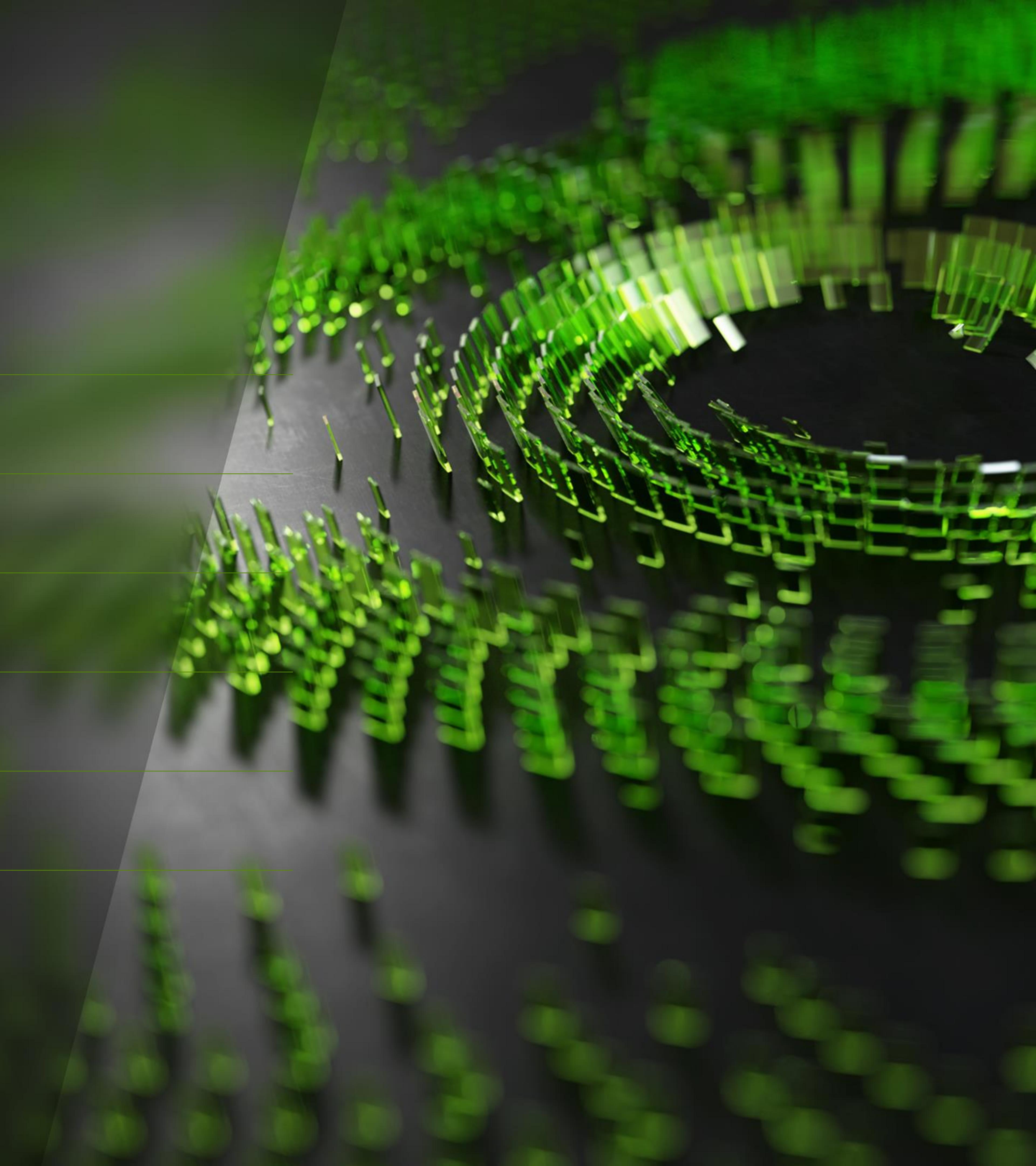
---

Frameworks Interoperability - *Data Conversion Bottleneck*

---

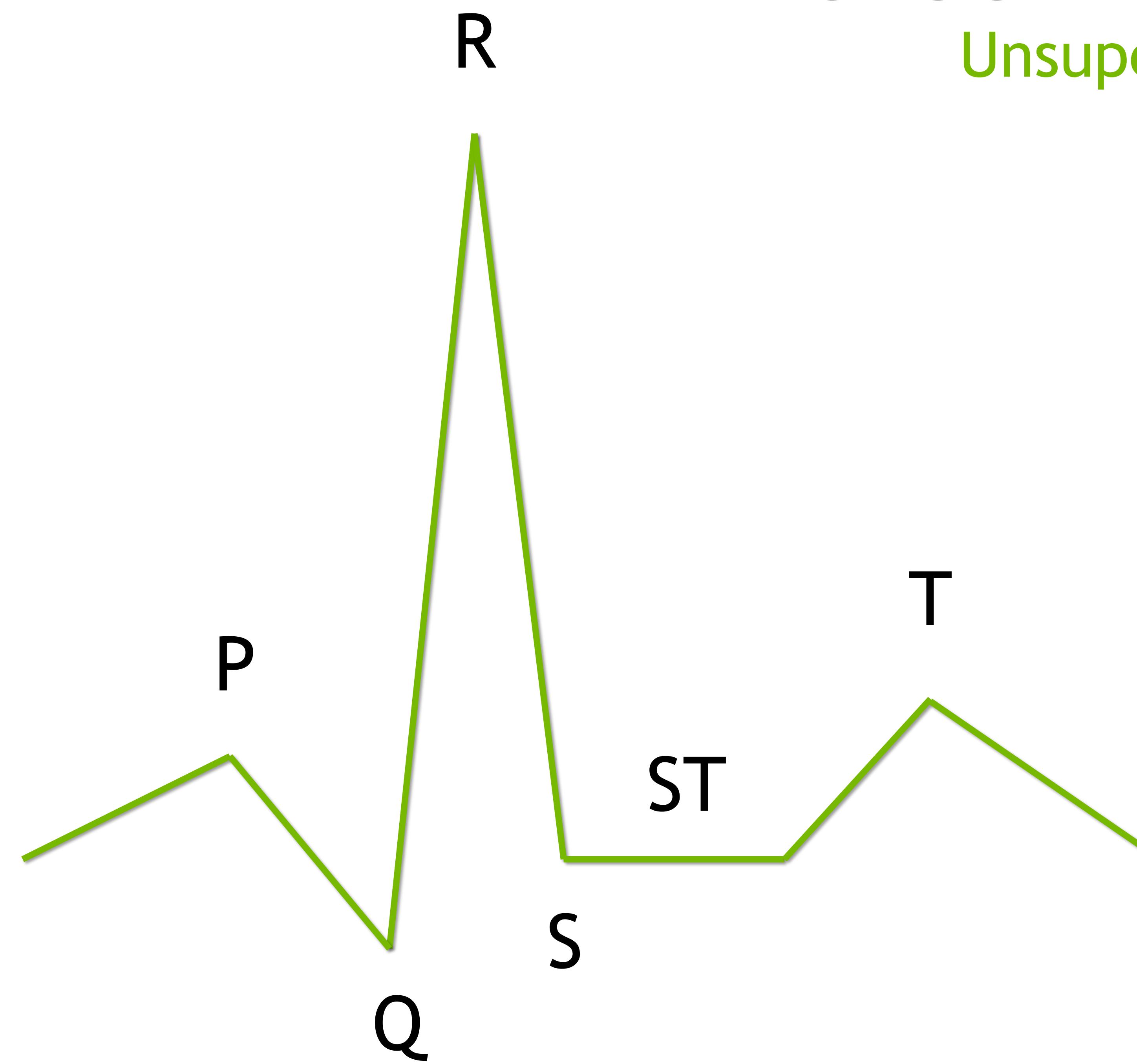
ZERO-COPY end-to-end pipeline - try out on *NVIDIA Brev*

---



# ZERO-COPY END-TO-END PIPELINE

Unsupervised outlier detection



What we have:

- 20 hours stream of continuously measured electrocardiogram (ECG) data.
- Univariate and uniformly sampled time series as CSV on disk.

What we are doing:

- Unsupervised segmentation of ECG stream into ~100k heartbeats.
- Training of Variational Autoencoder (VAE) for outlier detection.
- Visualization of the latent space & generated heartbeats.

*Disclaimer*

*Technical example pipeline demonstrating framework interoperability.  
Not suitable for production in medical environments.*

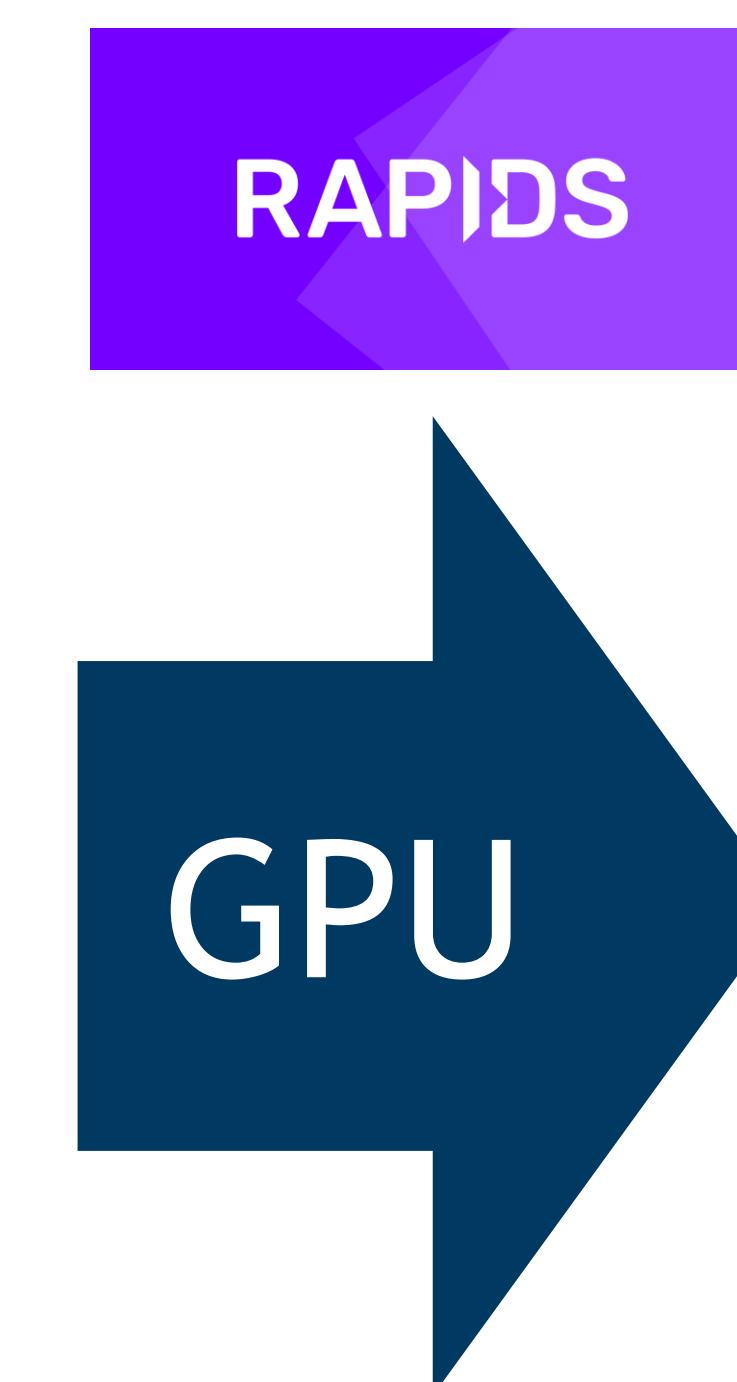
# END-TO-END PIPELINE

Parse ECG from CSV



```
1 data
2 1.3863000000000008e+00
3 1.734900000000000109e+00
4 2.186500000000000110e+00
5 2.7471999999999864e+00
6 3.75190000000000013e+00
7 5.0231000000000342e+00
8 6.4785000000000369e+00
9 8.0025999999999270e+00
10 9.4921000000000648e+00
11 1.082540000000013e+01
12 1.1867699999999925e+01
13 1.2486800000000057e+01
14 1.25873000000000082e+01
15 1.2142899999999914e+01
```

```
cudf.io.csv.read_csv(filepath_or_buffer,
lineterminator='\n', quotechar='', quoting=0,
doublequote=True, header='infer',
mangle_dupe_cols=True, usecols=None,
sep=',', delimiter=None,
delim_whitespace=False,
skipinitialspace=False, names=None,
dtype=None, skipfooter=0, skiprows=0,
dayfirst=False, compression='infer',
thousands=None, decimal=',',
true_values=None, false_values=None,
nrows=None, byte_range=None,
skip_blank_lines=True, parse_dates=None,
comment=None, na_values=None,
keep_default_na=True, na_filter=True,
prefix=None, index_col=None, **kwargs)
```



	data
0	1.3863
1	1.7349
2	2.1865
3	2.7472
4	3.7519
...	...

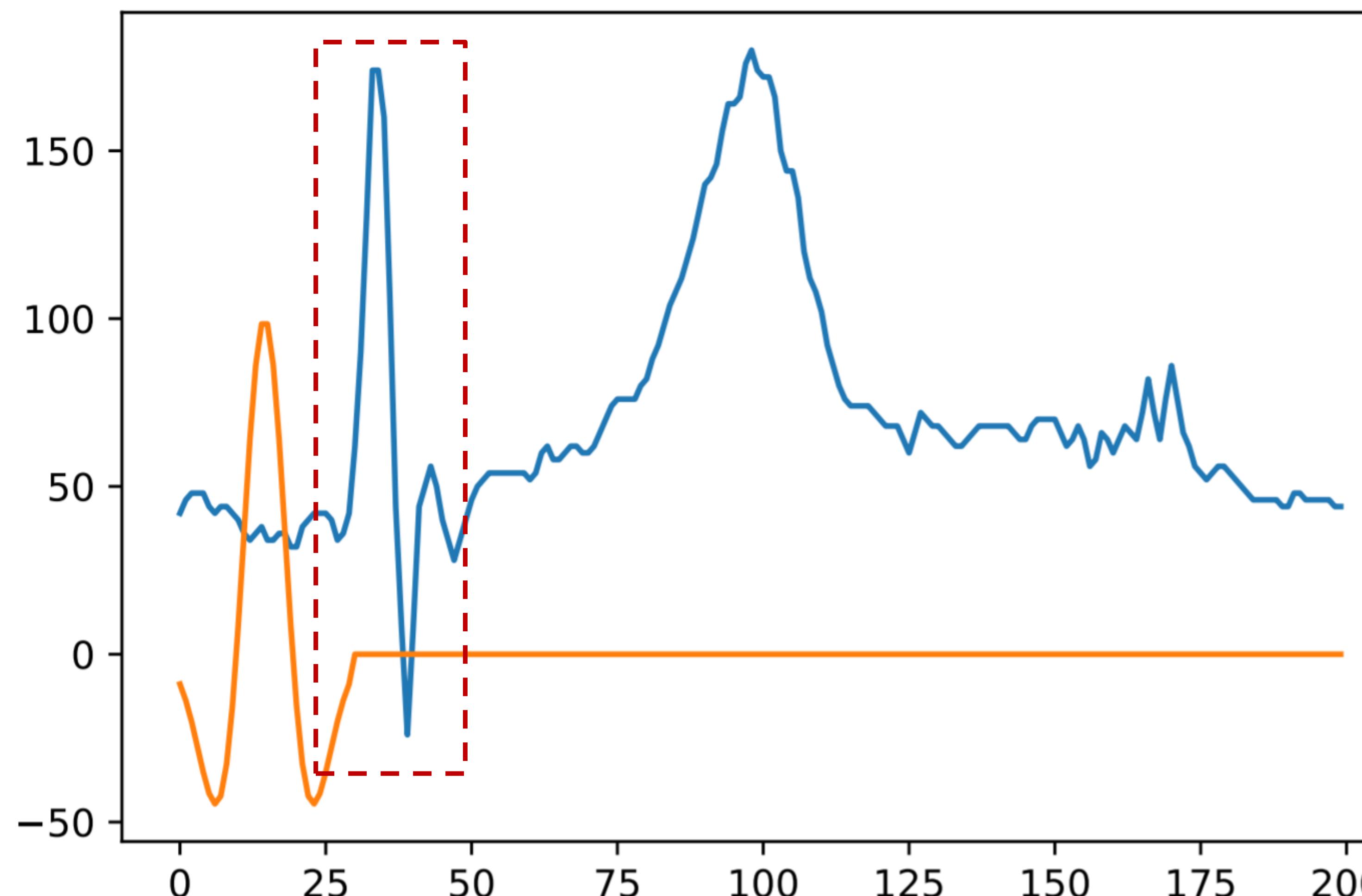
Load a comma-separated-values (CSV) dataset into a DataFrame

# END-TO-END PIPELINE

Band-Pass Filter

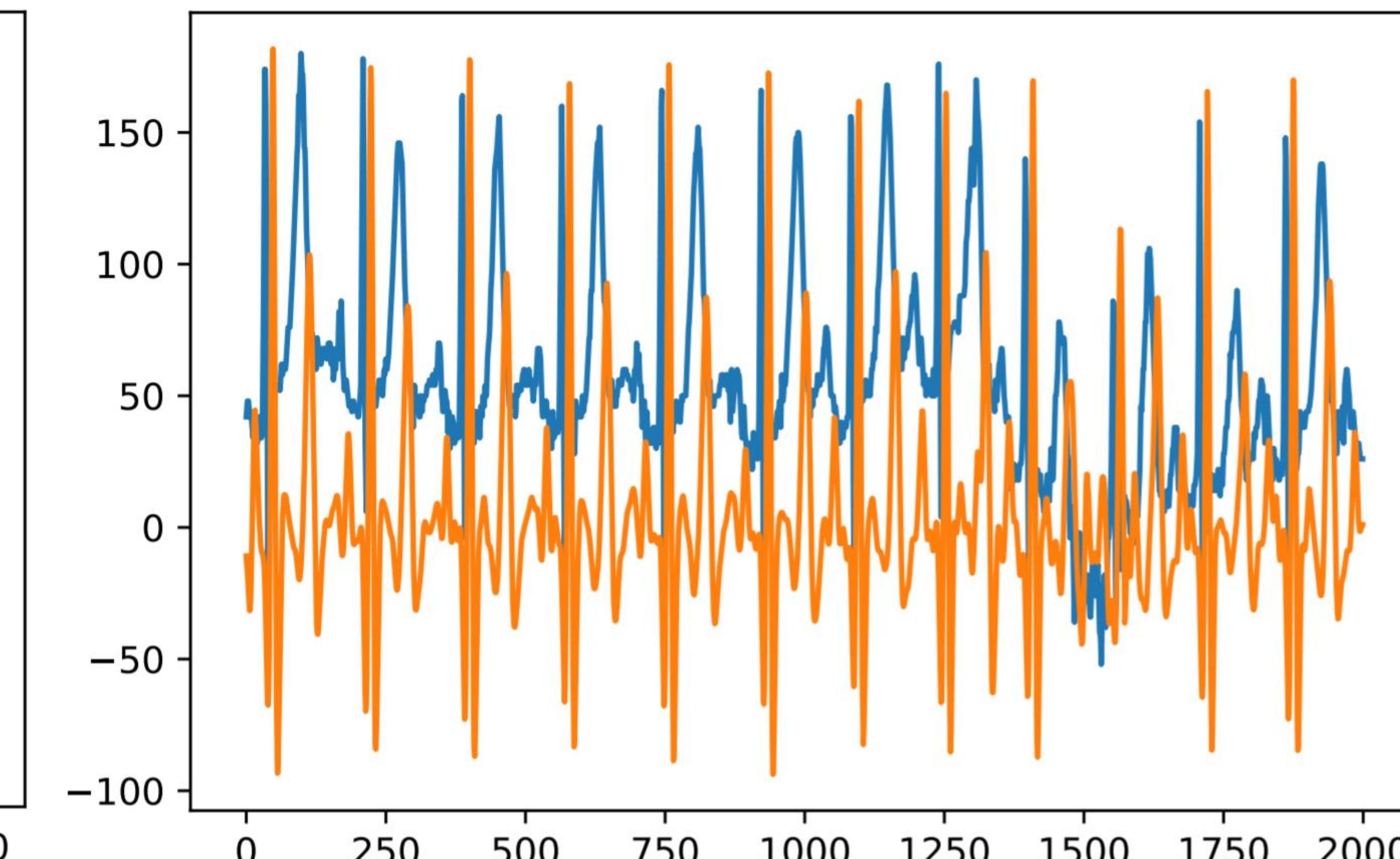


convolution with ricker wavelet



FFT based convolution

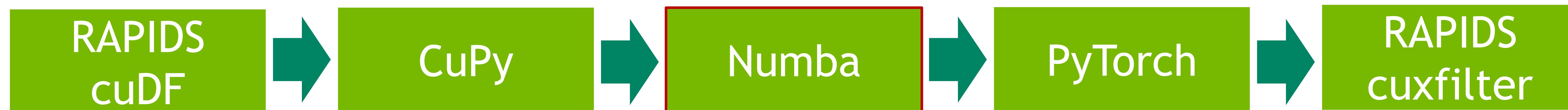
stream and smoothed curvature



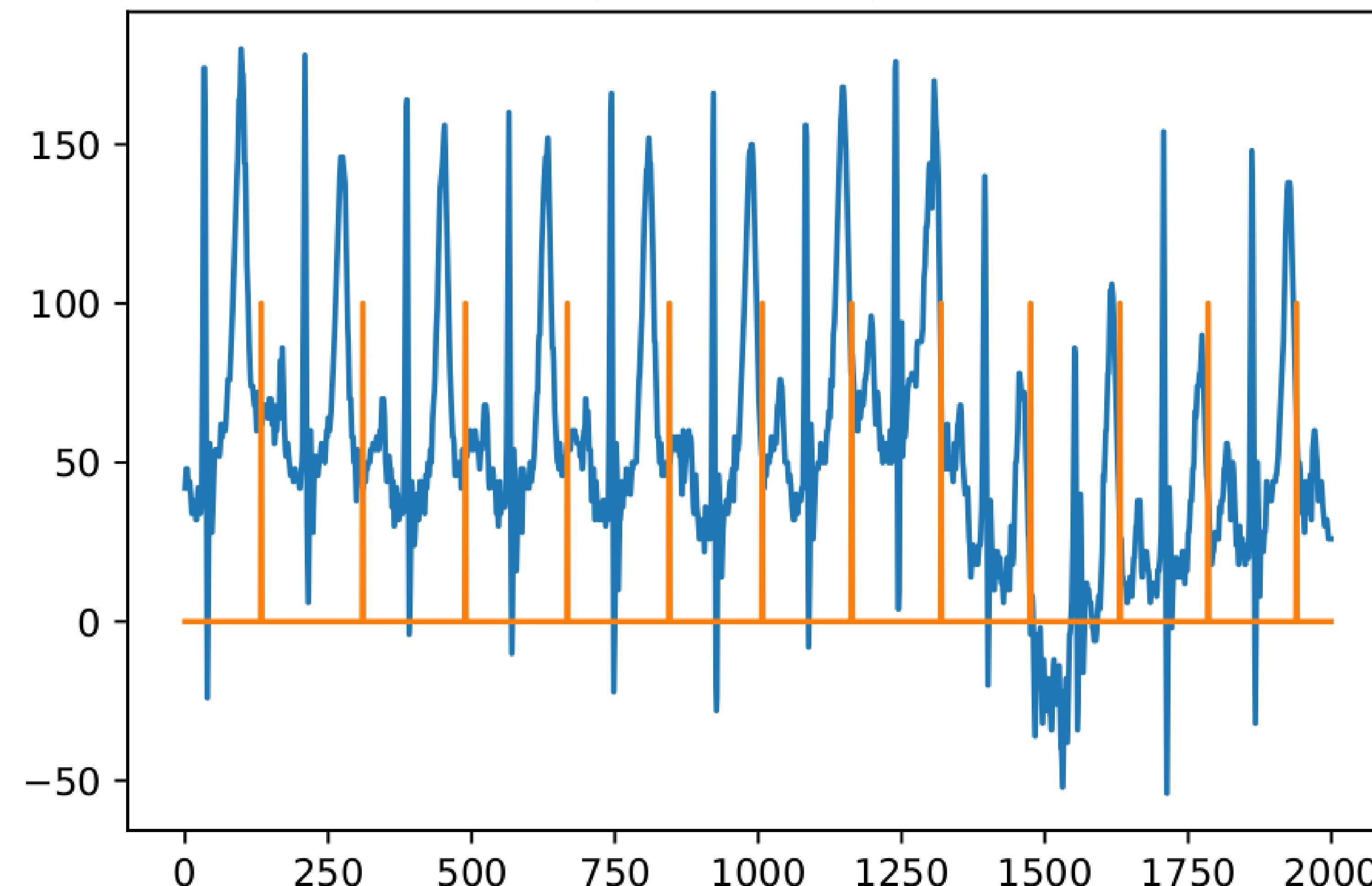
QRS Complex Detection

# END-TO-END PIPELINE

Non-trivial Preprocessing

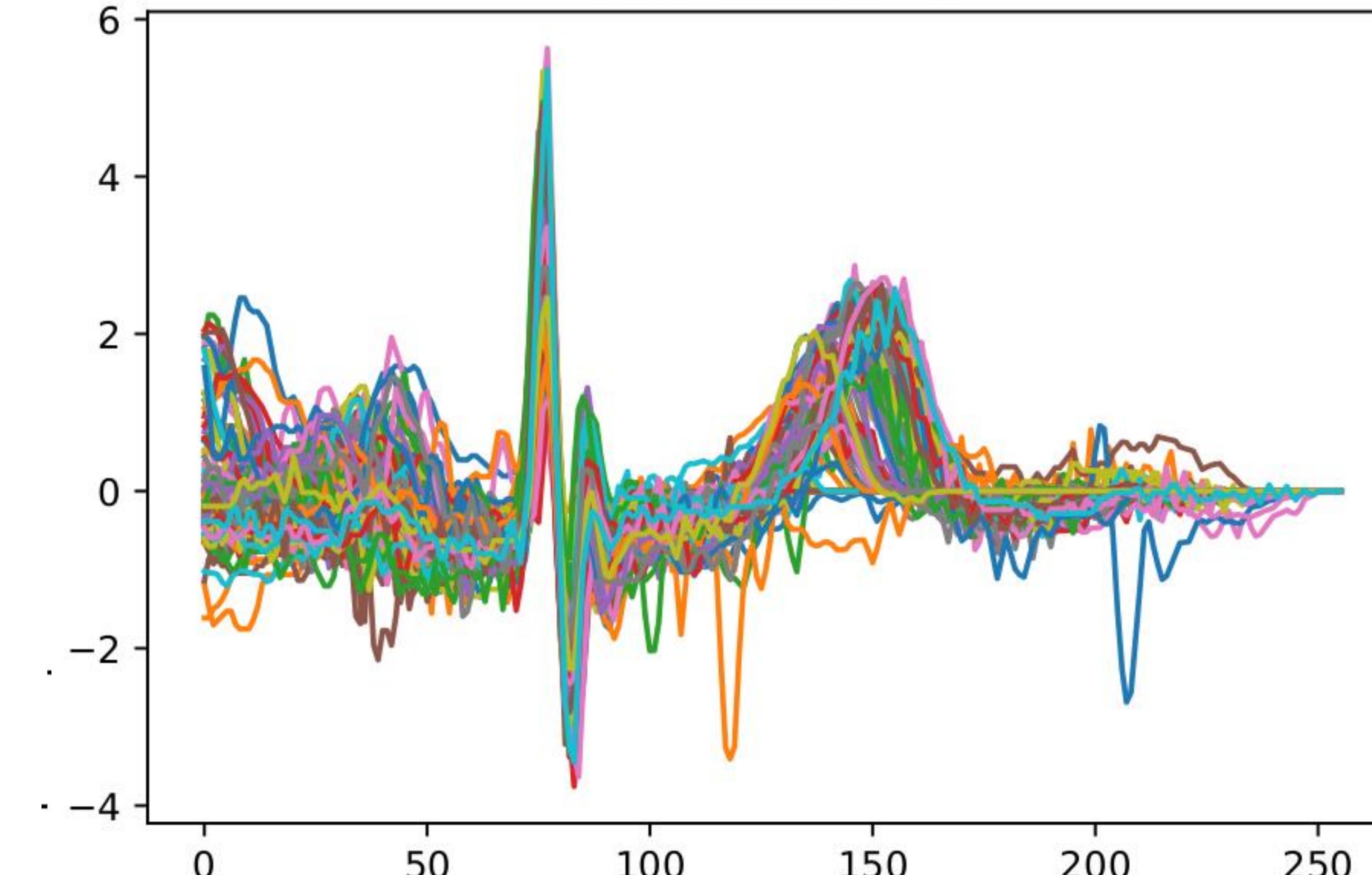


segmentation gates



1D non-max suppression

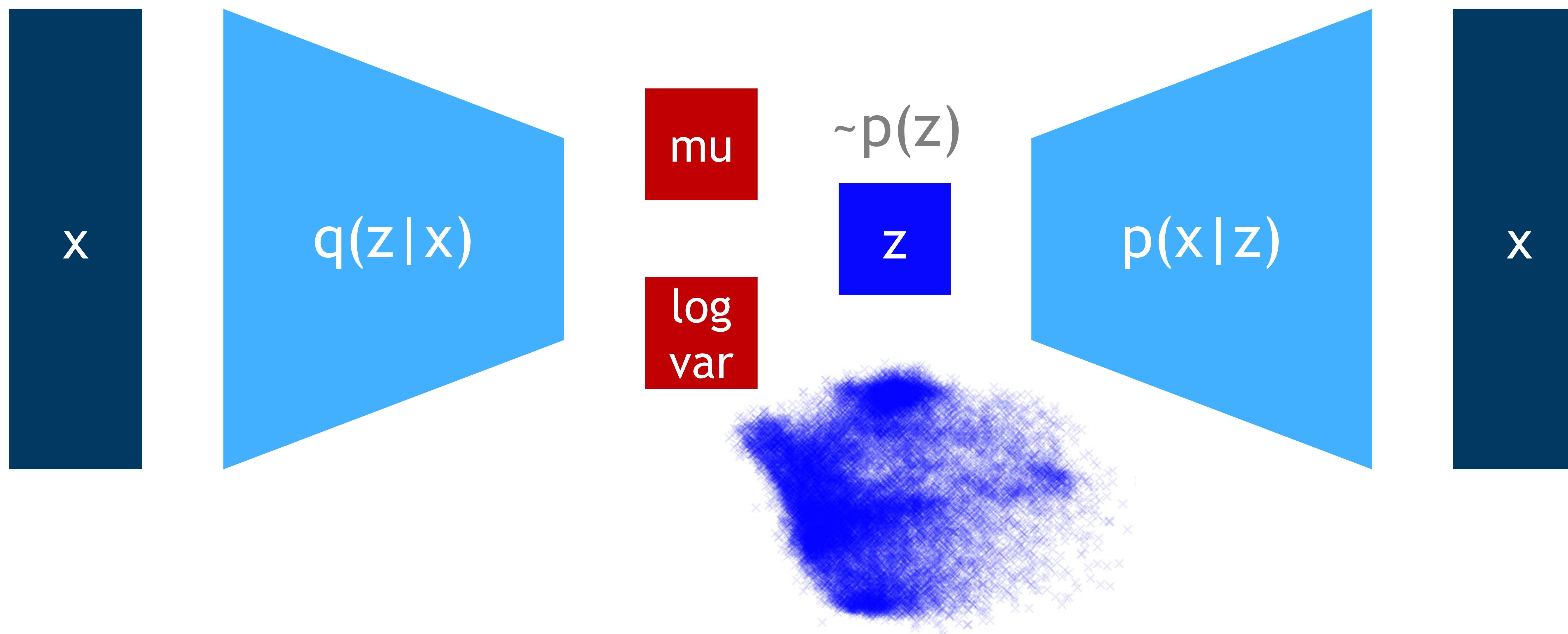
a few heartbeats



normalization and embedding

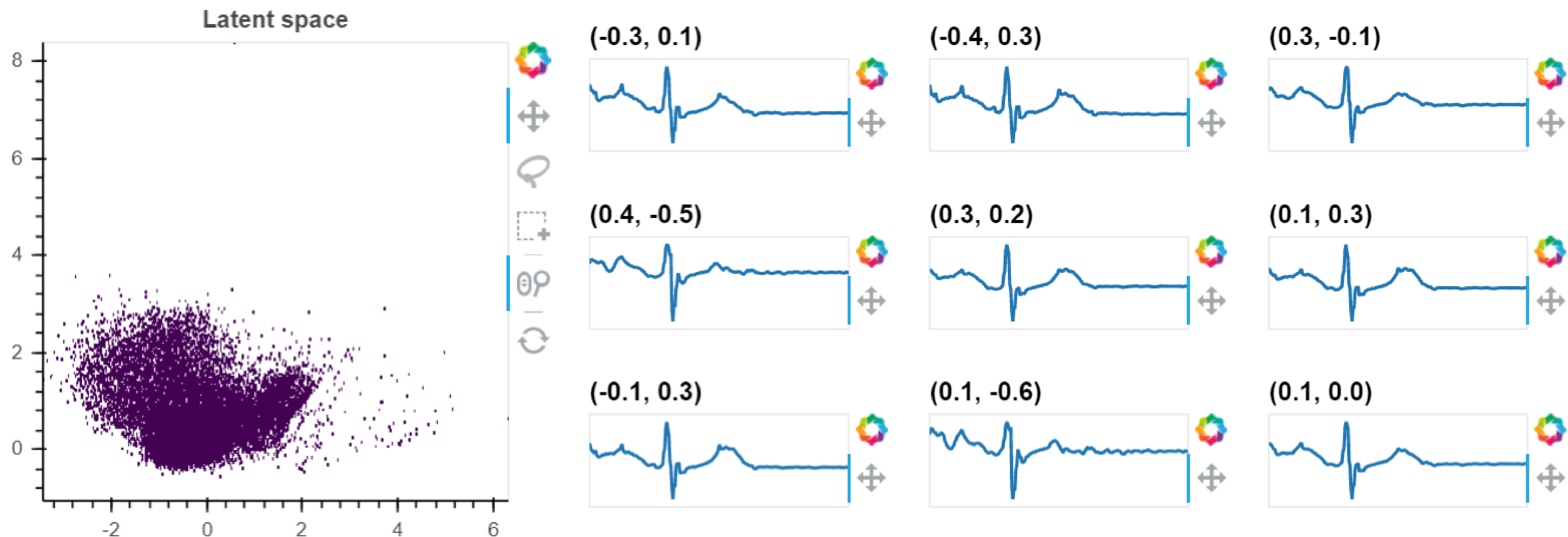
# END-TO-END PIPELINE

Variational Autoencoder (VAE)



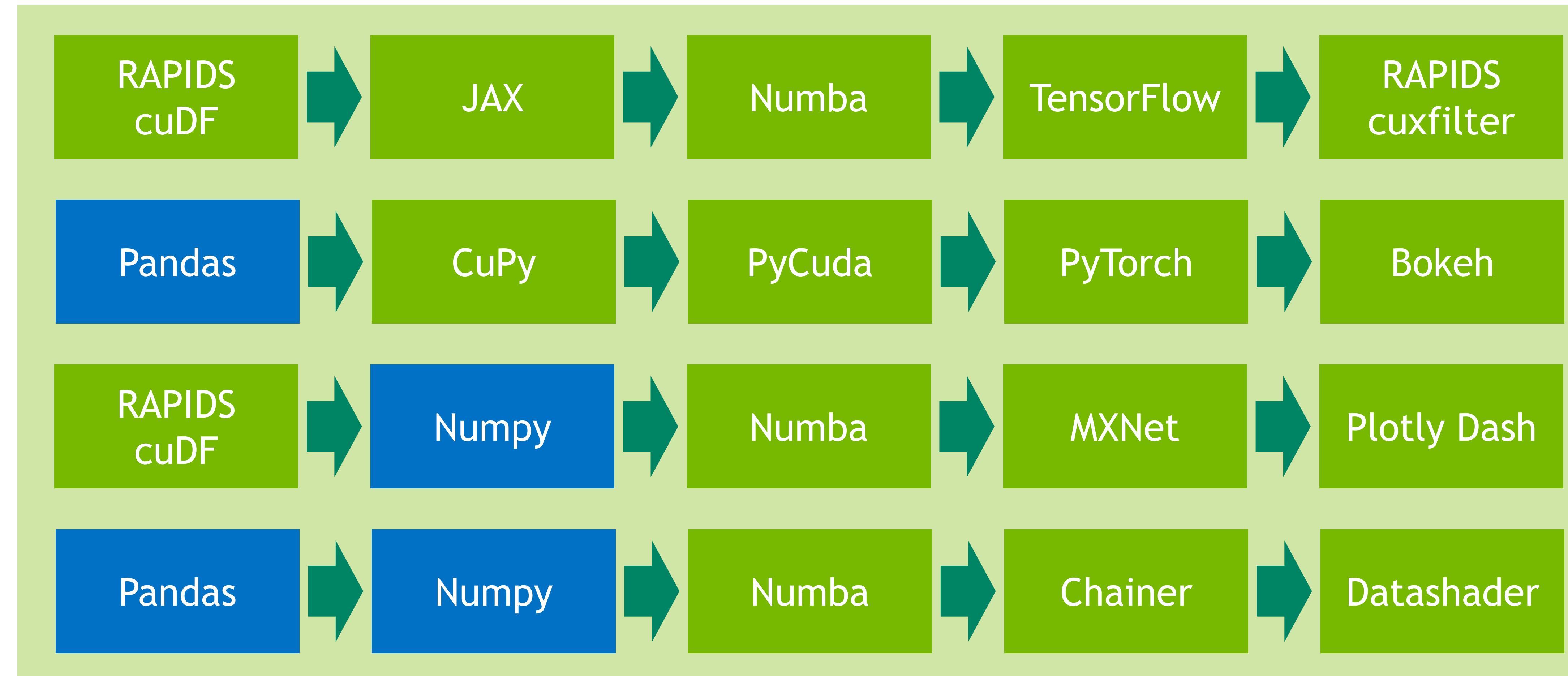
# END-TO-END PIPELINE

## Visualization



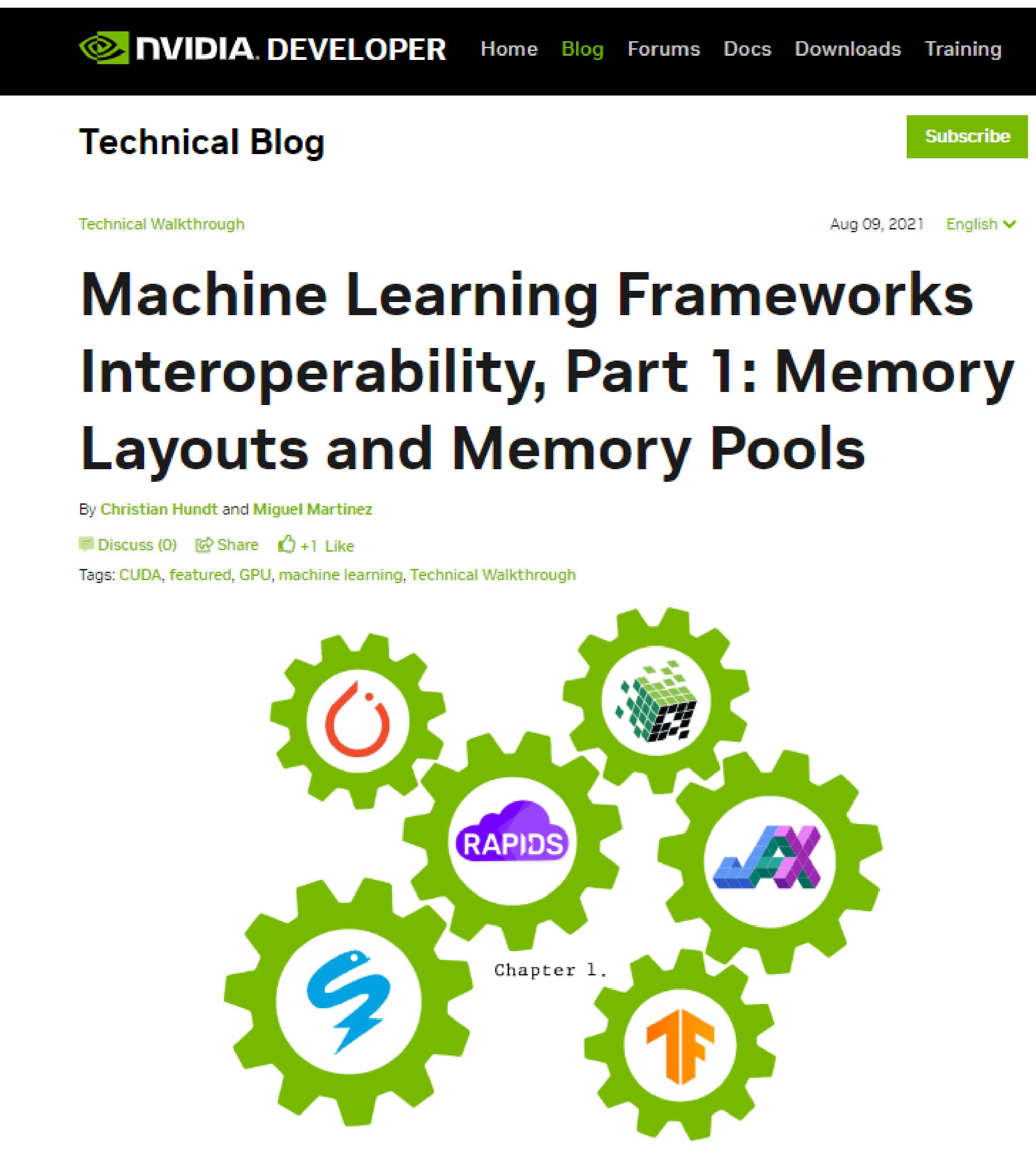
# MIX AND MATCH WORKFLOWS

Endless possibilities!



And many others!

# ADDITIONAL RESOURCE



Tech blog & GTC session

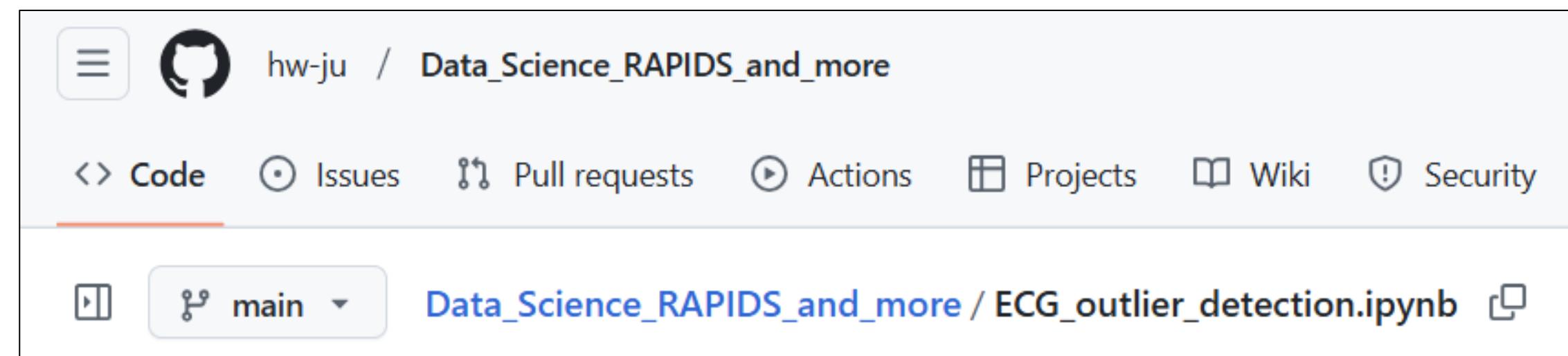
# Why Brev?

- User-friendly interface for GPU Cloud Computing.
- Launchable – template for building an instance -> Share POC with entire compute configuration, env setup.
- A wide range of GPU types -> meet budget and performance goals.
- Scale: CPU -> single GPU -> more advanced single GPU -> multi-GPU.

# Run on Brev

## - How to run the tutorial Jupyter Notebook

- Create your own Launchable using the [tutorial Jupyter Notebook](#) -> deploy



- OR one-click a prebuilt Launchable (**configured to run on a T4 instance**) on the [workshop github repo](#) -> deploy

