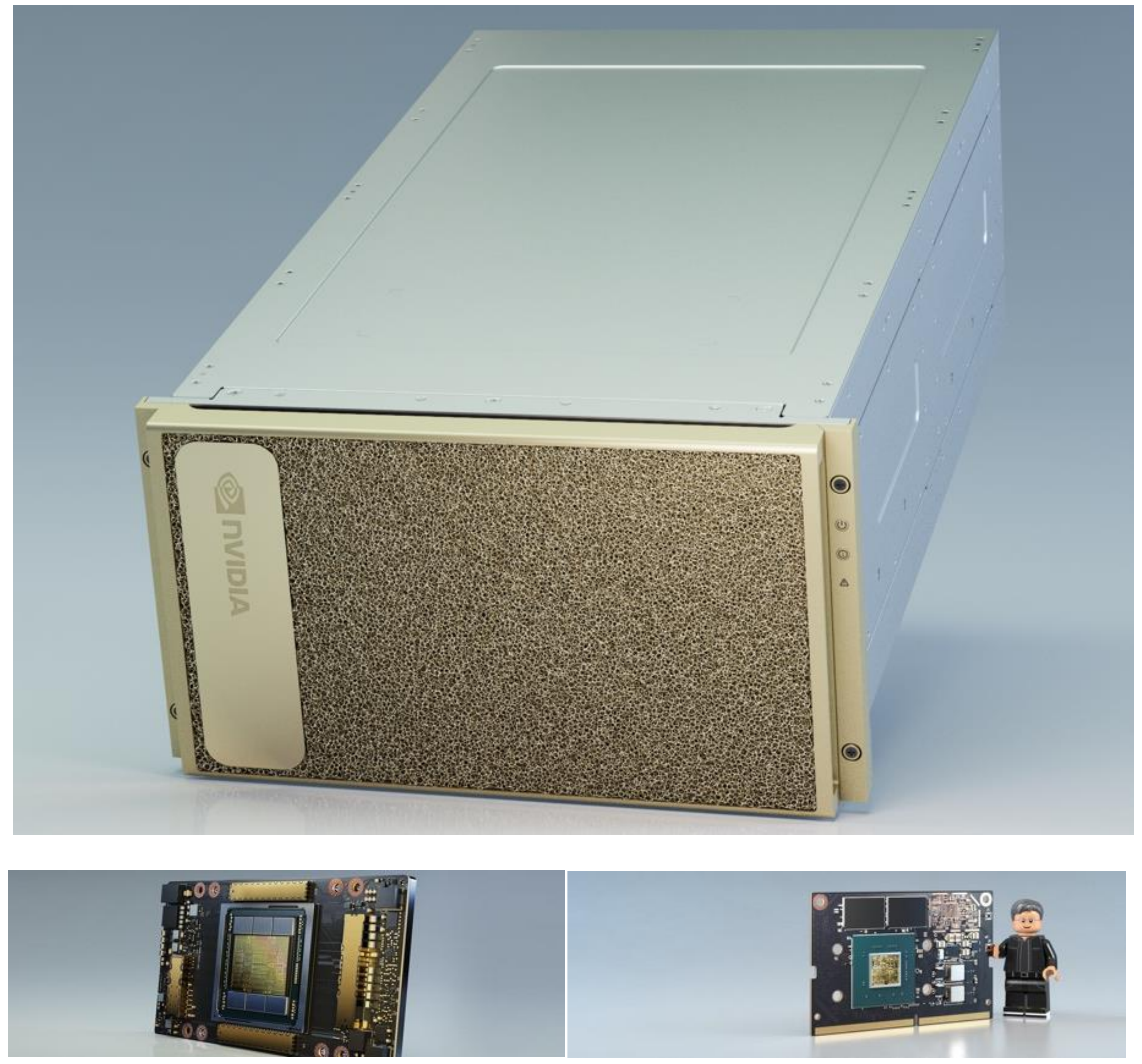
USING NVIDIA GPUS WITH PYTHON

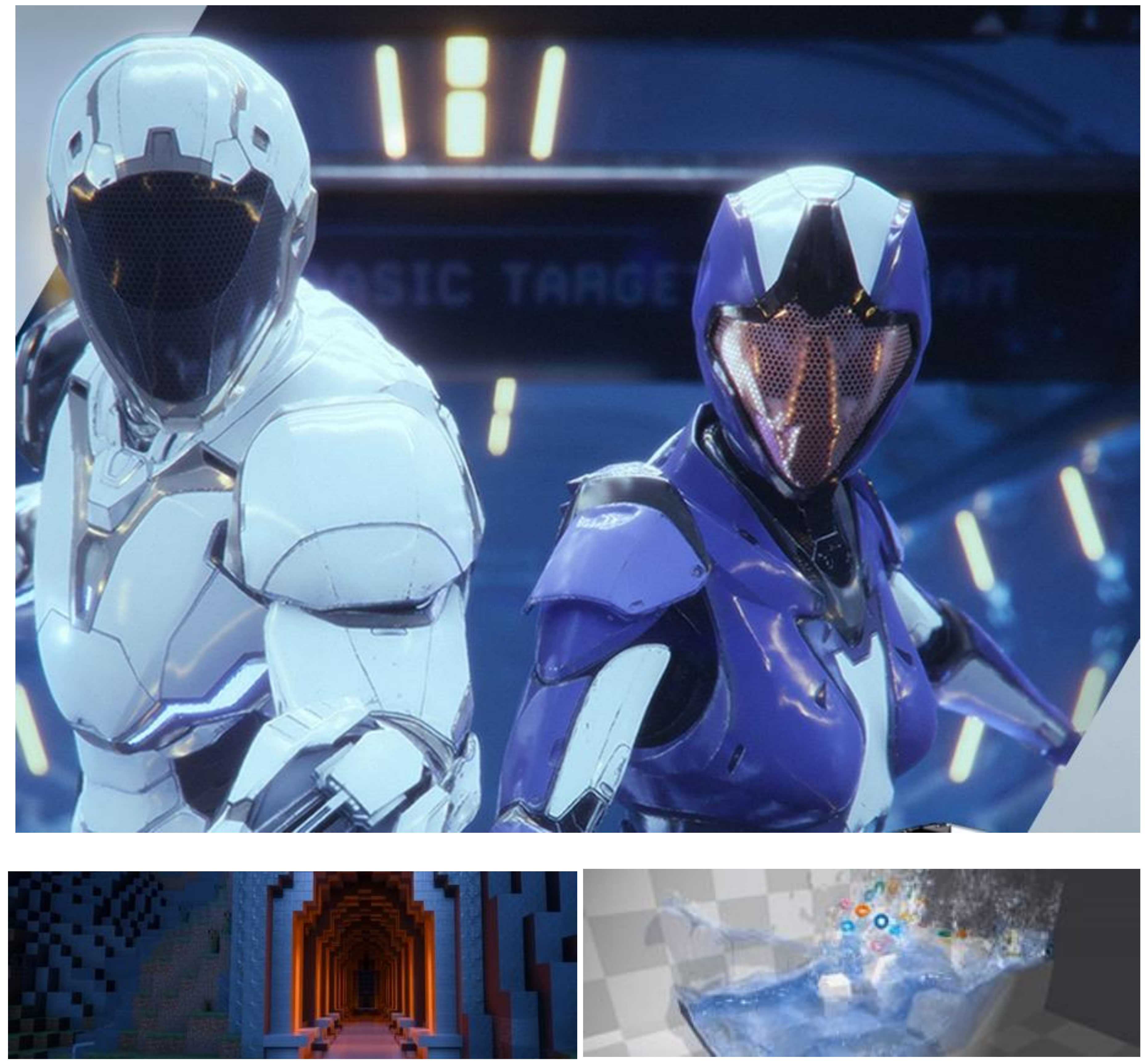
HANDS-ON WORKSHOP

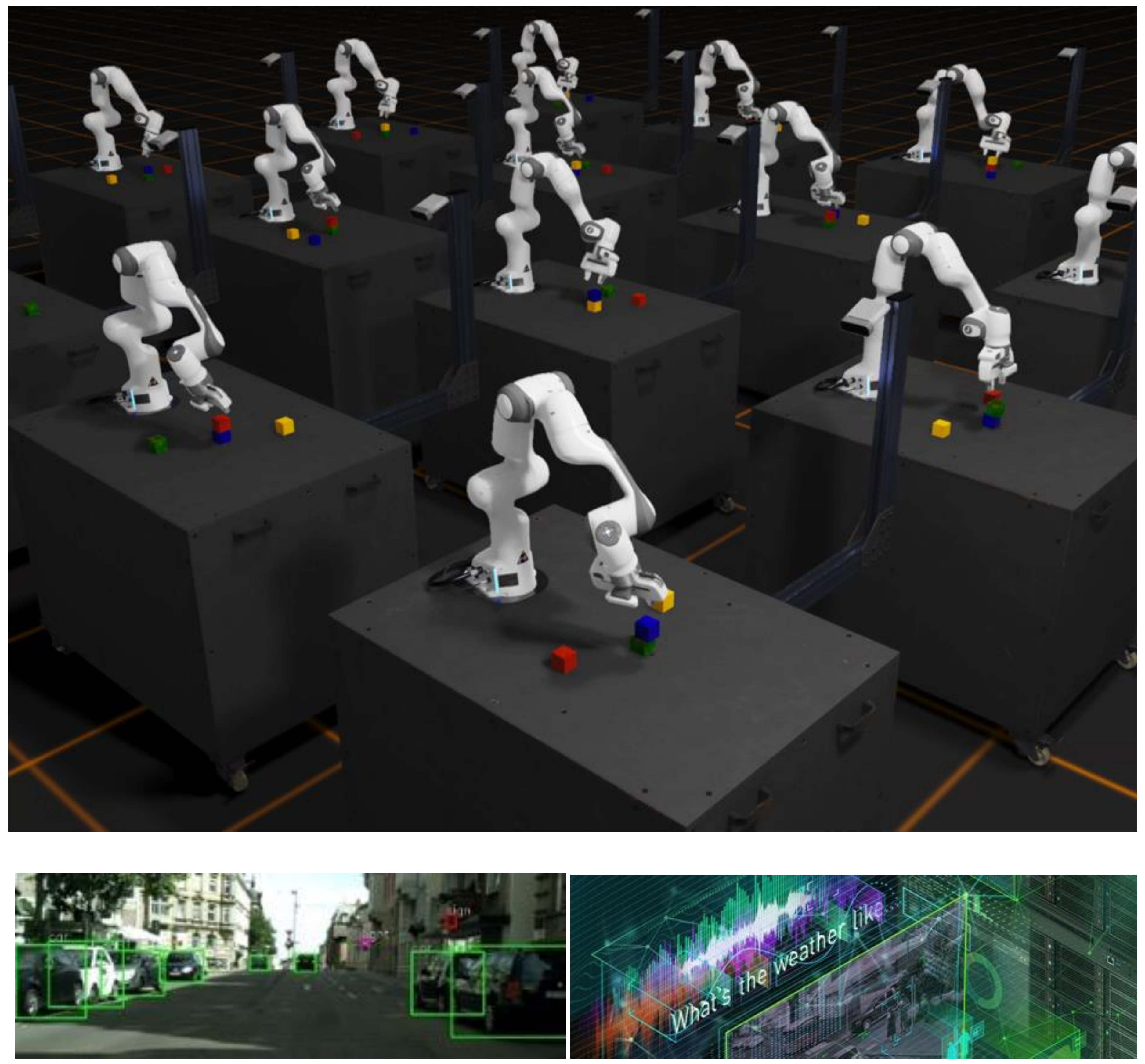
AGENDA

Overview of GPU Computing

NVIDIA DELIVERS END-TO-END ACCELERATION


GPU Computing


Computer Graphics



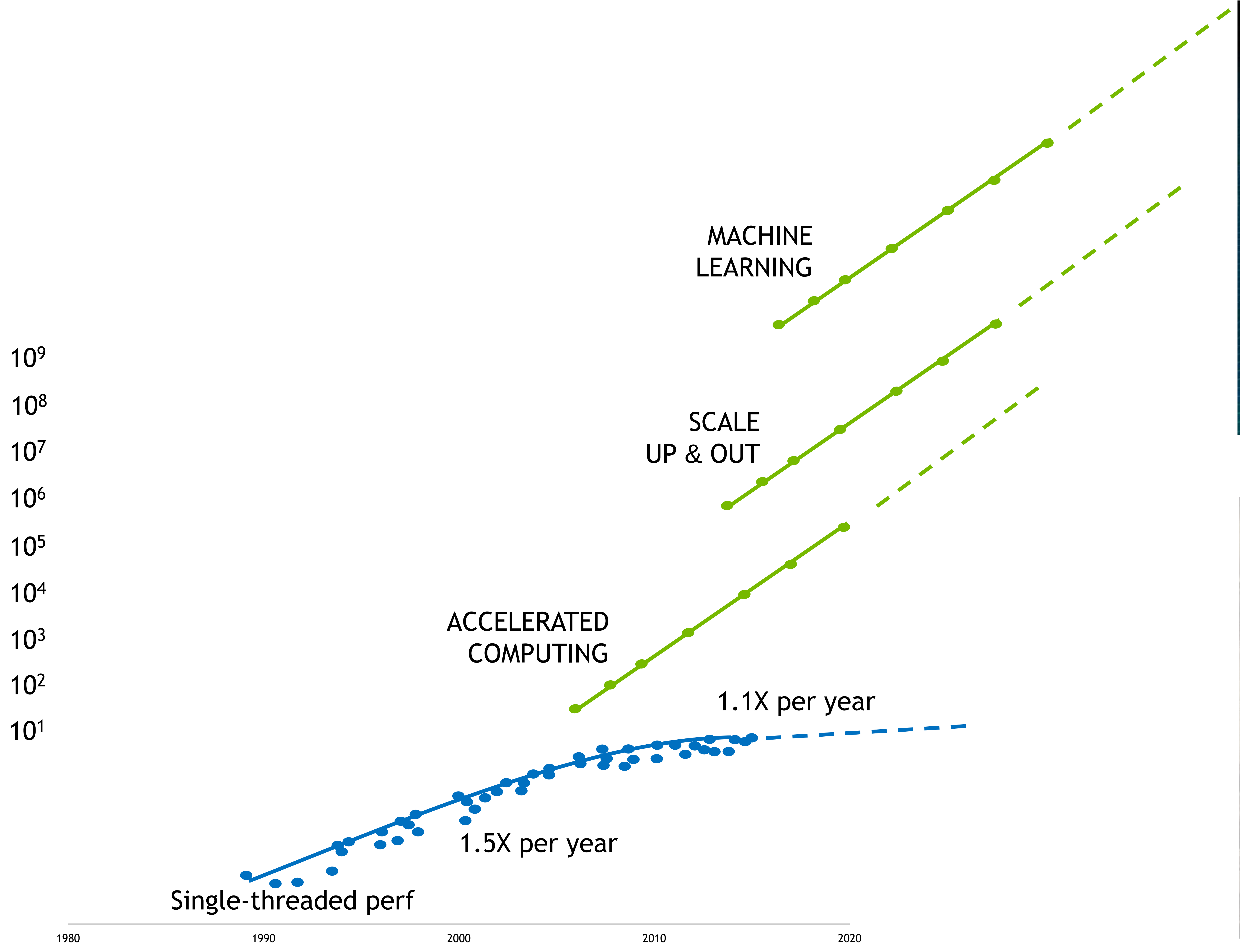
Artificial Intelligence

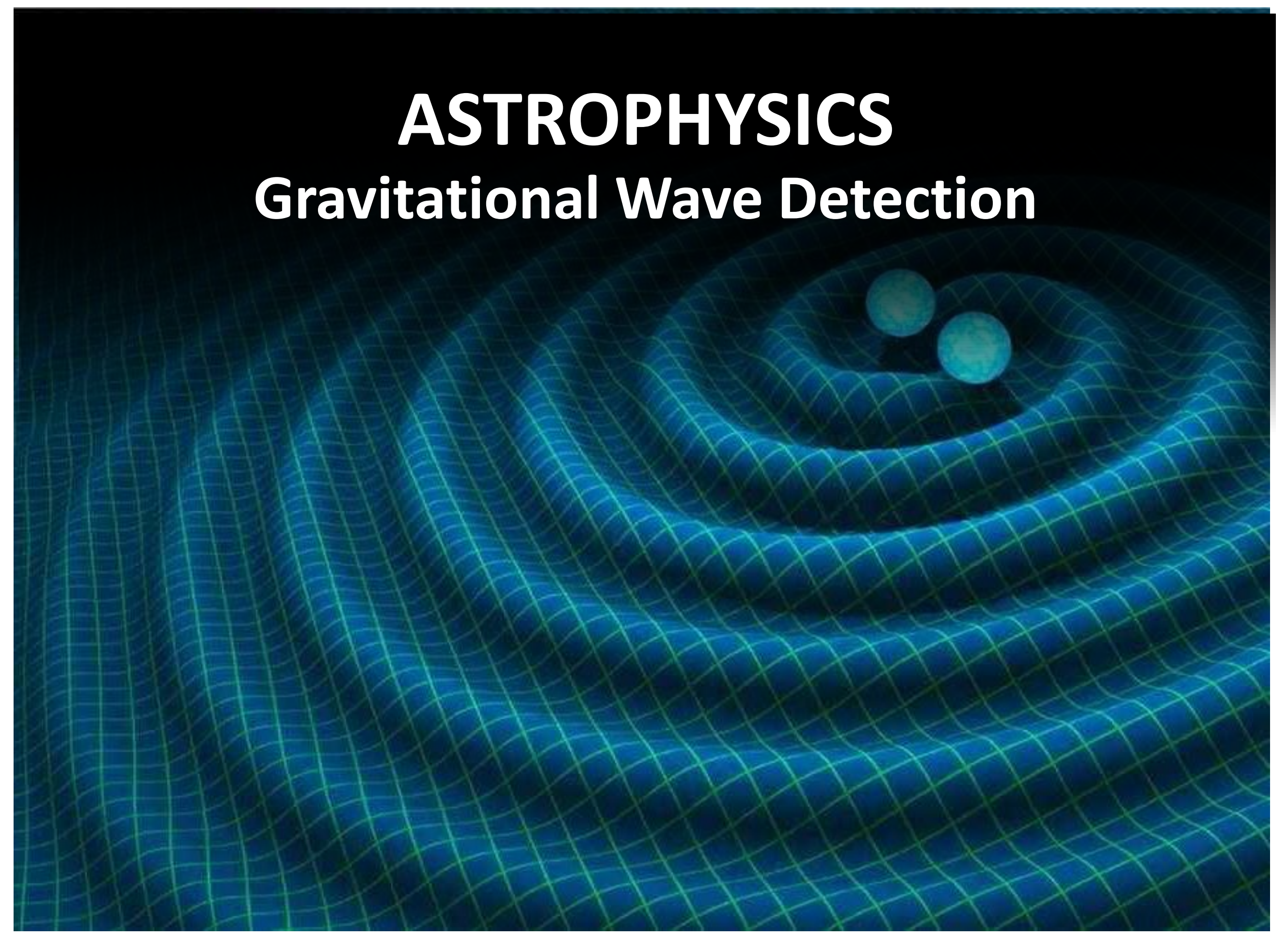
ACCELERATED COMPUTING WITH GPUS

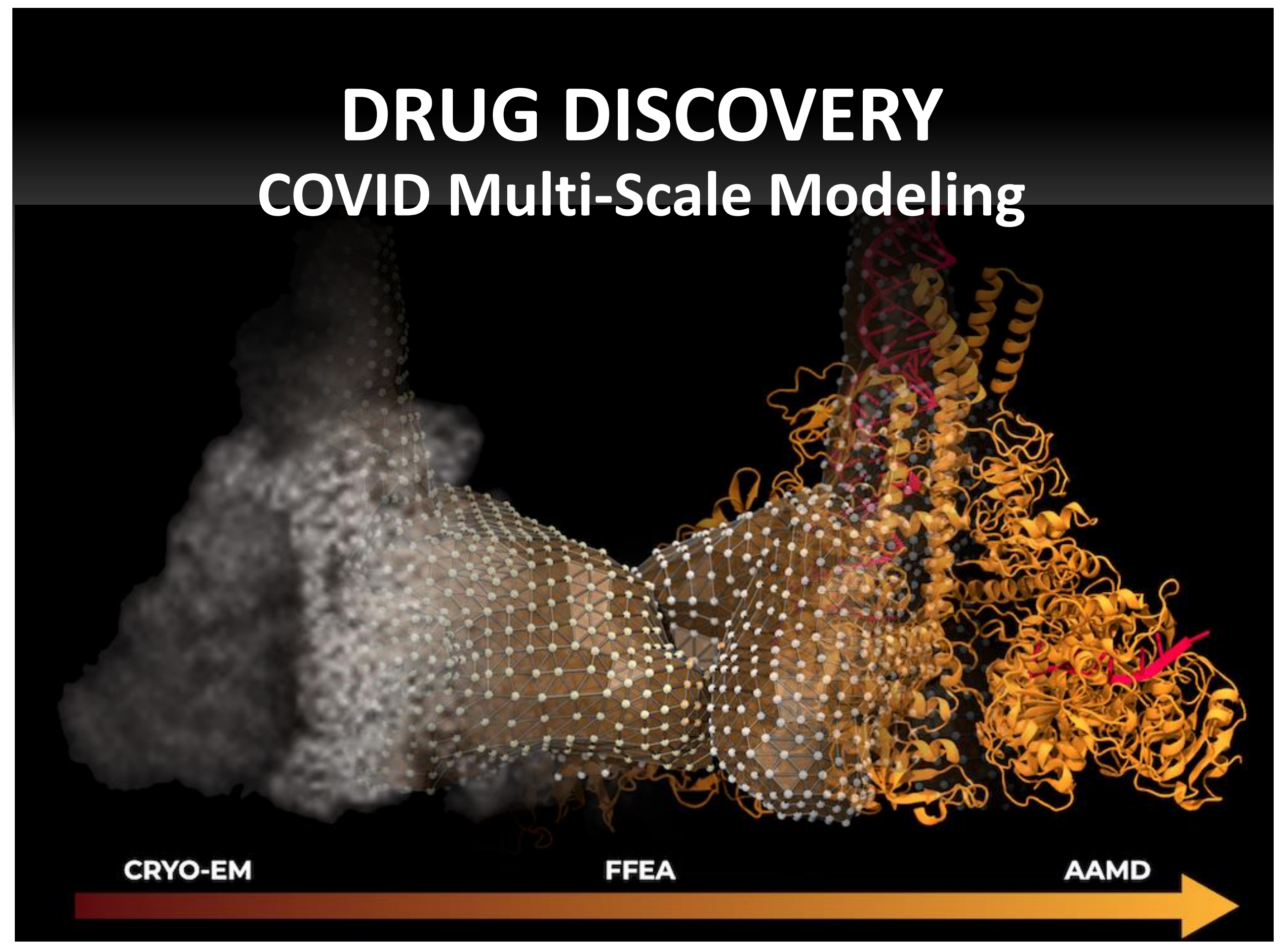
GPU
Optimized for Parallel Tasks
5% of Code
Rest of Sequential CPU Code
CPU
Optimized for Serial Tasks
+

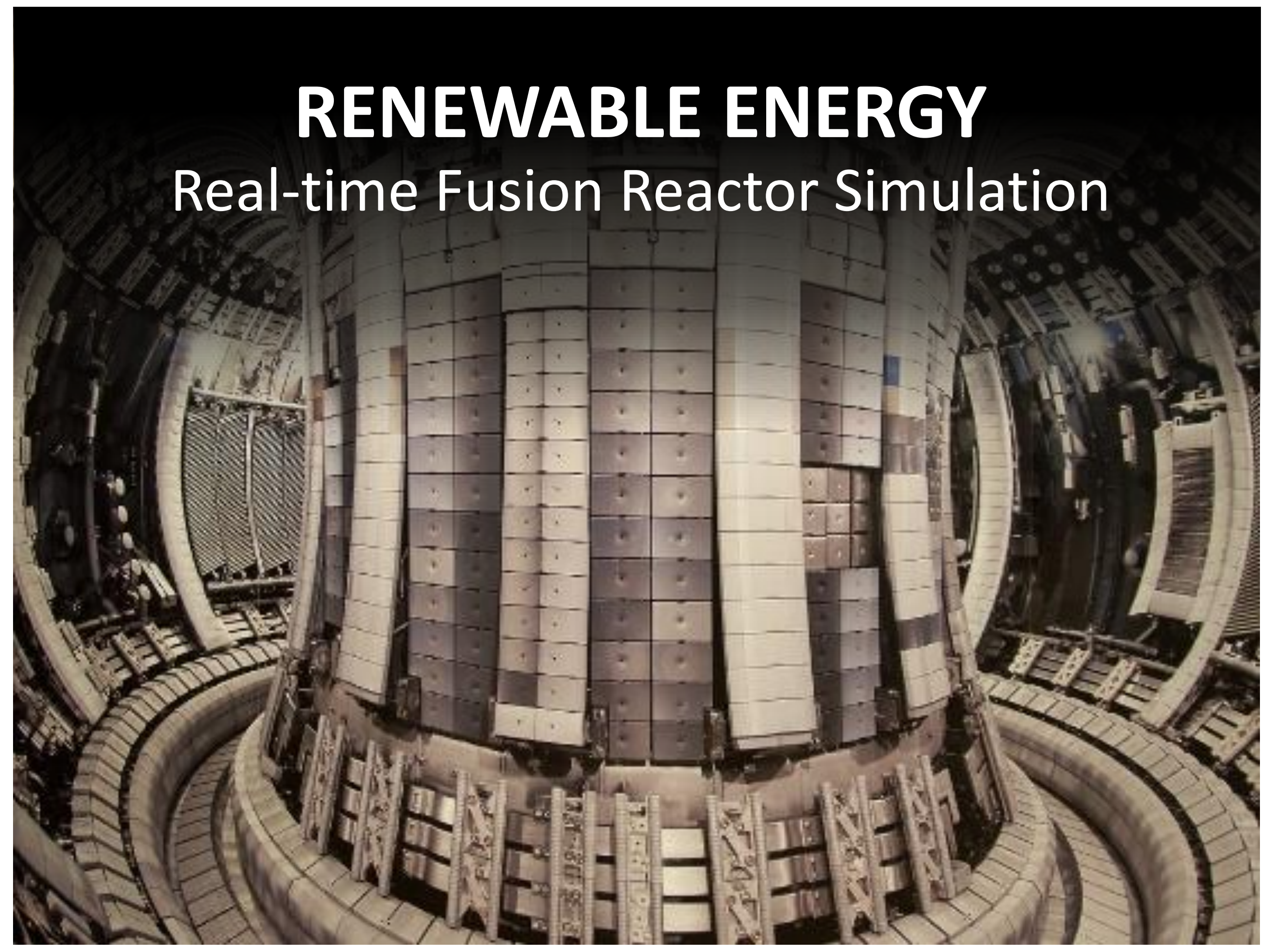
MILLION-X SPEEDUP FOR INNOVATION AND DISCOVERY

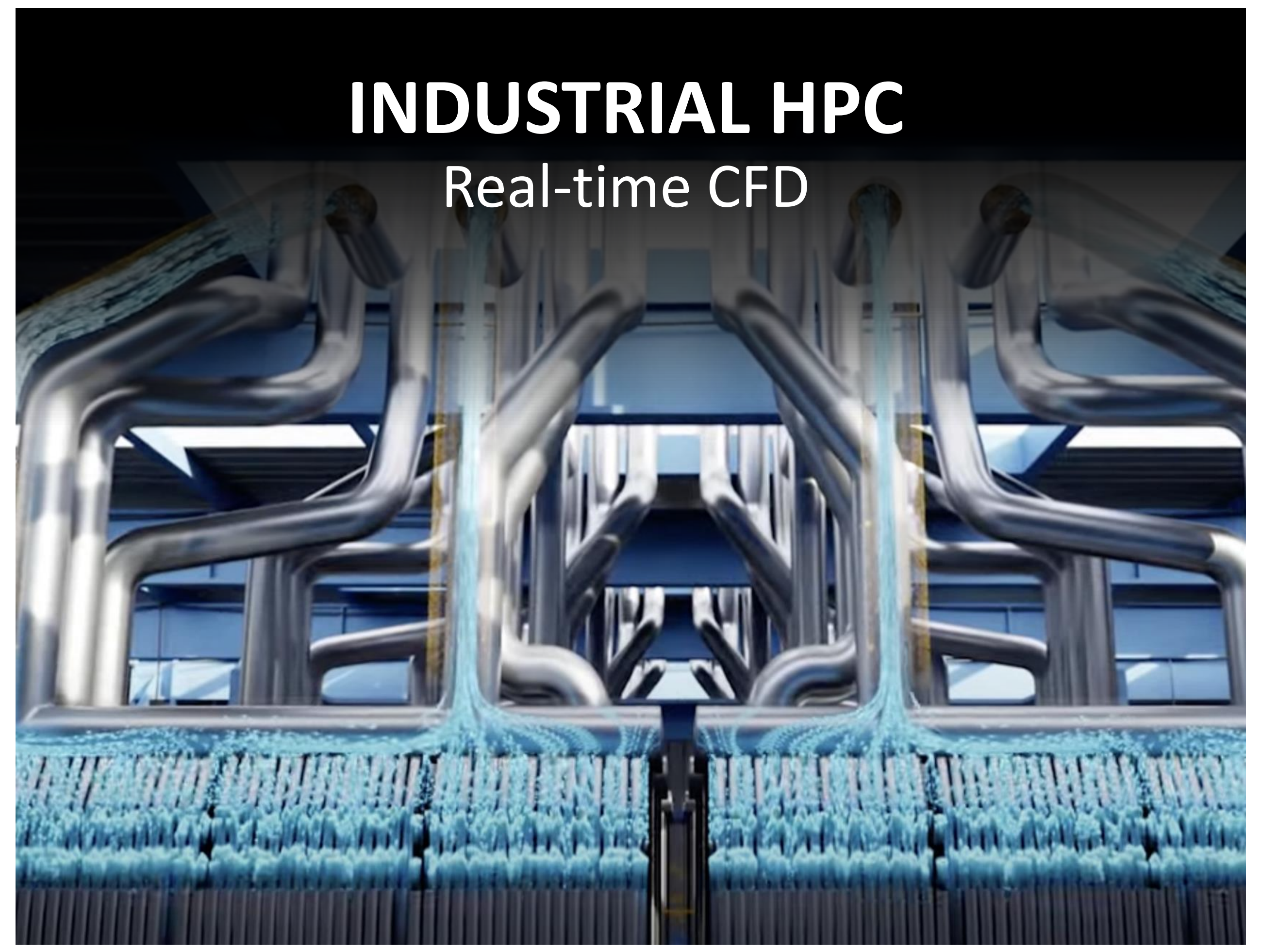
Combination of Accelerated Computing, Data Center Scale and AI











PROGRAMMING THE NVIDIA PLATFORM

CPU, GPU, and Network

ACCELERATED STANDARD LANGUAGES

ISO C++, ISO Fortran

```
std::transform(par, x, x+n, y, y,
    [=](float x, float y){ return y +
a*x; }
);
```

```
do concurrent (i = 1:n)
   y(i) = y(i) + a*x(i)
enddo
```

```
import cunumeric as np
…
def saxpy(a, x, y):
    y[:] += a*x
```

INCREMENTAL PORTABLE OPTIMIZATION

OpenACC, OpenMP

```
#pragma acc data copy(x,y) {
...
std::transform(par, x, x+n, y, y,
    [=](float x, float y){
        return y + a*x;
});
...
}
```

```
#pragma omp target data map(x,y) {
...
std::transform(par, x, x+n, y, y,
    [=](float x, float y){
        return y + a*x;
});
...
}
```

PLATFORM SPECIALIZATION

CUDA

```
__global__
void saxpy(int n, float a,
           float *x, float *y) {
  int i = blockIdx.x*blockDim.x +
          threadIdx.x;
  if (i < n) y[i] += a*x[i];
}

int main(void) {
  ...
  cudaMemcpy(d_x, x, ...);
  cudaMemcpy(d_y, y, ...);

  saxpy<<<(N+255)/256,256>>>(...);

  cudaMemcpy(y, d_y, ...);
```

ACCELERATION LIBRARIES

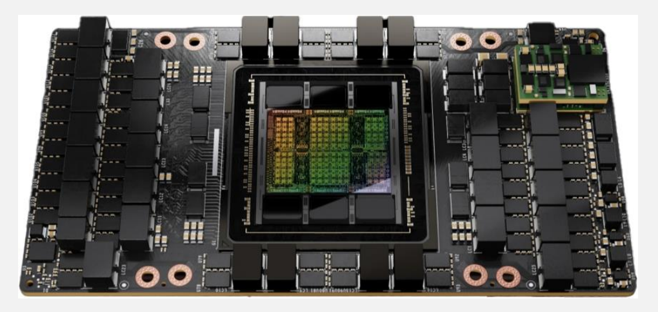
Core | Math | Communication | Data Analytics | AI | Quantum

THE GAP BETWEEN PYTHON AND GPUS

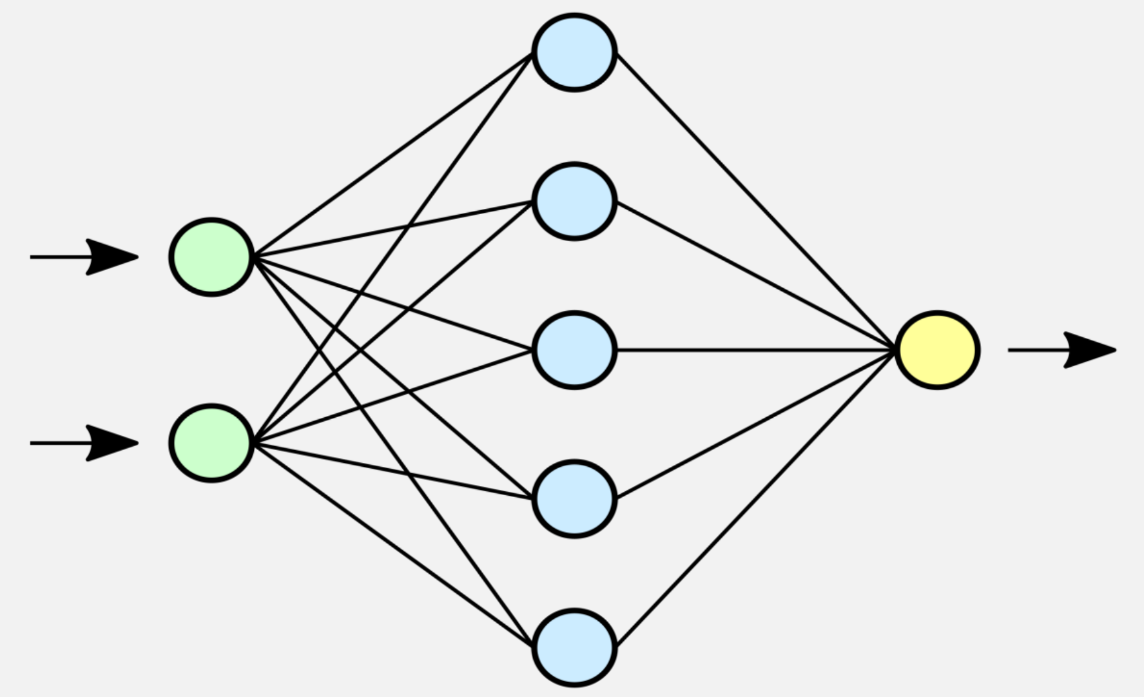
Python

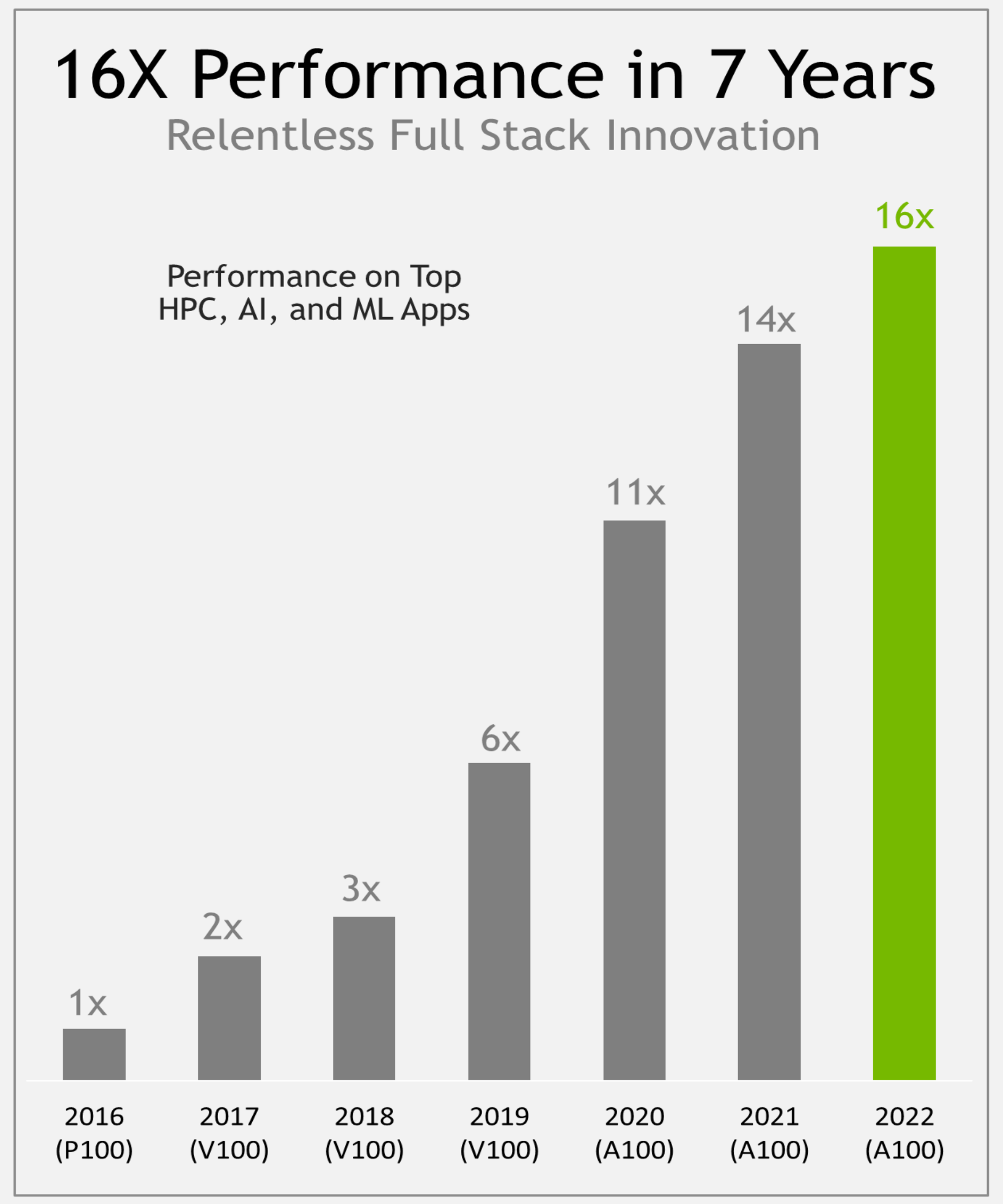
- High-level
- Interactive, versatile
- Easy to prototype
- Automatic memory management
- Dynamic typing
- Robust package management (conda, pip)
- MANY libraries available

NVIDIA GPUs



- C, Fortran
- Compiled
- High-Performance



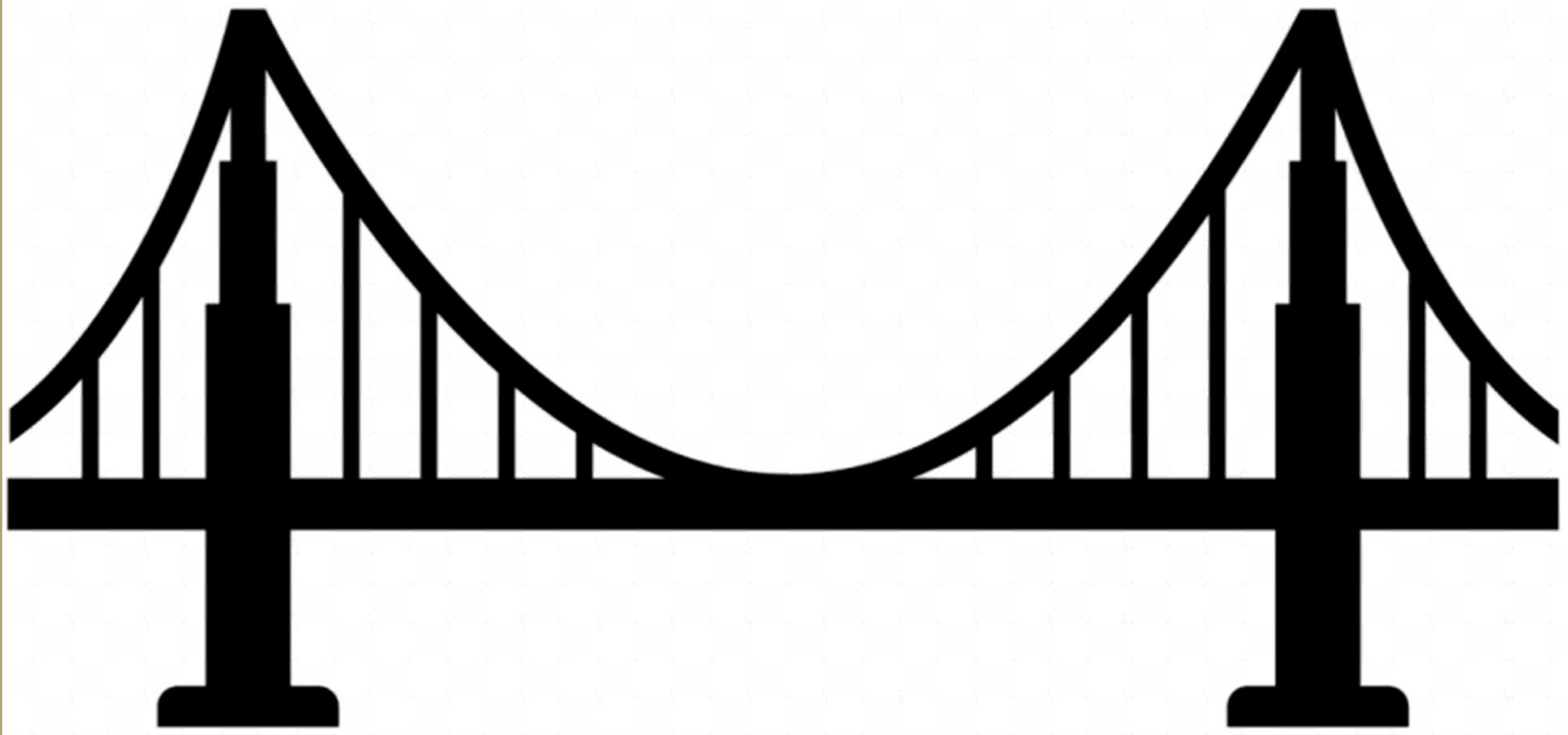

16X Performance in 7 Years
Relentless Full Stack Innovation
Performance on Top HPC, AI, and ML Apps
1x
2x
3x
6x
11x
14x
16x
2016 (P100)
2017 (V100)
2018 (V100)
2019 (V100)
2020 (A100)
2021 (A100)
2022 (A100)

BRIDGING THE GAP BETWEEN PYTHON AND GPUS

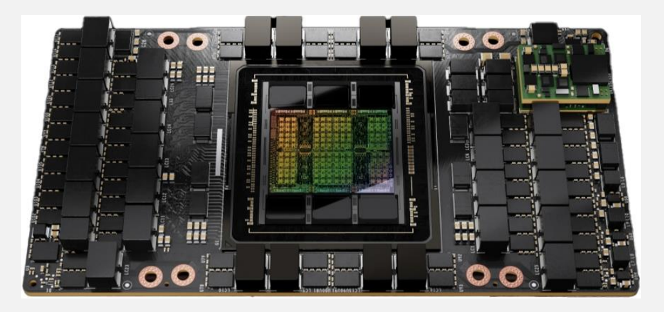
Python

- High-level
- Interactive, versatile
- Easy to prototype
- Automatic memory management
- Dynamic typing
- Robust package management (conda, pip)
- MANY libraries available

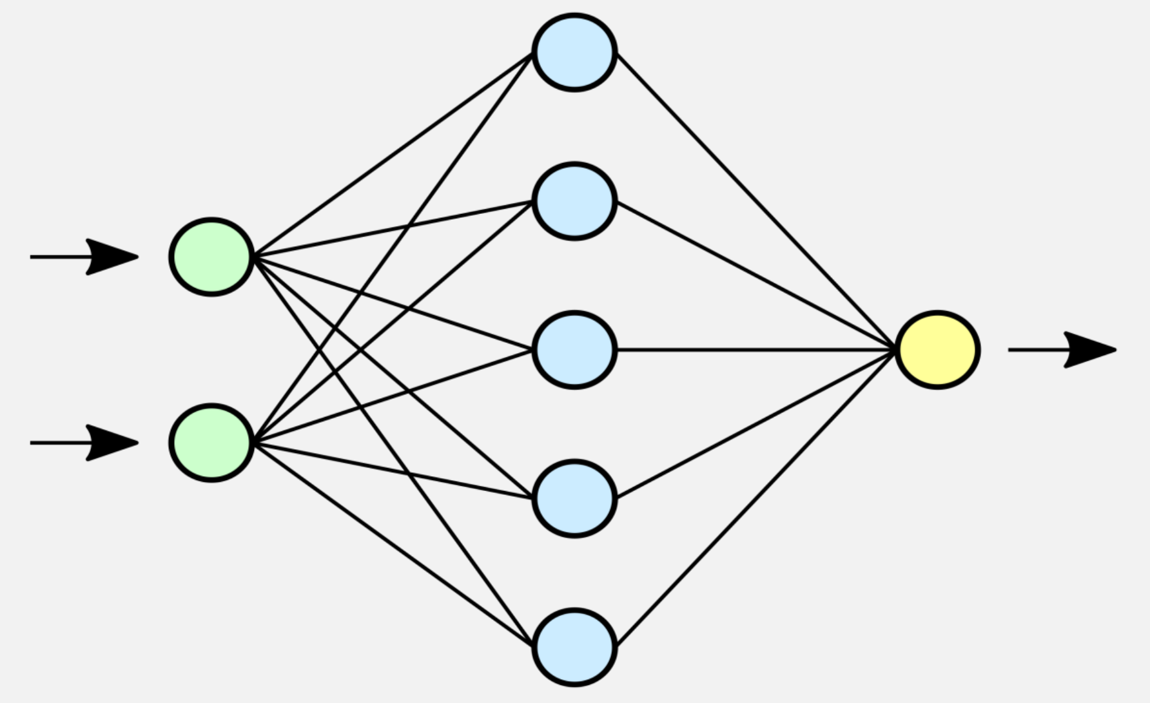
JIT Compilation → Low-Level Language Bindings → CUDA-X Libraries

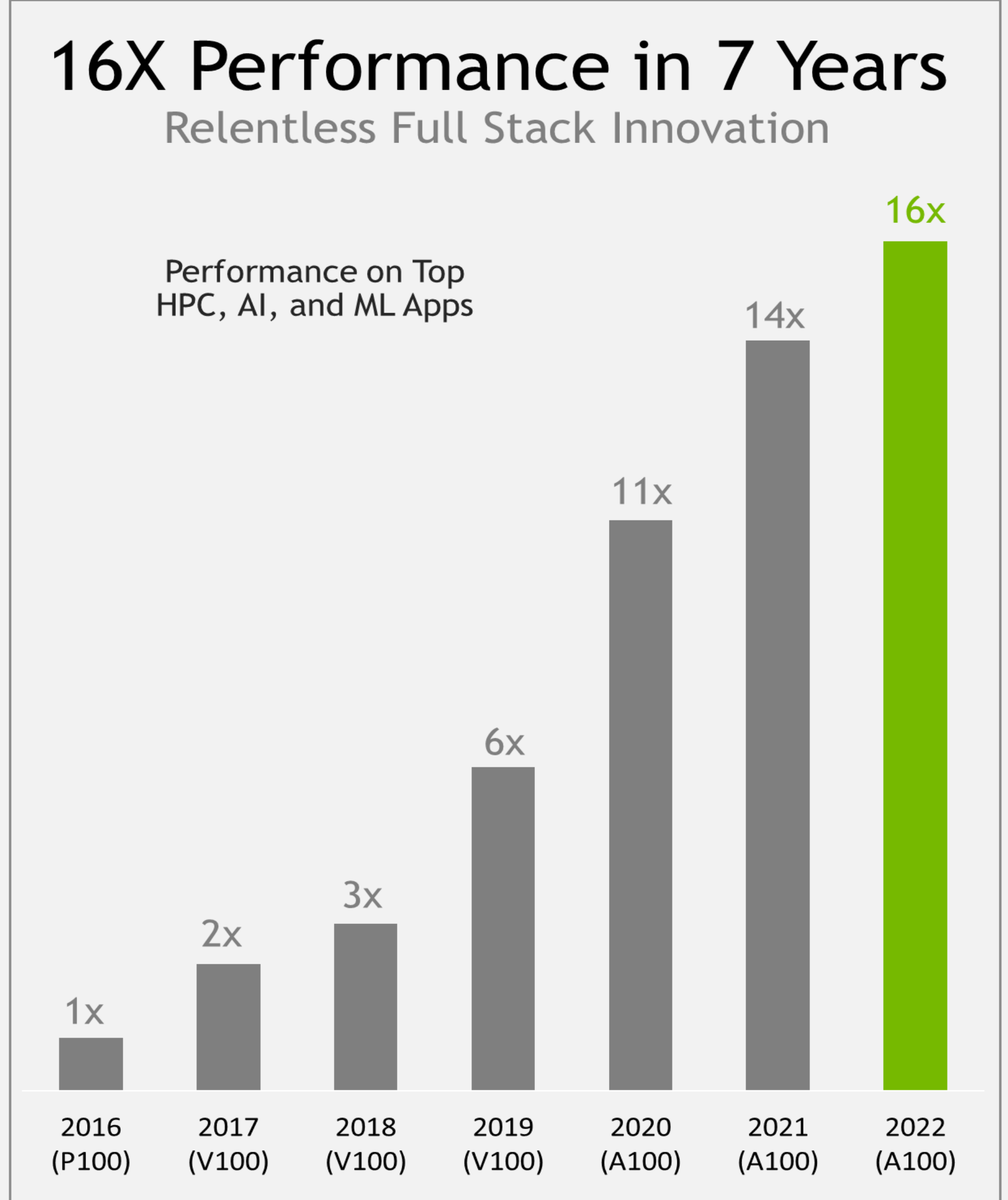


NVIDIA GPUs



- C, Fortran
- Compiled
- High-Performance




16X Performance in 7 Years
Relentless Full Stack Innovation
Performance on Top HPC, AI, and ML Apps
1x
2x
3x
6x
11x
14x
16x
2016 (P100)
2017 (V100)
2018 (V100)
2019 (V100)
2020 (A100)
2021 (A100)
2022 (A100)

A FEW GENERAL TIPS FOR SUCCESSFUL GPU COMPUTING

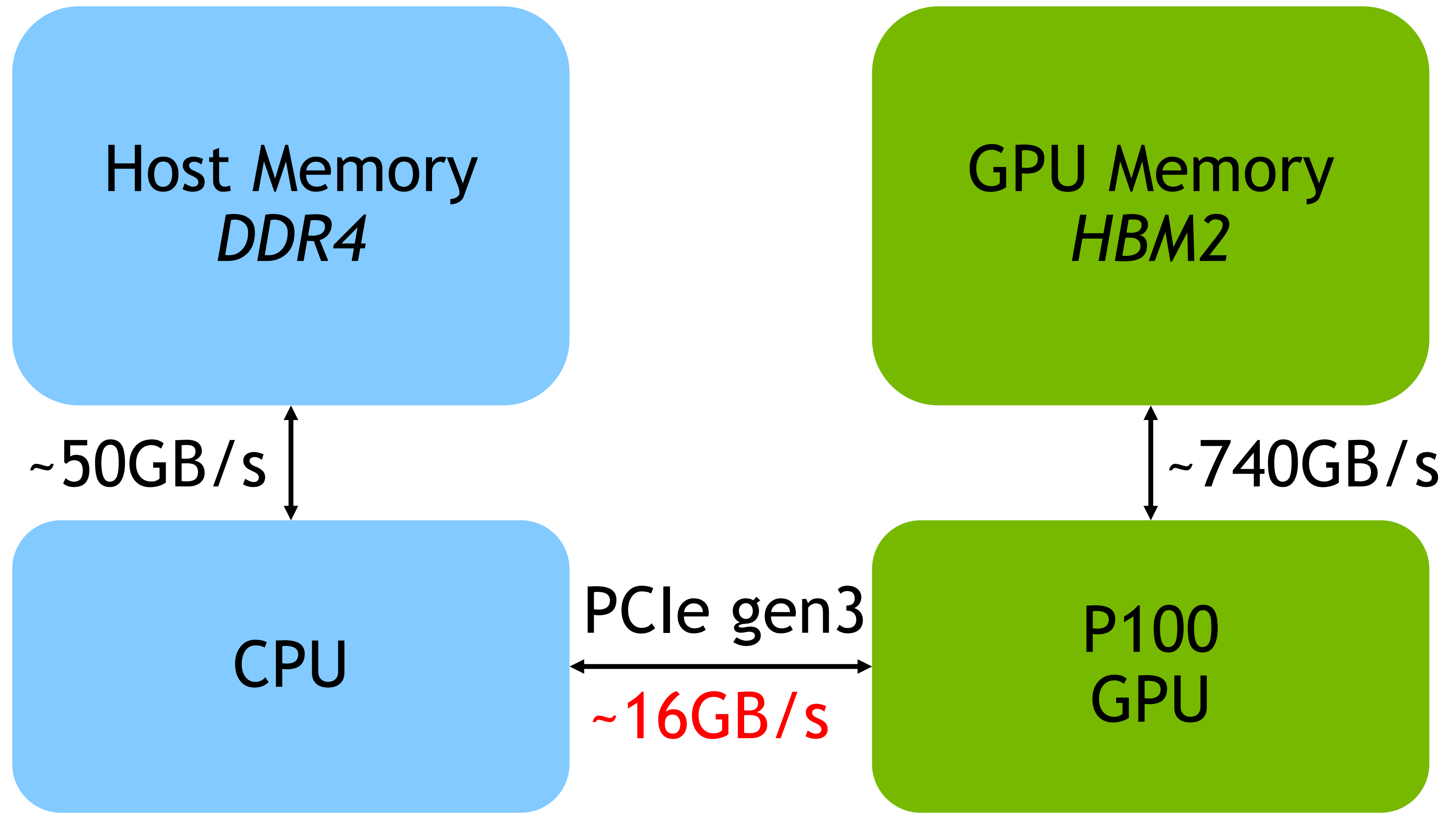
- **Minimize data movement to and from the GPU**
  - What happens on the GPU, stays on the GPU!
  - PCI express is a bottleneck for data movement
  - Try NVLink for GPU peer-to-peer, 600 GB/s!

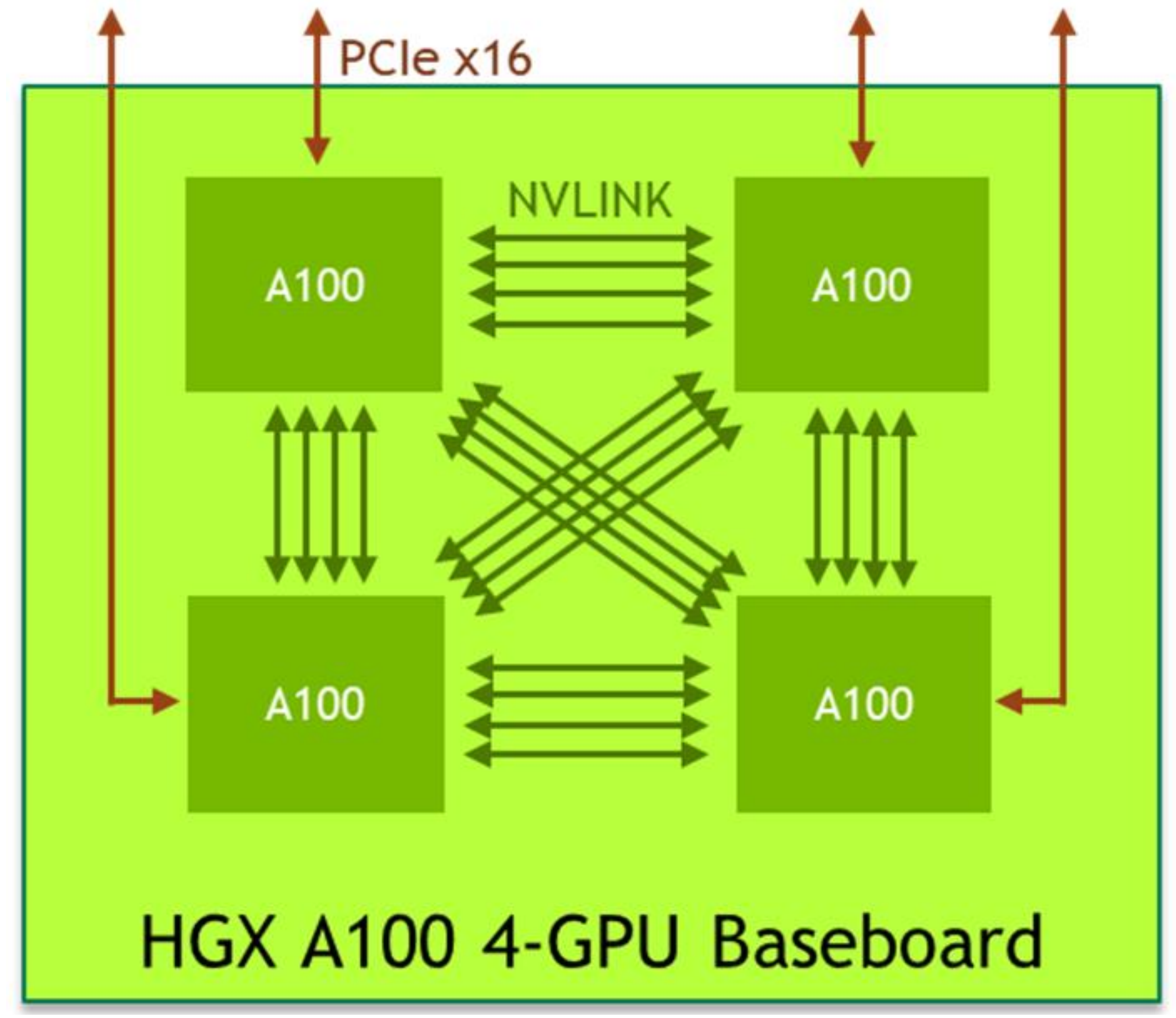
- **GPUs are parallel processing machines**
  - Leave serial operations to the CPU
  - Look for high arithmetic intensity, chunky loops, dense linear algebra
  - Experiment with reduced precision, mixed-precision iterative refinement
  - High memory bandwidth - Fast FFTs.

- **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/

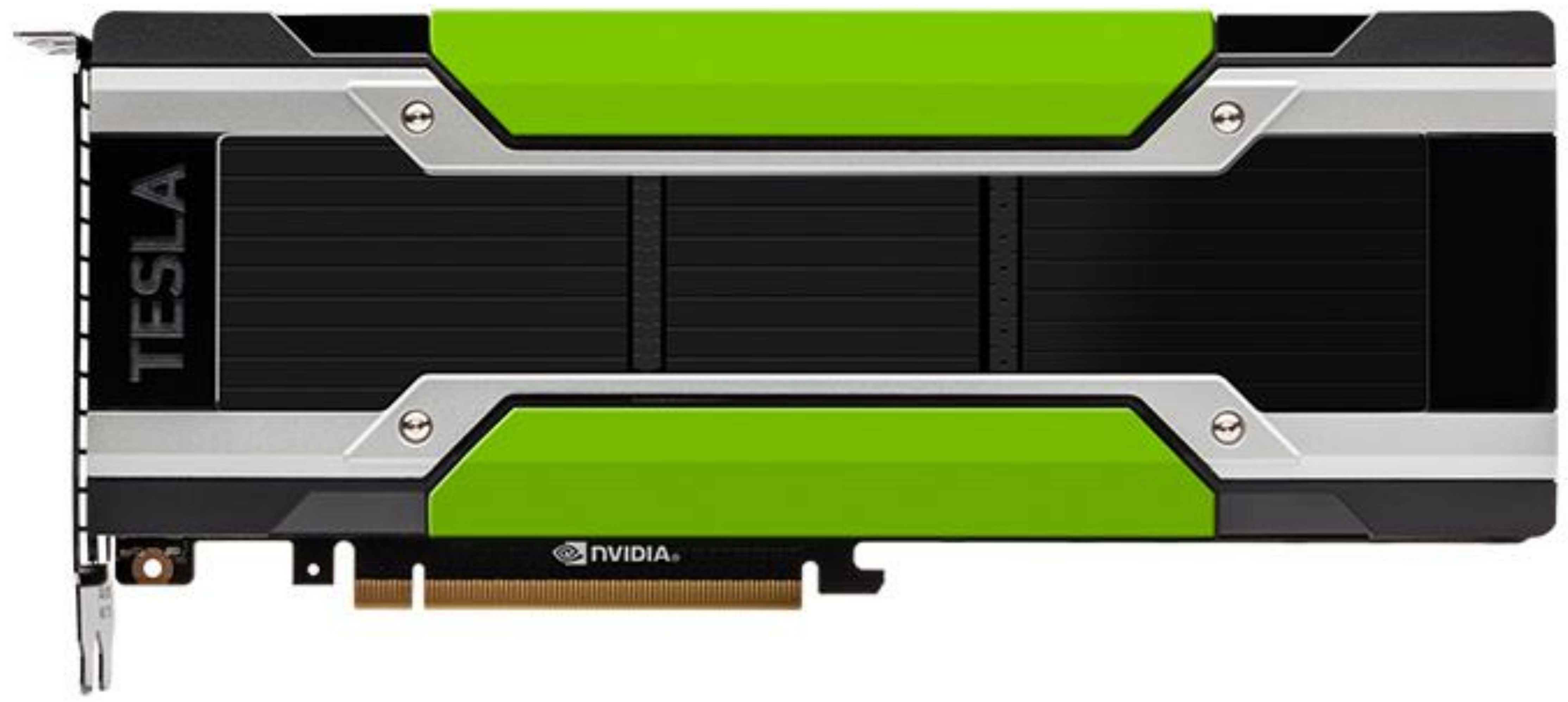




Accessing the Workshop Materials

DEMO SYSTEM

NVIDIA P100

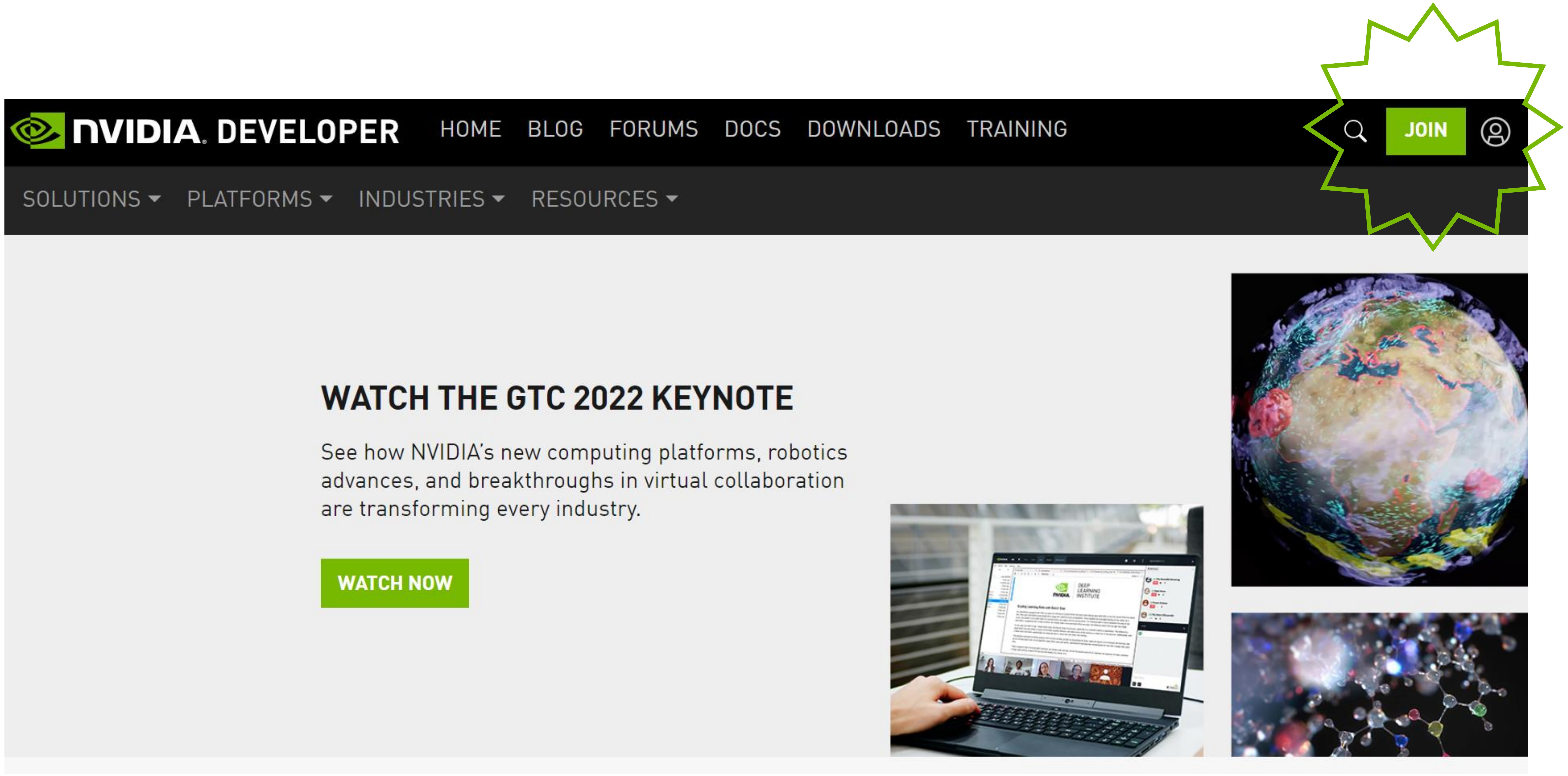


	P100 for PCIe-Based Servers
Double-Precision Performance	4.7 teraFLOPS
Single-Precision Performance	9.3 teraFLOPS
Half-Precision Performance	18.7 teraFLOPS
NVIDIA NVLink Interconnect Bandwidth	-
PCIe x16 Interconnect Bandwidth	32 GB/s
CoWoS HBM2 Stacked Memory Capacity	16 GB or 12 GB
CoWoS HBM2 Stacked Memory Bandwidth	732 GB/s or 549 GB/s
Enhanced Programmability with Page Migration Engine	✓
ECC Protection for Reliability	✓
Server-Optimized for Data Center Deployment	✓

Results contained within this presentation reflect workloads run on either a single P100 GPU or a multi-GPU setup. We have sized the problems to fit the available memory for this GPU. See the results pages at the end of the presentation for results on a larger DGX Station with four A100 Tensor Core GPUs each with 80 GB memory.

ACCESSING THE WORKSHOP MATERIALS

Do you have an NVIDIA Developer Account?
https://developer.nvidia.com/



WATCH THE GTC 2022 KEYNOTE

See how NVIDIA's new computing platforms, robotics advances, and breakthroughs in virtual collaboration are transforming every industry.

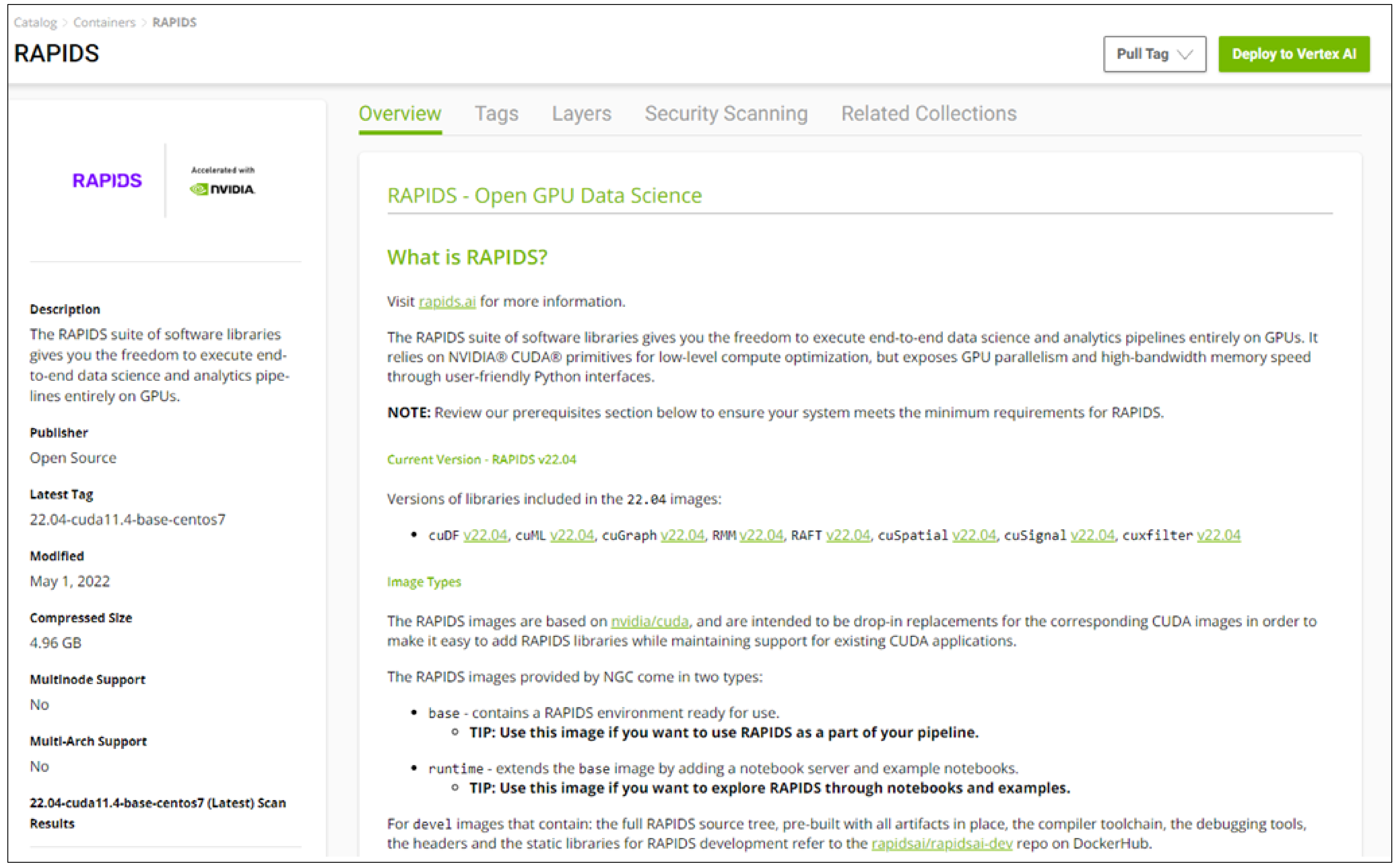
Notebook 1: Introduction to CuPy

Working in Section:
"Introduction to Workshop Lab Environment"

REPLICATING THE WORKSHOP ENVIRONMENT

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

RAPIDS
Release Selector
https://rapids.ai



RAPIDS

Description
The RAPIDS suite of software libraries gives you the freedom to execute end-to-end data science and analytics pipelines entirely on GPUs.

Publisher
Open Source

Latest Tag
22.04-cuda11.4-base-centos7

Modified
May 1, 2022

Compressed Size
4.96 GB

Multinode Support
No

Multi-Arch Support
No

22.04-cuda11.4-base-centos7 (Latest) Scan Results

RAPIDS - Open GPU Data Science

What is RAPIDS?

Visit rapids.ai for more information.

The RAPIDS suite of software libraries gives you the freedom to execute end-to-end data science and analytics pipelines entirely on GPUs. It relies on NVIDIA® CUDA® primitives for low-level compute optimization, but exposes GPU parallelism and high-bandwidth memory speed through user-friendly Python interfaces.

NOTE: Review our prerequisites section below to ensure your system meets the minimum requirements for RAPIDS.

Current Version - RAPIDS v22.04

Versions of libraries included in the `22.04` images:

- `cuDF` v22.04, `cuML` v22.04, `cuGraph` v22.04, `RMM` v22.04, `RAFT` v22.04, `cuSpatial` v22.04, `cuSignal` v22.04, `cuxfilter` v22.04

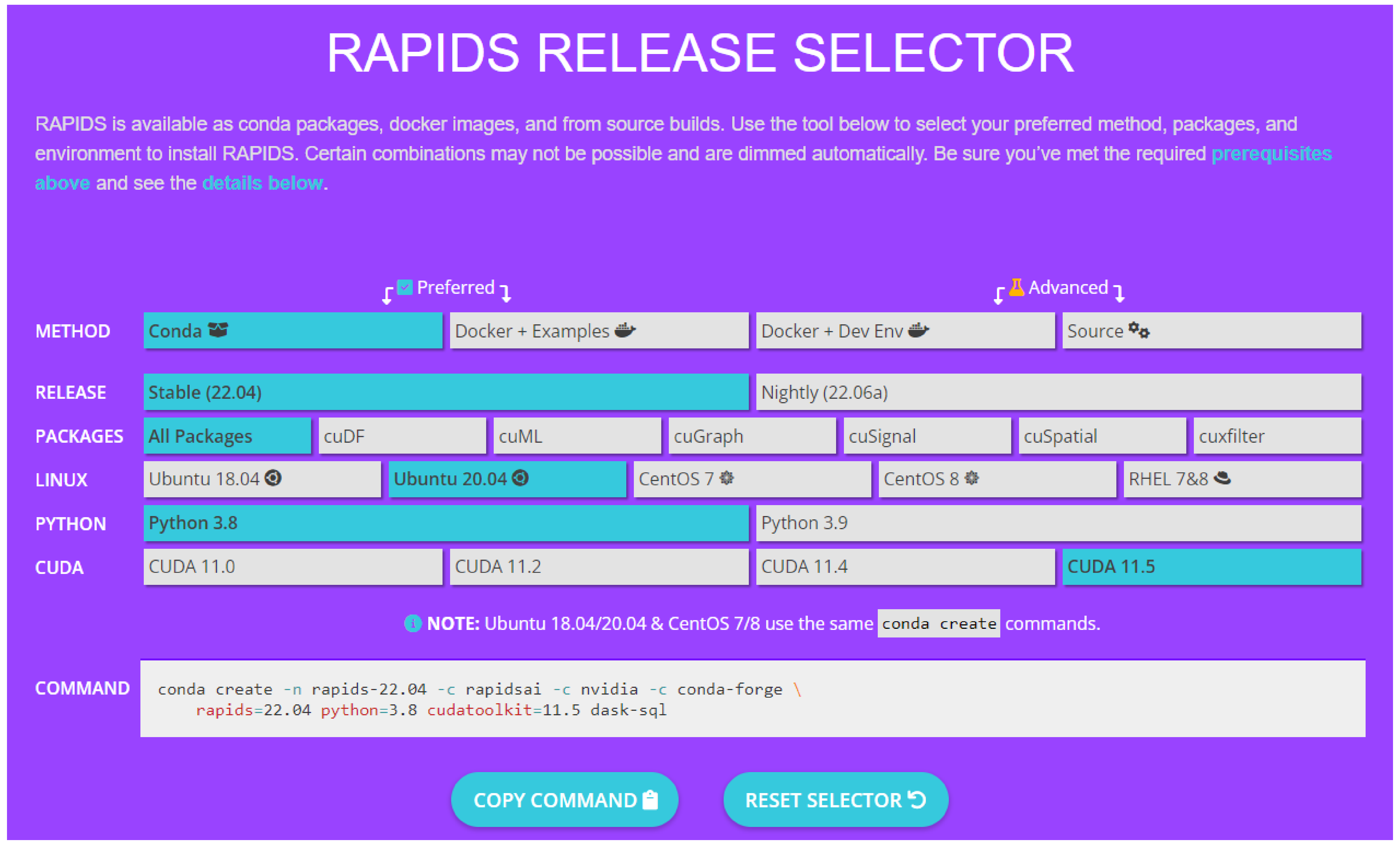
Image Types

The RAPIDS images are based on nvidia/cuda, and are intended to be drop-in replacements for the corresponding CUDA images in order to make it easy to add RAPIDS libraries while maintaining support for existing CUDA applications.

The RAPIDS images provided by NGC come in two types:

- `base` - contains a RAPIDS environment ready for use.
  - **TIP: Use this image if you want to use RAPIDS as a part of your pipeline.**
- `runtime` - extends the `base` image by adding a notebook server and example notebooks.
  - **TIP: Use this image if you want to explore RAPIDS through notebooks and examples.**

For `devel` images that contain: the full RAPIDS source tree, pre-built with all artifacts in place, the compiler toolchain, the debugging tools, the headers and the static libraries for RAPIDS development refer to the rapidsai/rapidsai-dev repo on DockerHub.



RAPIDS RELEASE SELECTOR

RAPIDS is available as conda packages, docker images, and from source builds. Use the tool below to select your preferred method, packages, and environment to install RAPIDS. Certain combinations may not be possible and are dimmed automatically. Be sure you've met the required prerequisites above and see the details below.

	Preferred		Advanced	
METHOD	[✓ Conda]	Docker + Examples	Docker + Dev Env	Source

RELEASE	[✓ Stable (22.04)]	Nightly (22.06a)					
PACKAGES	[✓ All Packages]	cuDF	cuML	cuGraph	cuSignal	cuSpatial	cuxfilter
LINUX	Ubuntu 18.04	[✓ Ubuntu 20.04]	CentOS 7	CentOS 8	RHEL 7&8		
PYTHON	[✓ Python 3.8]	Python 3.9					
CUDA	CUDA 11.0	CUDA 11.2	CUDA 11.4	[✓ CUDA 11.5]			

NOTE: Ubuntu 18.04/20.04 & CentOS 7/8 use the same `conda create` commands.

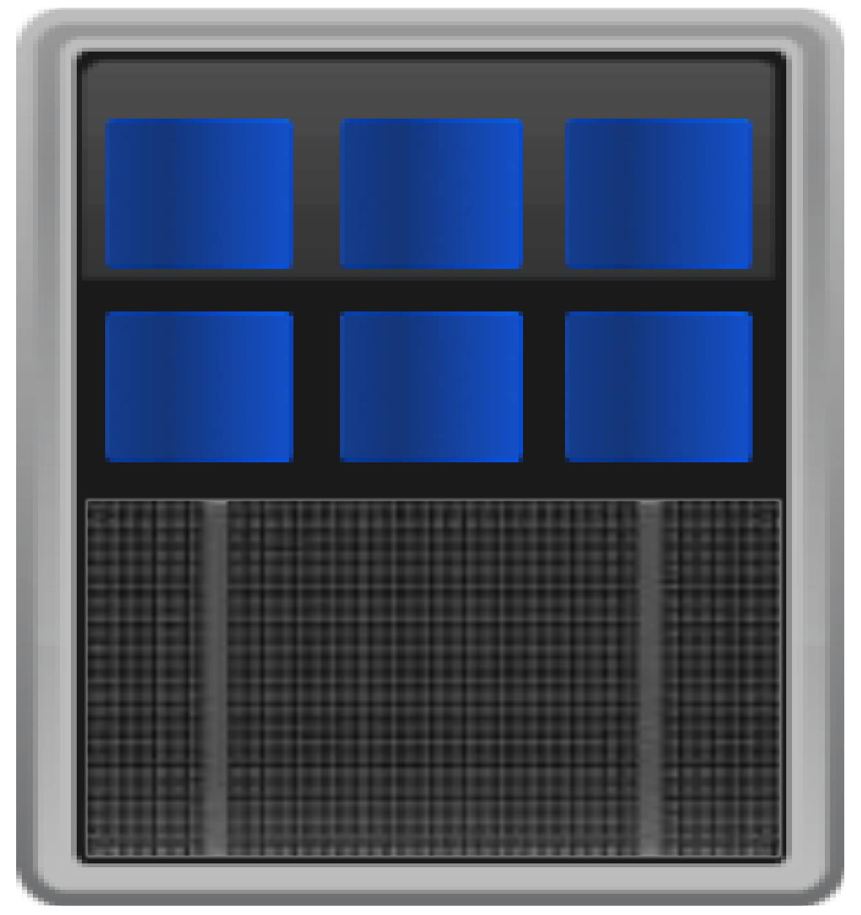
COMMAND

```
conda create -n rapids-22.04 -c rapidsai -c nvidia -c conda-forge \
    rapids=22.04 python=3.8 cudatoolkit=11.5 dask-sql
```

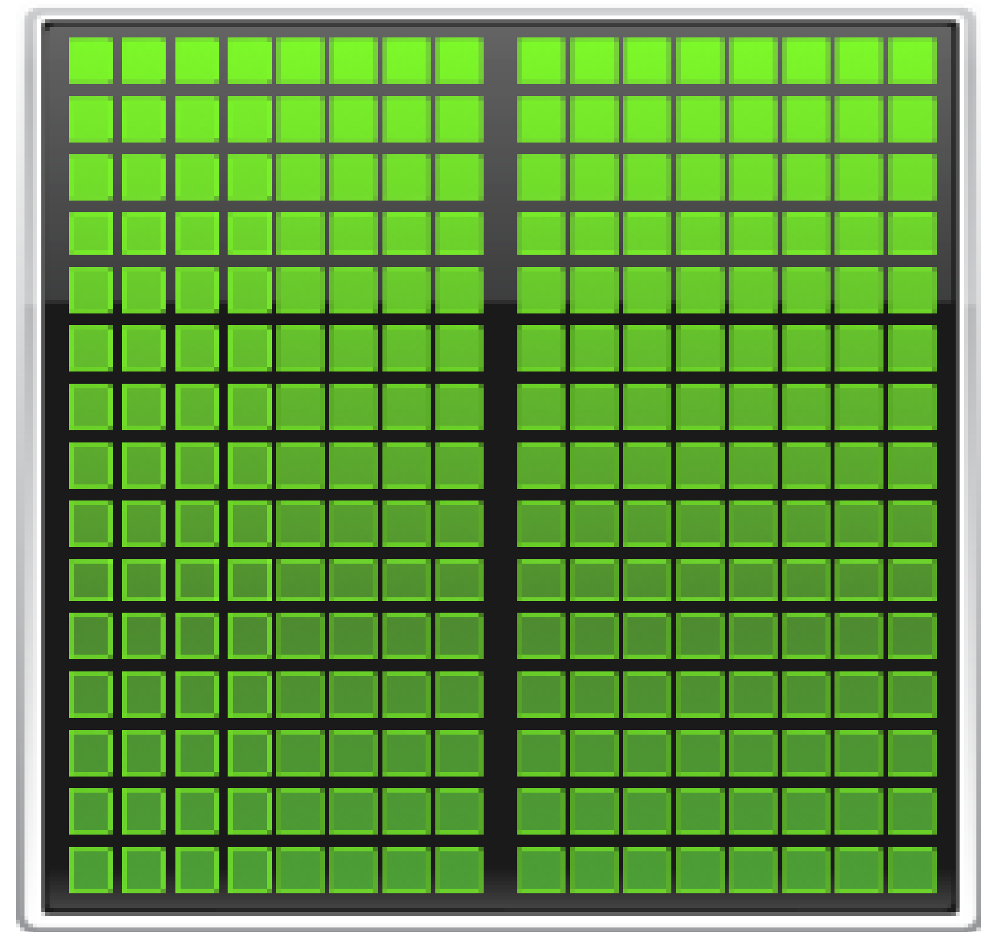
GPU-Accelerated Numerical Computing with ***CuPy***

NUMERICAL COMPUTING IN PYTHON

NumPy



CuPy



- 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.lingalg.qr(A)
```

AFTER

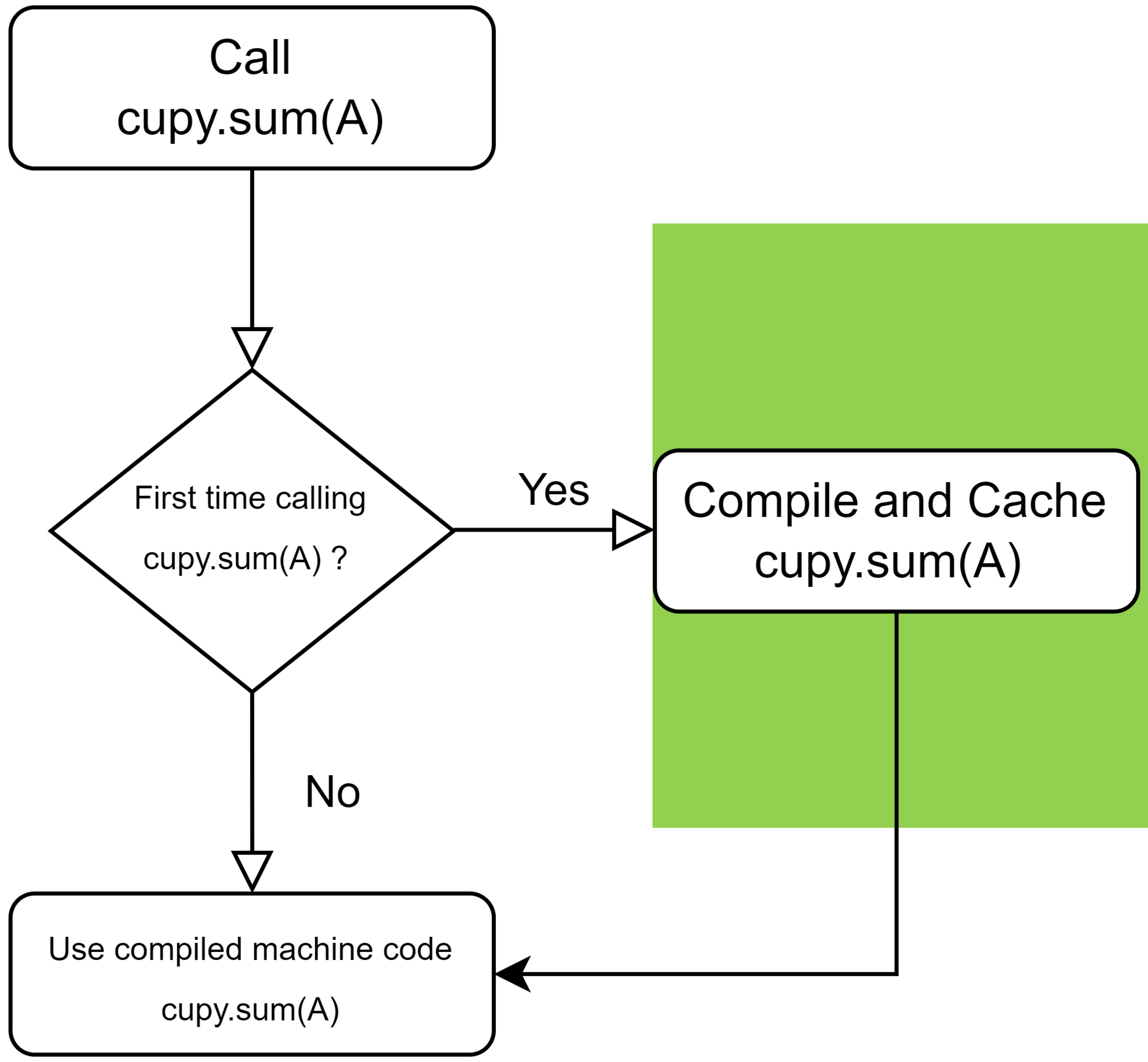
```
import cupy as cp

size = 4096
A = cp.random.randn(size,size)

Q, R = cp.lingalg.qr(A)
```

52x Speedup!

Notebook 1: Introduction to CuPy

Working in Section:
“Introduction to CuPy”

KERNEL OVERHEAD



JIT Compilation

- What is the size of A?
- What is the datatype?
- Which GPU-accelerated libraries are available?
- Compiler optimizations for custom kernels

KERNEL OVERHEAD

Kernel Launch Overhead

- Python wrappers around CUDA API call for kernel execution
- Kicking off the CUDA kernel (OS, CUDA Driver)
- GPU context switching from another task

- *Cost is on the order of microseconds*

Defensive Strategies

- Increase dataset size
- Combine small kernels together
  - @cupy.fuse for elementwise or reduction kernels
  - Dynamic programming
- Use CUDA streams to backfill GPU work queue
  - Available via CuPy, Numba

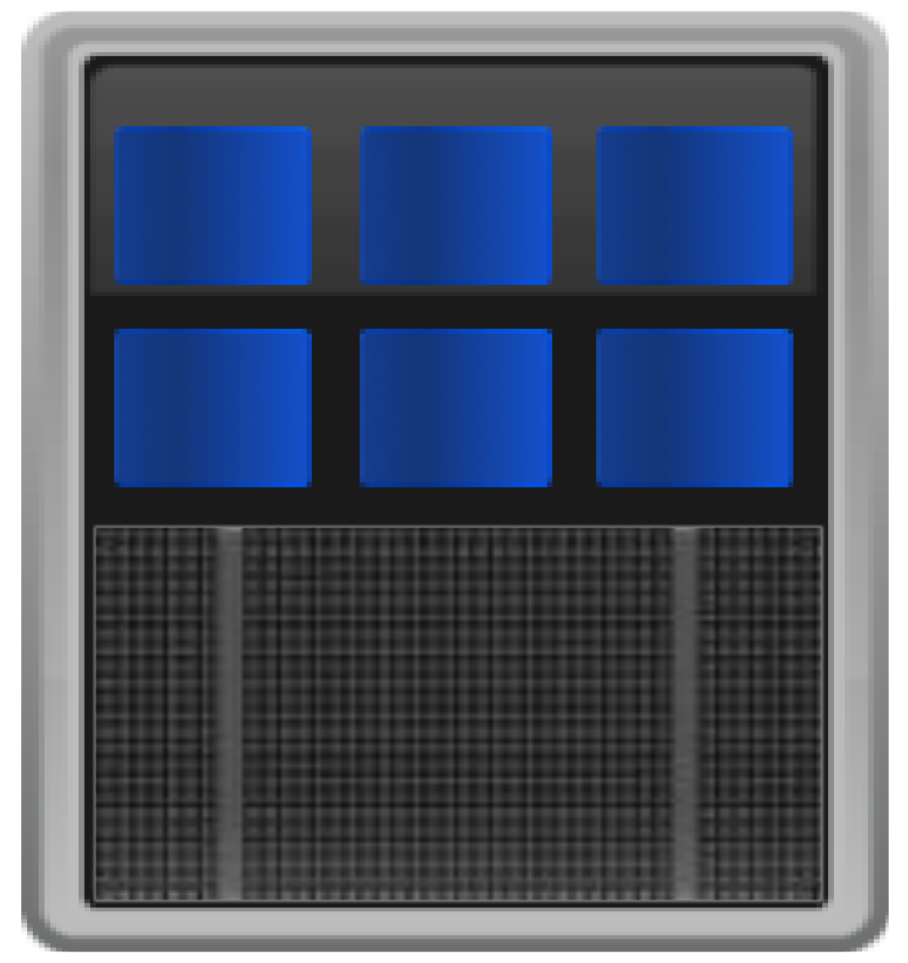
Python Decorators

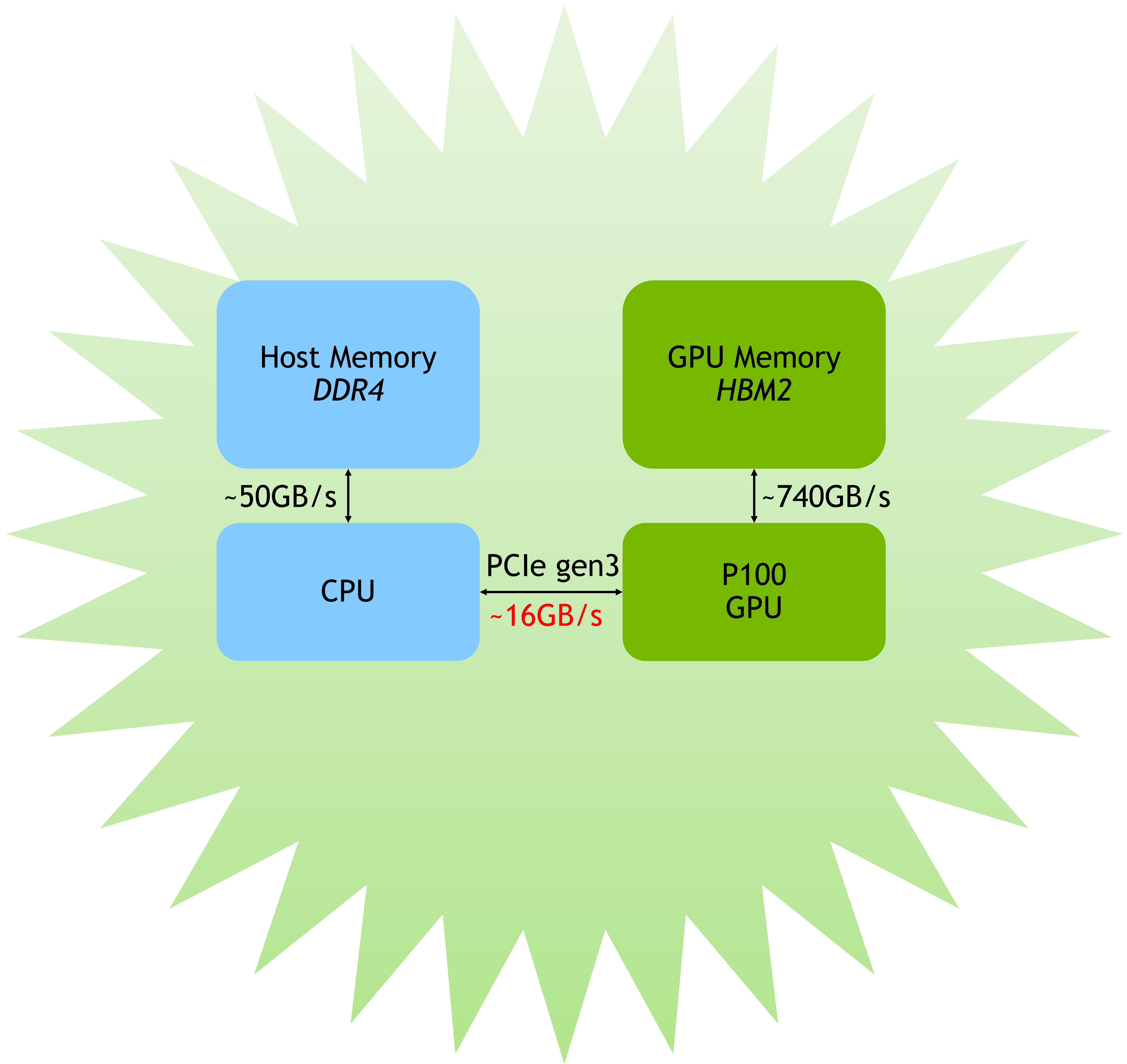
```
@cp.fuse
def fused_squared_diff(x, y):
    return (x - y) * (x - y)


@jit
def Add(a, b):
    return a + b

==================================

#pragma acc parallel
{
    #pragma acc loop
    for(int i= 0; j < N; i++) {
        a[i] = 0;
    }
}
```

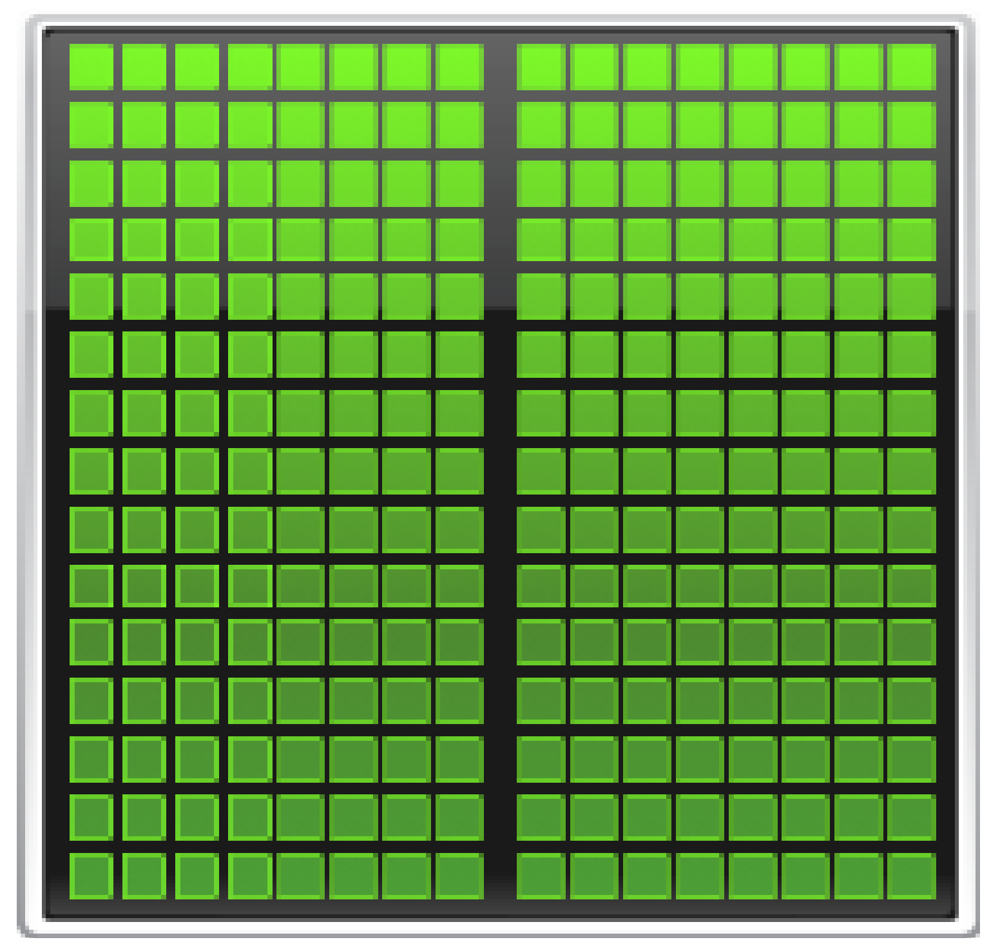
Notebook 1: Introduction to CuPy

Working in Section:
“Kernel Overhead”

DATA MOVEMENT OVERHEAD



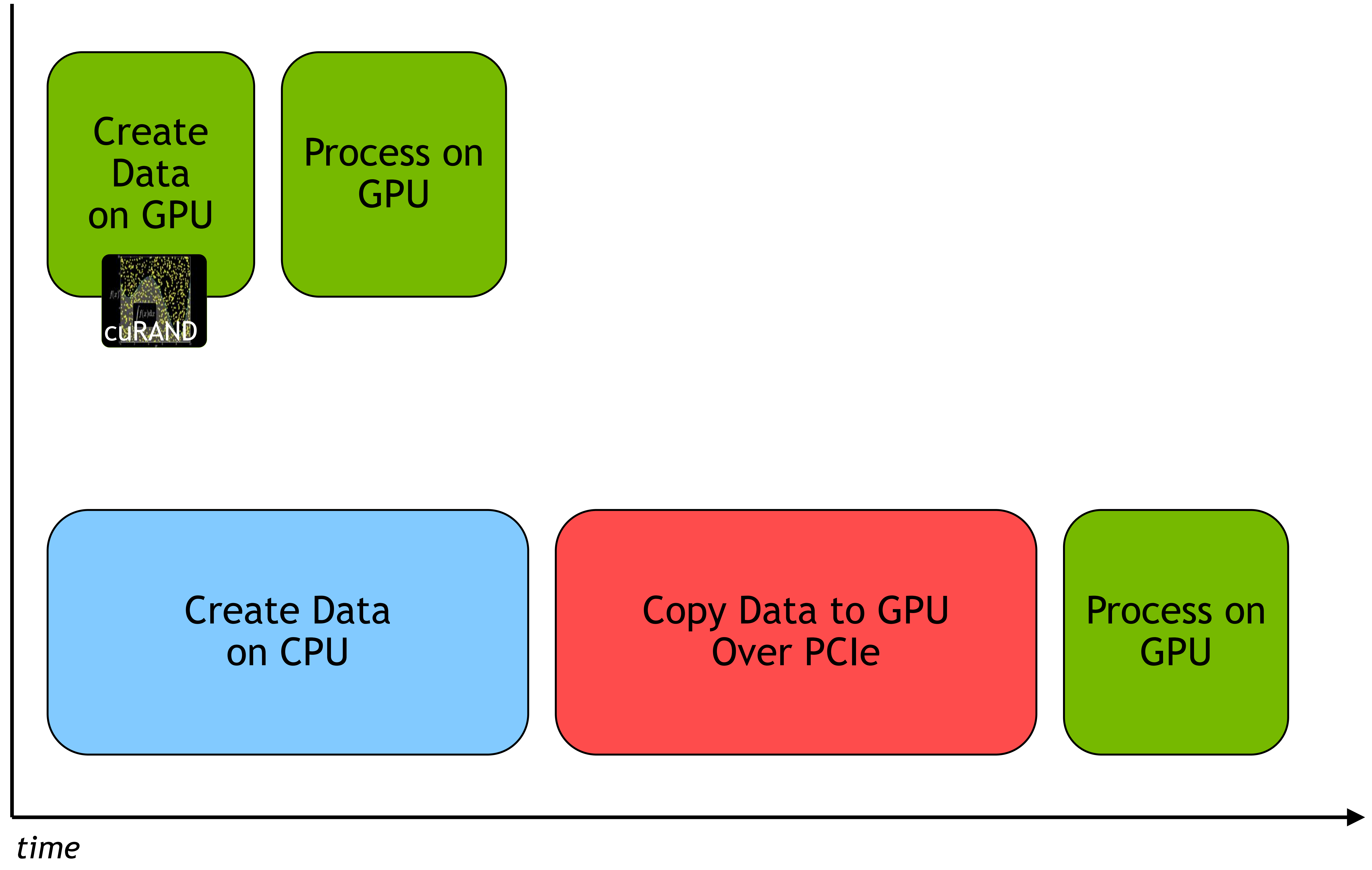




DATA MOVEMENT OVERHEAD

Data Movement Considerations

- Do we have enough work to amortize data transfer cost?
- Can we create our data on the GPU instead?
- CuPy ndarrays stay on GPU until retrieved
  - $Y = AX + XA^T$
  - Can we move more of the workflow to GPU?
- Overlap computation with data movement?



Notebook 1: Introduction to CuPy

Working in Section:
“Data Movement Overhead”

WORKING WITH GPU MEMORY

- How much GPU memory is available?
- Does my problem fit?
- Can I split my problem into stages?
- Split across Multiple GPUs?

WORKING WITH GPU MEMORY

- How much GPU memory is available?
- Does my problem fit?
- Can I split my problem into stages?
- Split across Multiple GPUs?

Explicit Data Movement
GPU 0
GPU 1
GPU Address Space
1. MoveArrayToGPU()
Array
Array
2. DoGPUWork()
Array
3. MoveArrayToCPU()
4. DoCPUWork()
Array
CPU 0
CPU 1
Host Address Space

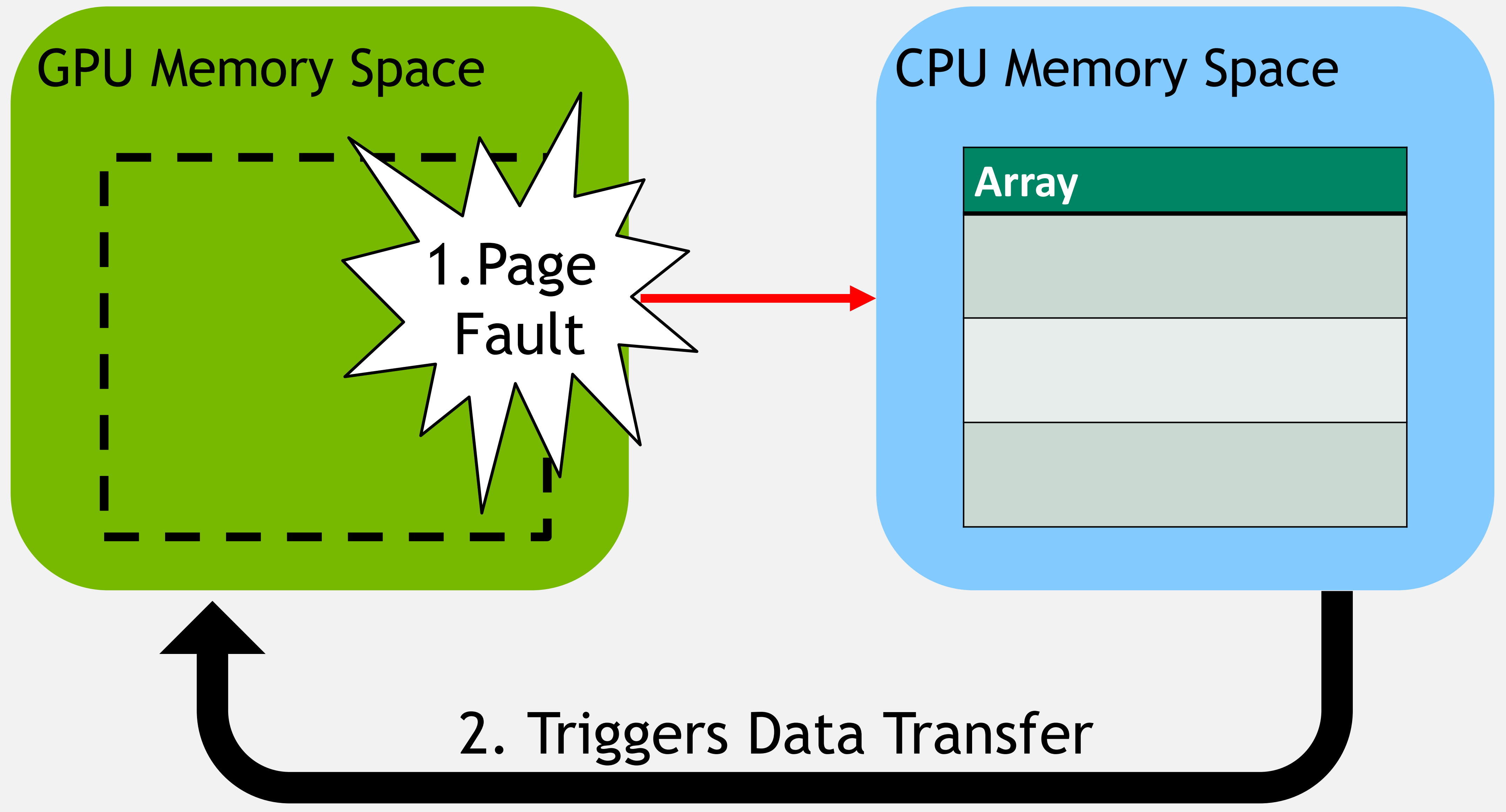
WORKING WITH GPU MEMORY

- How much GPU memory is available?
- Does my problem fit?
- Can I split my problem into stages?
- Split across Multiple GPUs?

Implicit Data Movement
GPU 0
GPU 1
GPU Address Space
1. MoveArrayToGPU()
Array
Array
2. DoGPUWork()
Array
3. MoveArrayToCPU()
4. DoCPUWork()
Array
CPU 0
CPU 1
Host Address Space

UNIFIED MEMORY

GPU 0
GPU 1
...
GPU 7
CPU 0
CPU 1
Unified Memory Address Space



Now we can...

- Oversubscribe GPU Memory
- Allocate data up to size of System Memory
- Program more easily with CPU/GPU Data Coherence
- Prefetch data with CuPy ManagedMemory API

Notebook 1: Introduction to CuPy

Working in Section:
"Managing GPU Memory"

Don't forget to restart the kernel

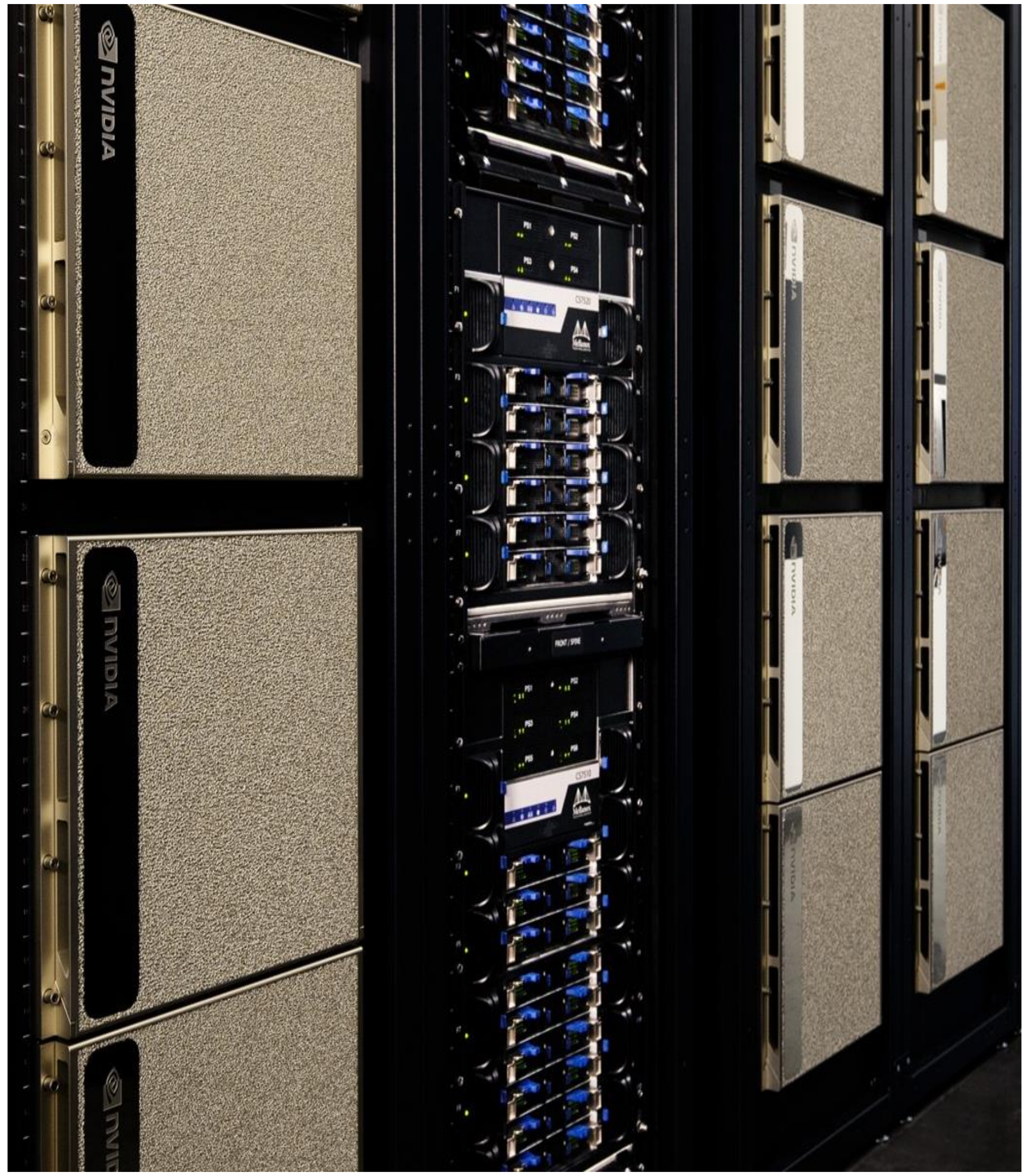
PYTHON ECOSYSTEM GOALS

Have Your Cake and Eat It Too

Productivity

```python
def cg_solve(A, b, conv_iters):
    x = np.zeros_like(b)
    r = b - A.dot(x)
    p = r
    rsold = r.dot(r)
    converged = False
    max_iters = b.shape[0]

    for i in range(max_iters):
        Ap = A.dot(p)
        alpha = rsold / (p.dot(Ap))
        x = x + alpha * p
        r = r - alpha * Ap
        rsnew = r.dot(r)

        if i % conv_iters == 0 and \
            np.sqrt(rsnew) < 1e-10:
            converged = i
            break

        beta = rsnew / rsold
        p = r + beta * p
        rsold = rsnew
```

Performance



CUNUMERIC

Automatic NumPy Acceleration and Scalability

cuNumeric

CuNumeric transparently accelerates and scales existing Numpy workloads

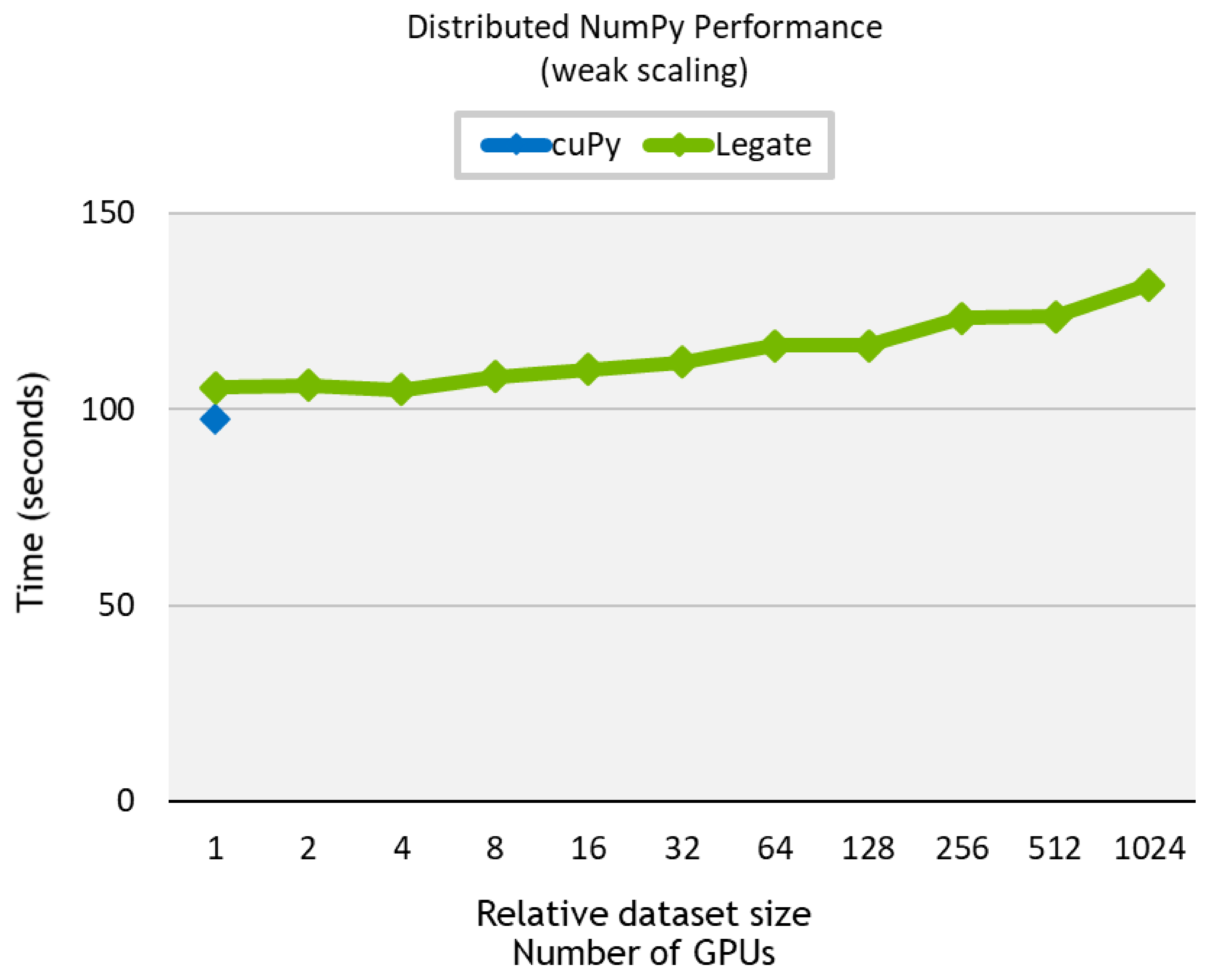
Program from the edge to the supercomputer in Python by changing 1 import line

Pass data between Legate libraries without worrying about distribution or synchronization requirements

Alpha release available at github.com/nv-legate

```
for _ in range(iter):
    un = u.copy()

    vn = v.copy()
    b = build_up_b(rho, dt, dx, dy, u, v)
    p = pressure_poisson_periodic(b, nit, p, dx, dy)

…
```

Extracted from "CFD Python" course at https://github.com/barbagroup/CFDPython
Barba, Lorena A., and Forsyth, Gilbert F. (2018). CFD Python: the 12 steps to Navier-Stokes equations. *Journal of Open Source Education*, **1**(9), 21, https://doi.org/10.21105/jose.00021



GPU-Accelerated Data Science with RAPIDS

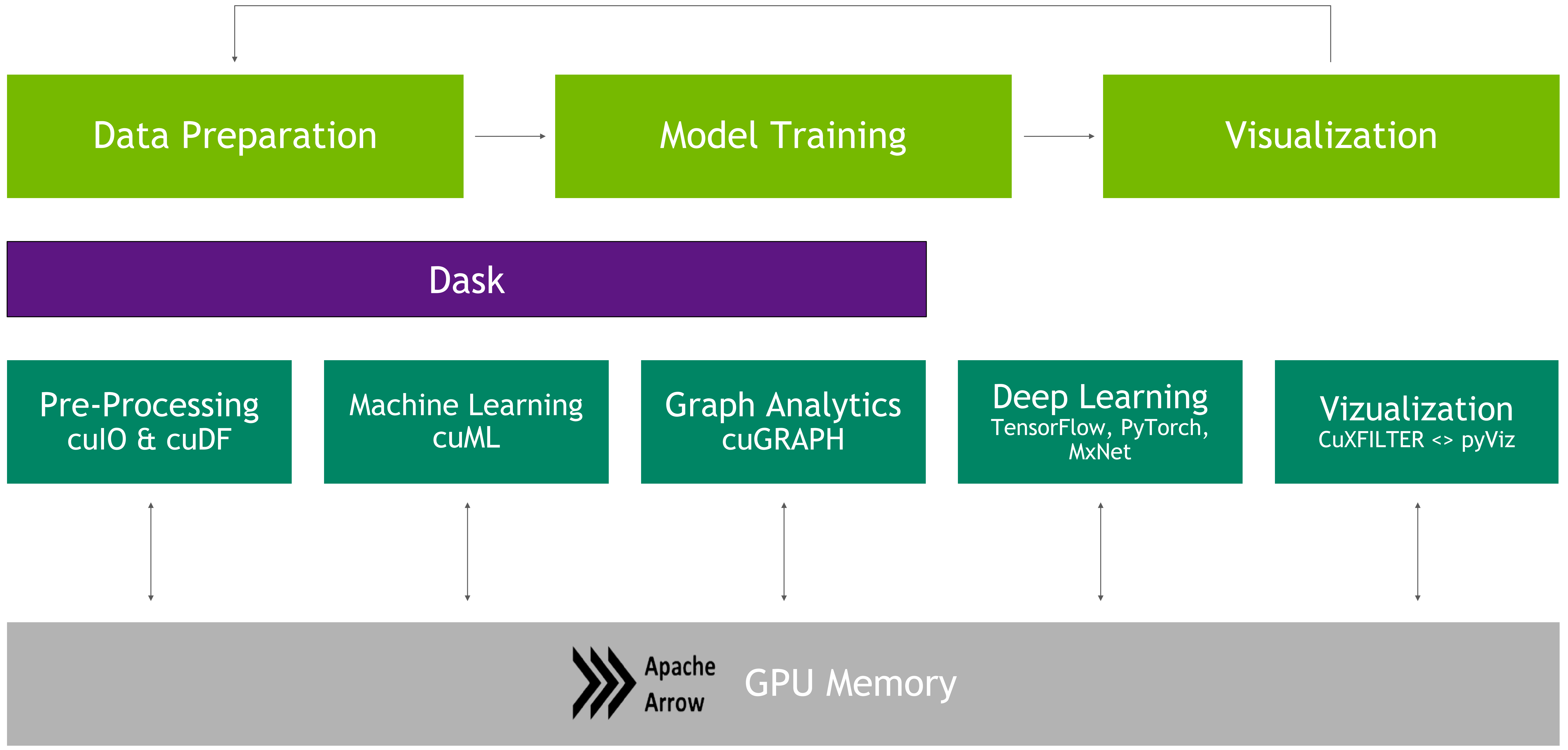
RAPIDS ACCELERATES POPULAR DATA SCIENCE TOOLS

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

DEVELOP IN PYTHON? LISTEN UP!

Data Scientists apply a wide spectrum of techniques to solve hard data problems

HPC

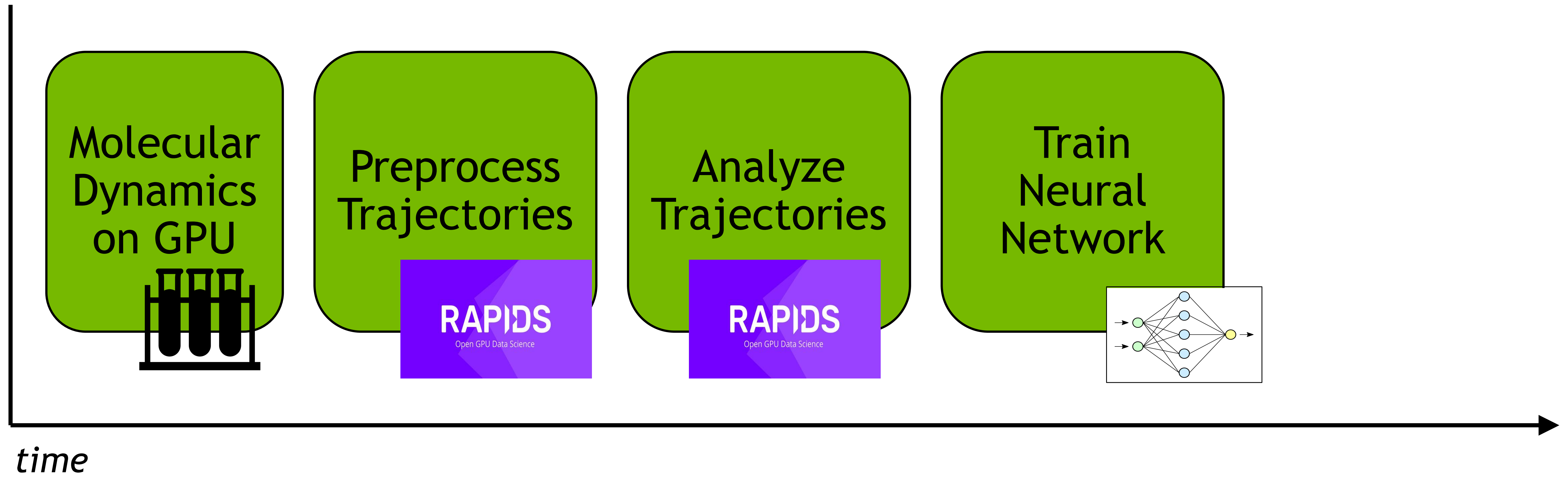
High Performance Computing systems are the backbone of today's cutting-edge research and production systems. Access to GPU-acceleration is one of the most essential tools fueling complex models on large datasets.

HPDA

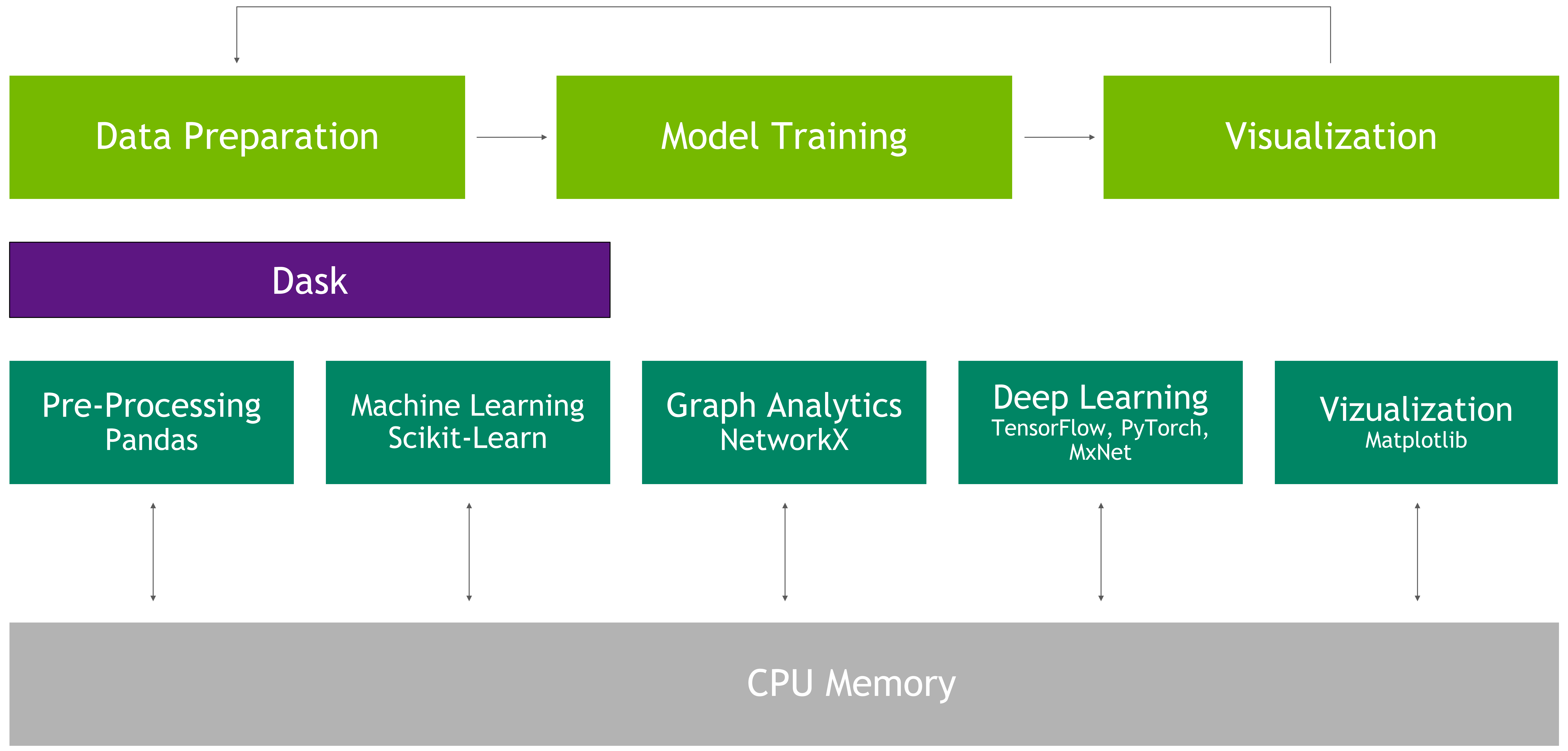
Data scientists leverage popular data analytics tools to perform quantitative investigations. Speed and scale are key to perform comprehensive analysis to deliver the best insights.

AI/ML

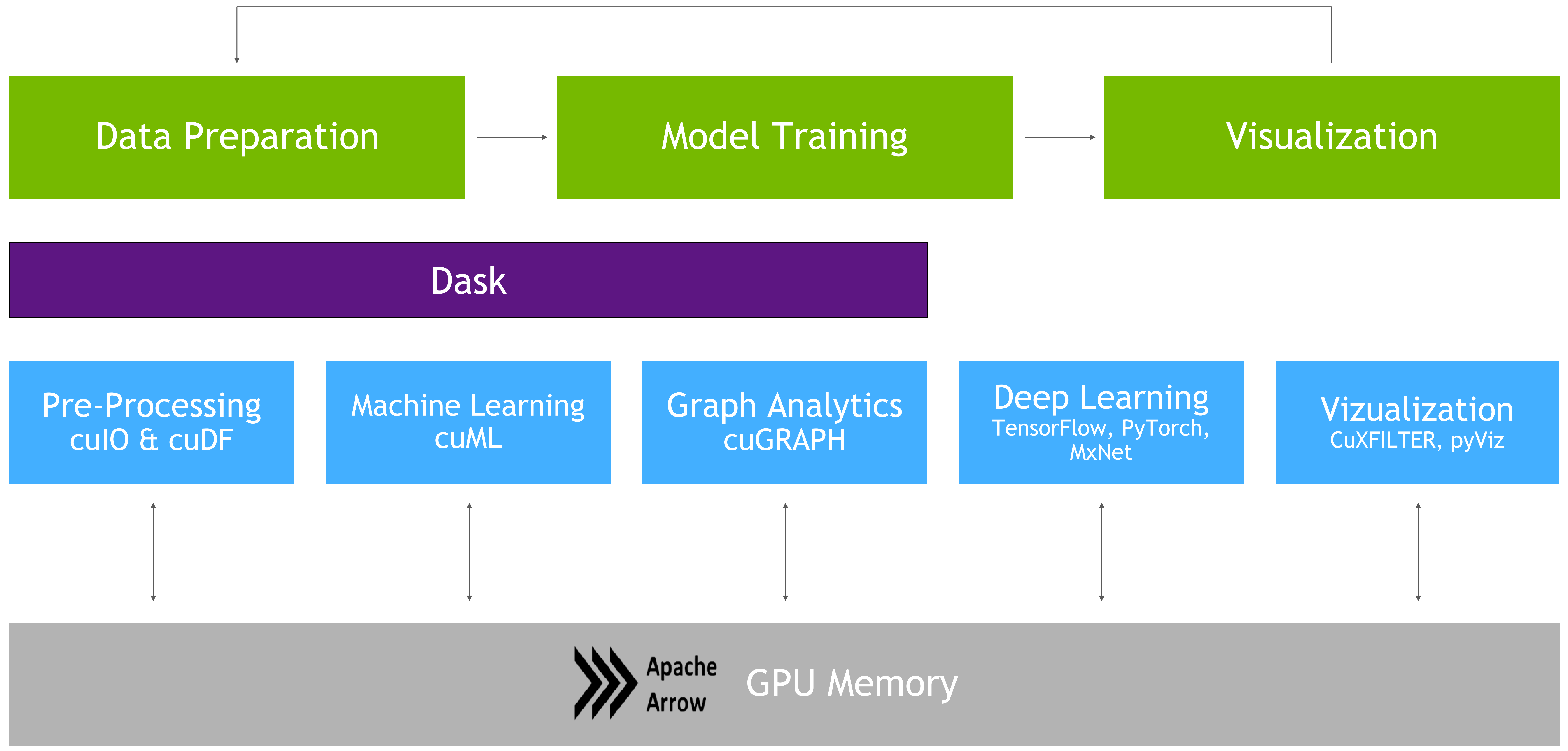
Today's Machine Learning models are increasingly complex, with language models containing billions of nodes. Training and inference require significant computing for to support production tasks.



TRADITIONAL DATA SCIENCE APPLICATIONS



RAPIDS: GPU-ACCELERATED DATA SCIENCE *WITH API ALIGNMENT*



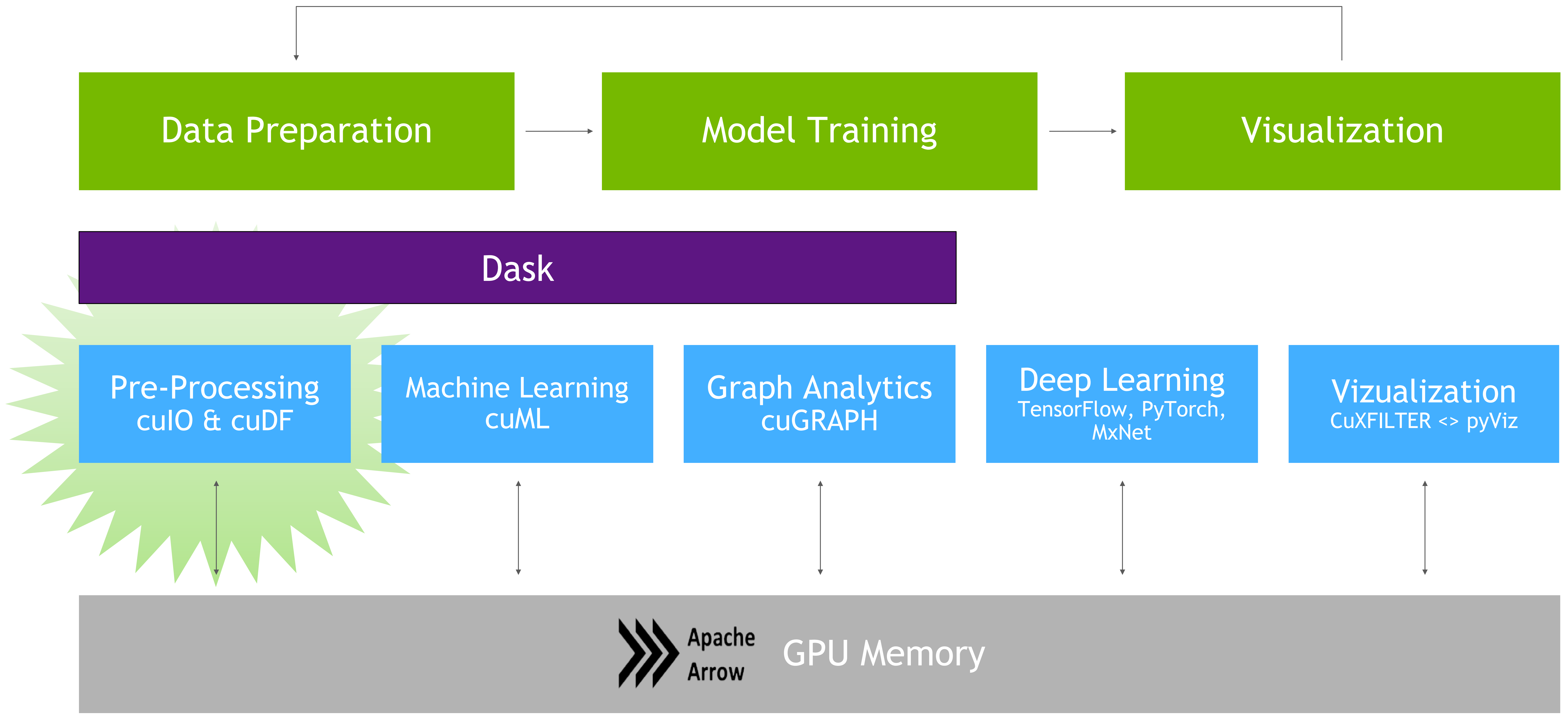
DATA SCIENCE API ALIGNMENT

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

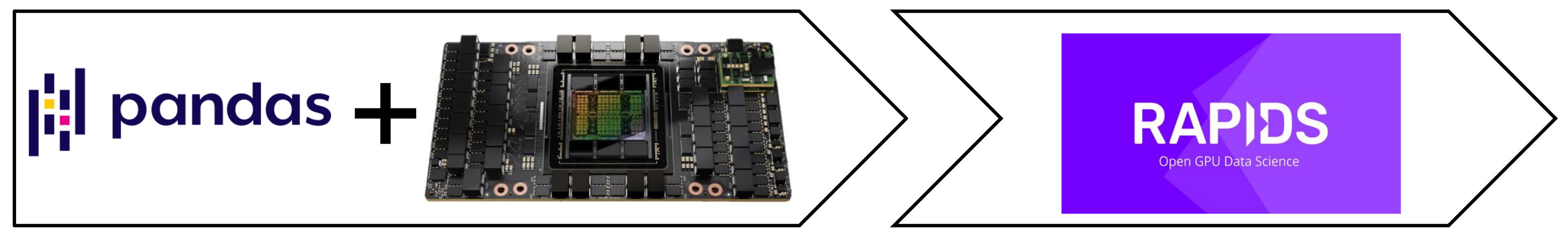
The RAPIDS and GPU-accelerated PyData stack bring GPGPU to data scientists at the Python layer providing familiar APIs without the steep curve of learning new programming language or paradigm

RAPIDS: GPU-ACCELERATED DATA SCIENCE *WITH API ALIGNMENT*



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 INTEROPERABILITY

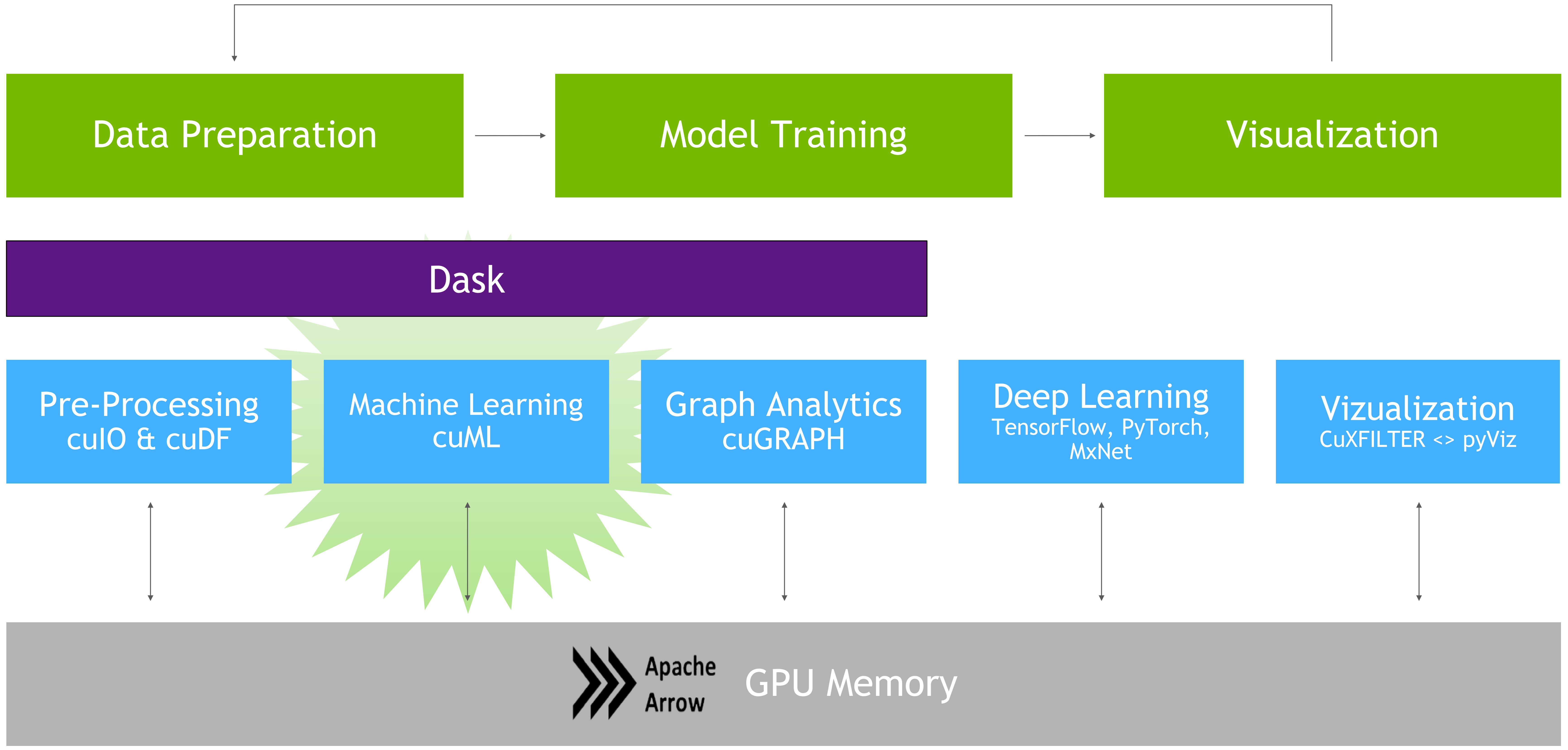
DLPack and __cuda_array_interface__

mxnet
mpi4py
PYTORCH
RAPIDS
Open GPU Data Science
Numba
Chainer
CuPy

Notebook 2: Introduction to RAPIDS

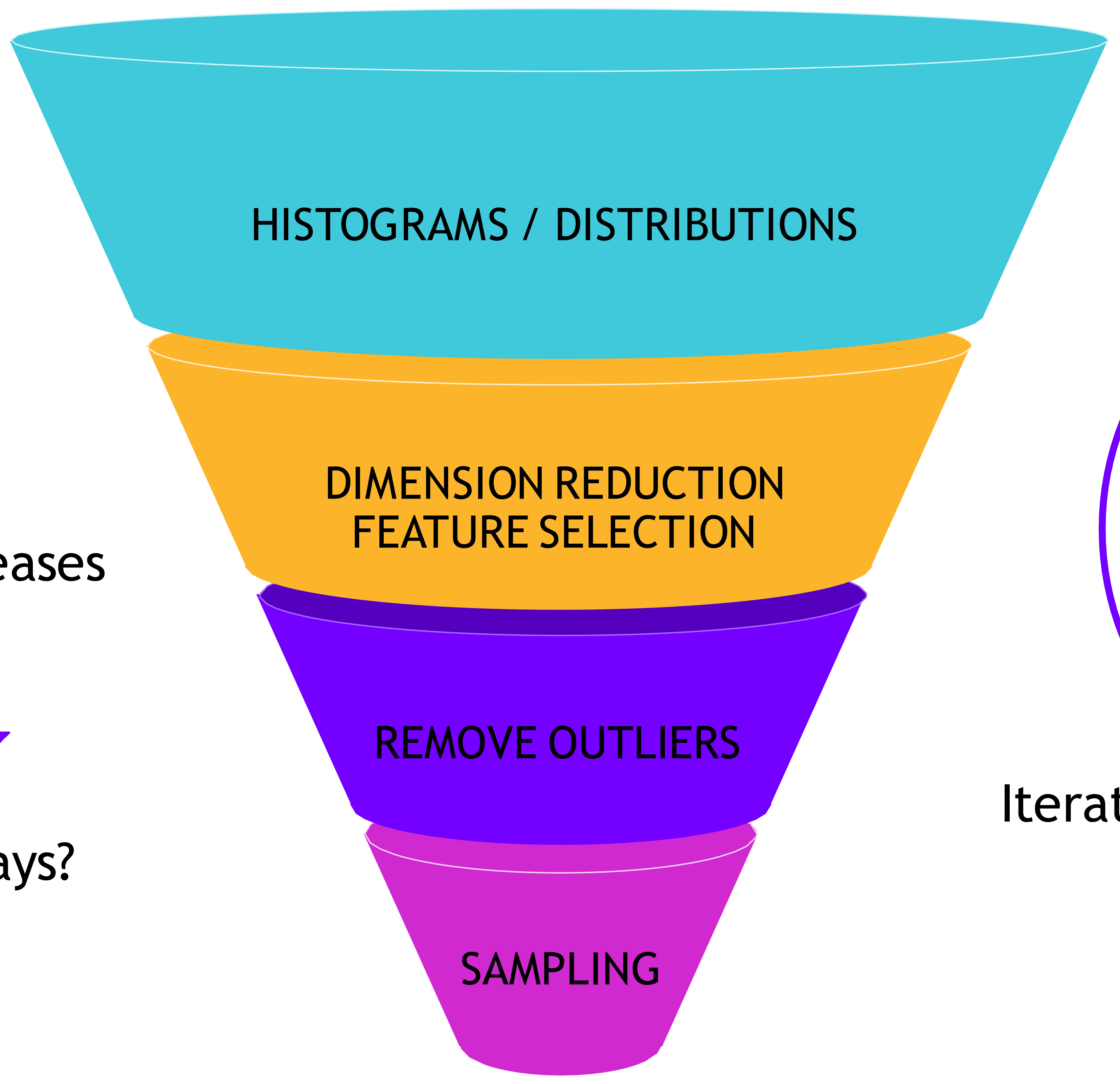
Working in Section:
“GPU-Accelerated Data Manipulation with CuDF”

RAPIDS: GPU-ACCELERATED DATA SCIENCE *WITH API ALIGNMENT*



DATASET SIZES CONTINUE TO GROW

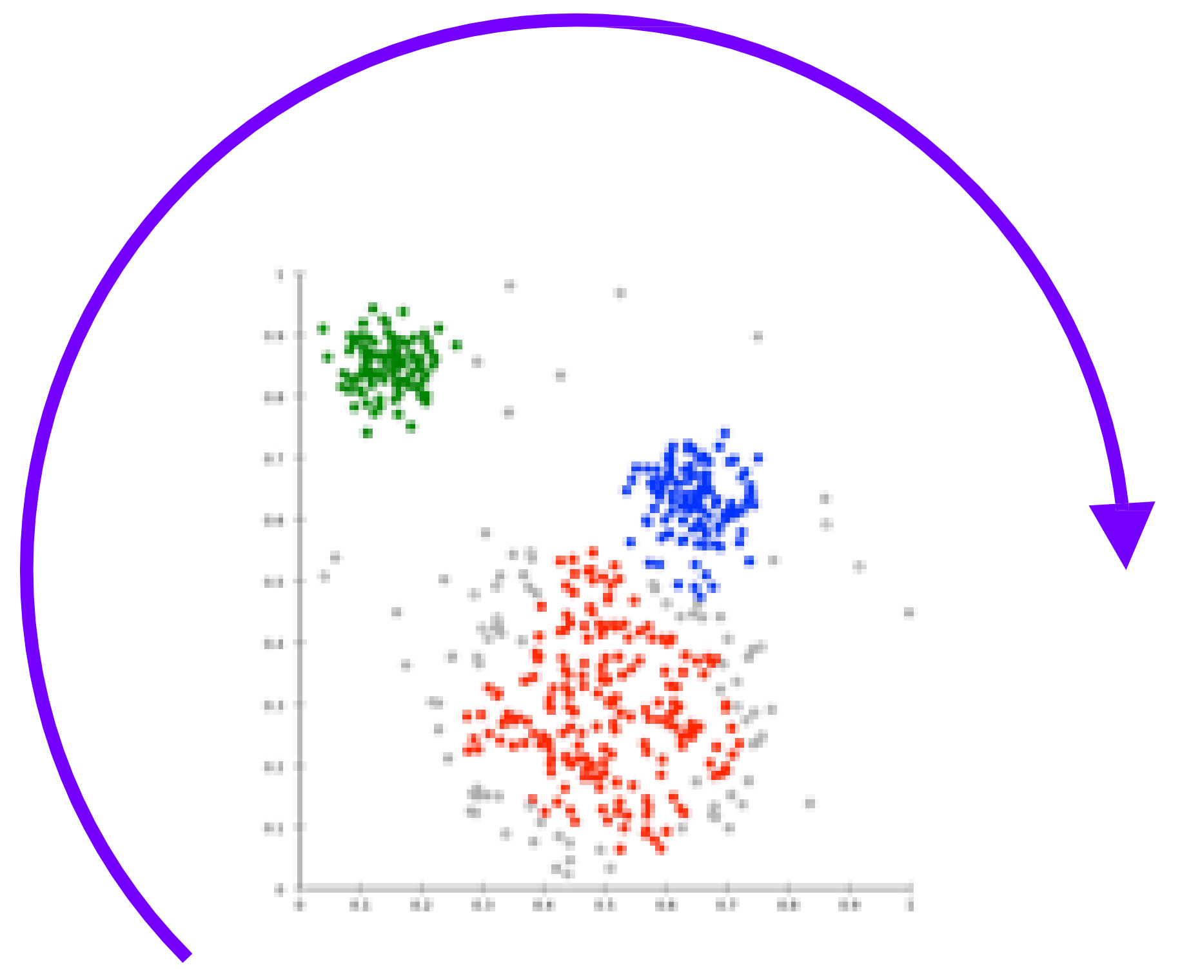
MASSIVE DATASET



Better to start with as much data as possible and explore / preprocess to scale to performance needs.

Time Increases

Hours? Days?



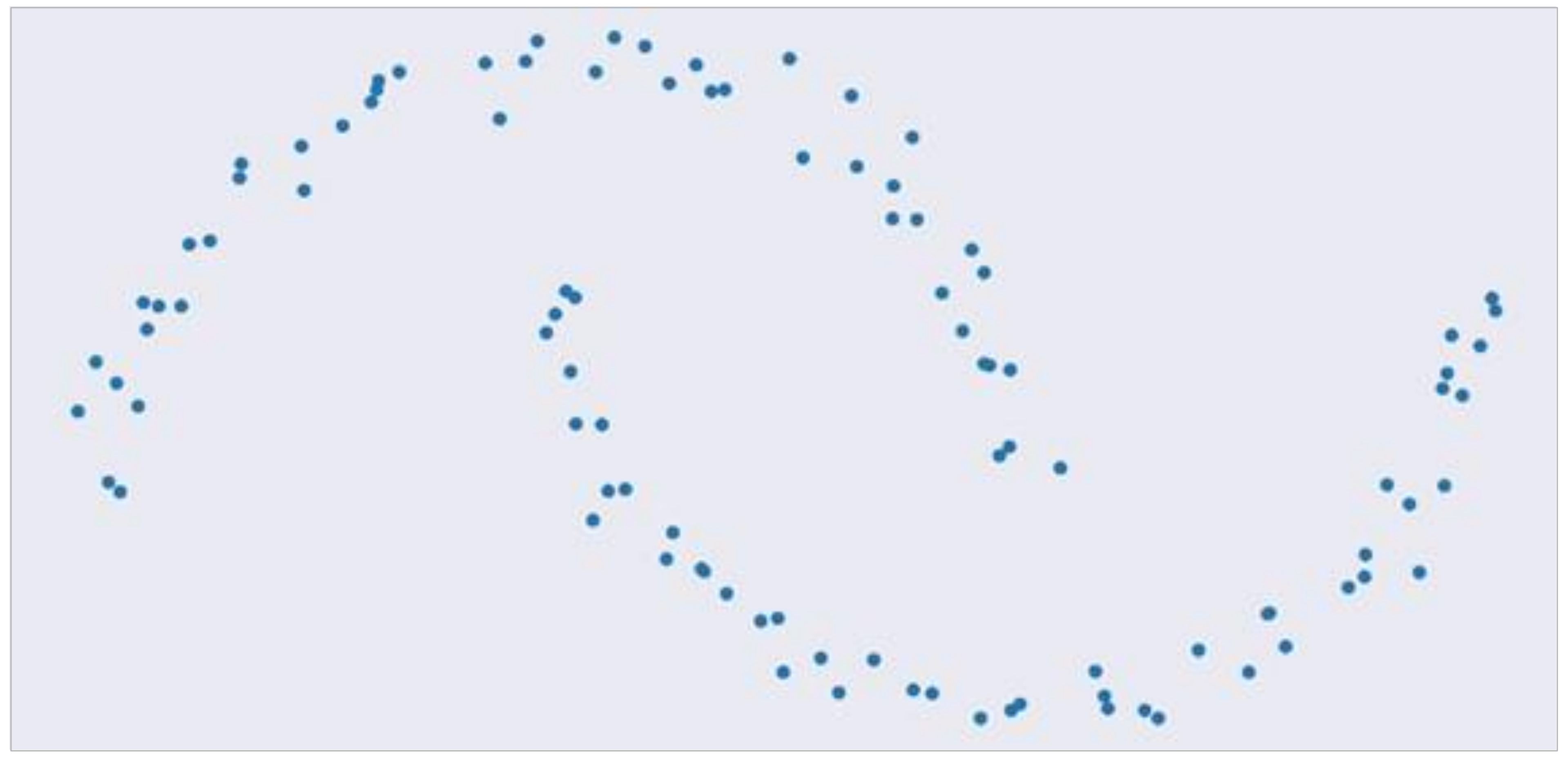
Iterate. Cross Validate & Grid Search.
Iterate Some More.

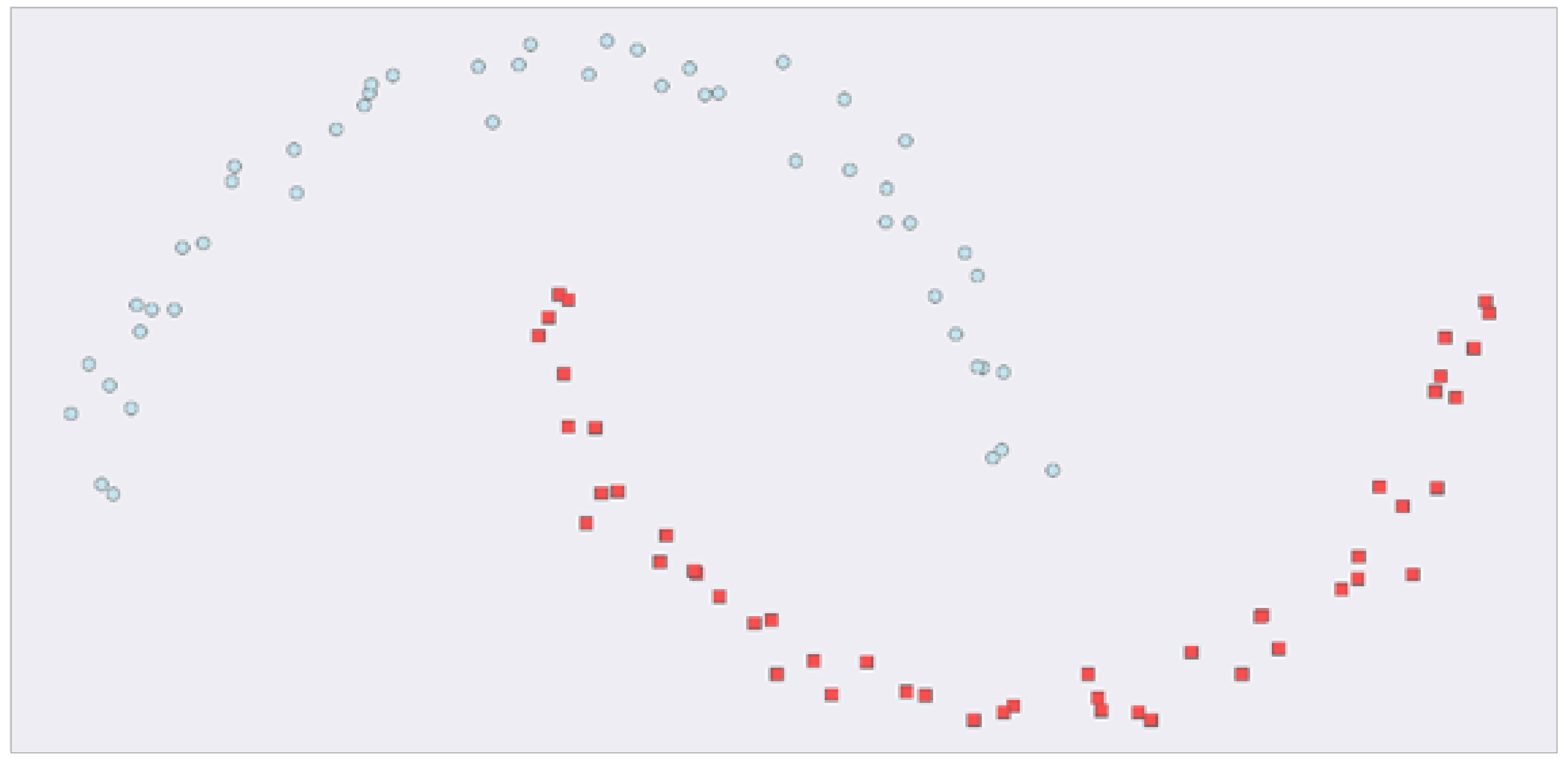
Meet Reasonable Speed vs Accuracy Trade-off

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({'fea%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)
```

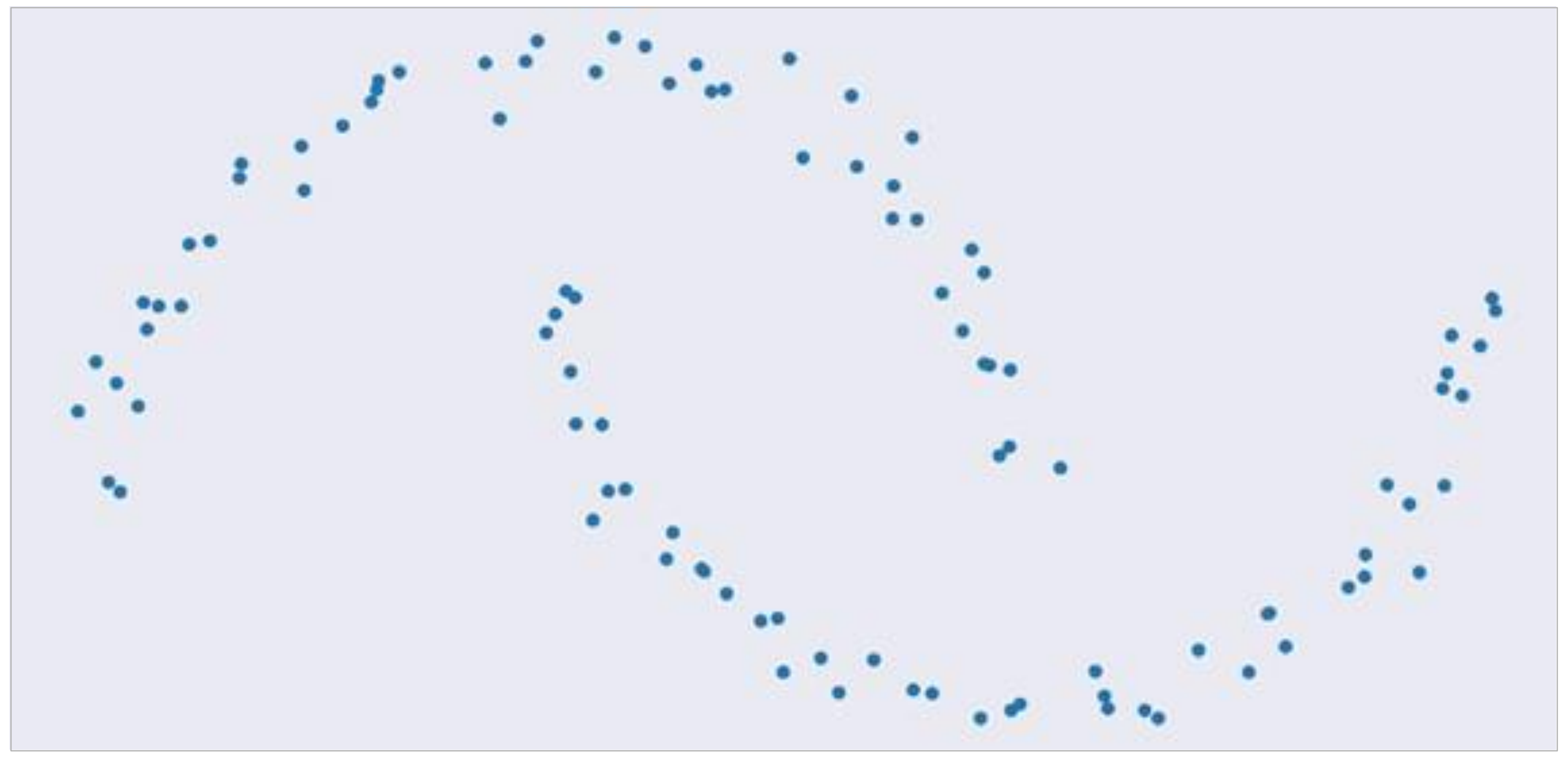


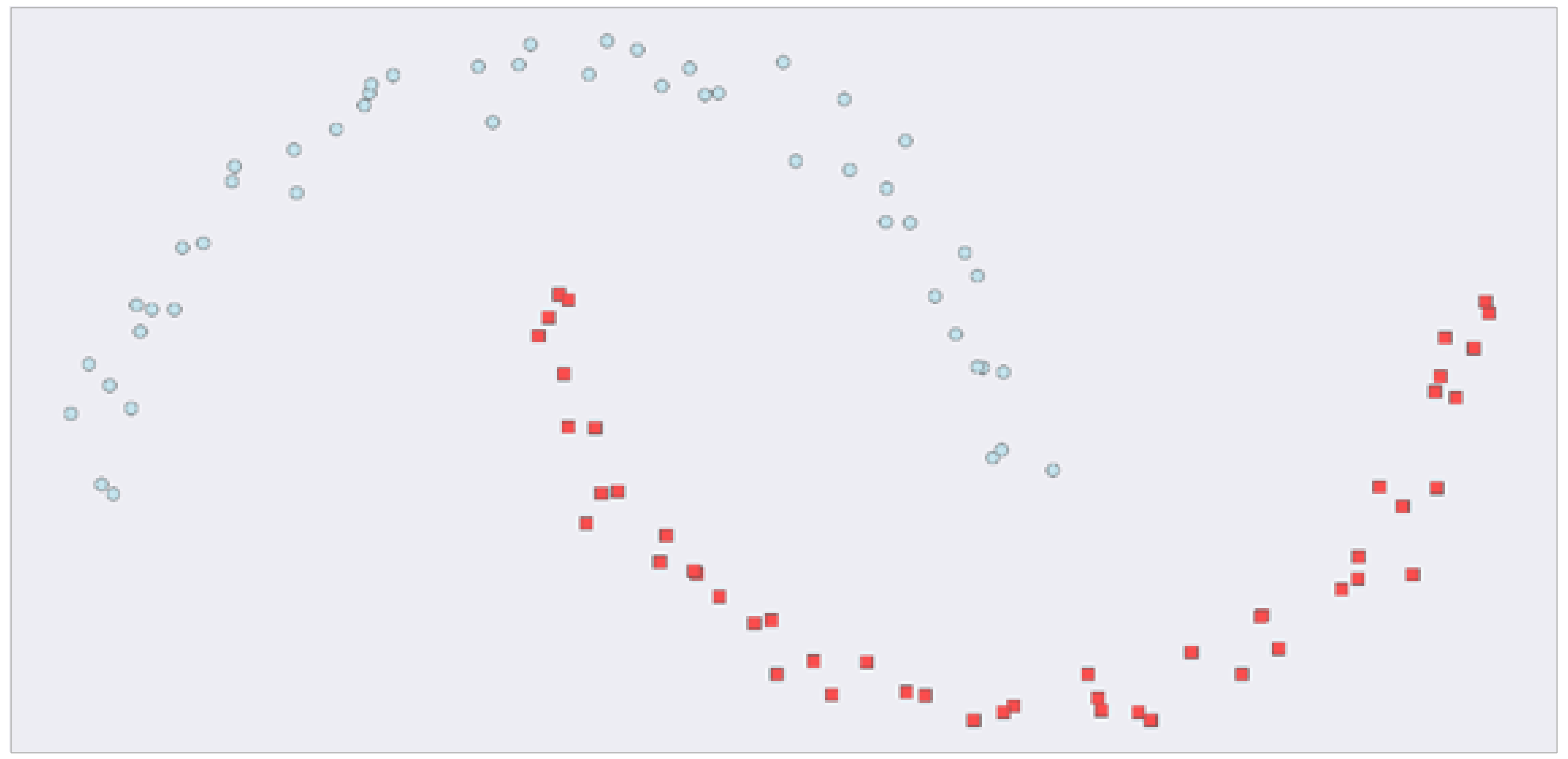


DATASET SIZES CONTINUE TO GROW

```
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'%i: 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)
```

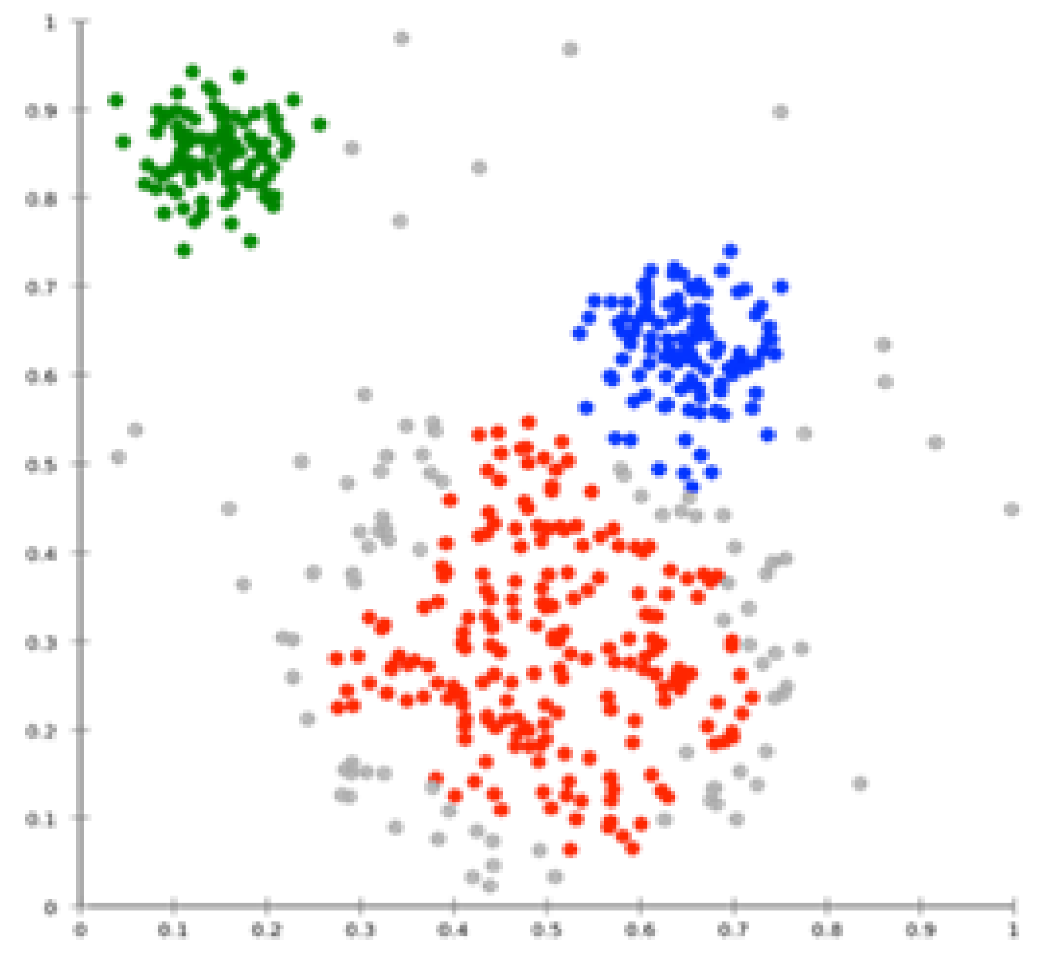




CUML ALGORITHMS

Classification / Regression

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



Inference

Random Forest / GBDT Inference (FIL)

Preprocessing

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

Clustering Decomposition Dimensionality Reduction

K-Means
DBSCAN
Spectral Clustering
Principal Components (including iPCA)
Singular Value Decomposition
UMAP
Spectral Embedding T-SNE

Time Series

Holt-Winters
Seasonal ARIMA / Auto ARIMA

Hyper-parameter Tuning

Cross Validation

More to come!

Notebook 2: Introduction to RAPIDS

Working in Section:
“GPU-Accelerated Machine Learning with CuML”

*****Don’t forget to restart the kernel*****

Custom GPU Kernels with *Numba*

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 UFUNCS

Function decorators that help us create Python functions that take in scalars arguments and can be used as NumPy ufuncs

@vectorize

- Operates on scalars
- Compile a pure Python function into a ufunc
- Operates over NumPy arrays as fast as traditional ufuncs written in C
- Numba generates surrounding loop allowing efficient iteration over the actual inputs

```
@vectorize([float64(float64, float64)])
def f(x, y):
    return x + y
```

@guvectorize

- Operate on higher dimensional arrays and scalars
- Can return arrays of differing dimensions
- Don't return their result value, they fill an array taken as an input

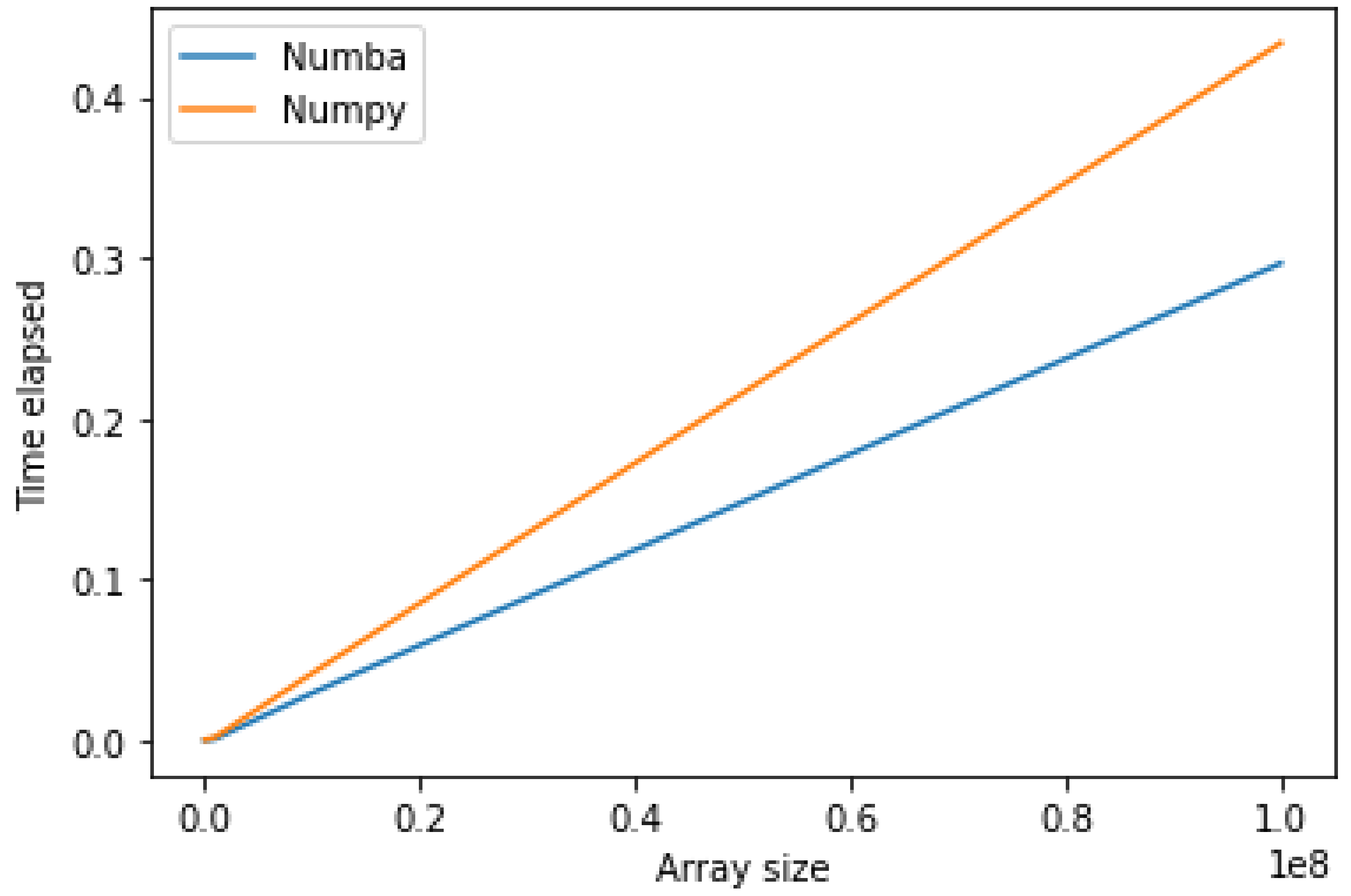
```
@guvectorize([(int64[:], int64, int64[:])],
'(n),()->(n)')
def g(x, y, res):
    for i in range(x.shape[0]):
        res[i] = x[i] + y
```

NUMBA VECTORIZE

NumPy ufuncs operate on data in element-by-element order, and 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)
```



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

Notebook 3: Introduction to Numba

Working in Section:
“Numba Vectorize/Guvectorize"

GPU SUPPORT IN NUMBA

Numba compiles on both the CPU and GPU, below are some additional useful features that can be used on device

Supported Features	Details
Built-in types	Int, float, complex, bool, None, tuple
Built-in functions	Abs(), bool, complex, enumerate(), len(), min(), max(), round(), zip()
Standard library modules	Most of math, cmath, and operator
Numpy Support	ndarray (.shape, .strides, .ndmin, .size), indexing, slicing

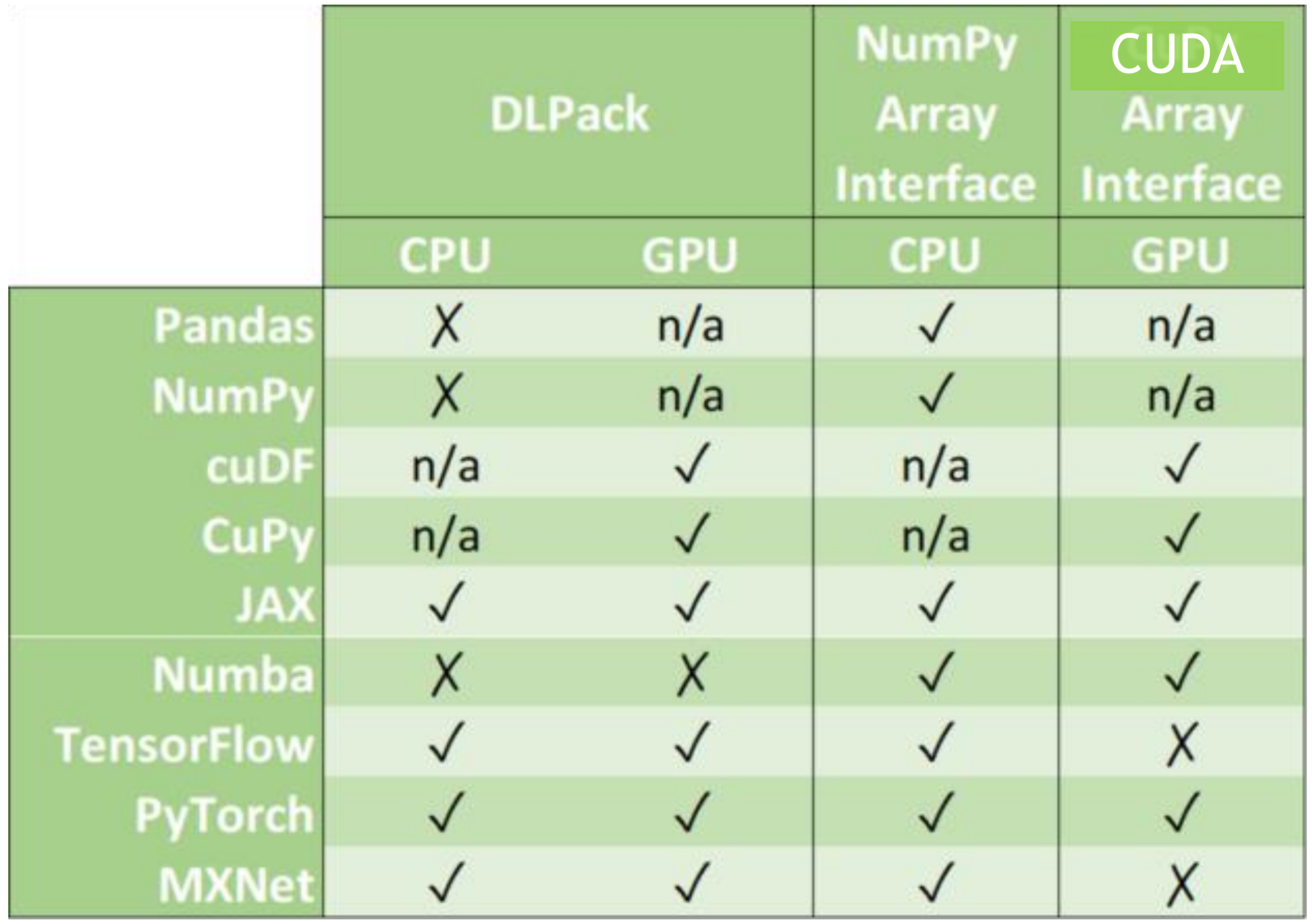
GPU support in Numba: https://numba.readthedocs.io/en/stable/cuda/overview.html

CUDA ARRAY INTERFACE

How can we utilize CuPy, RAPIDS, and Numba together in one program?

This interface offers a standard protocol for different libraries to use and exchange data that is stored on device.

Namely, for Numba we can pass these types of objects directly to our custom kernels.

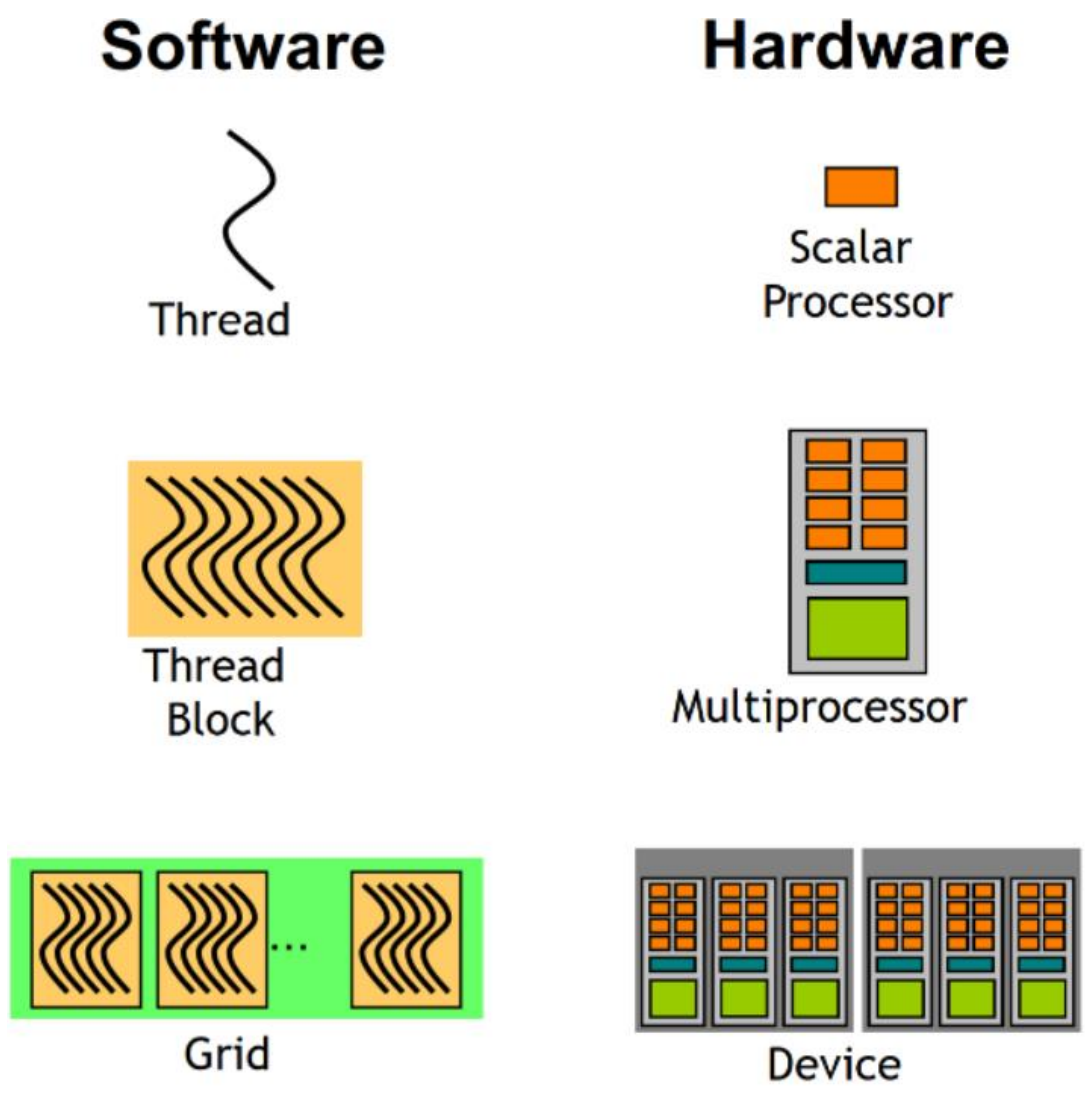


	DLPack		NumPy Array Interface	CUDA Array Interface
	CPU	GPU	CPU	GPU
Pandas	✗	n/a	✓	n/a
NumPy	✗	n/a	✓	n/a
cuDF	n/a	✓	n/a	✓
CuPy	n/a	✓	n/a	✓
JAX	✓	✓	✓	✓
Numba	✗	✗	✓	✓
TensorFlow	✓	✓	✓	✗
PyTorch	✓	✓	✓	✓
MXNet	✓	✓	✓	✗

CUDA Array Interface adopted by:

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

BASIC THREAD HIERARCHY



Threads are executed by scalar processors

Thread blocks are executed on multiprocessors

Thread blocks do not migrate

Several concurrent thread blocks can reside on one multiprocessor - limited by multiprocessor resources (shared memory and register file)

A kernel is launched as a grid of thread blocks

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

Numba

CUSTOM NUMBA KERNEL

Numba and CuPy interoperability can be achieved through the CUDA array interface. CuPy arrays can be used within our custom Numba kernels

```
from numba import (cuda,
                   float32,
                   jit)
import numpy as np
import cupy as cp
import time

@cuda.reduce
def sum_reduce(a, b):
    return a + b

@cuda.jit
def numba_l2_norm(x):
    start = cuda.grid(1)
    stride = cuda.gridsize(1)

    for i in range(start, x.shape[0], stride):
        x[i] = x[i] * x[i]
```

```
x = np.random.rand(2 ** 25)
d_x = cp.array(x)
threads_per_block = 512
blocks_per_grid = 128
```

```
numba_l2_norm[blocks_per_grid, threads_per_block](d_x)
output = cp.sqrt(sum_reduce(d_x))
```

~27 x speedup

```
np.linalg.norm(x, 1)
```

Notebook 3: Introduction to Numba

Working in Section:
"Custom Numba Kernels: Interoperability"

SAXPY METHODS

Below we observe the difference in a basic SAXPY example in CUDA C++ and Numba

```
__global__
void saxpy(int n, float a, float *x, float *y)
{
    for (int i = blockIdx.x * blockDim.x + threadIdx.x;
         i < n;
         i += blockDim.x * gridDim.x)
      {
         y[i] = a * x[i] + y[i];
      }
}
```

```
int N = 1<<20;
cudaMemcpy(d_x, x, N, cudaMemcpyHostToDevice);
cudaMemcpy(d_y, y, N, cudaMemcpyHostToDevice);

saxpy<<<4096,256>>>(N, 2.0, d_x, d_y);
cudaMemcpy(y, d_y, N, cudaMemcpyDeviceToHost);
```

```
@vectorize(['float32(float32, float32, float32)'],
target='cuda')
def saxpy(a, x, y):
    return a * x + y
```

```
N = 2**20

A=cp.random.randn(N).astype(cp.float32)
B=cp.random.randn(N).astype(cp.float32)

C = saxpy(2.0, A, B)
```

Notebook 3: Introduction to Numba

Working in Section:
“Numba custom kernel: SAXPY"

SHARED MEMORY INTRO

Numba can automatically transfer NumPy arrays to the device, but there are times when we want to speed up access to the data

`numba.cuda.shared.array(`*`shape`*`, `*`type`*`)`

- Allocate a shared array with shape and type
- Must be allocated on device
- Returned array-like object acts like a normal device array
- Often we have each thread populate an element in the shared array
- After we use syncthreads() to make sure all the threads have finished their work

Shared memory is readable and writable by all threads in the block. Threads can operator together on a solution.

Essentially, we have a manually-managed data cache

NUMBA MATRIX MULTIPLICATION

Utilize shared memory to accelerate our matrix multiplication

```
@cuda.jit
def fast_matmul(A, B, C):
    sA = cuda.shared.array(shape=(TPB, TPB), dtype=float32)
    sB = cuda.shared.array(shape=(TPB, TPB), dtype=float32)

    x, y = cuda.grid(2)

    tx = cuda.threadIdx.x
    ty = cuda.threadIdx.y
    bpg = cuda.gridDim.x

    tmp = float32(0.)
    for i in range(bpg):
        sA[ty, tx] = 0
        sB[ty, tx] = 0
        if y < A.shape[0] and (tx + i * TPB) < A.shape[1]:
            sA[ty, tx] = A[y, tx + i * TPB]
        if x < B.shape[1] and (ty + i * TPB) < B.shape[0]:
            sB[ty, tx] = B[ty + i * TPB, x]
        cuda.syncthreads()

        for j in range(TPB):
            tmp += sA[ty, j] * sB[j, tx]

        cuda.syncthreads()
    if y < C.shape[0] and x < C.shape[1]:
        C[y, x] = tmp
```

Defining our arrays in shared memory

```
x_d = cp.arange(576).reshape([24,24])
y_d = cp.ones([24,24])
z_d = cp.zeros([24,24])

threadsperblock = (TPB, TPB)
blockspergrid_x = ceil(z_h.shape[0] / threadsperblock[0])
blockspergrid_y = ceil(z_h.shape[1] / threadsperblock[1])
blockspergrid = (blockspergrid_x, blockspergrid_y)

%time fast_matmul[blockspergrid, threadsperblock](x_d, y_d, z_d)
```

Make sure to sync threads

Notebook 3: Introduction to Numba

Working in Section:

“Numba custom kernel: Matrix multiplication"

INTRODUCTION SUMMARY

What tools do we have at our disposal to start our main case study?

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

Case Study:
Accelerating Geospatial Nearest-Neighbor Search

Evaluating Your Options for Accelerated Numerical Computing in Pure Python
Matt Penn [GTC22 S41645]

CALCULATING DISTANCES UNDERPINS MANY NUMERICAL APPLICATIONS

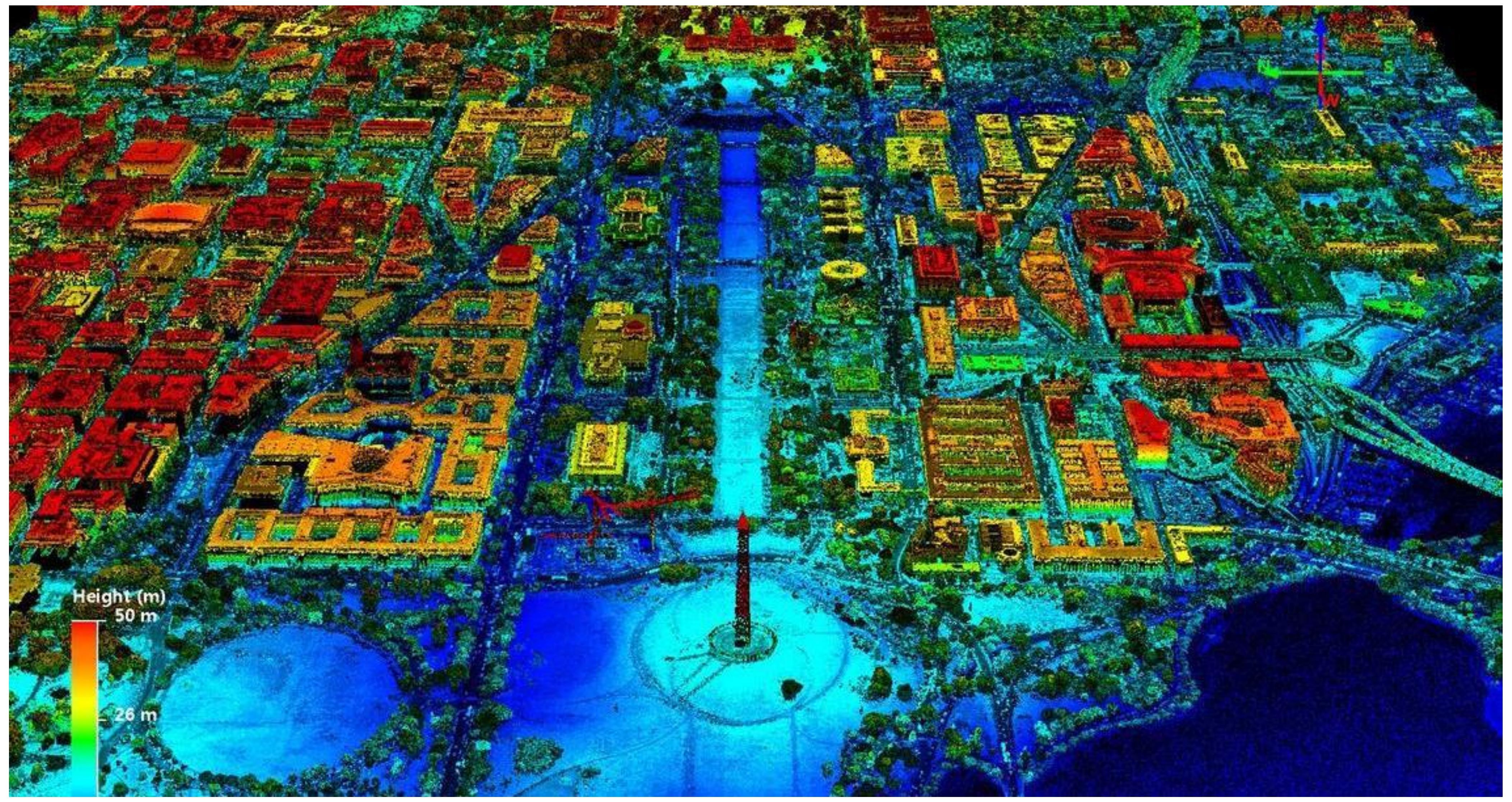
As the number of distances grows, so does the risk of compute bottleneck

Performing distance calculations are the cornerstone of many numerical applications from fundamental research to machine learning.

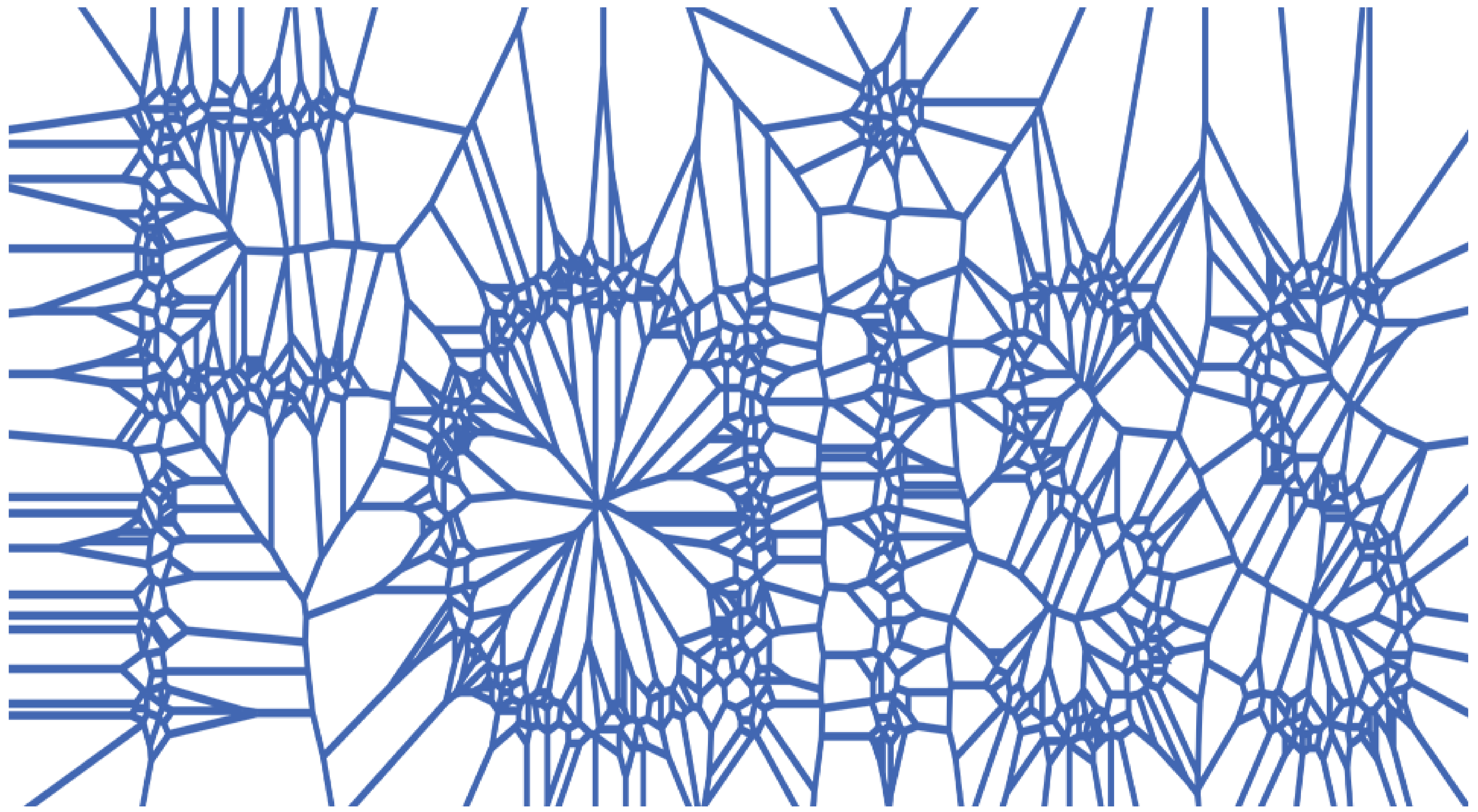
- *Distance matrices for clustering algorithms*
- *Training nearest neighbor models (sometimes relaxing precision for speed)*
- *Calculating exact nearest neighbors for applications where accuracy is necessary*
- *Gridding LiDAR point clouds to generate Digital Elevation Models in remote sensing applications*
- *Performing text similarity search for information retrieval applications*
- *Performing large scale image similarity searches*
- *This list goes on and on!*

In many of these applications, distances need to be calculated a tremendous number of times.

In this session, we will explore accelerated n-dimensional numerical computing through the lens of a proxy brute force exact nearest neighbor problem. Our techniques will be put to the test, calculating up to 274.88M geospatial distance calculations/comparisons.



https://www.usgs.gov/media/images/lidar-point-cloud-washington-dc-0



https://engineering.fb.com/2017/03/29/data-infrastructure/faiss-a-library-for-efficient-similarity-search/

USE CASE – DYNAMIC OBSERVATION TO REFERENCE POINT RESOLUTION

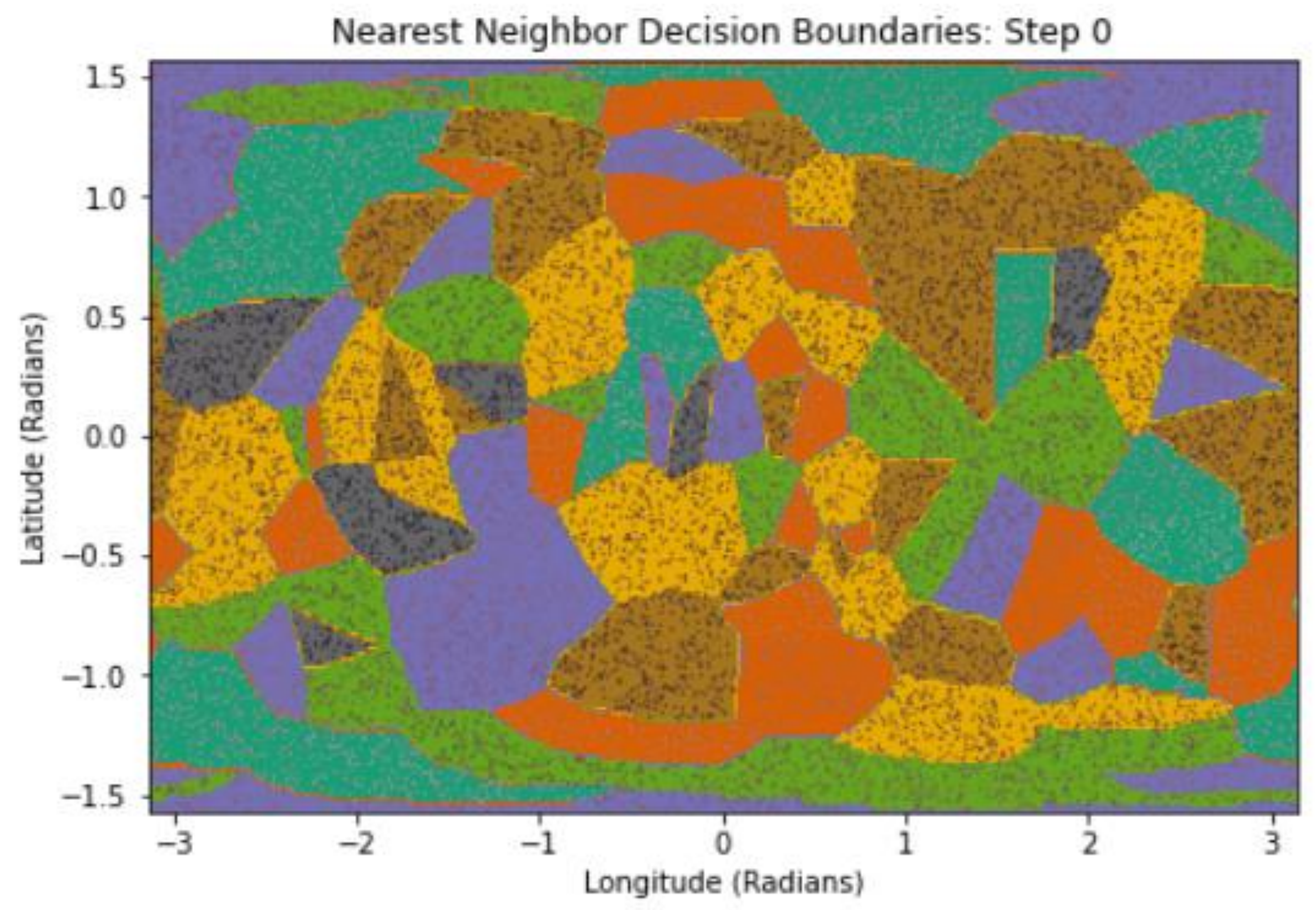
GPU-accelerated exact nearest neighbor calculation to reduce data pipeline bottlenecks

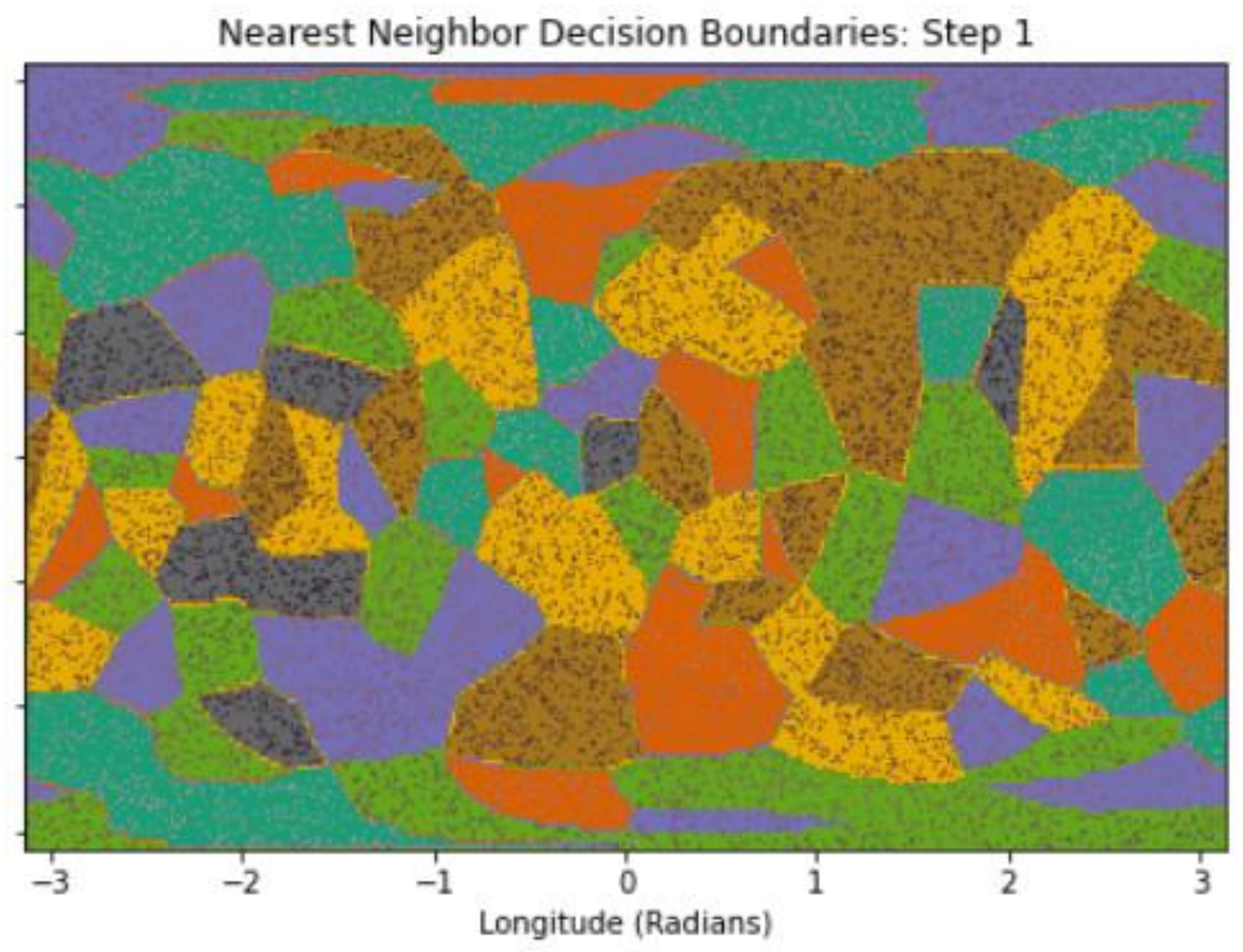
Goal: Let’s say we have an application that is attempting to resolve geospatial observations their closest reference points. In this scenario, observations and reference points are dynamic and variable. This makes indexing optimizations less viable and require complete recomputing at each timestep. This sounds an interesting opportunity for apply a brute force nearest neighbor algorithm.

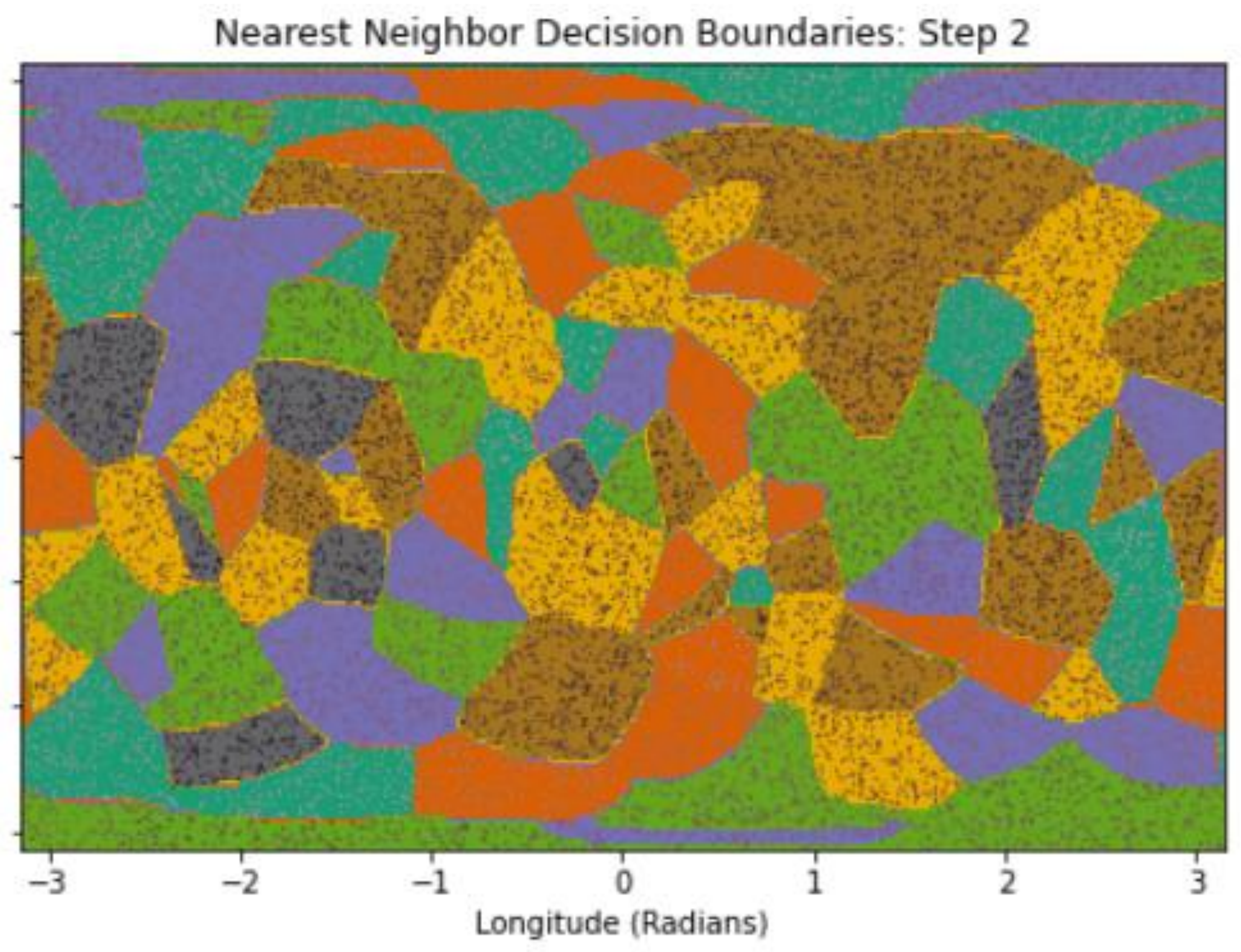
Constraints: Our near real-time application requires an exact solution when performing an MxN nearest neighbor resolution. Our developer team comprised primarily Python developers. Serial or low scale parallel techniques cannot keep up with throughput goals and constrain the size problem that can be solved. Developing in C/C++ would imply slower prototyping cycles, require new skills for the Python developers, and add to code maintenance complexity.

Dataset: We generate a synthetic dataset of geospatial coordinates arbitrarily distributed. These data comprise a set of M reference points and N unresolved observations. By selecting variations of M and N, we can run scenarios on larger and smaller problem sizes and analyze the performance implications.

Observations and reference points (and thereby decision boundaries) change/move between timesteps steps



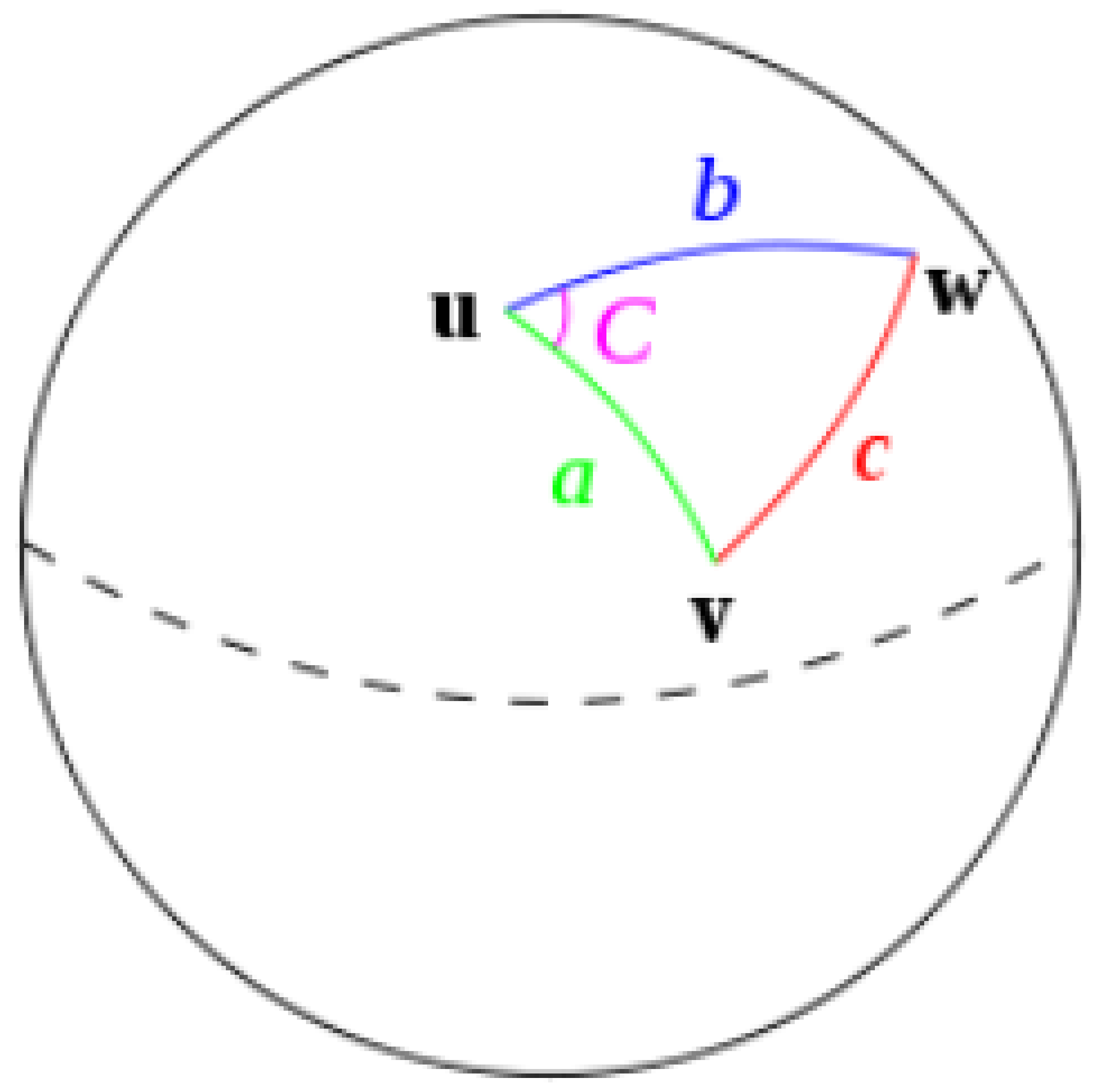




HIGH LEVEL ALGORITHM

Brute Force Geospatial Nearest Neighbor Search

We chose a brute force technique for its simplicity to explain and additional arithmetic intensity to magnify the tradeoffs between techniques



Source

Step 1
x_0
x_1
x_2
x_{m-1}
y_0
y_1
y_2
Y_{n-1}
Step 2
$X_0 - y_0$
$X_0 - y_1$
$X_0 - y_2$
$X_0 - y_{n-1}$
$X_1 - y_0$
$X_1 - y_1$
$X_1 - y_2$
$X_1 - y_{n-1}$
$X_2 - y_0$
$X_2 - y_1$
$X_2 - y_2$
$X_2 - y_{n-1}$
$X_{m-1} - y_0$
$X_{m-1} - y_1$
$X_{m-1} - y_2$
$X_{m-1} - y_{n-1}$
Step 3
idx_0
idx_1
idx_2
idx_{m-1}
$dist_0$
$dist_1$
$dist_2$
$dist_{n-1}$

Input observations X and Y of arbitrary size m and n, respectively

Calculate distances between each point in X and Y

Return the argmin along the Y-axis and its corresponding minimum distance

Single Threaded CPU & Single GPU Techniques

WHAT WE WILL EVALUATE

Single-CPU and Single-GPU Methodologies

Before thinking about scaling our code, developers typically prove out methods on single processors. In this section, we will discuss several popular approaches to solving our proxy problem and inspect some performance metrics. In the end, we will appreciate the benefit of parallel execution on GPUs. During this process, we will not write a single line of C/C++!

Single CPU
Conventional For Loop
NumPy Broadcasting
Scikit-Learn Brute Force KNN
Numba CPU Kernel
Single GPU
CuPy Broadcasting
cuML Brute Force KNN
Numba GPU Kernel

Problem Size: 540M
2^{16} (observations) * 2^{13} (reference points)

As we progress through the examples, each technique will have memory and speed implications

CPU - CLASSIC DOUBLE FOR LOOP

This example does not serve much purpose besides a lower limit

```
import numpy as np
import math

def loop_haversine(lat1, lon1, lat2, lon2):

    first_sin = math.sin((lat2 - lat1) / 2.)
    second_sin = math.sin((lon2 - lon1) / 2.)

    a = first_sin * first_sin + \
        math.cos(lat1) * \
        math.cos(lat2) * \
        second_sin * second_sin

    a = math.sqrt(a)

    if a > 1.:
        a = 1.
    elif a < 0:
        a = 0.

    a = math.asin(a)

    return 2.0 * a
```

```
def loop_solve(a, b):

    out_idx = np.empty(
        (a.shape[0]), dtype=np.uint32)

    out_dist = np.empty(
        (a.shape[0]), dtype=np.float32)

    for obs_idx in range(a.shape[0]):

        glob_min_dist = 1e11
        glob_min_idx = 0

        for ref_idx in range(b.shape[0]):

            temp_dist = loop_haversine(
                a[obs_idx, 0],
                a[obs_idx, 1],
                b[ref_idx, 0],
                b[ref_idx, 1])

            if temp_dist < glob_min_dist:
                glob_min_dist = temp_dist
                glob_min_idx = ref_idx

        out_dist[obs_idx] = glob_min_dist
        out_idx[obs_idx] = glob_min_idx

    return out_idx, out_dist
```

42 min 34 s

The code on the left applies a classic double for loop, not leveraging any specialized libraries or techniques -- those not familiar with numerical computing libraries might choose something like this first:

- Extremely straightforward implementation
- Painfully slow ... (almost 45 mins to complete)
- Probably what many people think when calling python "slow"

Good news, we have (many) other options!

SINGLE GPU VS SINGLE CPU - NUMPY/CUPY BROADCAST

NumPy and CuPy share nearly identical syntax but a huge speed disparity

NumPy

```
import numpy as np

def numpy_haversine(lat1, lon1, lat2, lon2):

    return 2.0 * np.arcsin(
        np.sqrt(np.sin((lat2 - lat1) / 2.0)**2 + \
                np.cos(lat1) * \
                np.cos(lat2) * \
                np.sin((lon2 - lon1) / 2.0)**2)
    )

def numpy_solve(a, b):

    a_broad = a[:,np.newaxis]

    temp = numpy_haversine(
        a_broad[:,:,0],
        a_broad[:,:,1],
        b[:,0],
        b[:,1]
    )

    np.abs(temp, out=temp)
    out_idx = temp.argmin(axis=1)
    out_dist = temp[np.arange(a.shape[0]), out_idx]

    return out_idx, out_dist
```

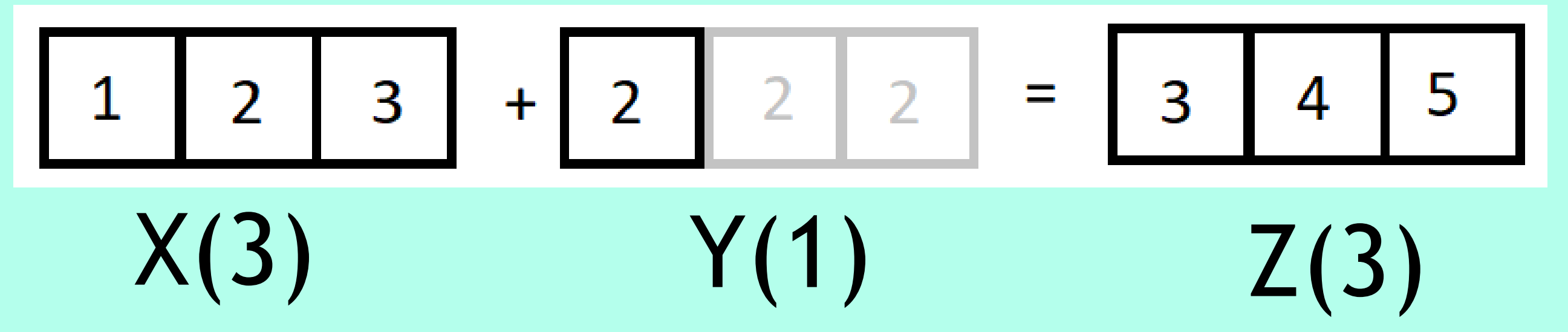
12.2 s

CuPy

```
import cupy as cp

def cupy_haversine(lat1, lon1, lat2, lon2):

    return 2.0 * cp.arcsin(
        cp.sqrt(cp.sin((lat2 - lat1) / 2.0)**2 + \
                cp.cos(lat1) * \
                cp.cos(lat2) * \
                cp.sin((lon2 - lon1) / 2.0)**2)
    )

def cupy_solve(a, b):

    a_broad = a[:,cp.newaxis]

    temp = cupy_haversine(
        a_broad[:,:,0],
        a_broad[:,:,1],
        b[:,0],
        b[:,1]
    )

    cp.abs(temp, out=temp, dtype=np.float32)
    out_idx = temp.argmin(axis=1)
    out_dist = temp[cp.arange(a.shape[0]), out_idx]

    return out_idx, out_dist
```

277 ms

Broadcasting



+Speed

-Memory

Code blocks are identical aside from import statements and variable names

Both cases experience a significant boost in performance when compared to the classic for loop example!

Boosts can come at the cost of memory

In this example, our CuPy option performs ~44x faster than the NumPy equivalent!

SINGLE-CPU VS SINGLE-GPU: SKLEARN/CUML BRUTE KNN

Sklearn and cuML share nearly identical syntax but a huge speed disparity

scikit-learn

```
from sklearn.neighbors import NearestNeighbors

def sklearn_knn_solve(a, b):

    knn = NearestNeighbors(
        algorithm="brute",
        metric="haversine")

    knn.fit(b)

    out_dist_sklrn, out_idx_sklrn = \
    knn.kneighbors(
        a,
        n_neighbors=1,
        return_distance=True)

    return (out_idx_sklrn.reshape(a.shape[0]),
            out_dist_sklrn.reshape(a.shape[0]))
```

46.9 s

RAPIDS cuML

```
from cuml.neighbors import NearestNeighbors

def cuml_knn_solve(a, b):

    cuknn = NearestNeighbors(
        algorithm="brute",
        metric="haversine")

    cuknn.fit(b)

    out_dist_cuml, out_idx_cuml = \
    cuknn.kneighbors(
        a,
        n_neighbors=1,
        return_distance=True)

    return (out_idx_cuml.reshape(a.shape[0]),
            out_dist_cuml.reshape(a.shape[0]))
```

306 ms

Less code & speedup for each option

Code blocks are identical aside from import statements and variable names

Another welcomed boost in speed!

In this example, our cuML option performs ~153x faster than the scikit-learn equivalent!

Working in Notebook 4:
“Evaluating Your Options for Numerical Computing in Pure Python with CuPy and RAPIDS”

SINGLE CPU - NUMBA KERNEL

JIT compile a nearest neighbor CPU kernel to boost speed of our double for loop

```
import numpy as np
from numba import jit

@jit(nopython=True)
def numba_cpu_solve(a, b):

    out_idx = np.empty(
        (a.shape[0]), dtype=np.uint32)

    out_dist = np.empty(
        (a.shape[0]), dtype=np.float32)

    for obs_idx in range(a.shape[0]):

        glob_min_dist = 1e11
        glob_min_idx = 0

        for ref_idx in range(b.shape[0]):

            temp_dist = numba_cpu_haversine(
                a[obs_idx,0],
                a[obs_idx, 1],
                b[ref_idx, 0],
                b[ref_idx, 1])

            if temp_dist < glob_min_dist:
                glob_min_dist = temp_dist
                glob_min_idx = ref_idx

        out_dist[obs_idx] = glob_min_dist
        out_idx[obs_idx] = glob_min_idx

    return out_idx, out_dist
```

```
@jit(nopython=True, fastmath=True)
def numba_cpu_haversine(lat1, lon1, lat2, lon2):

    first_sin = math.sin((lat2 - lat1) / 2.0)
    second_sin = math.sin((lon2 - lon1) / 2)

    return 2.0 * math.asin(
        math.sqrt(first_sin * first_sin + \
                math.cos(lat1) * \
                math.cos(lat2) * \
                second_sin * second_sin)
    )
```

Numba

35.4 s

In this example, we leverage the Numba JIT compiler generate a (much) faster double for loop kernel.

With Numba, we pay a one-time JIT compilation penalty but after that, JIT kernels can be extremely fast – approaching C/C++ speeds.

- Our Numba kernel is very Pythonic
- Completes in 35.4 s, 72x faster than our double naïve double for loop

Let's not get complacent, we still have another single GPU trick up our sleeve!

SINGLE GPU - NUMBA CUDA KERNEL

Fastest option using a hand-tuned CUDA kernel that implements warp level optimizations

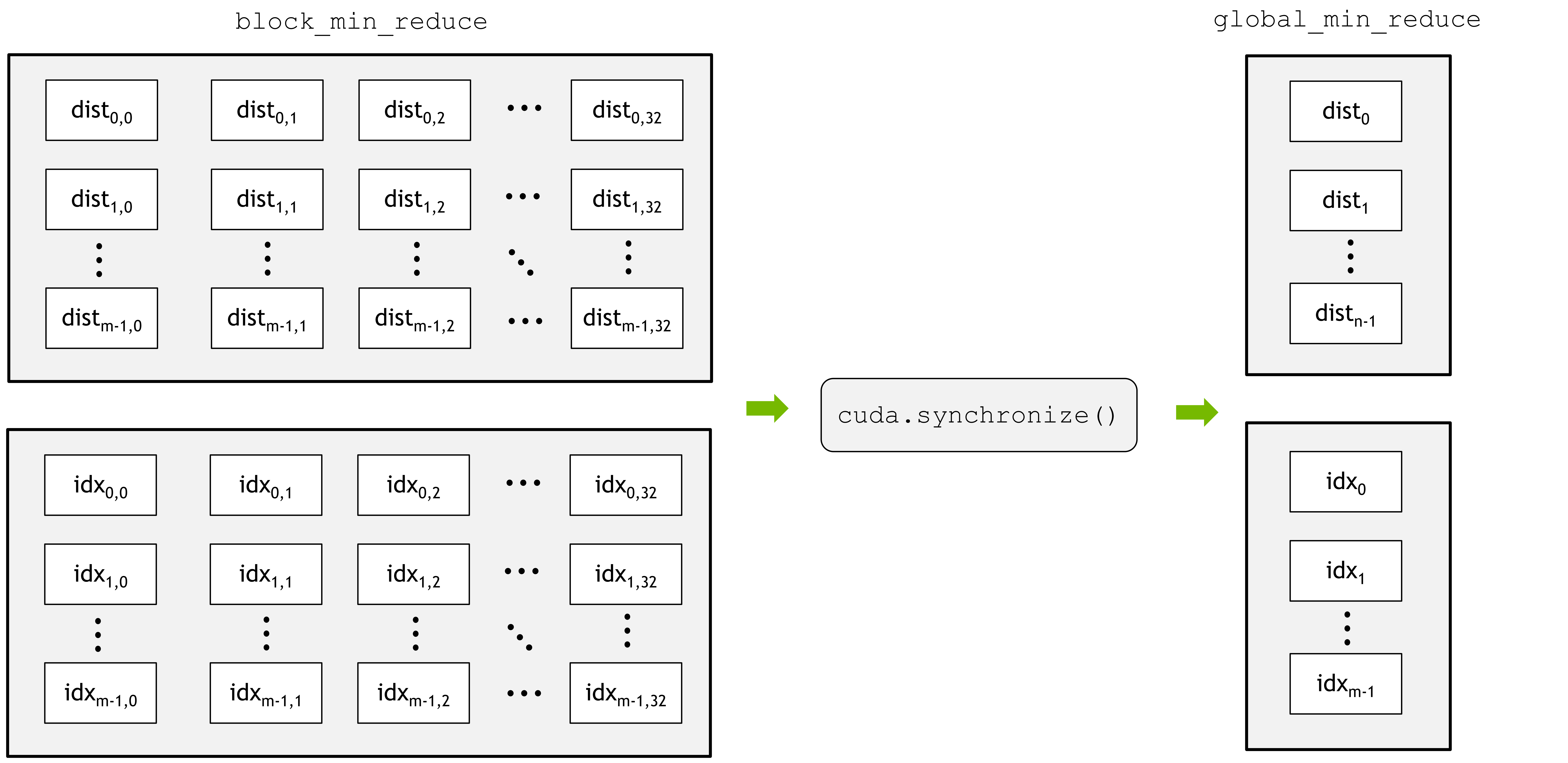
```
@cuda.jit(device=True, inline=True)
def warp_min_reduce_idx_unrolled(val, idx):

    mask  = 0xffffffff

    shfl_val = cuda.shfl_down_sync(
        mask, val, 16)

    shfl_idx = cuda.shfl_down_sync(
        mask, idx, 16)

    if val > shfl_val:
        val = shfl_val
        idx = shfl_idx

    shfl_val = cuda.shfl_down_sync(
        mask, val, 8)

    shfl_idx = cuda.shfl_down_sync(
        mask, idx, 8)

    if val > shfl_val:
        val = shfl_val
        idx = shfl_idx

     ……

    shfl_val = cuda.shfl_down_sync(
        mask, val, 1)

    shfl_idx = cuda.shfl_down_sync(
        mask, idx, 1)

    if val > shfl_val:
        val = shfl_val
        idx = shfl_idx

    return val, idx
```

```
@cuda.jit(
   "void(float32[:,:], float32[:,:], uint32[:,:], float32[:,:])",
   fastmath=True)
def block_min_reduce(coord1, coord2, block_idx, block_dist):

    """
    GPU accelerated pairwise distance comparisons in single
    precision.
    """

    startx, starty = cuda.grid(2)
    stridex, stridey = cuda.gridsize(2)

    seed = nb.float32(1e11)

    for i in range(starty, coord2.shape[0], stridey):

        b_min_val = seed
        b_min_idx = nb.uint32(0)

        for j in range(startx, coord1.shape[0], stridex):

            #simplified for slide presentation
            local_val = haversine(coord2, coord1)

            if local_val < b_min_val:
                b_min_val = local_val
                b_min_idx = j


        b_min_val, b_min_idx = \
            _warp_min_reduce_idx_unrolled(
            b_min_val, b_min_idx)

        if cuda.laneid == 0:
            block_dist[i, cuda.blockIdx.x] = b_min_val
            block_idx[i, cuda.blockIdx.x] = b_min_idx
```

```
@cuda.jit(
   "void(float32[:,:], uint32[:,:], float32[:], uint32[:])",
   fastmath=True)
def global_min_reduce(block_dist, block_idx, out_dist, out_idx):

    startx, starty = cuda.grid(2)
    stridex, stridey = cuda.gridsize(2)

    seed = float32(1e11)

    for i in range(starty, out_dist.shape[0], stridey):

        g_min_dist = seed
        g_min_idx = 0

        for j in range(startx, block_idx.shape[1], stridex):

            local_dist = block_dist[i, cuda.threadIdx.x]

            if local_dist < g_min_dist:
                g_min_dist = local_dist
                g_min_idx = block_idx[i, cuda.threadIdx.x]

        g_min_dist, g_min_idx = \
            _warp_min_reduce_idx_unrolled(
            g_min_dist, g_min_idx)

        if cuda.laneid == 0:
            out_dist[i] = g_min_dist
            out_idx[i] = g_min_idx
```

10.8 ms

HIGH LEVEL CUDA KERNEL



block_min_reduce
$dist_{0,0}$ $dist_{0,1}$ $dist_{0,2}$ ... $dist_{0,32}$
$dist_{1,0}$ $dist_{1,1}$ $dist_{1,2}$ ... $dist_{1,32}$
$dist_{m-1,0}$ $dist_{m-1,1}$ $dist_{m-1,2}$ ... $dist_{m-1,32}$
$idx_{0,0}$ $idx_{0,1}$ $idx_{0,2}$ ... $idx_{0,32}$
$idx_{1,0}$ $idx_{1,1}$ $idx_{1,2}$ ... $idx_{1,32}$
$idx_{m-1,0}$ $idx_{m-1,1}$ $idx_{m-1,2}$ ... $idx_{m-1,32}$
cuda.synchronize()
global_min_reduce
$dist_0$
$dist_1$
$dist_{n-1}$
idx_0
idx_1
idx_{m-1}

1st CUDA kernel calculates intermediate solution

After global synchronization, 2nd kernel calculates global solution

FIRST KERNEL COMPUTES INTERMEDIATE SOLUTION

Grid dimension determines width of intermediate solution
Warp minimum + argmin reduction
Lane
0 1 2 3 4 5 6 7
9 8 1 3 9 2 9 5
9 2 1 3
1 2
1
min (dist)
Lane
0 1 2 3 4 5 6 7
0 1 2 3 4 5 6 7
0 5 2 3
2 3
2
argmin

Note - For readability, we only show 8 of 32 threads per warp

```
mask  = 0xffffffff
shfl_val =
cuda.shfl_down_sync(
        mask, val, 16)

shfl_idx =
cuda.shfl_down_sync(
  mask, idx, 16)

if val > shfl_val:
  val = shfl_val
  idx = shfl_idx
```

Each block (green) in the grid (grey) solves a portion of the problem in parallel. Threads (yellow) communicate between other threads in the same warp (dotted line box) to perform a sequential addressing tree-based parallel reduction. An intermediate solution is generated of shape – m x cuda.blockSize.y

SECOND KERNEL COMPUTES GLOBAL SOLUTION

block dimension matches intermediate solution width
Warp minimum reduction + index
Lane 0 1 2 3 4 5 6 7
9 8 1 3 9 2 9 5
9 2 1 3
1 2
1
min (dist)
Lane 0 1 2 3 4 5 6 7
0 1 2 3 4 5 6 7
0 5 2 3
2 3
2
argmin

Note - For readability, we only show 8 of 32 threads per warp

```
mask  = 0xffffffff
shfl_val =
cuda.shfl_down_sync(
        mask, val, 16)

shfl_idx =
cuda.shfl_down_sync(
  mask, idx, 16)

if val > shfl_val:
  val = shfl_val
  idx = shfl_idx
```

Each block (green) in the grid (grey) solves a portion of the problem in parallel. Threads (yellow) communicate between other threads in the same warp (dotted line box) to perform a tree based parallel reduction. An intermediate solution is generated of shape – m x cuda.blockSize.y

Working in Notebook 5:
“Evaluating Your Options for Numerical Computing in Pure Python with Numba”

BENCHMARKS

Compare benchmarks from single threaded CPU and single GPU techniques

Single CPU	Timing	Single GPU	Timing	X-Factor (vs CPU equivalent)	X-Factor (vs Slowest CPU Option)	X-Factor (vs Fastest CPU Option)
Conventional For Loop	42 mins 34s					
NumPy Broadcast	12.2 s	CuPy Broadcast	277 ms	44 x	9,220 x	44 x
Sklearn Brute Force KNN	46.9 s	cuML Brute Force KNN	306 ms	153.2 x	8,355 x	39.8 x
Numba Kernel	35.4 s	**Numba CUDA**	**10.8 ms**	**3,277 x**	**236,481 x**	**1,129.6 x**

We observe the library and processor selected to perform the computations has a clear impact on performance.

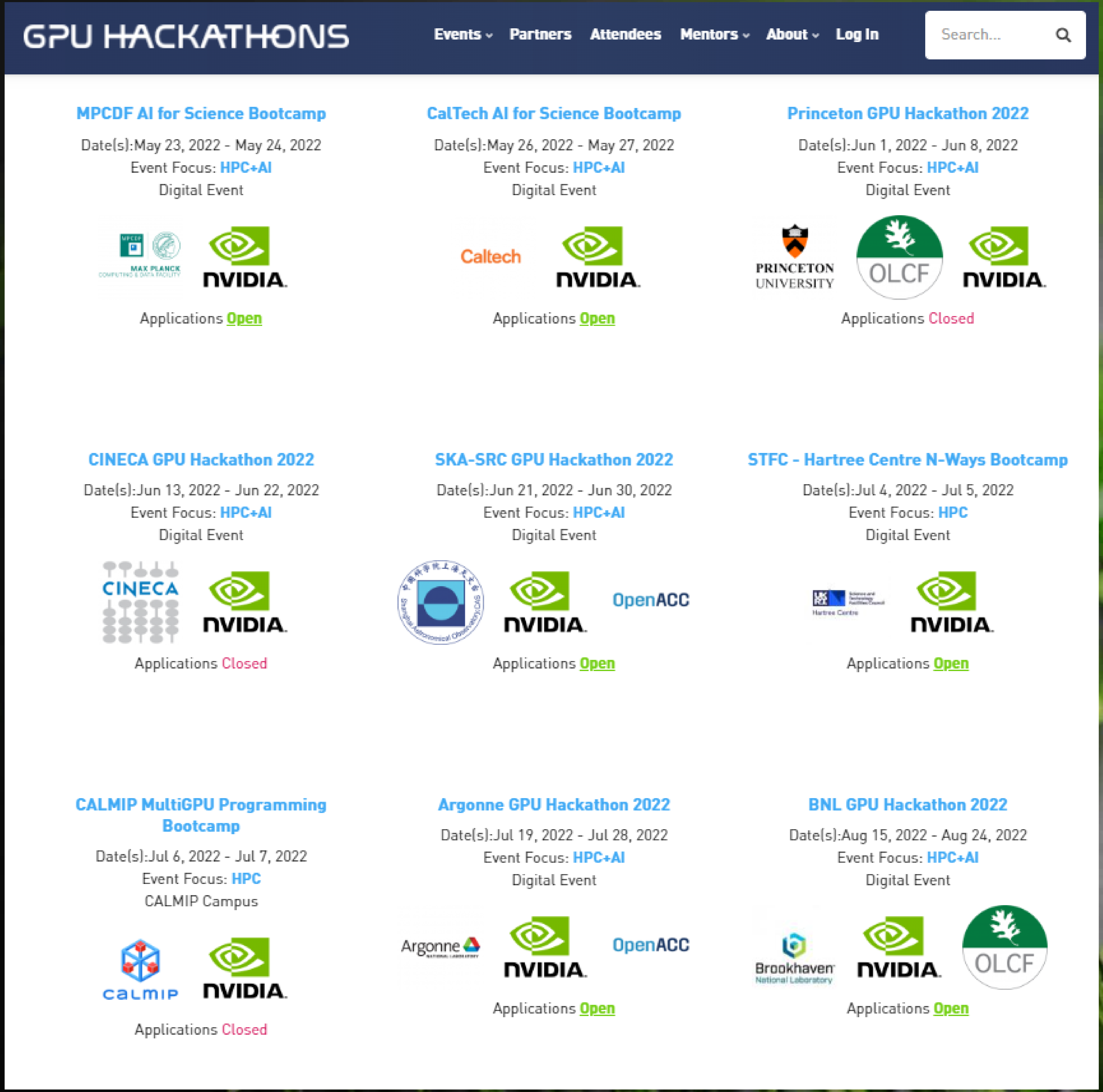
This is particularly true when choosing GPUs as the compute workhorse -- achieving anywhere from hundred(s) to million times speedup when compared to the single CPU techniques!

RECAP OF TOOLS USED TODAY

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

- The processor(s) selected to execute the numerical computing can have a tremendous impact on runtime speeds
- Every library is not created equally -- with little effort, a Python developer can achieve significant speedups in their code
- The GPU ecosystem of libraries available to developers has grown significantly -- RAPIDS, CuPy, Numba CUDA -- provide huge speedups with a familiar look and feel as their CPU counterparts

- *Python offers a great ecosystem for accelerating your applications with GPUs!*


GPU HACKATHONS

Events Partners Attendees Mentors About Log In Search...

MPCDF AI for Science Bootcamp
Date(s):May 23, 2022 - May 24, 2022
Event Focus: HPC+AI
Digital Event
Applications Open

CalTech AI for Science Bootcamp
Date(s):May 26, 2022 - May 27, 2022
Event Focus: HPC+AI
Digital Event
Applications Open

Princeton GPU Hackathon 2022
Date(s):Jun 1, 2022 - Jun 8, 2022
Event Focus: HPC+AI
Digital Event
Applications Closed

CINECA GPU Hackathon 2022
Date(s):Jun 13, 2022 - Jun 22, 2022
Event Focus: HPC+AI
Digital Event
Applications Closed

SKA-SRC GPU Hackathon 2022
Date(s):Jun 21, 2022 - Jun 30, 2022
Event Focus: HPC+AI
Digital Event
Applications Open

STFC - Hartree Centre N-Ways Bootcamp
Date(s):Jul 4, 2022 - Jul 5, 2022
Event Focus: HPC
Digital Event
Applications Open

CALMIP MultiGPU Programming Bootcamp
Date(s):Jul 6, 2022 - Jul 7, 2022
Event Focus: HPC
CALMIP Campus
Applications Closed

Argonne GPU Hackathon 2022
Date(s):Jul 19, 2022 - Jul 28, 2022
Event Focus: HPC+AI
Digital Event
Applications Open

BNL GPU Hackathon 2022
Date(s):Aug 15, 2022 - Aug 24, 2022
Event Focus: HPC+AI
Digital Event
Applications Open

https://openhackathons.org

NVIDIA®

Extra content:

Single Node, Multi-CPU & Multi-GPU Techniques

WHAT WE WILL EVALUATE

Single Node, Multi-CPU and Multi-GPU Methodologies for Scaling Compute

When the problem size increases (without relaxing latency expectations) slower and less efficient techniques become unusable…

Fortunately for Python developers, that doesn't always mean its time to port to C/C++!

Single Node, Multi-CPU
Numba Kernel (prange)
Single Node, Multi-GPU
Dask Numba CUDA Kernel
Python Threading + Numba CUDA

Scaled Problem Size: 274.88B
2^{24} (observations) * 2^{14} (reference points)

Compute time becomes extremely long for slower methods and memory footprint at runtime turns into a limiting factor. This forces us to triage our techniques and explore multi-processor methods.

NUMBA PRANGE

NumPy and CuPy share nearly identical syntax but a huge speed disparity

```
import numpy as np
from numba import jit, prange

@jit(nopython=True, parallel=True)
def numba_multi_cpu_solve(a, b):

    out_idx = np.empty(
        (a.shape[0]), dtype=np.uint32)

    out_dist = np.empty(
        (a.shape[0]), dtype=np.float32)

    for obs_idx in prange(a.shape[0]):

        glob_min_dist = 1e11
        glob_min_idx = 0

        for ref_idx in range(b.shape[0]):

            temp_dist = numba_cpu_haversine(
                a[obs_idx,0],
                a[obs_idx, 1],
                b[ref_idx, 0],
                b[ref_idx, 1])

            if temp_dist < glob_min_dist:
                glob_min_dist = temp_dist
                glob_min_idx = ref_idx

        out_dist[obs_idx] = glob_min_dist
        out_idx[obs_idx] = glob_min_idx

    return out_idx, out_dist
```

```
@jit(nopython=True, fastmath=True)
def numba_cpu_haversine(lat1, lon1, lat2, lon2):

    first_sin = math.sin((lat2 - lat1) / 2.0)
    second_sin = math.sin((lon2 - lon1) / 2)

    return 2.0 * math.asin(
        math.sqrt(first_sin * first_sin + \
                math.cos(lat1) * \
                math.cos(lat2) * \
                second_sin * second_sin)
    )
```

Numba

2hrs 3min 31s

Extremely similar to the Numba implementation – just replace “range” with “prange”

Numba’s “prange” function leverages ALL CPU cores to solve a problem

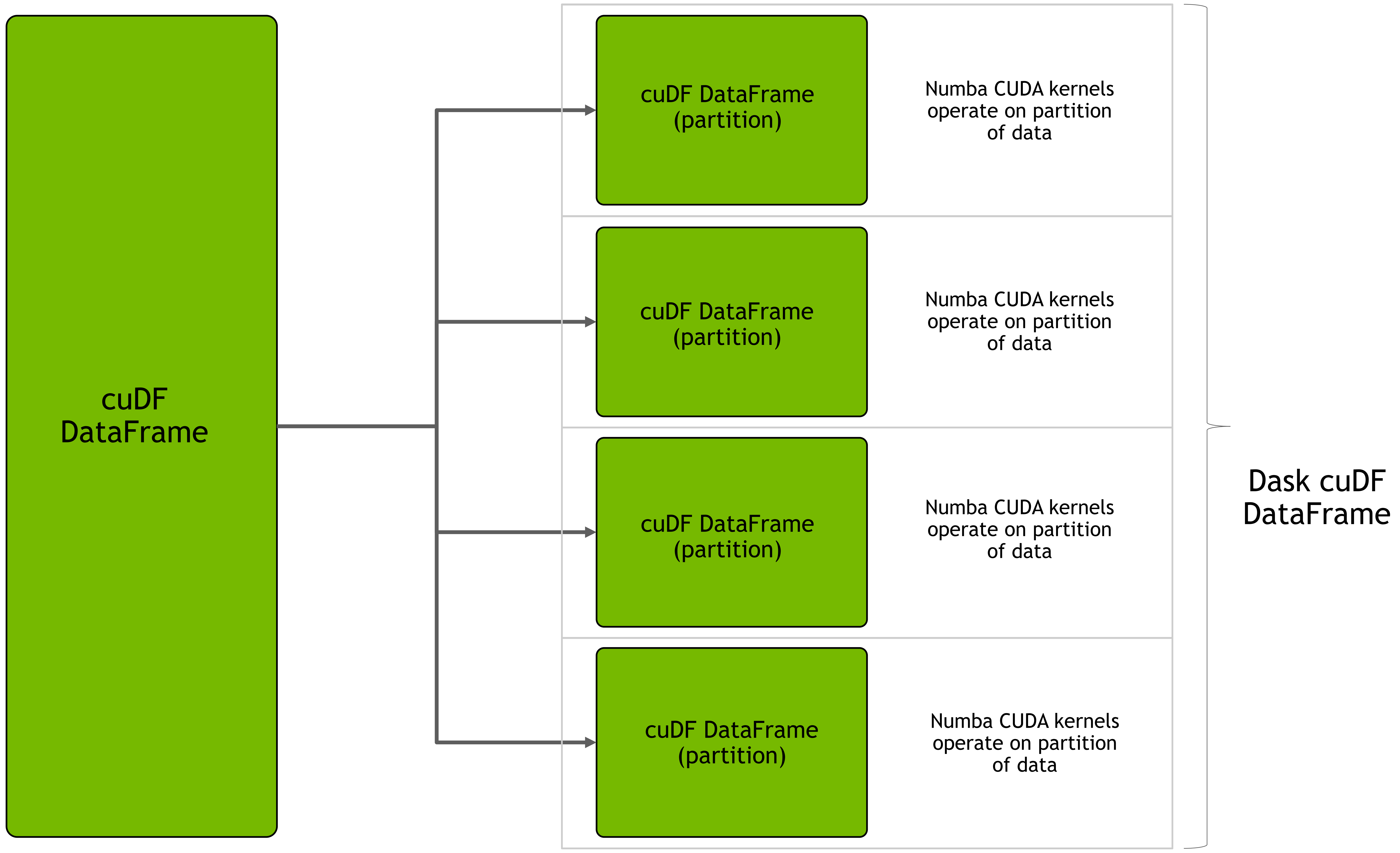
We achieve a “same day solution” but nothing usable in near real-time applications

Good news, we still have other options!

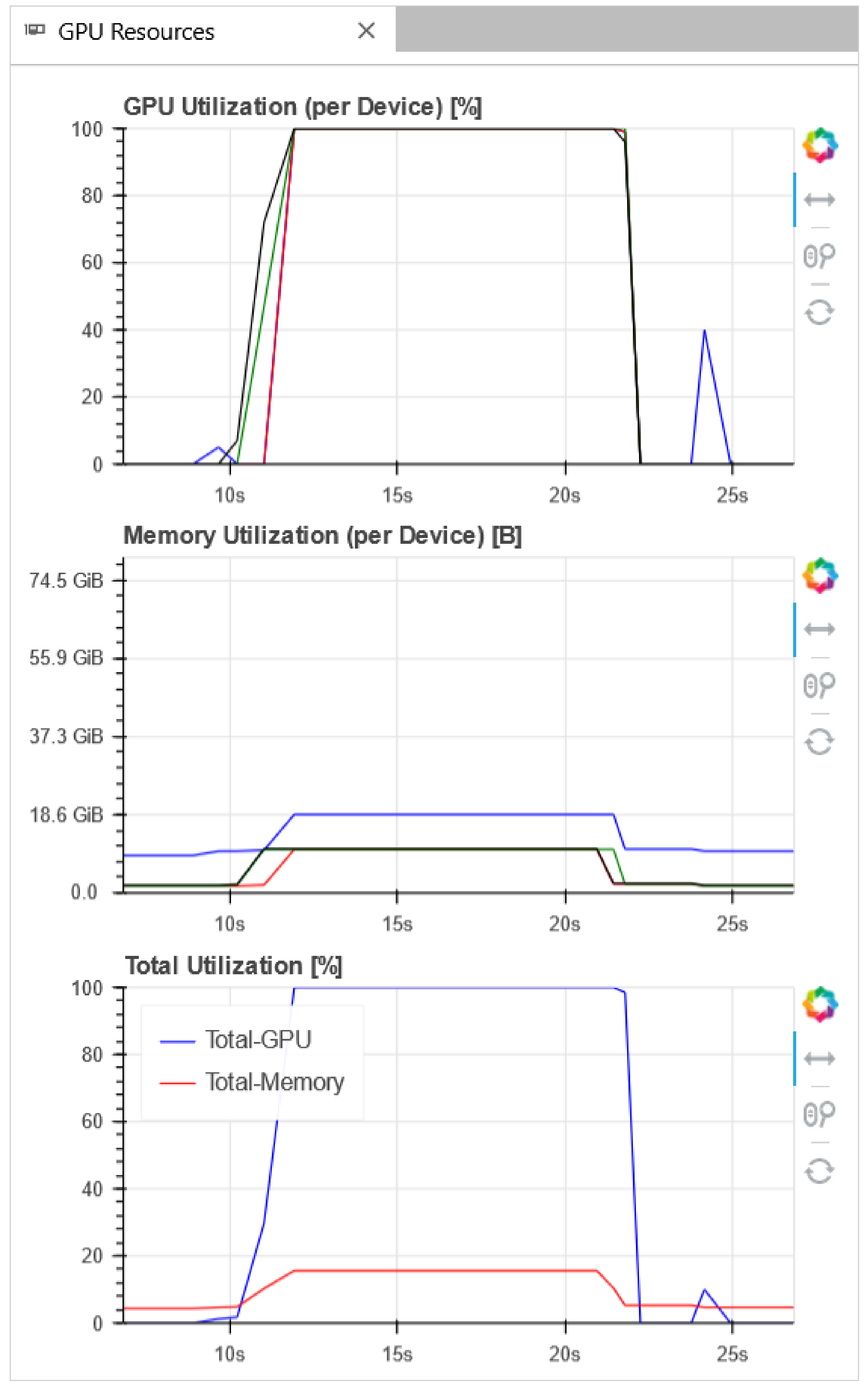
MULTI-GPU EXECUTION WITH DASK

Leverage the dask_cudf API to construct a local GPU cluster and farm out Numba CUDA kernel execution

Create Dask cuDF DataFrame to partition data and use a pool of GPUs to complete work


cuDF DataFrame
cuDF DataFrame (partition)
Numba CUDA kernels operate on partition of data
cuDF DataFrame (partition)
Numba CUDA kernels operate on partition of data
cuDF DataFrame (partition)
Numba CUDA kernels operate on partition of data
cuDF DataFrame (partition)
Numba CUDA kernels operate on partition of data
Dask cuDF DataFrame

All available GPUs fully utilized during runtime



JupyterLab extension for GPU utilization dashboards -- included in the RAPIDS container
jupyterlab_nvdashboard

COMPUTING A DASK TASK GRAPH

Behind the scenes, Dask creates a task graph that schedule compute until jobs are complete

DASK CUDF + NUMBA

NumPy and CuPy share nearly identical syntax but a huge speed disparity

Implicit use of the CUDA Array Interface

Double Numba CUDA call sequence

```
def get_nearest(
   part_df, coord2=None, block_idx=None, block_dist=None):

    coord1 = part_df[["LAT_RAD", "LON_RAD"]].as_gpu_matrix()
    coord2 = coord2.as_gpu_matrix()

    block_idx_mat = cp.empty((coord1.shape[0], 32), dtype=np.uint32)
    block_dist_mat = cp.empty((coord1.shape[0], 32),
dtype=np.float32)

    out_idx = cp.empty(
        (coord1.shape[0]),
        dtype=np.uint32
    )

    out_dist = cp.empty(
        (coord1.shape[0]),
        dtype=np.float32
    )

    bpg = 32, 108
    tpb = 32, 16

    _block_get_nearest_brute[bpg, tpb](
        coord2,
        coord1,
        block_idx_mat,
        block_dist_mat
    )

    bpg = (1, 108*20)
    tpb = (32, 16)

    _global_get_nearest_brute[bpg, tpb](
        block_dist_mat,
        block_idx_mat,
        out_dist,
        out_idx
    )

    cuda.synchronize()

    part_df["out_idx"] = out_idx
    part_df["out_dist"] = out_dist

    return (part_df)
```

```
from dask_cuda import LocalCUDACluster
from dask.distributed import Client
import cudf
import dask_cudf
from numba import cuda, int32, float32, jit

import numpy as np

cluster = LocalCUDACluster()
client = Client(cluster)

…

ddf = dask_cudf.from_cudf(gdf_obs, npartitions=4)
gdf_result = ddf.map_partitions(
    get_nearest,
    coord2=gdf_ref,
).compute()
```

14.5s

Instantiate a local and programmable CUDA GPU cluster

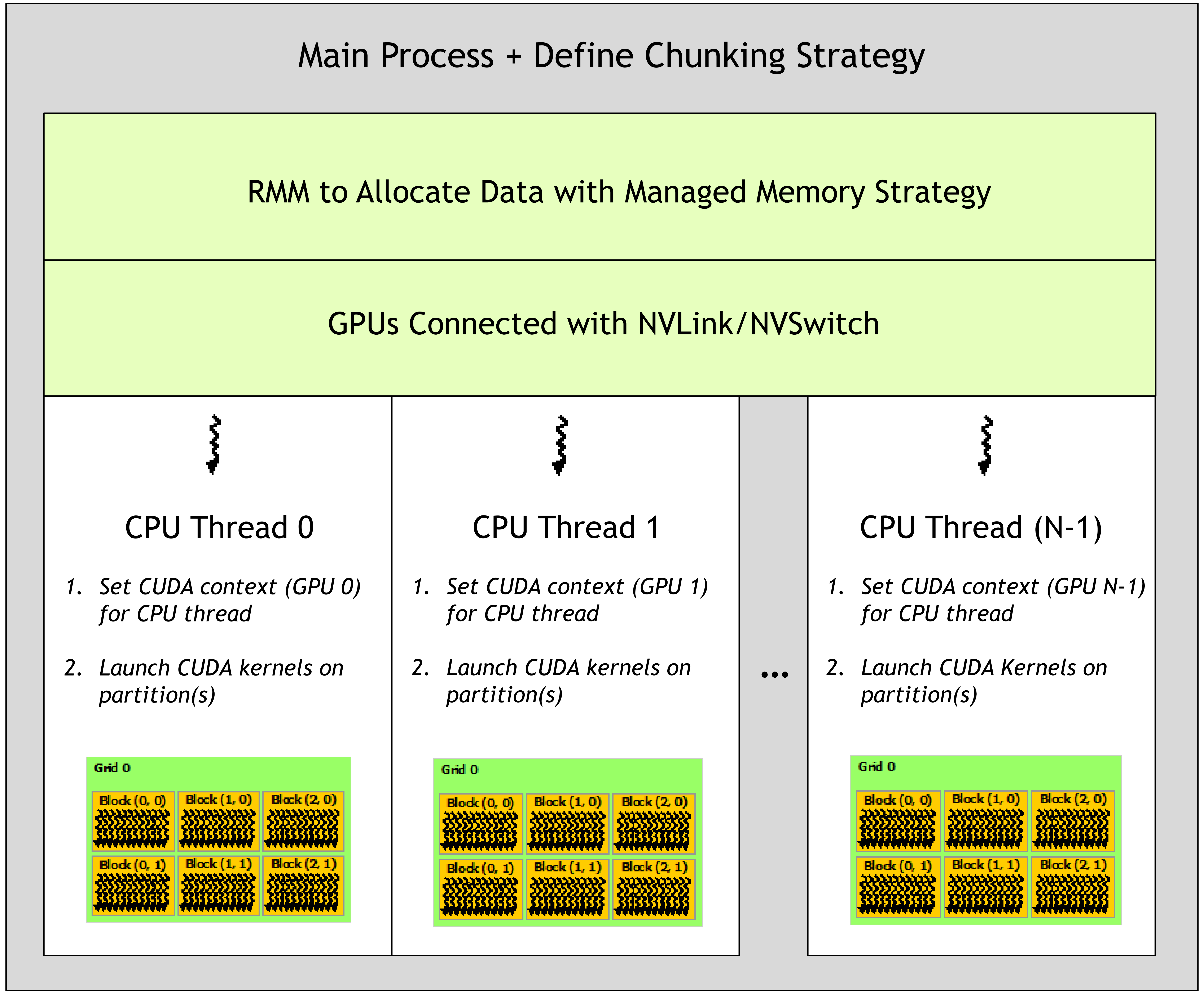
Distribute data cross cluster and launch kernel on data partitions

Results are returned to GPU DataFrame on the default GPU

Without much effort, we use Dask cuDF, we achieve a comfortable 511x boost in speed!

MULTI-THREADED, MULTI-GPU IMPLEMENTATION

Leverage Python threading library + Rapids Memory Manager + Numba CUDA kernels schedule work on available GPUs



Main Process + Define Chunking Strategy

RMM to Allocate Data with Managed Memory Strategy

GPUs Connected with NVLink/NVSwitch

CPU Thread 0

1. *Set CUDA context (GPU 0) for CPU thread*
2. *Launch CUDA kernels on partition(s)*

CPU Thread 1

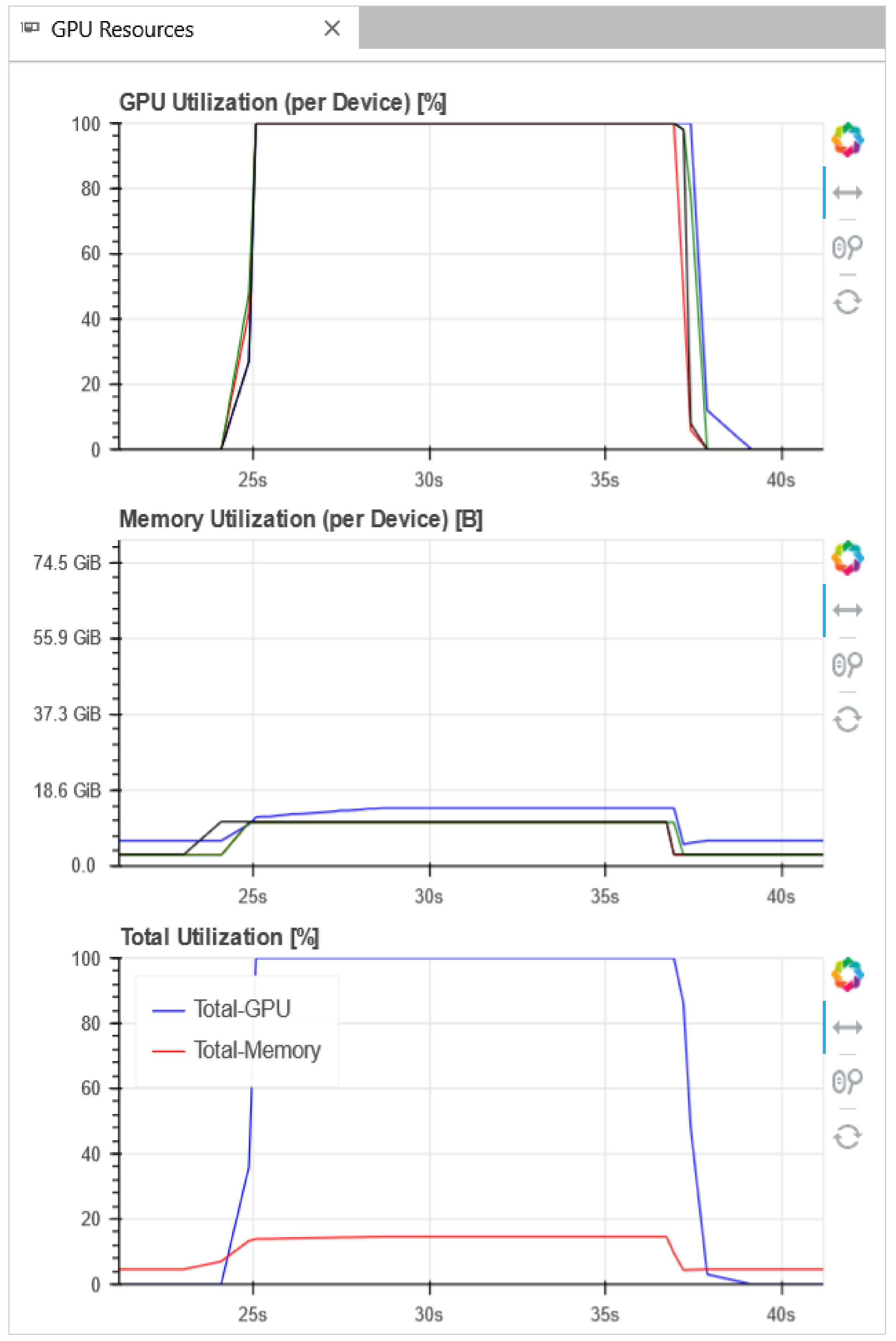
1. *Set CUDA context (GPU 1) for CPU thread*
2. *Launch CUDA kernels on partition(s)*

...

CPU Thread (N-1)

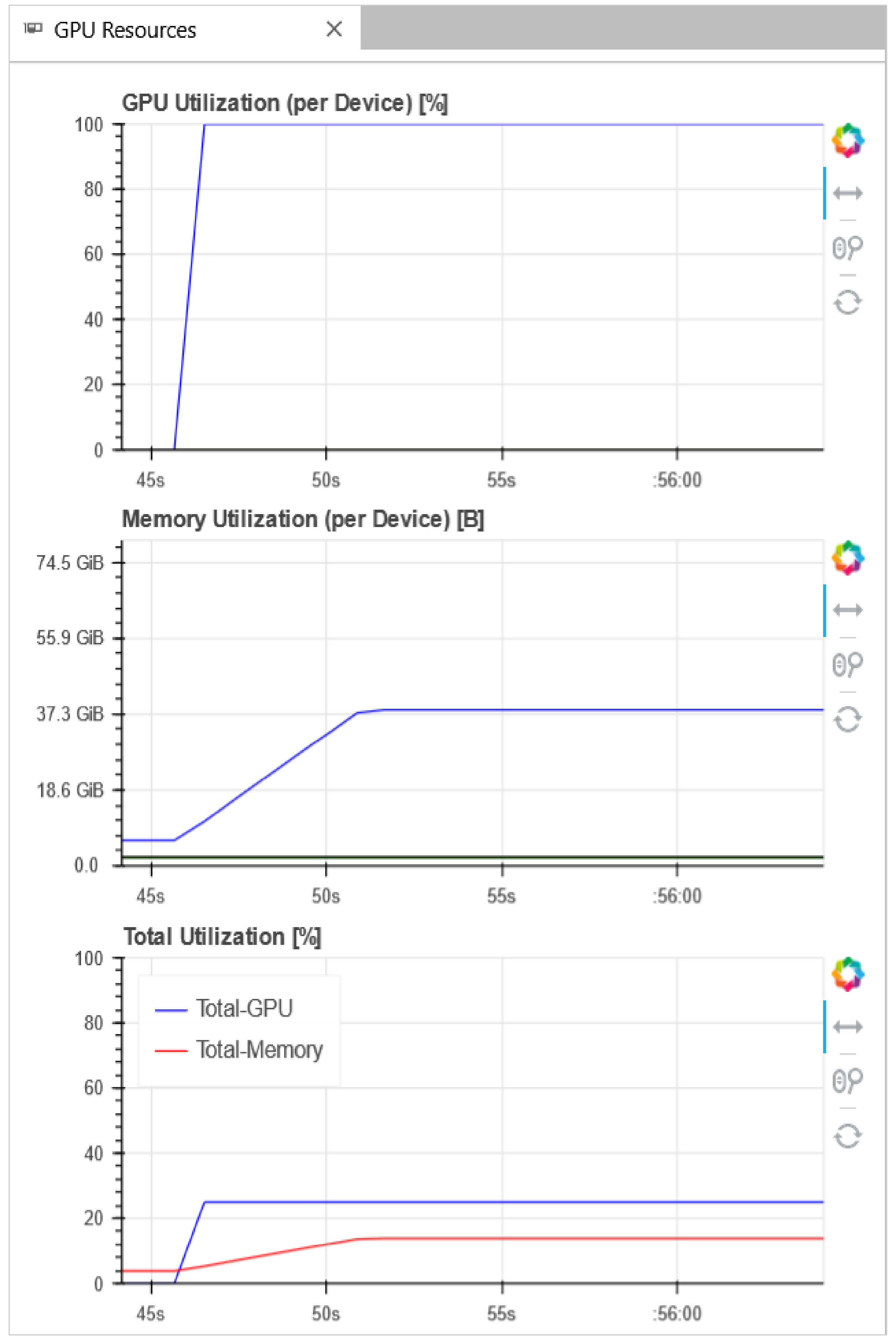
1. *Set CUDA context (GPU N-1) for CPU thread*
2. *Launch CUDA Kernels on partition(s)*

All available GPUs fully utilized during runtime



VS

Single GPU fully utilized during runtime (25% overall)



JupyterLab Extension for GPU utilization dashboards
jupyterlab_nvdashboard

MANUAL THREADING + RMM + NUMBA CUDA

```
# GPU imports
from numba import cuda, int32, float32, types
import cupy as cp
import numba as nb
import rmm
import cudf

# CPU imports
import threading
import queue
import numpy as np

# use managed memory for allocations
cuda.set_memory_manager(rmm.RMMNumbaManager)

rmm.mr.set_current_device_resource(
    rmm.mr.ManagedMemoryResource())
```

In this example, we demonstrate use of the RMM and Python threading to schedule asynchronous kernel execution on data chunks using our previously developed Numba CUDA kernels:

- Share data between GPUs using NVLink/NVSwitch
- Generates data partitions
- Build queues of compute on those partitions to be scheduled on available GPUs
- Completes our work in ~12.7s
- Roughly 12% faster than dask_cudf
- 584x faster than the multi-processing technique

```
def get_nearest(
    obs_points, ref_points, out_idx, out_dist,
    batch_size="auto", multigpu=True, n_gpus="auto"):
    …
    if n_gpus == "auto":
        n_gpus = len(cuda.list_devices())

    size = obs_points.shape[0]

    if batch_size == 'auto':
        batch_size = size/n_gpus

    batch_size = int(batch_size)
    n_jobs = int(size / min(batch_size, size))
    queues = [queue.Queue() for i in range(n_gpus)]
    qid = 0

    for j in range(n_jobs):

        if qid >= len(queues):
            qid = 0

        job = {}
        start = j * batch_size

        if j == (n_jobs - 1):
            end = size
        else:
            end = (j + 1) * batch_size

        job["start"] = start
        job["end"] = end
        job["d_m_ref"] = ref_points
        job["d_m_obs"] = obs_points
        job["d_m_out_idx"] = out_idx
        job["d_m_out_dist"] = out_dist

        queues[qid].put(job)
        qid += 1

    workers = []
    for qid in range(len(queues)):

        w = threading.Thread(
            target=_get_nearest_multi,
            args=[queues[qid], qid])

        w.start()
        workers.append(w)

    for w in workers:
        w.join()
```

```
def _get_nearest_multi(q, cid):

    cuda.select_device(cid) # bind device to thread

    while(q.unfinished_tasks > 0):

        job = q.get()

        d_ref = cuda.to_device(
            job["d_m_ref"]
        )

        d_obs = cuda.to_device(
            job["d_m_obs"][job["start"]:job["end"]]
        )

        d_block_idx = cuda.device_array(
            (job["end"] - job["start"], 32),
            dtype=np.uint32)

        d_block_dist = cuda.device_array(
            (job["end"] - job["start"], 32),
            dtype=np.float32)

        bpg = 32, 108
        tpb = 32, 16

        block_min_reduce[bpg, tpb](
            d_ref,
            d_obs,
            d_block_idx,
            d_block_dist
        )

        bpg = (1, 108*20)
        tpb = (32, 16)

        global_min_reduce[bpg, tpb](
            d_block_dist,
            d_block_idx,
            job["d_m_out_dist"][job["start"]:job["end"]],
            job["d_m_out_idx"][job["start"]:job["end"]]
        )

        cuda.synchronize()

        q.task_done()
```

Implicit use of CUDA Array Interface

Our familiar double kernel launch pattern

12.7s

This tailor-made implementation, squeezes an additional XX % performance on this problem!

BENCHMARKS

Compare benchmarks from multi-CPU and multi-GPU techniques

Implementation	Timing	X-Factor (vs Multi CPU Option)
Numba Multi CPU (prange)	2hrs 3min 31s	1x
Dask + Numba CUDA	14.5s	511x
Threading + RMM + Numba CUDA	12.7s	584x

As we scaled up our problem size, it became necessary to consider distributing work to multiple processors. In this section, we demonstrated an ability to implement this and achieve some excellent performance.

Utilizing all GPUs on a DGX Station A100 (4 GPUs) boosted performance by almost 600x over the multi-CPU processing method!

DISCUSSION – FURTHER BENCHMARKING

BENCHMARKS

All techniques on the initial problem size -- MxN = 0.54 B

	Methodology	Timing	X-Factor (vs slowest single-CPU)	X-Factor (vs fastest single CPU)	X-Factor (vs Multi-CPU)
Single CPU	Conventional For Loop	42 mins 34s	1x	0.0047 x	0.00043 x
	NumPy Broadcast	**12.2 s**	**209 x**	**1 x**	**0.09 x**
	Sklearn Brute Force KNN	46.9 s	54 x	0.26 x	0.023 x
	Numba Kernel	35.4 s	72 x	0.34 x	0.03x
Multi-CPU	**Numba Kernel (prange)**	**1.1 s**	**2.322 x**	**11.1 x**	**1 x**
Single GPU	CuPy Broadcast	277 ms	9,220 x	44 x	3.9 x
	cuML Brute Force KNN	306 ms	8,355 x	39.8 x	3.6 x
	Numba CUDA	**10.8 ms**	**236,481 x**	**1,129.6 x**	**101.9 x**
Multi-GPU	Dask cuDF + Numba CUDA	530 ms	4,818.9 x	23 x	2.1 x
	Threading + RMM + Numba CUDA	**9 ms**	**283,777.8 x**	**1,355.6 x**	**122.2 x**

Note – For completeness, we included multi-CPU and multi-GPU benchmarks for the small problem size

BENCHMARKS

All techniques on the initial problem size -- MxN = 0.54B

	Methodology	Timing	X-Factor (vs slowest single-CPU)	X-Factor (vs fastest single CPU)
Single CPU	Conventional For Loop	42 mins 34s	1x	0.0047 x
	NumPy Broadcast	**12.2 s**	**209 x**	**1 x**
	Sklearn Brute Force KNN	46.9 s	54 x	0.26 x
	Numba Kernel	35.4 s	72 x	0.34 x
Single GPU	CuPy Broadcast	277 ms	9,220 x	44 x
	cuML Brute Force KNN	306 ms	8,355 x	39.8 x
	Numba CUDA	**10.8 ms**	**236,481 x**	**1,129.6 x**

BENCHMARKS

All techniques on the initial problem size -- MxN = 4.29B

	Methodology	Timing	X-Factor (vs slowest single-CPU)	X-Factor (vs fastest single CPU)
Single CPU	Conventional For Loop	4 hr 52 mins 14s	1 x	0.0055 x
	NumPy Broadcast	**1 min 37 s**	**190.6 x**	**1 x**
	Sklearn Brute Force KNN	6 mins 14 s	46.9 x	0.26 x
	Numba Kernel	4 min 41 s	62.4 x	0.35 x
Single GPU	cuML Brute Force KNN	1.87 s	9,376 x	51.9 x
	Numba CUDA	**65 ms**	**269,753 x**	**1,492 x**

Note – CuPy broadcast methodology did not complete due to the limited memory of the T4. With larger GPUs this method would outperform our other options.

BENCHMARKS

All techniques on the initial problem size -- MxN = 4.29 B

	Methodology	Timing	X-Factor (vs slowest single-CPU)	X-Factor (vs fastest single CPU)	X-Factor (vs Multi-CPU)
Single CPU	Conventional For Loop	4 hr 52 mins 14s	1 x	0.0055 x	0.0005 x
	NumPy Broadcast	**1 min 37 s**	**190.6 x**	**1 x**	**0.09 x**
	Sklearn Brute Force KNN	6 mins 14 s	46.9 x	0.26 x	0.023 x
	Numba Kernel	4 min 41 s	62.4 x	0.35 x	0.03 x
Multi-CPU	**Numba Kernel (prange)**	**8.72 s**	**2,010.7 x**	**11.1 x**	**1 x**
Single GPU	cuML Brute Force KNN	1.87 s	9,376 x	51.9 x	4.66 x
	Numba CUDA	**65 ms**	**269,753 x**	**1,492 x**	**134.2 x**
Multi-GPU	Dask cuDF + Numba CUDA	554 ms	31,650 x	175.1 x	15.7 x
	Threading + RMM + Numba CUDA	**199 ms**	**88,110 x**	**487.4 x**	**43.8 x**

Note – For completeness, we included multi-CPU and multi-GPU benchmarks for the medium problem size

BENCHMARKS

Multi-CPU vs Fastest GPU Methods techniques on the scaled-up problem size -- MxN = 274.88 B

	Methodology	Timing	X-Factor (vs Multi-CPU)
Multi-CPU	Numba Kernel (prange)	9 min 17 s	1 x
Single GPU	cuML Brute Force KNN	2 min 11 s	4.25 x
	Numba CUDA	5.05 s	110 x
Multi-GPU	Dask cuDF + Numba CUDA	2.2 s	253 x
	Threading + RMM + Numba CUDA	1.77 s	314.7 x

From our prior performance tables, we recall the significant performance gains we obtain by leveraging all 24 cores of our CPU.

Our multi-GPU techniques achieve over a 200x speedup when compared to our multi-CPU technique. Also impressive, our single GPU techniques outperform the multi-CPU (24 cores) method by orders of magnitude on our scaled-up problem!

Note – Because of their speed, we also apply our most efficient single-GPU techniques to the scaled-up problem. Due to their slowness and time constraints, this was not feasible for the CPU alternatives.

KEY TAKEAWAYS

During this talk, we explored a wide range of n-dimensional array computing techniques through the lens of a popular nearest neighbors proxy problem. This problem allowed us to demonstrate the feasibility of each method and assess its strengths and weaknesses. Clearly, the motivated Python developer has many options to accelerate their numerical computing workloads while staying comfortably in their “native language” -- Python.

Single CPU
Conventional For Loop
Scikit-Learn Brute Force KNN
NumPy Broadcasting
Numba CPU Kernel
Single GPU
cuML Brute Force KNN
CuPy Broadcasting
Numba GPU Kernel

Single Node, Multi-CPU
Numba Kernel (prange)
Single Node, Multi-GPU
Dask Numba CUDA Kernel
Python Threading + Numba CUDA

- The processor(s) selected to execute the numerical computing can have a tremendous impact on runtime speeds
- Every library is not created equally -- with little effort, a Python developer can achieve significant speedups in their code
- The GPU ecosystem of libraries available to developers has grown significantly -- RAPIDS, CuPy, Numba CUDA -- provide huge speedups with a familiar look and feel as their CPU counterparts

Whether latency or cost of computation is more important, GPUs are the dominant strategy!

NVIDIA®