CUDA IN PYTHON

Modern computing for physics

J.Pazzini
Padova University

Physics of Data
AA 2024-2025

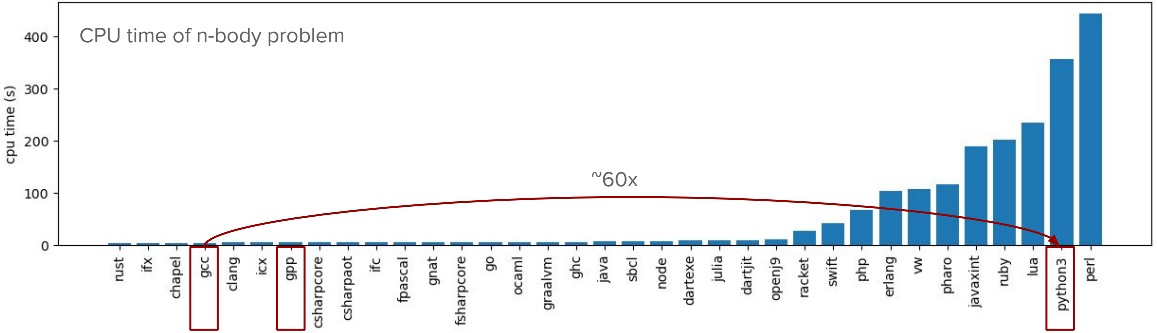
UNIVERSITÀ DEGLI STUDI DI PADOVA

Speeding up Python

Is Python really slow compared to other languages (e.g. C/C++/Fortran)?

➜ Unfortunately, yes…*

https://benchmarksgame-team.pages.debian.net/benchmarksgame/performance/nbody.html



This is due to a series of factors, typically boiling down to the Python design principles of ease of use, readability, and quick development:

- (Mostly) interpreted vs (purely) Compiled
- Dynamically (although it can be statically) vs Statically typed
- Automatic vs Manual memory management

* all languages are still evolving, and Python in its recent and future versions is changing dramatically, extending its scope and improving its performance

Speeding up Python

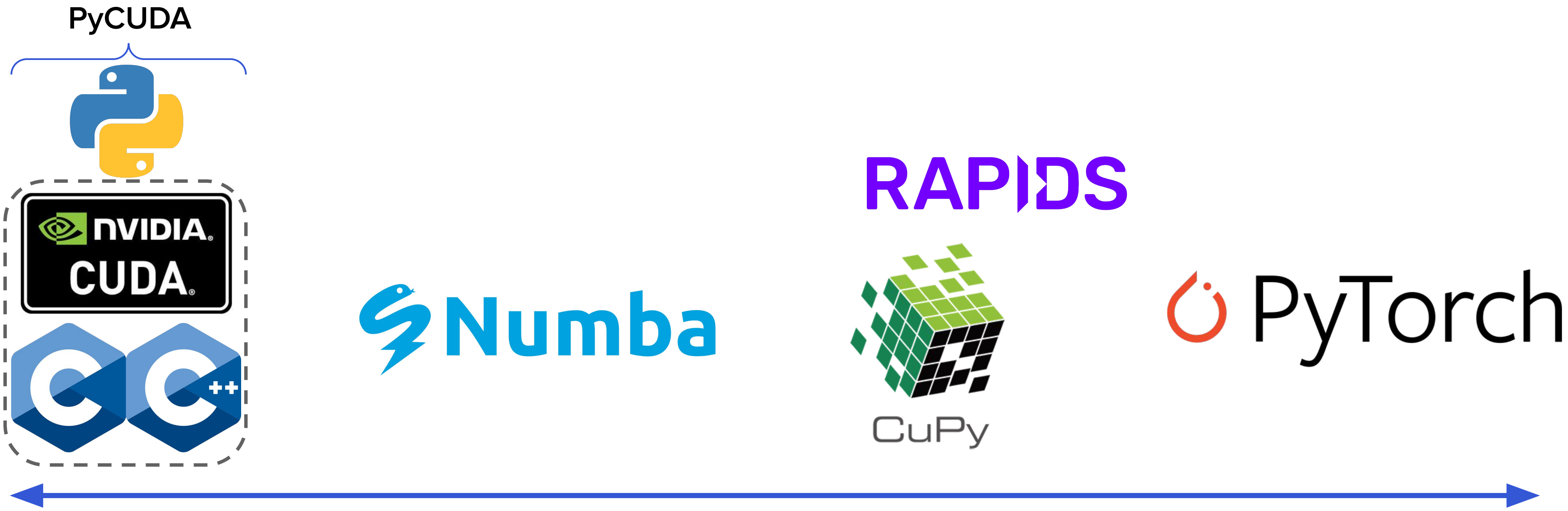
You know some of these tricks already:

- Use Built-In Functions and Libraries
  - Usually implemented in C and highly optimized
- Use dedicated libraries such as NumPy
  - At its core implemented in C, with parts in Cython (compiled Python)
  - Including CPU vectorization (via SIMD instructions) and multithreading
- Use multithreading and/or multiprocessing
  - Depending on the task being IO-bound or CPU-bound
- Use multiprocessing libraries/frameworks with lazy execution such as Polars, Dask
  - Typically relevant for very-large datasets, due to the optimized task scheduling

➔ What about including some “quick” compilation in Python?

Python on GPU - The alternatives

What about using Python on GPUs for achieving acceleration of (parts of) the task?



Lower level:

- Better description of custom kernels
- More complex deployment as drop-in code

Higher level:

- Very difficult description of custom kernels
- Easier deployment as drop-in code

Python on GPU - The alternatives

What about using Python on GPUs for achieving acceleration of (parts of) the task?

PyCUDA

A great inbetweener to achieve both CPU speedup of pure-Python code, and heterogeneous computing with CUDA kernels

Lower level:

- Better description of custom kernels
- More complex deployment as drop-in code

Higher level:

- Very difficult description of custom kernels
- Easier deployment as drop-in code

Numba

Numba - A general overview

Numba is a **Just-In-Time**, **type-specializing**, **function compiler** for accelerating **numerically-focused** Python

Numba - A general overview

Numba is a **Just-In-Time**, **type-specializing**, **function compiler** for accelerating **numerically-focused** Python

→ Numba works by generating optimized machine code compiling Python functions (not entire applications)

→ Numba speeds up functions by generating a specialized implementation for the specific data types used (e.g. int4, float8, etc)

→ Numba translates functions when they are first called

→ Numba is focused on numerical data types, like int, float, and complex. There is very limited string processing support

Numba also supports **CUDA GPU programming** by directly compiling a restricted subset of Python code into CUDA kernels

Numba - Decorators

Numba is most often used via decorators:

- **`@jit/@njit`** ➜ Just-In-Time compilation of purely pythonic/NumPy functions into machine code

- **`@vectorize/@guvectorize`** ➜ Creates a custom NumPy ufunc (universal function) operating on numpy ndarrays

- …
  (see the documentation for other interesting applications, such as @jitclass to JIT compile entire Python classes)

JIT - in short

Numba's most used feature is the Just-In-Time compiler:

- Converts Python code into fast, machine-level code at runtime
- Significantly speeds up numerical computations, especially with NumPy arrays
- Minimizes Python interpreter overhead
- Offers simple integration with minimal code changes

```
def compute_pi_montecarlo(num_samples):
    count_inside = 0
    for _ in range(num_samples):
        x, y = np.random.rand(), np.random.rand()
        if x**2 + y**2 <= 1.0:
            count_inside += 1
    return 4.0 * count_inside / num_samples
```

1.13 s ± 366 ms

```
@jit(nopython=True, parallel=True)
def compute_pi_montecarlo(num_samples):
    count_inside = 0
    for _ in numba.prange(num_samples):
        x, y = np.random.rand(), np.random.rand()
        if x**2 + y**2 <= 1.0:
            count_inside += 1
    return 4.0 * count_inside / num_samples
```

6.78 ms ± 3.17 ms

x166 speedup

VECTORIZE - in short

Additionally, Numba provides a way to create fast and compiled NumPy-like functions:

- Automatically creates element-wise operations for arrays
- Converts Python functions into fast, compiled **`ufuncs`** (universal functions)
- Supports broadcasting and parallel execution
- Greatly improves performance for array operations with minimal code changes

```
def sum_of_squares(a, b):
    return a**2 + b**2

a = np.linspace(0 ,10,1_000_000)
b = np.linspace(10,20,1_000_000)
```

`4.42 ms ± 346 µs`

```
@vectorize(['float64(float64,float64)'])
def sum_of_squares(a, b):
    return a**2 + b**2

a = np.linspace(0 ,10,1_000_000)
b = np.linspace(10,20,1_000_000)
```

`1.79 ms ± 278 µs`

x2.5 speedup

Numba + CUDA

Numba + CUDA

Numba also offers an interface to deploy computation on the GPU

- Numba can JIT-compile a Python code to low-level machine code
  - Including a backend PTX code compiler to create GPU-code using the CUDA Driver APIs

- It also contains a Python wrapper to part of the CUDA driver APIs (e.g. to inspect available devices, query their capabilities, etc)
  - This is what’s used under the hood, and most of the time you don’t need to know what it’s doing

- If any of the inputs are in host memory, then they need to be transferred to the device.
  - Numba-CUDA offers a NumPy-like array library, used for managing arrays on CUDA devices (referred to as device arrays)

Numba + CUDA - Decorators

An almost complete equivalent to the standard “plain” Numba:

- **`@cuda.jit`** ➜ Just-In-Time compilation of pythonic function into CUDA
  Decorating a function as **`@cuda.jit`** is like declaring a **`__global__`** kernel:
  it can be called as a function from the host, but will be executed on the device

- **`@vectorize(target='cuda')`** ➜ Creates a custom CUDA kernel operating elementwise
  `@guvectorize(target='cuda')` on all elements of the inputs

VECTORIZE(target=′ cuda′) - in short

We have already seen how Numba offers the possibility of declaring the GPU as the target of the vectorize function, converting pythonic code in CUDA kernels

- Automatically creates and compile element-wise CUDA kernel for NumPy-like arrays
- Kernels will act as ufunc on input data
- Supports broadcasting of input

```
@vectorize(['float32(float32, float32, float32)'],
           target='cuda')
def gaussian_pdf(x, mean, sigma):
    return math.exp(-0.5 * ((x - mean) / sigma)**2) / (sigma * SQRT_2PI)
```

CUDA.JIT - in short

For all those functions that are not elementwise operations, Numba still offers a way to write CUDA kernels in Python

- Automatically creates and compile arbitrary CUDA kernels
- Has access to the CUDA registers related to the thread index inside block and grid
- Enables shared memory and thread synchronization
- Must be called with a pythonic equivalent to `<<<blocks, threads>>>` logic

```python
@cuda.jit
def increment_a_2D_array(dev_array):
    # thread x and y index on the grid
    # alternative to cuda.threadIdx.x, cuda.blockIdx.x, cuda.blockDim.x
    x, y = cuda.grid(2)

    if x < dev_array.shape[0] and y < dev_array.shape[1]:
       an_array[x, y] += 1
```

```python
# define grid size
threadsperblock = (16, 16)
blockspergrid_x = math.ceil(array_on_device.shape[0] / threadsperblock[0])
blockspergrid_y = math.ceil(array_on_device.shape[1] / threadsperblock[1])
blockspergrid = (blockspergrid_x, blockspergrid_y)

# launch kernel
increment_a_2D_array[blockspergrid, threadsperblock](array_on_device)
```

Numba + CUDA - Memory management

- Numba can automatically transfer NumPy arrays to the device (!)
- However...
  - it can only do so conservatively by transferring device memory back to the host when a kernel finishes
  - we know that memory management is key to good GPU performance

- For these reasons, it's best to rely on manual memory handling:
  - Declaring arrays on the device memory and/or transferring data from host to device
  - Invoking kernels on the device arrays
  - Copy data back to host

```python
# host memory
a = np.arange(n)

# allocate device memory
d_a = cuda.device_array_like(a)
# and/or transfer h2d
d_a = cuda.to_device(a)

# transfer d2h
result = cuda.copy_to_host(d_a)
```

CuPy (and more...)

CuPy

CuPy is a library for GPU-accelerated computing in Python that provides a NumPy-like interface, and is meant to be as much as possible a drop-in replacement for NumPy

At its core, CuPy is implemented in C++ and CUDA, and provides functions and classes that mirror those in NumPy:

- ndarrays
- linear algebra
- part of the scipy routines
- ...

But can also provide interfaces to write simple CUDA kernels in a more Pythonic way:

- for elementwise computation
- and for reduction

And on top, it offers a way to write *plain CUDA-C kernels* that can be called directly from Python

CuPy - The three main features

1. Supported Numpy functions:

```python
# Just a random compute intensive function
def saxpy_trig(x, y, a):
    return cp.exp(a * cp.sin(x) + cp.cos(y))

res = saxpy_trig(dev_x, dev_y, 0.5)
```

Automatic number of threads definition

Bonus: Fuse operations in a single kernel

```python
@cp.fuse(kernel_name='saxpy_trig_fused')
def saxpy_trig_fused(x, y, a):
    return cp.exp(a * cp.sin(x) + cp.cos(y))

res = saxpy_trig_fused(dev_x, dev_y, 0.5)
```

2. Templated kernels for element-wise operations and reductions.

```python
saxpy_trig_elemwise = cp.ElementwiseKernel(
    'float32 x, float32 y, float32 a', # Input Types
    'float32 z',                       # Output Types
    'z = exp(a * sin(x) + cos(y))',    # Operation
    'saxpy_trig_elemwise'              # Kernel name
)

res = saxpy_trig_elemwise(dev_x, dev_y, 0.5)
```

Automatic number of threads definition

3. With "raw" CUDA code

```python
saxpy_trig_raw = cp.RawKernel(r'''
#include <cupy/complex.cuh>
extern "C" __global__
void saxpy_trig_raw(const float* x, const float* y,
                    float a, float*z, int n)
{
    int tid = blockDim.x * blockIdx.x + threadIdx.x;
    if (tid < n)
        z[tid] = exp(a * sin(x[tid]) + cos(y[tid]));

}
''', 'saxpy_trig_raw')


res = saxpy_trig_raw(args=(dev_x, dev_y, 0.5,
                           dev_out, len(dev_x)),
                     grid=((len(dev_x)+1023)//1024,),
                     block=(1024,))
```

- Manual number of threads definition
- Also supports loading pre-compiled kernels

https://indico.cern.ch/event/1334738/contributions/5814263/attachments/2833236/4950515/GPU%20Programming%20Made%20Easy%20with%20CuPy-ICSC%202024.pdf

CuPy (similarly to PyTorch, …)

CuPy aims at providing the flexibility of using CUDA in Python "as it if was really Python", instead of allowing you to write CUDA-like code in a pythonic way.

This is something that other great tools and frameworks are offering (PyTorch, Rapids, …), and is definitely worth using as much as possible (but with a pinch of salt).

CuPy for example, is great for starting up…

- ✔ Straightforward compatibility with NumPy w/ minimal effort for code migration
- ✔ Minimal prior knowledge for GPU programming
- ✔ Easier to prototype, debug, and maintain wrt CUDA-C (duh)
- ✔ Can be used as "first-order acceleration" tool for pythonic projects

CuPy (similarly to PyTorch, …)

CuPy aims at providing the flexibility of using CUDA in Python “as it if was really Python”, instead of allowing you to write CUDA-like code in a pythonic way.

This is something that other great tools and frameworks are offering (PyTorch, Rapids, …), and is definitely worth using as much as possible (but with a pinch of salt).

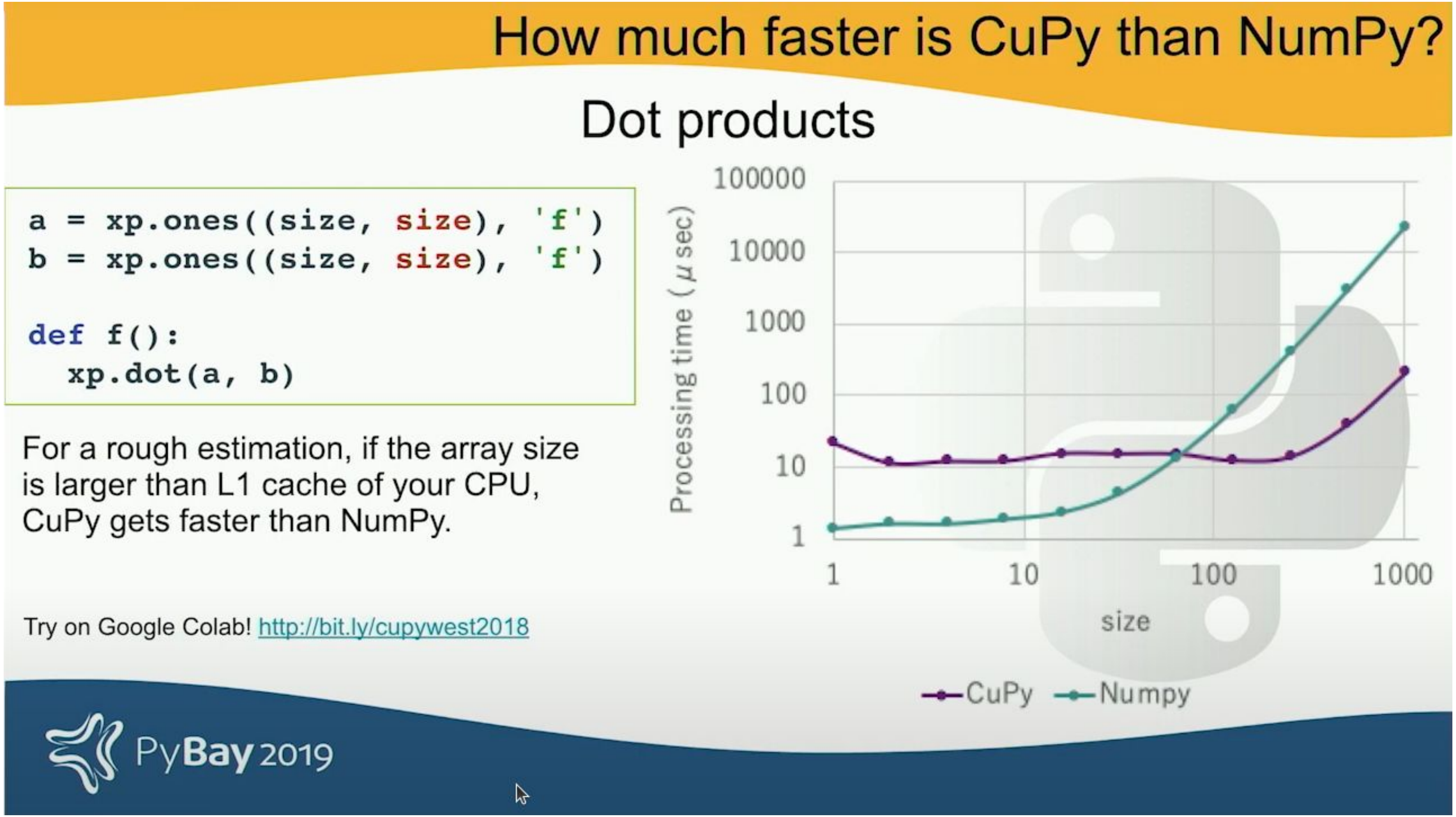
CuPy for example, is great for starting up…

- ✔ Straightforward compatibility with NumPy w/ minimal effort for code migration
- ✔ Minimal prior knowledge for GPU programming
- ✔ Easier to prototype, debug, and maintain wrt CUDA-C (duh)
- ✔ Can be used as “first-order acceleration” tool for pythonic projects

However…

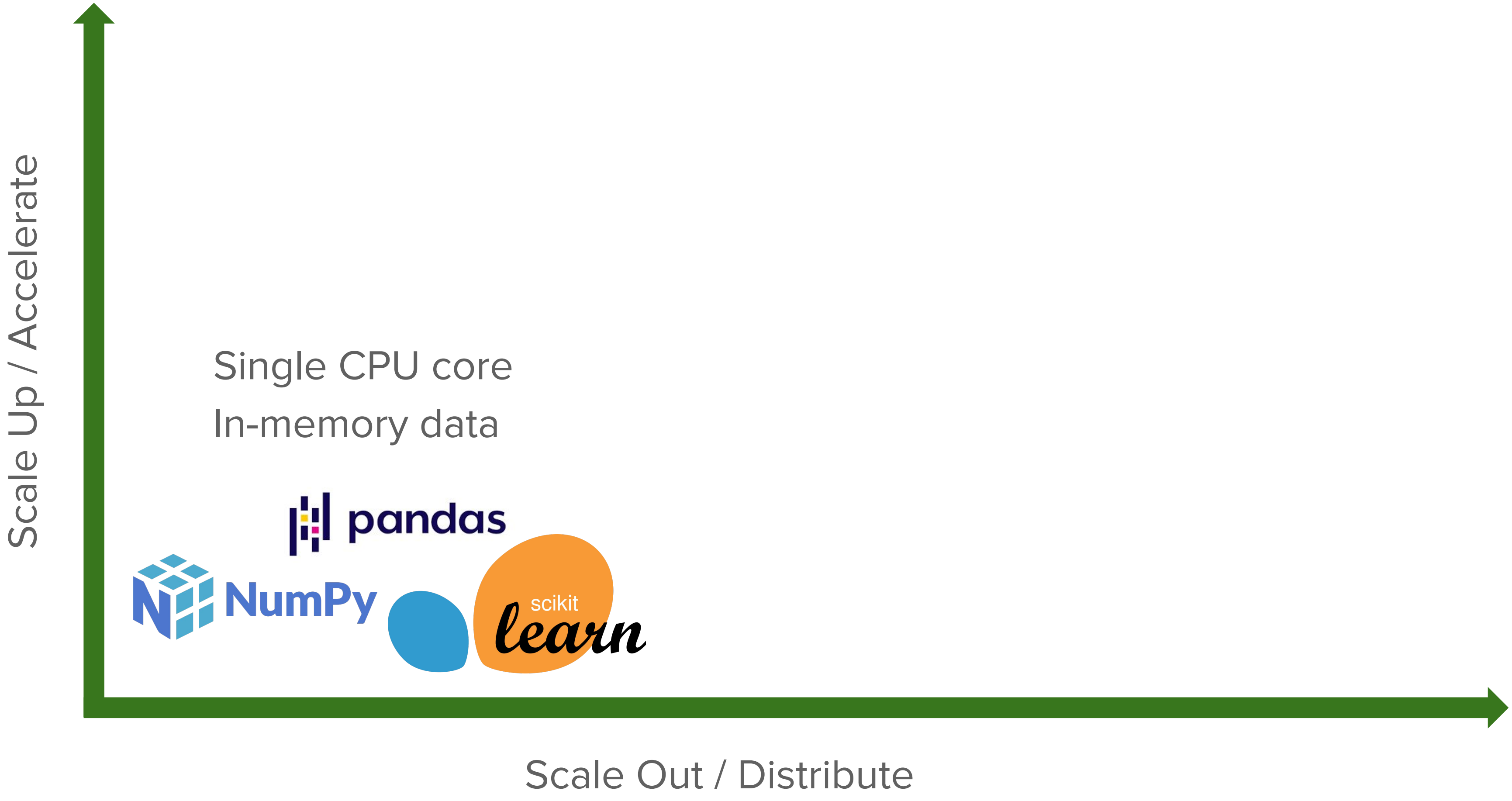
- ✗ It’s hard to reach 100% GPU utilization
- ✗ Memory allocation is often oversized or poorly handled
- ✗ Minimal optimization if possible over first-level implementation (if any)
- ✗ Missing features will still require to write custom CUDA kernels

CuPy (similarly to PyTorch, ...)



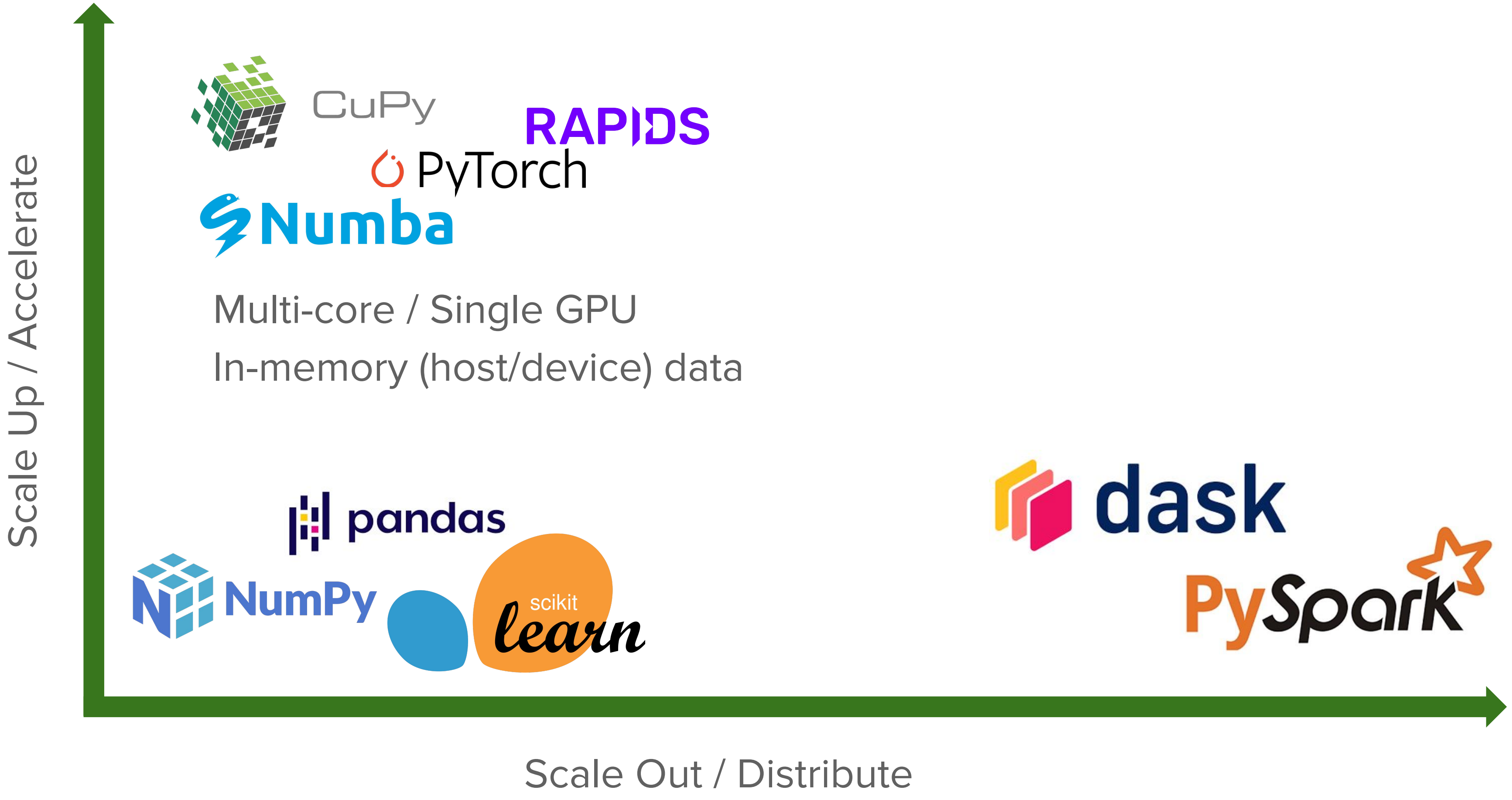
From CuPy's own presentation at PyBay 2019 [https://youtu.be/_AKDqw6Ii58]

Where to go from here in Python

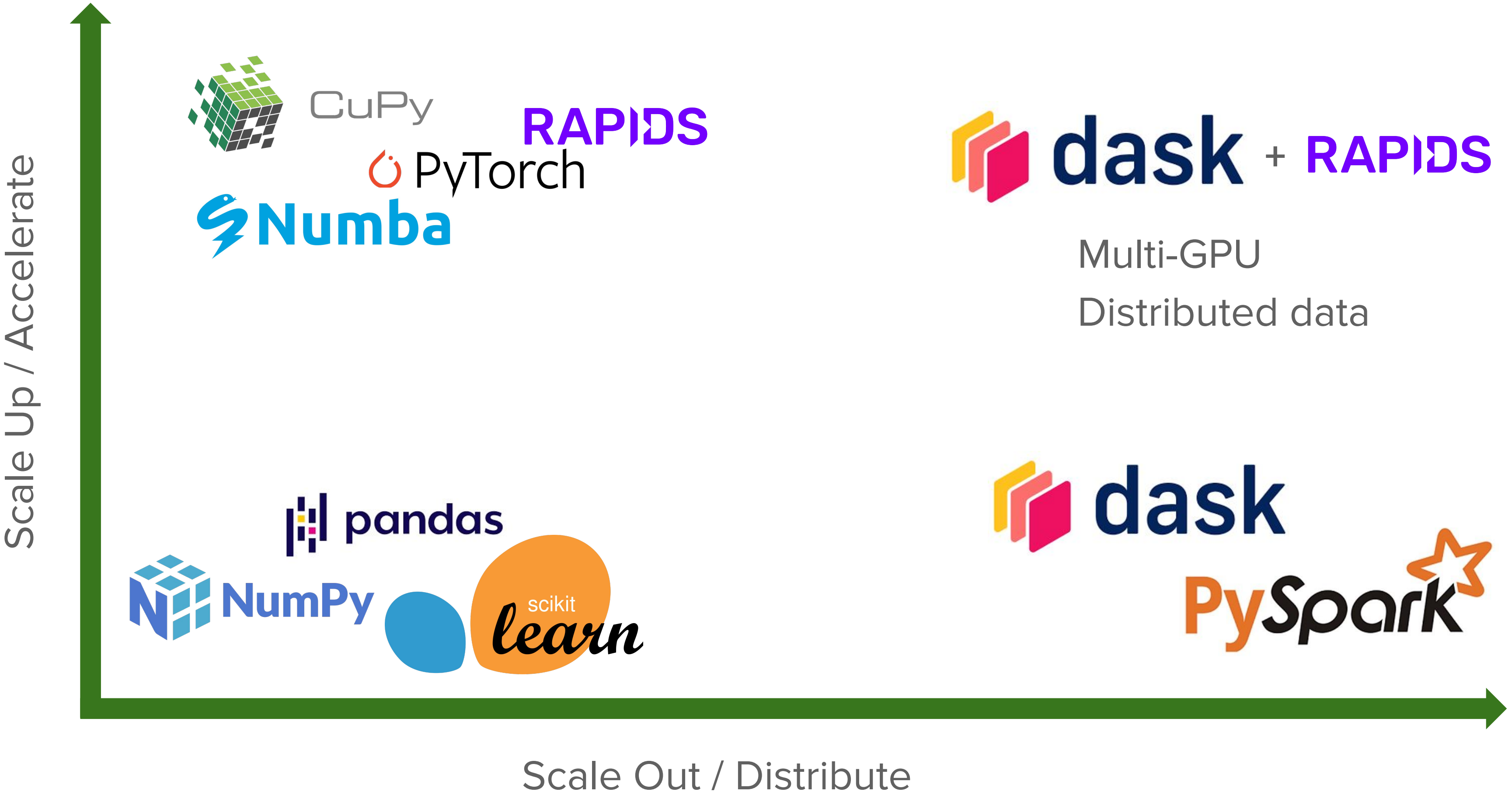


Where to go from here in Python

Where to go from here in Python



Where to go from here in Python



Where to go from here in Python → mix and match