High-Performance Python GPU Programming with CuPy

Yao-Lung “Leo” Fang (方耀龍)

Assistant Computational Scientist
Computational Science Initiative, Brookhaven National Lab, USA

Work supported in part by BNL LDRD 17-029, 20-024, DOE BES-FWP-PS001

Brookhaven National Laboratory

CuPy

Outline

- Motivation & introduction
- Simple use cases
- Just-in-time compilation
- Recent improvements (selected)
- Portability: CPU to GPU, CUDA to HIP
- Extensibility: custom kernels
- Interoperability with other Python GPU libraries
- Python Array API Standard
- Conclusion, outlook, acknowledgement

The slides will be available at https://leofang.github.io/assets/HPC_with_CuPy.pdf

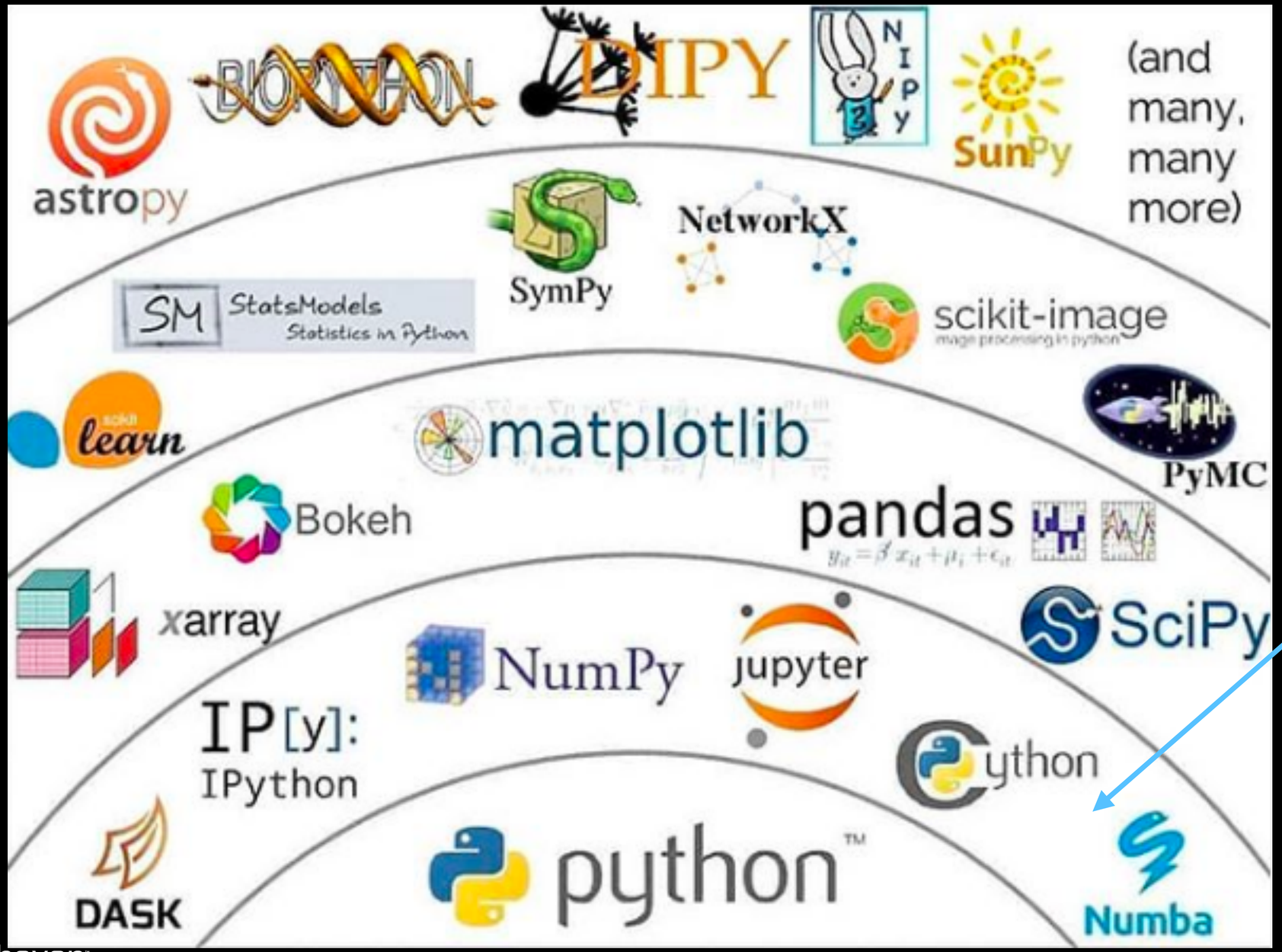
Takeaway

CuPy = NumPy + SciPy + Numba

for

NVIDIA CUDA

AMD ROCm


astropy
BIOPYTHON
IPY
NIPY
SunPy
(and many, many more)
SymPy
NetworkX
SM StatsModels Statistics in Python
scikit-image
image processing in python
learn
matplotlib
PyMC
Bokeh
pandas
xarray
SciPy
NumPy
jupyter
IP[y]: IPython
ython
DASK
python™
Numba

CuPy

~~PyCUDA~~

NumPy vs CuPy

- **Python** is the standard programming language adopted in NSLS-II, BNL, and we want to fully utilize it.
- **NumPy**, **SciPy**, **Matplotlib**, **Pandas**, **scikit-image**, **IPython/Jupyter**, etc, together form a rich and strong ecosystem for Python.
- However, Python programs can be (extremely) slow, and we want a painless solution for speedup and for easy prototyping.
- CuPy: a drop-in replacement for “running NumPy on NVIDIA / AMD* GPUs”

```
import numpy as np
a = np.arange(100, dtype=np.complex128)
b = np.exp(1j*a)
c = b[0:10]
```

```
import cupy as cp
a = cp.arange(100, dtype=cp.complex128)
b = cp.exp(1j*a)
c = b[0:10]
```

* AMD ROCm support is currently experimental

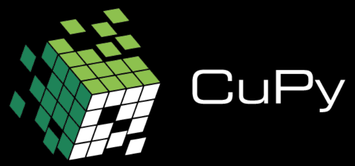
NumPy vs CuPy

CuPy: a drop-in replacement for running NumPy on NVIDIA/AMD GPUs

A silly example:


```
import numpy as np
a = np.arange(100, dtype=np.complex128)
b = np.exp(1j*a)
c = b[0:10]
d = np.fft.fft(c)
```

Initialize a (CPU) array
Perform a math operation
A “view” of array **b**
Perform FFT over a view




```
import cupy as cp
a = cp.arange(100, dtype=cp.complex128)
b = cp.exp(1j*a)
c = b[0:10]
d = cp.fft.fft(c)
```

Initialize a **GPU** array
Perform a math operation
Views of array is supported
Perform FFT over a view

Just need to know where the data reside

NumPy vs CuPy

Data movement:

```
>>> # move data to GPU
>>> arr_cp = cp.asarray(arr_np)
>>>
>>> # move data from GPU
>>> arr_np = cp.asnumpy(arr_cp)
```

(Other data copy methods exist, check out the docs!)

Just need to know where the data reside

NumPy vs CuPy

Another silly example:

```
>>> import cupy as cp
>>> x = cp.arange(6).reshape(2, 3).astype('f')
>>> x
array([[ 0.,  1.,  2.],
       [ 3.,  4.,  5.]], dtype=float32)
>>> x.sum(axis=1)
array([  3.,  12.], dtype=float32)
```

No need to learn new programming model, kernel launch, thread/block, etc

Just need to know where the data reside

More on CuPy

- https://cupy.dev/
- Latest release: v9.1.0 (release cycle ~6 months)
- `pip install cupy-cudaXY`
- `conda install -c conda-forge cupy cudatoolkit=X.Y`
- Open source development led by PFN (a Japanese company behind Chainer)
- Strongly coupled with many open-source projects (Dask, cuDF, cuML, cuCIM, etc)
- Very extensive test coverage + continuous integration
- NEP-29 compliant (Python community-wide version-support guideline)

More on CuPy

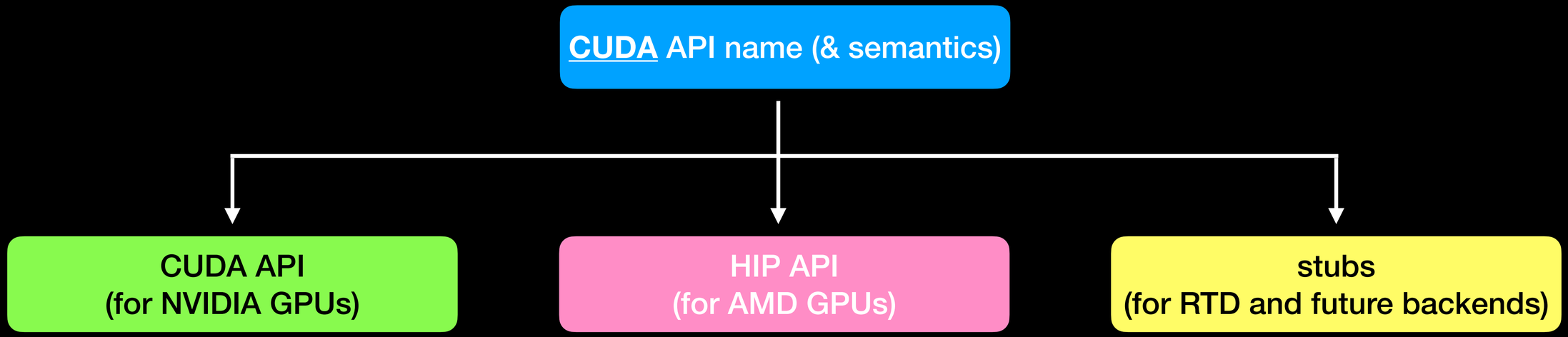
- Custom memory management (device/pinned/managed/async)
- Close to full coverage of NumPy API (ndarray, dtype, ufunc, linalg, fft, random, …)
- Partial coverage of SciPy API (ndimage, signal, sparse, fft, special, linalg, …)
- Extensive coverage of CUDA API (driver, runtime, nvrtc, math libraries, texture, …)
  - ➡ Use Cython to wrap CUDA/HIP C APIs
  - ➡ High-level Python CUDA objects (devices, stream, events, …)
  - ➡ CuPy as a proxy for low-level GPU programming in Python
- Support for AMD ROCm/HIP software stack is largely complete

More on CuPy

	NVIDIA CUDA GPUs	AMD ROCm GPUs
Basic Array Operations (dtype, creation, indexing…)	custom	
(Dense) Linear Algebra & Tensor Operation	cuBLAS / cuSOLVER / cuTENSOR	hipBLAS (rocBLAS) / rocSOLVER*
(Sparse) Matrix Operatios & Linear Algebra	cuSPARSE / cuSPARSELt / cuSOLVER	(planned)
Fast Fourier Transform	cuFFT	hipFFT (rocFFT)
Random Numbers	cuRAND / custom	hipRAND (rocRAND)
Sort / Search	Thrust	rocThrust
Reduction, Scan, Historgram	CUB / custom	hipCUB (rocPRIM) / custom
Multi-GPU communication	NCCL	RCCL
DNN utilities	cuDNN (maintenance mode)	(won't support)

CuPy internal: backends

- Namespaces:
  - **`cupy`**: Covers NumPy APIs as closely as possible
    - `cupy.cuda`: One notable exception (might be renamed/removed in the future)
  - **`cupyx`**: Covers anything not in NumPy; less stringent on compatibility
    - `cupyx.scipy`: Covers SciPy APIs (fft/fftpack, linalg, sparse, ndimage, etc)
    - `cupyx.jit`: CuPy's new JIT compiler (to be discussed later)
  - **`cupy_backends`**: Low-level CUDA & ROCm/HIP APIs, RTD stubs, etc (private)



CuPy internal: backends

Cython wrapper

```
cdef extern from "cupy_backend.h":
    int cudaGetDevice(int* device)

cpdef int getDevice() except -1:
    cdef int device, status
    with nogil:
        status = cudaGetDevice(&device)
    check_status(status)
    return device
```

(this is just a sketch)

cupy_backend.h

```
#if CUPY_USE_CUDA

#include <cuda_runtime.h>

#elif CUPY_USE_HIP

#include <hip/hip_runtime_api.h>
cudaError_t cudaGetDevice(int* deviceId) {
    return hipGetDevice(deviceId);
}

#else  // stubs for Read the Docs

cudaError_t cudaGetDevice(...) {
    return CUDA_SUCCESS;
}

#endif
```

CuPy internal: Just-in-time compilation

Two aspects of just-in-time (JIT) compilation in my talk:

1. JIT for compiling CUDA C++ kernels on the fly
2. JIT for transpiling Python functions to CUDA C++ kernels

Why No.1 is needed? *Combinatorial* explosion in the number of kernels to compile!*

Ex:

```
cupy.sum(a, axis=None, dtype=None, out=None, keepdims=False)
```

14 (input dtypes) × 14 (output dtypes) × 25 (MAX_NDIM) × 12 (sm_XX) × 7 (CUDA versions) × 2 (Windows/Linux) × 4 (Python versions) × (…)

Now do the pre- (ahead-of-time) compilation for all supported functions…

CuPy internal: Just-in-time compilation

Two aspects of just-in-time (JIT) compilation in my talk:

1. JIT for compiling CUDA C++ kernels on the fly
2. JIT for transpiling Python functions to CUDA C++ kernels

Why No.1 is needed? *Combinatorial* explosion in the number of kernels to compile!*

➡ The only sensible solution:

Generate and compile CUDA codes at runtime *only if needed*

*See, e.g., GTC 2019 talk by Jake Hemstad (S91043)

CuPy internal: Just-in-time compilation

Next question: How to generate CUDA code strings? ➡ Kernel templates, JIT, user-provided kernels, …

Ex: `b = cupy.sin(a)`

```
typedef double in0_type;
typedef double out0_type;
```

```
module_code = string.Template('''
${typedef_preamble}
${preamble}
extern "C" __global__ void ${name}(${params}) {
  ${loop_prep};
  CUPY_FOR(i, _ind.size()) {
    _ind.set(i);
    ${operation};
  }
  ${after_loop};
}
''').substitute(typedef_preamble=..., ...)
```

```
cupy_sin__float64_float64
```

```
const CArray<double, 1, 1, 1> _raw_in0,
CArray<double, 1, 1, 1> _raw_out0,
CIndexer<1> _ind
```

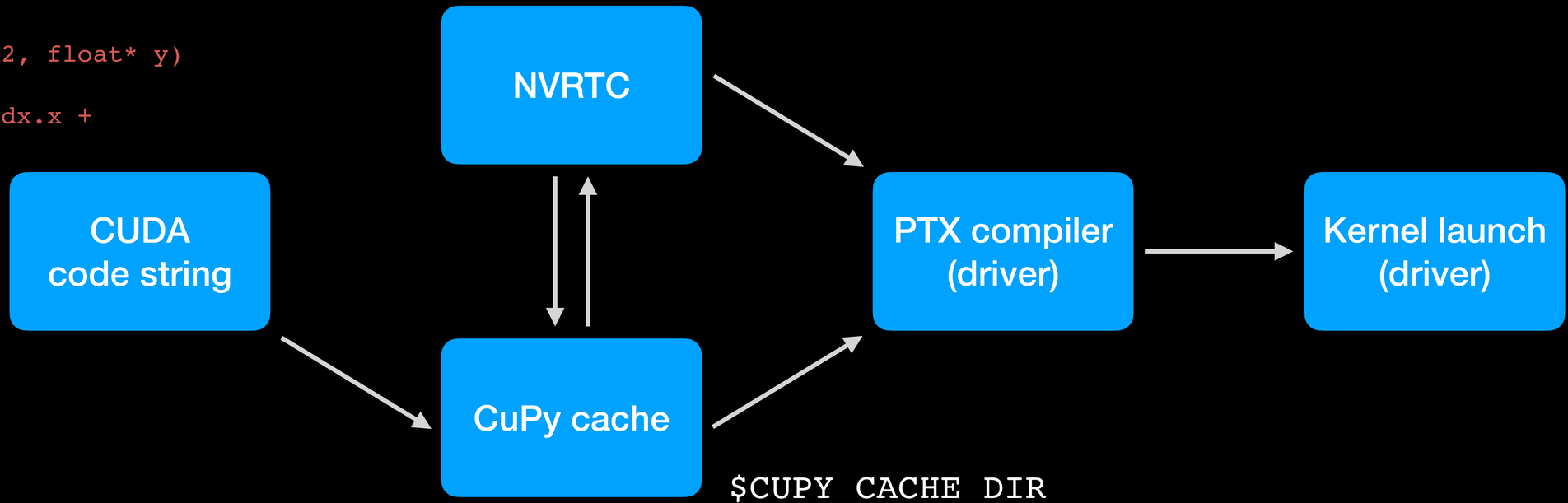
```
const in0_type in0(_raw_in0[_ind.get()]);
out0_type &out0 = _raw_out0[_ind.get()];
out0 = sin(in0);
```

Ideas: reuse templates as much as possible, cache generated kernels & binaries, etc

CuPy internal: Just-in-time compilation

- Code compiled by NVRTC (default) ➡ Faster than a subprocess call to NVCC
- Generated PTX* is cached in memory & on disk for reuse
- Use CUDA driver API to load & launch
- Dynamical parallelism / cooperative group are supported

```
some_cuda_code = r'''
extern "C" {
  __global__ void my_add(
  const float* x1, const float* x2, float* y)
{
    int tid = blockDim.x * blockIdx.x +
threadIdx.x;
    y[tid] = x1[tid] + x2[tid];
  }
}'''
```



*Also possible to generate .cubin since CUDA 11.1

`$CUPY_CACHE_DIR`
(default: `$HOME/.cupy/kerne_cache`**)**

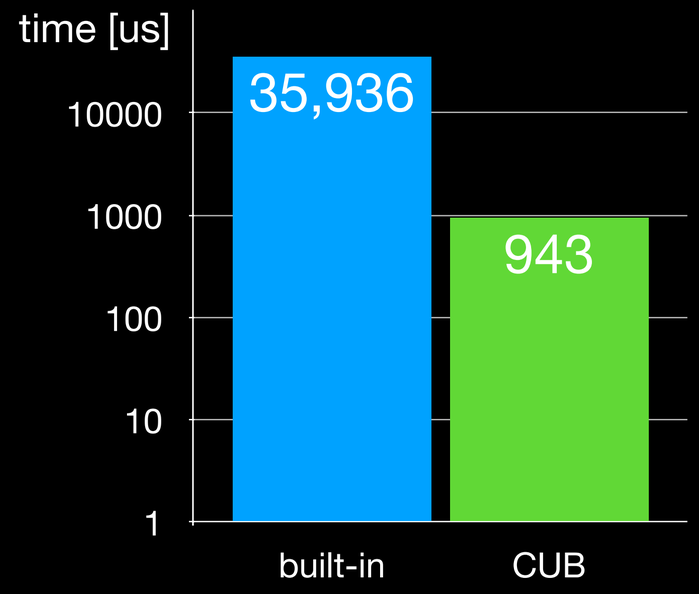
Improvements in CuPy relevant to end users

My philosophy:

You as a user focus on writing your code,
and let us worry about performance for you

Improvements in CuPy relevant to end users

- Typical ingredients in a ptychographical image reconstruction:
  - Reduction (ex: sum)
  - FFT
  - Element-wise operations



v6

```
import cupy as cp
a = cp.random.random((256, 512, 512))
b = cp.sum(cp.abs(a)**2)
```

v7

```
# 1. build CuPy from source with CUB_PATH
# 2. export CUB_DISABLED=0
import cupy as cp
cp.cuda.cub_enabled = True
a = cp.random.random((256, 512, 512))
b = cp.sum(cp.abs(a)**2)
```

v8

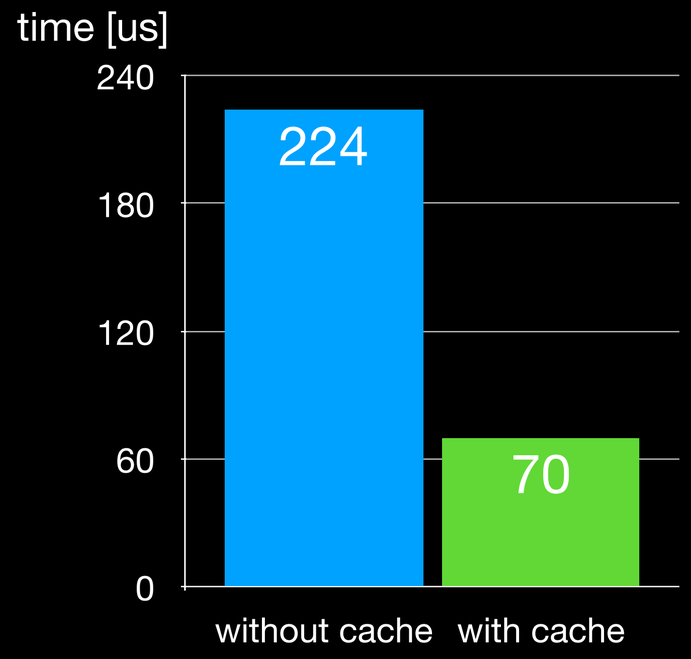
```
# export CUPY_ACCELERATORS=cub,cutensor
import cupy as cp
a = cp.random.random((256, 512, 512))
b = cp.sum(cp.abs(a)**2)
```

Brookhaven National Laboratory

CuPy

Improvements in CuPy relevant to end users

- Typical ingredients in a ptychographical image reconstruction:
  - Reduction (ex: sum)
  - FFT
  - Element-wise operations



plan is cached →

```
import cupy as cp
a = cp.random.random((256, 512, 512))
a = a.astype(cp.complex128)
b = cp.fft.fftn(a, axes=(1,2))
```

v6

```
import cupy as cp
import cupyx.scipy.fftpack as cufft
a = cp.random.random((256, 512, 512))
a = a.astype(cp.complex128)
plan = cufft.get_fft_plan(a, axes=(1,2))
b = cufft.fftn(a, axes=(1,2), plan=plan)
```

v7

```
import cupy as cp
import cupyx.scipy.fftpack as cufft
a = cp.random.random((256, 512, 512))
a = a.astype(cp.complex128)
plan = cufft.get_fft_plan(a, axes=(1,2))
with plan:
    b = cp.fft.fftn(a, axes=(1,2))
```

```
import cupy as cp
a = cp.random.random((256, 512, 512))
a = a.astype(cp.complex128)
b = cp.fft.fftn(a, axes=(1,2))
```

v8

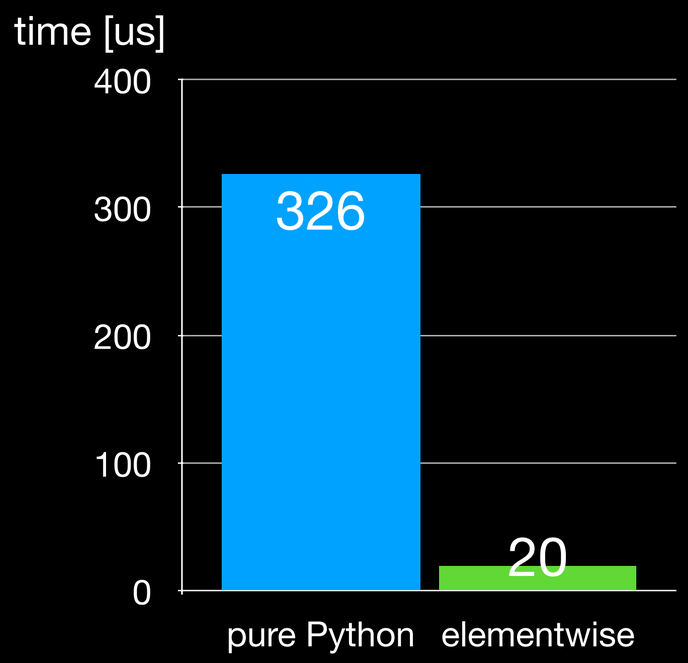
Brookhaven National Laboratory

CuPy

Extending CuPy with CUDA

Benefit (beauty?) of CuPy: users don't need to worry about grid, block, thread, …

- `cupy.ElementwiseKernel` support broadcasting & type inferring




```
>>> x = cp.arange(6, dtype='f').reshape(2, 3)
>>> y = cp.arange(3, dtype='f')
>>> kernel = cp.ElementwiseKernel(
...     'float32 x, float32 y',
...     'float32 z',
...     '''if (x - 2 > y) { z = x * y; }
...     else { z = x + y; }''',
...     'my_kernel')
>>> kernel(x, y)
array([[ 0.,  2.,  4.],
       [ 0.,  4.,  10.]], dtype=float32)
```

Brookhaven National Laboratory

CuPy

Extending CuPy with CUDA

Benefit (beauty?) of CuPy: ~~users don't need to worry about grid, block, thread, …~~

sometimes we'd like to make our lives harder

- `cupy.ElementwiseKernel`
- `cupy.RawKernel`

a legit CUDA kernel

```
>>> add_kernel = cp.RawKernel(r'''
... extern "C" __global__
... void my_add(const float* x1, const float* x2, float* y) {
...     int tid = blockDim.x * blockIdx.x + threadIdx.x;
...     y[tid] = x1[tid] + x2[tid];
... }''', 'my_add')
>>> x1 = cupy.arange(25, dtype=cupy.float32).reshape(5, 5)
>>> x2 = cupy.arange(25, dtype=cupy.float32).reshape(5, 5)
>>> y = cupy.zeros((5, 5), dtype=cupy.float32)
>>> add_kernel((5,), (5,), (x1, x2, y))  # grid, block and arguments
>>> y
array([[ 0.,  2.,  4.,  6.,  8.],
       [10., 12., 14., 16., 18.],
       [20., 22., 24., 26., 28.],
       [30., 32., 34., 36., 38.],
       [40., 42., 44., 46., 48.]], dtype=float32)
```

Brookhaven National Laboratory

CuPy

Extending CuPy with CUDA

Benefit (beauty?) of CuPy: ~~users don't need to worry about grid, block, thread, …~~

sometimes we'd like to make our lives harder

- `cupy.ElementwiseKernel`
- `cupy.RawKernel`
- `cupy.RawModule` (suitable for migrating existing CUDA codebase)

`some_cuda_code.cu`:

```
extern "C" {
__global__ void my_add(
const float* x1, const float* x2,
float* y) {
    int tid = blockDim.x *
blockIdx.x + threadIdx.x;
    y[tid] = x1[tid] + x2[tid];
  }
}
```

```
$ nvcc -arch=sm_XX -cubin -o cupy_mod.cubin \
some_cuda_code.cu
```

```
>>> # load the CUDA binary
... mod = cp.RawModule(path="/path/to/cupy_mod.cubin")
>>> # fetch the kernel as a RawKernel object
... add_kernel = mod.get_function("my_add")
>>> x1 = cp.arange(25, dtype=cp.float32).reshape(5, 5)
>>> x2 = cp.arange(25, dtype=cp.float32).reshape(5, 5)
>>> y = cp.zeros((5, 5), dtype=cp.float32)
>>> add_kernel((5,), (5,), (x1, x2, y))
```

Extending CuPy with CUDA

Benefit (beauty?) of CuPy: ~~users don't need to worry about grid, block, thread, …~~

sometimes we'd like to make our lives harder

- `cupy.ElementwiseKernel`
- `cupy.RawKernel`
- `cupy.RawModule` (suitable for migrating existing CUDA codebase)

```
some_cuda_code = r'''
extern "C" {
__global__ void my_add(
const float* x1, const float* x2,
float* y) {
    int tid = blockDim.x *
blockIdx.x + threadIdx.x;
    y[tid] = x1[tid] + x2[tid];
  }
}
'''
```

```
>>> # pass the code string
... mod = cp.RawModule(code=some_cuda_code)
>>> # fetch the kernel as a RawKernel object
... add_kernel = mod.get_function("my_add")
>>> x1 = cp.arange(25, dtype=cp.float32).reshape(5, 5)
>>> x2 = cp.arange(25, dtype=cp.float32).reshape(5, 5)
>>> y = cp.zeros((5, 5), dtype=cp.float32)
>>> add_kernel((5,), (5,), (x1, x2, y))
```

Support C++ template kernels, C++ std headers*, texture reference, global symbols, …

Extending CuPy with CUDA

Benefit (beauty?) of CuPy: ~~users don't need to worry about grid, block, thread, …~~

sometimes we'd like to make our lives harder

- `cupy.ElementwiseKernel`
- `cupy.RawKernel`
- `cupy.RawModule` (suitable for migrating existing CUDA codebase)
- `cupy.ReductionKernel` (CUB enabled)
- `@cupy.fuse()`
- `cupy.vectorize()` **(new in v9)**
- `@cupyx.jit.rawkernel()` **(new in v9)**

See the documentation for more info:
https://docs.cupy.dev/en/stable/user_guide/kernel.html

Extending CuPy with ~~CUDA~~ Python

Benefit (beauty?) of CuPy: users don't need to worry about grid, block, thread, …

NEW in CuPy v9!

```
import numpy as np

def func_while(x):
    y = 0
    while x > 0:
        y += x
        x -= 1
    return y

a = np.arange(100, dtype=np.int32)
f = np.vectorize(func_while)
b = f(a)
```

Loop over all elements
and apply the Python function

```
import cupy as cp

def func_while(x):
    y = 0
    while x > 0:
        y += x
        x -= 1
    return y

a = cp.arange(100, dtype=cp.int32)
f = cp.vectorize(func_while)
b = f(a)
```

Convert the Python function
to a CUDA C++ kernel

➡ **Just-in-time (JIT) compiler!**

Brookhaven National Laboratory

CuPy

Extending CuPy with ~~CUDA~~ Python

Benefit (beauty?) of CuPy: ~~users don't need to worry about grid, block, thread, …~~

sometimes we'd like to make our lives harder

NEW in CuPy v9!

```
from cupyx import jit

@jit.rawkernel()
def add_kernel(x1, x2, y, N):
    tid = jit.blockDim.x * jit.blockIdx.x + jit.threadIdx.x
    if tid < N:
        y[tid] = x1[tid] + x2[tid]

x1 = cp.random.random(25)
x2 = cp.random.random(25)
y = cp.empty_like(x1)
add_kernel((5,), (5,), (x1, x2, y, 25))
```

Pure Python syntax!

Python to CUDA transpilation via Python AST (no LLVM needed)

See the documentation for more info:
https://docs.cupy.dev/en/stable/user_guide/kernel.html

Brookhaven National Laboratory

CuPy

Portability: CUDA & HIP

My philosophy:

You as a user focus on writing your code,
and let us worry about performance for you

You as a user focus on writing your code,
and let us worry about portability for you

Portability: CUDA & HIP

Write your code once, run on both NVIDIA and AMD GPUs!

some_cuda_code.cu

```
// can also write C++ template code!
extern "C" __global__ void my_add(const float* x1,
                                  const float* x2,
                                  float* y,
                                  int N) {
    int tid = blockDim.x * blockIdx.x + threadIdx.x;
    if (tid < N) y[tid] = x1[tid] + x2[tid];
}
```

```
>>> # load the CUDA source code
>>> with open("/path/to/some_cuda_code.cu") as f:
...     code = f.read()
>>> # create a CUDA module
>>> mod = cp.RawModule(code=code)
>>> # fetch the kernel as a Python function object
... add_kernel = mod.get_function("my_add")
>>> x1 = cp.arange(25, dtype=cp.float32).reshape(5, 5)
>>> x2 = cp.arange(25, dtype=cp.float32).reshape(5, 5)
>>> y = cp.zeros((5, 5), dtype=cp.float32)
>>> add_kernel((5,), (5,), (x1, x2, y, 25))
```

No need to change anything to run on AMD GPUs!

Brookhaven National Laboratory

CuPy

Portability: CUDA & HIP

Write your code once, run on both NVIDIA and AMD GPUs!

```
__global__ void cupy_cub_sum_pass1(const void* _raw_in0, void* _raw_out0,
                                   const int _segment_size, const int _array_size) {
  ...

  // each block handles the reduction of 1 segment
  size_t segment_idx = blockIdx.x * _segment_size;
  ...
  int _seg_size = _segment_size;

  #if defined FIRST_PASS
  // for two-pass reduction only: "last segment" is special
  if (_array_size > 0) {
      if (_array_size - segment_idx <= _segment_size) {
          _seg_size = _array_size - segment_idx;
      }
      #ifdef __HIP_DEVICE_COMPILE__
      // We don't understand HIP...
      __syncthreads();
      #endif
  }
  #endif

  // loop over tiles within 1 segment
  ...
}
```

...or change something for AMD GPUs...

Brookhaven National Laboratory

CuPy

Portability: CUDA & HIP

Numba

Write your code once, run on both NVIDIA and AMD GPUs!

```
code = r'''
extern "C" __global__ void my_add(const float* x1,
                                  const float* x2,
                                  float* y,
                                  int N) {
    int tid = blockDim.x * blockIdx.x + threadIdx.x;
    if (tid < N) y[tid] = x1[tid] + x2[tid];
}'''

mod = cp.RawModule(code=code)
add_kernel = mod.get_function("my_add")
x1 = cp.random.random(25)
x2 = cp.random.random(25)
y = cp.empty_like(x1)
add_kernel((5,), (5,), (x1, x2, y, 25))
```

CuPy's machinery is preferred for both performance and portability

```
from numba import cuda
import cupy as cp

@cuda.jit()
def my_add(x1, x2, y):
    tid = cuda.grid(1)
    if tid < y.size:
        y[tid] = x1[tid] + x2[tid]

x1 = cp.random.random(25)
x2 = cp.random.random(25)
y = cp.empty_like(x1)
my_add[5, 5](x1, x2, y)
```

Numba's problem:

1. **Worse performance**
2. **@roc.jit() needs rewriting**
3. **@roc.jit() doesn't work…**

Brookhaven National Laboratory

CuPy

Portability: CUDA & HIP

NEW in CuPy v9!

Write your code once, run on both NVIDIA and AMD GPUs!

```
from cupyx import jit

@jit.rawkernel()
def add_kernel(x1, x2, y, N):
    tid = jit.blockDim.x * jit.blockIdx.x \
          + jit.threadIdx.x
    if tid < N:
        y[tid] = x1[tid] + x2[tid]


x1 = cp.random.random(25)
x2 = cp.random.random(25)
y = cp.empty_like(x1)
add_kernel((5,), (5,), (x1, x2, y, 25))
add_kernel[5, 5](x1, x2, y, 25)  # also works!
```

Faster compilation and performance!

```
from numba import cuda
import cupy as cp

@cuda.jit()
def my_add(x1, x2, y):
    tid = cuda.grid(1)
    if tid < y.size:
        y[tid] = x1[tid] + x2[tid]

x1 = cp.random.random(25)
x2 = cp.random.random(25)
y = cp.empty_like(x1)
my_add[5, 5](x1, x2, y)
```

Numba's problem:

1. **Worse performance**
2. **@roc.jit() needs rewriting**
3. **@roc.jit() doesn't work…**

Brookhaven National Laboratory

CuPy

CuPy + **x** : interoperability

- To build an ecosystem like those based on NumPy/SciPy, it is important that different Python GPU libraries can interoperate.
- I focus on passing CuPy arrays to other Python libraries in this talk.
- Zero-copy is a must, otherwise we lose performance!

CuPy + NumPy

CuPy supports **CPU/GPU agnostic codes**:
input a NumPy or CuPy array, and let the program decide which version to call

```
>>> def softplus(x):
...     # Stable implementation of log(1 + exp(x))
...     xp = cp.get_array_module(x)
...     return xp.maximum(0, x) + xp.log1p(xp.exp(-xp.abs(x)))
>>>
>>> x_np = np.random.random(10)
>>> out_np = softplus(x_np)
>>>
>>> x_cp = cp.asarray(x_np)
>>> out_cp = softplus(x_cp)
```

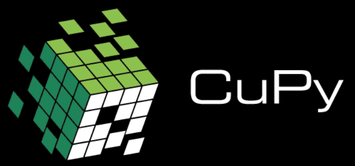
xp = numpy or cupy

create a NumPy array on CPU

computation is done by CPU

move data to GPU

computation is done by GPU



CuPy + NumPy

Leo's way of writing **CPU/GPU agnostic codes**:

```
>>> use_np_or_cp = 'np'  set the value according to input at start time or runtime
>>> if use_np_or_cp == 'np':
...     import numpy as xp
... else:
...     import cupy as xp
>>>
>>> def softplus(x):
...     return xp.maximum(0, x) + xp.log1p(xp.exp(-xp.abs(x)))
>>>
>>> x = xp.random.random(10)
>>> out = softplus(x)
```

CuPy + NumPy

Write **CPU/GPU agnostic code** based on

NEP-18: `__array_function__` dispatch protocol*

```
>>> import numpy as np
>>> import cupy as cp
>>>
>>> x = cp.arange(10).reshape(2,5)
>>> print(np.argmax(x))
array(9)
>>> print(np.sum(x, axis=0))
array([5, 7, 9, 11, 13])
```

x is a CuPy array

NumPy will dispatch to `cupy.argmax()`

NumPy will dispatch to `cupy.sum()`

*http://www.numpy.org/neps/nep-0018-array-function-protocol.html

CuPy + Dask

Write **CPU/GPU agnostic code** based on

NEP-18: `__array_function__` dispatch protocol*

```
>>> import dask, dask.array
>>> import numpy
>>> import cupy
>>>
>>> x = cupy.random.random((1000000, 1000))
>>> dx = dask.array.from_array(x, chunks=(10000, 1000),
...      asarray=False)
>>> u, s, v = numpy.linalg.svd(dx)
>>> u, s, v = dask.compute(u, s, v)
```

Brookhaven National Laboratory

*http://www.numpy.org/neps/nep-0018-array-function-protocol.html

CuPy

CuPy + NCCL

- NCCL is a library provided by NVIDIA for collective operations on *multiple GPUs*: reduce, broadcast, allgather, etc.
- Typically works with MPI or `multiprocessing`
- CuPy provides a Python wrapper of NCCL

```
>>> import cupy as cp
>>> from mpi4py import MPI
>>>
>>> comm = MPI.COMM_WORLD
>>> rank = comm.Get_rank()
>>> size = comm.Get_size()
>>>
>>> if rank == 0:
>>>     nccl_id = cp.cuda.nccl.get_unique_id()
>>> else:
>>>     nccl_id = None
>>> nccl_id = comm.bcast(nccl_id)
>>>
>>> nccl_comm = cp.cuda.nccl.NcclCommunicator(size, nccl_id, rank)
>>> arr_in = cp.random.random(10, dtype=cp.float64)
>>> arr_out = cp.zeros(10, dtype=cp.float64)
>>> nccl_comm.allReduce(arr_in.data.ptr, arr_out.data.ptr,
... 10, cp.cuda.nccl.NCCL_FLOAT64, cp.cuda.nccl.NCCL_SUM, 0)
```

`__cuda_array_interface__`

https://numba.readthedocs.io/en/stable/cuda/cuda_array_interface.html

- Proposed to solve the interoperating issue
- Following the path of NumPy's `__array_interface__`
- Adapted by many famous libraries:

Numba, CuPy, PyTorch, mpi4py, PyArrow, cuDF, cuML, …

- Each library provider implements a `__cuda_array_interface__` attribute returning a `dict`:

```
{shape: (array shape)          ,
 typestr: (NumPy type string),
 data: (device_ptr, 0),
 version: 3,
 ...}
```

CuPy + PyTorch

PyTorch

- PyTorch is a popular general purpose ML framework that provides high-performance, differentiable tensor operations

```
>>> import cupy as cp
>>> import torch
>>>
>>> # convert a torch tensor to a cupy array
>>> a = torch.rand((4, 4), device='cuda')
>>> b = cp.asarray(a)
>>> b *= b
>>>
>>> # convert a cupy array to a torch tensor
>>> a = cp.arange(10)
>>> b = torch.as_tensor(a, device='cuda')
>>> b += 3
>>> b
tensor([ 3,  4,  5,  6,  7,  8,  9, 10, 11, 12], device='cuda:0')
```

Brookhaven National Laboratory

CuPy

CuPy + Numba

```
>>> import math
>>>
>>> import cupy as cp
>>> from numba import cuda
>>>
>>> @cuda.jit
>>> def run_exp(arr_in):
>>>     x = cuda.grid(1)
>>>
>>>     if x < arr_in.shape[0]:
>>>         arr_in[x] = math.exp(arr_in[x])
>>>
>>> arr1 = cp.random.random(100)
>>> arr2 = arr1.copy()
>>> run_exp[10, 10](arr1)
>>> assert (arr1==cp.exp(arr2)).all()
```

- Numba provides a JIT compiler for NVIDIA GPUs
- Write Numba kernels in Python-like syntax
- Auto type inference and kernel cache
- Accept CuPy arrays via `__cuda_array_interface__`

CuPy + mpi4py

- MPI for Python (mpi4py) is a Python binding to the MPI implementation.
- Written in Cython (which then gets translated to C) for minimizing Python overhead.
- Accept CuPy arrays via `__cuda_array_interface__` (the MPI library needs to be CUDA-aware)

```
>>> import cupy
>>> from mpi4py import MPI
>>>
>>> comm = MPI.COMM_WORLD
>>> size = comm.Get_size()
>>>
>>> # Allreduce
>>> sendbuf = cupy.arange(10, dtype='i')
>>> recvbuf = cupy.empty_like(sendbuf)
>>> comm.Allreduce(sendbuf, recvbuf)
>>> assert cupy.allclose(recvbuf, sendbuf*size)
```

CuPy + mpi4py

MPI (Message Passing Interface) is a *de facto* standard for inter-process comm

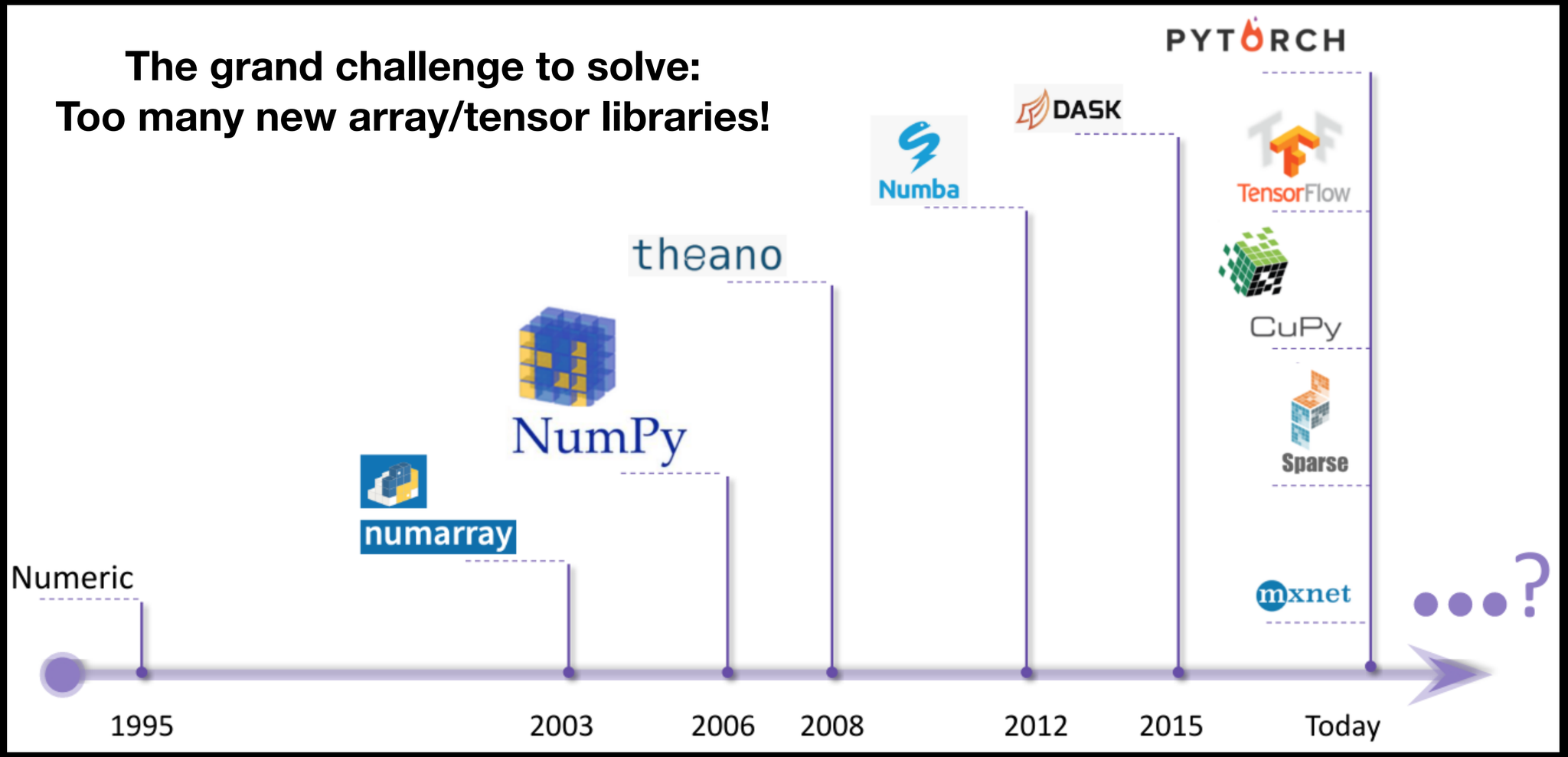
```
from mpi4py import MPI
import numpy as np

comm = MPI.COMM_WORLD
rank = comm.Get_rank()

# communicate underlying buffers, faster!
if rank == 0:
    data = np.arange(10, dtype=np.float64)
    comm.Send(data, dest=1, tag=11)
elif rank == 1:
    data = np.empty(10, dtype=np.float64)
    comm.Recv(data, source=0, tag=11)
    print(data)
```

```
from mpi4py import MPI          # CUDA-aware
import cupy as cp

comm = MPI.COMM_WORLD
rank = comm.Get_rank()

# communicate underlying buffers, faster!
if rank == 0:
    data = cp.arange(10, dtype=np.float64)
    comm.Send(data, dest=1, tag=11)
elif rank == 1:
    data = cp.empty(10, dtype=np.float64)
    comm.Recv(data, source=0, tag=11)
    print(data)
```

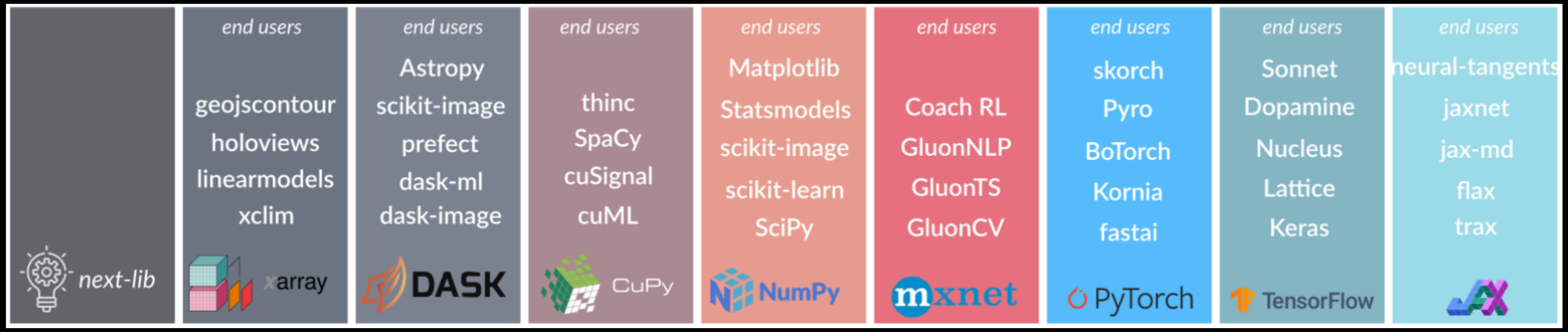
```
mpirun -n 2 python test_mpi.py
```

https://github.com/mpi4py/mpi4py

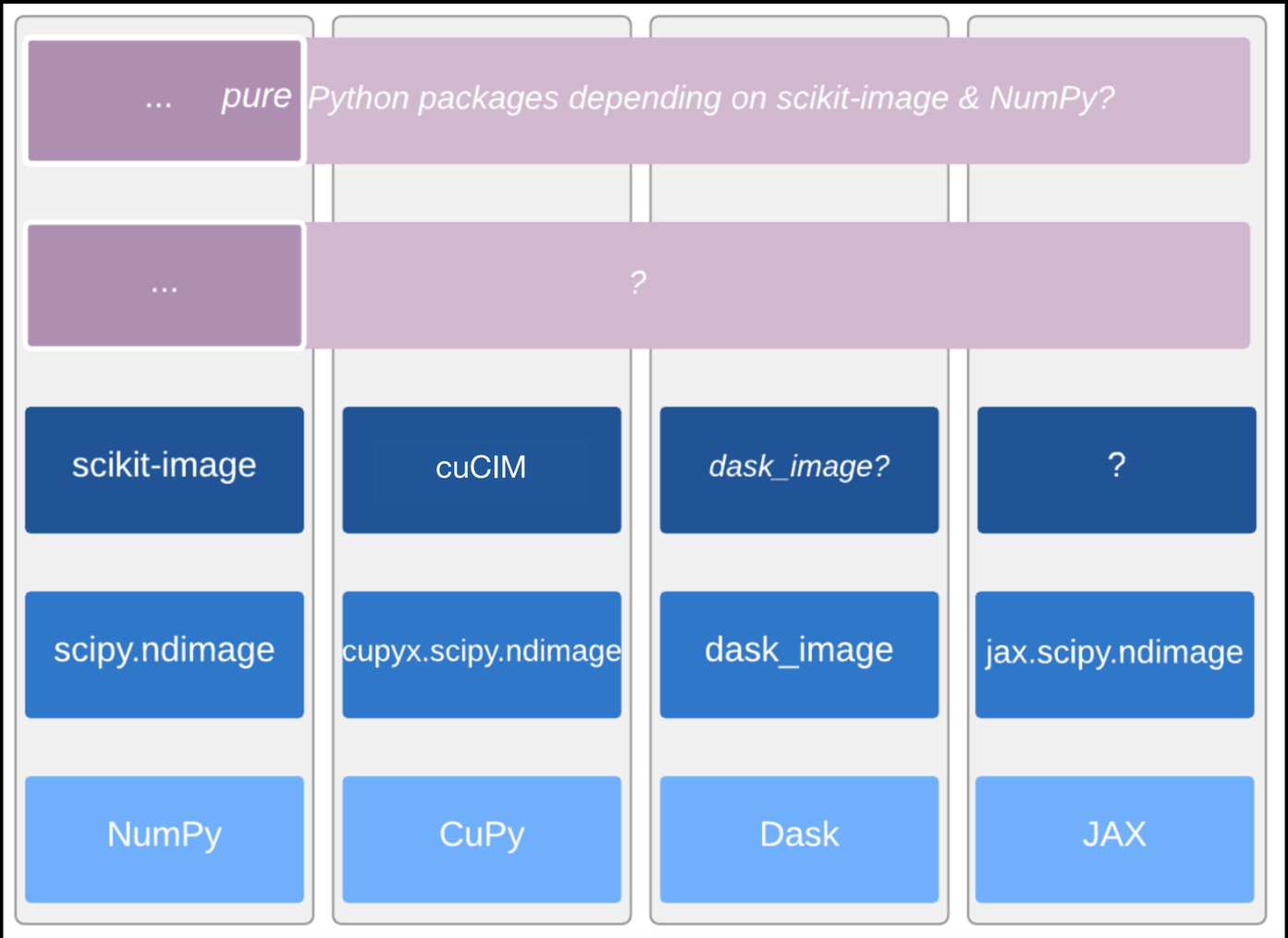
Python Array API standard



Python Array API standard



Python Array API standard



Following our NumPy vs CuPy discussion, it is natural to expect: Write my code once, switch to different libraries, and see how performance improves.

Is this possible nowadays…?

Python Array API standard

…Likely not, because each library's function signature (and potentially the semantics) is slightly different…

https://github.com/data-apis/array-api-comparison/blob/main/signatures/creation/arange.md

NumPy

```
numpy.arange([start, ]stop, [step, ]dtype=None, *, like=None) → ndarray
```

CuPy

```
cupy.arange(start, stop=None, step=1, dtype=None) → ndarray
```

dask.array

```
dask.array.arange(start=0, stop, step=1, chunks=<int>, dtype=None) → ndarray
```

JAX

```
jax.numpy.arange(start, stop=None, step=None, dtype=None) → ndarray
```

MXNet

```
np.arange(start, stop=None, step=1, dtype=None, ctx=None) → ndarray
```

PyTorch

```
torch.arange(start=0, end, step=1, out=None, dtype=None, layout=torch.strided, device=None, requires_grad=False) → Ten
```

TensorFlow

```
tf.range(limit, delta=1, dtype=None, name='range') → Tensor
tf.range(start, limit, delta=1, dtype=None, name='range') → Tensor
```

Brookhaven National Laboratory

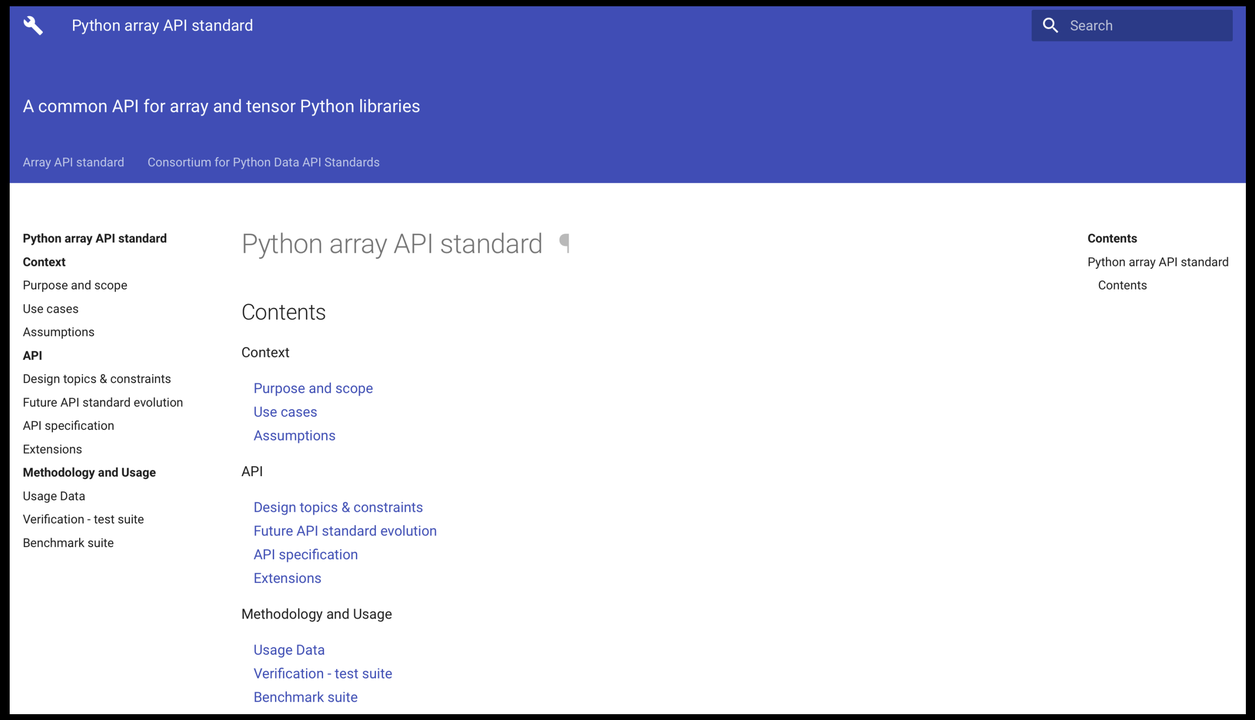
CuPy

Python Array API standard

Solution: Define a subset of NumPy-like ndarray API that everyone thinks is important to be fully compatible, and grow/evolve from there

Stakeholders:

- NumPy
- TensorFlow
- PyTorch
- MXNet
- JAX
- Dask
- **CuPy**



https://data-apis.org/array-api/latest/index.html

Python Data API standard

Consortium for Python Data API Standards: A new organization, with participation from maintainers of many array (a.k.a. tensor) and dataframe libraries. https://data-apis.org/

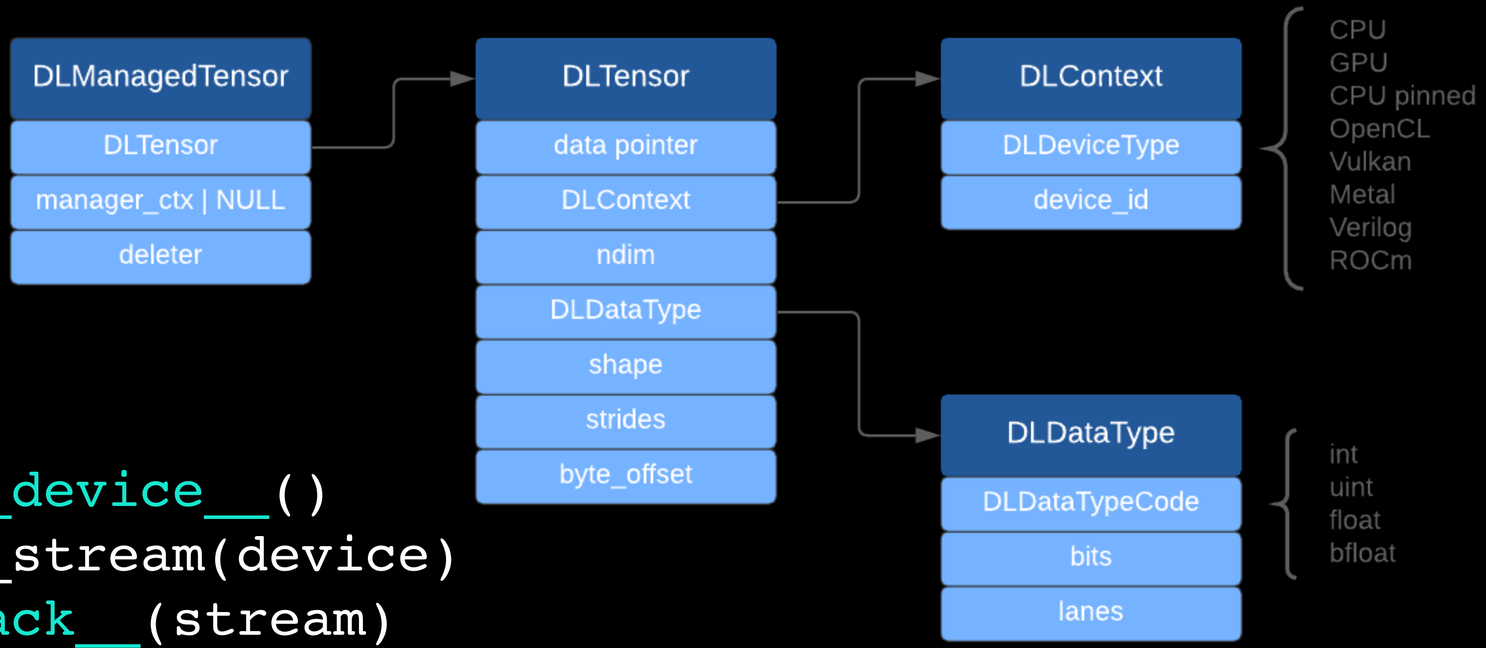
Concrete goals for first year:

1. Define a standardization methodology and necessary tooling for it ✓
2. Publish an RFC for an array API standard ✓
3. Publish an RFC for a dataframe API standard
4. Finalize 2021.0x API standards after community review

Python Array API standard

- 1 array object with
  - 6 attributes: `ndim`, `shape`, `size`, `dtype`, `device`, `T`
  - dunder methods to support all Python operators
  - `__array_api_version__`, `__array_namespace__`, `__dlpack__`
- 11 dtype literals: bool, (u)int8/16/32/64, float32/64
- 1 device object
- 4 constants: `inf`, `nan`, `pi`, `e`
- ~125 functions:
  - Array creation & manipulation (20)
  - Elementwise math & logic (6)
  - Statistics (7)
  - Linear algebra (22)
  - Search, sort & set (7)
  - Utilities, dtypes, broadcasting (8)

Python Array API standard




```
def from_dlpack(arr):
    device = arr.__dlpack_device__()
    stream = get_consumer_stream(device)
    dlcapsule = arr.__dlpack__(stream)
    return dlcapsule_to_consumer_arr(dlcapsule)
```

The Consumer library hands its current stream to the Producer library
(to either synchronize immediately or create a stream order)

Conclusion & Outlook

- CuPy = NumPy + SciPy + Numba, for GPUs
- A drop-in replacement for NumPy-based Python codebase
- Know where your data reside!
- Highly portable (CPU/GPU agnostic code, CUDA & ROCm/HIP)
- Highly extensible (JIT & other custom kernels)
- Highly performant (powered by the entire CUDA software stack)
- Well maintained, supported, and documented
- User / community / distribution friendly
- The library of choice for either new codebase or migration target

Conclusion & Outlook

- Toward better compatibility, interoperability, and performance
- Support Python Array API standard v1
- Full coverage multi-GPU CUDA library routines (cuFFT, cuSOLVER, …)
- libcu++?
- Let us know if your works use CuPy! We love to know!
- What do *you* need? New features? Better docs? Training materials?
- Talk to me if your DOE works need new functionalities in CuPy

GitHub: https://github.com/cupy/cupy
Gitter: https://gitter.im/cupy/community
Docs: https://docs.cupy.dev
Mailing list: https://groups.google.com/g/cupy

Special Thanks

- CuPy team @PFN
  - Kenichi Maehashi
  - ✓ Emilio Castillo
  - Akifumi Imanishi
  - Masayuki Takagi
  - Toshiki Kataoka
- NVIDIA
  - John Kirkham
  - Peter Andreas Entschev
  - Matthew Nicely
  - Keith Kraus
  - Max Katz
- conda-forge community
  - Matthew Becker @ANL
  - Isuru Fernando
- HPC group @BNL
  - Meifeng Lin
  - Zhihua Dong
- NSLS-II, BNL
  - Xiaojing Huang
  - Hanfei Yan
  - Stuart Campbell
  - Maksim Rakitin
- DOE users
  - Daniel Ching @ANL
  - Stefano Marchesini @LBL
  - Pablo Enfadeque @LBL
- AMD
- …and many more people!

Thank you for your attention!

@leofang
leofang@bnl.gov

Work supported in part by BNL LDRD 17-029, 20-024, DOE BES-FWP-PS001

Brookhaven National Laboratory

CuPy