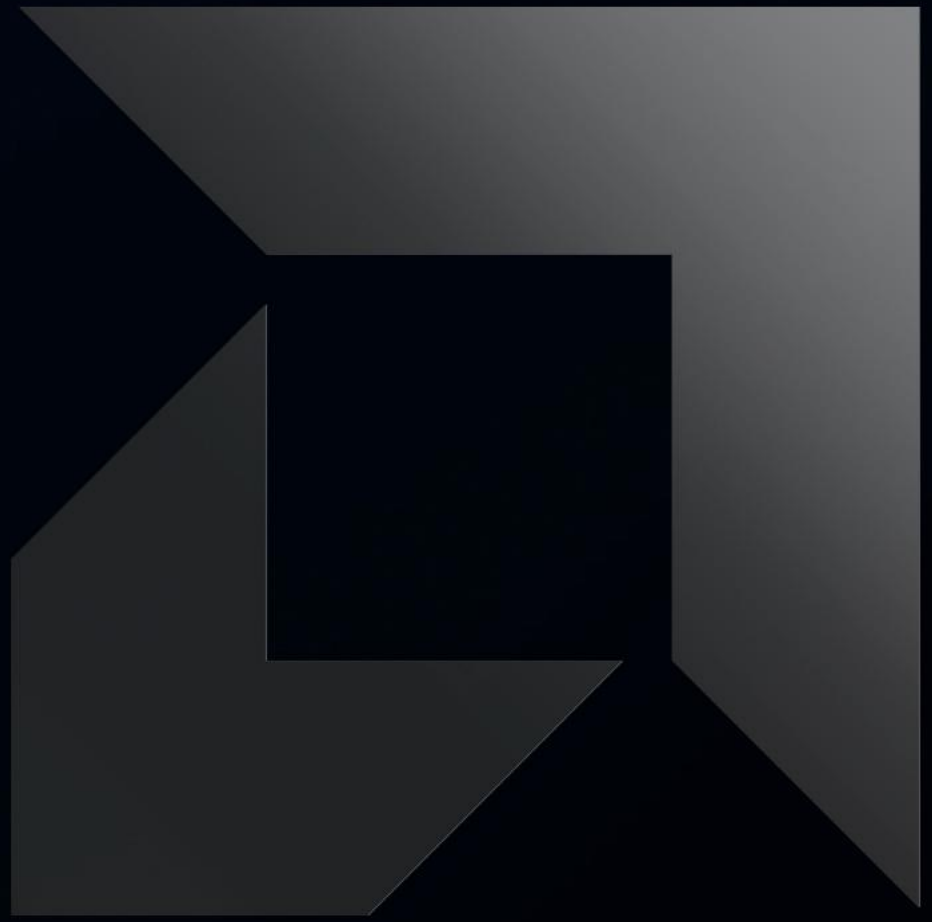


HIP-Python, Numba-HIP, CuPy and MPI4Py

Presenter: Bob Robey
Oct 21-23, 2025
AMD @ Tsukuba University

AMD
together we advance_

HIP-Python and Numba-HIP

[Public]

What is HIP-Python™?

- HIP-Python is a wrapper layer to access many HIP components
- HIP-Python provides low-level Cython and Python bindings for various components of HIP
  - HIP runtime – HIPRTC
  - hipBLAS
  - hipRAND
  - hipFFT
  - hipSparse
  - hipSolver
  - RCCL
  - roctx
- HIP-Python is interoperable with CuPY and CuPY xarrays
- HIP-Python is also a gateway capability that provides an entry point to
  - HIP-Numba
  - HIP-Numpy

- HIP-Python can also take advantage of the single address space on the MI300A and the managed memory capabilities that emulate shared memory on other AMD Instinct™ GPUs
  - Remember to set HSA_XNACK=1 !

[Public]

Installing HIP-Python™

- HIP-Python is installed on the Training system.
- Installing on your system
  - HIP-Python is posted to the TestPyPI site
    - https://test.pypi.org/simple/hip-python
  - Installing

```
python3 -m pip install -i https://test.pypi.org/simple hip-python~=6.4.0
```

- On PyPI, there is a fork of hip-python-fork (not tested)
- Warning: There has been another package on PyPI called hip-python that is not related

Links for hip-python

hip_python-5.4.3.414.6-cp310-cp310-manylinux_2_17_x86_64.whl
hip_python-5.4.3.414.6-cp311-cp311-manylinux_2_17_x86_64.whl
hip_python-5.4.3.414.6-cp38-cp38-manylinux_2_17_x86_64.whl
hip_python-5.4.3.414.6-cp39-cp39-manylinux_2_17_x86_64.whl
hip_python-5.4.3.443.14-cp310-cp310-manylinux_2_17_x86_64.whl

hip_python-6.4.0.549.36-cp313-cp313-manylinux_2_17_x86_64.whl
hip_python-6.4.0.549.36-cp38-cp38-manylinux_2_17_x86_64.whl
hip_python-6.4.0.549.36-cp39-cp39-manylinux_2_17_x86_64.whl
hip_python-6.4.1.552.39-cp310-cp310-manylinux_2_17_x86_64.whl
hip_python-6.4.1.552.39-cp311-cp311-manylinux_2_17_x86_64.whl
hip_python-6.4.1.552.39-cp312-cp312-manylinux_2_17_x86_64.whl
hip_python-6.4.1.552.39-cp313-cp313-manylinux_2_17_x86_64.whl
hip_python-6.4.1.552.39-cp38-cp38-manylinux_2_17_x86_64.whl
hip_python-6.4.1.552.39-cp39-cp39-manylinux_2_17_x86_64.whl

Above is the list of versions on the test PyPI server

Below is the fork on the PyPI server

hip-python-fork 6.3.3.540.31.1

```
pip install hip-python-fork
```

Project description

NOTICE

This is a fork of `hip-python. It will be removed when the hip-python team uploads to PyPI.

[Public]

HIP-Python™ Examples

1. Error Checking
2. Getting Device Properties and Attributes
3. Calling hipBLAS
4. Cython example
5. HIP Python Data Types

Examples are from the rocm documentation

https://rocm.docs.amd.com/projects /hip-python/en/latest/user_guide/ 1_usage.html

[Public]

1. Error checking

- HIP-Python routines always return a tuple. It is best practice to check the returns for errors.
  - The first value is the error code and the second (optional) argument is a string describing the error

```
def hip_check(call_result):
    err = call_result[0]
    result = call_result[1:]
    if len(result) == 1:
        result = result[0]
    if isinstance(err, hip.hipError_t) and err != hip.hipError_t.hipSuccess:
        raise RuntimeError(str(err))
    return result
```

- We won’t show error checks on the slides due to space. The hands-on exercises will include the error checks in the code.

[Public]

2. Getting Device Properties

There are two functions that can be called to get properties and attributes. There is an overlap in the items that can be retrieved by each of these functions.

- hipGetDeviceProperties
  - Over 100 properties can be retrieved with a call

```
props = hip.hipDeviceProp_t()
hip.hipGetDeviceProperties(props,0)

for attrib in sorted(props.PROPERTIES()):
    print(f"props.{attrib}={getattr(props,attrib)}")
```

- hipDeviceGetAttribute

```
device_num = 0
for attrib in (
   hip.hipDeviceAttribute_t.hipDeviceAttributeMaxBlockDimX,
   hip.hipDeviceAttribute_t.hipDeviceAttributeMaxBlockDimY,
   hip.hipDeviceAttribute_t.hipDeviceAttributeMaxBlockDimZ,
   hip.hipDeviceAttribute_t.hipDeviceAttributeMaxGridDimX,
   hip.hipDeviceAttribute_t.hipDeviceAttributeMaxGridDimY,
   hip.hipDeviceAttribute_t.hipDeviceAttributeMaxGridDimZ,
   hip.hipDeviceAttribute_t.hipDeviceAttributeWarpSize,):
      value = hip.hipDeviceGetAttribute(attrib,device_num)
      print(f"{attrib.name}: {value}")
```

[Public]

4. Calling hipBLAS using HIP-Python

Data created using Numpy and then passing the array into hipBLAS as the host pointer

```
import ctypes
import math
import numpy as np

from hip import hip, hipblas

num_elements = 100

# input data on host
alpha = ctypes.c_float(2)
x_h = np.random.rand(num_elements).astype(dtype=np.float32)
y_h = np.random.rand(num_elements).astype(dtype=np.float32)

# expected result
y_expected = alpha*x_h + y_h

# device vectors
num_bytes = num_elements * np.dtype(np.float32).itemsize
x_d = hip.hipMalloc(num_bytes)
y_d = hip.hipMalloc(num_bytes)
```

[Public]

Calling hipBLAS using HIP-Python™

```
# copy input data to device
hip.hipMemcpy(x_d,x_h,num_bytes,hip.hipMemcpyKind.hipMemcpyHostToDevice)
hip.hipMemcpy(y_d,y_h,num_bytes,hip.hipMemcpyKind.hipMemcpyHostToDevice)

# call hipblasSaxpy + initialization & destruction of handle
handle = hipblas.hipblasCreate()
hipblas.hipblasSaxpy(handle, num_elements, ctypes.addressof(alpha), x_d, 1, y_d, 1)
hipblas.hipblasDestroy(handle)

# copy result (stored in y_d) back to host (store in y_h)
hip.hipMemcpy(y_h,y_d,num_bytes,hip.hipMemcpyKind.hipMemcpyDeviceToHost)

# compare to expected result
if np.allclose(y_expected,y_h):
    print("ok")
else:
    print("FAILED")
print(f"{y_h=}")
print(f"{y_expected=}")

# clean up
hip.hipFree(x_d)
hip.hipFree(y_d)
```

Making hipBLAS calls from python

x_h & y_h from numpy

[Public]

4a. Unified Shared Memory version of hipBLAS using HIP-Python

```
import ctypes
import math
import numpy as np

from hip import hipblas

num_elements = 100

alpha = ctypes.c_float(2)
x_h = np.random.rand(num_elements).astype(dtype=np.float32)
y_h = np.random.rand(num_elements).astype(dtype=np.float32)

y_expected = alpha*x_h + y_h

handle = hipblas.hipblasCreate()
hipblas.hipblasSaxpy(handle, num_elements, ctypes.addressof(alpha), x_h, 1, y_h, 1)
hipblas.hipblasDestroy(handle)

if np.allclose(y_expected,y_h):
    print("ok")
else:
    print("FAILED")
```

Input data on host

Expected result

Call hipblasSaxpy + initialization & destruction of handle

Compare to expected result

[Public]

Cython

- With much of the computationally intensive code running on the GPU, it is helpful to compile some of the remaining code that still runs on the CPU.
- We place our code that we want to compile in a .pyx file called array_sum.pyx

```
from hip import hip, hiprtc

def array_sum(double[:, ::1] A):
    cdef int m = A.shape[0]
    cdef int n = A.shape[1]
    cdef int i, j
    cdef double result = 0

    for i in range(m):
        for k in range(n):
            result += A[i, k]

    return result
```

- Then we define an interface to the python routine in array_sum.pyd

```
from hip cimport chip, chiprtc

def array_sum(double[:, ::1] A):
```

[Public]

Cython – setup.py part 1 of 2

```
import os, sys

array_sum = "array_sum"

from setuptools import Extension, setup
from Cython.Build import cythonize

ROCM_PATH=os.environ.get("ROCM_PATH", "/opt/rocm")
HIP_PLATFORM = os.environ.get("HIP_PLATFORM", "amd")

if HIP_PLATFORM not in ("amd", "hcc"):
   raise RuntimeError("Currently only HIP_PLATFORM=amd is supported")

def create_extension(name, sources):
   global ROCM_PATH
   global HIP_PLATFORM
   rocm_inc = os.path.join(ROCM_PATH,"include")
   rocm_lib_dir = os.path.join(ROCM_PATH,"lib")
   rocm_libs = ["amdhip64"]
   platform = HIP_PLATFORM.upper()
   cflags = ["-D", f"__HIP_PLATFORM_{platform}__"]
```

[Public]

Cython – setup.py part 2 of 2

```
    return Extension(
        name,
        sources=sources,
        include_dirs=[rocm_inc],
        library_dirs=[rocm_lib_dir],
        libraries=rocm_libs,
        language="c",
        extra_compile_args=cflags,
    )

setup(
    ext_modules = cythonize(
        [create_extension(array_sum, [f"{array_sum}.pyx"]),],
        compiler_directives=dict(language_level=3),
        compile_time_env=dict(HIP_PYTHON=True),
    )
)
```

[Public]

Cython – compiling the array sum python code

- Create a virtual environment, pip install cython along with the hip-python module. Run setup.py build.

```
python3 -m venv cython_example
source cython_example/bin/activate
```

- Set up the environment by loading the rocm and hip-python module. Install cython

```
module load rocm hip-python
pip3 import cython
```

- Compile the array_sum python code with setup.py build

```
python3 setup.py build
```

- Cleanup

```
deactivate
rm -rf cython_example
```

[Public]

Numba-HIP

- Numba is a Just-inTime (JIT) compiler for Python™ numerical functions
- We have installed it as part of the hip-python module
- If experimenting with numba-hip and hip-python on your own system, you can install it with the following in a virtual environment

```
python3 -m venv hip-python-build
source hip-python-build/bin/activate
python3 -m pip install pip
python3 -m pip install -i https://test.pypi.org/simple hip-python~=6.4.0
python3 -m pip config set global.extra-index-url https://test.pypi.org/simple
python3 -m pip install "numba-hip[rocm-6-4-0] @ git+https://github.com/ROCm/numba-hip.git“
```

- To clean up after working in the virtual environment

```
deactivate
rm -rf hip-python-build
```

[Public]

Numba-HIP example

- File numba-hip.py
- from numba import hip
  - This example code differs from CUDA only in the line @cuda.jit which is @hip.jit for numba-hip

```
@hip.jit
def f(a, b, c):
    # like threadIdx.x + (blockIdx.x * blockDim.x)
    tid = hip.grid(1)
    size = len(c)

    if tid < size:
        c[tid] = a[tid] + b[tid]

print("Ok")
```

- Run the example with

```
module load rocm hip-python
python3 numba-hip.py
```

[Public]

Numba-HIP posing as CUDA

- We can also have numba-hip pose as CUDA so that we don’t have to change all the jit calls in our program

```
from numba import hip

hip.pose_as_cuda()
from numba import cuda

@cuda.jit
def f(a, b, c):
   # like threadIdx.x + (blockIdx.x * blockDim.x)
   tid = hip.grid(1)
   size = len(c)

   if tid < size:
       c[tid] = a[tid] + b[tid]

print("Ok")
```

Enable Numba HIP to
pose as Numba CUDA

CuPy

[Public]

What is CuPy

- NumPy is a python interface to optimized routines written in C that provide arrays, multi-dimensional arrays and common numerical operations on them. These are much faster than operating on Python lists
- SciPy provides fundamental algorithms common in scientific and numerical computing. The underlying code is a mixture of Fortran, C and C++
- CuPy is a NumPy/SciPy-compatible array library for GPU-accelerated computing with Python
- CuPy acts as a drop-in replacement to run existing NumPy/SciPy code on NVIDIA CUDA or AMD ROCm™ platforms
- CuPy provides the `ndarray`, sparse matrices, and the associated routines for GPU devices, most having the same API as NumPy and SciPy.
- CuPy provides interfaces to GPU optimized libraries such as rocBLAS, rocSPARSE, rocFFT, and RCCL

source: cupy documentation

[Public]

CuPy and HIP

- CuPy uses HIP as backhand to run on AMD GPUs

- **HIP**: **H**eterogeneous-compute **I**nterface for **P**ortability
  - C++ runtime API and kernel language
  - Works on AMD and Nvidia GPUs

- The CPU is often referred to as the host, and the GPU as the device

- In HIP, launching a kernel is **non-blocking for the host**
  - After sending instructions/data, the host continues to do more work while the device executes the kernel. This means GPU execution and CPU activity can overlap

- What it means for CuPy: appropriate synchronization calls have to be made after a kernel call:
  - `cupy.cuda.Device(0).synchronize()`
  - `cupy.cuda.Stream.synchronize()`

- In HIP, memory copies such as `hipMemcpy` is **blocking for the host**
  - All activity on the host stops until the copy has completed.
- What it means for CuPy: no need to sync if calling a memory copy right after a kernel.

[Public]

click here for differences between CuPy and NumPy

CuPy functions

CuPy vs NumPy API

CuPy-specific functions

Comparison Table

Here is a list of NumPy / SciPy APIs and its corresponding CuPy implementations.

`-` in CuPy column denotes that CuPy implementation is not provided yet. We welcome contributions for these functions.

NumPy / CuPy APIs

Module-Level

NumPy	CuPy
`numpy.DataSource`	`cupy.DataSource` (*alias of* `numpy.DataSource`)
`numpy.ScalarType`	-
`numpy.abs`	`cupy.abs`
`numpy.absolute`	`cupy.absolute`
`numpy.add`	`cupy.add`
`numpy.all`	`cupy.all`
`numpy.allclose`	`cupy.allclose`

full list here: cupy_documentation

CuPy-specific functions

CuPy-specific functions are placed under `cupyx` namespace.

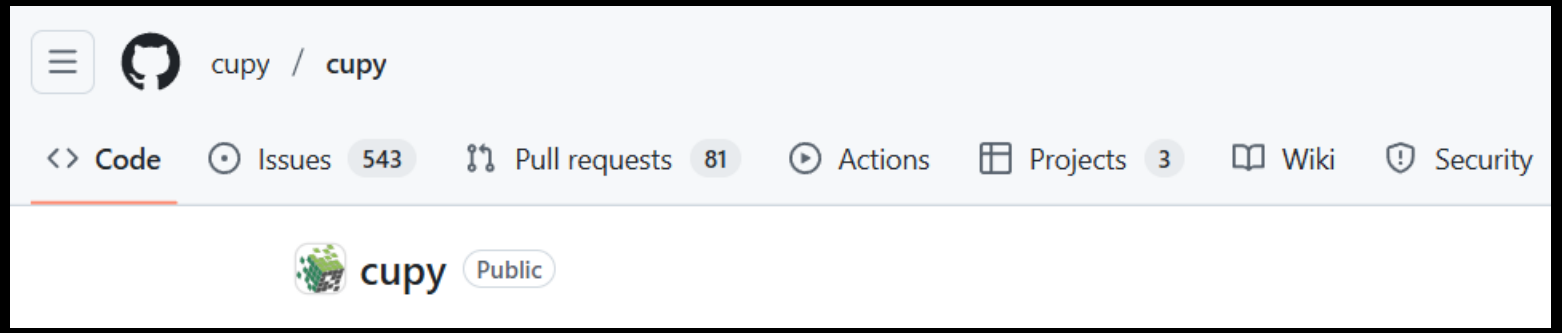
`cupyx.rsqrt`	Returns the reciprocal square root.
`cupyx.scatter_add` (a, slices, value)	Adds given values to specified elements of an array.
`cupyx.scatter_max` (a, slices, value)	Stores a maximum value of elements specified by indices to an array.
`cupyx.scatter_min` (a, slices, value)	Stores a minimum value of elements specified by indices to an array.
`cupyx.empty_pinned` (shape[, dtype, order])	Returns a new, uninitialized NumPy array with the given shape and dtype.
`cupyx.empty_like_pinned` (a[, dtype, order, ...])	Returns a new, uninitialized NumPy array with the same shape and dtype as those of the given array.

full list here: cupy_documentation

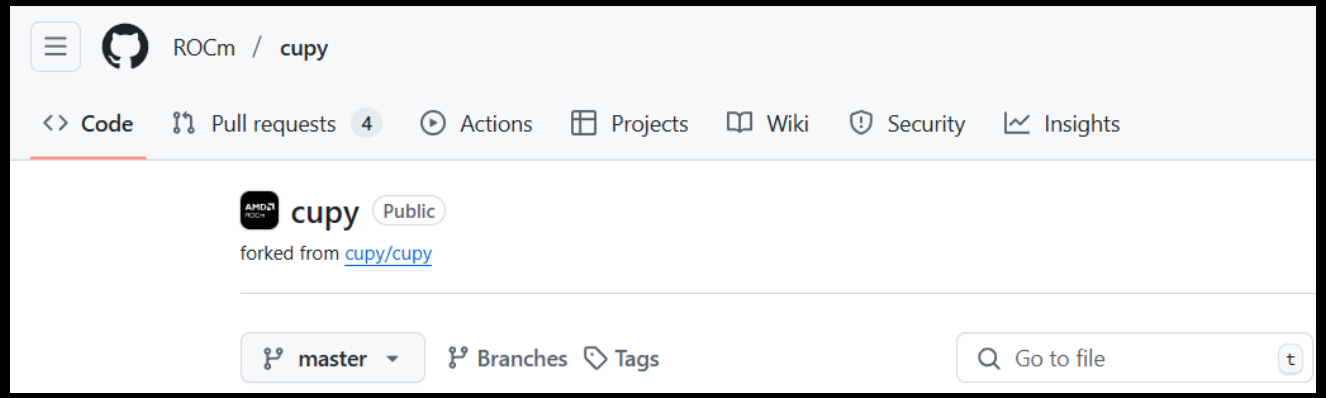
[Public]

CuPy Installation – GitHub Repos

- There are two GitHub repos to take the CuPy source code from to run on AMD GPUs
- We are using the upstream CuPy repository: https://github.com/cupy/cupy

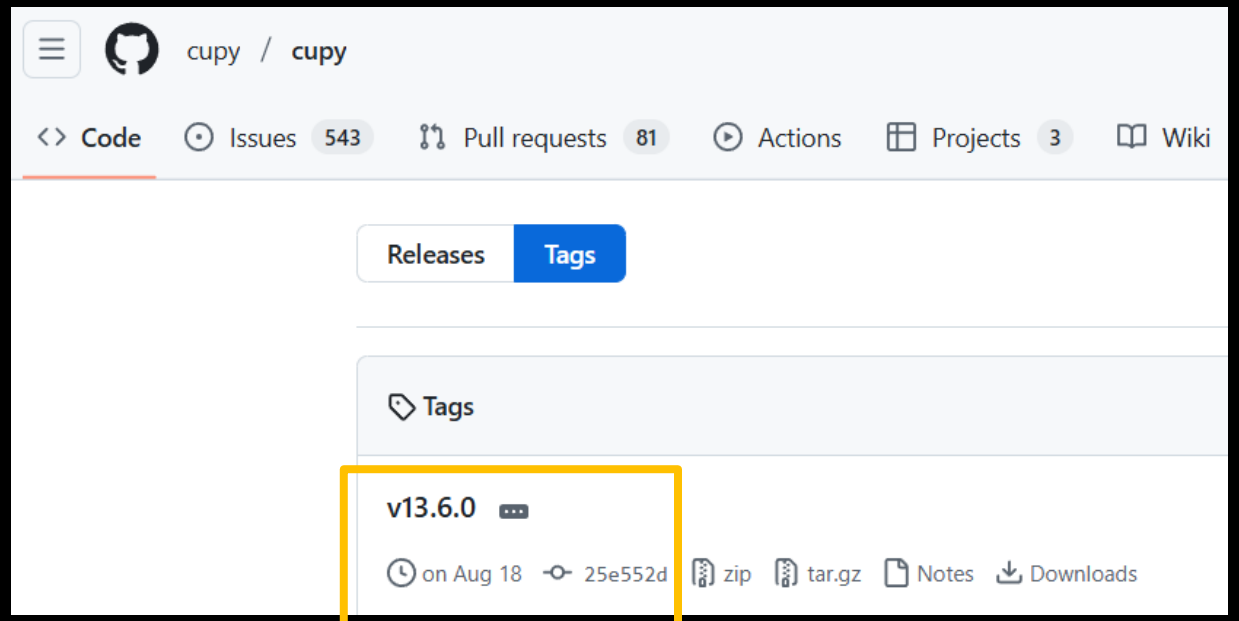


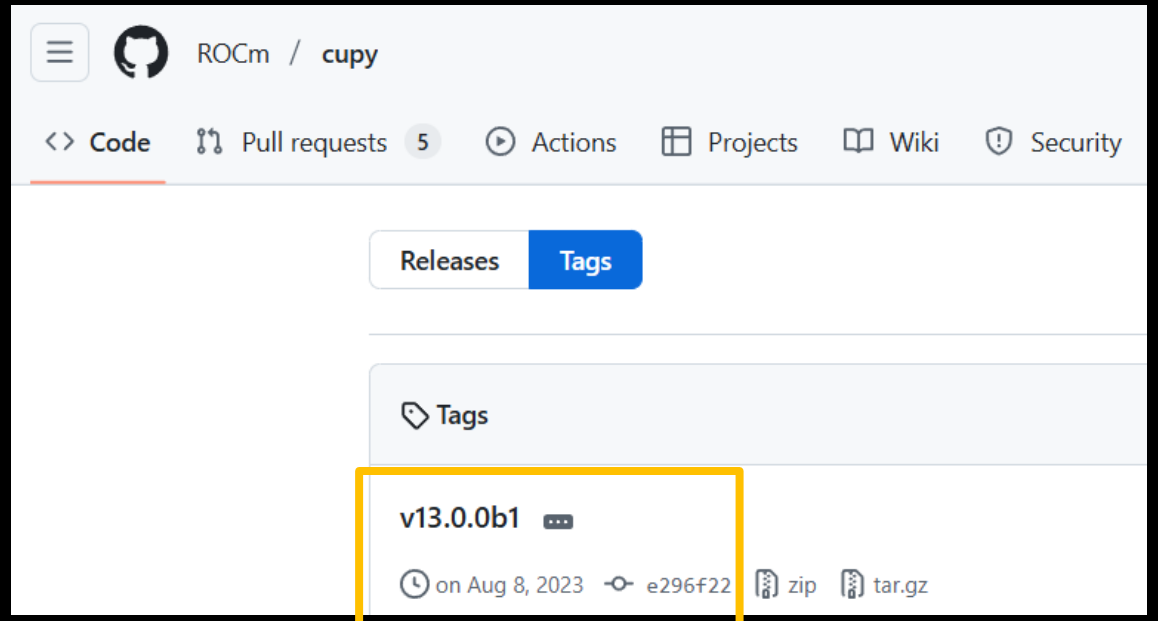
- There is also a fork of the CuPy upstream repository in the ROCm github: https://github.com/rocm/cupy



[Public]

CuPy Installation – Versions





Upstream versions are more recent

The one above is the one we have installed

As of September 23rd 2025

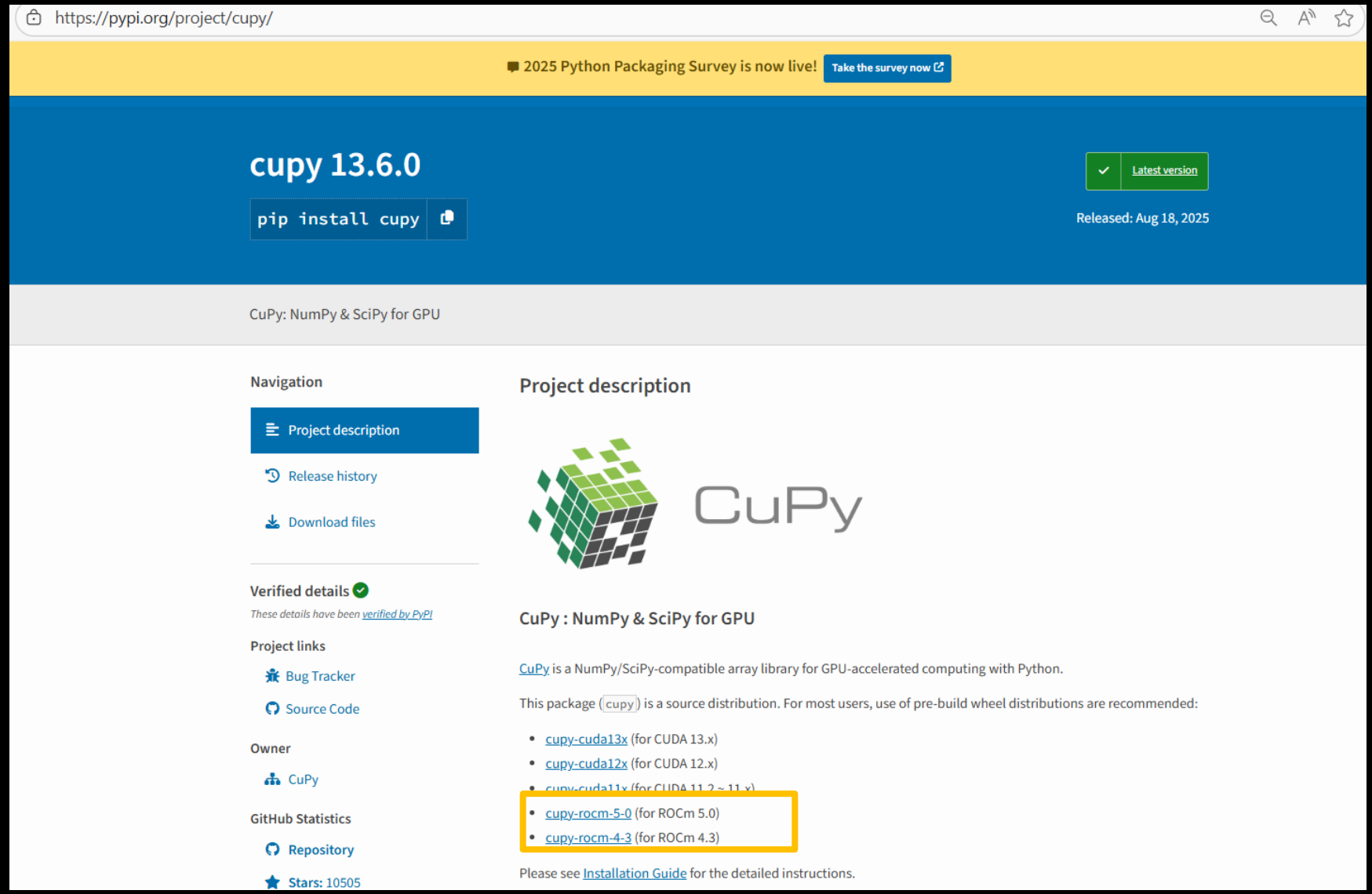
ROCm repo versions tend to be behind

There is work from AMD to get changes pushed directly to the upstream repo

The `ROCm/cupy` will soon be updated too

[Public]

CuPy – Installation with `pip3` (pre-built wheel for Linux® x86_64)



As of September 23rd 2025

```
pip3 install cupy-rocm-5-0

pip3 install cupy-rocm-4-3
```

Only old versions of ROCm currently available as pre-built wheels

Wheels for ROCm 6.4 and 7 will soon be available

[Public]

CuPy – Robust Installation from Source

Installation from source script available in our model installation repository:
https://github.com/amd/HPCTrainingDock/blob/main/extras/scripts/cupy_setup.sh

also installs **numpy-allocator** to leverage unified shared memory and **cupy-xarray**

… most relevant part reported below…

```
export CUPY_INSTALL_USE_HIP=1
export ROCM_HOME=${ROCM_PATH}
export HIPCC=${ROCM_HOME}/bin/hipcc
export HCC_AMDGPU_ARCH=${AMDGPU_GFXMODEL}
python3 -m venv cupy_build
source cupy_build/bin/activate
pip3 install -v --target=$CUPY_PATH pytest mock xarray[complete] dask build numpy-allocator --no-cache
export PYTHONPATH=$PYTHONPATH:$CUPY_PATH
# Get source from the upstream repository of CuPy.
git clone -q --depth 1 -b v$CUPY_VERSION --recursive https://github.com/cupy/cupy.git
cd cupy
python3 -m build --wheel
pip3 install -v --upgrade --target=$CUPY_PATH dist/*.whl
pip3 install -v --target=$CUPY_PATH cupy-xarray --no-deps
deactivate
```

[Public]

Basics of CuPy

- Must import the CuPy Python™ module in your Python code: `import cupy as cp`
- To create an array on the device use `cp.array`: `gpu_array = cp.array(cpu_array)`
- To copy data from GPU to CPU, use `cp.asnumpy`: `cpu_array = cp.asnumpy(gpu_array)`
- To copy back from CPU to GPU use `cp.asarray`: `gpu_array2 = cp.asarry(cpu_array)`
- Operations between GPU arrays will be done on the GPU: `result_gpu = gpu_array + gpu_array2`
- CuPy has the concept of a current device – usually GPU device 0: `gpu_array.device`
- Note that the device will be called `<CUDA Device 0>` even if you are on AMD GPUs.

[Public]

NumPy – CuPy Interoperability

- CuPy implements a subset of the NumPy interface by implementing `cupy.ndarray`, a counterpart to NumPy `ndarrays`

- The `cupy.ndarray` object implements the `__array_ufunc__` interface. This enables NumPy universal functions (ufunc) to be applied to CuPy arrays. Note that the return type of these operations **is still consistent** with the **initial type**.

```
>>> import cupy as cp
>>> import numpy as np
>>> gpu_arr = cp.random.randn(1, 2, 3, 4).astype(cp.float32)
>>> result = np.sum(gpu_arr)
>>> print(type(result))
<class 'cupy._core.core.ndarray'>
```

- `cupy.ndarray` also implements the `__array_function__` interface, meaning it is possible to do operations such as

```
a = np.random.randn(100, 100)
a_gpu = cp.asarray(a)
qr_gpu = np.linalg.qr(a_gpu)
```

source: numpy-documentation

[Public]

Simple CuPy code example

- First get the example to run from the training examples repository

```
git clone https://github.com/amd/HPCTrainingExamples
cd HPCTrainingExamples/Python/cupy
```

- Set up the environment: note the "module" below is not the Python™ module

```
module load cupy
```

- Run the example

```
python3 cupy_array_sum.py
```

- Output should be:

```
CuPy Array: [1 2 3 4 5]
Squared CuPy Array: [ 1  4  9 16 25]
NumPy Array: [5 6 7 8 9]
CuPy Array from NumPy: [5 6 7 8 9]
Addition Result on GPU: [ 6  8 10 12 14]
Result on CPU: [ 6  8 10 12 14]
```

[Public]

Simple CuPy code example: a closer look

```
import cupy as cp
import numpy as np

# Create a CuPy array
gpu_array = cp.array([1, 2, 3, 4, 5])
print("CuPy Array:", gpu_array)

# Perform operations on the GPU
gpu_array_squared = gpu_array ** 2
print("Squared CuPy Array:", gpu_array_squared)

# Create a NumPy array
cpu_array = np.array([5, 6, 7, 8, 9])
print("NumPy Array:", cpu_array)

# Transfer NumPy array to GPU
gpu_array_from_cpu = cp.asarray(cpu_array)
print("CuPy Array from NumPy:",
gpu_array_from_cpu)

# Perform element-wise addition
result_gpu = gpu_array + gpu_array_from_cpu
print("Addition Result on GPU:", result_gpu)

# Transfer result back to CPU
result_cpu = cp.asnumpy(result_gpu)
print("Result on CPU:", result_cpu)
```

Creates an array on the device

Operations occur on the GPU

Converts NumPy array to CuPy array

Operations occur on the GPU

Returns an array on the host memory from an arbitrary source array (device in this case)

[Public]

Verifying that CuPy code example runs on the AMD GPU

- Now run with the AMD_LOG_LEVEL environment variable set

```
export AMD_LOG_LEVEL=3
python3 cupy_array_sum.py
```

- Lots of output now – showing just a little bit:

```
hipMemcpyAsync ( 0x559ea98f65f0, 0x7f4556800000, 40, hipMemcpyDeviceToHost, stream:<null> )
Signal = (0x7f4d5efff280), Translated start/end = 1083534945452078 / 1083534945453358,
  Elapsed = 1280 ns, ticks start/end = 27091222405615 / 27091222405647, Ticks elapsed = 32
Host active wait for Signal = (0x7f4d5efff200) for -1 ns
Set Handler: handle(0x7f4d5efff180), timestamp(0x559eaabead90)
Host active wait for Signal = (0x7f4d5efff180) for -1 ns
hipMemcpyAsync: Returned hipSuccess : : duration: 5948d us
hipStreamSynchronize ( stream:<null> )
Handler: value(0), timestamp(0x559eaa7e7350), handle(0x7f4d5efff180)
hipStreamSynchronize: Returned hipSuccess :
hipSetDevice ( 0 )
hipSetDevice: Returned hipSuccess :
CuPy Array: [1 2 3 4 5]
```

[Public]

Unified Memory Programming on CuPy

Unified memory programming (UMP) support (**experimental!**)

It is possible to make both NumPy and CuPy use/share system allocated memory on Heterogeneous Memory Management (HMM) or Address Translation Services (ATS) enabled systems, such as the NVIDIA Grace Hopper Superchip. To activate this capability, currently you need to:

1. Install numpy_allocator
2. Set the environment variable `CUPY_ENABLE_UMP=1`
3. Make a memory pool for CuPy to draw system memory (`malloc_system`), for example:

```
import cupy as cp
cp.cuda.set_allocator(cp.cuda.MemoryPool(cp.cuda.memory.malloc_system).malloc)
```

4. Switch to the aligned allocator for NumPy to draw system memory

```
import cupy._core.numpy_allocator as ac
import numpy_allocator
import ctypes
lib = ctypes.CDLL(ac.__file__)
class my_allocator(metaclass=numpy_allocator.type):
    _calloc_ = ctypes.addressof(lib._calloc)
    _malloc_ = ctypes.addressof(lib._malloc)
    _realloc_ = ctypes.addressof(lib._realloc)
    _free_ = ctypes.addressof(lib._free)
my_allocator.__enter__()  # change the allocator globally
```

On AMD GPUs, you additionally need:
`export HSA_XNACK=1`

this will enable unified shared memory on MI300A

or managed memory on MI200s and MI300X

With this setup change, all the data movement APIs such as `get()`, `asnumpy()` and `asarray()` become no-op (no copy is done), and the following code is accelerated:

```
# a, b, c are np.ndarray, d is cp.ndarray
a = np.random.random(100)
b = np.random.random(100)
c = np.add(a, b)
d = cp.matmul(cp.asarray(a), cp.asarray(c))
```

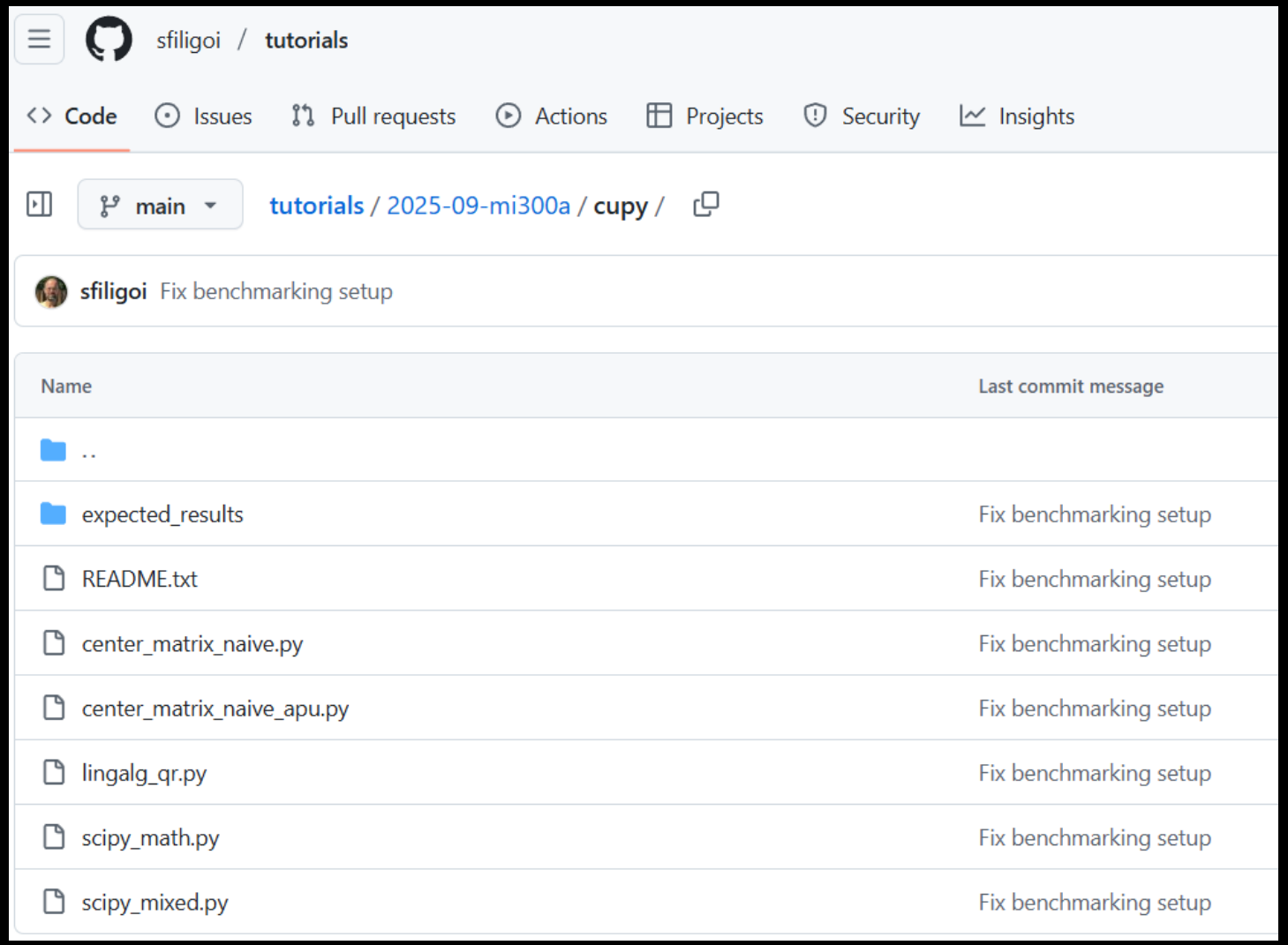
Essentially, the distinction of CPU/GPU *memory* spaces is gone, and `np` / `cp` are used to represent the *execution* space (whether the code should run on CPU or GPU).

Apart from the setup configuration for NumPy/CuPy, no user code chage is required.

source: [cupy docs]

[Public]

Additional Examples on CuPy



Code examples by Igor Sfiligoi

```
git clone https://github.com/sfiligoi/tutorials.git
cd tutorials/2025-09-mi300a/cupy
```

From 2025 ICPP AI tutorial
by San Diego Supercomputing Center
and AMD.

[Public]

CuPy-Xarray: Xarray on GPUs

- Xarray: Python™ library to work with labelled multi-dimensional arrays
  - Popular for applications where multi-dimensional data needs to be handled (such as climate modeling)
  - Built on top of NumPy
  - Has built-in support for NetCDF
  - Can wrap custom duck array objects (i.e. NumPy-like arrays) that follow specific protocols.

- When used together, Xarray and CuPy can provide an easy way to take advantage of GPU acceleration for scientific computing tasks.

- CuPy-Xarray provides an interface for using CuPy in Xarray, providing accessors on the Xarray objects.
  - CuPy-Xarray relies on an existing CuPy installation, install CuPy first

- Cupy-Xarray github repo: https://github.com/xarray-contrib/cupy-xarray
  - Install with `pip install cupy-xarray --no-deps` after installing CuPy

- Issue with `dask`: https://github.com/xarray-contrib/cupy-xarray/pull/62
  - Did not make it into the latest release
  - Make sure to install dask with `pip install dask`

source: cupy-xarray documentation

[Public]

Simple CuPy-Xarray code example

- First get the example to run from the training examples repository

```
git clone https://github.com/amd/HPCTrainingExamples
cd HPCTrainingExamples/Python/cupy
```

- Set up the environment: note the "module" below is not the Python™ module

```
module load cupy
```

- Run the example

```
python cupy_xarray_test.py
```

```
Is the array used to create da_np on device? False
Is the array used to create da_cp on device? True
da_cp.data is of type: <class 'cupy.ndarray'>
check that arr_gpu and cupy_array are the same with CuPy: True
check the arr_gpu and cupy_array are the same with NumPy (interoperability): True
arr_gpu is on device: <CUDA Device 0>
arr_cpu is on device: cpu
total number of available devices: 8
arr_gpu2 is on device: <CUDA Device 2>
```

source: cupy-xarray documentation

[Public]

Simple CuPy-Xarray code example: a closer look

```
import cupy as cp
import numpy as np
import xarray as xr
import cupy_xarray
```

Adds .cupy to Xarray objects

```
arr_cpu = np.random.rand(10, 10, 10)
arr_gpu = cp.random.rand(10, 10, 10)
```

Creates an array on the CPU with NumPy

Creates an array on the GPU with CuPy

```
da_np = xr.DataArray(arr_cpu, dims=["x", "y", "time"])
da_cp = xr.DataArray(arr_gpu, dims=["x", "y", "time"])
```

Creates a DataArray using NumPy array

Creates a DataArray using CuPy array

```
. . . (some code omitted) . . .
cupy_array = da_cp.data
```

Access the underlying CuPy array used to create the xarray.DataArray

```
print("check that arr_gpu and cupy_array are the same with CuPy:",
cp.allclose(cupy_array,arr_gpu))
```

Use CuPy to check that the array used to create the xarray and the one given by xarray are the same

```
print("check the arr_gpu and cupy_array are the same with NumPy
(interoperability):", np.allclose(cupy_array,arr_gpu))
. . . (some code omitted) . . .
```

Use NumPy to check that the array used to create the xarray and the one given by xarray are the same

```
with cp.cuda.Device(2):
    arr_gpu2 = cp.array([1, 2, 3, 4, 5])
print("arr_gpu2 is on device:", arr_gpu2.device)
```

Use the device context manager to create data on a different device

MPI4Py

[Public]

What is MPI4Py

- The Message Passing Interface (MPI) is a standardized and portable message-passing system designed to function on a wide variety of parallel computers
- The MPI standard defines the syntax and semantics of library routines and allows users to write portable programs in the main scientific programming languages (Fortran, C, or C++).
- MPI for Python™ provides (MPI4Py) MPI bindings for the Python™ programming language, allowing any Python program to exploit multiple processors across multiple nodes.
- MPI4Py can send data directly from one GPU to another GPU by using GPU-aware MPI.
- MPI4Py can be configured to use any MPI implementation

source: mpi4py documentation

[Public]

MPI4Py Installation

- Installation script uses the MPI version specified in the environment variable `MPI_PATH`
- Current installation script uses the OpenMPI GPU-Aware MPI
- `MPI_PATH` is defined in the OpenMPI module

```
module load rocm
git clone --branch 4.0.3 https://github.com/mpi4py/mpi4py.git
cd mpi4py
echo "[model]              = ${MPI_PATH}" >> mpi.cfg
echo "mpi_dir              = ${MPI_PATH}" >> mpi.cfg
echo "mpicc                = ${MPI_PATH}"/bin/mpicc >> mpi.cfg
echo "mpic++               = ${MPI_PATH}"/bin/mpic++ >> mpi.cfg
echo "library_dirs         = %(mpi_dir)s/lib" >> mpi.cfg
echo "include_dirs         = %(mpi_dir)s/include" >> mpi.cfg
CC=${ROCM_PATH}/bin/amdclang CXX=${ROCM_PATH}/bin/amdclang++ python3 setup.py build --mpi=model
CC=${ROCM_PATH}/bin/amdclang CXX=${ROCM_PATH}/bin/amdclang++ python3 setup.py bdist_wheel
pip3 install -v --target=${MPI4PY_PATH} dist/mpi4py-*.whl
```

[Public]

MPI4Py vs OpenMPI API Comparison

MPI4Py

Allreduce(*sendbuf, recvbuf, op=SUM*)

Reduce to All.

Parameters:
- **sendbuf** (*BufSpec* | *InPlace*)
- **recvbuf** (*BufSpec*)
- **op** (*Op*)

Return type: None

Bcast(*buf, root=0*)

Broadcast data from one process to all other processes.

Parameters:
- **buf** (*BufSpec*)
- **root** (*int*)

Return type: None

Send(*buf, dest, tag=0*)

Blocking send.

Note

This function may block until the message is received. Whether `Send` blocks or not depends on several factors and is implementation dependent.

Parameters:
- **buf** (*BufSpec*)
- **dest** (*int*)
- **tag** (*int*)

Return type: None

OpenMPI

```
int MPI_Allreduce(const void *sendbuf, void *recvbuf, int count,
                  MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)
```

```
int MPI_Bcast(void *buffer, int count, MPI_Datatype datatype,
    int root, MPI_Comm comm)
```

```
int MPI_Send(const void *buf, int count, MPI_Datatype datatype, int dest,
     int tag, MPI_Comm comm)
```

Notes on MPI4Py API

- Use methods with all-lowercase name for generic Python™ objects
- Use methods with an upper-case letter for buffer-like objects
- Source: mpi4py tutorial

[Public]

Note about GPU Aware MPI and MPI4Py

- If mpi4py is built against a GPU-aware MPI implementation, GPU arrays can be passed to upper-case methods as long as they have either the `__dlpack__` and `__dlpack_device__` methods or the `__cuda_array_interface__` attribute that are compliant with the respective standard specifications.
- Only C-contiguous or Fortran-contiguous GPU arrays are supported.
- GPU buffers must be fully ready before any MPI routines operate on them to avoid race conditions. This can be ensured by using the synchronization API of your array library (as we'll see in the next example). mpi4py does not have access to any GPU-specific functionality and thus cannot perform this operation automatically for users.

source: mpi4py tutorial

[Public]

MPI4Py and CuPy example: Allreduce and Bcast

Find the example in our exercises repo:
https://github.com/amd/HPCTrainingExamples/blob/main/Python/mpi4py/mpi4py_cupy.py

```python
def mpi4py_cupy_test():

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

    # Allreduce
    if rank == 0:
      print("Starting allreduce test...")
    sendbuf = cupy.arange(10, dtype='i')
    recvbuf = cupy.empty_like(sendbuf)
    # always make sure the GPU buffer is ready before any MPI operation
    cupy.cuda.get_current_stream().synchronize()
    comm.Allreduce(sendbuf, recvbuf)
    assert cupy.allclose(recvbuf, sendbuf*size)

    # Bcast
    if rank == 0:
      print("Starting bcast test...")
    if rank == 0:
        buf = cupy.arange(100, dtype=cupy.complex64)
    else:
        buf = cupy.empty(100, dtype=cupy.complex64)
    cupy.cuda.get_current_stream().synchronize()
    comm.Bcast(buf)
    assert cupy.allclose(buf, cupy.arange(100, dtype=cupy.complex64))
```

Returns an array with evenly spaced values within a given interval:
in this case it will be 10,11,…,19

Returns a new array with same shape and dtype of `sendbuf`.

Similar to the corresponding numpy calls but happening on the GPU

Note that the call `cupy.cuda.get_current_stream()` returns an object of type `cupy.cuda.Stream`, see the documentation for the full list of methods, including `synchronize()`

alias for `numpy.complex64` which is a `float complex` in C

Returns `True` if the two arrays are element-wise equal within a tolerance, Using this formula:

$$|a - b| \leq a * tol + |b| * rtol$$

where `a` is `recvbuf`, `b` is `sendbuf*size`, and by default `tol=1.e-08` and `rtol=1.e-05`

[Public]

MPI4Py and CuPy example: Send-Recv

Find the example in our exercises repository:
https://github.com/amd/HPCTrainingExamples/blob/main/Python/mpi4py/mpi4py_cupy.py

```
# Send-Recv
if rank == 0:
  print("Starting send-recv test...")

if rank == 0:
    buf = cupy.arange(20, dtype=cupy.float64)
    cupy.cuda.get_current_stream().synchronize()
    for j in range(1,size):
       comm.Send(buf, dest=j, tag=88+j)
else:
    buf = cupy.empty(20, dtype=cupy.float64)
    cupy.cuda.get_current_stream().synchronize()
    comm.Recv(buf, source=0, tag=88+rank)
    assert cupy.allclose(buf, cupy.arange(20, dtype=cupy.float64))

if rank == 0:
  print("Success")
```

Add:
`print("Rank is:", rank)`
to show that multiple processes are executing

Then run with:
```
module load mpi4py cupy
mpirun -n 4 python3 mpi4py_cupy.py
```
and see this output:

```
Rank is: 2
Rank is: 1
Rank is: 3
Rank is: 0
Starting allreduce test...
Starting bcast test...
Starting send-recv test...
Success
```

[Public]

Verifying that MPI4Py and CuPy example runs on the GPU

Set the AMD_LOG_LEVEL

```
export AMD_LOG_LEVEL=3
```

Then run again

```
mpirun -n 4 python3 mpi4py_cupy.py
```

and see a lot more output including:

```
hiprtcCreateProgram ( 0x7fffa382ee28, #include <cupy/complex.cuh>
#include <cupy/carray.cuh>
#include <cupy/atomics.cuh>
#include <cupy/math_constants.h>
#include <cupy/hip_workaround.cuh>

typedef bool type_in0_raw;
typedef bool type_out0_raw;
typedef int IndexT;

#define REDUCE(a, b) (a & b)
#define POST_MAP(a) (out0 = a)
#define _REDUCE(_offset) if (_tid < _offset) {   _type_reduce _a = _sdata[_tid], _b = _sdata[(_tid + _offset)];   _sdata[_tid] =
REDUCE(_a, _b); }

typedef bool _type_reduce;
extern "C" __global__ void cupy_all(const CArray<bool, 1, 1, 1> _raw_in0, CArray<bool, 0, 1, 1> _raw_out0, CIndexer<1, 1> _in_ind,
CIndexer<0, 1> _out_ind, const int _block_stride) {
  __shared__ char _sdata_raw[256 * sizeof(_type_reduce)];
  _type_reduce *_sdata = reinterpret_cast<_type_reduce*>(_sdata_raw);
  unsigned int _tid = threadIdx.x;
```


AMD
together we advance_

[Public]

Additional Resources

- CuPy vs NumPy speed comparison: https://cupy-xarray.readthedocs.io/latest/examples/01_cupy-basics.html#cupy-vs-numpy-speed-comparison

- Real world example of Cupy-Xarray: https://cupy-xarray.readthedocs.io/latest/examples/06_real-example.html
    - Note: you might need to modify the data read line to this if it is taking too long to get the data:

```
da = xr.open_mfdataset(file_objs, engine="h5netcdf", compat="override", coords='minimal')[var].load()
```

- Cupy with Xarray vs NumPy with Xarray performance comparison: https://cupy-xarray.readthedocs.io/latest/examples/03_basic-computations.html#comparing-performance-cupy-with-xarray-vs-numpy-with-xarray

- MPI presentation (touching C,Fortran and Python™) from Rolf Rabenseifner at HLRS: https://fs.hlrs.de/projects/par/par_prog_ws/pdf/mpi_3.1_rab.pdf

[Public]

Disclaimer

The information presented in this document is for informational purposes only and may contain technical inaccuracies, omissions, and typographical errors. The information contained herein is subject to change and may be rendered inaccurate for many reasons, including but not limited to product and roadmap changes, component and motherboard version changes, new model and/or product releases, product differences between differing manufacturers, software changes, BIOS flashes, firmware upgrades, or the like. Any computer system has risks of security vulnerabilities that cannot be completely prevented or mitigated. AMD assumes no obligation to update or otherwise correct or revise this information. However, AMD reserves the right to revise this information and to make changes from time to time to the content hereof without obligation of AMD to notify any person of such revisions or changes.

THIS INFORMATION IS PROVIDED ‘AS IS.” AMD MAKES NO REPRESENTATIONS OR WARRANTIES WITH RESPECT TO THE CONTENTS HEREOF AND ASSUMES NO RESPONSIBILITY FOR ANY INACCURACIES, ERRORS, OR OMISSIONS THAT MAY APPEAR IN THIS INFORMATION. AMD SPECIFICALLY DISCLAIMS ANY IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY, OR FITNESS FOR ANY PARTICULAR PURPOSE. IN NO EVENT WILL AMD BE LIABLE TO ANY PERSON FOR ANY RELIANCE, DIRECT, INDIRECT, SPECIAL, OR OTHER CONSEQUENTIAL DAMAGES ARISING FROM THE USE OF ANY INFORMATION CONTAINED HEREIN, EVEN IF AMD IS EXPRESSLY ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

Third-party content is licensed to you directly by the third party that owns the content and is not licensed to you by AMD. ALL LINKED THIRD-PARTY CONTENT IS PROVIDED “AS IS” WITHOUT A WARRANTY OF ANY KIND. USE OF SUCH THIRD-PARTY CONTENT IS DONE AT YOUR SOLE DISCRETION AND UNDER NO CIRCUMSTANCES WILL AMD BE LIABLE TO YOU FOR ANY THIRD-PARTY CONTENT. YOU ASSUME ALL RISK AND ARE SOLELY RESPONSIBLE FOR ANY DAMAGES THAT MAY ARISE FROM YOUR USE OF THIRD-PARTY CONTENT.

© 2025 Advanced Micro Devices, Inc. All rights reserved. AMD, the AMD Arrow logo, AMD CDNA, AMD ROCm, AMD Instinct, and combinations thereof are trademarks of Advanced Micro Devices, Inc. in the United States and/or other jurisdictions. Other names are for informational purposes only and may be trademarks of their respective owners.

Git and the Git logo are either registered trademarks or trademarks of Software Freedom Conservancy, Inc., corporate home of the Git Project, in the United States and/or other countries
Linux is the registered trademark of Linus Torvalds in the U.S. and other countries.


AMD