

CuPy and MPI4Py: Overview of Installation and Operation

Presenter: Bob Robey
May 5th, 2025
AMD @ HLRS

AMD 
together we advance_

CuPy

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](#)

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
<code>numpy.DataSource</code>	<code>cupy.DataSource</code> (alias of <code>numpy.DataSource</code>)
<code>numpy.ScalarType</code>	-
<code>numpy.abs</code>	<code>cupy.abs</code>
<code>numpy.absolute</code>	<code>cupy.absolute</code>
<code>numpy.add</code>	<code>cupy.add</code>
<code>numpy.all</code>	<code>cupy.all</code>
<code>numpy.allclose</code>	<code>cupy.allclose</code>

full list here: [cupy documentation](#)

CuPy-specific functions

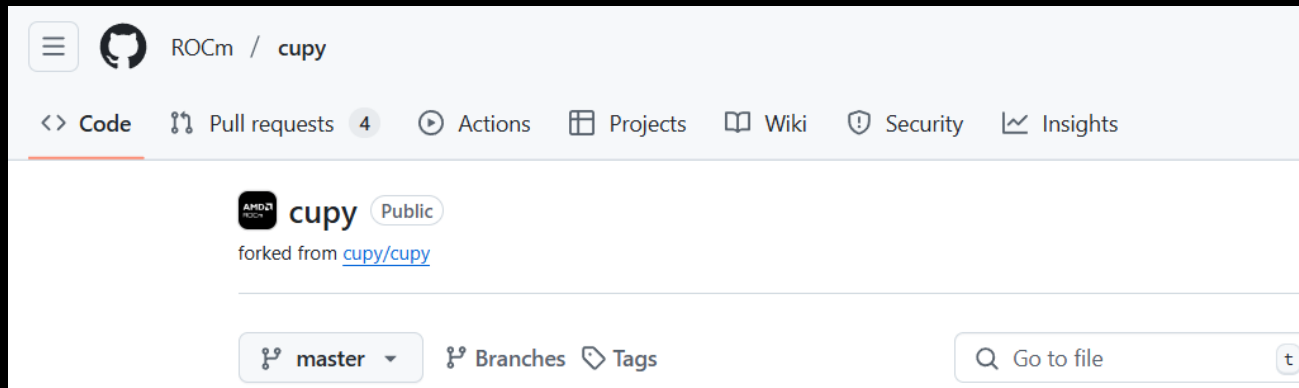
CuPy-specific functions are placed under `cupyx` namespace.

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

CuPy - Installation

- Despite its name, CuPy supports AMD GPUs as well.
- ROCm changes have been posted to the upstream repository, but not all of them have been integrated into the main CuPy repository – <https://github.com/cupy/cupy>
- There is a fork of the CuPy repository with changes being tested for pushing upstream at <https://github.com/rocm/cupy>



CuPy - Installation

Installation from source script available in our model installation repository:

https://github.com/amd/HPCTrainingDock/blob/main/extras/scripts/cupy_setup.sh

... most relevant part reported below...

```
export CUPY_INSTALL_USE_HIP=1
export ROCM_HOME=${ROCM_PATH}
export HCC_AMDGPU_ARCH=${AMDGPU_GFXMODEL}
# Get source from the ROCm repository of CuPy.
git clone -q --depth 1 --recursive https://github.com/ROCm/cupy.git
cd cupy
git reset --hard 9cdf1737eaa44aba657cb17f7e0cc421d7cca34 (this is currently the head of the master branch)
python3 -m pip install argcomplete==1.9.4
sed -i -e '/numpy/s/1.27/1.25/' setup.py
PYTHONPATH=${CUPY_PATH}
python3 setup.py -q bdist_wheel
pip3 install -v --target=${CUPY_PATH} pytest mock
pip3 install -v --upgrade --target=${CUPY_PATH} dist/cupy-13.0.0b1-cp310-cp310-linux_x86_64.whl
```

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.asarray(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.

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](#)

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

```
python 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]

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]) ← Creates an array on the device
print("CuPy Array:", gpu_array)

# Perform operations on the GPU
gpu_array_squared = gpu_array ** 2 ← Operations occur on the GPU
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) ← Converts NumPy array to CuPy array
print("CuPy Array from NumPy:",
      gpu_array_from_cpu)

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

# Transfer result back to CPU
result_cpu = cp.asnumpy(result_gpu) ← Returns an array on the host memory from an
print("Result on CPU:", result_cpu)      arbitrary source array (device in this case)
```

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
```

```
python intro.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]
```

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](#)

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](#)

Simple CuPy-Xarray code example: a closer look

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

← Adds .copy 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) ...
```

```
copy_array = da_cp.data
```

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

```
print("check that arr_gpu and copy_array are the same with CuPy:",
      cp.allclose(copy_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 copy_array are the same with NumPy (interoperability):",
      np.allclose(copy_array, arr_gpu))
```

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

```
... (some code omitted) ...
```

```
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

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](#)

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
```

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](#)

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](#)

MPI4Py and CuPy example: Allreduce and Bcast

Find the example in our exercises repo:

https://github.com/amd/HPCTrainingExamples/blob/main/Python/mmpi4py/mmpi4py_cupy.py

```
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()`

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 $rtol=1.e-05$

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

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
```

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;
```

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

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

AMD 