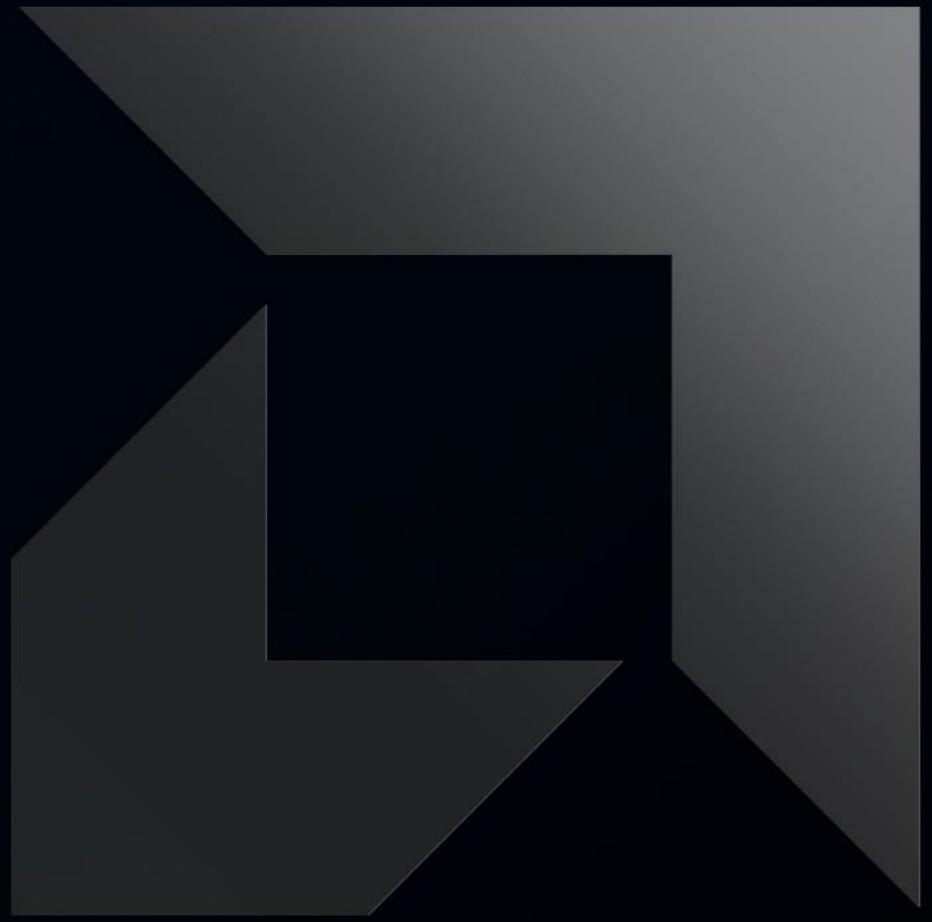


HIP Python

Presenter: Bob Robey
Oct 14, 2025
AMD @ CASTIEL

AMD
together we advance_

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 also a gateway capability that provides an entry point to
  - Numba HIP
- HIP Python is interoperable with CuPy and Numba HIP device array types
- HIP Python accepts Python buffers (bytes, bytearray, array.array, Numpy arrays, …) as input for host data
- 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 !

Installing HIP Python

- HIP Python is already pre-installed on the Training system.
- Installing on your system
  - HIP Python is currently* posted to the TestPyPI site
    - https://test.pypi.org/simple/hip-python
  - *) Eventually, we will distribute it via an *.amd.com URL
  - **) 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
- Typical install command

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

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.

HIP Python Examples

1. Error Checking
2. Getting Device Properties and Attributes
3. HIP streams and passing Python data to HIP calls
4. Calling hipBLAS
5. Calling hipFFT
6. Calling RCCL
7. Cython example
8. 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

1. Error checking & HIP runtime log

- 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]           # extract error code
    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.

- **Logging:** As HIP Python calls into the HIP runtime (and other HIP/ROCm libraries), you can use the usual HIP logging utilities, e.g. `export AMD_LOG_LEVEL=3`.

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

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

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

- hipDeviceGetAttribute

```
from hip import hip
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}")
```

3. HIP Streams and passing Python arrays to HIP calls

```
import ctypes
import random
import array
from hip import hip

n = 100
x_h = array.array("i",[int(random.random()*10) for i in range(0,n)])
num_bytes = x_h.itemsize * len(x_h)
x_d = hip.hipMalloc(num_bytes)

stream = hip.hipStreamCreate()
hip.hipMemcpyAsync(x_d,x_h,num_bytes,hip.hipMemcpyKind.hipMemcpyHostToDevice,stream)
hip.hipMemsetAsync(x_d,0,num_bytes,stream)
hip.hipMemcpyAsync(x_h,x_d,num_bytes,hip.hipMemcpyKind.hipMemcpyDeviceToHost,stream)
hip.hipStreamSynchronize(stream)
hip.hipStreamDestroy(stream)

# deallocate device data
hip.hipFree(x_d)

for i,x in enumerate(x_h):
    if x != 0:
        raise ValueError(f"expected '0' for element {i}, is: '{x}'")
```

Creating streams from Python – only 1 stream in this example

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)
```

#continues on next slide

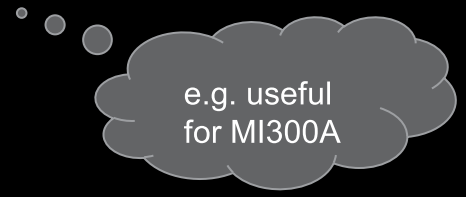
Calling hipBLAS using HIP Python

```
# 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

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

5. Calling hipFFT from Python using HIP Python

```
import numpy as np
from hip import hip, hipfft

N = 100
hx = np.zeros(N,dtype=np.cdouble)
hx[:] = 1 - 1j

dx = hip.hipMalloc(hx.size*hx.itemsize)
hip.hipMemcpy(dx, hx, dx.size, hip.hipMemcpyKind.hipMemcpyHostToDevice)

plan = hipfft.hipfftPlan1d(N, hipfft.hipfftType.HIPFFT_Z2Z, 1)

hipfft.hipfftExecZ2Z(plan, idata=dx, odata=dx, direction=hipfft.HIPFFT_FORWARD)
hip.hipDeviceSynchronize()

hip.hipMemcpy(hx,dx,dx.size,hip.hipMemcpyKind.hipMemcpyDeviceToHost)
hip.hipFree(dx)

if not np.isclose(hx[0].real,N) or not np.isclose(hx[0].imag,-N):
    raise RuntimeError("element 0 must be '{N}-j{N}'.")
for i in range(1,N):
  if not np.isclose(abs(hx[i]),0):
      raise RuntimeError(f"element {i} must be '0'")

hipfft.hipfftDestroy(plan)
```

Initialize data on host

Allocate and copy host data to device

Create FFT plan

Execute FFT plan

Copy to host and free device data

Destroy FFT plan

5a. Unified Shared Memory version of hipFFT using HIP Python

```
import numpy as np
from hip import hip, hipfft

N = 100
hx = np.zeros(N,dtype=np.cdouble)
hx[:] = 1 - 1j

plan = hipfft.hipfftPlan1d(N, hipfft.hipfftType.HIPFFT_Z2Z, 1)

hipfft.hipfftExecZ2Z(plan, idata=hx, odata=hx, direction=hipfft.HIPFFT_FORWARD)
hip.hipDeviceSynchronize()

if not np.isclose(hx[0].real,N) or not np.isclose(hx[0].imag,-N):
     raise RuntimeError("element 0 must be '{N}-j{N}'.")
for i in range(1,N):
   if not np.isclose(abs(hx[i]),0):
        raise RuntimeError(f"element {i} must be '0'")

hipfft.hipfftDestroy(plan)
print("ok")
```

Initialize data on host

Create FFT plan

Execute FFT plan

Destroy FFT plan

6. Calling RCCL from Python using HIP Python

```
import numpy as np
from hip import hip, rccl

num_gpus = hip.hipGetDeviceCount()
comms = np.empty(num_gpus,dtype="uint64") # size of pointer type, such as ncclComm
devlist = np.array(range(0,num_gpus),dtype="int32")
rccl.ncclCommInitAll(comms, num_gpus, devlist)

N = 4
ones = np.ones(N,dtype="int32")
zeros = np.zeros(ones.size,dtype="int32")
dxlist = []
for dev in devlist:
    hip.hipSetDevice(dev)
    dx = hip.hipMalloc(ones.size*ones.itemsize) # items are bytes
    dxlist.append(dx)
    hx = ones if dev == 0 else zeros
    hip.hipMemcpy(dx,hx,dx.size,hip.hipMemcpyKind.hipMemcpyHostToDevice)

# continued on next page
```

Initialize the communicators

Initialize data on the devices

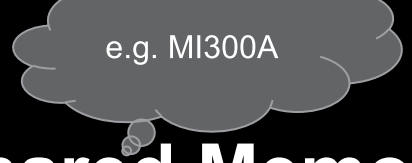
Calling RCCL from Python using HIP Python

```
rccl.ncclGroupStart()
for dev in devlist:
    hip.hipSetDevice(dev)
    rccl.ncclBcast(dxlist[dev], N, rccl.ncclDataType_t.ncclInt32, 0, int(comms[dev]), None)
    # convert to Python int so that the numpy datatype is not interpreted as single-element Py_buffer
rccl.ncclGroupEnd()

hx = np.empty(N,dtype="int32")
for dev in devlist:
    dx=dxlist[dev]
    hx[:] = 0
    hip.hipMemcpy(hx,dx,dx.size,hip.hipMemcpyKind.hipMemcpyDeviceToHost)
    for i,item in enumerate(hx):
        if item != 1:
            raise RuntimeError(f"failed for element {i}")

for dx in dxlist:
    hip.hipFree(dx)
for comm in comms:
    rccl.ncclCommDestroy(int(comm))
    # convert to Python int so that the numpy datatype is not interpreted as single-element Py_buffer
```

Perform a broadcast

Download and check the output;
confirm all entries are one

Clean up



6a. Unified Shared Memory Calling RCCL from HIP Python

```
import numpy as np
from hip import hip, rccl

num_gpus = hip.hipGetDeviceCount()
comms = np.empty(num_gpus,dtype="uint64") # size of pointer type, such as ncclComm
devlist = np.array(range(0,num_gpus),dtype="int32")
rccl.ncclCommInitAll(comms, num_gpus, devlist)

N = 4
ones = np.ones(N,dtype="int32")
zeros = np.zeros(ones.size,dtype="int32")
dxlist = []
for dev in devlist:
    hip.hipSetDevice(dev)
    hx = ones if dev == 0 else zeros
    dxlist.append(hx)

rccl.ncclGroupStart()
for dev in devlist:
    hip.hipSetDevice(dev)
    rccl.ncclBcast(dxlist[dev], N, rccl.ncclDataType_t.ncclInt32, 0, int(comms[dev]), None)
rccl.ncclGroupEnd()

# continued on next slide
```

Initialize the communicators

Initialize data on the devices

Convert to Python int so that the numpy datatype is not interpreted as single-element Py_buffer

Perform a broadcast

Unified Shared Memory Calling RCCL from HIP Python

```
hx = np.empty(N,dtype="int32")
for dev in devlist:
    hx=dxlist[dev]
    for i,item in enumerate(hx):
        if item != 1:
            raise RuntimeError(f"failed for element {i}")

for comm in comms:
    rccl.ncclCommDestroy(int(comm))

print("ok")
```

Check the output; confirm all entries are one

Clean up

Convert to Python int so that the numpy datatype is not interpreted as single-element Py_buffer

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):
```

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}__"]

# continues on next slide
```

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),
    )
)
```

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
```

HIP Python Datatypes

- Datatypes are used to convert Python objects to C datatypes
  - Defined in hip._util.types and hip._hip_helpers
- Pointer (types.Pointer) – maps Python objects to C pointers
  - Handles various types ranging from NULL, pointers, values and Python array objects via addresses or handles
- DeviceArray – return value of hipMalloc and other HIP memory allocations
- List Types – for groups of options, headers, and includeNames in hiprtcCompileProgram or hiprtcCreateProgram

Numba HIP

- Numba is a Just-in-Time (JIT) compiler for Python numerical functions
- Numba HIP is a frontend extension + backend for AMD GPUs that plugs into Numba
- We have installed both as part of the HIP Python module
- If you want to experiment with Numba HIP and HIP Python on your own system, you can install both 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
```

Numba HIP example

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

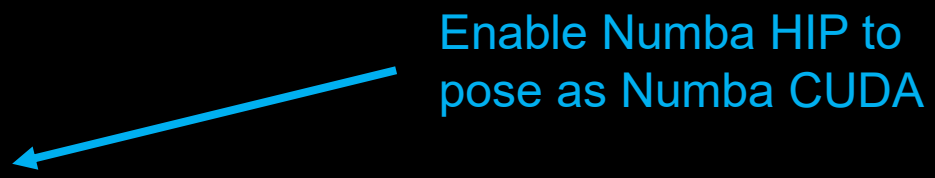
```
from numba import hip

@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
```

Numba HIP posing as Numba CUDA

- We can also have Numba HIP pose as Numba 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")
```

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.

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