# HIP (ROCm) semantics

ROCm™ is AMDâ€™s open source software platform for GPU-accelerated high performance computing and machine learning. HIP is ROCm's C++ dialect designed to ease conversion of CUDA applications to portable C++ code. HIP is used when converting existing CUDA applications like PyTorch to portable C++ and for new projects that require portability between AMD and NVIDIA.

## HIP Interfaces Reuse the CUDA Interfaces

PyTorch for HIP intentionally reuses the existing :mod:`torch.cuda` interfaces. This helps to accelerate the porting of existing PyTorch code and models because very few code changes are necessary, if any.

**System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\[pytorch-master][docs][source][notes]hip.rst`, line 17); *backlink***

Unknown interpreted text role "mod".

The example from :ref:`cuda-semantics` will work exactly the same for HIP:

**System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\[pytorch-master][docs][source][notes]hip.rst`, line 21); *backlink***

Unknown interpreted text role "ref".

```
cuda = torch.device('cuda')     # Default HIP device
cuda0 = torch.device('cuda:0')  # 'rocm' or 'hip' are not valid, use 'cuda'
cuda2 = torch.device('cuda:2')  # GPU 2 (these are 0-indexed)

x = torch.tensor([1., 2.], device=cuda0)
# x.device is device(type='cuda', index=0)
y = torch.tensor([1., 2.]).cuda()
# y.device is device(type='cuda', index=0)

with torch.cuda.device(1):
    # allocates a tensor on GPU 1
    a = torch.tensor([1., 2.], device=cuda)

    # transfers a tensor from CPU to GPU 1
    b = torch.tensor([1., 2.]).cuda()
    # a.device and b.device are device(type='cuda', index=1)

    # You can also use ``Tensor.to`` to transfer a tensor:
    b2 = torch.tensor([1., 2.]).to(device=cuda)
    # b.device and b2.device are device(type='cuda', index=1)

    c = a + b
    # c.device is device(type='cuda', index=1)

    z = x + y
    # z.device is device(type='cuda', index=0)

    # even within a context, you can specify the device
    # (or give a GPU index to the .cuda call)
    d = torch.randn(2, device=cuda2)
    e = torch.randn(2).to(cuda2)
    f = torch.randn(2).cuda(cuda2)
    # d.device, e.device, and f.device are all device(type='cuda', index=2)
```

## Checking for HIP

Whether you are using PyTorch for CUDA or HIP, the result of calling :meth:`~torch.cuda.is_available` will be the same. If you are using a PyTorch that has been built with GPU support, it will return *True*. If you must check which version of PyTorch you are using, refer to this example below:

**System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\[pytorch-master][docs][source][notes]hip.rst`, line 62); *backlink***

Unknown interpreted text role "meth".

```
if torch.cuda.is_available() and torch.version.hip:
    # do something specific for HIP
elif torch.cuda.is_available() and torch.version.cuda:
    # do something specific for CUDA
```

## TensorFloat-32(TF32) on ROCm

TF32 is not supported on ROCm.

## Memory management

PyTorch uses a caching memory allocator to speed up memory allocations. This allows fast memory deallocation without device synchronizations. However, the unused memory managed by the allocator will still show as if used in `rocm-smi`. You can use :meth:`~torch.cuda.memory_allocated` and :meth:`~torch.cuda.max_memory_allocated` to monitor memory occupied by tensors, and use :meth:`~torch.cuda.memory_reserved` and :meth:`~torch.cuda.max_memory_reserved` to monitor the total amount of memory managed by the caching allocator. Calling :meth:`~torch.cuda.empty_cache` releases all **unused** cached memory from PyTorch so that those can be used by other GPU applications. However, the occupied GPU memory by tensors will not be freed so it can not increase the amount of GPU memory available for PyTorch.

**System Message: ERROR/3** (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\[pytorch-master][docs][source][notes]hip.rst`, line 87); *backlink*

Unknown interpreted text role "meth".

**System Message: ERROR/3** (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\[pytorch-master][docs][source][notes]hip.rst`, line 87); *backlink*

Unknown interpreted text role "meth".

**System Message: ERROR/3** (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\[pytorch-master][docs][source][notes]hip.rst`, line 87); *backlink*

Unknown interpreted text role "meth".

**System Message: ERROR/3** (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\[pytorch-master][docs][source][notes]hip.rst`, line 87); *backlink*

Unknown interpreted text role "meth".

**System Message: ERROR/3** (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\[pytorch-master][docs][source][notes]hip.rst`, line 87); *backlink*

Unknown interpreted text role "meth".

For more advanced users, we offer more comprehensive memory benchmarking via :meth:`~torch.cuda.memory_stats`. We also offer the capability to capture a complete snapshot of the memory allocator state via :meth:`~torch.cuda.memory_snapshot`, which can help you understand the underlying allocation patterns produced by your code.

**System Message: ERROR/3** (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\[pytorch-master][docs][source][notes]hip.rst`, line 99); *backlink*

Unknown interpreted text role "meth".

**System Message: ERROR/3** (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\[pytorch-master][docs][source][notes]hip.rst`, line 99); *backlink*

Unknown interpreted text role "meth".

To debug memory errors, set `PYTORCH_NO_CUDA_MEMORY_CACHING=1` in your environment to disable caching.

## hipFFT/rocFFT plan cache

Setting the size of the cache for hipFFT/rocFFT plans is not supported.

## torch.distributed backends

Currently, only the "nccl" and "gloo" backends for torch.distributed are supported on ROCm.

## CUDA API to HIP API mappings in C++

Please refer: https://rocmdocs.amd.com/en/latest/Programming_Guides/HIP_API_Guide.html

NOTE: The CUDA_VERSION macro, cudaRuntimeGetVersion and cudaDriverGetVersion APIs do not semantically map to the same values as HIP_VERSION macro, hipRuntimeGetVersion and hipDriverGetVersion APIs. Please do not use them interchangeably when doing version checks.

Eg: Instead of #if defined(CUDA_VERSION) && CUDA_VERSION >= 11000 If it is desired to not take the code path for ROCm/HIP: #if defined(CUDA_VERSION) && CUDA_VERSION >= 11000 && !defined(USE_ROCM) If it is desired to take the code path for ROCm/HIP: #if (defined(CUDA_VERSION) && CUDA_VERSION >= 11000) || defined(USE_ROCM) If it is desired to take the code path for ROCm/HIP only for specific HIP versions: #if (defined(CUDA_VERSION) && CUDA_VERSION >= 11000) || (defined(USE_ROCM) && ROCM_VERSION >= 40300)

## Refer to CUDA Semantics doc

For any sections not listed here, please refer to the CUDA semantics doc: :ref:`cuda-semantics`

**System Message: ERROR/3** (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\[pytorch-master][docs][source][notes]hip.rst`, **line 146**); ***backlink***

Unknown interpreted text role "ref".