CUDA semantics

:mod:`torch.cuda` is used to set up and run CUDA operations. It keeps track of the currently selected GPU, and all CUDA tensors you allocate will by default be created on that device. The selected device can be changed with a :any:`torch.cuda.device` context manager.

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\(pytorch-master) (docs) (source) (notes)cuda.rst`, line 6); *backlink*

Unknown interpreted text role "mod".

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\(pytorch-master) (docs) (source) (notes)cuda.rst`, line 6); *backlink*

Unknown interpreted text role "any".

However, once a tensor is allocated, you can do operations on it irrespective of the selected device, and the results will be always placed in on the same device as the tensor.

Cross-GPU operations are not allowed by default, with the exception of :meth:`~torch.Tensor.copy_` and other methods with copy-like functionality such as :meth:`~torch.Tensor.to` and :meth:`~torch.Tensor.cuda`. Unless you enable peer-to-peer memory access, any attempts to launch ops on tensors spread across different devices will raise an error.

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\(pytorch-master) (docs) (source) (notes)cuda.rst`, line 15); *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)cuda.rst`, line 15); *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)cuda.rst`, line 15); *backlink*

Unknown interpreted text role "meth".

Below you can find a small example showcasing this:

```
cuda = torch.device('cuda')     # Default CUDA device
cuda0 = torch.device('cuda:0')
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)
```

TensorFloat-32(TF32) on Ampere devices

Starting in PyTorch 1.7, there is a new flag called *allow_tf32* which defaults to true. This flag controls whether PyTorch is allowed to use the TensorFloat32 (TF32) tensor cores, available on new NVIDIA GPUs since Ampere, internally to compute matmul (matrix multiplies and batched matrix multiplies) and convolutions.

TF32 tensor cores are designed to achieve better performance on matmul and convolutions on *torch.float32* tensors by rounding input data to have 10 bits of mantissa, and accumulating results with FP32 precision, maintaining FP32 dynamic range.

matmuls and convolutions are controlled separately, and their corresponding flags can be accessed at:

```
# The flag below controls whether to allow TF32 on matmul. This flag defaults to True.
torch.backends.cuda.matmul.allow_tf32 = True

# The flag below controls whether to allow TF32 on cuDNN. This flag defaults to True.
torch.backends.cudnn.allow_tf32 = True
```

Note that besides matmuls and convolutions themselves, functions and nn modules that internally uses matmuls or convolutions are also affected. These include *nn.Linear*, *nn.Conv**, cdist, tensordot, affine grid and grid sample, adaptive log softmax, GRU and LSTM.

To get an idea of the precision and speed, see the example code below:

```
a_full = torch.randn(10240, 10240, dtype=torch.double, device='cuda')
b_full = torch.randn(10240, 10240, dtype=torch.double, device='cuda')
ab_full = a_full @ b_full
mean = ab_full.abs().mean()  # 80.7277

a = a_full.float()
b = b_full.float()

# Do matmul at TF32 mode.
ab_tf32 = a @ b  # takes 0.016s on GA100
error = (ab_tf32 - ab_full).abs().max()  # 0.1747
relative_error = error / mean  # 0.0022

# Do matmul with TF32 disabled.
torch.backends.cuda.matmul.allow_tf32 = False
ab_fp32 = a @ b  # takes 0.11s on GA100
error = (ab_fp32 - ab_full).abs().max()  # 0.0031
relative_error = error / mean  # 0.000039
```

From the above example, we can see that with TF32 enabled, the speed is ~7x faster, relative error compared to double precision is approximately 2 orders of magnitude larger. If the full FP32 precision is needed, users can disable TF32 by:

```
torch.backends.cuda.matmul.allow_tf32 = False
torch.backends.cudnn.allow_tf32 = False
```

To toggle the TF32 flags off in C++, you can do

```
at::globalContext().setAllowTF32CuBLAS(false);
at::globalContext().setAllowTF32CuDNN(false);
```

For more information about TF32, see:

- TensorFloat-32
- CUDA 11
- Ampere architecture

Reduced Precision Reduction in FP16 GEMMs

fp16 GEMMs are potentially done with some intermediate reduced precision reductions (e.g., in fp16 rather than fp32). These selective reductions in precision can allow for higher performance on certain workloads (particularly those with a large *k* dimension) and GPU architectures at the cost of numerical precision and potential for overflow.

Some example benchmark data on V100:

```
[--------------------------- bench_gemm_transformer --------------------------]
      [  m ,  k  ,  n  ]    |  allow_fp16_reduc=True  |  allow_fp16_reduc=False
1 threads: ---------------------------------------------------------------------
      [4096, 4048, 4096]    |          1634.6         |          1639.8
      [4096, 4056, 4096]    |          1670.8         |          1661.9
      [4096, 4080, 4096]    |          1664.2         |          1658.3
```

```
      [4096, 4096, 4096]    |          1639.4        |          1651.0
      [4096, 4104, 4096]    |          1677.4        |          1674.9
      [4096, 4128, 4096]    |          1655.7        |          1646.0
      [4096, 4144, 4096]    |          1796.8        |          2519.6
      [4096, 5096, 4096]    |          2094.6        |          3190.0
      [4096, 5104, 4096]    |          2144.0        |          2663.5
      [4096, 5112, 4096]    |          2149.1        |          2766.9
      [4096, 5120, 4096]    |          2142.8        |          2631.0
      [4096, 9728, 4096]    |          3875.1        |          5779.8
      [4096, 16384, 4096]   |          6182.9        |          9656.5
(times in microseconds).
```

If full precision reductions are needed, users can disable reduced precision reductions in fp16 GEMMs with:

```
torch.backends.cuda.matmul.allow_fp16_reduced_precision_reduction = False
```

To toggle the reduced precision reduction flags in C++, you can do

```
at::globalContext().setAllowFP16ReductionCuBLAS(false);
```

Asynchronous execution

By default, GPU operations are asynchronous. When you call a function that uses the GPU, the operations are *enqueued* to the particular device, but not necessarily executed until later. This allows us to execute more computations in parallel, including operations on CPU or other GPUs.

In general, the effect of asynchronous computation is invisible to the caller, because (1) each device executes operations in the order they are queued, and (2) PyTorch automatically performs necessary synchronization when copying data between CPU and GPU or between two GPUs. Hence, computation will proceed as if every operation was executed synchronously.

You can force synchronous computation by setting environment variable `CUDA_LAUNCH_BLOCKING=1`. This can be handy when an error occurs on the GPU. (With asynchronous execution, such an error isn't reported until after the operation is actually executed, so the stack trace does not show where it was requested.)

A consequence of the asynchronous computation is that time measurements without synchronizations are not accurate. To get precise measurements, one should either call :func:`torch.cuda.synchronize()` before measuring, or use :class:`torch.cuda.Event` to record times as following:

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\(pytorch-master)(docs)(source)(notes)cuda.rst, line 195); *backlink*

Unknown interpreted text role "func".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\(pytorch-master)(docs)(source)(notes)cuda.rst, line 195); *backlink*

Unknown interpreted text role "class".

```
start_event = torch.cuda.Event(enable_timing=True)
end_event = torch.cuda.Event(enable_timing=True)
start_event.record()

# Run some things here

end_event.record()
torch.cuda.synchronize()  # Wait for the events to be recorded!
elapsed_time_ms = start_event.elapsed_time(end_event)
```

As an exception, several functions such as :meth:`~torch.Tensor.to` and :meth:`~torch.Tensor.copy_` admit an explicit :attr:`non_blocking` argument, which lets the caller bypass synchronization when it is unnecessary. Another exception is CUDA streams, explained below.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\(pytorch-master)(docs)(source)(notes)cuda.rst, line 210); *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)cuda.rst, line 210); *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)cuda.rst, line 210); *backlink*

Unknown interpreted text role "attr".

CUDA streams

A CUDA stream is a linear sequence of execution that belongs to a specific device. You normally do not need to create one explicitly: by default, each device uses its own "default" stream.

Operations inside each stream are serialized in the order they are created, but operations from different streams can execute concurrently in any relative order, unless explicit synchronization functions (such as :meth:`~torch.cuda.synchronize` or :meth:`~torch.cuda.Stream.wait_stream`) are used. For example, the following code is incorrect:

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\(pytorch-master)(docs)(source)(notes)cuda.rst, line 222); *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)cuda.rst, line 222); *backlink*

Unknown interpreted text role "meth".

```
cuda = torch.device('cuda')
s = torch.cuda.Stream()  # Create a new stream.
A = torch.empty((100, 100), device=cuda).normal_(0.0, 1.0)
with torch.cuda.stream(s):
    # sum() may start execution before normal_() finishes!
    B = torch.sum(A)
```

When the "current stream" is the default stream, PyTorch automatically performs necessary synchronization when data is moved around, as explained above. However, when using non-default streams, it is the user's responsibility to ensure proper synchronization.

Stream semantics of backward passes

Each backward CUDA op runs on the same stream that was used for its corresponding forward op. If your forward pass runs independent ops in parallel on different streams, this helps the backward pass exploit that same parallelism.

The stream semantics of a backward call with respect to surrounding ops are the same as for any other call. The backward pass inserts internal syncs to ensure this even when backward ops run on multiple streams as described in the previous paragraph. More concretely, when calling :func:`autograd.backward<torch.autograd.backward>`, :func:`autograd.grad<torch.autograd.grad>`, or :meth:`tensor.backward<torch.Tensor.backward>`, and optionally supplying CUDA tensor(s) as the initial gradient(s) (e.g., :func:`autograd.backward(..., grad_tensors=initial_grads)<torch.autograd.backward>`, :func:`autograd.grad(..., grad_outputs=initial_grads)<torch.autograd.grad>`, or :meth:`tensor.backward(..., gradient=initial_grad)<torch.Tensor.backward>`), the acts of

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\(pytorch-master)(docs)(source)(notes)cuda.rst, line 249); *backlink*

Unknown interpreted text role "func".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\(pytorch-master)(docs)(source)(notes)cuda.rst, line 249); *backlink*

Unknown interpreted text role "func".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\(pytorch-master)(docs)(source)(notes)cuda.rst, line 249); *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)cuda.rst, line 249); *backlink*

Unknown interpreted text role "func".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\(pytorch-master)(docs)(source)(notes)cuda.rst, line 249); *backlink*

Unknown interpreted text role "func".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\(pytorch-master)(docs)(source)(notes)cuda.rst, line 249); *backlink*

Unknown interpreted text role "meth".

1. optionally populating initial gradient(s),
2. invoking the backward pass, and
3. using the gradients

have the same stream-semantics relationship as any group of ops:

```
s = torch.cuda.Stream()

# Safe, grads are used in the same stream context as backward()
with torch.cuda.stream(s):
    loss.backward()
    use grads

# Unsafe
with torch.cuda.stream(s):
    loss.backward()
use grads

# Safe, with synchronization
with torch.cuda.stream(s):
    loss.backward()
torch.cuda.current_stream().wait_stream(s)
use grads

# Safe, populating initial grad and invoking backward are in the same stream context
with torch.cuda.stream(s):
    loss.backward(gradient=torch.ones_like(loss))

# Unsafe, populating initial_grad and invoking backward are in different stream contexts,
# without synchronization
initial_grad = torch.ones_like(loss)
with torch.cuda.stream(s):
    loss.backward(gradient=initial_grad)

# Safe, with synchronization
initial_grad = torch.ones_like(loss)
s.wait_stream(torch.cuda.current_stream())
with torch.cuda.stream(s):
    initial_grad.record_stream(s)
    loss.backward(gradient=initial_grad)
```

BC note: Using grads on the default stream

In prior versions of PyTorch (1.9 and earlier), the autograd engine always synced the default stream with all backward ops, so the following pattern:

```
with torch.cuda.stream(s):
    loss.backward()
use grads
```

was safe as long as `use grads` happened on the default stream. In present PyTorch, that pattern is no longer safe. If `backward()` and `use grads` are in different stream contexts, you must sync the streams:

```
with torch.cuda.stream(s):
    loss.backward()
torch.cuda.current_stream().wait_stream(s)
use grads
```

even if `use grads` is on the default stream.

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 `nvidia-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)cuda.rst, line 331); *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)cuda.rst, line 331); *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)cuda.rst, line 331); *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)cuda.rst, line 331); *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)cuda.rst, line 343); *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)cuda.rst, line 343); *backlink*

Unknown interpreted text role "meth".

Use of a caching allocator can interfere with memory checking tools such as `cuda-memcheck`. To debug memory errors using `cuda-memcheck`, set `PYTORCH_NO_CUDA_MEMORY_CACHING=1` in your environment to disable caching.

The behavior of caching allocator can be controlled via environment variable `PYTORCH_CUDA_ALLOC_CONF`. The format is `PYTORCH_CUDA_ALLOC_CONF=<option>:<value>,<option2><value2>...` Available options:

- `max_split_size_mb` prevents the allocator from splitting blocks larger than this size (in MB). This can help prevent fragmentation and may allow some borderline workloads to complete without running out of memory. Performance cost can range from 'zero' to 'substatial' depending on allocation patterns. Default value is unlimited, i.e. all blocks can be split. The :meth:`~torch.cuda.memory_stats` and :meth:`~torch.cuda.memory_summary` methods are useful for tuning. This option should be used as a last resort for a workload that is aborting due to 'out of memory' and showing a large amount of inactive split blocks.

  System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\(pytorch-master)(docs)(source)(notes)cuda.rst, line 358); *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)cuda.rst, line 358); *backlink*

  Unknown interpreted text role "meth".

- `roundup_power2_divisions` helps with rounding the requested allocation size to nearest power-2 division and making better use of the blocks. In the current CUDACachingAllocator, the sizes are rounded up in multiple of blocks size of 512, so this works fine for smaller sizes. However, this can be inefficient for large near-by allocations as each will go to different size of blocks and re-use of those blocks are minimized. This might create lots of unused blocks and will waste GPU memory capacity. This option enables the rounding of allocation size to nearest power-2 division. For example, if we need to round-up size of 1200 and if number of divisions is 4, the size 1200 lies between 1024 and 2048 and if we do 4 divisions between them, the values are 1024, 1280, 1536, and 1792. So, allocation size of 1200 will be rounded to 1280 as the nearest ceiling of power-2 division.
- `garbage_collection_threshold` helps actively reclaiming unused GPU memory to avoid triggering expensive sync-and-reclaim-all operation (release_cached_blocks), which can be unfavorable to latency-critical GPU applications (e.g., servers). Upon setting this threshold (e.g., 0.8), the allocator will start reclaiming GPU memory blocks if the GPU memory capacity usage exceeds the threshold (i.e., 80% of the total memory allocated to the GPU application). The algorithm prefers to free old & unused blocks first to avoid freeing blocks that are actively being reused. The threshold value should be between greater than 0.0 and less than 1.0.

cuFFT plan cache

For each CUDA device, an LRU cache of cuFFT plans is used to speed up repeatedly running FFT methods (e.g., :func:`torch.fft.fft`) on CUDA tensors of same geometry with same configuration. Because some cuFFT plans may allocate GPU memory, these caches have a maximum capacity.

> **System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\(pytorch-master)(docs)(source)(notes)cuda.rst, line 393); *backlink***
>
> Unknown interpreted text role "func".

You may control and query the properties of the cache of current device with the following APIs:

- `torch.backends.cuda.cufft_plan_cache.max_size` gives the capacity of the cache (default is 4096 on CUDA 10 and newer, and 1023 on older CUDA versions). Setting this value directly modifies the capacity.
- `torch.backends.cuda.cufft_plan_cache.size` gives the number of plans currently residing in the cache.
- `torch.backends.cuda.cufft_plan_cache.clear()` clears the cache.

To control and query plan caches of a non-default device, you can index the `torch.backends.cuda.cufft_plan_cache` object with either a :class:`torch.device` object or a device index, and access one of the above attributes. E.g., to set the capacity of the cache for device `1`, one can write `torch.backends.cuda.cufft_plan_cache[1].max_size = 10`.

> **System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\(pytorch-master)(docs)(source)(notes)cuda.rst, line 410); *backlink***
>
> Unknown interpreted text role "class".

Just-in-Time Compilation

PyTorch just-in-time compiles some operations, like torch.special.zeta, when performed on CUDA tensors. This compilation can be time consuming (up to a few seconds depending on your hardware and software) and may occur multiple times for a single operator since many PyTorch operators actually select from a variety of kernels, each of which must be compiled once, depending on their input. This compilation occurs once per process, or just once if a kernel cache is used.

By default, PyTorch creates a kernel cache in $XDG_CACHE_HOME/torch/kernels if XDG_CACHE_HOME is defined and $HOME/.cache/torch/kernels if it's not (except on Windows, where the kernel cache is not yet supported). The caching behavior can be directly controlled with two environment variables. If USE_PYTORCH_KERNEL_CACHE is set to 0 then no cache will be used, and if PYTORCH_KERNEL_CACHE_PATH is set then that path will be used as a kernel cache instead of the default location.

Best practices

Device-agnostic code

Due to the structure of PyTorch, you may need to explicitly write device-agnostic (CPU or GPU) code; an example may be creating a new tensor as the initial hidden state of a recurrent neural network.

The first step is to determine whether the GPU should be used or not. A common pattern is to use Python's `argparse` module to read in user arguments, and have a flag that can be used to disable CUDA, in combination with :meth:`~torch.cuda.is_available`. In the following, `args.device` results in a :class:`torch.device` object that can be used to move tensors to CPU or CUDA.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\(pytorch-master)(docs)(source)(notes)cuda.rst, line 445); *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)cuda.rst, line 445); *backlink*

Unknown interpreted text role "class".

```
import argparse
import torch

parser = argparse.ArgumentParser(description='PyTorch Example')
parser.add_argument('--disable-cuda', action='store_true',
                    help='Disable CUDA')
args = parser.parse_args()
args.device = None
if not args.disable_cuda and torch.cuda.is_available():
    args.device = torch.device('cuda')
else:
    args.device = torch.device('cpu')
```

Now that we have `args.device`, we can use it to create a Tensor on the desired device.

```
x = torch.empty((8, 42), device=args.device)
net = Network().to(device=args.device)
```

This can be used in a number of cases to produce device agnostic code. Below is an example when using a dataloader:

```
cuda0 = torch.device('cuda:0')  # CUDA GPU 0
for i, x in enumerate(train_loader):
    x = x.to(cuda0)
```

When working with multiple GPUs on a system, you can use the `CUDA_VISIBLE_DEVICES` environment flag to manage which GPUs are available to PyTorch. As mentioned above, to manually control which GPU a tensor is created on, the best practice is to use a :any:`torch.cuda.device` context manager.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\(pytorch-master)(docs)(source)(notes)cuda.rst, line 483); *backlink*

Unknown interpreted text role "any".

```
print("Outside device is 0")  # On device 0 (default in most scenarios)
with torch.cuda.device(1):
    print("Inside device is 1")  # On device 1
print("Outside device is still 0")  # On device 0
```

If you have a tensor and would like to create a new tensor of the same type on the same device, then you can use a `torch.Tensor.new_*` method (see :class:`torch.Tensor`). Whilst the previously mentioned `torch.*` factory functions (:ref:`tensor-creation-ops`) depend on the current GPU context and the attributes arguments you pass in, `torch.Tensor.new_*` methods preserve the device and other attributes of the tensor.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\(pytorch-master)(docs)(source)(notes)cuda.rst, line 495); *backlink*

Unknown interpreted text role "class".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\(pytorch-master)(docs)(source)(notes)cuda.rst, line 495); *backlink*

Unknown interpreted text role "ref".

This is the recommended practice when creating modules in which new tensors need to be created internally during the forward pass.

```
cuda = torch.device('cuda')
x_cpu = torch.empty(2)
x_gpu = torch.empty(2, device=cuda)
x_cpu_long = torch.empty(2, dtype=torch.int64)

y_cpu = x_cpu.new_full([3, 2], fill_value=0.3)
print(y_cpu)
```

```
        tensor([[ 0.3000,  0.3000],
                [ 0.3000,  0.3000],
                [ 0.3000,  0.3000]])

y_gpu = x_gpu.new_full([3, 2], fill_value=-5)
print(y_gpu)

        tensor([[-5.0000, -5.0000],
                [-5.0000, -5.0000],
                [-5.0000, -5.0000]], device='cuda:0')

y_cpu_long = x_cpu_long.new_tensor([[1, 2, 3]])
print(y_cpu_long)

        tensor([[ 1,  2,  3]])
```

If you want to create a tensor of the same type and size of another tensor, and fill it with either ones or zeros, :meth:`~torch.ones_like` or :meth:`~torch.zeros_like` are provided as convenient helper functions (which also preserve :class:`torch.device` and :class:`torch.dtype` of a Tensor).

> **System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\(pytorch-master)(docs)(source)(notes)cuda.rst, line 533); *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)cuda.rst, line 533); *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)cuda.rst, line 533); *backlink***
>
> Unknown interpreted text role "class".

> **System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\(pytorch-master)(docs)(source)(notes)cuda.rst, line 533); *backlink***
>
> Unknown interpreted text role "class".

```
x_cpu = torch.empty(2, 3)
x_gpu = torch.empty(2, 3)

y_cpu = torch.ones_like(x_cpu)
y_gpu = torch.zeros_like(x_gpu)
```

Use pinned memory buffers

> **Warning**
>
> This is an advanced tip. If you overuse pinned memory, it can cause serious problems when running low on RAM, and you should be aware that pinning is often an expensive operation.

Host to GPU copies are much faster when they originate from pinned (page-locked) memory. CPU tensors and storages expose a :meth:`~torch.Tensor.pin_memory` method, that returns a copy of the object, with data put in a pinned region.

> **System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\(pytorch-master)(docs)(source)(notes)cuda.rst, line 557); *backlink***
>
> Unknown interpreted text role "meth".

Also, once you pin a tensor or storage, you can use asynchronous GPU copies. Just pass an additional `non_blocking=True` argument to a :meth:`~torch.Tensor.to` or a :meth:`~torch.Tensor.cuda` call. This can be used to overlap data transfers with computation.

> **System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\(pytorch-master)(docs)(source)(notes)cuda.rst, line 561); *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)cuda.rst, line 561); *backlink*

Unknown interpreted text role "meth".

You can make the :class:`~torch.utils.data.DataLoader` return batches placed in pinned memory by passing `pin_memory=True` to its constructor.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\(pytorch-master)(docs)(source)(notes)cuda.rst, line 566); *backlink*

Unknown interpreted text role "class".

Use nn.parallel.DistributedDataParallel instead of multiprocessing or nn.DataParallel

Most use cases involving batched inputs and multiple GPUs should default to using :class:`~torch.nn.parallel.DistributedDataParallel` to utilize more than one GPU.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\(pytorch-master)(docs)(source)(notes)cuda.rst, line 574); *backlink*

Unknown interpreted text role "class".

There are significant caveats to using CUDA models with :mod:`~torch.multiprocessing`; unless care is taken to meet the data handling requirements exactly, it is likely that your program will have incorrect or undefined behavior.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\(pytorch-master)(docs)(source)(notes)cuda.rst, line 578); *backlink*

Unknown interpreted text role "mod".

It is recommended to use :class:`~torch.nn.parallel.DistributedDataParallel`, instead of :class:`~torch.nn.DataParallel` to do multi-GPU training, even if there is only a single node.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\(pytorch-master)(docs)(source)(notes)cuda.rst, line 583); *backlink*

Unknown interpreted text role "class".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\(pytorch-master)(docs)(source)(notes)cuda.rst, line 583); *backlink*

Unknown interpreted text role "class".

The difference between :class:`~torch.nn.parallel.DistributedDataParallel` and :class:`~torch.nn.DataParallel` is: :class:`~torch.nn.parallel.DistributedDataParallel` uses multiprocessing where a process is created for each GPU, while :class:`~torch.nn.DataParallel` uses multithreading. By using multiprocessing, each GPU has its dedicated process, this avoids the performance overhead caused by GIL of Python interpreter.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\(pytorch-master)(docs)(source)(notes)cuda.rst, line 587); *backlink*

Unknown interpreted text role "class".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\(pytorch-master)(docs)(source)(notes)cuda.rst, line 587); *backlink*

Unknown interpreted text role "class".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\(pytorch-master)(docs)(source)(notes)cuda.rst, line 587); *backlink*

Unknown interpreted text role "class".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\(pytorch-master)(docs)(source)(notes)cuda.rst, line 587); *backlink*

Unknown interpreted text role "class".

If you use :class:`~torch.nn.parallel.DistributedDataParallel`, you could use *torch.distributed.launch* utility to launch your program, see :ref:`distributed-launch`.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\(pytorch-master)(docs)(source)(notes)cuda.rst, line 594); *backlink*

Unknown interpreted text role "class".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\(pytorch-master)(docs)(source)(notes)cuda.rst, line 594); *backlink*

Unknown interpreted text role "ref".

CUDA Graphs

A CUDA graph is a record of the work (mostly kernels and their arguments) that a CUDA stream and its dependent streams perform. For general principles and details on the underlying CUDA API, see Getting Started with CUDA Graphs and the Graphs section of the CUDA C Programming Guide.

PyTorch supports the construction of CUDA graphs using stream capture, which puts a CUDA stream in *capture mode*. CUDA work issued to a capturing stream doesn't actually run on the GPU. Instead, the work is recorded in a graph.

After capture, the graph can be *launched* to run the GPU work as many times as needed. Each replay runs the same kernels with the same arguments. For pointer arguments this means the same memory addresses are used. By filling input memory with new data (e.g., from a new batch) before each replay, you can rerun the same work on new data.

Why CUDA Graphs?

Replaying a graph sacrifices the dynamic flexibility of typical eager execution in exchange for **greatly reduced CPU overhead**. A graph's arguments and kernels are fixed, so a graph replay skips all layers of argument setup and kernel dispatch, including Python, C++, and CUDA driver overheads. Under the hood, a replay submits the entire graph's work to the GPU with a single call to cudaGraphLaunch. Kernels in a replay also execute slightly faster on the GPU, but eliding CPU overhead is the main benefit.

You should try CUDA graphs if all or part of your network is graph-safe (usually this means static shapes and static control flow, but see the other :ref:`constraints<capture-constraints>`) and you suspect its runtime is at least somewhat CPU-limited.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\(pytorch-master)(docs)(source)(notes)cuda.rst, line 628); *backlink*

Unknown interpreted text role "ref".

PyTorch API

Warning

This API is in beta and may change in future releases.

PyTorch exposes graphs via a raw :class:`torch.cuda.CUDAGraph` class and two convenience wrappers, :class:`torch.cuda.graph` and :class:`torch.cuda.make_graphed_callables`.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\(pytorch-master)(docs)(source)(notes)cuda.rst, line 647); *backlink*

Unknown interpreted text role "class".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\(pytorch-master)(docs)(source)(notes)cuda.rst, line 647); *backlink*

Unknown interpreted text role "class".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\(pytorch-master)(docs)(source)(notes)cuda.rst, line 647); *backlink*

Unknown interpreted text role "class".

:class:`torch.cuda.graph` is a simple, versatile context manager that captures CUDA work in its context. Before capture, warm up the workload to be captured by running a few eager iterations. Warmup must occur on a side stream. Because the graph reads from and writes to the same memory addresses in every replay, you must maintain long-lived references to tensors that hold input and output data during capture. To run the graph on new input data, copy new data to the capture's input tensor(s), replay the graph, then read the new output from the capture's output tensor(s). Example:

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\(pytorch-master) (docs) (source) (notes) cuda.rst`, **line 652**); ***backlink***

Unknown interpreted text role "class".

```
g = torch.cuda.CUDAGraph()

# Placeholder input used for capture
static_input = torch.empty((5,), device="cuda")

# Warmup before capture
s = torch.cuda.Stream()
s.wait_stream(torch.cuda.current_stream())
with torch.cuda.stream(s):
    for _ in range(3):
        static_output = static_input * 2
torch.cuda.current_stream().wait_stream(s)

# Captures the graph
# To allow capture, automatically sets a side stream as the current stream in the context
with torch.cuda.graph(g):
    static_output = static_input * 2

# Fills the graph's input memory with new data to compute on
static_input.copy_(torch.full((5,), 3, device="cuda"))
g.replay()
# static_output holds the results
print(static_output)  # full of 3 * 2 = 6

# Fills the graph's input memory with more data to compute on
static_input.copy_(torch.full((5,), 4, device="cuda"))
g.replay()
print(static_output)  # full of 4 * 2 = 8
```

See :ref:`Whole-network capture<whole-network-capture>`, :ref:`Usage with torch.cuda.amp<graphs-with-amp>`, and :ref:`Usage with multiple streams<multistream-capture>` for realistic and advanced patterns.

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\(pytorch-master) (docs) (source) (notes) cuda.rst`, **line 692**); ***backlink***

Unknown interpreted text role "ref".

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\(pytorch-master) (docs) (source) (notes) cuda.rst`, **line 692**); ***backlink***

Unknown interpreted text role "ref".

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\(pytorch-master) (docs) (source) (notes) cuda.rst`, **line 692**); ***backlink***

Unknown interpreted text role "ref".

:class:`~torch.cuda.make_graphed_callables` is more sophisticated. :class:`~torch.cuda.make_graphed_callables` accepts Python functions and :class:`torch.nn.Module`s. For each passed function or Module, it creates separate graphs of the forward-pass and backward-pass work. See :ref:`Partial-network capture<partial-network-capture>`.

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\(pytorch-master) (docs) (source) (notes) cuda.rst`, **line 698**); ***backlink***

Unknown interpreted text role "class".

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\pytorch-

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\(pytorch-master)(docs)(source)(notes)cuda.rst, line 698); *backlink*

Unknown interpreted text role "class".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\(pytorch-master)(docs)(source)(notes)cuda.rst, line 698); *backlink*

Unknown interpreted text role "class".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\(pytorch-master)(docs)(source)(notes)cuda.rst, line 698); *backlink*

Unknown interpreted text role "ref".

Constraints

A set of ops is *capturable* if it doesn't violate any of the following constraints.

Constraints apply to all work in a :class:`torch.cuda.graph` context and all work in the forward and backward passes of any callable you pass to :func:`torch.cuda.make_graphed_callables`.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\(pytorch-master)(docs)(source)(notes)cuda.rst, line 711); *backlink*

Unknown interpreted text role "class".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\(pytorch-master)(docs)(source)(notes)cuda.rst, line 711); *backlink*

Unknown interpreted text role "func".

Violating any of these will likely cause a runtime error:

- Capture must occur on a non-default stream. (This is only a concern if you use the raw :meth:`CUDAGraph.capture_begin<torch.cuda.CUDAGraph.capture_begin>` and :meth:`CUDAGraph.capture_end<torch.cuda.CUDAGraph.capture_end>` calls. :class:`~torch.cuda.graph` and :func:`~torch.cuda.make_graphed_callables` set a side stream for you.)

  System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\(pytorch-master)(docs)(source)(notes)cuda.rst, line 717); *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)cuda.rst, line 717); *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)cuda.rst, line 717); *backlink*

  Unknown interpreted text role "class".

  System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\(pytorch-master)(docs)(source)(notes)cuda.rst, line 717); *backlink*

  Unknown interpreted text role "func".

- Ops that synchronize the CPU with the GPU (e.g., `.item()` calls) are prohibited.
- CUDA RNG ops are allowed, but must use default generators. For example, explicitly constructing a new :class:`torch.Generator` instance and passing it as the `generator` argument to an RNG function is prohibited.

> **System Message: ERROR/3** (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\(pytorch-master)(docs)(source)(notes)cuda.rst, line 723); *backlink*
>
> Unknown interpreted text role "class".

Violating any of these will likely cause silent numerical errors or undefined behavior:

- Within a process, only one capture may be underway at a time.
- No non-captured CUDA work may run in this process (on any thread) while capture is underway.
- CPU work is not captured. If the captured ops include CPU work, that work will be elided during replay.
- Every replay reads from and writes to the same (virtual) memory addresses.
- Dynamic control flow (based on CPU or GPU data) is prohibited.
- Dynamic shapes are prohibited. The graph assumes every tensor in the captured op sequence has the same size and layout in every replay.
- Using multiple streams in a capture is allowed, but there are :ref:`restrictions<multistream-capture>`.

> **System Message: ERROR/3** (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\(pytorch-master)(docs)(source)(notes)cuda.rst, line 736); *backlink*
>
> Unknown interpreted text role "ref".

Non-constraints

- Once captured, the graph may be replayed on any stream.

Whole-network capture

If your entire network is capturable, you can capture and replay an entire iteration:

```
N, D_in, H, D_out = 640, 4096, 2048, 1024
model = torch.nn.Sequential(torch.nn.Linear(D_in, H),
                            torch.nn.Dropout(p=0.2),
                            torch.nn.Linear(H, D_out),
                            torch.nn.Dropout(p=0.1)).cuda()
loss_fn = torch.nn.MSELoss()
optimizer = torch.optim.SGD(model.parameters(), lr=0.1)

# Placeholders used for capture
static_input = torch.randn(N, D_in, device='cuda')
static_target = torch.randn(N, D_out, device='cuda')

# warmup
# Uses static_input and static_target here for convenience,
# but in a real setting, because the warmup includes optimizer.step()
# you must use a few batches of real data.
s = torch.cuda.Stream()
s.wait_stream(torch.cuda.current_stream())
with torch.cuda.stream(s):
    for i in range(3):
        optimizer.zero_grad(set_to_none=True)
        y_pred = model(static_input)
        loss = loss_fn(y_pred, static_target)
        loss.backward()
        optimizer.step()
torch.cuda.current_stream().wait_stream(s)

# capture
g = torch.cuda.CUDAGraph()
# Sets grads to None before capture, so backward() will create
# .grad attributes with allocations from the graph's private pool
optimizer.zero_grad(set_to_none=True)
with torch.cuda.graph(g):
    static_y_pred = model(static_input)
    static_loss = loss_fn(static_y_pred, static_target)
    static_loss.backward()
    optimizer.step()

real_inputs = [torch.rand_like(static_input) for _ in range(10)]
real_targets = [torch.rand_like(static_target) for _ in range(10)]

for data, target in zip(real_inputs, real_targets):
```

```
        # Fills the graph's input memory with new data to compute on
        static_input.copy_(data)
        static_target.copy_(target)
        # replay() includes forward, backward, and step.
        # You don't even need to call optimizer.zero_grad() between iterations
        # because the captured backward refills static .grad tensors in place.
        g.replay()
        # Params have been updated. static_y_pred, static_loss, and .grad
        # attributes hold values from computing on this iteration's data.
```

Partial-network capture

If some of your network is unsafe to capture (e.g., due to dynamic control flow, dynamic shapes, CPU syncs, or essential CPU-side logic), you can run the unsafe part(s) eagerly and use :func:`torch.cuda.make_graphed_callables` to graph only the capture-safe part(s).

> **System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\(pytorch-master)(docs)(source)(notes)cuda.rst, line 807); *backlink***
>
> Unknown interpreted text role "func".

By default, callables returned by :func:`~torch.cuda.make_graphed_callables` are autograd-aware, and can be used in the training loop as direct replacements for the functions or :class:`nn.Module<torch.nn.Module>`s you passed.

> **System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\(pytorch-master)(docs)(source)(notes)cuda.rst, line 812); *backlink***
>
> Unknown interpreted text role "func".

> **System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\(pytorch-master)(docs)(source)(notes)cuda.rst, line 812); *backlink***
>
> Unknown interpreted text role "class".

:func:`~torch.cuda.make_graphed_callables` internally creates :class:`~torch.cuda.CUDAGraph` objects, runs warmup iterations, and maintains static inputs and outputs as needed. Therefore (unlike with :class:`torch.cuda.graph`) you don't need to handle those manually.

> **System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\(pytorch-master)(docs)(source)(notes)cuda.rst, line 816); *backlink***
>
> Unknown interpreted text role "func".

> **System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\(pytorch-master)(docs)(source)(notes)cuda.rst, line 816); *backlink***
>
> Unknown interpreted text role "class".

> **System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\(pytorch-master)(docs)(source)(notes)cuda.rst, line 816); *backlink***
>
> Unknown interpreted text role "class".

In the following example, data-dependent dynamic control flow means the network isn't capturable end-to-end, but :func:`~torch.cuda.make_graphed_callables` lets us capture and run graph-safe sections as graphs regardless:

> **System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\(pytorch-master)(docs)(source)(notes)cuda.rst, line 821); *backlink***
>
> Unknown interpreted text role "func".

```
N, D_in, H, D_out = 640, 4096, 2048, 1024

module1 = torch.nn.Linear(D_in, H).cuda()
module2 = torch.nn.Linear(H, D_out).cuda()
module3 = torch.nn.Linear(H, D_out).cuda()

loss_fn = torch.nn.MSELoss()
optimizer = torch.optim.SGD(chain(module1.parameters(),
```

```
                                module2.parameters(),
                                module3.parameters()),
                            lr=0.1)

# Sample inputs used for capture
# requires_grad state of sample inputs must match
# requires_grad state of real inputs each callable will see.
x = torch.randn(N, D_in, device='cuda')
h = torch.randn(N, H, device='cuda', requires_grad=True)

module1 = torch.cuda.make_graphed_callables(module1, (x,))
module2 = torch.cuda.make_graphed_callables(module2, (h,))
module3 = torch.cuda.make_graphed_callables(module3, (h,))

real_inputs = [torch.rand_like(x) for _ in range(10)]
real_targets = [torch.randn(N, D_out, device="cuda") for _ in range(10)]

for data, target in zip(real_inputs, real_targets):
    optimizer.zero_grad(set_to_none=True)

    tmp = module1(data)  # forward ops run as a graph

    if tmp.sum().item() > 0:
        tmp = module2(tmp)  # forward ops run as a graph
    else:
        tmp = module3(tmp)  # forward ops run as a graph

    loss = loss_fn(tmp, target)
    # module2's or module3's (whichever was chosen) backward ops,
    # as well as module1's backward ops, run as graphs
    loss.backward()
    optimizer.step()
```

Usage with torch.cuda.amp

For typical optimizers, :meth:`GradScaler.step<torch.cuda.amp.GradScaler.step>` syncs the CPU with the GPU, which is prohibited during capture. To avoid errors, either use :ref:`partial-network capture<partial-network-capture>`, or (if forward, loss, and backward are capture-safe) capture forward, loss, and backward but not the optimizer step:

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\(pytorch-master)(docs)(source)(notes)cuda.rst, line 872); *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)cuda.rst, line 872); *backlink*

Unknown interpreted text role "ref".

```
# warmup
# In a real setting, use a few batches of real data.
s = torch.cuda.Stream()
s.wait_stream(torch.cuda.current_stream())
with torch.cuda.stream(s):
    for i in range(3):
        optimizer.zero_grad(set_to_none=True)
        with torch.cuda.amp.autocast():
            y_pred = model(static_input)
            loss = loss_fn(y_pred, static_target)
        scaler.scale(loss).backward()
        scaler.step(optimizer)
        scaler.update()
torch.cuda.current_stream().wait_stream(s)

# capture
g = torch.cuda.CUDAGraph()
optimizer.zero_grad(set_to_none=True)
with torch.cuda.graph(g):
    with torch.cuda.amp.autocast():
        static_y_pred = model(static_input)
        static_loss = loss_fn(static_y_pred, static_target)
    scaler.scale(static_loss).backward()
    # don't capture scaler.step(optimizer) or scaler.update()

real_inputs = [torch.rand_like(static_input) for _ in range(10)]
real_targets = [torch.rand_like(static_target) for _ in range(10)]

for data, target in zip(real_inputs, real_targets):
```

```
    static_input.copy_(data)
    static_target.copy_(target)
    g.replay()
    # Runs scaler.step and scaler.update eagerly
    scaler.step(optimizer)
    scaler.update()
```

Usage with multiple streams

Capture mode automatically propagates to any streams that sync with a capturing stream. Within capture, you may expose parallelism by issuing calls to different streams, but the overall stream dependency DAG must branch out from the initial capturing stream after capture begins and rejoin the initial stream before capture ends:

```
with torch.cuda.graph(g):
    # at context manager entrance, torch.cuda.current_stream()
    # is the initial capturing stream

    # INCORRECT (does not branch out from or rejoin initial stream)
    with torch.cuda.stream(s):
        cuda_work()

    # CORRECT:
    # branches out from initial stream
    s.wait_stream(torch.cuda.current_stream())
    with torch.cuda.stream(s):
        cuda_work()
    # rejoins initial stream before capture ends
    torch.cuda.current_stream().wait_stream(s)
```

> **Note**
>
> To avoid confusion for power users looking at replays in nsight systems or nvprof: Unlike eager execution, the graph interprets a nontrivial stream DAG in capture as a hint, not a command. During replay, the graph may reorganize independent ops onto different streams or enqueue them in a different order (while respecting your original DAG's overall dependencies).

Usage with DistributedDataParallel

NCCL < 2.9.6

NCCL versions earlier than 2.9.6 don't allow collectives to be captured. You must use :ref:`partial-network capture<partial-network-capture>`, which defers allreduces to happen outside graphed sections of backward.

> **System Message: ERROR/3** (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\(pytorch-master)(docs)(source)(notes)cuda.rst, line 955); *backlink*
>
> Unknown interpreted text role "ref".

Call :func:`~torch.cuda.make_graphed_callables` on graphable network sections *before* wrapping the network with DDP.

> **System Message: ERROR/3** (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\(pytorch-master)(docs)(source)(notes)cuda.rst, line 959); *backlink*
>
> Unknown interpreted text role "func".

NCCL >= 2.9.6

NCCL versions 2.9.6 or later allow collectives in the graph. Approaches that capture an :ref:`entire backward pass<whole-network-capture>` are a viable option, but need three setup steps.

> **System Message: ERROR/3** (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\(pytorch-master)(docs)(source)(notes)cuda.rst, line 965); *backlink*
>
> Unknown interpreted text role "ref".

1. Disable DDP's internal async error handling:

   ```
   os.environ["NCCL_ASYNC_ERROR_HANDLING"] = "0"
   torch.distributed.init_process_group(...)
   ```

2. Before full-backward capture, DDP must be constructed in a side-stream context:

   ```
   with torch.cuda.stream(s):
   ```

```
model = DistributedDataParallel(model)
```

3. Your warmup must run at least 11 DDP-enabled eager iterations before capture.

Graph memory management

A captured graph acts on the same virtual addresses every time it replays. If PyTorch frees the memory, a later replay can hit an illegal memory access. If PyTorch reassigns the memory to new tensors, the replay can corrupt the values seen by those tensors. Therefore, the virtual addresses used by the graph must be reserved for the graph across replays. The PyTorch caching allocator achieves this by detecting when capture is underway and satisfying the capture's allocations from a graph-private memory pool. The private pool stays alive until its :class:`~torch.cuda.CUDAGraph` object and all tensors created during capture go out of scope.

> **System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\(pytorch-master)(docs)(source)(notes)cuda.rst, line 986); *backlink***
>
> Unknown interpreted text role "class".

Private pools are maintained automatically. By default, the allocator creates a separate private pool for each capture. If you capture multiple graphs, this conservative approach ensures graph replays never corrupt each other's values, but sometimes needlessly wastes memory.

Sharing memory across captures

To economize the memory stashed in private pools, :class:`torch.cuda.graph` and :func:`torch.cuda.make_graphed_callables` optionally allow different captures to share the same private pool. It's safe for a set of graphs to share a private pool if you know they'll always be replayed in the same order they were captured, and never be replayed concurrently.

> **System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\(pytorch-master)(docs)(source)(notes)cuda.rst, line 1004); *backlink***
>
> Unknown interpreted text role "class".

> **System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\(pytorch-master)(docs)(source)(notes)cuda.rst, line 1004); *backlink***
>
> Unknown interpreted text role "func".

:class:`torch.cuda.graph`'s `pool` argument is a hint to use a particular private pool, and can be used to share memory across graphs as shown:

> **System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\(pytorch-master)(docs)(source)(notes)cuda.rst, line 1011); *backlink***
>
> Unknown interpreted text role "class".

```
g1 = torch.cuda.CUDAGraph()
g2 = torch.cuda.CUDAGraph()

# (create static inputs for g1 and g2, run warmups of their workloads...)

# Captures g1
with torch.cuda.graph(g1):
    static_out_1 = g1_workload(static_in_1)

# Captures g2, hinting that g2 may share a memory pool with g1
with torch.cuda.graph(g2, pool=g1.pool()):
    static_out_2 = g2_workload(static_in_2)

static_in_1.copy_(real_data_1)
static_in_2.copy_(real_data_2)
g1.replay()
g2.replay()
```

With :func:`torch.cuda.make_graphed_callables`, if you want to graph several callables and you know they'll always run in the same order (and never concurrently) pass them as a tuple in the same order they'll run in the live workload, and :func:`~torch.cuda.make_graphed_callables` will capture their graphs using a shared private pool.

> **System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\(pytorch-master)(docs)(source)(notes)cuda.rst, line 1032); *backlink***
>
> Unknown interpreted text role "func".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\(pytorch-master)(docs)(source)(notes)cuda.rst, line 1032); *backlink*

Unknown interpreted text role "func".

If, in the live workload, your callables will run in an order that occasionally changes, or if they'll run concurrently, passing them as a tuple to a single invocation of :func:`~torch.cuda.make_graphed_callables` is not allowed. Instead, you must call :func:`~torch.cuda.make_graphed_callables` separately for each one.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\(pytorch-master)(docs)(source)(notes)cuda.rst, line 1038); *backlink*

Unknown interpreted text role "func".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\(pytorch-master)(docs)(source)(notes)cuda.rst, line 1038); *backlink*

Unknown interpreted text role "func".