CUDA semantics — PyTorch 1.7.0 documentation

https://pytorch.org/docs/stable/notes/cuda.html

None Sun Nov, 22 03:48

`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 `torch.cuda.device` context manager.

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 `copy_()` and other methods with copy-like functionality such as `to()` and `cuda()`. Unless you enable peer-to-peer memory access, any attempts to launch ops on tensors spread across different devices will raise an error.

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 truncating 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
```

For more information about TF32, see:

- TensorFloat-32
- CUDA 11
- Ampere architecture

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 `torch.cuda.synchronize()` before measuring, or use `torch.cuda.Event` to record times as following:

```
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 `to()` and `copy_()` admit an explicit `non_blocking` argument, which lets the caller bypass synchronization when it is unnecessary. Another exception is CUDA streams, explained below.

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 `synchronize()` or `wait_stream()`) are used. For example, the following code is incorrect:

```
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.

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 `memory_allocated()` and `max_memory_allocated()` to monitor memory occupied by tensors, and use `memory_reserved()` and `max_memory_reserved()` to monitor the total amount of memory managed by the caching allocator. Calling `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.

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

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.

cuFFT plan cache¶

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

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 `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`.

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 `is_available()`. In the following, `args.device` results in a `torch.device` object that can be used to move tensors to CPU or CUDA.

```
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 `torch.cuda.device` context manager.

```
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 `torch.Tensor`). Whilst the previously mentioned `torch.*` factory functions (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.

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, `ones_like()` or `zeros_like()` are provided as convenient helper functions (which also preserve `torch.device` and `torch.dtype` of a Tensor).

```
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¶

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

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

You can make the `DataLoader` return batches placed in pinned memory by passing `pin_memory=True` to its constructor.

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

Most use cases involving batched inputs and multiple GPUs should default to using `DistributedDataParallel` to utilize more than one GPU.

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

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

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

If you use `DistributedDataParallel`, you could use *torch.distributed.launch* utility to launch your program, see Third-party backends.