torchgpipe

Release 0.0.7

Kakao Brain

Sep 19, 2020

CONTENTS

A GPipe implementation in PyTorch.

```
from torchgpipe import GPipe

model = nn.Sequential(a, b, c, d)
model = GPipe(model, balance=[1, 1, 1, 1], chunks=8)

for input in data_loader:
    output = model(input)
```

CHAPTER

ONE

WHAT IS GPIPE?

GPipe is a scalable pipeline parallelism library published by Google Brain, which allows efficient training of large, memory-consuming models. According to the paper, GPipe can train a 25x larger model by using 8x devices (TPU), and train a model 3.5x faster by using 4x devices.

GPipe: Efficient Training of Giant Neural Networks using Pipeline Parallelism

Google trained AmoebaNet-B with 557M parameters over GPipe. This model has achieved 84.3% top-1 and 97.0% top-5 accuracy on ImageNet classification benchmark (the state-of-the-art performance as of May 2019).

CHAPTER

TWO

DOCUMENTATIONS

2.1 Understanding GPipe

GPipe uses (a) *Pipeline Parallelism* and (b) automatic recomputation of the forward propagation during the back-propagation, hence leverages training a large model. We refer to (b) as *Checkpointing*, following the well-known terminology in PyTorch community.

2.1.1 Pipeline Parallelism

GPipe splits a model into multiple partitions and places each partition on a different device to occupy more memory capacity. For example, we may split a model occupying 40GB of CUDA memory into 4 partitions each occupying 10GB, respectively.

This approach is called *model parallelism*. However, typical deep learning models are composed of sequential layers. In other words, usually the latter layer wouldn't work until the prior layer has finished. If a model is composed of fully sequential layers, even if we spread the model over two or more devices, only one device can be utilized at once.

GPipe splits a mini-batch into multiple micro-batches to make the devices work as parallel as possible. It is called *pipeline parallelism*. Basically, pipeline parallelism is a stack of small data parallelism. When each partition has finished processing a micro-batch, it can toss the output to the next partition and immediately can start to work on the next micro-batch. Now the partitions can be overlapped.

See also:

Model Parallel Best Practices in PyTorch Tutorials

There is still idle time called *bubble* because every partition has to wait for the first micro-batch from the prior partition. The bubble can be reduced by choosing a smaller size of micro-batches. But usually, larger batch size can utilize GPU more efficiently. Hence, GPU may be underutilized if too small size of micro-batches is chosen.

A faster partition should wait for adjacent slower partition. Therefore, imbalance over partitions also may cause GPU underutilization. Note that the overall performance is determined by the slowest partition.

2.1.2 Checkpointing

Checkpointing is applied to each partition to minimize the overall memory consumption by a model. During forward propagation, only the tensors at the boundaries between partitions are remembered. All other intermediate tensors are volatilized, and recomputed during backpropagation when necessary. Specifically, hidden layers consume the memory which is required by only a single micro-batch with checkpointing.

Checkpointing is a trade-off between performance and memory, because recomputation spends time just as much as the forward propagation. When you use *torchgpipe.GPipe*, you can decide to turn off checkpointing by `checkpoint='never'` option.

2.1.3 Deferred Batch Normalization

One of the goals of GPipe is *transparency*. GPipe shouldn't affect existing hyperparameters and output during training. However, if a module processes per mini-batch, not per single sample, it might be affected by GPipe since each module could see only a micro-batch at once.

Meanwhile, batch normalization is a module commonly used in CNN. The forward propagation of this module performs two procedures in training. Both the procedures are per mini-batch, not micro-batch:

1. Normalizing a mini-batch by the average and variance of the just given mini-batch.
2. Tracking moving statistics (mean and variance) of mini-batches to normalize batches in evaluation.

GPipe couldn't provide transparency for the first procedure (normalization). Per mini-batch normalization introduces a dependency among the micro-batches, hence it breaks the parallelism of GPipe. But the second procedure (tracking moving statistics) could be transparent with GPipe by accumulating statistics of all micro-batches.

torchgpipe provides this functionality as *deferred batch normalization*. But in the current implementation, it is slower than the vanilla batch normalization. That is why we turn off by default. If you need transparent moving statistics, turn on by `deferred_batch_norm=True` option in *GPipe*:

```
model = GPipe(model, balance=[1, 1, 1, 1], chunks=8,
              # Turn on deferred batch normalization.
              deferred_batch_norm=True)
```

2.2 User Guide

2.2.1 Installation

torchgpipe is available on PyPI. Install by `pip`:

```
$ pip install torchgpipe
```

Python 3.6+ (CPython) is required.

PyTorch 1.1+ will be installed automatically if you don't have a satisfied one. However, we highly recommend you to use the latest version of PyTorch.

2.2.2 Applying GPipe

To train a module with GPipe, simply wrap it with *torchgpipe.GPipe*. Your module must be a `nn.Sequential` as *GPipe* will automatically split the module into partitions. A partition is a group of consecutive layers that run on a single device together. *balance* argument determines the number of layers in each partition. *chunks* argument specifies the number of micro-batches. Input, output, and intermediate tensors must be `Tensor` or `Tuple[Tensor, ...]`. See also *Restrictions* for more details.

The below example code shows how to split a module with four layers into two partitions each having two layers. This code also splits a mini-batch into 8 micro-batches:

```
from torchgpipe import GPipe

model = nn.Sequential(a, b, c, d)
model = GPipe(model, balance=[2, 2], chunks=8)

# 1st partition: nn.Sequential(a, b) on cuda:0
# 2nd partition: nn.Sequential(c, d) on cuda:1

for input in data_loader:
    output = model(input)
```

GPipe optimizes training using CUDA. You should not move the module to a GPU yourself, because *GPipe* automatically moves each partition over different devices. By default, available GPUs starting from `cuda:0` are selected in order for each partition. You can also specify GPUs to use with *devices* parameter:

```
model = GPipe(model,
              balance=[2, 2],
              devices=[4, 2],  # Specify GPUs.
              chunks=8)
```

Input and Output Device

Unlike a typical module, with *GPipe*, the input device is different from the output device except for when there is only one partition. This is because the first partition and last partition are placed in different devices.

Therefore, you have to move the input and target to the corresponding devices. It can be done with *GPipe.devices*, which holds the list of devices for each partition:

```
in_device = model.devices[0]
out_device = model.devices[-1]

for input, target in data_loader:
    # input on in_device
    input = input.to(in_device, non_blocking=True)

    # target on out_device
    target = target.to(out_device, non_blocking=True)

    # output on out_device
    output = model(input)
    loss = F.cross_entropy(output, target)
    loss.backward()
    ...
```

Nested Sequentials

When *GPipe* splits a `nn.Sequential` module, it regards every child of the module as a single, non-divisible layer. However, it may be the case that some child is another sequential module and one may want to split them further.

This kind of recursive split of a nested sequential module is not intended nor supported by *GPipe*. It's your responsibility to flatten the module. Fortunately, this is not hard in PyTorch. Follow this code snippet which shows how a nested sequential module can be flattened:

```
_3_layers = nn.Sequential(...)  # len(_3_layers) == 3
_4_layers = nn.Sequential(...)  # len(_4_layers) == 4
model = nn.Sequential(_3_layers, _4_layers)  # len(model) == 2

def flatten_sequential(module):
    def _flatten(module):
        for name, child in module.named_children():
            if isinstance(child, nn.Sequential):
                for sub_name, sub_child in _flatten(child):
                    yield (f'{name}_{sub_name}', sub_child)
            else:
                yield (name, child)
    return nn.Sequential(OrderedDict(_flatten(module)))

model = flatten_sequential(model)  # len(model) == 7
model = GPipe(model, balance=[2, 3, 2], chunks=4)
```

Typical Model Parallelism

The typical model parallelism is a special case of GPipe. Model parallelism is equivalent to GPipe if micro-batching and checkpointing are disabled. Set `chunks=1` and `checkpoint='never'` for this:

```
model = GPipe(model, balance=[2, 2], chunks=1, checkpoint='never')
```

2.2.3 Automatic Balancing

It could be hard to determine the optimal balance of a model. In particular, if you are still designing a model, the model architecture may change over time. In this case, we highly recommend *torchgpipe.balance* for automatic balancing. This won't give you the optimal balance, but a good-enough balance. Note that this is provided by *torchgpipe*, and is not from the GPipe paper by Huang et al.

There are two balancing tools, *balance_by_time()* and *balance_by_size()*. Both are based on per-layer profiling. Just like PyTorch JIT, you need to feed a sample input into the model. *balance_by_time()* traces elapsed time of each layer, while *balance_by_size()* detects the CUDA memory usage of each layer. Choose the balancing tool for your needs:

```
from torchgpipe import GPipe
from torchgpipe.balance import balance_by_time

partitions = torch.cuda.device_count()
sample = torch.rand(128, 3, 224, 224)
balance = balance_by_time(partitions, model, sample)

model = GPipe(model, balance, chunks=8)
```

2.2.4 Trade-offs

Number of Micro-batches

Number of micro-batches has a trade-off between GPU utilization per micro-batch and total area of bubble. You need to find the best number of micro-batches for your model.

GPU may slow down when processing many small micro-batches compared to larger micro-batches. GPU will not be fully utilized if each CUDA kernel is too cheap to compute, hence too small micro-batches cause underutilization. On the other hand, the area of bubble is minimized when the size of each micro-batch is minimal. Ideally, you should choose the largest number of micro-batches that doesn't underutilize GPUs.

As a side note, BatchNorm tends to perform worse with smaller batch size. Large number of micro-batches may affect the final performance of model using BatchNorm negatively just like in `nn.DataParallel`.

Checkpointing

Checkpointing drastically helps to reduce memory usage, but the overall training would slow down by about 25%. You can handle how to apply checkpointing on your model. There are three options:

- `'always'` – Apply checkpointing over all micro-batches.
- `'except_last'` (default) – Apply checkpointing except the last micro-batch.
- `'never'` – Checkpointing is never applied.

Usually, checkpointing at the last micro-batch may not be useful because the saved memory will be reconstructed immediately. That's why we choose `'except_last'` as the default option.

If you decide not to use checkpointing at all, `nn.DataParallel` might be more efficient than GPipe.

2.2.5 Referential Transparency

Checkpointing executes forward propagation again at backpropagation, which is called *recomputation*. We assume that both the executions are identical. Hence, all layers should be referentially transparent in forward propagation. Here are the typical cases that break referential transparency:

In-place Operations: We do not recommend using in-place operations with checkpointing. Especially, if an in-place operation such as `add_(1)` is applied to the input of a checkpointed partition, then the recomputation can't recover the original input.

Randomness not managed by PyTorch: The randomness managed by PyTorch, including `torch.manual_seed()`, `torch.rand()`, or `nn.Dropout`, is deterministically reproduced in recomputation. But other randomnesses, such as Python standard `random` or `numpy.random`, are not. We highly recommend to use PyTorch randomness for referential transparency.

Side Effects: Some modules such as BatchNorm update their state in forward propagation. Hence, updated state in recomputation might not be identical to the original state.

2.2.6 Restrictions

If you get any errors, check the following restrictions first.

Sequential: Your module must be `nn.Sequential`. For example, the models in `torchvision` are not sequential. They can't be wrapped by *GPipe* directly:

```
>>> from torchvision.models.resnet import resnet101
>>> model = resnet101()
>>> type(model)
torchvision.models.resnet.ResNet
>>> GPipe(model, balance=..., chunks=...)
Traceback (most recent call last)
  ...
TypeError: module must be nn.Sequential to be partitioned
```

See the sequential ResNet example to figure out how to make a model into a `nn.Sequential` model.

`nn.Sequential` assumes that every underlying layer takes only one argument. Calling `forward(x)` on `nn.Sequential(A(), B(), C())` is essentially the same as calling `C(B(A(x)))`. Hence, you can't design an underlying layer with multiple arguments:

```
class MyModule(nn.Module):
    def forward(self, a, b, c):
        return a + b - c

model = nn.Sequential(..., MyModule(), ...)
model(input)  # FAILS!
```

Tensor or Tensors: As we discussed above, each layer must take only one argument due to `nn.Sequential`. There is one more restriction. Every underlying layers' input and output must be `Tensor` or `Tuple[Tensor, ...]`:

```
# OK
def forward(input: Tensor) -> Tensor: ...
def forward(input: Tensor) -> Tuple[Tensor, Tensor]: ...
def forward(input: Tuple[Tensor, Tensor]) -> Tensor: ...

# Error
def forward(input1: Tensor, input2: Tensor) -> Tensor: ...
def forward(input: Tensor, label: str) -> Tensor: ...
def forward(input: Tensor) -> Dict[str, Tensor]: ...
def forward(input: Tensor) -> Tuple[Tensor, str]: ...
```

The reason is that *GPipe* can't assume how the non-tensor inputs for a mini-batch can be split for micro-batches.

Unique Parameters: *GPipe* places each partition on the corresponding device. When placing a partition, the parameters of the partition are also moved to the destination. *GPipe* cannot support a module with a parameter on two or more devices:

```
>>> conv1 = nn.Conv2d(3, 3, 1)
>>> conv2 = nn.Conv2d(3, 3, 1)
>>> conv1.weight = conv2.weight
>>> model = nn.Sequential(conv1, conv2)
>>> model = GPipe(model, balance=[1, 1], ...)
Traceback (most recent call last)
  ...
ValueError: module with duplicate parameters in distinct children is not supported
```

2.2.7 Complex Modules

This part of the documentation discusses how to implement a complex module compatible with *GPipe*. First, you should understand how GPipe works. See *Understanding GPipe*.

Skip Connections

Many deep learning models, such as ResNet, AmoebaNet, or U-Net, contain skip connections. There are two ways to implement skip connections. Let's assume we have to implement a skip connection like this:

```
latent = layer1(input)
latent = layer2(latent)
output = layer3(latent) + input  # skip connection
```

To make this module sequential, we define modules for each layer. Simply, a skip connection can be implemented by making underlying layers with `Tuple[Tensor, Tensor]` parameter and return type:

```
class Layer1(nn.Module):
    #          ┌──────────────────┐
    # input --┼-+-> layer1 ----┼--> output
    #          │ '--------------┼--> skip
    #          └──────────────────┘
    def forward(self, input):
        skip = input
        return layer1(input), skip

class Layer2(nn.Module):
    #          ┌──────────────────┐
    # input --┼---> layer2 ----┼--> output
    #  skip --┼-----------------┼--> skip
    #          └──────────────────┘
    def forward(self, input_and_skip):
        input, skip = input_and_skip
        return layer2(input), skip

class Layer3(nn.Module):
    #          ┌──────────────────┐
    # input --┼---> layer3 --+-┼--> output
    #  skip --┼--------------' │
    #          └──────────────────┘
    def forward(self, input_and_skip):
        input, skip = input_and_skip
        return layer3(input) + skip

model = nn.Sequential(Layer1(), Layer2(), Layer3())
```

Because of the skip connection being represented as a normal parameter, *GPipe* can move the tensors from partition to partition:

```
model = GPipe(model, balance=[1, 1, 1], chunks=8)
```

This seems a fairly straightforward way to implement skip connections. But there is a disadvantage. In the above example, the skip tensor is copied to the second device, but it is never used at the device. Unnecessary copies of skip tensors may waste time and memory. The following section introduces an alternative approach for skip connection.

Long Skip Connections

The disadvantage mentioned above might be catastrophic if it involves unnecessary copies of a large tensor, and/or over many devices. The second case often occurs when implementing long skip connections.

Let's assume now we have 8 layers between input and output:

```
latent = layer1(input)
latent = layer2(latent)
latent = layer3(latent)
latent = layer4(latent)
latent = layer5(latent)
latent = layer6(latent)
latent = layer7(latent)
output = layer8(latent) + input  # skip connection
```

With the prior approach, the skip tensor will be copied to every device, but six devices do not need it. The alternative approach is to expose in which layer the skip tensor is produced and consumed. We introduce the *@skippable* class decorator to toss the tensor directly, without needing to pass it to irrelevant layers in between. A module can stash a tensor into the storage or pop. This functionality works perfectly fine even when the module is not wrapped by *GPipe*.

The decorator declares which skip tensors would be stashed or popped in the decorated module. Let us explain how to implement the 8-layer example above using *torchgpipe.skip*. Here we use the name "skip" for the skip connection between `Layer1` and `Layer8`:

```
# Layer1 stashes 'skip'.
@skippable(stash=['skip'])
class Layer1(nn.Module):
    ...

# Layer8 pops 'skip'.
@skippable(pop=['skip'])
class Layer8(nn.Module):
    ...
```

When `Layer1` prepares a skip tensor, it can stash the tensor into the hidden storage by *yield stash()*. As you may have noticed, we define `forward()` as a generator instead of a normal function:

```
@skippable(stash=['skip'])
class Layer1(nn.Module):
    def forward(self, input):
        skip = input
        yield stash('skip', skip)
        return layer1(input)
```

Similarly, `Layer8` also can pop the stashed skip tensor by *yield pop()*:

```
@skippable(pop=['skip'])
class Layer8(nn.Module):
    def forward(self, input):
        skip = yield pop('skip')
        return layer8(input) + skip
```

Now the intermediate layers do not interact with the skip tensor at all:

```
class Layer2(nn.Module):
    def forward(self, input):
```

(continues on next page)

(continued from previous page)

```
        return layer2(input)
...
class Layer7(nn.Module):
    def forward(self, input):
        return layer7(input)
```

You can design any complex skip connections with *@skippable* since a skippable module could stash and/or pop multiple skip tensors. However, there are restrictions:

- Every skip name must be unique within a sequential module.
- Every skip tensor must be stashed and popped exactly once.

Then, how can we instantiate multiple skippable modules from the same class in a sequential module? You can isolate some skip names into a *Namespace*. For example, a conceptual U-Net can be designed like this. There are 3 pairs of `Encoder` and `Decoder`:

```
# 1F. Encoder -------- Decoder -- Segment
#       \                 /
# 2F.  Encoder ------ Decoder
#         \             /
# 3F.   Encoder ---- Decoder
#           \         /
# 4F.        Bottleneck

@skippable(stash=['skip'])
class Encoder(nn.Module):
    ...

@skippable(pop=['skip'])
class Decoder(nn.Module):
    ...

ns_1f = Namespace()
ns_2f = Namespace()
ns_3f = Namespace()

model = nn.Sequential(
    Encoder().isolate(ns_1f),
    Encoder().isolate(ns_2f),
    Encoder().isolate(ns_3f),
    Bottleneck(),
    Decoder().isolate(ns_3f),
    Decoder().isolate(ns_2f),
    Decoder().isolate(ns_1f),
    Segment(),
)
```

Some skip connection may be conditional on input. However, *@skippable* doesn't allow *stash()* or *pop()* missing. Instead, it allows `None` in place of skip tensor:

```
@skippable(stash=['skip'])
class MaybeStash(nn.Module):
    def forward(self, input):
        skip = input if test(input) else None
        yield stash('skip', skip)
        return f(input)
```

(continues on next page)

(continued from previous page)

```
@skippable(pop=['skip'])
class MaybePop(nn.Module):
    def forward(self, input):
        output = f(input)
        skip = yield pop('skip')
        if skip is not None:
            output += skip
        return output
```

Detecting Recomputation

Checkpointing in GPipe performs forward propagations twice. The second forward propagation is called *recomputation*. This may cause a problem when a module such as `nn.BatchNorm2d` updates its running estimates of batch statistics on each forward propagation. It should not update the running estimates again during the recomputation. To avoid updating the running estimates twice, modules' `forward` method needs be able to detect that this is the recomputation.

It can be done by *`is_recomputing()`*. This function returns `True` if called during the recomputation:

```
class Counter(nn.Module):
    def __init__(self):
        super().__init__()
        self.counter = 0

    def forward(self, input):
        if not is_recomputing():
            self.counter += 1
        return input
```

Note: `deferred_batch_norm=True` on *`GPipe`* will prevent updating the running statistics twice.

2.3 API

2.3.1 GPipe Module

class torchgpipe.**GPipe**(*module*, *balance*, ***kwargs*)

Wraps an arbitrary `nn.Sequential` module to train on GPipe. If the module requires lots of memory, GPipe will be very efficient.

```
model = nn.Sequential(a, b, c, d)
model = GPipe(model, balance=[1, 1, 1, 1], chunks=8)
output = model(input)
```

GPipe combines pipeline parallelism with checkpointing to reduce peak memory required to train while minimizing device under-utilization.

You should determine the balance when defining a *`GPipe`* module, as balancing will not be done automatically. The module will be partitioned into multiple devices according to the given balance. You may rely on heuristics to find your own optimal configuration.

Parameters

- **module** (*torch.nn.Sequential*) – sequential module to be parallelized
- **balance** (*ints*) – list of number of layers in each partition

Keyword Arguments

- **devices** (*iterable of devices*) – devices to use (default: all CUDA devices)
- **chunks** (*int*) – number of micro-batches (default: `1`)
- **checkpoint** (*str*) – when to enable checkpointing, one of `'always'`, `'except_last'`, or `'never'` (default: `'except_last'`)
- **deferred_batch_norm** (*bool*) – whether to use deferred BatchNorm moving statistics (default: `False`, see *Deferred Batch Normalization* for more details)

Raises

- **TypeError** – the module is not a `nn.Sequential`.
- **ValueError** – invalid arguments, or wrong balance
- **IndexError** – the number of devices is fewer than the number of partitions.

forward(*input*)

GPipe is a fairly transparent module wrapper. It doesn't modify the input and output signature of the underlying module. But there's type restriction. Input and output have to be a `Tensor` or a tuple of tensors. This restriction is applied at partition boundaries too.

Parameters **input** (*torch.Tensor or tensors*) – input mini-batch

Returns output mini-batch

Return type tensor or tensors

Raises **TypeError** – input is not a tensor or tensors.

balance

The number of layers in each partition.

devices

The devices mapped to each partition.

`devices[-1]` refers to the device of the last partition, which means it is the output device. Probably, you need to use it to transfer the target to calculate the loss without a device mismatch `RuntimeError`. For example:

```
out_device = gpipe.devices[-1]

for input, target in loader:
    target = target.to(out_device, non_blocking=True)
    output = gpipe(input)
    loss = F.cross_entropy(output, target)
```

chunks

The number of micro-batches.

checkpoint

The checkpoint mode to determine when to enable checkpointing. It is one of `'always'`, `'except_last'`, or `'never'`.

2.3.2 Skip Connections

`@torchgpipe.skip.`**`skippable`**(*[stash][, pop]*)

The decorator to define a `nn.Module` with skip connections. Decorated modules are called "skippable". This functionality works perfectly fine even when the module is not wrapped by *GPipe*.

Each skip tensor is managed by its name. Before manipulating skip tensors, a skippable module must statically declare the names for skip tensors by *stash* and/or *pop* parameters. Skip tensors with pre-declared name can be stashed by `yield stash(name, tensor)` or popped by `tensor = yield pop(name)`.

Here is an example with three layers. A skip tensor named "1to3" is stashed and popped at the first and last layer, respectively:

```
@skippable(stash=['1to3'])
class Layer1(nn.Module):
    def forward(self, input):
        yield stash('1to3', input)
        return f1(input)

class Layer2(nn.Module):
    def forward(self, input):
        return f2(input)

@skippable(pop=['1to3'])
class Layer3(nn.Module):
    def forward(self, input):
        skip_1to3 = yield pop('1to3')
        return f3(input) + skip_1to3

model = nn.Sequential(Layer1(), Layer2(), Layer3())
```

One skippable module can stash or pop multiple skip tensors:

```
@skippable(stash=['alice', 'bob'], pop=['carol'])
class StashStashPop(nn.Module):
    def forward(self, input):
        yield stash('alice', f_alice(input))
        yield stash('bob', f_bob(input))
        carol = yield pop('carol')
        return input + carol
```

Every skip tensor must be associated with exactly one pair of *stash* and *pop*. *GPipe* checks this restriction automatically when wrapping a module. You can also check the restriction by *verify_skippables()* without *GPipe*.

Note: *@skippable* changes the type of the wrapped class. But currently (mypy v0.740), mypy could not understand class decorators yet (#3135).

There are two workarounds:

1. Naively ignore type errors by `# type: ignore`.
2. Use `skippable()()` as a function instead of a decorator.

See also:

Long Skip Connections

Skippable.**isolate**(*ns*[, *only=names*])

Isolates a specified subset or the whole set of skip tensors into a namespace. In a single sequential module, skip tensors with the same name are not allowed unless they are isolated by different namespaces.

Here's an example using the same name for skip tensors twice. Each pair of `Layer1` and `Layer2` is isolated with its own namespace `ns1` and `ns2`. There is no conflict anymore:

```
ns1 = Namespace()
ns2 = Namespace()

model = nn.Sequential(
    Layer1().isolate(ns1),
    Layer1().isolate(ns2),
    Layer2(),
    Layer3().isolate(ns2),
    Layer3().isolate(ns1),
)
```

When *only* parameter is omitted, all skip tensors are isolated. You can isolate a subset of skip tensors by passing *only* parameter:

```
ns_alice = Namespace()
ns_bob = Namespace()

model = nn.Sequential(
    ...
    StashStashPop().isolate(ns_alice, only=['alice']) \
                   .isolate(ns_bob, only=['bob']),
    ...
)
```

Parameters **ns** (Namespace) – namespace for isolation

Keyword Arguments **only** (*iterable of strs*) – names of specific skip tensors to be isolated (omit this option to isolate all skip tensors declared in this module)

Returns this module itself

torchgpipe.skip.**stash**(*name*, *tensor*)

The command to stash a skip tensor.

```
def forward(self, input):
    yield stash('name', input)
    return f(input)
```

Parameters

- **name** (*str*) – name of skip tensor
- **input** (*torch.Tensor or None*) – tensor to pass to the skip connection

torchgpipe.skip.**pop**(*name*)

The command to pop a skip tensor.

```
def forward(self, input):
    skip = yield pop('name')
    return f(input) + skip
```

Parameters **name** (*str*) – name of skip tensor

Returns the skip tensor previously stashed by another layer under the same name

class torchgpipe.skip.**Namespace**
Namespace for isolating skip tensors used by *isolate()*.

torchgpipe.skip.**verify_skippables**(*module*)
Verifies if the underlying skippable modules satisfy integrity.

Every skip tensor must have only one pair of *stash* and *pop*. If there are one or more unmatched pairs, it will raise TypeError with the detailed messages.

Here are a few failure cases. *verify_skippables()* will report failure for these cases:

```
# Layer1 stashes "1to3".
# Layer3 pops "1to3".

nn.Sequential(Layer1(), Layer2())
#               └─────  ?

nn.Sequential(Layer2(), Layer3())
#                   ?  ─────

nn.Sequential(Layer1(), Layer2(), Layer3(), Layer3())
#               └──────────────────────        ^^^^^^

nn.Sequential(Layer1(), Layer1(), Layer2(), Layer3())
#             ^^^^^^      └──────────────────────
```

To use the same name for multiple skip tensors, they must be isolated by different namespaces. See *isolate()*.

Raises **TypeError** – one or more pairs of *stash* and *pop* are not matched.

2.3.3 Inspecting GPipe Timeline

torchgpipe.**is_checkpointing**()
Whether the current forward propagation is under checkpointing.

Returns True if it's under checkpointing.

Return type bool

torchgpipe.**is_recomputing**()
Whether the current forward propagation is under checkpoint recomputation. Use this to prevent duplicated side-effects at forward propagation:

```
class Counter(nn.Module):
    def __init__(self):
        super().__init__()
        self.counter = 0

    def forward(self, input):
        if not is_recomputing():
            self.counter += 1
        return input
```

Returns True if it's under checkpoint recomputation.

Return type bool

See also:

Detecting Recomputation

2.3.4 Automatic Balancing

torchgpipe.balance.**balance_by_time**(*partitions*, *module*, *sample*, *timeout=1.0*, *device=torch.device('cuda')*)

Naive automatic balancing by elapsed time per layer.

```
sample = torch.empty(128, 3, 224, 224)
balance = balance_by_time(torch.cuda.device_count(), model, sample)
gpipe = GPipe(model, balance, chunks=8)
```

Parameters

- **partitions** (*int*) – intended number of partitions
- **module** (*torch.nn.Sequential*) – sequential module to be partitioned
- **sample** (*torch.Tensor*) – example input with arbitrary batch size

Keyword Arguments

- **timeout** (*float*) – profiling iterates again if the timeout (in second) is not exceeded (default: `1.0`)
- **device** (*'cpu' or 'cuda' device*) – CPU or CUDA device where each layer is profiled (default: the current CUDA device)

Returns A list of number of layers in each partition. Use it for the *balance* parameter of *GPipe*.

Note: *module* and *sample* must be placed on the same device.

---

torchgpipe.balance.**balance_by_size**(*partitions*, *module*, *input*, *chunks=1*, *param_scale=2.0*, *device=torch.device('cuda')*)

Naive automatic balancing by CUDA memory usage per layer.

During training, required memory for parameters depends on which optimizer is used. Optimizers may use buffers for each parameter to track optimization statistics internally, such as momentum buffer in SGD.

To get more reliable size based balance, you should specify *param_scale* with regard to your optimizer. The default *param_scale* is 2 instead of 1 due to gradient accumulation which is necessary for every optimizer.

Follow this guide to choose correct *param_scale* for typical optimizers:

Optimizer	*param_scale*	Internal State
SGD	2–3	(momentum_buffer)
Adam	4–5	exp_avg, exp_avg_sq, (max_exp_avg_sq)
Adadelta	4	square_avg, acc_delta
Adagrad	3	sum
RMSprop	3–5	square_avg, (momentum_buffer), (grad_avg)

Here's a simple example with the Adam optimizer:

```
balance = balance_by_size(
    torch.cuda.device_count(),
    model,

    # Same size with mini-batch to train
    torch.empty(1024, 3, 224, 224),

    # Number of micro-batches to train with GPipe
    chunks=8,

    # 4 for Adam
    param_scale=4.0,
)

gpipe = GPipe(model, balance, chunks=8)
adam = Adam(gpipe.parameters())
```

Parameters

- **partitions** (*int*) – intended number of partitions
- **module** (*torch.nn.Sequential*) – sequential module to be partitioned
- **input** (*torch.Tensor*) – example mini-batch with the same size to train

Keyword Arguments

- **chunks** (*int*) – number of micro-batches will be used to train (default: `1`)
- **param_scale** (*float*) – how many copies of parameters would be allocated for training. It depends on optimizer. See the above guide. (default: `2.0`)
- **device** (*'cuda' device*) – CUDA device where each layer is profiled (default: the current CUDA device)

Returns A list of number of layers in each partition. Use it for the *balance* parameter of *GPipe*.

Note: *module* and *input* must be placed on the same CUDA device.

2.4 Benchmarks

Every experiment is reproducible on Tesla P40 GPUs. Follow the link to code for each benchmark.

2.4.1 Transparency

ResNet-101 Accuracy Benchmark

Batch size	torchgpipe	nn.DataParallel	Goyal et al.
256	21.99±0.13	22.02±0.11	22.08±0.06
1K	22.24±0.19	22.04±0.24	N/A
4K	22.13±0.09	N/A	N/A

GPipe should be transparent not to introduce additional hyperparameter tuning. To verify the transparency, we reproduced top-1 error rate of ResNet-101 on ImageNet, as reported in Table 2(c) of Accurate, Large Minibatch SGD by Goyal et al.

The reproducible code and experiment details are available in benchmarks/resnet101-accuracy.

2.4.2 Memory

U-Net (B, C) Memory Benchmark

Experiment	U-Net (B, C)	Parameters	Memory usage
baseline	(6, 72)	362.2M	20.3 GiB
pipeline-1	(11, 128)	2.21B	20.5 GiB
pipeline-2	(24, 128)	4.99B	43.4 GiB
pipeline-4	(24, 160)	7.80B	79.1 GiB
pipeline-8	(48, 160)	15.82B	154.1 GiB

The table shows how GPipe facilitates scaling U-Net models. *baseline* denotes the baseline without pipeline parallelism nor checkpointing, and *pipeline-1*, *-2*, *-4*, *-8* denotes that the model is trained with GPipe with the corresponding number of partitions.

Here we used a simplified U-Net architecture. The size of a model is determined by hyperparameters B and C which are proportional to the number of layers and filters, respectively.

The reproducible code and experiment details are available in benchmarks/unet-memory.

AmoebaNet-D (L, D) Memory Benchmark

Experiment	baseline	pipeline-1	pipeline-2	pipeline-4	pipeline-8
AmoebaNet-D (L, D)	(18, 208)	(18, 416)	(18, 544)	(36, 544)	(72, 512)
torchgpipe					
Parameters	81.5M	319.0M	542.7M	1.06B	1.84B
Model Memory	0.91 GiB	3.57 GiB	6.07 GiB	11.80 GiB	20.62 GiB
Peak Activation Memory	Out of memory	0.91 GiB	3.39 GiB	6.91 GiB	10.83 GiB
Huang et al.					
Parameters	82M	318M	542M	1.05B	1.8B
Model Memory	1.05GB	3.8GB	6.45GB	12.53GB	24.62GB
Peak Activation Memory	6.26GB	3.46GB	8.11GB	15.21GB	26.24GB

The table shows the better memory utilization of AmoebaNet-D with GPipe, as stated in Table 1 of GPipe by Huang et al. The size of an AmoebaNet-D model is determined by two hyperparameters L and D which are proportional to the number of layers and filters, respectively.

We reproduced the same settings in the paper with regardless of memory capacity of Tesla P40 GPUs. The reproducible code and experiment details are available in benchmarks/amoebanetd-memory.

2.4.3 Speed

U-Net (5, 64) Speed Benchmark

Experiment	Throughput	Speed up
baseline	28.500/s	1×
pipeline-1	24.456/s	0.858×
pipeline-2	35.502/s	1.246×
pipeline-4	67.042/s	2.352×
pipeline-8	88.497/s	3.105×

To verify efficiency with skip connections, we measured the throughput of U-Net with various number of devices. We chose to use U-Net since it has several long skip connections.

The reproducible code and experiment details are available in benchmarks/unet-speed.

AmoebaNet-D (18, 256) Speed Benchmark

Table 1: (*n*: number of partitions, *m*: number of micro-batches)

Experiment	Throughput	torchgpipe	Huang et al.
n=2, m=1	26.733/s	1×	1×
n=2, m=4	41.133/s	1.539×	1.07×
n=2, m=32	47.386/s	1.773×	1.21×
n=4, m=1	26.827/s	1.004×	1.13×
n=4, m=4	44.543/s	1.666×	1.26×
n=4, m=32	72.412/s	2.709×	1.84×
n=8, m=1	24.918/s	0.932×	1.38×
n=8, m=4	70.065/s	2.621×	1.72×
n=8, m=32	132.413/s	4.953×	3.48×

The table shows the reproduced speed benchmark on AmoebaNet-D (18, 256), as reported in Table 2 of GPipe by Huang et al. Note that we replaced *K* in the paper with *n*.

The reproducible code and experiment details are available in benchmarks/amoebanetd-speed.

ResNet-101 Speed Benchmark

Experiment	Throughput	torchgpipe	Huang et al.
baseline	95.862/s	1×	1×
pipeline-1	81.796/s	0.853×	0.80×
pipeline-2	135.539/s	1.414×	1.42×
pipeline-4	265.958/s	2.774×	2.18×
pipeline-8	411.662/s	4.294×	2.89×

The table shows the reproduced speed benchmark on ResNet-101, as reported in Figure 3(b) of the fourth version of GPipe by Huang et al.

The reproducible code and experiment details are available in benchmarks/resnet101-speed.

2.5 Changelog

2.5.1 v0.0.7

Released on September 18, 2020.

Changed the license to BSD-3-Clause.

2.5.2 v0.0.6

Released on July 29, 2020.

- Updated docs.
- Added support for PyTorch 1.5.

2.5.3 v0.0.5

Released on November 29, 2019.

Featured: *@skippable* for efficient skip connections. With this interface, *GPipe* copies skip tensors directly to the destination device.

Improvements:

- Checkpointing deterministically handles randomness managed by PyTorch.
- *balance_by_size()* analyzes parameters as well.

Breaking Changes:

- Moved `torchgpipe_balancing` module to *torchgpipe.balance*.
- Redesigned interface of *balance_by_time()* and *balance_by_size()*.

2.5.4 v0.0.4

Released on October 8, 2019.

- Reduced GPU memory fragmentation by caching CUDA streams for copy.
- Fixed potential GPU memory violation on tuple of multiple tensors.
- Fixed potential GPU memory violation on shifted view tensors. (issue #27366 and pull request #27371 on PyTorch)

2.5.5 v0.0.3

Released on September 30, 2019.

Featured: *torchgpipe* now overlaps copy and computation using the separate CUDA streams. Previously, GPU could not compute a partition while copying micro-batches across different GPUs because they all happened on the same default CUDA stream.

Other Improvements:

- Added support for PyTorch 1.2.

- Redesigned the internal pipeline parallelism to represent dependencies transparently.
- Reduced memory usage for backpropagation by forgetting recomputation results at the right time.
- Fixed the hanging issue when an exception is raised in a partition.
- Fixed the unintended size accumulation (issue #3 by Shiyan Deng) of `balance_by_size()`.

Breaking Changes:

- No more support for PyTorch 1.0.
- Changed type of *`GPipe.devices`* from `tuple` to `list`.
- Removed `current_microbatch`. This approach turned out to be incompatible with checkpointing.

2.5.6 v0.0.2

Released on June 26, 2019.

- Added support for PyTorch 1.1.
- Refined public APIs.
- Detailed documentation.
- Proper exceptions for invalid usage.
- Provided *automatic balancing*.
- Provided inspecting utilities: `current_microbatch` (DO NOT USE, deprecated since v0.0.3) and *`is_recomputing()`*
- Reimplemented deferred batch normalization by subclassing.

2.5.7 v0.0.1

Released on May 14, 2019 to evaluate usability and efficiency internally.

- Provided a functional GPipe implementation, including pipeline parallelism, checkpointing, and deferred batch normalization.
- Supported Python 3.6+ and PyTorch 1.0.

CHAPTER

THREE

AUTHORS AND LICENSING

This project is developed by Heungsub Lee, Myungryong Jeong, and Chiheon Kim at Kakao Brain, with Sungbin Lim, Ildoo Kim, Woonhyuk Baek, and Boogeon Yoon's help. It is distributed under the 3-clause BSD license.

If you apply this library to any project and research, please cite our code:

```
@article{kim2020torchgpipe,
    title={torchgpipe: On-the-fly Pipeline Parallelism for Training Giant Models},
    author={Chiheon Kim and Heungsub Lee and Myungryong Jeong and Woonhyuk Baek and
↪Boogeon Yoon and Ildoo Kim and Sungbin Lim and Sungwoong Kim},
    year={2020},
    eprint={2004.09910},
    archivePrefix={arXiv}
}
```

PYTHON MODULE INDEX

t

INDEX