torchgpipe

Release 0.0.2

Jun 26, 2019

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 1

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 2

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 1.0+ 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 `nn.Sequential` as GPipe will automatically split the module into partitions with consecutive layers. *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 four partitions each having a single layer. 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=[1, 1, 1, 1], chunks=8)

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 select by *devices* parameter:

```
mode = GPipe(model,
             balance=[1, 1, 1, 1],
             devices=[4, 5, 6, 7],  # Specify GPUs.
             chunks=8)
```

2.2.3 Input and Output Device

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

Therefore, you should 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()
    ...
```

2.2.4 Automatic Balancing

It could be hard to determine the optimal balance of a model. In particular, if you are still designing a model, probably the model architecture may change over time. In this case, we highly recommend `torchgpipe_balancing` for

automatic balancing. This library is also a part of *torchgpipe* package but not a part of the GPipe paper.

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 a balancing tool for your needs:

```
from torchgpipe import GPipe
from torchgpipe_balancing import balance_by_time

sample = torch.rand(128, 3, 224, 224)
balance = balance_by_time(model, sample, partitions=4)

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

2.2.5 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: non-sequential module cannot 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: ...
```

(continues on next page)

(continued from previous page)

```
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.

2.2.6 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 or AmoebaNet, 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 will 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: Tensor) -> Tuple[Tensor, Tensor]:
        return layer1(input), input

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

class Layer3(nn.Module):
    #          ┌──────────────────┐
    # input --┤---> layer3 --+-┤--> output
    #  skip --┤--------------' │
    #          └──────────────────┘
    def forward(self, input_and_skip: Tuple[Tensor, Tensor]) -> Tensor:
        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)
```

It is the most straightforward approach to implement skip connections. But there is a disadvantage. In the above example, the skipping input tensor is copied to the second device, but it is never used at the device. Unnecessarily copied tensor wastes time and memory.

The following section introduces alternative approach for skip connection.

Long Skip Connections

The disadvantage mentioned above might be catastrophic if the unnecessarily copied tensor is very large, or it is copied 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, GPipe will copy the skipping input tensor to all devices, but 6 of them are unnecessary. The alternative approach is managing the skipping tensor manually in the module code. Now we will introduce a shared memory between `Layer1` and `Layer8` to toss the tensor without going through regardless layers. We might use a global variable named `skip_buf` for the shared memory. But actually, this approach doesn't work:

```
# !!!!!!!!!!!!!!!!!!!!!!!!
# THIS IS A FAILING EXAMPLE
# !!!!!!!!!!!!!!!!!!!!!!!!

# The shared memory between Layer1 and Layer8.
skip_buf = None

class Layer1(nn.Module):
    def forward(self, input: Tensor) -> Tensor:
        # Remember the skipping tensor.
        global skip_buf
        skip_buf = input

        return layer1(input)

class Layer2(nn.Module):
    def forward(self, input: Tensor) -> Tensor:
        return layer2(input)

...  # Layer3-7 are similar to Layer2.

class Layer8(nn.Module):
    def forward(self, input: Tensor) -> Tensor:
        # Retrieve the skipping tensor.
        global skip_buf
        skip = skip_buf

        # Release the shared memory.
```

(continues on next page)

(continued from previous page)

```
        skip_buf = None

        # The tensor should be copied to the device manually.
        skip = skip.to(input.device)

        return layer8(input) + skip
```

Each layer is executed several times due to the multiple micro-batches. Partitions work together concurrently, so the shared memory would be overwritten in non-deterministic order.

For example, when `Layer8` processes the first micro-batch, it might receive the third (or any) micro-batch as a skipping tensor if `Layer1` has just processed the latter micro-batch. We need to separate the shared memory for different micro-batches.

Therefore, the key is an identifier of each micro-batch. How do we identify which micro-batch the partition is currently processing? `torchgpipe` provides *`current_microbatch()`* for this purpose. If you call the function in a GPipe context, it will return some tensor. The tensor identifies the current micro-batch. You can use this as simply a dictionary key, or the target of a weak reference:

```
from torchgpipe import current_microbatch

# The shared memory between Layer1 and Layer8 indexed by micro-batch.
skips: Dict[Tensor, Tensor] = {}

class Layer1(nn.Module):
    def forward(self, input: Tensor) -> Tensor:
        # Remember the skipping tensor per micro-batch.
        skips[current_microbatch()] = input

        return layer1(input)

...  # Layer2-7 are folded.

class Layer8(nn.Module):
    def forward(self, input: Tensor) -> Tensor:
        # Retrieve the skipping tensor for the current micro-batch.
        skip = skips.pop(current_microbatch())

        # The tensor should be copied to the device manually.
        skip = skip.to(input.device)

        return layer8(input) + skip
```

This approach is not required to everyone. Furthermore, we didn't intend to modify user's module to apply GPipe. However, long skip connections are one of the common building blocks in modern CNN models. That is why we have provided this functionality.

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 buffers on each forward propagation. It should not update the buffers again during the recomputation. To achieve it, modules' `forward` method should be able to detect that is the recomputation or the first forward progagation.

It can be done by *`is_recomputing()`*. This function returns `True` if the code is running on 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 `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** (*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 BatchNorm for more details)

Raises

- `TypeError` – the module is not a `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** (*tensor or tensors*) – input mini-batch

Returns output mini-batch

Return type tensor or tensors

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

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)
```

2.3.2 Inspecting GPipe Timeline

torchgpipe.**current_microbatch**()

Gets the current micro-batch identifier as a tensor.

If your module relies on where the current micro-batch lane, use it to identify the lane.

Returns A tensor which identifies the current micro-batch lane, or `None` for out of a GPipe context.

Return type tensor or `None`

See also:

Long Skip Connections

torchgpipe.**is_recomputing**()

Whether if the current thread is under checkpoint recomputation.

Returns `True` if it's under checkpoint recomputation.

Return type bool

See also:

Detecting Recomputation

2.3.3 Automatic Balancing

torchgpipe_balancing.**balance_by_time**(*module*, *canary*, *partitions*, *device*, *timeout*)

Balances the given seqeuntial module by elapsed time per layer.

Parameters

- **module** (*nn.Sequential*) – sequential module to be partitioned

- **sample** (*Tensor*) – example input

Keyword Arguments

- **partitions** (*int*) – intended number of partitions (default: `1`)
- **device** (*torch.device*) – CUDA device where the module is profiled (default: any related CUDA device or `torch.device('cuda')`)
- **timeout** (*float*) – profiling iterates again if the timeout (in second) is not exceeded (default: `1.0`)

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

`torchgpipe_balancing.`**balance_by_size**(*module*, *canary*, *partitions*, *device*)

Balances the given seqeuntial module by memory usage per layer.

Note: This function relies on `torch.cuda.reset_max_memory_allocated()` which is introduced at PyTorch 1.1. Therefore, it doesn't support neither CPU tensors nor PyTorch 1.0.x.

Parameters

- **module** (*nn.Sequential*) – sequential module to be partitioned
- **sample** (*Tensor*) – example input

Keyword Arguments

- **partitions** (*int*) – intended number of partitions (default: `1`)
- **device** (*torch.device*) – CUDA device where the module is profiled (default: any related CUDA device or `torch.device('cuda')`)

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

2.4 Benchmarks

2.4.1 ResNet-101

ResNet-101 Performance Benchmark

Experiment	Throughput	Speedup
naive-1	100.506 samples/sec	1.000x
pipeline-1	73.925 samples/sec	0.736x
pipeline-2	135.691 samples/sec	1.350x
pipeline-4	230.216 samples/sec	2.291x
pipeline-8	312.945 samples/sec	3.114x

The code which is reproducible on Tesla P40 GPUs, and the experiment details can be found in examples/resnet101_performance_benchmark.

2.4.2 AmoebaNet-D

AmoebaNet-D Memory Benchmark

Experiment	AmoebaNet-D (L, F)	# of Model Parameters	Total Model Parameter Memory	Total Peak Activation Memory
naive-1	(6, 208)	90M	1.00GB	–
pipeline-1	(6, 416)	358M	4.01GB	6.64GB
pipeline-2	(6, 544)	613M	6.45GB	11.31GB
pipeline-4	(12, 544)	1.16B	13.00GB	18.72GB
pipeline-8	(24, 512)	2.01B	22.42GB	35.78GB

2.5 Changelog

2.5.1 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()` and `is_recomputing()`
- Reimplemented deferred batch normalization by subclassing.

2.5.2 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 3

Authors and Licensing

This project is developed by Heungsub Lee and Myungryong Jeong at Kakao Brain, with Sungbin Lim, Chiheon Kim, Ildoo Kim, and Woonhyuk Baek's help. It is distributed under Apache License 2.0.

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

```
@misc{torchgpipe,
  author       = {Kakao Brain},
  title        = {torchgpipe, {A} {GPipe} implementation in {PyTorch}},
  howpublished = {\url{https://github.com/kakaobrain/torchgpipe}},
  year         = {2019}
}
```

Index