#

Zero-DP Memory Optimization

This is an implementation of Zero-DP introduced in the paper ZeRO: Memory Optimization Towards Training A Trillion Parameter Models,

It keeps shards of the optimizer state, gradients and parameters into multiple devices/nodes. It reduces the memory consumption to $\frac{(2+2+K)\Psi}{N_d}$ of the original model, where Ψ is the number of parameters, N_d is the number of shards, and K is number of optimizer bytes per parameter. $2+2$ are the parameter and gradient memory assuming 16-bit precision; i.e. 2 bytes per parameter and gradient. $K = 12$ for Adam optimizer because it maintains a copy of parameters, and two moments per parameter in fp32.

The communication volume of Zero-DP is $\mathcal{O}(3\Psi)$. For comparison data-parallel training has a communication volume of $\mathcal{O}(2\Psi)$.

```python
import functools
from typing import List, Optional, Tuple

import torch
import torch.distributed as dist
from torch import nn
```

Although this is named `Zero3` , we have only implemented the Zero-DP part of it and not the Zero-R memory optimizations which target residual memory consumption. Out implementation supports training only a subset of parameters.

This implementation is inspired by Fairscale FSDP.

Here's a script to fine-tune GPT NeoX using Zero-DP memory optimization.

Zero3 Layer

```
class Zero3Layer(nn.Module):
```

Each layer of the model (or a combination of a few consecutive layers) should be wrapped in this module.

Each shard keeps parameters in `chunk` list. The `chunk[0]` is for trainable parameters and `chunk[1]` is for fixed parameters.

```
    chunk: List[nn.Parameter]
```

This is the sizes of the chunks in `chunk` list.

```
    chunk_size: List[int]
```

The first chunk is for trainable parameters.

```
    TRAINING_PARAMS_IDX = 0
```

This is the list of parameters split into lists as trainable and fixed parameters.

```
    param_refs: List[List[nn.Parameter]]
```

CUDA stream to featch parameters

```
    fetch_stream: Optional[torch.cuda.Stream]
```

CUDA stream to backup/accumulate gradients

```
    backup_stream: Optional[torch.cuda.Stream]
```

List of layers right before this layer

```
    prev_layer: List['Zero3Layer']
```

List of layers right after this layer

```
    next_layer: List['Zero3Layer']
```

The position of the current layer; used this for debugging logs

```
    layer_idx: int
```

Whether parameters have been fetched

```
    is_fetched: bool
```

Device of the layer

```
    device: torch.device
```

Data type of the layer

```
    dtype: torch.dtype
```

The module to be wrapped

```
    module: nn.Module
```

Number of nodes/devices the data is sharded across

```
    world_size: int
```

- `module` The module to be wrapped.
- `rank` The rank of the current node.
- `world_size` The number of nodes/devices the data is sharded across.
- `device` The device of the layer.
- `dtype` The data type of the layer.

```
    def __init__(self, module: nn.Module, rank: int, world_size: int, device: torch.device, dtype: torch.dtype):
```

```python
        super().__init__()
```

Initialize the properties

```python
        self.device = device
        self.dtype = dtype
        self.module = module
        self.prev_layer = []
        self.next_layer = []
        self.is_fetched = False
        self.world_size = world_size
        self.layer_idx = -1
        self.fetch_stream = None
        self.backup_stream = None

        with torch.no_grad():
```

Collect all the parameters of the layer

```python
            all_param_refs = [p for p in self.parameters()]
```

Store the shape of the parameters because we need it later to reconstruct them

```python
            for p in all_param_refs:
                p._orig_shape = p.shape
```

All parameters should have the same type

```python
            for p in all_param_refs:
                assert p.dtype == dtype, "All parameters should have same dtype"
```

Separate parameters as trainable and fixed

```python
            self.param_refs = [[p for p in all_param_refs if p.requires_grad],
                               [p for p in all_param_refs if not p.requires_grad]]
            del all_param_refs
```

The `rank = 0` node will calculate the size each device/node should store, and distribute the parameters accordingly.

```python
            if rank == 0:
```

Merge and pad trainable (`merged_params[0]`) and fixed (`merged_params[1]`) parameters

```
            merged_params = [self._merge_and_pad_params(ps) for ps in self.param_refs]
```

Calculate the chunk sizes of trainable and fixed params

```
            self.chunk_size = [(len(p) // world_size if p is not None else 0) for p in merged_params]
```

Broadcast the sizes

```
            dist.broadcast(torch.tensor(self.chunk_size, device=device), src=0)
        else:
```

Create an empty tensor to receive the sizes

```
            chunk_size = torch.tensor([0, 0], device=device)
```

Receive the sizes

```
            dist.broadcast(chunk_size, src=0)
            self.chunk_size = chunk_size.tolist()
```

Create parameters for trainable (`self.chunk[0]`) and fixed (`self.chunk[1]`) parameters to be stored in current device/node

```
        self.chunk = [nn.Parameter(self._empty((s,)), requires_grad=i == self.TRAINING_PARAMS_IDX)
                      for i, s in enumerate(self.chunk_size)]
```

An empty tensor to receive the trainable and fixed parameters combined

```
        chunk = self._empty((sum(self.chunk_size),))

        if rank == 0:
```

Concatenate both trainable and fixed params

```
            all_params = torch.cat([p.view(world_size, -1) for p in merged_params], dim=-1).view(-1)
            del merged_params
```

Scatter them to all the nodes/devices

```
            dist.scatter(chunk, list(all_params.split(sum(self.chunk_size))))
            del all_params
        else:
```

Receive the parameters

```
                dist.scatter(chunk)
```

Collect the chunk data

```
            chunk = chunk.split(self.chunk_size)
            for i, c in enumerate(chunk):
                self.chunk[i].data[:] = c
            del chunk
```

Cleanup the normal parameters

```
            self._cleanup_params()
```

Add a backward hook. This gets called when the gradients relative to the module are computed.

```
            self._backward_hook_ref = self.register_full_backward_hook(self._backward_hook)  # type: ignore
```

Merge all the parameters and pad it so that it's divisible by `world_size`.

```
    def _merge_and_pad_params(self, params: List[nn.Parameter]) -> torch.Tensor:
```

Total number of parameters

```
        size = sum(p.shape.numel() for p in params)
```

If it is not divisible by `world_size`, pad it

```
        if size % self.world_size != 0:
            padding_fixed = self.world_size - (size % self.world_size)
```

Otherwise, no need to pad

```
        else:
            padding_fixed = 0
```

Create an empty padding tensor

```
        padding = self._empty((padding_fixed,))
```

Concatenate all the parameters and pad it

```
        return torch.cat([p.view(-1) for p in params] + [padding], dim=0)
```

Get trainable chunk/shard of the parameters.

```
    def get_trainable_chunk(self) -> List[nn.Parameter]:
```

This is what we pass on to the optimizer on the current node.

Return and empty list if there are no trainable parameters

```
        if len(self.chunk[self.TRAINING_PARAMS_IDX]) == 0:
            return []
```

Return the trainable chunk as a list

```
        return [self.chunk[self.TRAINING_PARAMS_IDX]]
```

Create an empty tensor of the given shape.

```
    def _empty(self, shape: Tuple[int, ...]) -> torch.Tensor:
```

```
        return torch.empty(shape, device=self.device, dtype=self.dtype)
```

Cleanup the parameter data

This will release all the memory used by the layer parameters.

```
    @torch.no_grad()
    def _cleanup_params(self):
```

Set the flag to indicate that the parameters are not fetched

```
        self.is_fetched = False
```

Iterate through all parameters

```
        for ps in self.param_refs:
            for p in ps:
```

Wait for operations on the parameters to complete before any new operations

```
                p.data.record_stream(torch.cuda.current_stream())
```

Check to make sure the parameter is not sharing storage with anything else

```
                assert p.data.storage_offset() == 0, "The tensor is not the sole occupant of the storage."
```

```
                p.data.storage().resize_(0)  # This is what actually clears the memory
```

Resize the storage to 0. This will release the memory used by the parameter.

Setting `p.data` will not release the memory, since the autograd graph keeps a reference to it.

Make sure the parameter has no gradient data

```
            assert p.grad is None, 'Gradients should be None'
```

Fetch the parameters from all shards

This will fetch all the parameter data from all the nodes and rebuild the parameters on each node.

```
@torch.no_grad()
def fetch_params(self):
```

Skip is already fetched

```
    if self.is_fetched:
        return
```

Set the flag

```
    self.is_fetched = True
```

Skip if there's nothing to fetch or share.

```
    if sum(self.chunk_size) == 0:
        return
```

Use `fetch_stream` to fetch the parameters from all the shards

```
    with torch.cuda.stream(self.fetch_stream):
```

Create an empty tensor to receive the parameters

```
        buffer = self._empty((self.world_size * sum(self.chunk_size),))
```

Split the continuous buffer into the number of nodes. These splits are views of `buffer'.

```
        buffers = list(buffer.split(sum(self.chunk_size)))
```

Concatenate both trainable and fixed chunks

```
        chunk = torch.cat(self.chunk, dim=0)
```

Gather the parameters from all the nodes/devices

```
        dist.all_gather(buffers, chunk)
```

Split the gathered parameters into the trainable and fixed chunks

```
        params = buffer.view(-1, sum(self.chunk_size)).split(self.chunk_size, dim=1)
```

Wait for the gather operation to complete and then clear the references to the buffers

```
        buffer.record_stream(self.fetch_stream)
        for b in buffers:
            b.record_stream(self.fetch_stream)
        buffer.record_stream(self.fetch_stream)
        del buffer
        del buffers
```

Reshape the trainable and fixed parameters to continuous tensors

```
        params = [p.reshape(-1) for p in params]
```

Collect the individual parameter tensors

```
        for cont, ps in zip(params, self.param_refs):
```

If there are no parameters, skip

```
            if not ps:
                continue
```

Offset of the continuous tensor

```
            offset = 0
```

Iterate through model parameters and assign the values from the continuous tensor

```
            for p in ps:
```

Original parameter shape

```
                shape = p._orig_shape  # type: ignore[attr-defined]
```

Change the storage size of the parameter. This was set to 0 when we cleaned up the parameters.

```
                p.data.storage().resize_(shape.numel())
```

Assign the values from the continuous tensor

```
                p.data[:] = cont[offset: offset + shape.numel()].reshape(shape)
```

Wait for the operations to complete before other operations can be performed

```
                p.data.record_stream(self.fetch_stream)
```

Update the offset

```
                offset += shape.numel()
```

Wait for the operation to complete before other operations can be performed

```
            cont.record_stream(self.fetch_stream)
```

```
        del params
```

Forward pass

```
def forward(self, *args, **kwargs):
```

Fetch all the parameters of the current node. This gets called by the previous layer so this call is just to make sure parameters are fetched.

```
    self.fetch_params()
```

Wait for parameter fetching to complete.

```
    torch.cuda.current_stream().wait_stream(self.fetch_stream)
```

Start fetching parameters of the proceeding layers, so that they will fetch them which the current layer does its computations.

```
    for layer in self.next_layer:
        layer.fetch_params()
```

Add backward hooks to the parameters of the current layer if autograd is enabled.

```
    if torch.is_grad_enabled():
        self._add_backward_hooks()
```

Compute the outputs of the current layer

```
    res = self.module(*args, **kwargs)
```

Cleanup the parameters of the layer.

Skip cleaning up if autograd is enabled and this is the last layer in the network, because we will need to fetch the parameters again for the backward pass.

```
        if not torch.is_grad_enabled() or self.next_layer:
            self._cleanup_params()

        return res
```

Add backward hooks to the parameters of the current layer.

```
    def _add_backward_hooks(self):
```

Number of backward hooks added

```
        self._backward_hook_handles = 0
```

Loop through trainable parameters of the current layer

```
        for p in self.param_refs[self.TRAINING_PARAMS_IDX]:
```

Make sure a hook hasn't already been added

```
            assert not hasattr(p, "_hook_handle"), 'Parameter has already been hooked'
```

Use `expand_as` to create an autograd step which we can intercept

```
            p_tmp = p.expand_as(p)
```

Get a handle to add the backward hook. [This blog discusses about `grad_acc`].

```
            grad_acc = p_tmp.grad_fn.next_functions[0][0]
```

Add the backward hook

```
            handle = grad_acc.register_hook(
                functools.partial(self._post_backward_hook, p))
```

Keep a reference to the handle

```
            p._hook_handle = handle
```

Increment the number of hooks added

```
            self._backward_hook_handles += 1
```

Handle a backward event

This gets called by parameter backward hooks and the module backward hook.

```python
    def _backward_event(self):
```

Decrement the hooks counter

```python
        self._backward_hook_handles -= 1
```

If all the hooks (including the module hook) have been called, then we can back up gradients and clean up the parameters.

```python
        if self._backward_hook_handles == -1:
            self._backup_grads()
            self._cleanup_params()
```

Start fetch parameters of the previous layer, because autograd will next process the gradients of it.

```python
        for layer in self.prev_layer:
            layer.fetch_params()
```

Parameter backward hook

```python
    def _post_backward_hook(self, p: nn.Parameter, *args):
```

Remove the handle from the parameter

```python
        p._hook_handle.remove()  # type: ignore[attr-defined]
        delattr(p, "_hook_handle")
```

Handle a backward event

```python
        self._backward_event()
```

Module backward hook

```python
    def _backward_hook(self, *args, **kwargs):
```

Handle a backward event

```python
        self._backward_event()
```

The previous layer will start computing gradients. We need to make sure it has finished fetching params.

```python
        torch.cuda.current_stream().wait_stream(self.fetch_stream)
```

```
        return None
```

Backup the gradients of the current layer

```
    @torch.no_grad()
    def _backup_grads(self):
```

Skip if there are no trainable parameters

```
        if self.chunk_size[self.TRAINING_PARAMS_IDX] == 0:
            return
```

Use the backup stream to backup the gradients

```
        with torch.cuda.stream(self.backup_stream):
```

Buffer to store the gradients

```
            buffer = self._empty((self.world_size * self.chunk_size[self.TRAINING_PARAMS_IDX],))
```

Split the continuous buffer into number of nodes. These splits are views of `buffer'.

```
            buffers = list(buffer.split(self.chunk_size[self.TRAINING_PARAMS_IDX]))
```

Offset of the continuous buffer

```
            offset = 0
```

Iterate through trainable parameters

```
            for p in self.param_refs[self.TRAINING_PARAMS_IDX]:
```

Collect gradients

```
                shape = p._orig_shape  # type: ignore[attr-defined]
                buffer[offset: offset + shape.numel()] = p.grad.view(-1)
```

Update the offset

```
                offset += shape.numel()
```

Clean the gradients

```
                p.grad = None
```

Empty tensor to accumulate the gradients of the current shard

```
            grad = self._empty((self.chunk_size[self.TRAINING_PARAMS_IDX],))
```

Accumulate the gradients of each shard. It scatters the buffers across the nodes, and each node accumulates (reduces) the tensors it receives.

```
            dist.reduce_scatter(grad, buffers)
```

Wait for the operation to complete and then clear the references to the buffers

```
            for b in buffers:
                b.record_stream(self.fetch_stream)
            buffer.record_stream(self.fetch_stream)
            del buffer
            del buffers
```

Set the chunk gradients. This is what the optimizer sees.

```
            self.chunk[self.TRAINING_PARAMS_IDX].grad = grad
            del grad
```

Sequential module for `Zero3Layer` layers

```
class Zero3Sequential(nn.Module):
```

- `modules` List of `Zero3Layer` layers

```
    def __init__(self, modules: List[Zero3Layer]):
```

```
        super().__init__()
```

CUDA stream to fetch parameters

```
        self.fetch_stream = torch.cuda.Stream()
```

CUDA stream to back up (accumulate) gradients

```
        self.backup_stream = torch.cuda.Stream()
```

Set the streams and preceding and proceeding layers for each `Zero3Layer` layer

```
        for i in range(len(modules)):
```

Set layer index

```
            modules[i].layer_idx = i
```

Set streams

```
            modules[i].fetch_stream = self.fetch_stream
            modules[i].backup_stream = self.backup_stream
```

Set proceeding layers

```
            if i + 1 < len(modules):
                modules[i].next_layer.append(modules[i + 1])
```

Set preceding layers

```
            if i - 1 >= 0:
                modules[i].prev_layer.append(modules[i - 1])
```

Store list of modules

```
        self.module_list = nn.ModuleList(modules)
```

```
    def get_trainable_chunk(self):
```

Return the list of trainable chunks from each layer

```
        return sum([m.get_trainable_chunk() for m in self.module_list], [])
```

```
    def forward(self, x: torch.Tensor):
```

Make sure gradient back up is complete

```
        torch.cuda.current_stream().wait_stream(self.backup_stream)
```

Forward pass

```
        for m in self.module_list:
            x = m(x)
```

```
        return x
```