

FluxManager

1. 总体架构

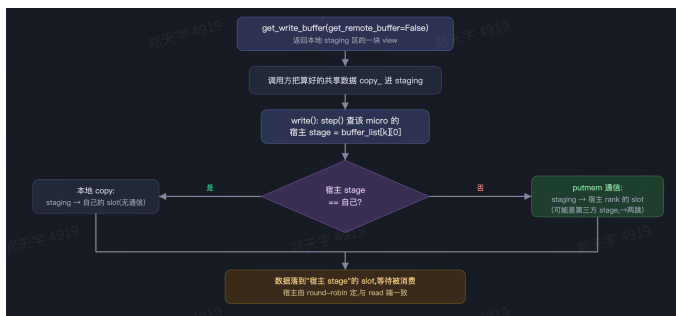
所有rank都初始化一整片连续的NVSHMEM，然后用offset切片管理：



其中staging area是本地临时工作区，而transfer slots则是和其他rank进行putmem/getmem的交换区，末尾还有一些额外的dsp_base的区域，用来做dsp的a2a。

普通路径：

写与读流程：



一般路线，我们在写的时候，要用到`get_write_buffer (get_remote_buffer=False)` 会拿到数据需要写的位置，在本地的staging area，然后把当前计算的共享数据写入，然后调用putmem，把当前共享数据写入dst位置，而这个dst位置不一定是当前层要share的层所在的rank，这里用了一个轮训的逻辑。

轮训的buffer设置在一个安全值： $\max(\text{pp_size}, \min(\text{num_micro}, \text{pp_size} * 2 + \text{num_micro} \% \text{pp_size} + 2))$ ，我感觉这里的意思是（假设不重算），首先group[0]和group[1]分别分pp_size个buffer，然后 $\text{num_micro} \% \text{pp_size}$ 表示余数，我们假设余数都分给group[0]和group[1]其实这就够了。+2不需要。

而在我们的训练场景，num_micro远大于pp_size，num_micro%pp<=num_micro//pp成立时，group[0]最大为pp_size+1，所以这里最小安全值，可以设置成 pp_size * 2+2。

这里轮训buffer的分配方式是，首先假定需要的安全大小为buffer_num，然后按照pp_stage来轮训分配，也就是buffer_num会被除以约pp_size，最终每个pp_stage会按pp_stage中最大的size来初始化每一个pp_stage这里的buffer（所以能看出来这里是有部分buffer被申请但用不上的），因此可以把buffer_num个slot想象成一个共享buffer，只不过分在不同的pp_stage上。但这里每一个不同的数据（hidden state、kv cache）在不同的from_layer都会创建一个这样的共享的buffer_num大小的区域，靠轮训的方式达到负载均衡，最终全部注册完毕，会按最大的size来初始化每个pp_stage的buffer，这样也能尽可能少的注册无效buffer（即没有用到的）。

而这里staging area对于每类数据，每个from layer只会注册一个对应数据size大小的区域（因为这是个临时区，fwd和bwd共用一个，除了mte embedding可能会fwd 和 bwd分别创建一个），这块区域主要是用来在write时，先分配staging的区域，让当前数据写入，然后再调用通信，将数据写入到，根据slot_offset计算出来的对应stage上的slot区；后续read时，从对应stage的slot区取数据，如果本身写在当前stage上，就不需要通信，否则通过通信获取数据，然后放在staging的区域来使用。

use_only_getmem_for_multi_stage_comm路径（其实应该叫only_putmem）：

1. buffer_size计算不同：

- 设置buffer_size的条件？

```
y = FluxManager._effective_n_layer - (max(to_layers) - from_layer)
if y < FluxManager.pp_size:
    return FluxManager.x1 + FluxManager.x2
if FluxManager._effective_n_layer - from_layer * 2 + 2 * FluxManager.pp_size <= 0:
    return FluxManager.x1
return FluxManager.x1 + FluxManager.x2
```

这里根据不同的调度规则来控制buffer_size，目前看是个安全值，其实不是最小安全值，可以简单算一下：



这里用 $n - \text{from_layer} * 2 + 2 * \text{pp_size} \leq 0$ ，来控制是否只需要x1，其实，转换过来是： $\text{from_layer} \geq n/2 + \text{pp_size}$ ，从图中我们可以看出来，其实在 $n/2 + 2$ 层的时候，已经可以只用x1来，具体这个怎么算？

这里其实简单算一下：

vpp_size : $n_layer // \text{pp_size}$

warmup_step: (vpp_size-1)*group[0].size()+(pp_size-1-pp_stage)*2。

group_micro_batch_idx_0: 每个micro_batch在第0个group内的位置。

group_0_step : group_0_size*vpp_size(第0个group的总step数)

- 当前层在group1中第一次出现的fwd位置(1f1b算1步):

group_micro_batch_idx_f_1 = group_0_step +
(group_micro_batch_idx_0//group_0.size())*group[1].size()+1

- 当前层在group0中第一次出现的bwd位置(1f1b算1步):

group_micro_batch_idx_b_0 = ((group_0_step -
group_micro_batch_idx_0)//group_0.size())*group_0.size()+warmup_step+1

- 最小buffer_size:

If group_micro_batch_idx_b_0-group_micro_batch_idx_f_1 < group[1].size() :

group[0].size()+group_micro_batch_idx_b_0-group_micro_batch_idx_f_1

else if group_micro_batch_idx_f_1-group_micro_batch_idx_b_0 < group[1].size() :

group[0].size()+group[1].size()-(group_micro_batch_idx_f_1 -
group_micro_batch_idx_b_0)

Else:

group[0]+group[1]

- x1+x2?

这里x1, x2就是pp最大的两个group_size, 在我们场景下, group_nums>2。

为什么是x1+x2:



看一下这个编排图中的a部分, 因为我们any1f1b的调度特点, 第一个group的最后一层的fwd会进入1F1B阶段, 这时候中间就是 f b 等量交错, 具体可以看any1f1b文档, 所以第二个group的f完成时, 也伴随着第一个group的b完成, 此时就可以覆盖第一个group用到buffer。

最坏的情况第二组fwd运行完, 第一组的bwd只运行一个, 但是这个一般不会, 因为any1f1b强制了每个group>=pp_size, 所以至少第二组的fwd运行完, 会运行pp_size个bwd, 而且后面再运行第三组的fwd的时候, 是跟运行的bwd成对出现的, 所以就是+-同样多的buffer, 因此最多就用group[0]+group[1]。

这里可以算一下，普通路径因为轮训共享buffer，所以buffer_size是 $2pp + \text{num_micro} \% pp_size + 2$ ，假设 $pp=4$ ， $\text{num_micro} \% pp_size=3$ ，即取最大， $\text{num_micro}=11$ 此时是13个buffer。又因为分散在不同pp_stage上共享，所以是 $13 + 4 - 1 // 4 = 4$ 个buffer

而如果是only_getmem模式，如果是one_to_many，此时是 $(x1+x2)$ ， $\text{num_micro}=11$ ，所以 $x1=6$ ， $x2=4$ ， $pp=4$ ，每个pp_stage都创建一样的buffer_size，就是10个buffer。

buffer大小跟实际的参数hidden_state大小相关。

2. read方式不同

```
if FluxManager.model_config.use_only_getmem_for_multi_stage_comm:
    assert src_pp_stage == -1
    src_pp_stage = FluxManager.pp_rank
    symm_buffer = FluxManager.symm_buffer
    num_bytes = src_end_byte - src_start_byte
    staging_start = FluxManager.staging_dict[(name, from_layer, is_forward)]
    if avoid_d2d:
        outputs = symm_buffer[src_start_byte:src_end_byte].view(dtype)
    else:
        symm_buffer[staging_start : staging_start + num_bytes].copy_(symm_buffer[src_start_byte:src_end_byte].view(dtype))
        outputs = symm_buffer[staging_start : staging_start + num_bytes].view(dtype)
    event = DummyEvent()

    if not is_recompute and is_forward and layer_index in bufferManager.to_layers[:-1]:
        putmem_kernel = FluxManager._get_putmem_kernel()
        dst_pp_stage = (FluxManager.pp_rank + 1) % FluxManager.pp_size
```

强制本地读，如果avoid_d2d，则直接读slot，非avoid_d2d只有 shortcut_to+yoco这一层，主要是bigop里面的一个操作，会在读取后直接原地写复用显存，因此需要copy到staging临时区，slot区需要保留。其余层都是avoid_d2d=True。同时如果当前read层不是最后一个consumer，则会把这份数据推给下一个层，即下一个pp_stage，接力传递，而不用跨层put。

3. write方式不同

```
class FluxManager:
    def get_write_buffer(
        name, from_layer, is_forward, dst_pp_stage, dst_start_byte, dst_end_byte, symm_buffer, num_bytes, staging_start
    ):
        bufferManager = FluxManager.buffer_dict[(name, from_layer)]

        if use_local_buffer:
            assert (
                len(bufferManager.to_layers) == 1
            ), "use_local_buffer is only supported for one-to-one BufferManagers (len(to_layers) == 1)"
            assert not get_remote_buffer, "use_local_buffer and get_remote_buffer are mutually exclusive"

            dst_pp_stage, dst_start_byte, dst_end_byte = bufferManager.step(layer_index, is_forward, symm_buffer)
            num_bytes = dst_end_byte - dst_start_byte
            staging_start = FluxManager.staging_dict[(name, from_layer, is_forward)]

            if use_local_buffer:
                # step(move=False, read=True) under multi-stage one-to-one returns this rank's
                # local slot. Hand that slot back to the caller directly so the subsequent
                # write() can putmem from it without going through the staging area.

                buffer_idx = bufferManager.buffer_idx_list[layer_index][is_forward]
                if is_forward:
                    dst_start_byte, dst_end_byte = bufferManager.forward_buffer_dict[FluxManager.to_layers[buffer_idx]]
                else:
                    dst_start_byte, dst_end_byte = bufferManager.backward_buffer_dict[FluxManager.to_layers[buffer_idx]]

                local_buffer = symm_buffer[dst_start_byte:dst_end_byte].view(dtype)
                event = torch.cuda.Event()
                event.record(FluxManager.TORCH_DEFAULT_STREAM)
                return local_buffer, event
```

- get_write_buffer:

getmem + one-to-one下，会直接返回slot区，把当前计算结果写入，后续write会直接从slot区使用putmem来写需要传递的数据。

getmem+ one-to-many，和普通路径一样，返回staging区域，会把结果写入到staging区。

```
buffer = symm_buffer[staging_start : staging_start + num_bytes].view(dtype)
if get_remote_buffer:
    if FluxManager.model_config.use_only_getmem_for_multi_stage_comm:
        assert dst_pp_stage == -1
        dst_pp_stage = FluxManager.pp_rank
        buffer.copy_(symm_buffer[dst_start_byte:dst_end_byte].view(dtype))
        return buffer, DummyEvent()
    else:
        if dst_pp_stage == FluxManager.pp_rank:
            buffer.copy_(symm_buffer[dst_start_byte:dst_end_byte].view(dtype))
            return buffer, DummyEvent()

        from .dist_triton_kernel import getmem_kernel

        peer = dst_pp_stage * FluxManager.ep_size + FluxManager.ep_rank
        grid_size = FluxManager._get_nvshmem_grid_size(peer)
        timers = get_timers() if get_env_config().megatron_use_cuda_timer else None
        real_rank = FluxManager._group_global_ranks[peer]
        torch.cuda.set_stream(FluxManager.NVSHMEM_STREAM)
        FluxManager.NVSHMEM_STREAM.wait_stream(FluxManager.TORCH_DEFAULT_STREAM)
        if timers is not None:
            timers('flux-read').start(
                # 郑天宇, last week * [megatron] fix: add m15 megatron
                stream=FluxManager.NVSHMEM_STREAM, tag={"peer_rank": real_rank, "bytes": num_bytes}
            )
        _ = getmem_kernel([grid_size,])(
            symm_buffer,
            symm_buffer,
            dst_start_byte,
            staging_start,
            num_bytes
        )
```

同时backward在getmem+one-to-many下，反传梯度时，因为正向传递是逐层传递，每次都是直接在本stage读，因此反传梯度时，首先把slot中下一层传回的梯度放到staging区，然后做_add累加梯度，再回传给上一层（回传在write中）。第一个回传的，直接覆盖清零，不累加，后续的层需要累加。

```
def write(
    num_bytes = dst_end_byte - dst_start_byte
    staging_start = FluxManager.staging_dict[(name, from_layer, is_forward)]

    if FluxManager.model_config.use_only_getmem_for_multi_stage_comm:
        assert dst_pp_stage == -1
        if len(bufferManager.to_layers) > 1:
            if is_forward:
                dst_pp_stage = (FluxManager.pp_rank + FluxManager.pp_size + 1) % FluxManager.pp_size
            else:
                dst_pp_stage = (FluxManager.pp_rank + FluxManager.pp_size - 1) % FluxManager.pp_size
        elif is_forward:
            dst_pp_stage = bufferManager.to_stages[0]
        else:
            dst_pp_stage = bufferManager.from_stage

    if dst_pp_stage == FluxManager.pp_rank:
        if not use_local_buffer:
            symm_buffer[dst_start_byte:dst_end_byte].copy_(symm_buffer[staging_start : staging_start + num_bytes])
            return DummyEvent()

    putmem_kernel = FluxManager._get_putmem_kernel()

    peer = dst_pp_stage * FluxManager.ep_size + FluxManager.ep_rank
    grid_size = FluxManager._get_nvshmem_grid_size(peer)
    timers = get_timers() if get_env_config().megatron_use_cuda_timer else None
    real_rank = FluxManager._group_global_ranks[peer]
    if FluxManager.model_config.use_only_getmem_for_multi_stage_comm:
        nvshmem_stream = bufferManager.nvshmem_stream
    else:
        # 郑天宇, last week * [megatron] fix: add m15 megatron ...
        nvshmem_stream = FluxManager.NVSHMEM_STREAM
```

- write时：

如果是one-to-one，且开了use_local_buffer，如果dst stage等于当前stage，即两个层在一个pp rank上，就直接从staging区copy到slot里，否则就直接调用 putmem 写在对应dst stage的slot里面，（use_local_buffer在hidden state都开了，kv都没开，因为kv是one-to-many）。

如果是one-to-many，就直接调用putmem写在下一层的slot里，但这里use_local_buffer一定为false，所以是承接上面get_write_buffer拿到的地址，从staging区直接写在下一层的slot里（注意

one-to-many共享的层，consumer一定跟producer连续），而后续共享层的写，就有read来代替了，每个consumer只会读自己slot区，不会再跨stage通信。

4. BufferManager独立stream+延迟等待 (EventManager)

```
def read(  
    from_layer, dtype = FluxManager.transfer_dict[(name, layer_index, is_forward, 'read')]  
    bufferManager = FluxManager.buffer_dict[(name, from_layer)]  
    src_pp_stage, src_start_byte, src_end_byte = bufferManager.step(  
        layer_index, is_forward, is_recompute=is_recompute  
    )  
    if FluxManager.model_config.use_only_getmem_for_multi_stage_comm:  
        assert src_pp_stage == -1  
        src_pp_stage = FluxManager.pp_rank  
        symm_buffer = FluxManager.symm_buffer  
        num_bytes = src_end_byte - src_start_byte  
        staging_start = FluxManager.staging_dict[(name, from_layer, is_forward)]  
        if avoid_d2d:  
            outputs = symm_buffer[src_start_byte:src_end_byte].view(dtype)  
        else:  
            symm_buffer[staging_start : staging_start + num_bytes].copy_(symm_buffer[src_start  
            outputs = symm_buffer[staging_start : staging_start + num_bytes].view(dtype)  
        event = DummyEvent()
```

```
class EventManager:  
    @staticmethod  
    def wait(handle, is_forward, name, layer_index, read_or_write):  
        if not FluxManager.is_init or not FluxManager.model_config.use_only_getmem_for_multi_stage:  
            handle.wait()  
            return  
        # read path: caller is the consumer, must wait synchronously.  
        if read_or_write == 'read' or read_or_write == 'get_write_buffer':  
            handle.wait()  
            return  
        # 'get_write_buffer' shares the 'write' transfer_dict entry; resolve  
        # both via the 'write' key so the caller doesn't need to know.  
        from_layer, _ = FluxManager.transfer_dict[(name, layer_index, is_forward, read_or_write)]  
        bufferManager = FluxManager.buffer_dict[(name, from_layer)]  
        # one-to-many: cross-stage forwarding chain, fall back to immediate wait.  
        if len(bufferManager.to_layers) > 1:  
            handle.wait()  
            return
```

在only_getmem模式下，这里read返回的是空event，因此需要有保障它能立即read到正确数据的event，对于one-to-many来说，wait是立即等待的，因为producer层发给第一个consumer是从直接从staging区putmem，这个临时区只有一份，需要保证完整写到dst的slot位置；而one-to-one，则只需要在consumer层消费前写入即可：

```
class EventManager:  
    @staticmethod  
    def flush(is_forward):  
        if not FluxManager.is_init or not FluxManager.model_config.use_only_getmem_for_multi_stage:  
            return  
        if is_forward:  
            bucket = EventManager._forward_handles  
            new_bucket = list[Any]()  
            for handle, expected_count in bucket:  
                assert expected_count >= EventManager._forward_count  
                if expected_count == EventManager._forward_count:  
                    handle.wait()  
                else:  
                    new_bucket.append((handle, expected_count))  
            EventManager._forward_handles = new_bucket  
        else:  
            return
```

```
if is_forward:  
    EventManager._forward_count += 1  
    assert EventManager._forward_idx < len(group_sizes), (  
        f"EventManager forward flush overflow: idx={EventManager._forward_idx} "  
        f"len(group_sizes)={len(group_sizes)}")  
    target = layers_per_pp * group_sizes[EventManager._forward_idx]  
    if EventManager._forward_count >= target:  
        assert EventManager._forward_count == target, (  
            f"EventManager forward count {EventManager._forward_count} exceeded "  
            f"target {target} for group idx {EventManager._forward_idx}")  
        EventManager._forward_idx += 1  
        EventManager._forward_count = 0
```

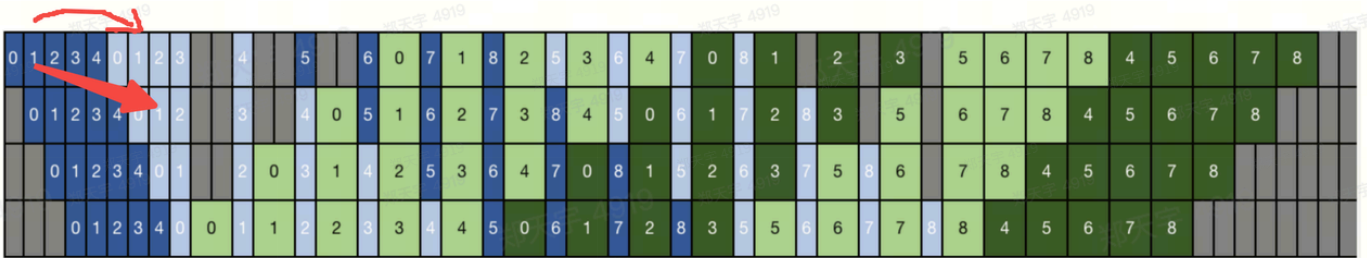
```

if name == 'mtp_emb':
    layer_chunk_count = FluxManager.model_config.num_hidden_layers // FluxManager.pp_size - 1
else:
    if layer_index == bufferManager.from_layer:
        peer_layer_index = bufferManager.to_layers[0]
    else:
        assert layer_index == bufferManager.to_layers[0], (
            f"layer_index {layer_index} not in one-to-one BufferManager "
            f"(from_layer={bufferManager.from_layer}, to_layers={bufferManager.to_layers})"
        )
        peer_layer_index = bufferManager.from_layer
    layer_chunk_count = (abs(peer_layer_index - layer_index) - 1) // FluxManager.pp_size

if is forward:
    expected_count = (
        layer_chunk_count * FluxManager.group_sizes[EventManager._forward_idx] + EventManager._forward_count
    )
    EventManager._forward_handles.append([handle, expected_count])
else:
    expected_count = (
        layer_chunk_count * FluxManager.group_sizes[EventManager._backward_idx] + EventManager._backward_count
    )

```

这里一个group内的micro_batch共用一个_forward_count，这个group内的每一个micro_batch跑完当前所有rank的所有层时清零，此时从**单个rank**的视角来看一个group_batch在当前rank上的所有层的forward刚好跑完，forward_idx标识当前的group_idx，所以这里layer_chunk_count是计算当前两个层间的距离经过了几个pp_size，因为每pp_size步，在单个pp_rank上会更换一层，所以layer_chunk_count*group_sizes，即这一组micro_batch经过了多少层的fwd，而每一个micro_batch经过一层的fwd时，forward_count会+1，当forward_count==expected_count时，就必须wait，而中间可以用通信和计算做overlap。



即如果pp_0要从第0层发到pp_1的第5层，以micro_batch_1为例，这时候需要在pp_0的下一层的micro_batch1做forward后，强制wait，因为后续层肯定要用到这份数据，需要确保已经putmem完成了。

5. Register sort

```

def register_finished():
    pp_size=FluxManager.pp_size,
    vpp_size=vpp_size,
)

if FluxManager.model_config.use_only_getmem_for_multi_stage_comm:
    # Sort registers by len(to_layers) descending: one-to-many first,
    # one-to-one last. One-to-many registers allocate slots on every
    # pp_stage and rely on all stages' offset counters being in sync;
    # one-to-one registers only touch from/to stages (sparse) and skip
    # advancing the other stages' counters. Allocating one-to-many first
    # keeps the cross-stage offsets aligned inside each one-to-many
    # register. Index 4 of the register tuple is ``to_layers``.
    FluxManager.register_list.sort(key=lambda r: len(r[4]), reverse=True)

```

需要对one-to-many，按layer长度大小排序，将one-to-one放到后面，因为one-to-many按层间接力传递，那就必须保证每个pp_stage的这块offset是相同的，而在one-to-one下只有from和to两个layer所在的stage会创建buffer，这样就会把buffer打乱，因此必须要排序。

6. 接入的一个异常问题总结与修复

接入m15的时候, loss后续step不能bitwise对齐: 第一步loss 能完全bitwise对齐, 而grad在第一步 3-9层是bitwise的, 而0-2层不能bitwise, 为什么? 是不是很神奇?

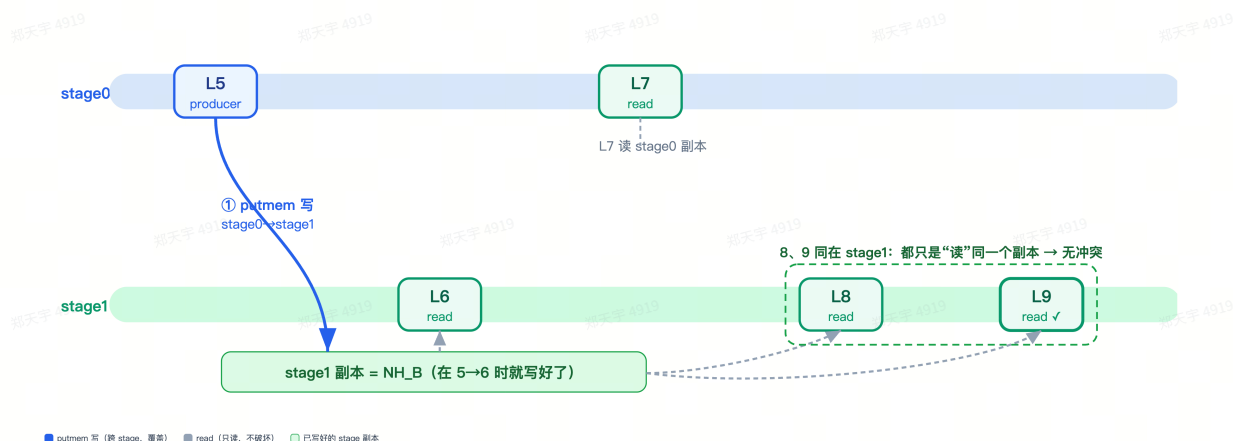


首先讲一下这里适配为什么m14对, m15错? 错在, 之前m14的时候,最后一层是share hidden state的, 前面的层one-to-many share kv cache时, 是连续的层。现在m15除了share kv cache的那些层, 最后一层也是share hidden state的, 而在开启only_getmem (其实应该是putmem), 会按照stage交替的逻辑来一层一层传数据。

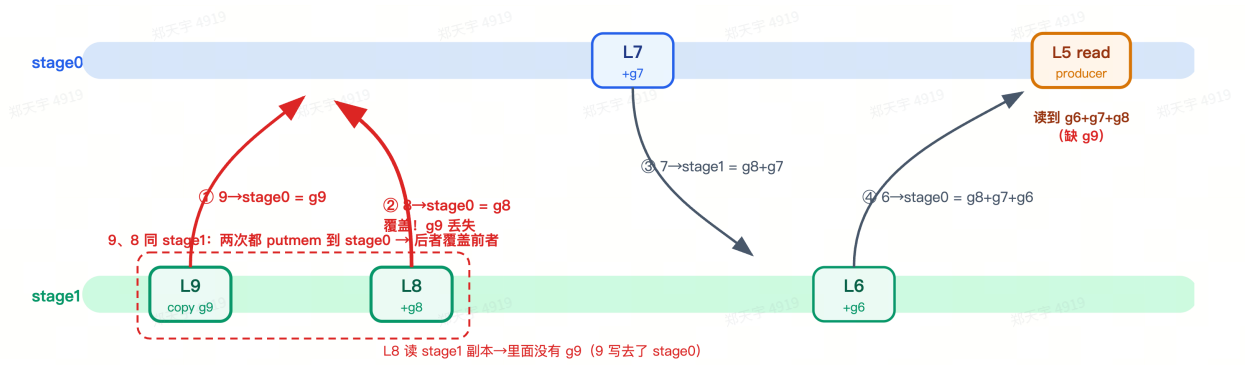
这里存在3个问题:

- a. fwd为什么正确?
- b. bwd为什么错误?
- c. layer5也是producer为什么grad是正常的?

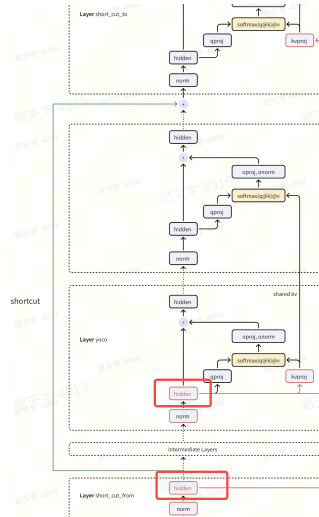
- o fwd过程为什么正确? :



- o bwd过程为什么错误? :



- 为什么 layer5也是 producer层，但是grad是正确的？

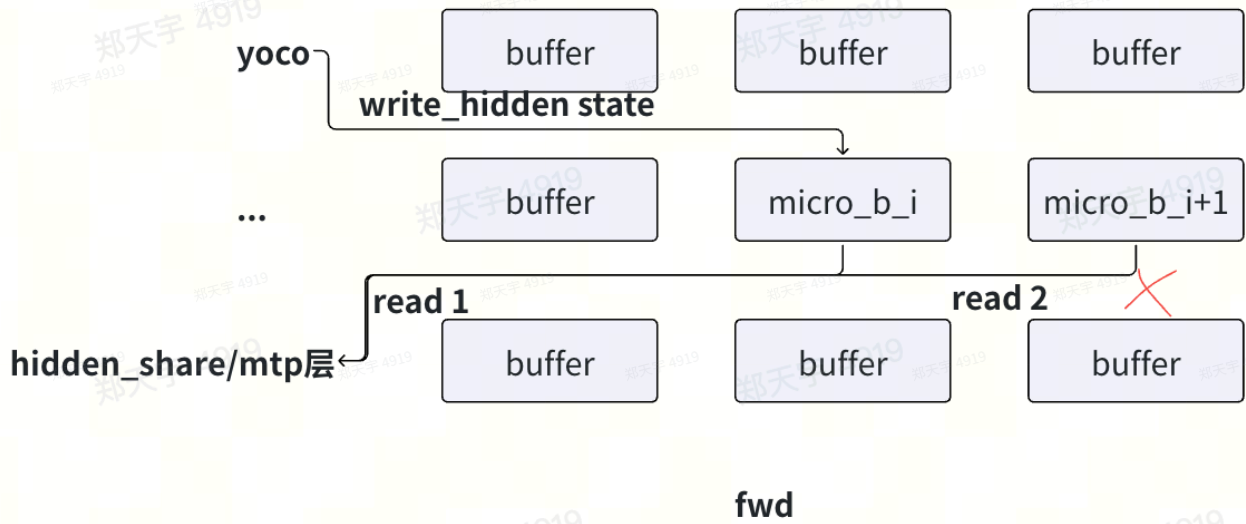


从图中可以看到，layer5虽然是producer，但是不用自己的数据produce，所以backward对它并没有影响。

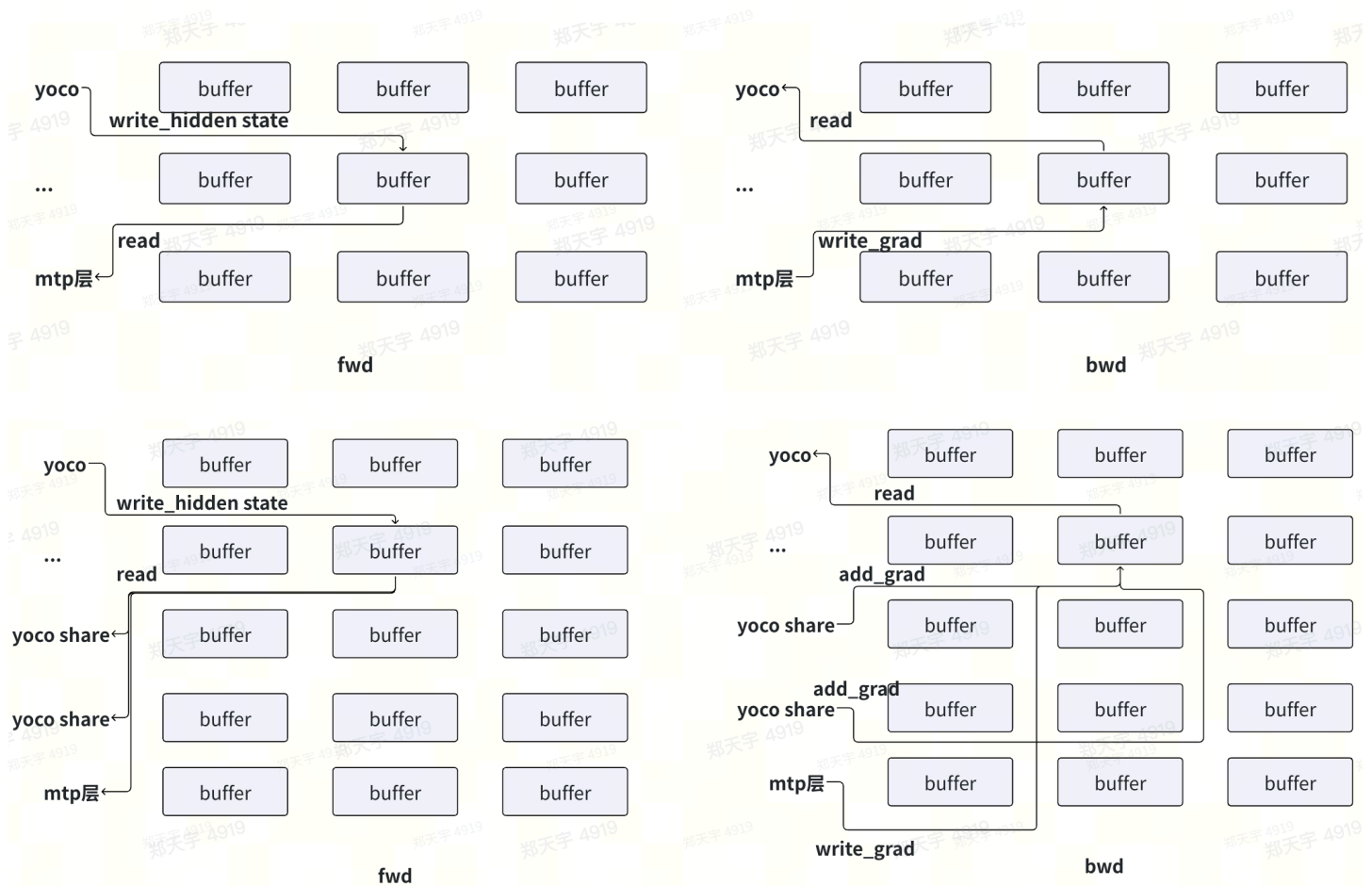
2. 两种fluxmanager接入m15存在的坑记录

Flux manager 显存均摊

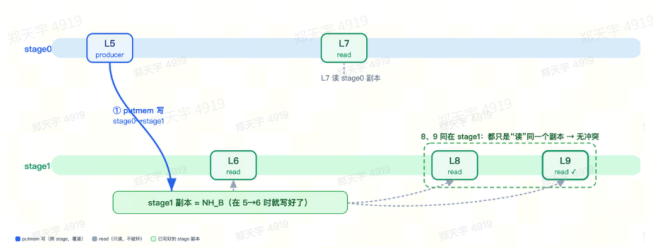
mtp层fwd行为变了：



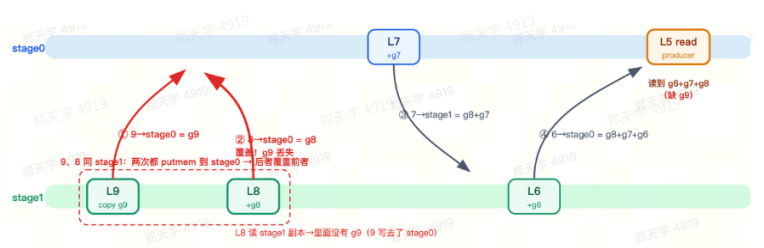
yoco share bwd行为变了:



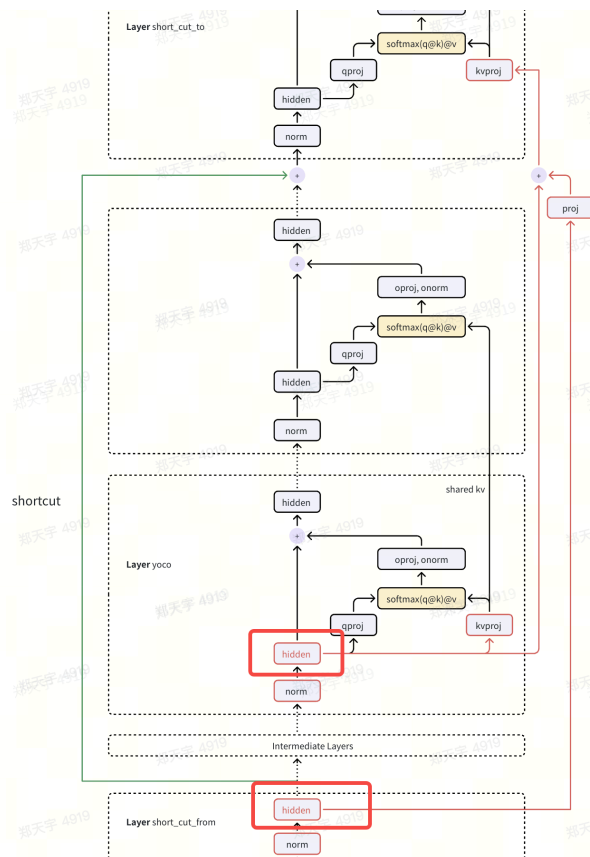
Flux manager only_putmem



fwd(层间轮训传递模式)



bwd(层间轮训传递模式)



不会影响layer5，因为short_to（第二个yoco层）只传递这份hidden state，来源跟layer5无关

3. 重要数据结构

- `transfer_dict`：回答“当前layer的 read/write 对应哪个 from_layer”。

代码块

```
1 key:
2   (name, current_layer, is_forward, read/write)
3 value:
4   (from_layer, dtype)
```

比如：

('normedhidden', 7, True, 'read') -> (3, bf16)

意思是 layer 7 forward 读 `normedhidden`，源通信项来自 layer 3。

- `staging_dict`：回答“这个通信项的本地临时区在哪”。

代码块

```
1 key:
2   (name, from_layer, is_forward)
3 value:
```

4 staging_start_byte

- `buffer_dict`：回答“这个通信项有哪些正式通信 slot”。

代码块

```
1 key:
2     (name, from_layer)
3 value:
4     BufferManager
```

- `buffer_idx_list`：在 `BufferManager` 里面，回答“当前 layer 当前方向这次用第几个 slot”。

代码块

```
1 buffer_idx_list[layer_index][is_forward]
```

完整流程是：

`FluxManager.read(name, current_layer, is_forward)`

1. `transfer_dict` 找 `from_layer`
2. `buffer_dict[(name, from_layer)]` 找 `BufferManager`
3. `BufferManager.step()` 选本次 formal slot
4. `staging_dict` 找本地 staging offset
5. formal slot -> staging
6. 返回 staging view

`write()` 类似：

`FluxManager.get_write_buffer(...)`

1. `transfer_dict` 找 `from_layer`
2. `staging_dict` 找 staging offset
3. 返回 staging view 给 caller 写

`FluxManager.write(...)`

1. `transfer_dict` 找 `from_layer`
2. `BufferManager.step()` 选本次 formal slot
3. staging -> formal slot

4. 不同角色对应关系

4.1 实现细节

4.1.1 YocoNormedHiddenHook

- fwd:

hidden_gen_layer和shortcut_to绑定，且目前都是1

normed_hidden_action_type	normed_hidden_write_to_buffer(只管理yoco，不管shortcut)	含义
0	False/True	用上一层传入进来的hidden state，不做任何处理/ 写入buffer第一个yoco层(目前只有2个yoco) kv计算的norm_hidden 来自当前层输入
1	False/True	consumer层/ shortcut_to+yoco 层（现在shortcut_to跟yoco强绑定了） kv计算的norm_hidden 来自buffer read
2	True（一定为True）	shortcut_from层 kv计算的norm_hidden 来自当前层输入

- bwd:

normed_hidden_action_type	normed_hidden_write_to_buffer(只管理yoco，不管shortcut)	bwd_grad_copy_to_buffer	含义

0	False/True	-	1. yoco前的普通层用下一层传入进来的grad，不做任何处理 2. 第一个yoco层(目前只有2个yoco)，读buffer，buffer里已经累加了后续share layer的grad。 需要返回grad给当前层
1	False/True	1. 普通consumer层 (除了shortcut_to和最后一层，均为False)	1. 普通consumer层使用_add，非最后一个消费层，需要累加到buffer，防止覆盖 2. 最后一层直接使用_copy，覆盖写入，不读。 3. short_to层使用_copy，覆盖写入，给from层读，from和to层1对1，并且需要读buffer，即share给其它层累加后的grad 不需要返回grad给当前层，grad返回None

		2. 最后一层和shortcut_to为True	
2	True（一定为True）	-	shortcut_from层，直接读shortcut_to层传来的grad 需要返回grad给当前层

YocoKVCacheHook 全部移除

3.2 FluxManager.register

之前是共享kv cache，只有mtp层和shortcut to 层会共享hidden state，因此是1对1的register，需要改成1对多，同时岂止点为：

起：前一个yoco 层的 下一层

止：下一个yoco层