ZOMI 大模型：分布式训练

流水并行代码剖析 Pipeline Parallel

大模型业务全流程

数据 & 模型算法

2. 数据处理

- 开源数据
- 数据预处理
- 向量数据库

3. 模型算法

- LLM模型架构
- 多模态一切皆Tokens

模型训练 & 微调

4. 模型训练

- 混合精度
- 梯度检查
- 梯度累积
- …
- 5. 分布式并行
- 训练集群稳定性

6. 模型微调

- 全参微调
- 低参微调
- 指令微调

模型验证 & 推理部署

7. 模型验证

- 下游任务
- 测评标准
- bench mark

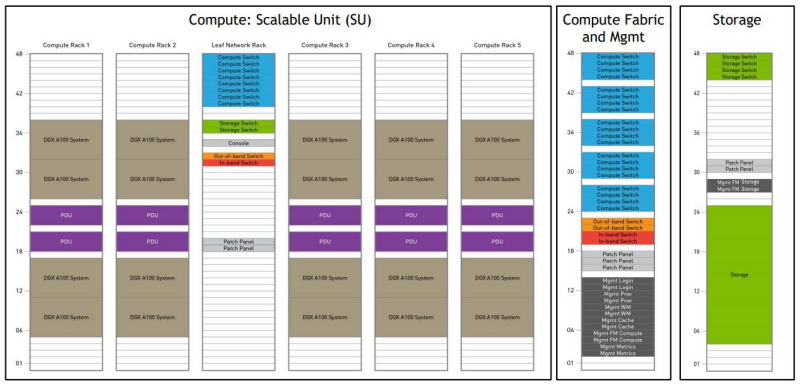
8. 推理与智能体

- 量化压缩
- 推理加速
- 9. Agent 智能体

1. AI 集群建设

- 计算、存储、网络
- AI 集群机房建设
- AI 集群上线与运维

集群算力准备

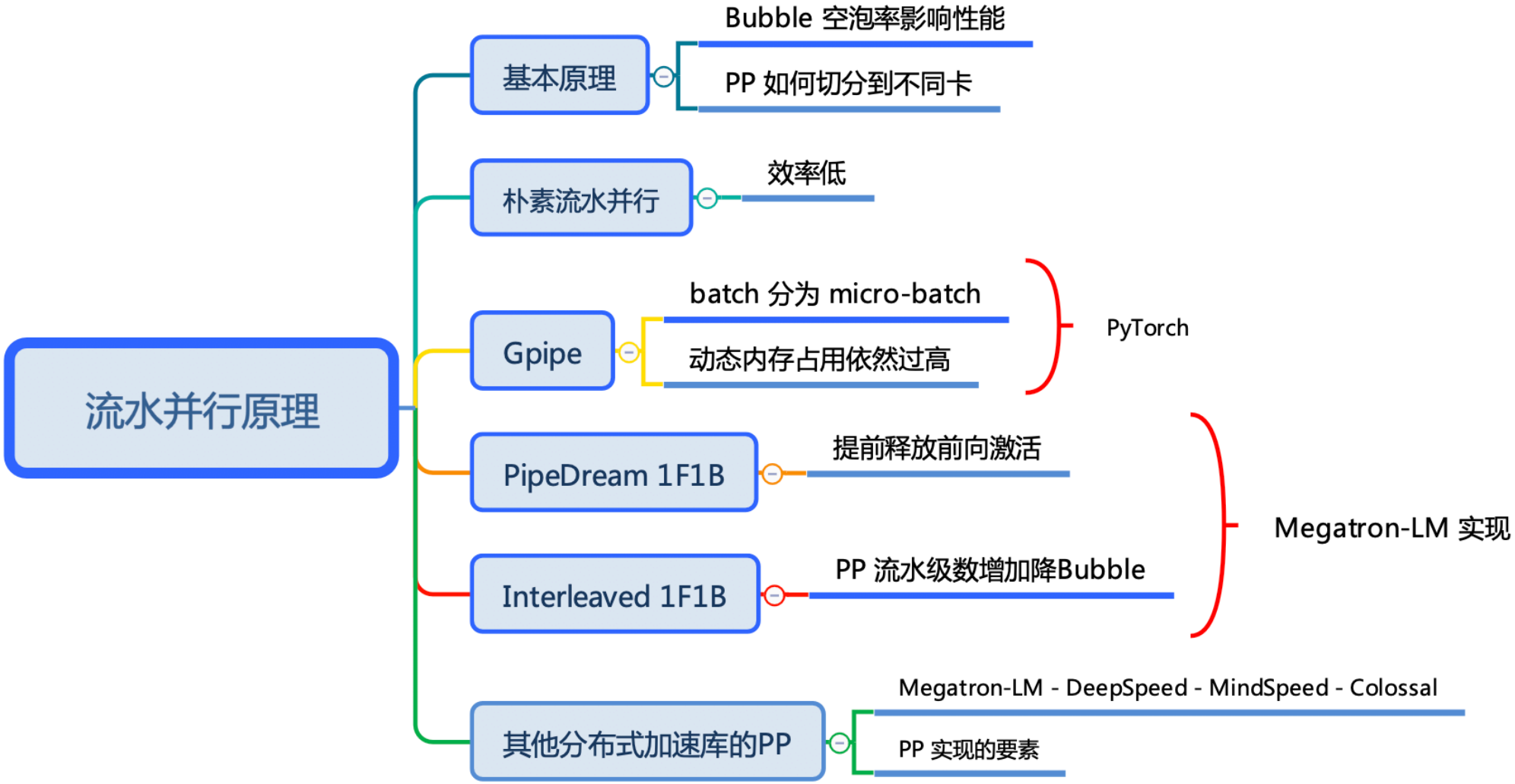


大模型系列 – 分布式训练加速

- **具体内容**

1. **分布式加速库：**
   - 业界常用分布式加速库 & 作用
2. **DeepSpeed 特性：**
   - 基本概念 - 整体框架 – Zero-1/2/3 – ZeRO-Offload – ZeRO-Infinity
3. **Megatron 特性：**
   1. 总体介绍 – 整体流程 – 并行配置 – DP – TP – PP

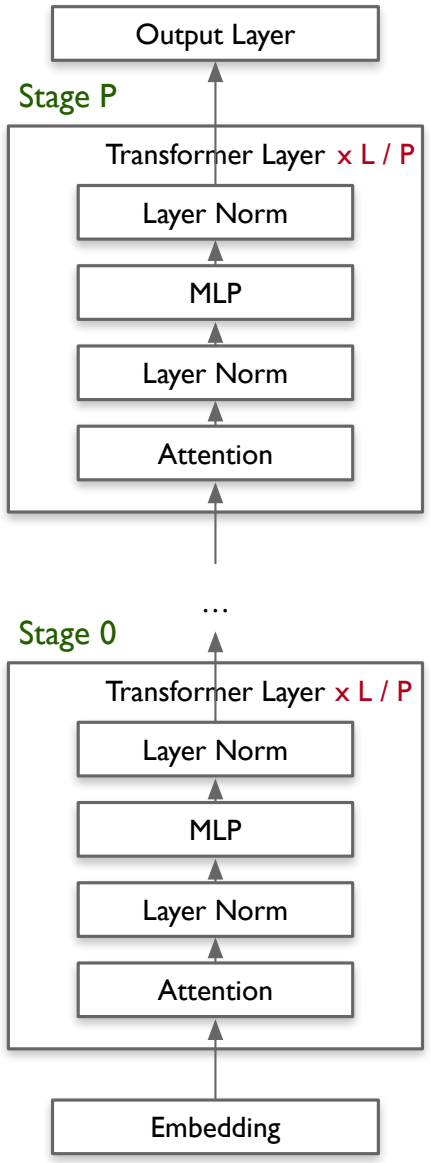
思维导图


流水并行原理
基本原理
Bubble 空泡率影响性能
PP 如何切分到不同卡
朴素流水并行
效率低
Gpipe
batch 分为 micro-batch
动态内存占用依然过高
PyTorch
PipeDream 1F1B
提前释放前向激活
Interleaved 1F1B
PP 流水级数增加降Bubble
Megatron-LM 实现
其他分布式加速库的PP
Megatron-LM - DeepSpeed - MindSpeed - Colossal
PP 实现的要素

Megatron-LM 01

流水并行配置

PP 输入输出




```
CHECKPOINT_PATH=<Specify path>
VOCAB_FILE=<Specify path to file>/gpt2-vocab.json
MERGE_FILE=<Specify path to file>/gpt2-merges.txt
DATA_PATH=<Specify path and file prefix>_text_document

GPT_ARGS="
    --tensor-model-parallel-size 1 \
    --pipeline-model-parallel-size 2 \
    -- sequence_parallel \
    --num-layers 24 \
    --hidden-size 1024 \
    --num-attention-heads 16 \
    --seq-length 1024 \
    --max-position-embeddings 1024 \
    --micro-batch-size 4 \
    --global-batch-size 8 \
    --lr 0.00015 \
    --train-iters 500000 \
    --lr-decay-iters 320000 \
    --lr-decay-style cosine \
    --min-lr 1.0e-5 \
    --weight-decay 1e-2 \
    --lr-warmup-fraction .01 \
    --clip-grad 1.0 \
    --fp16
"
```

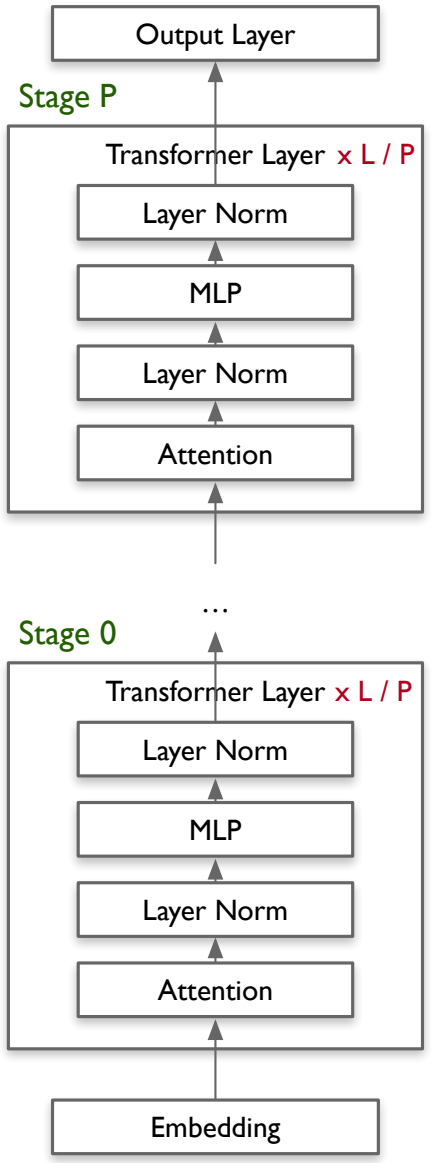
```
if __name__ == "__main__":

    # Temporary for transition to core datasets
    train_valid_test_datasets_provider.is_distributed = True

    pretrain(train_valid_test_datasets_provider,
             model_provider,
             ModelType.encoder_or_decoder,
             forward_step,
             args_defaults={'tokenizer_type': 'GPT2BPETokenizer'})
```

```
def setup_model_and_optimizer(model_provider_func,
                              model_type,
                              no_wd_decay_cond=None,
                              scale_lr_cond=None,
                              lr_mult=1.0):
    """Setup model and optimizer."""
    args = get_args()
    timers = get_timers()

    model = get_model(model_provider_func, model_type)
    unwrapped_model = unwrap_model(model)
```

- pre_train_gpt.py 调用 get_model 判断：
  - pipeline-parallel 按PP个数分为多个 Stage，每个 Stage 运行在 1个 NPU上
  - 第一个 Stage pre_process=True 表示输入为 Embedding
  - 第二个 Stage post_process=True 表示输出为 Model Loss
  - 其余 Stage 不需要 pre_process or post_process

```
def get_model(model_provider_func, model_type=ModelType.encoder_or_decoder, wrap_wit
    # Build model.
    if mpu.get_pipeline_model_parallel_world_size() > 1 and \
       args.virtual_pipeline_model_parallel_size is not None:
        assert model_type != ModelType.encoder_and_decoder, \
            "Interleaved schedule not supported for model with both encoder and decc
        model = []
        for i in range(args.virtual_pipeline_model_parallel_size):
            mpu.set_virtual_pipeline_model_parallel_rank(i)
            # Set pre_process and post_process only after virtual rank is set.
            pre_process = mpu.is_pipeline_first_stage()
            post_process = mpu.is_pipeline_last_stage()
            this_model = model_provider_func(
                pre_process=pre_process,
                post_process=post_process
            )
            this_model.model_type = model_type
            model.append(this_model)
    else:
        pre_process = mpu.is_pipeline_first_stage()
        post_process = mpu.is_pipeline_last_stage()
        add_encoder = True
        add_decoder = True
        if model_type == ModelType.encoder_and_decoder:
            if mpu.get_pipeline_model_parallel_world_size() > 1:
```

PP 模型实例化



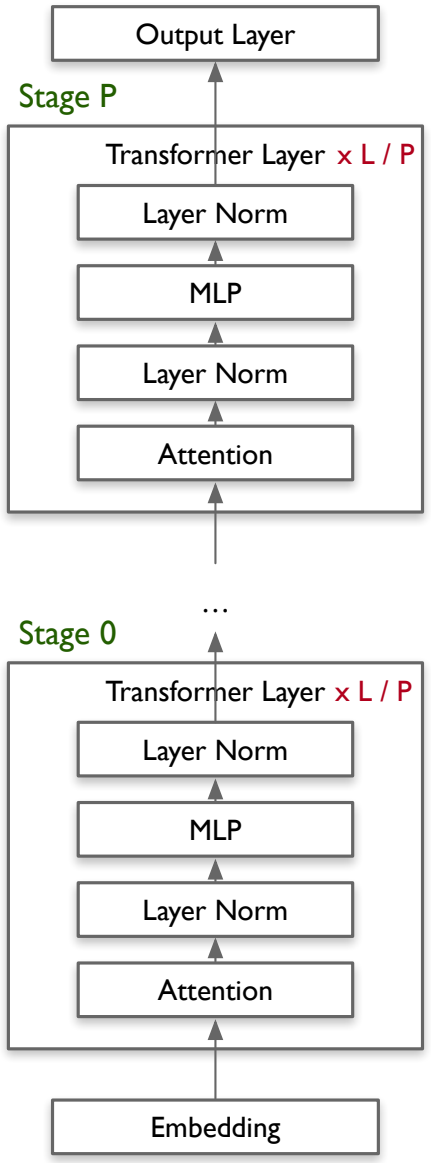

```python
def get_model(model_provider_func, model_type=ModelType.encoder_or_decoder, wrap
    model = []
    for i in range(args.virtual_pipeline_model_parallel_size):
        mpu.set_virtual_pipeline_model_parallel_rank(i)
        # Set pre_process and post_process only after virtual rank is set.
        pre_process = mpu.is_pipeline_first_stage()
        post_process = mpu.is_pipeline_last_stage()
        this_model = model_provider_func(
            pre_process=pre_process,
            post_process=post_process
        )
        this_model.model_type = model_type
        model.append(this_model)
```

```python
class GPTModel(LanguageModule):
    def __init__(

        # Transformer.
        self.decoder = TransformerBlock(
            config=self.config,
            spec=transformer_layer_spec,
            pre_process=self.pre_process,
            post_process=self.post_process,
        )
```

- get_model 构建 GPTModel 实例
- 通过 get_gpt_layer_local_spec() 函数来构建 GPT 模型

```python
# Use this spec for an implementation using only modules in megatron core
def get_gpt_layer_local_spec(
    num_experts: int = None, moe_grouped_gemm: bool = False, qk_layernorm: bool = Fal
) -> ModuleSpec:
    mlp = _get_mlp_module_spec(
        use_te=False, num_experts=num_experts, moe_grouped_gemm=moe_grouped_gemm
    )
    return ModuleSpec(
        module=TransformerLayer,
        submodules=TransformerLayerSubmodules(
            input_layernorm=FusedLayerNorm,
            self_attention=ModuleSpec(
                module=SelfAttention,
                params={"attn_mask_type": AttnMaskType.causal},
                submodules=SelfAttentionSubmodules(
                    linear_qkv=ColumnParallelLinear,
                    core_attention=DotProductAttention,
                    linear_proj=RowParallelLinear,
                    q_layernorm=FusedLayerNorm if qk_layernorm else IdentityOp,
                    k_layernorm=FusedLayerNorm if qk_layernorm else IdentityOp,
                ),
            ),
            self_attn_bda=get_bias_dropout_add,
            pre_mlp_layernorm=FusedLayerNorm,
            mlp=mlp,
            mlp_bda=get_bias_dropout_add,
```

PP 获取需要执行的层数



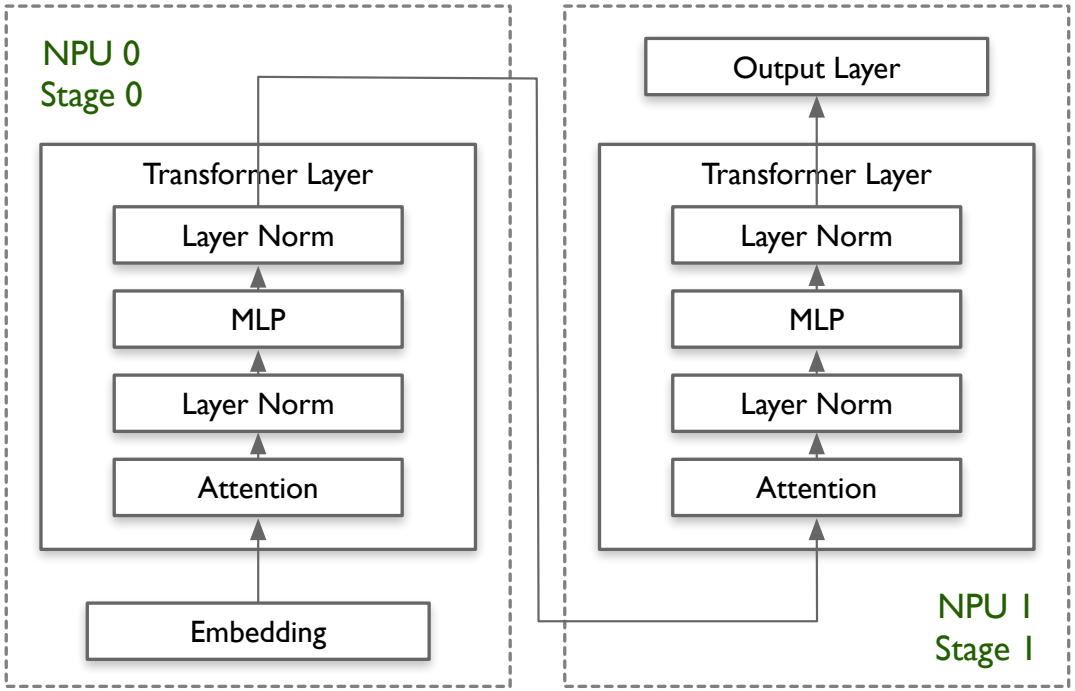

```python
def get_num_layers_to_build(config: TransformerConfig) -> int:

    num_layers_per_pipeline_rank = (
        config.num_layers // parallel_state.get_pipeline_model_parallel_world_size()
    )
```

megatron > core > transformer > transformer_block.py > ...

```python
class TransformerBlockSubmodules:
    layer_specs: List[ModuleSpec] = None


def _get_block_submodules(
    config: TransformerConfig, spec: Union[TransformerBlockSubmodules, ModuleSpe
) -> TransformerBlockSubmodules:

    # Transformer block submodules.
    if isinstance(spec, TransformerBlockSubmodules):
        return spec

    # ModuleSpec here is generally assumed to be for a transformer layer that
    # is implemented in `transformer_layer.py` or if it subclasses
    # `BaseTransformerLayer` from the `transformer_layer.py` file.
    elif isinstance(spec, ModuleSpec):
        if issubclass(spec.module, TransformerBlock):
            return spec.submodules
        elif issubclass(spec.module, BaseTransformerLayer):
            num_layers = get_num_layers_to_build(config)
            return TransformerBlockSubmodules(layer_specs=[spec] * num_layers)
        else:
            raise Exception(f"specialize for {spec.module.__name__}.")
    else:
        raise Exception(f"specialize for {type(spec).__name__}.")
```

```python
class TransformerBlock(MegatronModule):
    def _build_layers(self):
        # @jcasper can we improve how we deal with layer_number?
        # currently it's only used in CoreAttention?
        # if self.apply_query_key_layer_scaling:
        #     coeff = self.layer_number
        #     self.norm_factor *= coeff
        def build_layer(layer_spec, layer_number):
            return build_module(layer_spec, config=self.config, layer_number=la

        # offset is implicit in TransformerLayer
        self.layers = torch.nn.ModuleList(
            [
                build_layer(layer_spec, i + 1)
                for i, layer_spec in enumerate(self.submodules.layer_specs)
            ]
        )
```

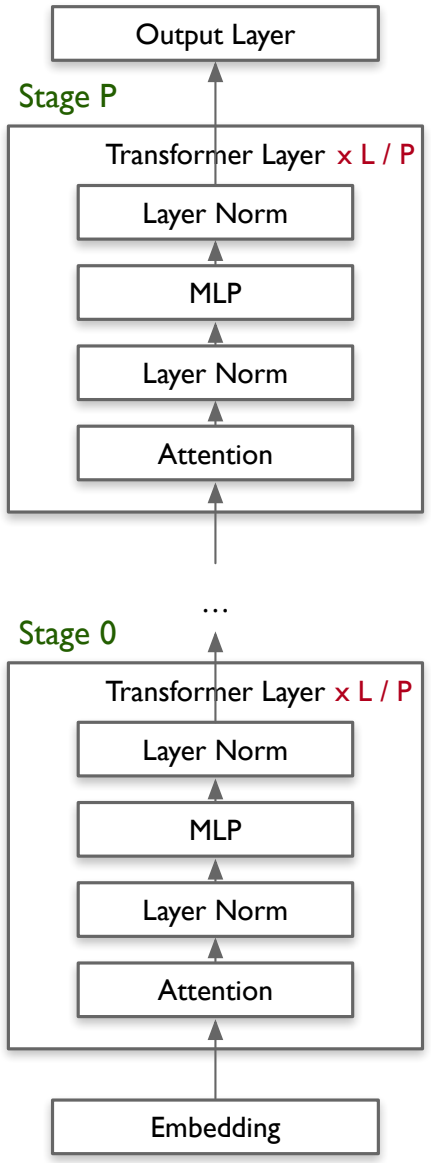
- Transformer block 注册通过 get_num_layers 计算当前 Stage 包含几个 Transformer Layer
- 在 GPT 模型运行示例中每个 Stage build_layer 的个数为 number _lyaer = L / PP_num



Megatron-LM 02

流水并行调用关系

PP Stage 执行



megatron > training > training.py > train

```python
def train(forward_step_func, model, optimizer, opt_param_scheduler,

        args.curr_iteration = iteration
        loss_dict, skipped_iter, grad_norm, num_zeros_in_grad = \
            train_step(forward_step_func,
                       train_data_iterator,
                       model,
                       optimizer,
                       opt_param_scheduler,
                       config)
        iteration += 1
        batch_size = mpu.get_data_parallel_world_size() * \
                     args.micro_batch_size * \
                     get_num_microbatches()
        args.consumed_train_samples += batch_size
        num_floating_point_operations_so_far += num_floating_point_operati
```

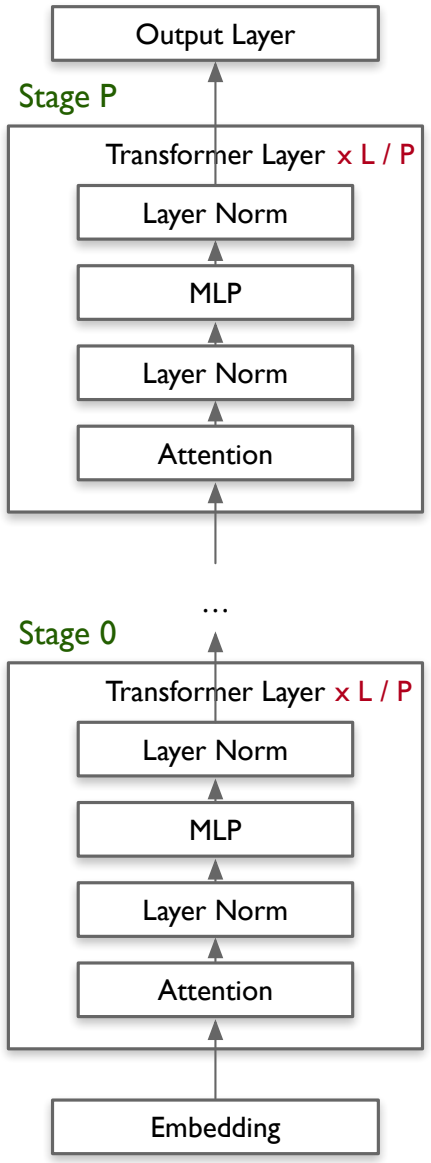
megatron > core > pipeline_parallel > schedules.py > ...

```python
def get_forward_backward_func():
    """Retrieves the appropriate forward_backward function given the…
    pipeline_model_parallel_size = parallel_state.get_pipeline_model_parallel_world_size
    if pipeline_model_parallel_size > 1:
        if parallel_state.get_virtual_pipeline_model_parallel_world_size() is not None:
            forward_backward_func = forward_backward_pipelining_with_interleaving
        else:
            forward_backward_func = forward_backward_pipelining_without_interleaving
    else:
        forward_backward_func = forward_backward_no_pipelining
    return forward_backward_func
```

- 假设 NPU0 执行 Stage0：
  - GPT 调用 pretrain → train → train_step 执行一个 iteration
  - train_step 通过 get_forward_backward_fun 函数进入 schedulers.py 模块
  - 根据当前的 PP 模式返回 forward_backward_pipelining_with_interleaving 用于执行前向和反向计算

```python
def train_step(forward_step_func, data_iterator,
               model, optimizer, opt_param_scheduler, config):
    """Single training step."""
    args = get_args()
    timers = get_timers()

    # Set grad to zero.
    for model_chunk in model:
        model_chunk.zero_grad_buffer()
    optimizer.zero_grad()

    # Forward pass.
    forward_backward_func = get_forward_backward_func()
    losses_reduced = forward_backward_func(
        forward_step_func=forward_step_func,
        data_iterator=data_iterator,
        model=model,
        num_microbatches=get_num_microbatches(),
        seq_length=args.seq_length,
        micro_batch_size=args.micro_batch_size,
        decoder_seq_length=args.decoder_seq_length,
        forward_only=False)
```

NPU 0 执行 Stage 0

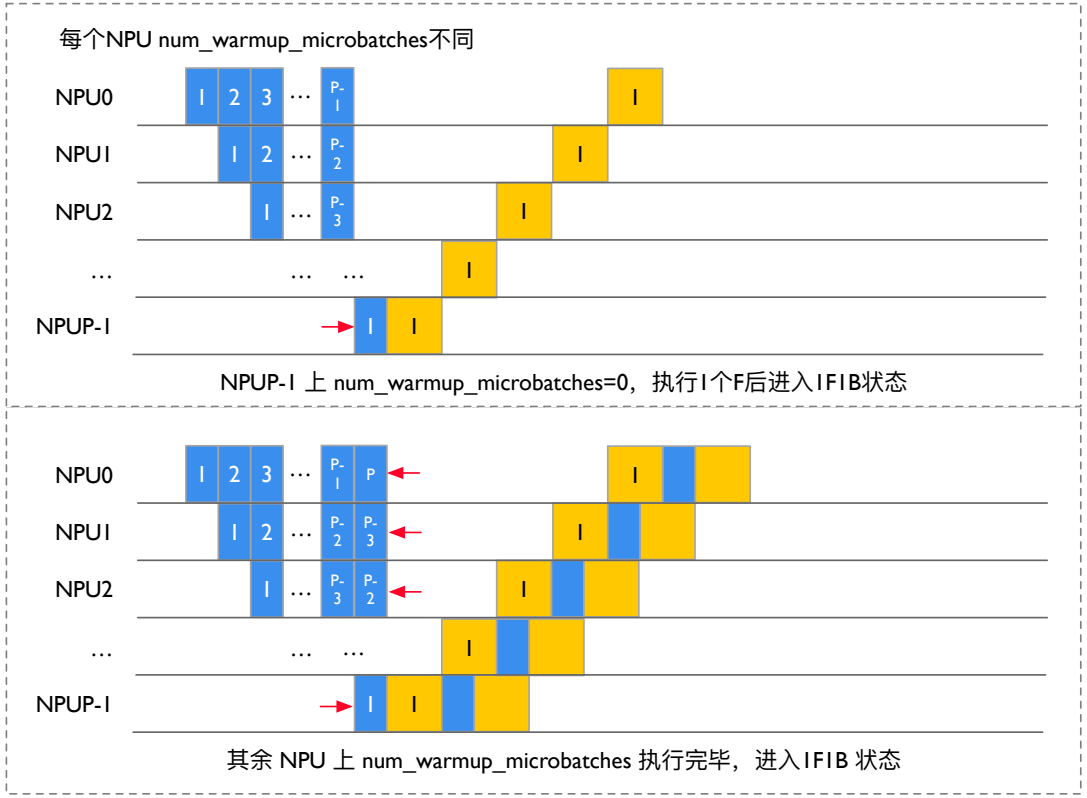



```python
def get_forward_backward_func():
    """Retrieves the appropriate forward_backward function given the…
    pipeline_model_parallel_size = parallel_state.get_pipeline_model_parallel_world_s
    if pipeline_model_parallel_size > 1:
        if parallel_state.get_virtual_pipeline_model_parallel_world_size() is not Non
            forward_backward_func = forward_backward_pipelining_with_interleaving
        else:
            forward_backward_func = forward_backward_pipelining_without_interleaving
    else:
        forward_backward_func = forward_backward_no_pipelining
    return forward_backward_func
```

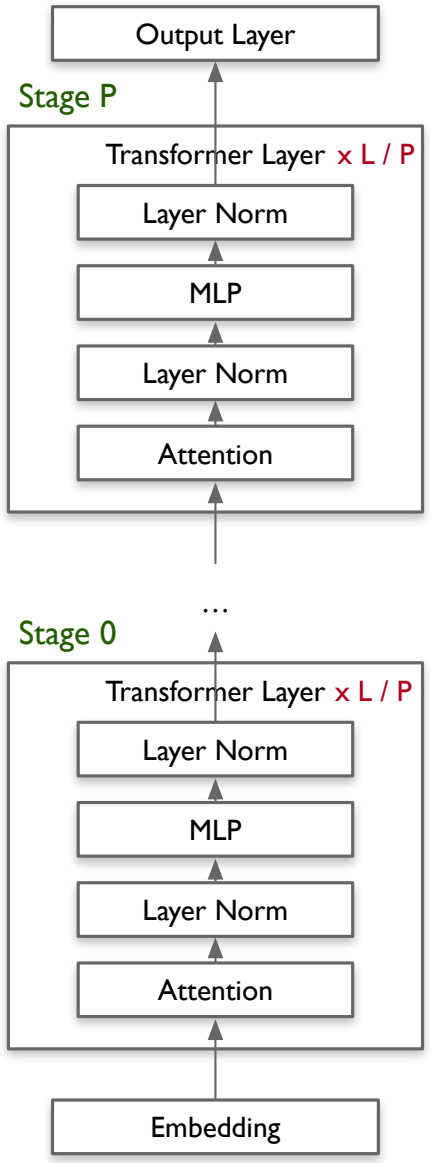
```python
def forward_backward_pipelining_without_interleaving(
    *,
    forward_step_func,
    data_iterator: Union[Iterator, List[Iterator]],
    model: Union[torch.nn.Module, List[torch.nn.Module]],
    num_microbatches: int,
    seq_length: int,
    micro_batch_size: int,
    decoder_seq_length: int = None,
    forward_only: bool = False,
    collect_non_loss_data: bool = False,
    first_val_step: bool = None,
):
    """Run non-interleaved 1F1B schedule, with communication between pipeline
    stages.

    Returns dictionary with losses if the last stage, empty dict otherwise."""
```

```python
def forward_backward_pipelining_without_interleaving(

    disable_grad_sync()

    # Compute number of warmup microbatches.
    num_warmup_microbatches = (
        parallel_state.get_pipeline_model_parallel_world_size()
        - parallel_state.get_pipeline_model_parallel_rank()
        - 1
    )
    num_warmup_microbatches = min(num_warmup_microbatches, num_microbatches)
    num_microbatches_remaining = num_microbatches - num_warmup_microbatches
```

- NPU0 执行 Stage0：
  - 执行 Forward 计算进入 forward_backward_pipelining_without_interleaving (Pipeline 1F1B) 先关闭梯度更新，等所有的 microbatch(MB) 执行完毕才更新梯度



NPU 0 执行 Stage 0

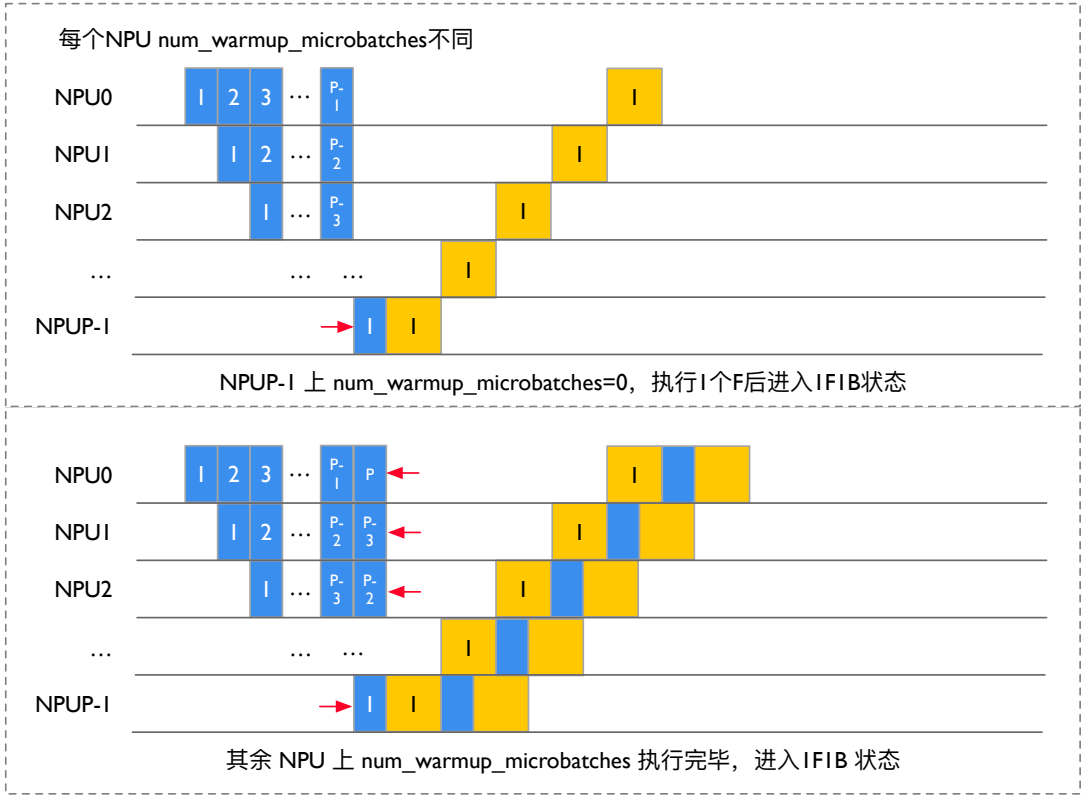



```python
def get_forward_backward_func():
    """Retrieves the appropriate forward_backward function given the…
    pipeline_model_parallel_size = parallel_state.get_pipeline_model_parallel_world_s
    if pipeline_model_parallel_size > 1:
        if parallel_state.get_virtual_pipeline_model_parallel_world_size() is not Non
            forward_backward_func = forward_backward_pipelining_with_interleaving
        else:
            forward_backward_func = forward_backward_pipelining_without_interleaving
    else:
        forward_backward_func = forward_backward_no_pipelining
    return forward_backward_func
```

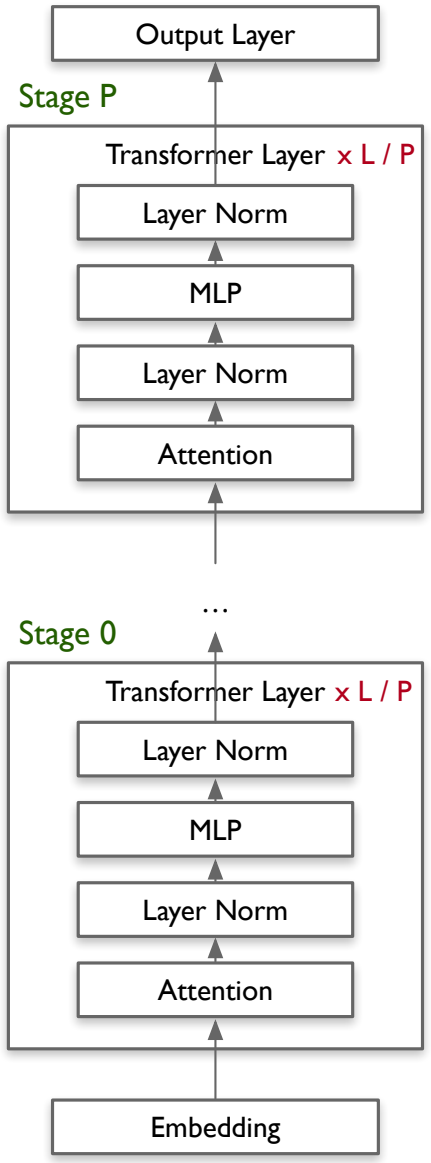
```python
def forward_backward_pipelining_without_interleaving(
    *,
    forward_step_func,
    data_iterator: Union[Iterator, List[Iterator]],
    model: Union[torch.nn.Module, List[torch.nn.Module]],
    num_microbatches: int,
    seq_length: int,
    micro_batch_size: int,
    decoder_seq_length: int = None,
    forward_only: bool = False,
    collect_non_loss_data: bool = False,
    first_val_step: bool = None,
):
    """Run non-interleaved 1F1B schedule, with communication between pipeline
    stages.

    Returns dictionary with losses if the last stage, empty dict otherwise."""
```

```python
def forward_backward_pipelining_without_interleaving(
    disable_grad_sync()

    # Compute number of warmup microbatches.
    num_warmup_microbatches = (
        parallel_state.get_pipeline_model_parallel_world_size()
        - parallel_state.get_pipeline_model_parallel_rank()
        - 1
    )
    num_warmup_microbatches = min(num_warmup_microbatches, num_microbatches)
    num_microbatches_remaining = num_microbatches - num_warmup_microbatches
```

- NPU0 执行 Stage0：
  - num_microbatches：总的 micro batch 个数 = global batch size(6) / micro batch(2) → 3
  - num_warmup_microbatchew：根据 PP 数以及当前 NPU rank 计算需要等到 1F1B 状态的 warmup microbatch 个数，示例 NPU0 = 2(PP) - rank(0) - 1=1, NPU1 = 2(PP) – rank(1) - 1 = 0



NPU 0 执行 Stage 0

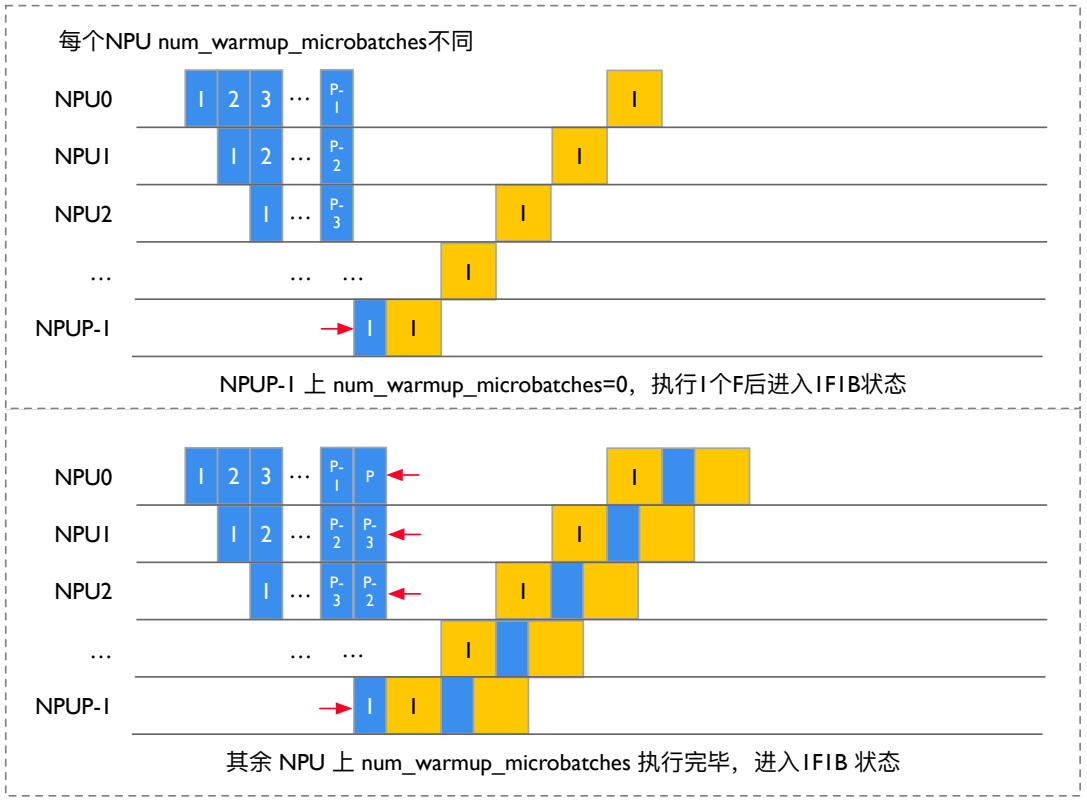



```python
def get_forward_backward_func():
    """Retrieves the appropriate forward_backward function given the…
    pipeline_model_parallel_size = parallel_state.get_pipeline_model_parallel_world_s
    if pipeline_model_parallel_size > 1:
        if parallel_state.get_virtual_pipeline_model_parallel_world_size() is not Non
            forward_backward_func = forward_backward_pipelining_with_interleaving
        else:
            forward_backward_func = forward_backward_pipelining_without_interleaving
    else:
        forward_backward_func = forward_backward_no_pipelining
    return forward_backward_func
```

```python
def forward_backward_pipelining_without_interleaving(
    *,
    forward_step_func,
    data_iterator: Union[Iterator, List[Iterator]],
    model: Union[torch.nn.Module, List[torch.nn.Module]],
    num_microbatches: int,
    seq_length: int,
    micro_batch_size: int,
    decoder_seq_length: int = None,
    forward_only: bool = False,
    collect_non_loss_data: bool = False,
    first_val_step: bool = None,
):
    """Run non-interleaved 1F1B schedule, with communication between pipeline
    stages.

    Returns dictionary with losses if the last stage, empty dict otherwise."""
```

```python
def forward_backward_pipelining_without_interleaving(
    disable_grad_sync()

    # Compute number of warmup microbatches.
    num_warmup_microbatches = (
        parallel_state.get_pipeline_model_parallel_world_size()
        - parallel_state.get_pipeline_model_parallel_rank()
        - 1
    )
    num_warmup_microbatches = min(num_warmup_microbatches, num_microbatches)
    num_microbatches_remaining = num_microbatches - num_warmup_microbatches
```

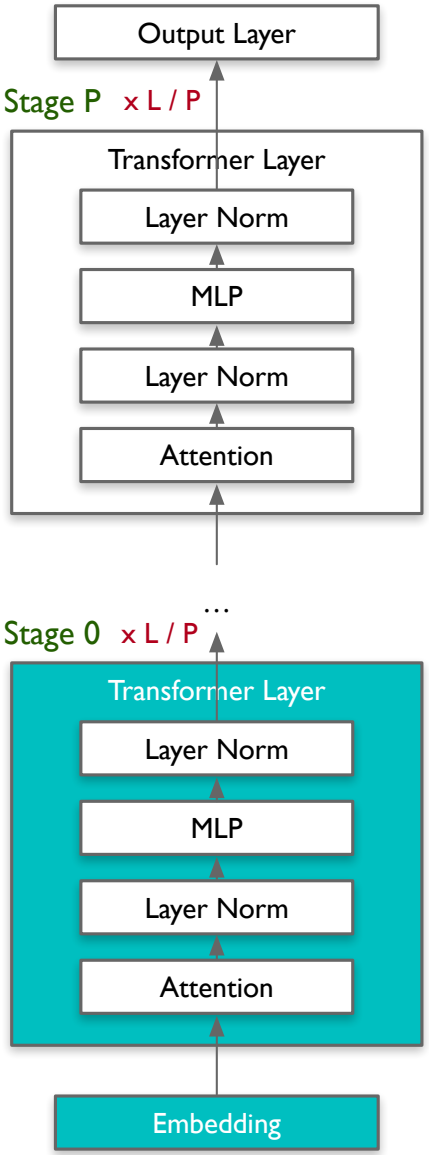
- NPU0 执行 Stage0：
  - num_microbatches_remaining：执行完 warmup 后还有少数 microbatch 需要执行，即 num_microbatches - num_warmup_microbatches，NPU0 = 3 – 1 =2， NPU1 = 3 - 0 = 3



Megatron-LM 03

执行第一个 Stage

NPU 0 执行 Stage 0

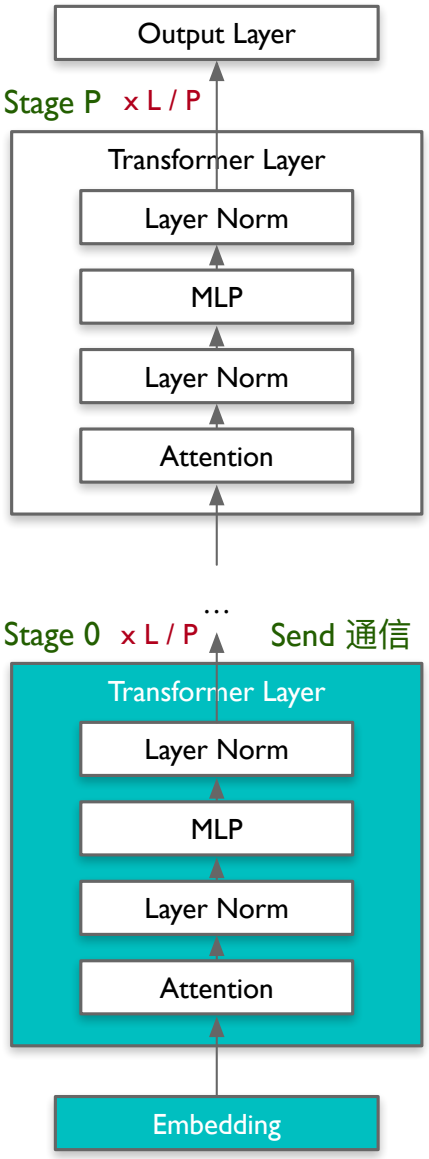



```python
def forward_backward_pipelining_without_interleaving(
    # Run warmup forward passes.
    for i in range(num_warmup_microbatches):
        # Decide to checkpoint all layers' activations of the current micro-batc
        if max_outstanding_backprops is not None:
            checkpoint_activations_microbatch = (
                i % max_outstanding_backprops
                >= config.num_microbatches_with_partial_activation_checkpoints
            )
        else:
            checkpoint_activations_microbatch = None

        input_tensor = recv_forward(recv_tensor_shapes, config)
        output_tensor = forward_step(
            forward_step_func,
            data_iterator,
            model,
            num_microbatches,
            input_tensor,
            forward_data_store,
            config,
            collect_non_loss_data,
            checkpoint_activations_microbatch,
            check_first_val_step(first_val_step, forward_only, i == 0),
        )
        send_forward(output_tensor, send_tensor_shapes, config)
```

```python
def forward_step(
    set_input_tensor = get_attr_wrapped_model(model, "set_input_tensor")
    set_input_tensor(input_tensor)

    if config.enable_autocast:
        context_manager = torch.autocast("cuda", dtype=config.autocast_dtype)
    else:
        context_manager = contextlib.nullcontext()
    with context_manager:
        if checkpoint_activations_microbatch is None:
            output_tensor, loss_func = forward_step_func(data_iterator, model)
        else:
            output_tensor, loss_func = forward_step_func(
                data_iterator, model, checkpoint_activations_microbatch
            )
```

- NPU0 执行 Stage0：
  - NPU0 recv_forward() 在 Stage0 没有其他 Stage 的输入
  - forward_step 调用 forward_step_func 真正调用模型执行

NPU 0 执行 Stage 0



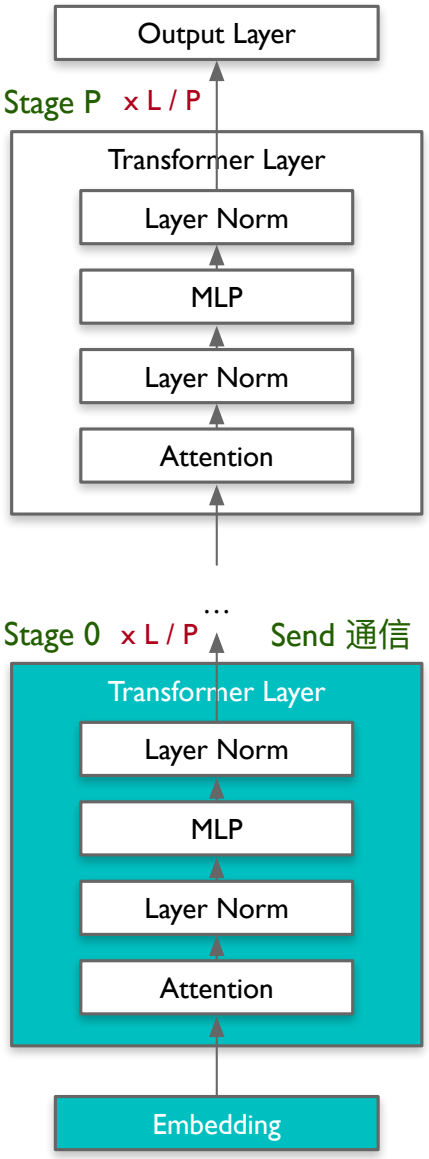

```python
def forward_backward_pipelining_without_interleaving(
    # ...
    input_tensor = recv_forward(recv_tensor_shapes, config)
    output_tensor = forward_step(
        forward_step_func,
        data_iterator,
        model,
        num_microbatches,
        input_tensor,
        forward_data_store,
        config,
        collect_non_loss_data,
        checkpoint_activations_microbatch,
        check_first_val_step(first_val_step, forward_only, i == 0),
    )
    send_forward(output_tensor, send_tensor_shapes, config)
```

```python
def send_forward(output_tensors, tensor_shapes, config):
    if not isinstance(output_tensors, list):
        output_tensors = [output_tensors]
    for (output_tensor, tensor_shape) in zip(output_tensors, tensor_
        if tensor_shape is None:
            continue
        p2p_communication.send_forward(output_tensor, config)
```

```python
def _communicate(
    reqs = p2p_func(
        tensor_send_prev=tensor_send_prev,
        tensor_recv_prev=tensor_recv_prev,
        tensor_send_next=tensor_send_next,
        tensor_recv_next=tensor_recv_next,
        group=get_pipeline_model_parallel_group(),
    )

    if wait_on_reqs and len(reqs) > 0:
        for req in reqs:
            req.wait()
        reqs = None

    if config.batch_p2p_comm and config.batch_p2p_sync:
        # To protect against race condition when using batch_isend_
        # User should assert that we have a modern enough PyTorch
        torch.cuda.synchronize()

    return tensor_recv_prev, tensor_recv_next, reqs
```

- NPU0 执行 Stage0：
  - NPU0 上输出 Stage0 output_tensor 后 send_forward 发送给下一个 Stage
  - 通过 P2P_communication.send_forward 发送 output_tensor，通过 torch.distributed.P2POp 异步 send output_tensor
  - 最后调用 torch.cuda.synchronize() 执行同步

NPU 0 执行 Stage 0




```python
def forward_backward_pipelining_with_interleaving(
    # Run 1F1B in steady state.
    for k in range(num_microbatches_remaining):
        # Forward pass.
        forward_k = k + num_warmup_microbatches

        output_tensor_grad, bwd_wait_handles = \
            p2p_communication.send_backward_recv_backward(
            input_tensor_grad,
            recv_next=recv_next,
            tensor_shape=tensor_shape,
            config=config,
            overlap_p2p_comm=True,
        )
```

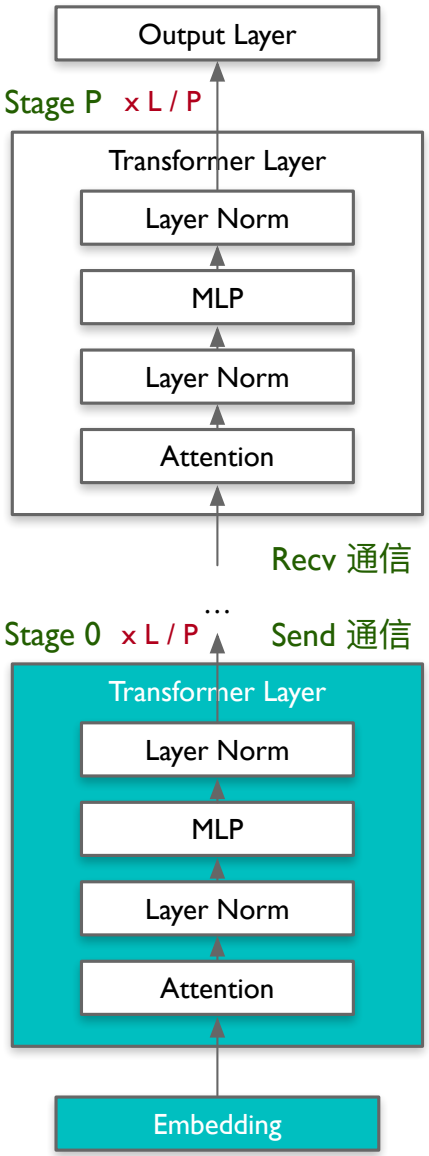
```python
def _communicate(
    else:
        p2p_func = _p2p_ops

    reqs = p2p_func(
        tensor_send_prev=tensor_send_prev,
        tensor_recv_prev=tensor_recv_prev,
        tensor_send_next=tensor_send_next,
        tensor_recv_next=tensor_recv_next,
        group=get_pipeline_model_parallel_group(),
    )

    if wait_on_reqs and len(reqs) > 0:
        for req in reqs:
            req.wait()
        reqs = None

    if config.batch_p2p_comm and config.batch_p2p_sync:
        # To protect against race condition when using batch_isend_irecv(
        # User should assert that we have a modern enough PyTorch to not
        torch.cuda.synchronize()

    return tensor_recv_prev, tensor_recv_next, reqs
```

- NPU0 执行 Stage0：
  - 继续执行 forward_step 执行 Stage0 的前向计算得到第二个 output_tenso r (1F1B 状态)
  - 执行 sedn_forward_recv_backward 发送output_tensor 等待 backward，进入1F1B状态
  - 通过 send_backward_recv_backward 底层试下通过 P2PPp异步 send outpu t_tesnor， 并且异步 recv tensor_recv_next
  - 最后调用 synchronize() 等待 recv backward，NPU0 进入等待状态

Megatron-LM 04

执行第二个 Stage

NPU1 执行 Stage1



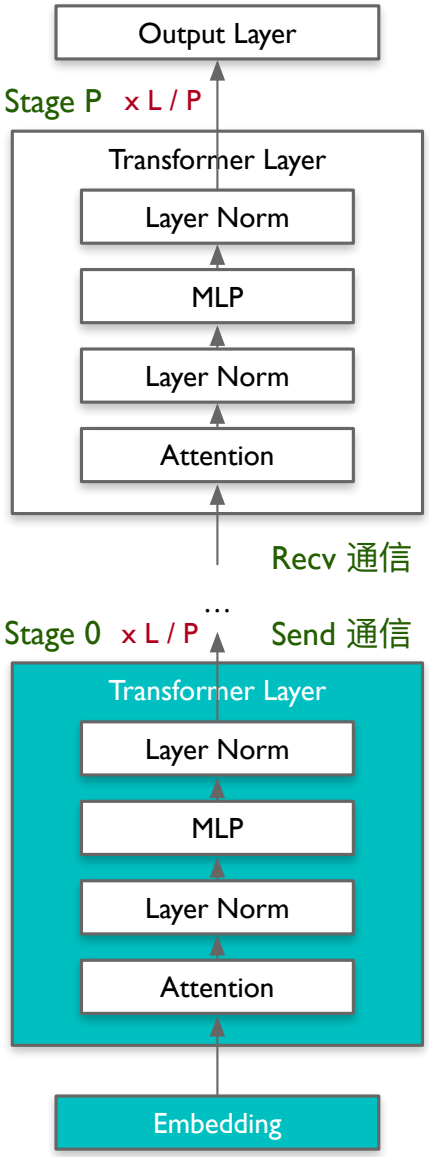

```python
def forward_backward_pipelining_without_interleaving(
        input_tensor = recv_forward(recv_tensor_shapes, config)
        output_tensor = forward_step(
            forward_step_func,
            data_iterator,
            model,
            num_microbatches,
            input_tensor,
            forward_data_store,
            config,
            collect_non_loss_data,
            checkpoint_activations_microbatch,
            check_first_val_step(first_val_step, forward_only, i == 0),
        )
        send_forward(output_tensor, send_tensor_shapes, config)
```

```python
def recv_forward(tensor_shapes, config):
    input_tensors = []
    for tensor_shape in tensor_shapes:
        if tensor_shape is None:
            input_tensors.append(None)
        else:
            input_tensors.append(p2p_communication.recv_forward(tensor_shape, config))
    return input_tensors
```

```python
    reqs = p2p_func(
        tensor_send_prev=tensor_send_prev,
        tensor_recv_prev=tensor_recv_prev,
        tensor_send_next=tensor_send_next,
        tensor_recv_next=tensor_recv_next,
        group=get_pipeline_model_parallel_group(),
    )

    if wait_on_reqs and len(reqs) > 0:
        for req in reqs:
            req.wait()
        reqs = None

    if config.batch_p2p_comm and config.batch_p2p_sync:
        # To protect against race condition when using batch_is
        # User should assert that we have a modern enough PyTor
        torch.cuda.synchronize()

    return tensor_recv_prev, tensor_recv_next, reqs
```

- NPU1 执行 Stage1：
  - NPU1 进入 forward_backward_pipeline_without_interleaving
  - NPU1 num_warmup_microbatches=0，进入1F1B状态
  - NPU1 num_microbatches_remaining=3，recv_forward 调用 P2POp 异步recv
  - NPU1 最后调用synchronize() 执行同步等待 NPU0 Stage0 输出
  - 从而保证NPU0 to NPU1 的执行序

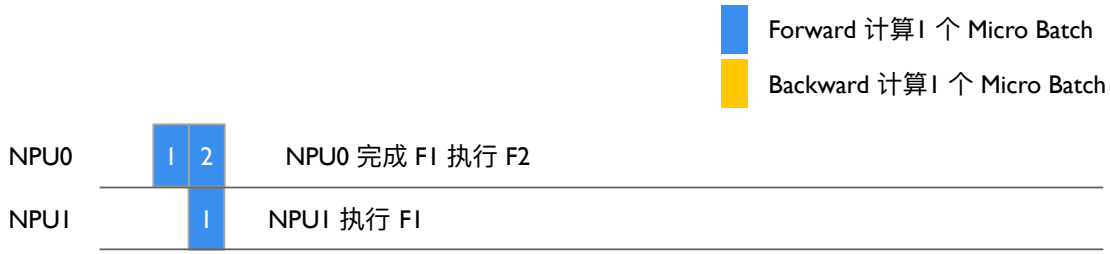
NPU1 执行 Stage1



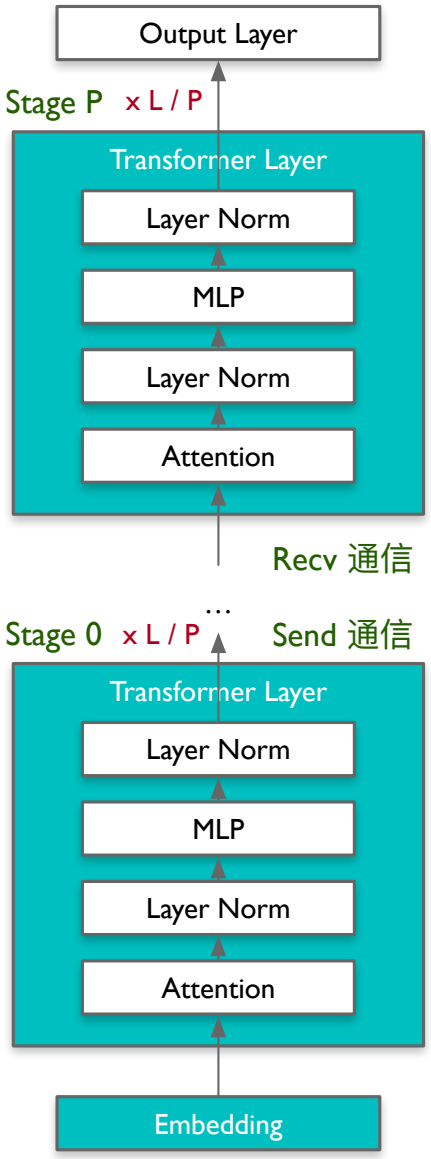

```python
def forward_backward_pipelining_without_interleaving(

        input_tensor = recv_forward(recv_tensor_shapes, config)
        output_tensor = forward_step(
            forward_step_func,
            data_iterator,
            model,
            num_microbatches,
            input_tensor,
            forward_data_store,
            config,
            collect_non_loss_data,
            checkpoint_activations_microbatch,
            check_first_val_step(first_val_step, forward_only, i == 0),
        )
        send_forward(output_tensor, send_tensor_shapes, config)
```

```python
def forward_step(
    set_input_tensor = get_attr_wrapped_model(model, "set_input_tensor")
    set_input_tensor(input_tensor)

    if config.enable_autocast:
        context_manager = torch.autocast("cuda", dtype=config.autocast_dtype)
    else:
        context_manager = contextlib.nullcontext()
    with context_manager:
        if checkpoint_activations_microbatch is None:
            output_tensor, loss_func = forward_step_func(data_iterator, model)
        else:
            output_tensor, loss_func = forward_step_func(
                data_iterator, model, checkpoint_activations_microbatch
            )
```

- NPU1 执行 Stage1：
  - NPU1 recv_forward 等待 NPU0 Stage0 发送 intput_tensor 后
  - NPU1 forward_step 设置 iNPUt_tensor，实现 NPU0&NPU1 交换输入输出
  - NPU1 进入 1F1B 循环，forward_step_func 调用 GPTModel 执行前向计算



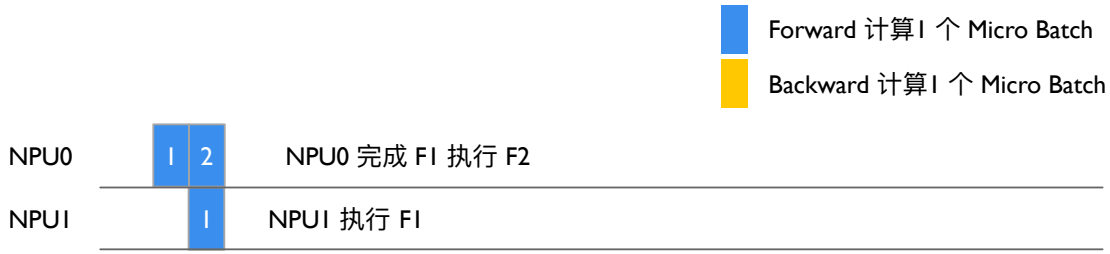
NPU1 执行 Stage1



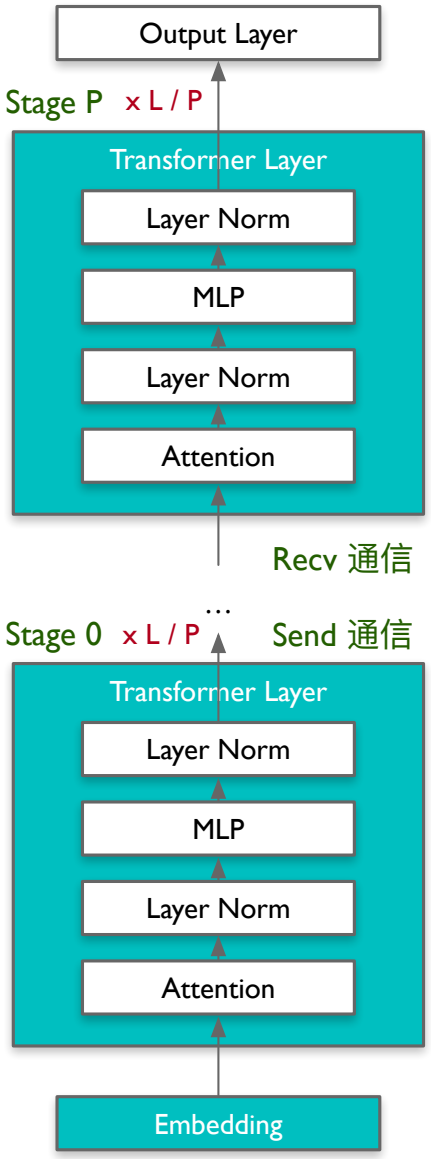

```python
class TransformerBlock(MegatronModule):
    """Transformer class."""

    def __init__(
        self,
        config: TransformerConfig,
        spec: Union[TransformerBlockSubmodules, ModuleSpec],
        post_layer_norm: bool = True,
        pre_process: bool = True,
        post_process: bool = True,
    ):
        super().__init__(config=config)

        self.submodules = _get_block_submodules(config, spec)
        self.post_layer_norm = post_layer_norm
        self.pre_process = pre_process
        self.post_process = post_process
```

```
ron > core > transformer > transformer_block.py > TransformerBlock >
class TransformerBlock(MegatronModule):
    def forward(
        self,
        hidden_states: Tensor,
        attention_mask: Tensor,
        context: Tensor = None,
        context_mask: Tensor = None,
        rotary_pos_emb: Tensor = None,
        inference_params: InferenceParams = None,
        packed_seq_params: PackedSeqParams = None,
    ):
        # hidden_states (float): [s, b, h]
        # attention_mask (bool): [1, 1, s, s]

        if not self.pre_process:
            # See set_input_tensor()
            hidden_states = self.input_tensor

        # Viewless tensor
```

- NPU1 执行 Stage1：
  - NPU1 上 TransformerBlock 执行第一个 Stage，Pre_process=False
  - 即不会把 iNPUt_embeddings 作为 Transformer 的输入
  - 使用 NPU0 Stage0 输入的 iNPUt_tensor 作为输入执行得到 output tensor



NPU1 执行 Stage1

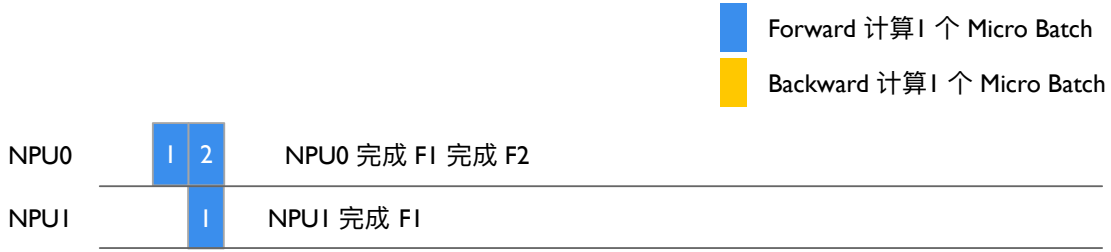



```
n > core > pipeline_parallel > schedules.py > forward_step
def forward_step(

    if parallel_state.is_pipeline_last_stage():
        if not collect_non_loss_data:
            output_tensor = loss_func(output_tensor)
            loss, loss_reduced = output_tensor
            output_tensor = loss / num_microbatches
            forward_data_store.append(loss_reduced)
        else:
            data = loss_func(output_tensor, non_loss_data=True)
            forward_data_store.append(data)
```

```
class TransformerBlock(MegatronModule):
    """Transformer class."""

    def __init__(
        self,
        config: TransformerConfig,
        spec: Union[TransformerBlockSubmodules, ModuleSpec],
        post_layer_norm: bool = True,
        pre_process: bool = True,
        post_process: bool = True,
    ):
        super().__init__(config=config)
```

```
class TransformerBlock(MegatronModule):
    def forward(
                    if (
                        torch.is_grad_enabled()
                        and self.config.cpu_offloading
                        and self.group_prefetch_offload_commit_async
                    ):
                        hidden_states = self.group_prefetch_offload_co

        # Final layer norm.
        if self.post_process and self.post_layer_norm:
            hidden_states = self.final_layernorm(hidden_states)

        return hidden_states
```

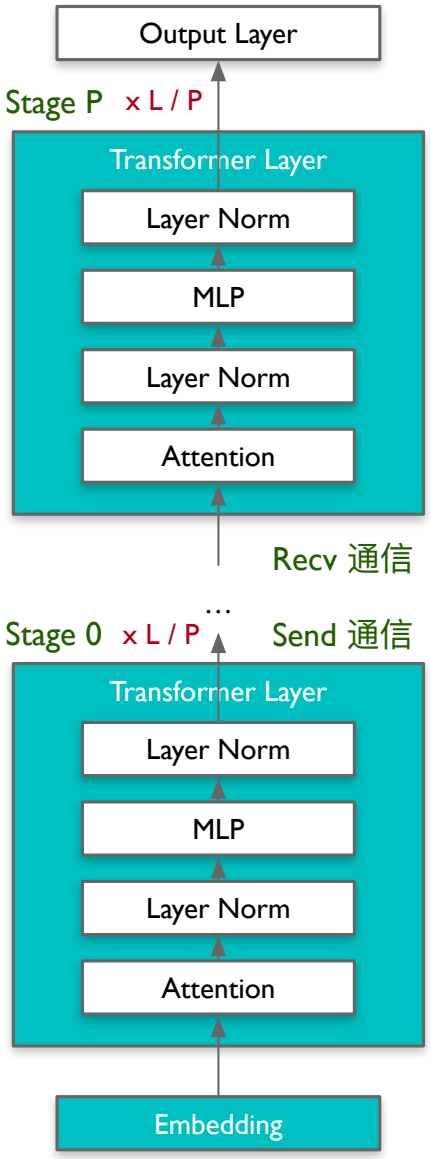
- NPU1 执行 Stage1：
  - 示例中 NPU1 Stage1 是最后一层 Staege，因此 post_process=True
  - 执行 is_pipeline_last_stage 计算 GPT 模型的output_tensor 和 loss



Megatron-LM 05

反向执行 Stage

NPU1 执行 Stage1



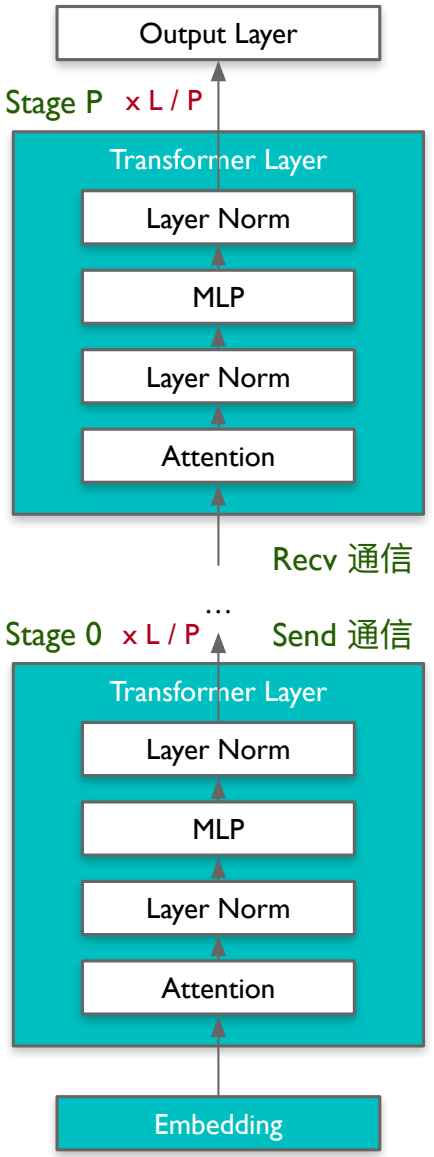

```
core > pipeline_parallel > schedules.py > forward_backward_pipelining_without_interleavin
def forward_backward_pipelining_without_interleaving(

            # Enable grad sync for the last microbatch in the batch if the full
            # backward pass completes in the 1F1B stage.
            if num_warmup_microbatches == 0 and last_iteration:
                if config.grad_sync_func is None or rank == 0:
                    enable_grad_sync()

            input_tensor_grad = backward_step(
                input_tensor, output_tensor, output_tensor_grad, model_type, config
            )

            if last_iteration:
                input_tensor = None
                send_backward(input_tensor_grad, recv_tensor_shapes, config)
            else:
                input_tensor = send_backward_recv_forward(
                    input_tensor_grad, recv_tensor_shapes, config
                )
```

```
def send_backward_recv_forward(input_tensor_grads, tensor_shapes, config):
    if not isinstance(input_tensor_grads, list):
        input_tensor_grads = [input_tensor_grads]
    input_tensors = []
    for (input_tensor_grad, tensor_shape) in zip(input_tensor_grads, tensor_shapes):
        if tensor_shape is None:
            input_tensors.append(None)
            continue
        input_tensor = p2p_communication.send_backward_recv_forward(
            input_tensor_grad, tensor_shape, config
        )
        input_tensors.append(input_tensor)
    return input_tensors
```

```
def send_backward_recv_forward(
    input_tensor_grad: torch.Tensor, tensor_shape: Shape, config: ModelParallelConf
) -> torch.Tensor:
    """Batched send and recv with previous rank in pipeline.

    See _communicate for argument details.
    """
    if core.parallel_state.is_pipeline_first_stage():
        input_tensor = None
    else:
        if config.timers is not None:
            config.timers('backward-send-forward-recv', log_level=2).start()
        input_tensor, _, _ = _communicate(
            tensor_send_next=None,
            tensor_send_prev=input_tensor_grad,
            recv_prev=True,
            recv_next=False,
            tensor_shape=tensor_shape,
            config=config,
        )
```

- NPU1 执行 Stage1：
  - 执行完 forward_step 后执行 backward_step 得到 iNPUt_tensor_grad，并进入 1F1B 状态
  - 执行 send_backward_recc_forward → _communication → 异步发送 iNPUt_tensor_grad 给 NPU0 并等待 NPU0 发送下一个 MB forward 结果

NPU0 执行 Stage0

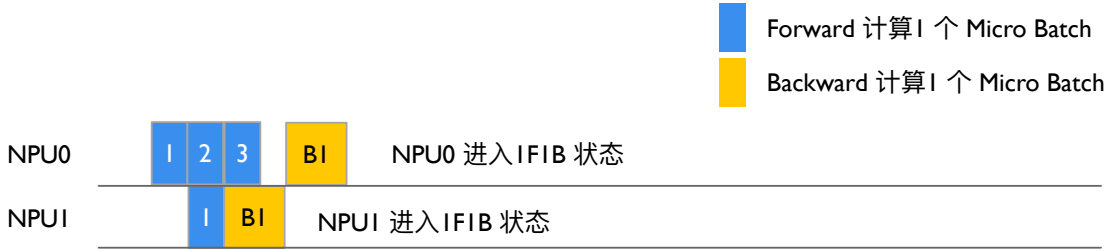



```
core > pipeline_parallel > schedules.py > forward_backward_pipelining_without_interleavin
def forward_backward_pipelining_without_interleaving(

            # Enable grad sync for the last microbatch in the batch if the full
            # backward pass completes in the 1F1B stage.
            if num_warmup_microbatches == 0 and last_iteration:
                if config.grad_sync_func is None or rank == 0:
                    enable_grad_sync()

            input_tensor_grad = backward_step(
                input_tensor, output_tensor, output_tensor_grad, model_type, config
            )

            if last_iteration:
                input_tensor = None
                send_backward(input_tensor_grad, recv_tensor_shapes, config)
            else:
                input_tensor = send_backward_recv_forward(
                    input_tensor_grad, recv_tensor_shapes, config
                )
```

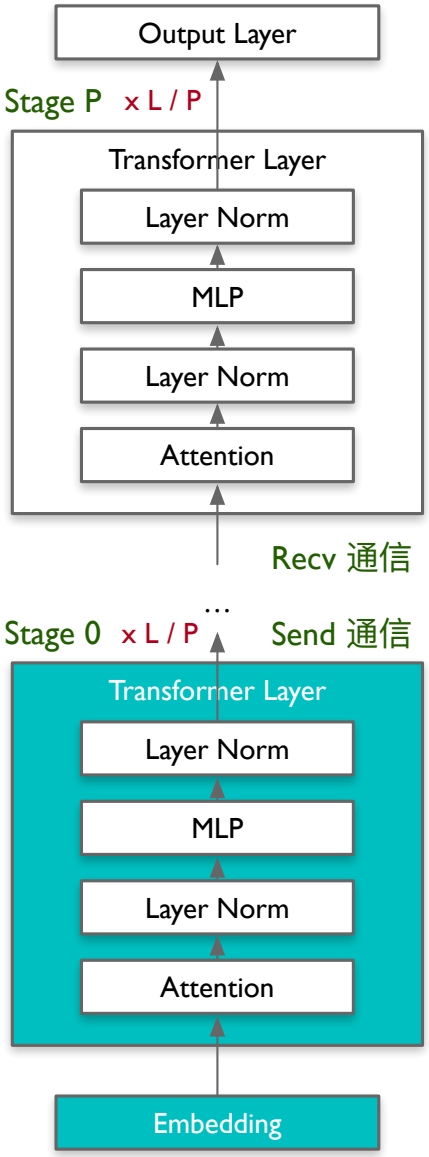
```
def send_backward_recv_forward(input_tensor_grads, tensor_shapes, config):
    if not isinstance(input_tensor_grads, list):
        input_tensor_grads = [input_tensor_grads]
    input_tensors = []
    for (input_tensor_grad, tensor_shape) in zip(input_tensor_grads, tensor_shapes):
        if tensor_shape is None:
            input_tensors.append(None)
            continue
        input_tensor = p2p_communication.send_backward_recv_forward(
            input_tensor_grad, tensor_shape, config
        )
        input_tensors.append(input_tensor)
    return input_tensors
```

```
def send_backward_recv_forward(
    input_tensor_grad: torch.Tensor, tensor_shape: Shape, config: ModelParallelConf
) -> torch.Tensor:
    """Batched send and recv with previous rank in pipeline.

    See _communicate for argument details.
    """
    if core.parallel_state.is_pipeline_first_stage():
        input_tensor = None
    else:
        if config.timers is not None:
            config.timers('backward-send-forward-recv', log_level=2).start()
        input_tensor, _, _ = _communicate(
            tensor_send_next=None,
            tensor_send_prev=input_tensor_grad,
            recv_prev=True,
            recv_next=False,
            tensor_shape=tensor_shape,
            config=config,
        )
```

- NPU0 执行 Stage0：
  - NPU0 Srage0 等待 send_backward_recv_forward 被唤醒后获得 NPU1 Staege1 发送的 output_tensor_grad
  - NPU0 Stage0 执行 backward_step 输出 intput_tensor_grad，NPU0 计入 1F1B 状态



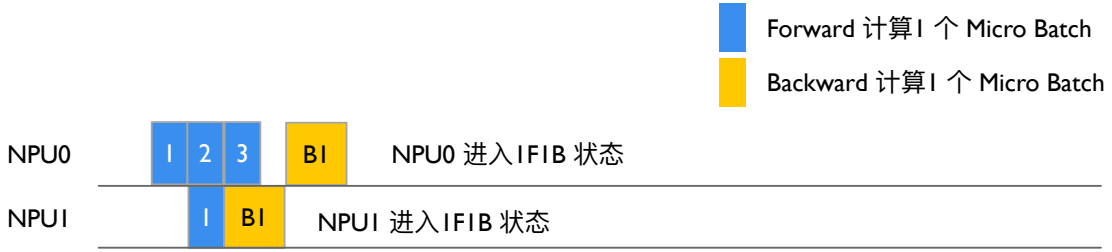
NPU0 执行 Stage0



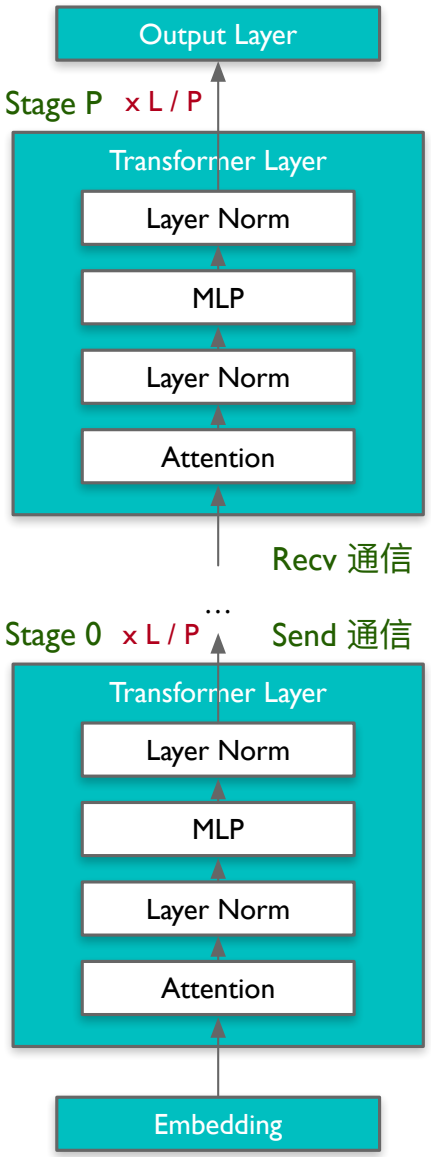

```
on > core > pipeline_parallel > schedules.py > forward_backward_pipelinir
def forward_backward_pipelining_without_interleaving(
    # Run 1F1B in steady state.
    for i in range(num_microbatches_remaining):
        last_iteration = i == (num_microbatches_remaining - 1)

        # Decide to checkpoint all layers' activations of the curr
        if max_outstanding_backprops is not None: ...
        else:
            checkpoint_activations_microbatch = None

        output_tensor = forward_step( ...

        if forward_only: ...

        else:
            output_tensor_grad = send_forward_recv_backward(
                output_tensor, send_tensor_shapes, config
            )

            # Add input_tensor and output_tensor to end of list.
            input_tensors.append(input_tensor)
            output_tensors.append(output_tensor)
            deallocate_output_tensor(output_tensor[0], config.deal
```

```
on > core > pipeline_parallel > p2p_communication.py > _communicate
def _communicate(
    if recv_prev:
        if config.pipeline_dtype is None:
            raise RuntimeError("pipeline_dtype must be provided
        if tensor_shape is None:
            raise RuntimeError(
                "tensor_shape must be specified if recv_prev is
                "Common tensor_shape is (seq_length, micro_batch_
            )
        tensor_recv_prev = torch.empty( ...
    if recv_next:
        if config.pipeline_dtype is None:
            raise RuntimeError("dtype must be provided if recv_ne
        if tensor_shape is None:
            raise RuntimeError(
                "tensor_shape must be specified if recv_next is
                "Common tensor_shape is (seq_length, micro_batch_
            )
        tensor_recv_next = torch.empty( ...
```

- NPU0 执行 Stage0：
  - NPU0 num_warmup_mbs=1, num_mbs_remaining=2，进入 1F1B 循环
  - 执行 forward_step 执行 Stargel 前向计算得到 output_tensor (Forward 3)
  - 执行 send_forward_recv_backward 发送 output_tensor 等待 backward
  - 异步 recv tensor_recv_next，调用 synnchronize() 同步等待 backward，NPU0 进入等待状态



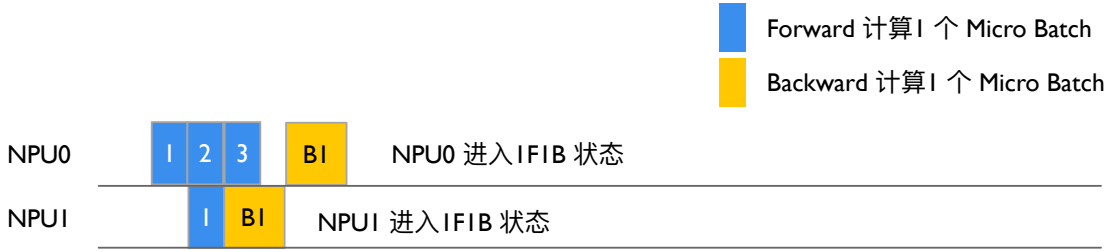
NPU1 执行 Stage1



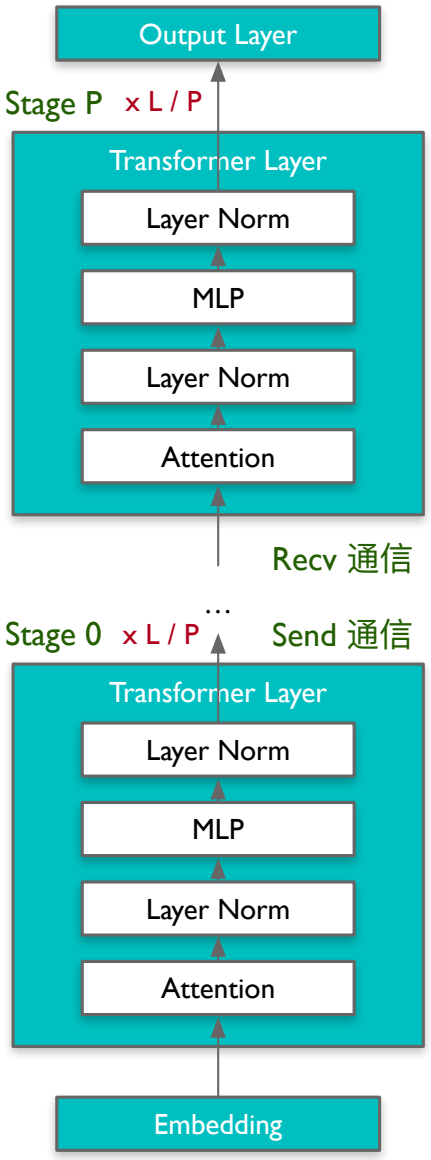

```
on > core > pipeline_parallel > schedules.py > forward_backward_pipelinir
def forward_backward_pipelining_without_interleaving(
    # Run 1F1B in steady state.
    for i in range(num_microbatches_remaining):
        last_iteration = i == (num_microbatches_remaining - 1)

        # Decide to checkpoint all layers' activations of the curr
        if max_outstanding_backprops is not None:
        else:
            checkpoint_activations_microbatch = None

        output_tensor = forward_step(

        if forward_only:

        else:
            output_tensor_grad = send_forward_recv_backward(
                output_tensor, send_tensor_shapes, config
            )

            # Add input_tensor and output_tensor to end of list.
            input_tensors.append(input_tensor)
            output_tensors.append(output_tensor)
            deallocate_output_tensor(output_tensor[0], config.deal
```

```
on > core > pipeline_parallel > p2p_communication.py > _communicate
def _communicate(
    if recv_prev:
        if config.pipeline_dtype is None:
            raise RuntimeError("pipeline_dtype must be provided
        if tensor_shape is None:
            raise RuntimeError(
                "tensor_shape must be specified if recv_prev is
                "Common tensor_shape is (seq_length, micro_batch_
            )
        tensor_recv_prev = torch.empty(
    if recv_next:
        if config.pipeline_dtype is None:
            raise RuntimeError("dtype must be provided if recv_ne
        if tensor_shape is None:
            raise RuntimeError(
                "tensor_shape must be specified if recv_next is
                "Common tensor_shape is (seq_length, micro_batch_
            )
        tensor_recv_next = torch.empty(
```

- NPU1 执行 Stage1：
  - NPU1 Stage1 上执行 send_backward_recv_forward 同步等待收到 NPU0 Stge0 发送iNPUt_tensor（Forward 2）
  - NPU1 Stage1 将 iNPUt_tensor （Forward 2）作为 TransformerBlock 执行 forward_step 得到输出output_tensor

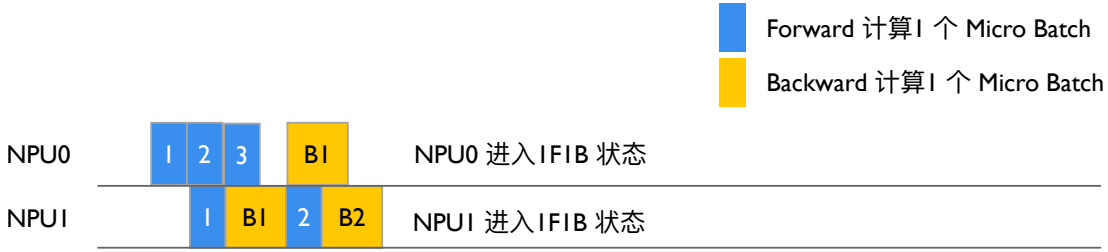


NPU1 执行 Stage1

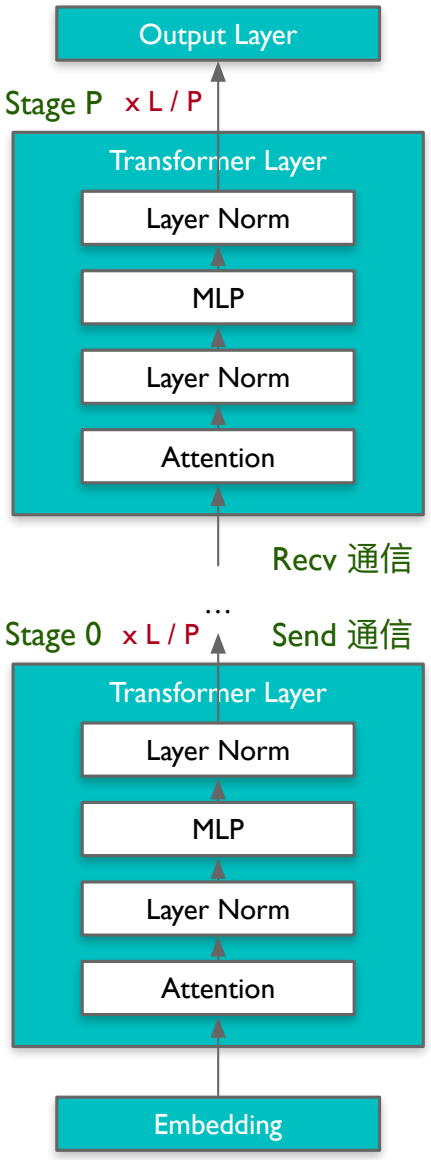



```
core > pipeline_parallel > schedules.py > forward_backward_pipelining_withou
def forward_backward_pipelining_without_interleaving(
    # Run 1F1B in steady state.
    for i in range(num_microbatches_remaining):
        last_iteration = i == (num_microbatches_remaining - 1)

        # Decide to checkpoint all layers' activations of the current mic
        if max_outstanding_backprops is not None:...
        else:
            checkpoint_activations_microbatch = None

        output_tensor = forward_step(...

        if forward_only:
            send_forward(output_tensor, send_tensor_shapes, config)

            if not last_iteration:
                input_tensor = recv_forward(recv_tensor_shapes, config)

        else:
            output_tensor_grad = send_forward_recv_backward(
                output_tensor, send_tensor_shapes, config
            )

            # Add input_tensor and output_tensor to end of list.
            input_tensors.append(input_tensor)
            output_tensors.append(output_tensor)
            # Add input_tensor and output_tensor to end of list.
            input_tensors.append(input_tensor)
            output_tensors.append(output_tensor)
            deallocate_output_tensor(output_tensor[0], config.deallocate_pip

            # Pop input_tensor and output_tensor from the start of the list
            # the backward pass.
            input_tensor = input_tensors.pop(0)
            output_tensor = output_tensors.pop(0)

            # Enable grad sync for the last microbatch in the batch if the f
            # backward pass completes in the 1F1B stage.
            if num_warmup_microbatches == 0 and last_iteration:
                if config.grad_sync_func is None or rank == 0:
                    enable_grad_sync()

            input_tensor_grad = backward_step(
                input_tensor, output_tensor, output_tensor_grad, model_type,
            )

            if last_iteration:
                input_tensor = None
                send_backward(input_tensor_grad, recv_tensor_shapes, config)
            else:
                input_tensor = send_backward_recv_forward(
                    input_tensor_grad, recv_tensor_shapes, config
```

- NPU1 执行 Stage1：
  - NPU1 num_warmup_mbs=0, num_mbs_remaining=3，再次进入 1F1B 循环
  - 执行 forward_step 执行 Stage1 前向计算得到 output_tensor (Forward2)
  - 执行 backward_step 得到 iNPUt_tensor_grad (Backward 2)
  - 执行 send_backward_recv_forward 异步发送 Bwckward 2 给 NPU0
  - 等待NPU0 发送下一个 mbs 的forward计算结果



NPU0 执行 Stage0

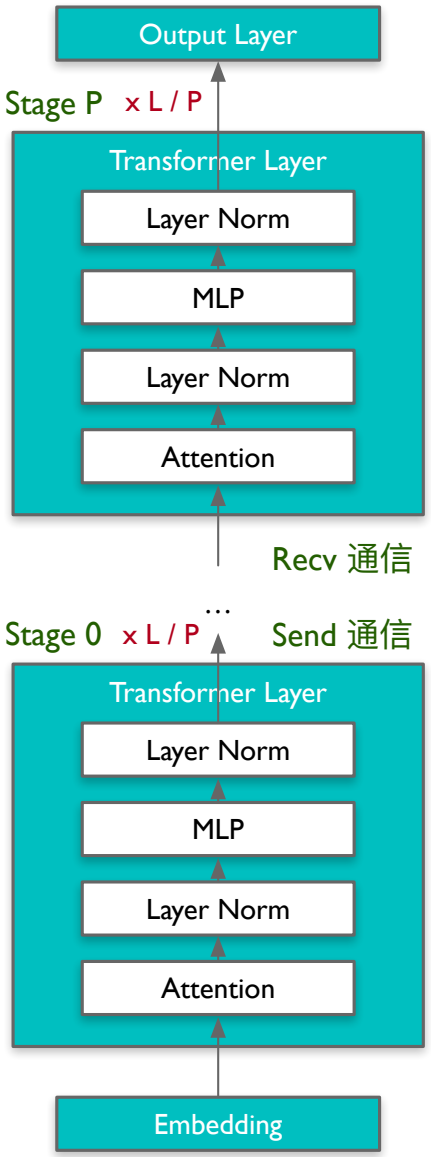



```
n > core > pipeline_parallel > schedules.py > forward_backward_pipelining_without_interleavin
def forward_backward_pipelining_without_interleaving(
            output_tensor_grad = send_forward_recv_backward(
                output_tensor, send_tensor_shapes, config
            )

            # Add input_tensor and output_tensor to end of list.
            input_tensors.append(input_tensor)
            output_tensors.append(output_tensor)
            deallocate_output_tensor(output_tensor[0], config.deallocate_pipeline_ou

            # Pop input_tensor and output_tensor from the start of the list for
            # the backward pass.
            input_tensor = input_tensors.pop(0)
            output_tensor = output_tensors.pop(0)

            # Enable grad sync for the last microbatch in the batch if the full
            # backward pass completes in the 1F1B stage.
            if num_warmup_microbatches == 0 and last_iteration:
                if config.grad_sync_func is None or rank == 0:
                    enable_grad_sync()

            input_tensor_grad = backward_step(
                input_tensor, output_tensor, output_tensor_grad, model_type, config
            )
```

```
def forward_backward_pipelining_without_interleaving(
    # Run cooldown backward passes.
    if not forward_only:
        for i in range(num_warmup_microbatches):

            # Enable async grad reduction in the last backward pass
            # Note: If grad sync function is provided, only enable
            # async grad reduction in first pipeline stage. Other
            # pipeline stages do grad reduction during pipeline
            # bubble.
            if i == num_warmup_microbatches - 1:
                if config.grad_sync_func is None or rank == 0:
                    enable_grad_sync()

            input_tensor = input_tensors.pop(0)
            output_tensor = output_tensors.pop(0)

            output_tensor_grad = recv_backward(send_tensor_shapes, config)

            input_tensor_grad = backward_step(
                input_tensor, output_tensor, output_tensor_grad, model_type, config
            )

            send_backward(input_tensor_grad, recv_tensor_shapes, config)
```

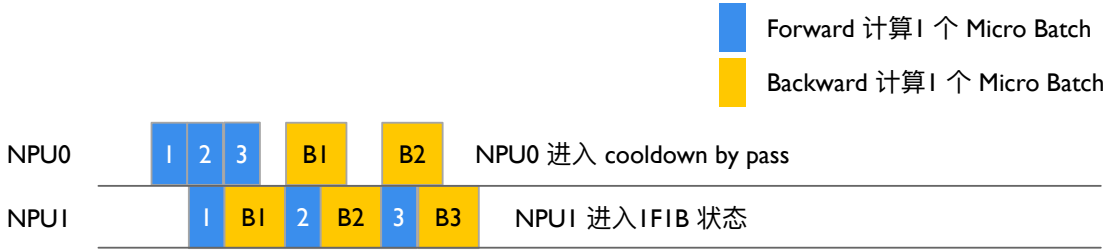
- NPU0 执行 Strage0：
  - NPU0 等待 send_forward_recv_backward 执行 NPU1 输出output_grad(B2)
  - 执行 backward_step 输出 iNPUt_tensor_grad（B2）
  - NPU0 num_warmup_mbs=1, num_mbs_remaing=2, i=2，退出 1F1B
  - 进入 cooldown backwrd pass enable_grad_sync 打开模型梯度更新
  - recv_backward 等待 NPU1 发送最后一个 mbs 的backward（B3）
  - NPU0 准备更新模型的梯度和参数

NPU1 执行 Stage1




```
core > pipeline_parallel > schedules.py > forward_backward_pipelining_without_int
def forward_backward_pipelining_without_interleaving(
    # Run cooldown backward passes.
    if not forward_only:
        for i in range(num_warmup_microbatches):

            # Enable async grad reduction in the last backward pass
            # Note: If grad sync function is provided, only enable
            # async grad reduction in first pipeline stage. Other
            # pipeline stages do grad reduction during pipeline
            # bubble.
            if i == num_warmup_microbatches - 1:
                if config.grad_sync_func is None or rank == 0:
                    enable_grad_sync()

            input_tensor = input_tensors.pop(0)
            output_tensor = output_tensors.pop(0)

            output_tensor_grad = recv_backward(send_tensor_shapes, config)

            input_tensor_grad = backward_step(
                input_tensor, output_tensor, output_tensor_grad, model_type,
            )

            send_backward(input_tensor_grad, recv_tensor_shapes, config)
```

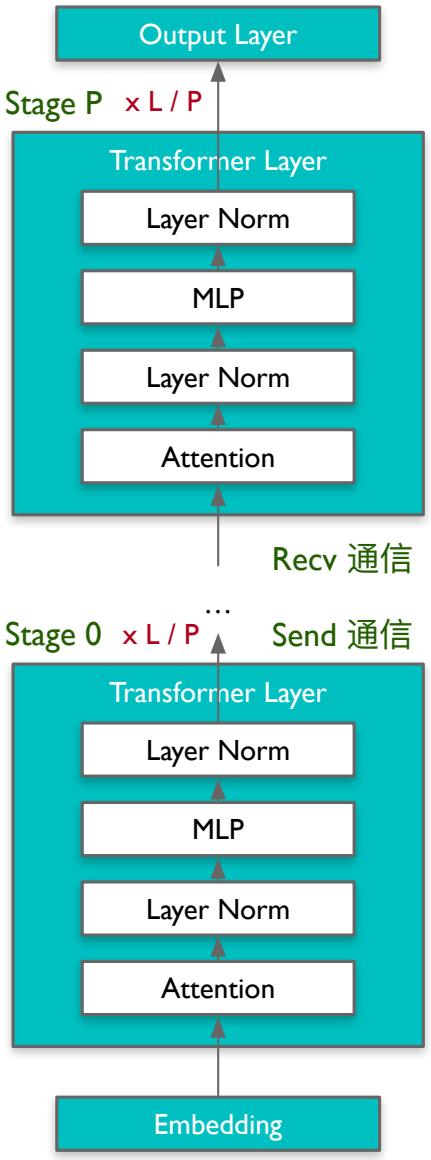
- NPU1 执行 Stage1：
  - NPU1 Stage1 执行 send_backward_recv_forward 同步等待 iNPUt (F3)
  - NPU1 num_warmup_mbs=0，num_mbs_remaining=3，进入 1F1B 循环
  - 将 NPU0 Stage0 发送 iNPUt (F3) 作为 TransformerBlock 的 iNPUt 计算前向
  - forward_step() 输出output (F3) 执行 backward_step() 得到 iNPUt_tensor_grad(B3)
  - send_backward() 异步发送 iNPUt_tesnor_grad(B3) 给 NPU0



Megatron-LM 05

执行完 Iteration

NPU0 执行 Stage0

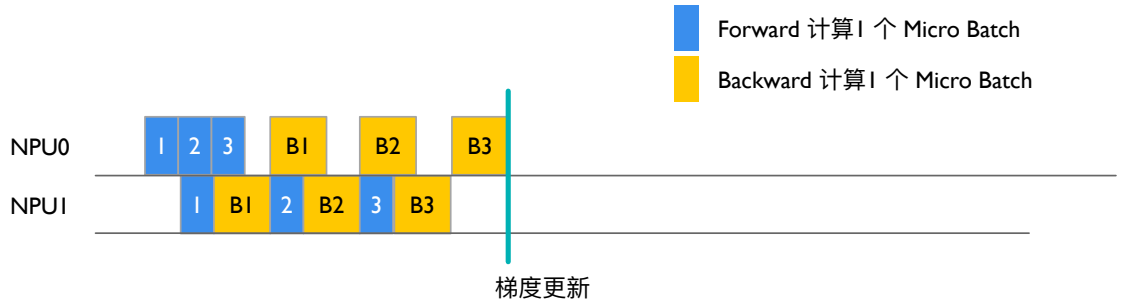



```python
def get_forward_backward_func():
    """Retrieves the appropriate forward_backward function given the…
    pipeline_model_parallel_size = parallel_state.get_pipeline_model_parallel_world_size()
    if pipeline_model_parallel_size > 1:
        if parallel_state.get_virtual_pipeline_model_parallel_world_size() is not None:
            forward_backward_func = forward_backward_pipelining_with_interleaving
        else:
            forward_backward_func = forward_backward_pipelining_without_interleaving
    else:
        forward_backward_func = forward_backward_no_pipelining
    return forward_backward_func
```

```python
def forward_backward_pipelining_without_interleaving(
                output_tensor_grad = recv_backward(send_tensor_shapes, config)

                input_tensor_grad = backward_step(
                    input_tensor, output_tensor, output_tensor_grad, model_type, co
                )

                send_backward(input_tensor_grad, recv_tensor_shapes, config)

            # Launch any remaining grad reductions.
            if no_sync_context is not None:
                enable_grad_sync()
                if config.grad_sync_func is not None:
                    config.grad_sync_func(model.parameters())

    if config.timers is not None:
        config.timers('forward-backward').stop()

    if config.finalize_model_grads_func is not None and not forward_only:
        # Finalize model grads (perform full grad all-reduce / reduce-scatter f
        # data parallelism, layernorm all-reduce for sequence parallelism, and
        # embedding all-reduce for pipeline parallelism).
        config.finalize_model_grads_func([model])

    return forward_data_store
```

- NPU0 执行 Stage0：
  - NPU0 等待 cooldown backward 的 recv_backward() 获得 NPU1 输出 (B3)
  - 执行 backward_step() 输出 iNPUt_tensor_grad (B3)
  - forward_backward_func() 返回 LOSS
  - enable_grad_sync() 累加更新模型梯度
  - finalize_model_grads_func() 更新模型参数

- 执行完一个完整 iteration：
  - NPU0 3 次 send 3 次 recv P2P 通信
  - NPU1 3 次 send 3 次 recv P2P 通信
  - Bubble Time Overhead = (PP-1)/mbs = 1/3



Thank you

把AI系统带入每个开发者、每个家庭、每个组织，构建万物互联的智能世界

Bring AI System to every person, home and organization for a fully connected, intelligent world.

ZOMI

Course chenzomi12.github.io

GitHub github.com/chenzomi12/DeepLearningSystem