PR #2135 完整报告

THUDM/slime

feat(gemma4): add Gemma4 dense and MoE support

合并时间：2026-06-29 15:43

原文链接：http://prhub.com.cn/THUDM/slime/pull/2135

执行摘要

- 一句话：新增 Gemma4 密集型和 MoE 模型完整支持
- 推荐动作：该 PR 设计质量高，实现完整，测试覆盖全面。建议精读以下设计点：① Gemma4Router 中 renormalise-then-scale 的顺序决策（与 HF 一致）；② DualRotaryEmbedding 拼接而非元组的设计理由；③ _Gemma4LogitSoftcap 使用 autograd Function 实现就地 softcapping；④ 通过 hook 而非子类化注入模型行为的选择。这些决策值得在其他模型支持中借鉴。

功能与动机

在 Slime 框架中原生支持 Google 最新发布的 Gemma4 系列模型（31B dense 和 26B-A4B MoE），使框架用户能够直接使用 Gemma4 进行对话训练和 RL 微调。本 PR 在早期集成分支 #1855 的基础上进行了刷新和精简，单独处理了 colocated weight bucket 的空桶修复（#2134）以避免混合依赖。

实现拆解

1. Megatron Transformer 层实现(slime_plugins/models/gemma4.py)：定义 Gemma4TransformerConfig 扩展自 Gemma3 配置，新增 global_kv_channels、attention_k_eq_v、enable_moe_block等字段。实现VNorm（无学习尺度的RMSNorm）、Gemma4Router（含 per-expert scale 的自定义路由方程）、Gemma4MoELayer（插入 Megatron MoE 架构）以及异构注意力层（全局层 head_dim=512，num_kv_heads=4；滑动层 head_dim=256，num_kv_heads=16；全局层 K=V 共享）。
2. 模型 Provider(slime_plugins/models/gemma4_provider.py)：提供 DualRotaryEmbedding 合并全局和局部 RoPE 为单个张量以避免 Megatron 注意力模块误解。通过 _install_hooks 注入 embedding 缩放、logit softcapping（使用自定义 _Gemma4LogitSoftcap autograd Function 实现就地操作）和 layer_scalar。
3. HF ↔ Megatron 桥接(slime_plugins/mbridge/gemma4.py)：Gemma4Bridge 继承 Gemma3Bridge，定义了 attention、MLP、other 三套权重名称映射规则，支持 dense 和 MoE 模式；对 MoE 专家权重进行堆叠拼合以匹配 SGLang 加载器期望的 3D 张量格式。
4. Megatron → HF 转换(slime/backends/megatron_utils/megatron_to_hf/gemma4.py)：convert_gemma4_to_hf 根据层类型（全局 / 滑动）解包 QKV 权重，处理 K=V 共享时跳过 V_proj；_buffer_expert_and_maybe_flush 累加所有专家权重后一次性发射堆叠张量。

5. 损失掩码(slime/utils/mask_utils.py)：新增 Gemma4 聊天模板专用的 loss mask 类型，在测试中验证。
6. GSM8K 验证脚本(scripts/run-gemma4-31B-gsm8k.sh、scripts/run-gemma4-26B-A4B-gsm8k.sh)：提供可直接运行的 slurm 训练脚本，并关联公开 W&B 运行证明。文档中中英文说明了验证拓扑和命令。
7. 测试(tests/gemma4/ 目录)：test_gemma4_router.py 验证路由方程正确性；test_gemma4_provider.py 验证 softcap hook；test_gemma4_bridge.py 验证桥接映射；test_gemma4_qkv_roundtrip.py 验证 QKV 转换往复一致性；test_gemma4_cp_attention.py 验证上下文并行注意力；test_gemma4_layer_integration.py 验证层构建和正向传播；tests/utils/test_loss_mask_type_gemma4.py 验证 loss mask；_standalone_imports.py 提供 Megatron 和 mbridge stub 用于无环境运行测试。

关键文件：

- slime_plugins/models/gemma4.py（模块 模型层；类别 source；类型 core-logic；符号 Gemma4TransformerConfig, VNorm, Gemma4Router, Gemma4MoELayer）：核心模型层：定义 Gemma4 的 Transformer 配置、VNorm、异构注意力、自定义路由器 (Gemma4Router) 及 MoE 层，是本次支持的最关键文件。
- slime_plugins/models/gemma4_provider.py（模块 模型提供器；类别 source；类型 core-logic；符号 model_provider, DualRotaryEmbedding, _Gemma4LogitSoftcap, _install_hooks）：模型 Provider：提供 GPTModel 构建、DualRotaryEmbedding、logit softcapping 和 hook 注入机制，是模型初始化的入口。
- slime_plugins/mbridge/gemma4.py（模块 桥接转换；类别 source；类型 dependency-wiring；符号 Gemma4Bridge, _weight_name_mapping_attention, _weight_name_mapping_mlp, _weight_name_mapping_other）：桥接转换：实现 Gemma4 HF 检查点与 Megatron 权重的双向映射，特别是处理 dense 与 MoE 模式下的不同键名和专家权重堆叠。
- slime/backends/megatron_utils/megatron_to_hf/gemma4.py（模块 权重转换；类别 source；类型 dependency-wiring；符号 convert_gemma4_to_hf, _get_config, _buffer_expert_and_maybe_flush, reset_expert_buffers）：Megatron → HF 权重转换：将 Megatron 分布式权重转换为 HF 格式，处理异构注意力 QKV 解包和 MoE 专家堆叠。
- tests/gemma4/test_gemma4_router.py（模块 路由器测试；类别 test；类型 test-coverage；符号 _make_router_config, test_router_outputs_have_correct_shapes, test_router_weights_sum_to_one_before_per_expert_scale, test_router_per_expert_scale_multiplies_output）：路由器测试：验证 Gemma4Router 的路由方程正确性，包括形状、权重和、per-expert scale 以及 MoE route 方法的打包逻辑。
- tests/gemma4/test_gemma4_provider.py（模块 提供器测试；类别 test；类型 test-coverage；符号 test_install_hooks_softcap_wraps_tensor_output, test_install_hooks_softcap_reuses_storage_with_correct_gradient, _CaptureOutput, test_install_hooks_softcap_wraps_tuple_output）：Provider 测试：验证 logit softcapping hook 的正确性（前向、反向、存储复用、tuple 输出、关闭时行为）。

关键符号：Gemma4Router.init, Gemma4Router.forward, DualRotaryEmbedding.forward, _Gemma4LogitSoftcap.forward, _Gemma4LogitSoftcap.backward, model_provider, convert_gemma4_to_hf, Gemma4Bridge._weight_name_mapping_attention, Gemma4Bridge._weight_to_mcore_format, reset_expert_buffers

关键源码片段

slime_plugins/models/gemma4.py

核心模型层：定义 Gemma4 的 Transformer 配置、VNorm、异构注意力、自定义路由器 (Gemma4Router) 及 MoE 层，是本次支持的最关键文件。

```
class Gemma4Router(nn.Module):
    """Gemma4 MoE 路由器

    路由方程（与 HuggingFace Gemma4TextTopkRouter 一致）：
        h_norm   = VNorm(h)                          # RMSNorm 无学习参数
        h_scaled = h_norm * scale / sqrt(H)          # 可学习的 per-hidden 缩放
        logits   = proj(h_scaled)                    # [T, E]
        probs    = softmax(logits, dim=-1)
        top_w, top_i = topk(probs, k=top_k)
        top_w    = top_w / top_w.sum(dim=-1, keepdim=True)  # 重新归一化
        top_w    = top_w * per_expert_scale[top_i]           # 每个专家缩放

    注意：先归一化再缩放，顺序与 HF 一致，逆序会抵消 per_expert_scale。
    测试 test_router_matches_hf_reference_equation 保护此行为。
    """

    def __init__(self, config):
        super().__init__()
        self.hidden_size = config.hidden_size
        self.num_experts = config.num_moe_experts
        self.top_k = config.moe_router_topk
        self.scalar_root_size = self.hidden_size ** -0.5
        self.norm = VNorm(self.hidden_size, eps=config.layernorm_epsilon)
        self.proj = nn.Linear(self.hidden_size, self.num_experts, bias=False)
        self.scale = nn.Parameter(torch.ones(self.hidden_size))
        # per_expert_scale 是从 HF 检查点加载的固定缓冲，不含梯度
        self.register_buffer("per_expert_scale", torch.ones(self.num_experts))

    def forward(self, hidden_states: torch.Tensor):
        # 输入形状：[T, H] 或 [B, T, H]；若为 3D 则展平第一维
        if hidden_states.dim() == 3:
            batch, seq, _ = hidden_states.shape
            hidden_states = hidden_states.reshape(-1, self.hidden_size)
        else:
            batch, seq = None, None
        T = hidden_states.size(0)

        # 步骤 1-2: VNorm + 缩放
```

```
        h_normed = self.norm(hidden_states)
        h_scaled = h_normed * self.scale * self.scalar_root_size

        # 步骤 3: 投影到专家 logits
        logits = self.proj(h_scaled)

        # 步骤 4: softmax 得到概率
        probs = F.softmax(logits, dim=-1, dtype=torch.float32)

        # 步骤 5: top-k 选取
        top_weights, top_indices = torch.topk(probs, k=self.top_k, dim=-1)

        # 步骤 6: 重新归一化（先归一化再缩放，顺序重要）
        top_weights = top_weights / top_weights.sum(dim=-1, keepdim=True)

        # 步骤 7: per_expert_scale 缩放
        top_weights = top_weights * self.per_expert_scale[top_indices]

        # 如果输入是 3D，恢复形状
        if batch is not None:
            top_weights = top_weights.view(batch, seq, -1)
            top_indices = top_indices.view(batch, seq, -1)

        return top_weights, top_indices
```

slime_plugins/models/gemma4_provider.py

模型 Provider：提供 GPTModel 构建、DualRotaryEmbedding、logit softcapping 和 hook 注入机制，是模型初始化的入口。

```
class DualRotaryEmbedding(torch.nn.Module):
    """双 RoPE 嵌入：将全局和局部 RoPE 拼接成一个张量。

    选择 `torch.cat` 而非元组，因为 Megatron 的 SelfAttention.forward
    会将 2 元组解释为 (self_attn_rope, cross_attn_rope)，导致误解。
    拼接后全局部分在前，Gemma4TransformerLayer 根据 is_sliding 切片。
    """

    def __init__(self, local_rope, global_rope, global_dim: int):
        super().__init__()
        self.local_rope = local_rope
        self.global_rope = global_rope
        self.global_dim = global_dim # 全局 RoPE 维度，用于切片

    def get_rotary_seq_len(self, *args, **kwargs):
        # 委托给 local_rope，因为它管理序列长度逻辑
        return self.local_rope.get_rotary_seq_len(*args, **kwargs)

    def forward(self, seq_len, **kwargs):
        global_emb = self.global_rope(seq_len, **kwargs)
```

```
        local_emb = self.local_rope(seq_len, **kwargs)
        # 拼接：[global_part, local_part]
        return torch.cat([global_emb, local_emb], dim=-1)


class _Gemma4LogitSoftcap(torch.autograd.Function):
    """Gemma4 最终 logit 软裁剪，原地操作避免额外分配。

    forward: logits = tanh(logits / scale) * scale
    backward: 手动实现 tanh 导数，梯度就地回写。
    """

    @staticmethod
    def forward(ctx, logits: torch.Tensor, scale: float) -> torch.Tensor:
        ctx.scale = scale
        ctx.mark_dirty(logits)
        logits.div_(scale)
        logits.tanh_()
        logits.mul_(scale)
        ctx.save_for_backward(logits) # 保存裁剪后的值用于梯度
        return logits

    @staticmethod
    def backward(ctx, grad_output: torch.Tensor) -> tuple[torch.Tensor, None]:
        (softcapped,) = ctx.saved_tensors
        scale = ctx.scale
        # d(output)/d(input) = 1 - tanh(x/scale)^2
        grad_logits = softcapped / scale
        grad_logits.pow_(2)
        grad_logits.neg_()
        grad_logits.add_(1.0)
        grad_logits.mul_(grad_output)
        return grad_logits, None
```

slime_plugins/mbridge/gemma4.py

桥接转换：实现 Gemma4 HF 检查点与 Megatron 权重的双向映射，特别是处理 dense 与 MoE 模式下的不同键名和专家权重堆叠。

```
@register_model(["gemma4", "gemma4_text", "gemma4_unified_text"])
class Gemma4Bridge(Gemma3Bridge):
    """Gemma4 桥接：处理文本 dense 和 MoE 变体。

    Megatron 侧键名不含 language_model. 前缀（纯文本模型），
    HF 侧键名带有 model.language_model. 前缀。
    """

    # Attention 权重映射
    _ATTENTION_MAPPING = {
        "decoder.layers.{layer_number}.self_attention.linear_qkv.weight": [
```

```
            "model.language_model.layers.{layer_number}.self_attn.q_proj.weight",
            "model.language_model.layers.{layer_number}.self_attn.k_proj.weight",
            "model.language_model.layers.{layer_number}.self_attn.v_proj.weight",
        ],
        "decoder.layers.{layer_number}.self_attention.linear_proj.weight": [
            "model.language_model.layers.{layer_number}.self_attn.o_proj.weight",
        ],
        ... # 其他 Q/K 层归一化映射省略
    }

    # MLP 映射：同时覆盖 dense_mlp 和 mlp（MoE 场景）
    _MLP_MAPPING = {
        "decoder.layers.{layer_number}.mlp.linear_fc1.weight": [
            "model.language_model.layers.{layer_number}.mlp.gate_proj.weight",
            "model.language_model.layers.{layer_number}.mlp.up_proj.weight",
        ],
        "decoder.layers.{layer_number}.dense_mlp.linear_fc2.weight": [
            "model.language_model.layers.{layer_number}.mlp.down_proj.weight",
        ],
        "decoder.layers.{layer_number}.mlp.router.proj.weight": [
            "model.language_model.layers.{layer_number}.router.proj.weight",
        ],
        ...
    }

    # 正则表达式匹配 MoE 专家权重
    _RE_MOE_EXPERT = re.compile(
        r"^decoder\.layers\.(\d+)\.mlp\.experts\.linear_fc([12])\.weight(\d+)$"
    )
```

评论区精华

PR 获得了一次性批准（zhuzilin 批准）。主要讨论围绕 PR body 中提到的与 #2134 的依赖关系以及早期分支 #1855 的关系。该 PR 刻意将 colocated weight bucket 空桶修复分离到 #2134，使本 PR 专注于模型支持本身。

- 分离 colocated weight bucket 修复到 #2134 (design): 接受分离设计，本 PR 专注于模型支持，#2134 需单独合并以避免 MoE 训练时 crash。

风险与影响

- 风险： 1) MoE 专家权重堆叠的累积逻辑依赖于 _buffer_expert_and_maybe_flush 中的全局状态，如果转换中断可能导致状态泄漏，但提供了 reset_expert_buffers 函数可重置。 2) Gemma4 损失掩码类型需要与训练脚本中的 --loss_mask_type 参数正确配置，若使用错误的 mask 类型可能导致训练信号损坏。 3) 异构注意力的实现依赖于层类型索引的全局注意力集合，该集合在桥接转换时静态生成，如果层类型配置变化需同步更新。 4) 性能风险：全局层和滑动层的 head_dim 不同，可能导致 Megatron 的某些融合操作无法最优发挥。 5) 依赖 #2134 的修复如果未应用，MoE 路径在 colocated 权重同步时可能 crash。

- 影响：用户可以直接在 Slime 中使用 --model_type gemma4 加载 HuggingFace 上的 Gemma4 官方检查点进行 Megatron 分布式训练和 SGLang 推理。系统新增约 5200 行代码，主要分布在 slime_plugins/models/、slime_plugins/mbridge/、slime/backends/megatron_utils/ 和测试目录。对后端现有模型无影响，但扩展了框架支持的模型家族。团队需熟悉 Gemma4 的异构注意力和 MoE 架构以进行后续维护。
- 风险标记：核心路径变更，依赖未合入修复，大规模新代码引入，异构注意力路径验证，MoE 专家堆叠状态管理

关联脉络

- PR #2134 fix: handle empty colocated weight buckets: 本 PR MoE 验证脚本依赖此修复，PR body 明确要求先合入 #2134 以避免 colocated 权重同步 crash。
- PR #1855 早期 Gemma4 集成分支：本 PR 是 #1855 的刷新和精简版本，保留了相同目标但重新调整了范围，分离了无关修复。
- PR #2102 Support top_p mask: 同为 loss mask 相关功能，本 PR 新增了 Gemma4 特定的 loss mask 类型，与 top_p mask 共用 mask_utils.py 基础设施。