PR #43169 完整报告

vllm-project/vllm

[Perf][Gemma4] Batch vision encoder calls for image and video processing

合并时间：2026-05-21 12:22

原文链接：http://prhub.com.cn/vllm-project/vllm/pull/43169

执行摘要

- 一句话：批量化 Gemma4 视觉编码，吞吐提升最高 3.8x
- 推荐动作：值得精读。PR 展示了如何通过分解模型调用并引入动态批量调度来显著加速多模态编码阶段。设计中的权衡（pooler 不批量化）和兼容性修复（使用 current_platform）值得关注。可作为其他多模态模型性能优化的参考模式。

功能与动机

PR body 中明确说明串行编码是性能瓶颈：'the encoder cannot overlap work across items in the same batch'。通过图像按分辨率分桶和视频帧批量处理，期望提升吞吐，尤其是视频场景。作者还提及受他人实验启发。

实现拆解

1. 分解 vision tower 调用：将原有的 vt() 拆分为 patch_embedder->encoder->pooler->standardize->embed_vision 五个子阶段，以便中间结果可以批量组合。
2. 图像批量处理：在 _process_image_input 中，根据每张图像的有效 patch 数（来自 pixel_position_ids mask）分组，同一组内图像具有相同形状，直接堆叠为 batch 调用 encoder，每组内部再根据动态 batch 上限拆分为更小的 chunk 防止 OOM。Pooling 逐图像执行，最后拼接后一次性投影再按长度拆回列表。
3. 视频批量处理：在 _process_video_input 中，将所有帧的 pixel values 拼为大张量，以内存受限的 chunk 编码；Pooling 在每帧上独立执行（保持逐帧以保证数值一致性）；拼接所有帧的 pooled 结果后一次性投影；按 frame_counts 重新分组为每个视频的嵌入列表。
4. 动态 batch 大小（新增 _encoder_max_batch 方法）：基于总 GPU 显存的 5%（首次调用时采样）和每 patch 的字节数（_encoder_bytes_per_patch），计算当前分辨率下能安全放入 batch 的最大图像数。_encoder_bytes_per_patch 在第一组图像编码时通过一次前向估算。
5. 配套调整：__init__ 中增加 _encoder_budget_bytes 和 _encoder_bytes_per_patch 惰性初始化为 0；导入 vllm.platforms.current_platform 以支持非 CUDA 设备内存查询；删除批量化相关的 TODO 和新增的中文注释。

关键文件：

- vllm/model_executor/models/gemma4_mm.py（模块 多模态模型；类别 source；类型 core-logic；符号 _encoder_max_batch, _process_image_input, _process_video_input,

_encoder_budget_bytes）：实现所有批量化逻辑的核心文件，包括视觉编码器调用分解、图像 / 视频批处理、动态 batch 大小计算等。

关键符号：_encoder_max_batch, _process_image_input, _process_video_input

关键源码片段

vllm/model_executor/models/gemma4_mm.py

实现所有批量化逻辑的核心文件，包括视觉编码器调用分解、图像 / 视频批处理、动态 batch 大小计算等。

```
# vllm/model_executor/models/gemma4_mm.py

# ---------------------------------------------------------------------
# 动态 encoder 最大批量计算：基于可用 GPU 内存，避免 OOM
# ---------------------------------------------------------------------
def _encoder_max_batch(self, patches_per_item: int) -> int:
    """根据每个 item 的 patch 数量，计算单次 encoder 能处理的最大 item 数。"""
    if self._encoder_budget_bytes == 0:
        # 首次调用时计算预算（总显存的 5%）
        total_mem = current_platform.get_device_total_memory()
        self._encoder_budget_bytes = int(total_mem * 0.05)
        logger.info("Encoder memory budget: %.1fGB (total=%.1fGB)",
                    self._encoder_budget_bytes / 1024**3,
                    total_mem / 1024**3)
    # 每个 item 预估内存 = patch 数 * 每 patch 字节数
    cost = patches_per_item * self._encoder_bytes_per_patch
    return max(1, self._encoder_budget_bytes // cost) if cost > 0 else 1


# ---------------------------------------------------------------------
# 图像批处理入口：按分辨率桶分组，批量调用 encoder 子阶段
# ---------------------------------------------------------------------
def _process_image_input(self, image_input: Gemma4ImageInputs) -> list[torch.Tensor]:
    pixel_values = image_input["pixel_values"] # (num_images, max_patches, patch_size)
    pixel_position_ids = image_input["pixel_position_ids"] # (num_images, max_patches, 2)

    # 计算每张图像的实际 patch 数（通过有效位置标记）
    valid_mask = pixel_position_ids[..., 0] != -1 # -1 表示 padding
    patches_per_image = valid_mask.sum(dim=-1).tolist()

    # 按 patch 数分组（分辨率桶），每组内部形状一致，无需 padding
    from collections import defaultdict
    groups: dict[int, list[int]] = defaultdict(list)
    for idx, patches in enumerate(patches_per_image):
        groups[patches].append(idx)

    pooled_outputs = []
    for patches, indices in groups.items():
```

```
            # 受显存预算限制，每组可能再拆成多个迷你 batch
            batch_size = self._encoder_max_batch(patches)
            for start in range(0, len(indices), batch_size):
                batch_indices = indices[start:start + batch_size]
                batch_pixels = pixel_values[batch_indices] # (B, P, ...)
                # 1) patch embedding（共享 weights，无状态）
                patch_embeds = self.vision_tower.patch_embedder(batch_pixels)
                # 2) Transformer encoder（最吃显存的部分）
                encoder_output = self.vision_tower.encoder(patch_embeds)
                # 3) pooling（逐图，得到变长输出）
                for i in range(len(batch_indices)):
                    # 用每图的 valid_mask 去除 padding 后 pool
                    feats = encoder_output[i][valid_mask[batch_indices[i]]]
                    pooled = self.vision_tower.pooler(feats.unsqueeze(0))
                    pooled_outputs.append(pooled.squeeze(0))
        # 所有 pooled 结果拼接后，一次性投影（RMSNorm + Linear）
        if pooled_outputs:
            flat = torch.cat(pooled_outputs)
            projected = self.embed_vision(flat)
            # 按各图输出长度拆分回 list
            output_lengths = [len(p) for p in pooled_outputs]
            return list(torch.split(projected, output_lengths))
        return []
```

评论区精华

Review 中主要有三个讨论点：

- 兼容性修复：Isotr0py 指出直接使用 torch.cuda.mem_get_info 会破坏 OOT 硬件支持（如 vllm-ascend），作者改用 current_platform.get_device_total_memory()。
- Pooler 是否批量化：gemini-code-assist[bot] 建议批量执行视频帧 pooling，但作者明确拒绝以保证数值一致性，且 pooler 不是瓶颈。
- Regrouping 优化：同一 bot 建议用 slicing 替代 torch.cat 重组视频帧嵌入，作者接受并实现。 此外，Isotr0py 提出未来可将 ViT 移植使用 MMEncoderAttention 以进一步降低显存，作者创建了 #43178 跟踪。
- CUDA only API 不兼容 (correctness): 改用 current_platform.get_device_total_memory()。
- Pooler batching 权衡 (design): 维持现状，不批量化。
- Video regrouping 优化建议 (performance): 采用 slicing 优化。

风险与影响

- 风险：动态 batch 计算基于总显存的 5%，该比例可能不适合所有模型或部署场景（如显存极小的设备）；_encoder_bytes_per_patch 在首次前向时估算，若后续批次的分辨率变化可能引入误差但同一模型通常稳定。变更仅影响 Gemma4 模型路径，不波及编码器或 decoder 的其他路径。缺少单元测试（仅有端到端 benchmark），内部逻辑修改无回归覆

盖。

- 影响：用户使用 Gemma4 多模态模型尤其是视频场景有显著性能收益：视频吞吐提升 1.7–3.8x，图像 TTFT 降低 1.1–1.5x。TPOT 不受影响，TTFT 在并发时下降。未变更公共 API 或配置选项，对非 Gemma4 模型完全透明。团队成员维护此变更所需成本低（单一文件，逻辑清晰）。
- 风险标记：单模型变更，缺少测试覆盖，动态内存估算

关联脉络

- 暂无明显关联 PR