PR #31438 完整报告

sgl-project/sglang

vlm: parallelize multimodal preprocessing with customized worker num

合并时间：2026-07-21 08:44

原文链接：http://prhub.com.cn/sgl-project/sglang/pull/31438

执行摘要

- 一句话：并行化 VLM 多模态预处理并支持自定义工作线程数
- 推荐动作：## 推荐意见
- 该 PR 设计稳健，性能收益显著，建议合并。
- 值得深入阅读 executor.py 中线程本地克隆的实现，以及 review 中安全性改进的流程。
- 对于部署了多模态模型的团队，推荐启用此功能（默认即开启），并根据实际负载微调 --mm-processor-worker-num。
- 建议后续增加监控指标，便于调优。

功能与动机

Qwen VLM preprocessing currently runs synchronously on the tokenizer event loop. A burst of image requests therefore serializes image/video loading, Hugging Face processor work, and request dispatch before the GPU scheduler can form a useful prefill batch.

实现拆解

实现拆解

1. 新增并行执行器（executor.py）：创建 MultimodalProcessorExecutor 类，内部使用 ThreadPoolExecutor 和线程本地存储（_WorkerState）管理处理器克隆。每个线程首次使用时从预创建列表中弹出一个深拷贝的处理器，避免共享可变 tokenizer 状态。弹出操作受锁保护，列表耗尽时自动退化为深拷贝原始处理器。
2. 基类模型声明（base_processor.py）：BaseMultimodalProcessor 增加三个类变量 auto_mm_processor_worker_num（默认 1）、auto_mm_io_worker_num（默认 4）、supports_mm_processor_concurrency（默认 False）。__init__ 中根据 server_args.mm_processor_worker_num、环境变量 SGLANG_IO_WORKERS 及模型缺省值确定实际工作线程数，并按需创建 MultimodalProcessorExecutor；若并发处理器不被支持或克隆失败，自动将线程数降为 1（同步模式）。
3. 异步处理方法（base_processor.py）：新增 process_and_combine_mm_data_async 异步方法，内部通过 self.mm_processor_executor.run 将同步处理函数调度到线程池，避免阻塞事件循环。

4. 模型适配（qwen_vl.py）：QwenVLImageProcessor.__init__ 中对 Qwen2/2.5/3 VL、Qwen3.5、InternS2 模型设置 auto_mm_processor_worker_num=2、auto_mm_io_worker_num=16、supports_mm_processor_concurrency=True；process_mm_data_async 方法将原有的同步调用改为 await self.process_and_combine_mm_data_async。
5. 新增命令行参数（server_args.py）：添加 --mm-processor-worker-num 和 --mm-io-worker-num，默认 0 表示使用模型默认值；check_server_args 增加了非负断言。
6. 测试配套（test_mm_process_config.py）：新增 TestMultimodalProcessorConcurrency 测试类，覆盖模型特定默认值启用 executor、显式 1 线程禁用 executor、并发要求模型支持、I/O 数显式覆盖自动值、专用 executor 离线运行、单线程保持同步路径等场景。

关键文件：

- python/sglang/srt/multimodal/processors/executor.py（模块 处理器执行器；类别 source；类型 core-logic；符号 _WorkerState, init, MultimodalProcessorExecutor, run）：新增 MultimodalProcessorExecutor 类，实现线程池和线程本地处理器克隆的核心逻辑。
- python/sglang/srt/multimodal/processors/base_processor.py（模块 处理器基类；类别 source；类型 dependency-wiring；符号 _resolve_processor, process_and_combine_mm_data_async）：基类添加并发支持：类变量、初始化 executor、新增异步方法。
- test/registered/unit/managers/test_mm_process_config.py（模块 配置测试；类别 test；类型 test-coverage；符号 _make_processor, test_model_specific_auto_worker_count_enables_executor, test_explicit_single_worker_disables_executor, test_parallel_workers_require_processor_support）：新增测试覆盖自动配置、显式配置、并发降级、线程安全回归等场景。
- python/sglang/srt/multimodal/processors/qwen_vl.py（模块 VL 处理器；类别 source；类型 core-logic）：为 Qwen VL 系列设置默认并发参数，并切到异步方法。
- python/sglang/srt/server_args.py（模块 服务参数；类别 source；类型 core-logic）：添加 --mm-processor-worker-num 和 --mm-io-worker-num 命令行参数。

关键符号：MultimodalProcessorExecutor.init, MultimodalProcessorExecutor.run, MultimodalProcessorExecutor._run, MultimodalProcessorExecutor.shutdown, BaseMultimodalProcessor.init, BaseMultimodalProcessor.process_and_combine_mm_data_async, QwenVLImageProcessor.init, ServerArgs.check_server_args

关键源码片段

python/sglang/srt/multimodal/processors/executor.py

新增 MultimodalProcessorExecutor 类，实现线程池和线程本地处理器克隆的核心逻辑。

```
import asyncio
import concurrent.futures
import copy
import threading
```

```
from typing import Any, Callable, TypeVar

T = TypeVar("T")

class _WorkerState(threading.local):
    """线程本地存储，每个线程持有自己的处理器克隆引用。"""
    def __init__(self):
        self.processor = None

class MultimodalProcessorExecutor:
    """在隔离的线程本地处理器克隆上运行处理器调用。"""
    def __init__(self, processor: Any, max_workers: int):
        self._processor = processor
        # 预创建指定数量的处理器深拷贝，避免共享可变 tokenizer 状态
        self._processor_clones = [copy.deepcopy(processor) for _ in range(max_workers)]
        self._executor = concurrent.futures.ThreadPoolExecutor(
            max_workers=max_workers,
            thread_name_prefix="sglang-mm-processor",
        )
        self._worker_state = _WorkerState()
        self._clone_lock = threading.Lock()

    async def run(self, function: Callable[..., T], *args: Any, **kwargs: Any) -> T:
        """异步接口，将同步 _run 调度到线程池执行。"""
        loop = asyncio.get_running_loop()
        return await loop.run_in_executor(
            self._executor, self._run, function, args, kwargs
        )

    def _run(self, function, args, kwargs):
        """在各自线程中执行：获取线程本地处理器（延迟分配克隆），然后调用函数。"""
        processor = self._worker_state.processor
        if processor is None:
            with self._clone_lock:
                # 优先从预创建的克隆列表中弹出，若耗尽则深拷贝原始处理器作为后备
                processor = (
                    self._processor_clones.pop()
                    if self._processor_clones
                    else copy.deepcopy(self._processor)
                )
            self._worker_state.processor = processor
        return function(*args, processor=processor, **kwargs)

    def shutdown(self) -> None:
        self._executor.shutdown()
```

python/sglang/srt/multimodal/processors/base_processor.py

基类添加并发支持：类变量、初始化 executor、新增异步方法。

```python
# 在 BaseMultimodalProcessor 类定义前导入
from sglang.srt.multimodal.processors.executor import MultimodalProcessorExecutor

class BaseMultimodalProcessor(ABC):
    models = []
    gpu_image_decode = True
    auto_mm_processor_worker_num = 1
    auto_mm_io_worker_num = 4
    supports_mm_processor_concurrency = False

    def __init__(self, hf_config, server_args, _processor, transport_mode, *args, **kwargs):
        # 决定 I/O 工作线程数：显式 > 环境变量 > 模型默认值
        requested_mm_io_worker_num = self.server_args.mm_io_worker_num
        env_mm_io_worker_num = os.environ.get("SGLANG_IO_WORKERS")
        if requested_mm_io_worker_num:
            self.mm_io_worker_num = requested_mm_io_worker_num
        elif env_mm_io_worker_num is not None:
            self.mm_io_worker_num = int(env_mm_io_worker_num)
        else:
            self.mm_io_worker_num = self.auto_mm_io_worker_num

        # 创建 I/O 线程池
        self.io_executor = concurrent.futures.ThreadPoolExecutor(
            max_workers=self.mm_io_worker_num,
            thread_name_prefix="sglang-mm-io",
        )

        skip_mm_pool = kwargs.get("skip_mm_pool", False)
        requested_mm_processor_worker_num = self.server_args.mm_processor_worker_num
        self.mm_processor_worker_num = (
            1 if skip_mm_pool
            else requested_mm_processor_worker_num or self.auto_mm_processor_worker_num
        )

        # 如果模型不支持并发但请求了 >1，则降级
        if self.mm_processor_worker_num > 1 and not self.supports_mm_processor_concurrency:
            self.mm_processor_worker_num = 1

        self.mm_processor_executor = None
        if self.mm_processor_worker_num > 1:
            try:
                self.mm_processor_executor = MultimodalProcessorExecutor(
                    self._processor, self.mm_processor_worker_num
                )
            except Exception:
                # 克隆失败时降级为同步
                self.mm_processor_worker_num = 1

    async def process_and_combine_mm_data_async(self, base_output, mm_tokens, **kwargs):
```

```
        if self.mm_processor_executor:
            return await self.mm_processor_executor.run(
                self._run_process_and_combine_mm_data,
                base_output, mm_tokens, **kwargs
            )
        else:
            return self._run_process_and_combine_mm_data(base_output, mm_tokens, **kwargs)

    def _run_process_and_combine_mm_data(self, base_output, mm_tokens, **kwargs):
        return self.process_and_combine_mm_data(base_output, mm_tokens, **kwargs)
```

test/registered/unit/managers/test_mm_process_config.py

新增测试覆盖自动配置、显式配置、并发降级、线程安全回归等场景。

```
  def test_model_specific_auto_worker_count_enables_executor(self):
      from sglang.srt.multimodal.processors.base_processor import (
          BaseMultimodalProcessor,
      )
      # 模拟模型设置自己的默认值，并标记支持并发
      with patch.object(BaseMultimodalProcessor, "auto_mm_processor_worker_num", 4), \
           patch.object(BaseMultimodalProcessor, "auto_mm_io_worker_num", 16), \
           patch.object(BaseMultimodalProcessor, "supports_mm_processor_concurrency", True):
          proc = self._make_processor({})
          try:
              self.assertEqual(proc.mm_processor_worker_num, 4)
              self.assertEqual(proc.mm_io_worker_num, 16)
              self.assertIsNotNone(proc.mm_processor_executor)
          finally:
              proc.mm_processor_executor.shutdown()

  def test_explicit_single_worker_disables_executor(self):
      from sglang.srt.multimodal.processors.base_processor import (
          BaseMultimodalProcessor,
      )
      # 即使模型默认支持并发，用户显式指定 1 个处理器工作线程也会禁用 executor
      with patch.object(BaseMultimodalProcessor, "auto_mm_processor_worker_num", 4):
          proc = self._make_processor({}, mm_processor_worker_num=1)
          self.assertEqual(proc.mm_processor_worker_num, 1)
          self.assertIsNone(proc.mm_processor_executor)

  def test_parallel_workers_require_processor_support(self):
      # 请求 2 个处理器线程但模型不支持并发时，应降级为 1
      proc = self._make_processor({}, mm_processor_worker_num=2)
      self.assertEqual(proc.mm_processor_worker_num, 1)
      self.assertIsNone(proc.mm_processor_executor)
```

评论区精华

评论区精华

- gemini-code-assist在 review 中指出原实现使用单调递增索引 self._next_mm_processor_clone 分配克隆，若线程池因空闲超时或异常重建线程，新线程会继续递增索引导致越界，引发 RuntimeError 并永久破坏 worker 池。建议改用锁保护的 pop 并从预创建列表中移除，同时提供深拷贝后备。
- mickqian（作者）在提交 cdea8b36f 中采纳了建议，修正为基于 pop 和锁的实现，并在后续提交中添加回归测试。问题已解决。
- 参数默认值的选择在 PR 描述中有详细性能数据支撑：2 个处理器线程在 Blackwell 上比 4 个线程表现更好（burst 吞吐 +11.7%，TPOT -36.8%）。
- 线程池中处理器克隆分配的正确性 (correctness): 作者在提交 cdea8b36f 中修正为基于 pop 和锁的实现，并添加回归测试验证。

风险与影响

- 风险：## 风险分析
- 线程安全：初始版本存在索引越界风险，已在 commit cdea8b36f 修复并通过测试验证。当前实现使用锁和线程本地存储，基本安全。
- 兼容性：对于未显式声明 supports_mm_processor_concurrency 的模型，即使设置 --mm-processor-worker-num>1 也会自动降级为同步模式，不会出错但可能让用户感到困惑。建议模型维护者及时声明。
- 资源占用：默认新增 2 个处理器线程和 16 个 I/O 线程，在内存和句柄方面占用较小。但若用户自定义过大值，可能增加调度开销。
- 监控缺失：目前没有提供 mm_processor_executor 的监控指标，如排队长度、处理耗时等，故障排查可能依赖日志。
- 影响：## 影响分析
- 用户影响：对于 Qwen VL 系列用户，突发多图像请求场景下性能大幅提升（H200 TTFT -47.6%，吞吐 +80.8%）。其他模型无影响。用户可通过命令行调整并发度。
- 系统影响：新增线程池不会增加 CUDA IPC 池大小（仍为 512 MiB），对 GPU 资源无额外压力。CPU 侧内存占用略有增加（每个线程一个处理器深拷贝）。
- 团队维护：模型添加并行支持的声明方式简单（设置三个类变量），但需要维护者注意线程安全性。修复过程体现了代码审查的价值。
- 风险标记：线程安全修正，默认值调优，降级路径覆盖，模型兼容性

关联脉络

- 暂无明显关联 PR