PR #30918 完整报告

sgl-project/sglang

[Benchmark] Add optional steady-state window for serving metrics

合并时间：2026-08-25 05:49

原文链接：http://prhub.com.cn/sgl-project/sglang/pull/30918

执行摘要

- 一句话：新增 PD 稳态吞吐基准模式，不改共享 serving 代码
- 推荐动作：中等优先级，值得精读。重点关注三点：一是零侵入集成模式——用 inspect.signature().bind() 包装 serving.calculate_metrics 采集数据而不改共享代码；二是事件扫描式稳态窗口算法与确定性 tie-break；三是基于 TTFT/ITL 的输出吞吐近似估算及其明确的局限性。对维护 sglang.benchmark 或做 PD 压测评估的工程师有直接参考价值。

功能与动机

PR body 指出：在 prefill-decode（PD）分离部署中，“Decode workers have not reached their sustained concurrency during ramp-up”且“Requests gradually leave decode workers during drain”，因此“Full-run throughput therefore does not accurately represent steady-state PD performance”。为此新增专用稳态模式，并明确约束“without changing the shared serving.py implementation or affecting existing benchmarks”。

实现拆解

1. 稳态算法模块 python/sglang/benchmark/steady_state.py：用 Protocol 定义 RequestOutput、InputRequest、Tokenizer 的最小字段子集，模块只依赖数据形状而非 serving 内部类型，便于独立测试。find_steady_state_window 采用事件扫描：请求启动 +1、结束（启动 + 延迟）-1，排序后生成并发区间；阈值 = max(1, ceil(峰值并发 × 比例))；合并连续高于阈值的区间为 span，max() 取最长者，时长并列时保留第一个 span 保证确定性。输出吞吐通过 _token_timestamps 用 TTFT 与 ITL 递推估算每个输出 token 的时间戳，再由 _output_tokens_in_window 统计窗口内 token 数（跨边界按比例计入）；输入吞吐按“到达定义”估算——窗口内启动的成功请求的 prompt 长度合计 ÷ 窗口时长，多轮等无法对齐输入的场景报 N/A。
2. 独立入口 python/sglang/benchmark/steady_state_serving.py：新增 python -m sglang.benchmark.steady_state_serving，除 --steady-state-concurrency-ratio 与 --steady-state-output-file 外，其余参数原样转发给 serving。_run_with_capture 通过 inspect.signature().bind() 包装 serving.calculate_metrics，采集 input_requests、outputs、tokenizer 后调用原函数，finally 中还原，实现零侵入采集；结果按固定格式打印，并可追加写入独立 JSONL 文件。
3. 测试配套 test/registered/bench_fn/test_steady_state_benchmark.py：注册 CPU CI（register_cpu_ci(est_time=5, suite='base-a-test-cpu')），覆盖 ramp-up/drain 裁剪与指

标数值、最长连续高并发 span 选择、非法 / 空输入拒绝、monkey-patch 还原四类场景。

4. 配置与部署：无 schema、配置或部署改动；serving.py 与 base 分支保持一致。

关键文件：

- python/sglang/benchmark/steady_state.py（模块 稳态指标；类别 source；类型 core-logic；符号 RequestOutput, InputRequest, Tokenizer, SteadyStateWindow）：稳态窗口算法与指标定义的核心模块，PR 的主体实现；负责并发区间事件扫描、最长高并发窗口选择及输入 / 输出吞吐估算。
- python/sglang/benchmark/steady_state_serving.py（模块 基准入口；类别 source；类型 entrypoint；符号 _run_with_capture, capture_calculate_metrics, run_steady_state_benchmark, _custom_parser）：独立 CLI 入口，承载零侵入采集设计——monkey-patch serving.calculate_metrics 后运行原有 benchmark 并还原，保证 serving.py 不被修改。
- test/registered/bench_fn/test_steady_state_benchmark.py（模块 基准测试；类别 test；类型 test-coverage；符号 _StringTokenizer, _RequestOutput, _InputRequest, _request）：覆盖稳态窗口算法与入口行为的 CPU 单测，验证 ramp-up/drain 裁剪、最长 span 选择、非法输入与采集还原。

关键符号：find_steady_state_window, calculate_steady_state_metrics, steady_state_output_throughput, _token_timestamps, _output_tokens_in_window, _run_with_capture, capture_calculate_metrics, run_steady_state_benchmark, _custom_parser, cli_main, run_and_report, _print_metrics, _append_metrics

关键源码片段

python/sglang/benchmark/steady_state.py

稳态窗口算法与指标定义的核心模块，PR 的主体实现；负责并发区间事件扫描、最长高并发窗口选择及输入 / 输出吞吐估算。

```
# python/sglang/benchmark/steady_state.py —— 稳态窗口选择与输出 token 时序估算

def find_steady_state_window(
    outputs: Sequence[RequestOutput], concurrency_ratio: float
) -> SteadyStateWindow:
    """选取并发度持续高于阈值的最长连续时间窗口。"""
    # 比例必须落在 (0, 1] 区间，0 会让阈值退化为 1 而失去裁剪意义
    if not 0 < concurrency_ratio <= 1:
        raise ValueError('steady-state concurrency ratio must be in (0, 1]')

    # 只保留成功且延迟大于 0 的请求，避免占位结果污染窗口
    successful = [output for output in outputs if output.success and output.latency > 0]
    if not successful:
        raise ValueError('no successful requests with positive latency')

    # 事件扫描：请求启动 +1、结束 -1，把并发度变化折成时间区间
    events = {}
```

```
    for output in successful:
        events[output.start_time] = events.get(output.start_time, 0) + 1
        end_time = output.start_time + output.latency
        events[end_time] = events.get(end_time, 0) - 1

    concurrency = 0
    intervals = []
    event_times = sorted(events)
    for index, event_time in enumerate(event_times[:-1]):
        concurrency += events[event_time]
        next_time = event_times[index + 1]
        if next_time > event_time:
            intervals.append((event_time, next_time, concurrency))

    # 阈值 = max(1, ceil( 峰值并发 * 比例 ))，并保证至少为 1
    peak_concurrency = max((item[2] for item in intervals), default=0)
    threshold = max(1, int(np.ceil(peak_concurrency * concurrency_ratio)))

    # 合并连续高于阈值的区间为 span，随后取最长 span 作为稳态窗口
    spans: List[Tuple[float, float]] = []
    span_start: Optional[float] = None
    span_end: Optional[float] = None
    for start, end, active_requests in intervals:
        if active_requests >= threshold:
            if span_start is None: # 开启一个新的高并发 span
                span_start = start
            span_end = end
        elif span_start is not None and span_end is not None:
            spans.append((span_start, span_end)) # 并发跌破阈值，闭合当前 span
            span_start = span_end = None
    if span_start is not None and span_end is not None:
        spans.append((span_start, span_end))

    if not spans:
        raise ValueError('no steady-state measurement window could be determined')

    # max() 在时长相等时保留第一个 span，让选择结果确定可复现
    window_start, window_end = max(spans, key=lambda span: span[1] - span[0])
    return SteadyStateWindow(
        start=window_start,
        end=window_end,
        duration=window_end - window_start,
        concurrency_threshold=threshold,
        peak_concurrency=peak_concurrency,
    )


def _token_timestamps(output: RequestOutput) -> List[float]:
    # 输出 token 没有逐 token 到达时间戳，用 TTFT 与 ITL 递推估算
```

```
    timestamps = [output.start_time + output.ttft] # 首个输出 token 的到达时刻
    for inter_token_latency in output.itl:
        timestamps.append(timestamps[-1] + inter_token_latency) # 后续 token 依次累加 ITL
    return timestamps
```

python/sglang/benchmark/steady_state_serving.py

独立 CLI 入口，承载零侵入采集设计——monkey-patch serving.calculate_metrics 后运行原有 benchmark 并还原，保证 serving.py 不被修改。

```
# python/sglang/benchmark/steady_state_serving.py —— 零侵入采集：包装 calculate_metrics 后还原

def _run_with_capture(
    args: argparse.Namespace,
    concurrency_ratio: float,
    output_file: Optional[str],
    run_serving_benchmark: Callable[[argparse.Namespace], Dict[str, Any]],
) -> Tuple[Dict[str, Any], SteadyStateMetrics]:
    captured: Dict[str, Any] = {}
    original_calculate_metrics = serving.calculate_metrics
    calculate_signature = inspect.signature(original_calculate_metrics)

    def capture_calculate_metrics(*call_args, **call_kwargs):
        # 用签名绑定调用参数，兼容位置参数与关键字参数两种调用方式
        bound = calculate_signature.bind(*call_args, **call_kwargs)
        input_requests = bound.arguments['input_requests']
        captured['input_requests'] = (
            None if input_requests is None else list(input_requests)
        )
        captured['outputs'] = list(bound.arguments['outputs'])
        captured['tokenizer'] = bound.arguments['tokenizer']
        # 采集完成后仍调用原函数，保证全运行指标计算不受影响
        return original_calculate_metrics(*call_args, **call_kwargs)

    # 临时替换 serving.calculate_metrics，运行完原 benchmark 后立即还原，
    # 从而在不修改 serving.py 的前提下拿到 request 级原始数据
    serving.calculate_metrics = capture_calculate_metrics
    try:
        benchmark_result = run_serving_benchmark(args)
    finally:
        serving.calculate_metrics = original_calculate_metrics

    if 'outputs' not in captured:
        raise RuntimeError('serving benchmark finished without producing request results')

    steady_state_metrics = calculate_steady_state_metrics(
        outputs=captured['outputs'],
        tokenizer=captured['tokenizer'],
        concurrency_ratio=concurrency_ratio,
```

```
        input_requests=captured['input_requests'],
    )
    _print_metrics(steady_state_metrics)
    if output_file:
        _append_metrics(output_file, steady_state_metrics)

    return benchmark_result, steady_state_metrics
```

评论区精华

PR 无内联 review 评论，Fridge003 直接 APPROVED，零侵入采集 + 独立入口的设计未引发争议。唯一交互是 CI 复跑：Fridge003 通过 /rerun-test test/registered/bench_fn/test_steady_state_benchmark.py 请求重跑，github-actions 在 ubuntu-latest 上 1 个测试通过。PR body 中"Keep python/sglang/benchmark/serving.py identical to the base branch"是最核心的设计契约，整套采集方案都围绕它展开。

- 稳态基准单测的 CI 复跑 (testing): 复跑通过，说明新增单测在 CPU CI 环境稳定；无需返工。

风险与影响

- 风险：
  1. monkey-patch 签名耦合：capture_calculate_metrics 依赖 serving.calculate_metrics 的参数名（input_requests、outputs、tokenizer）与调用方式，若未来 serving.py 重构（重命名参数、改签名），bound.arguments[...] 会抛 KeyError；当前仅对"没有 outputs"做了 RuntimeError 兜底，对 KeyError 无保护。
  2. 时序估算近似：_token_timestamps 假设 ITL 列表逐 token 覆盖输出，若 output_len 大于 len(itl) + 1（ITL 未采样全量 token），窗口内输出 token 会被低估；TTFT/ITL 本身是采样而非精确到达时间。输入吞吐是到达定义的估算，多轮路径报 N/A，跨 workload 不可直接对比。
  3. 窗口退化：若并发度恰好在阈值附近抖动，可能选出极短窗口，或完全选不出窗口（find_steady_state_window 抛 ValueError），造成基准结果缺失。
  4. 线程安全：captured 为共享 dict，若未来 serving.run_benchmark 引入并行采集会互相覆盖；当前单线程场景安全。
  5. 兼容性：新增文件不修改任何既有模块，serving.py 零变更，既有基准结果不变，回归面很小。 - 影响：用户：PD 分离部署压测可一键获得稳态吞吐口径，替代被 ramp-up/drain 污染的全运行均值。系统：无运行时路径变更（调度器、缓存等均未触及），零运行时回归面。团队：新增 benchmark 模块与配套单测，后续演进 serving.py 时需同步维护该采集点；对 PD 场景的性能评估方法学是一处补充。 - 风险标记：monkey-patch 签名耦合，token 时序估算近似，窗口选择退化风险

关联脉络

- PR #35840 Add PD test for inkling with mxfp8 KV: 同属 PD 分离部署验证线：该 PR 为 PD 路径补充正确性测试，本 PR 为 PD 压测提供稳态吞吐口径，共同支撑 PD 场景的可信

评估。

- PR #35957 Fix recurrent state loss on decode retraction: 修复 PD 分离下 decode retraction 的状态丢失，与本 PR 关注的 decode worker 稳态并发表现直接相关——稳态基准数值的可信度依赖此类正确性修复落地。