

PR #45187 完整报告

vllm-project/vllm

Add NVFP4 KV 4-over-6 scale search

合并时间: 2026-08-08 04:22

原文链接: <http://prhub.com.cn/vllm-project/vllm/pull/45187>

执行摘要

- 一句话: 新增 nvfp4_4over6 KV cache dtype, 按重建误差优选 scale 提升量化精度
- 推荐动作: 值得精读, 重点看两点: 一是 csrc/libtorch_stable/nvfp4_kv_cache_kernels.cu 中误差引导的 scale 选择实现 (软件模拟 e2m1 舍入做重建误差评估, 再以 SSE 为准则择优), 这是将「量化质量」转化为可在 kernel 内计算的代价函数的典型做法; 二是 review 中从独立 algo 配置收敛为 dtype 的设计决策, 对 vLLM 这类配置面繁多的项目很有借鉴意义。若团队在 Blackwell 上使用 NVFP4 KV cache 且对精度敏感, 可评测 nvfp4_4over6 在自身负载下的精度 - 吞吐权衡。

功能与动机

PR body 明确指出: "For each 16-element NVFP4 group, nvfp4_4over6 evaluates scales derived from $\max / 6$ and $\max / 4$, then stores the candidate with lower reconstruction error." 意图是在不改变 NVFP4 物理存储的前提下, 通过 store 阶段的 scale 选择降低量化误差, 换取接近免费的精度提升。端到端评测显示 AIME25 从 64.12% 提升到 64.71%, LCB pass@1 从 52.42% 提升到 53.74%, 为量化 KV cache 的精度 - 吞吐权衡提供了新选项。

实现拆解

本 PR 的实现分 5 步拆解:

1. 注册新 dtype 到配置与工具链: vllm/config/cache.py 的 KV_CACHE_DTYPE 增加 nvfp4_4over6 并补充 docstring; vllm/utils/torch_utils.py 的 STR_DTYPE_TO_TORCH_DTYPE 映射到 torch.uint8, 将 is_quantized_kv_cache 与 create_kv_caches_with_random_flash 中 == "nvfp4" 放宽为 startswith("nvfp4"); vllm/v1/kv_cache_interface.py 的 get_kv_quant_mode 同样改为 startswith, 使新变体自动归入 KVQuantMode.NVFP4; vllm/config/vllm.py 的 validate_nvfp4_kv_cache_with_mla 同步放宽, 保证 MLA 模型的拒绝逻辑覆盖新 dtype。
2. FlashInfer backend 接入: vllm/v1/attention/backends/flashinfer.py 中 FlashInferBackend.supported_kv_cache_dtypes 增加 nvfp4_4over6; get_kv_cache_shape、get_dtype_for_flashinfer、supports_kv_cache_dtype、get_q_data_type 全部从相等判断改为 startswith("nvfp4")。关键设计是 FlashInferImpl 内部将 cache_dtype (用户配置) 与 kv_cache_dtype (传给 FlashInfer 的存储类型) 分离: 用户传 nvfp4_4over6 时, kv_cache_dtype 统一归一化为 "nvfp4", 因为 scale search 只影响 store kernel, 读取路径仍按标准 NVFP4 布局解析; do_kv_cache_update 则改用

`self.cache_dtype` 将变体信息传给写缓存内核。

3. CUDA kernel 实现 scale search: `csrc/libtorch_stable/nvfp4_kv_cache_kernels.cu` 新增 `NVFP4KVScaleSearch` 枚举 (`DEFAULT / FOUR_OVER_SIX`) 作为 `reshape_and_cache_nvfp4_kernel` 的模板参数; 新增 `round_to_nearest_e2m1` (模拟 e2m1 舍入的软件实现)、`nvfp4_candidate_scale` (由 `vecMax / denominator` 计算候选 FP8 scale 及其逆缩放)、`nvfp4_reconstruction_error` (对量化 - 反量化残差求平方和, 跨 lane 归约)、`cvt_warp_fp16_to_fp4_4over6` (并行计算两组候选并选择误差更小者)。host 侧 `reshape_and_cache_nvfp4_dispatch` 增加 `kv_cache_dtype` 参数, 按字符串分支实例化两个模板版本; `csrc/libtorch_stable/cache_kernels.cu` 的 `reshape_and_cache_flash` 入口将 `nvfp4_4over6` 转发进 NVFP4 分派路径。
4. 测试配套: `tests/utils/test_torch_utils.py` 新增 `test_nvfp4_4over6_cache_dtype`, 验证 `CacheConfig(cache_dtype="nvfp4_4over6")` 的 dtype 映射、量化判定与 KV 量化模式; `tests/kernels/attention/test_cache.py` 新增 `test_nvfp4_4over6_selects_lower_error_scale`, 构造 `[6.0, 4.5 * 15]` 的极端分布, 断言 4over6 反量化 MSE 严格低于默认路径; `tests/kernels/attention/test_attention_selector.py` 补充新 dtype 的覆盖。
5. 方案演进 (重要): 最初的提交以环境变量 `VLLM_NVFP4_KV_QUANT_ALGO` 加独立 `kv_cache_quant_algo` 参数承载该能力, 经 reviewer 讨论后整体折叠为新的 `kv_cache_dtype`, 避免新增一套并行的配置面, 最终提交为 "Expose NVFP4 4-over-6 as a cache dtype"。

关键文件:

- `csrc/libtorch_stable/nvfp4_kv_cache_kernels.cu` (模块 量化内核; 类别 other; 类型 core-logic; 符号 `NVFP4KVScaleSearch`, `round_to_nearest_e2m1`, `nvfp4_candidate_scale`, `nvfp4_reconstruction_error`): 核心实现文件: 新增 `NVFP4KVScaleSearch` 模板参数、`round_to_nearest_e2m1` 软件舍入、`nvfp4_candidate_scale`、`nvfp4_reconstruction_error`、`cvt_warp_fp16_to_fp4_4over6` 等关键函数, host 侧按 `kv_cache_dtype` 分支实例化两套 kernel。
- `vllm/v1/attention/backends/flashinfer.py` (模块 注意力后端; 类别 source; 类型 core-logic; 符号 `FlashInferBackend.supported_kv_cache_dtypes`, `FlashInferBackend.get_kv_cache_shape`, `FlashInferBackend.get_dtype_for_flashinfer`, `FlashInferBackend.supports_kv_cache_dtype`): `FlashInfer` backend 主路径: 注册 `nvfp4_4over6` 支持, 并将用户配置的 `cache_dtype` 与传给 `FlashInfer` 的存储 dtype 分离, 统一归一化为 `nvfp4` 以便复用读取路径。
- `vllm/config/cache.py` (模块 缓存配置; 类别 source; 类型 core-logic; 符号 `KV_CACHE_DTYPE`): `KV cache dtype` 定义的源头: 在 `KV_CACHE_DTYPE Literal` 中新增 `nvfp4_4over6` 并补充算法说明, 是所有配置校验和文档生成的入口。
- `vllm/utils/torch_utils.py` (模块 类型工具; 类别 source; 类型 core-logic; 符号 `STR_DTYPE_TO_TORCH_DTYPE`, `is_quantized_kv_cache`, `create_kv_caches_with_random_flash`): dtype 映射与量化判定的公共工具: `STR_DTYPE_TO_TORCH_DTYPE` 将 `nvfp4_4over6` 映射为 `uint8`, `is_quantized_kv_cache` 和 `create_kv_caches_with_random_flash` 的 `startswith` 放宽保证所有 NVFP4 变体走同一路径。

- tests/kernels/attention/test_cache.py (模块 内核测试; 类别 test; 类型 test-coverage; 符号 test_nvfp4_4over6_selects_lower_error_scale, quantize_and_dequantize) : 核心行为测试: 构造 [6.0, 4.5*15] 的极端分布验证 4over6 能选择误差更小的 scale, 直接证明 kernel 择优逻辑有效。
- vllm/model_executor/layers/attention/attention.py (模块 注意力层; 类别 source; 类型 data-contract; 符号 Attention.init, Attention.forward) : 数据契约调整: query 量化初始化与 forward 中的 kv_cache_dtype 断言从相等判断放宽为 startswith, 确保 nvfp4_4over6 触发 FP8 query 量化。
- csrch/libtorch_stable/cache_kernels.cu (模块 内核入口; 类别 other; 类型 core-logic; 符号 reshape_and_cache_flash) : 入口转发: reshape_and_cache_flash 将 nvfp4_4over6 识别为 NVFP4 路径, 并把 kv_cache_dtype 传给 nvfp4_dispatch 用于选择 kernel 模板。
- vllm/config/vllm.py (模块 配置校验; 类别 source; 类型 core-logic; 符号 validate_nvfp4_kv_cache_with_mla) : 配置校验: validate_nvfp4_kv_cache_with_mla 放宽为 startswith, 保证 MLA 模型 (如 DeepSeek) 使用任何 NVFP4 变体都会被正确拒绝。
- vllm/v1/kv_cache_interface.py (模块 缓存接口; 类别 source; 类型 core-logic; 符号 get_kv_quant_mode) : KV 量化模式解析: get_kv_quant_mode 放宽为 startswith, 使 nvfp4_4over6 正确归类到 KVQuantMode.NVFP4, 后续所有基于 KVQuantMode 的分支自动生效。
- tests/utils/_test_torch_utils.py (模块 配置测试; 类别 test; 类型 test-coverage; 符号 test_nvfp4_4over6_cache_dtype) : 配置链测试: 验证 CacheConfig 能接受 nvfp4_4over6, 并正确映射 torch.uint8、判定为量化 KV cache、解析为 KVQuantMode.NVFP4。
- tests/kernels/attention/test_attention_selector.py (模块 选择器测试; 类别 test; 类型 test-coverage) : 补充新 dtype 在注意力后端选择器中的覆盖, 防止 nvfp4_4over6 在选择阶段被遗漏。

关键符号: cvt_warp_fp16_to_fp4_4over6, nvfp4_reconstruction_error, nvfp4_candidate_scale, round_to_nearest_e2m1, reshape_and_cache_nvfp4_kernel, reshape_and_cache_nvfp4_dispatch, reshape_and_cache_flash, get_kv_quant_mode, is_quantized_kv_cache, validate_nvfp4_kv_cache_with_mla, FlashInferBackend.supports_kv_cache_dtype, FlashInferImpl.do_kv_cache_update, test_nvfp4_4over6_selects_lower_error_scale, test_nvfp4_4over6_cache_dtype

关键源码片段

csrch/libtorch_stable/nvfp4_kv_cache_kernels.cu

核心实现文件: 新增 NVFP4KVScaleSearch 模板参数、round_to_nearest_e2m1 软件舍入、nvfp4_candidate_scale、nvfp4_reconstruction_error、cvt_warp_fp16_to_fp4_4over6 等关键函数, host 侧按 kv_cache_dtype 分支实例化两套 kernel。

```
// 对 16 元素 NVFP4 分组执行 4-over-6 scale search:
// 并行计算 max/6 与 max/4 两个候选 scale, 分别做完整的
// 量化 - 反量化重建, 选择平方误差更小的候选写入缓存。
```

```

template <class Type, int CVT_FP4_NUM_THREADS_PER_SF>
__device__ __forceinline__ fp4_packed_t cvt_warp_fp16_to_fp4_4over6(
    PackedVec<Type, CVT_FP4_PACK16>& vec, float SFScaleVal, uint8_t* SFout) {
    // 第一步：求分组内绝对值最大值，双 lane 配置下用 shuffle 归约
    auto localMax = __habs2(vec.elts[0]);
#pragma unroll
    for (int i = 1; i < CVT_FP4ELTS_PER_THREAD / 2; i++) {
        localMax = __hmax2(localMax, __habs2(vec.elts[i]));
    }
    if constexpr (CVT_FP4_NUM_THREADS_PER_SF == 2) {
        localMax = __hmax2(__shfl_xor_sync(0xffffffffu, localMax, 1), localMax);
    }
    const float vecMax = float(__hmax(localMax.x, localMax.y));

    // 第二步：由 max/6 与 max/4 推导两个候选 FP8 scale
    uint8_t sf6, sf4;
    float outputScale6, outputScale4;
    nvfp4_candidate_scale(SFScaleVal, vecMax, 6.0f, &sf6, &outputScale6);
    nvfp4_candidate_scale(SFScaleVal, vecMax, 4.0f, &sf4, &outputScale4);

    // 第三步：用重建误差（残差平方和）挑选更优候选，
    // SFout 为空时跳过 scale 写入（非 lane0 线程）
    const float err6 =
        nvfp4_reconstruction_error<Type, CVT_FP4_NUM_THREADS_PER_SF>(
            vec, outputScale6);
    const float err4 =
        nvfp4_reconstruction_error<Type, CVT_FP4_NUM_THREADS_PER_SF>(
            vec, outputScale4);
    const bool use4 = err4 < err6;
    const float outputScale = use4 ? outputScale4 : outputScale6;
    if (SFout) *SFout = use4 ? sf4 : sf6;

    // 第四步：按选定 scale 完成实际量化打包（与默认路径共用 pack_fp4）
    float2 fp2Vals[CVT_FP4ELTS_PER_THREAD / 2];
#pragma unroll
    for (int i = 0; i < CVT_FP4ELTS_PER_THREAD / 2; i++) {
        fp2Vals[i] = cast_to_float2(vec.elts[i]);
        fp2Vals[i].x *= outputScale;
        fp2Vals[i].y *= outputScale;
    }
    return pack_fp4(fp2Vals);
}

// 重建误差计算：对量化结果做反量化后与原始值求残差平方和，
// 用于在多个候选 scale 之间做择优，是 4-over-6 的核心判据。
template <class Type, int CVT_FP4_NUM_THREADS_PER_SF>
__device__ __forceinline__ float nvfp4_reconstruction_error(
    PackedVec<Type, CVT_FP4_PACK16>& vec, float outputScale) {
    float error = 0.0f;

```

```

const float dequantScale =
    outputScale != 0.0f ? reciprocal_approximate_ftz(outputScale) : 0.0f;
#pragma unroll
for (int i = 0; i < CVT_FP4_ELTS_PER_THREAD / 2; i++) {
    float2 vals = cast_to_float2(vec.elts[i]);
    if (outputScale == 0.0f) {
        // scale 为 0 时整组归零，误差即原始值平方
        error += vals.x * vals.x + vals.y * vals.y;
    } else {
        // 用软件 e2m1 舍入模拟量化，再反量化求残差
        float qx = round_to_nearest_e2m1(vals.x * outputScale);
        float qy = round_to_nearest_e2m1(vals.y * outputScale);
        float dx = qx * dequantScale - vals.x;
        float dy = qy * dequantScale - vals.y;
        error += dx * dx + dy * dy;
    }
}
// 双 lane 共担一个 16 元素分组时，跨 lane 归并误差
if constexpr (CVT_FP4_NUM_THREADS_PER_SF == 2) {
    error += __shfl_xor_sync(0xffffffffu, error, 1);
}
return error;
}

```

vllm/v1/attention/backends/flashinfer.py

FlashInfer backend 主路径：注册 nvfp4_4over6 支持，并将用户配置的 cache_dtype 与传给 FlashInfer 的存储 dtype 分离，统一归一化为 nvfp4 以便复用读取路径。

```

# FlashInferImpl 初始化中的关键设计：
# 把用户配置的 cache_dtype 与传给 FlashInfer 的存储 dtype 解耦。
# nvfp4_4over6 只是 store 阶段的 scale 选择策略，
# FlashInfer 读取路径按标准 nvfp4 布局解析，无需新增读路径。
if self.kv_cache_spec.kv_quant_mode != KVQuantMode.NONE:
    self.cache_dtype = self.cache_config.cache_dtype
    # 注意：kv_cache_spec.dtype 是存储 dtype，与 op dtype 可能不同
    # (uint8 vs fp8_e4m3)，所以这里用 cache_config 的值
    self.is_kvcache_nvfp4 = self.cache_dtype.startswith("nvfp4")
    if self.is_kvcache_nvfp4:
        if (force_use_trtllm_attention() is False
            or not supports_trtllm_attention(is_prefill=True)
            or not supports_trtllm_attention(is_prefill=False)):
            raise ValueError(
                f"--kv-cache-dtype {self.cache_dtype} requires the "
                "SM100 trtllm-gen FlashInfer path."
            )
        # scale search 只影响 store kernel，读取统一按 nvfp4
        self.kv_cache_dtype = "nvfp4"
    else:
        self.kv_cache_dtype = FlashInferBackend.get_dtype_for_flashinfer(

```

```

        self.cache_dtype
    )
else:
    self.cache_dtype = "auto"
    self.is_kvcache_nvfp4 = False
    assert self.kv_cache_spec.dtype == self.model_config.dtype
    self.kv_cache_dtype = self.kv_cache_spec.dtype

```

tests/kernels/attention/test_cache.py

核心行为测试：构造 [6.0, 4.5*15] 的极端分布验证 4over6 能选择误差更小的 scale，直接证明 kernel 择优逻辑有效。

```

@torch.inference_mode()
def test_nvfp4_4over6_selects_lower_error_scale(
    kv_cache_factory_flashinfer,
) -> None:
    # 仅在 Blackwell 上运行 NVFP4 测试
    if not current_platform.has_device_capability(100):
        pytest.skip("NVFP4 requires compute capability >= 10.0 (Blackwell).")

    device = CUDA_DEVICES[0]
    block_size, head_size = 16, 64
    # 构造极端分布：一个 6.0 加上 15 个 4.5。
    # 默认 max/6 = 1.0 会把 4.5 全部截断到 1.0，量化误差大；
    # 4-over-6 同时考虑 max/4 = 1.5，能更好覆盖 4.5 的分布。
    source = torch.tensor(
        [6.0] + [4.5] * 15,
        dtype=torch.bfloat16,
        device=device,
    ).repeat(1, 1, head_size // 16)
    slot_mapping = torch.zeros(1, dtype=torch.long, device=device)
    scale = torch.ones(1, dtype=torch.float32, device=device)

    def quantize_and_dequantize(cache_dtype: str) -> torch.Tensor:
        # 用 flashinfer 工厂建缓存，走真实 reshape_and_cache_flash 路径
        key_caches, value_caches = kv_cache_factory_flashinfer(
            1, block_size, 1, 1, head_size, cache_dtype,
            source.dtype, device=device, cache_layout="NHD",
        )
        key_cache, value_cache = key_caches[0], value_caches[0]
        key_cache.zero_()
        value_cache.zero_()
        ops.reshape_and_cache_flash(
            source, source, key_cache, value_cache,
            slot_mapping, cache_dtype, scale, scale,
        )
        # 拆出 data 与 block_scales 后反量化，得到重建值
        from tests.kernels.quantization.nvfp4_utils import (
            dequant_nvfp4_kv_cache,

```

```

)
data, block_scales = nvfp4_split_data_scale(key_cache)
return dequant_nvfp4_kv_cache(
    data.permute(0, 2, 1, 3),
    block_scales.permute(0, 2, 1, 3),
    scale.item(), head_size, block_size,
)[0, 0, 0]

default = quantize_and_dequantize("nvfp4")
four_over_six = quantize_and_dequantize("nvfp4_4over6")
source_f32 = source[0, 0].float()

# 核心断言：4-over-6 的重建 MSE 必须严格低于默认 max/6 路径
default_mse = (default - source_f32).square().mean()
four_over_six_mse = (four_over_six - source_f32).square().mean()
assert four_over_six_mse < default_mse

```

评论区精华

核心讨论围绕「算法开关」还是「缓存 dtype」的抽象位置展开，并最终由维护者拍板：

- mgoin 在最终 APPROVE 中明确要求: "I think we should fold this in as a new kv_cache_dtype rather than as a separate 'algo' config", 并在另一条评论中指出 PR 描述与最终用法不一致（实际为 `--kv-cache-dtype nvfp4_4over6`），要求更新描述。
- pavanimajety 曾提出相反意见: "We should uniformly use kv-cache-quant-algo or kv-cache-dtype. IMO it makes sense to call it algo, because the NVFP4 representation is not changing.", 担心 kv_cache_dtype 变量承载算法语义会造成用户困惑。
- 作者在讨论中回应: kernel 层已把 "NVFP4 representation dtype" 与 "scale-search algorithm" 分离为两个参数，内部 kv_cache_dtype 保持 nvfp4，仅新增 kv_cache_quant_algo 分支；最终按 mgoin 意见统一收敛为 dtype 方案，环境变量方案被废弃。
- pavanimajety 还要求为算法差异补充 docstring 或技术报告链接，作者在后续提交中补充了默认 `max/6`、4-over-6 搜索和 K-only 变体的说明。
- 独立 algo 配置 vs 折叠为 kv_cache_dtype (design): 最终采用 mgoin 的方案：废弃环境变量与 algo 参数，整体收敛为新的 kv_cache_dtype = "nvfp4_4over6"，提交 "Expose NVFP4 4-over-6 as a cache dtype" 落地。
- algo 与 dtype 命名混淆的担忧 (design): mgoin 的 dtype 方案以更高优先级胜出，表示层统一为 nvfp4，算法差异由 dtype 名 nvfp4_4over6 承载；PR 最终描述也更新为 `--kv-cache-dtype nvfp4_4over6`。
- 补充算法差异的文档说明 (documentation): 作者在后续提交补充了 docstring，说明默认 max/6 路径、4-over-6 搜索（max/6 与 max/4 间按平方重建误差择优）及 K-only 变体。
- kernel 分支变量的语义疑问 (question): 作者回应已将 NVFP4 表示 dtype 与 scale-search 算法分离为独立参数，kernel 按 kv_cache_quant_algo 分支，kv_cache_dtype 保持 nvfp4；该中间设计后续仍被 dtype 方案取代。

- PR 描述过期 (documentation): 作者按要求更新 PR 描述, 最终用法统一为 `--kv-cache-dtype nvfp4_4over6`。

风险与影响

- 风险:
 1. 平台限制: `nvfp4_4over6` 仅在 Blackwell (SM100 及以上) 的 FlashInfer trtllm-gen 路径可用, `supports_kv_cache_dtype` 中已做能力门控, 但非 NVIDIA 平台与 SM90 路径会直接拒绝, 需确认用户侧报错信息足够清晰。
 2. 性能回退: 每个 16 元素分组需要额外计算一组候选 `scale` 和重建误差 (含 `__shfl_xor_sync` 归约), `store kernel` 开销上升, PR 报告 E2E 吞吐约下降 5% (LCB 场景 20.1k vs 21.0k tok/s), 长 `prefill` 场景可能更明显, 需关注真实负载下 CUDA graph 捕获行为。
 3. 字符串匹配放宽: 多处 `== "nvfp4"` 改为 `startswith("nvfp4")`, 如 `is_quantized_kv_cache`、`get_kv_quant_mode`、`validate_nvfp4_kv_cache_with_mla`, 若未来出现 `nvfp4_*` 前缀的其他非量化语义变量会被误匹配; 当前受控, 但属于潜在契约风险。
 4. 归一化掩盖变体: FlashInferImpl 强制把 `kv_cache_dtype` 覆盖为 `"nvfp4"`, 任何依赖该字段区分变体的下游逻辑都会失效, 目前仅 `do_kv_cache_update` 使用 `cache_dtype` 传递变体, 需警惕后续重构重新混用两字段。
 5. 测试覆盖有限: MSE 验证只覆盖 `[6.0, 4.5*15]` 单一分布, 未覆盖随机分布下 `4over6` 不劣于默认路径的性质 (理论上候选包含 `max/6` 且取误差小者, 必然不劣, 但 `kernel` 舍入模拟的边界未充分验证)。- 影响: 用户侧: 新增可选命令行参数 `--kv-cache-dtype nvfp4_4over6`, 在不增加显存占用、不改变缓存布局的前提下获得约 0.6~1.3 个百分点的精度提升 (AIME25 / LCB), 代价是吞吐最多下降 5%; 默认 `nvfp4` 行为完全不变, 属于低风险 opt-in 能力。系统侧: 仅影响 NVIDIA Blackwell (SM100 系列) + FlashInfer trtllm-gen 路径, 涉及 `store kernel` 的模板分支与 FlashInfer backend 的 `dtype` 归一化逻辑, 读取路径零改动。团队侧: 为 KV cache 量化建立了「`scale` 选择策略」这一新研究方向 (store-time search), 后续可扩展更多候选 `scale` (如 `max/4`、`K-only` 变体); PR 的 `dtype/algo` 抽象之争也为类似功能提供了设计先例。- 风险标记: 仅限 Blackwell SM100 路径, `store kernel` 吞吐回退约 5%, `startswith` 前缀匹配放宽, `dtype` 归一化掩盖变体信息, 测试分布覆盖有限

关联脉络

- PR #49347 [Online quantization] Add online MXFP4 quantization support: 同属 Blackwell 量化存储基础设施的建设 (MXFP4 在线量化), 涉及 `quantization` 配置注册与 CUDA kernel 分派路径, 与本 PR 的 `dtype` 扩展模式可对照。
- PR #49390 [Perf] Raise Blackwell CUDA graph capture default to 1024: 同为 Blackwell (SM100) 行为调整, NVFP4 KV cache 依赖 trtllm-gen CUDA graph 路径, 图捕获上限变化会影响 `nvfp4_4over6` 的端到端性能表现。
- PR #51417 [CI] Fix Batch Invariance (B200): B200 CI 环境修复, 与本 PR 依赖的 Blackwell 测试环境相关, 说明 NVFP4 测试对 B200 环境的稳定性依赖。