PR #35314 完整报告

sgl-project/sglang

Support deepseek v4 and kimi k3 on ssd

合并时间：2026-08-26 10:05

原文链接：http://prhub.com.cn/sgl-project/sglang/pull/35314

执行摘要

- 一句话：新增 SSD Expert Pack 加载，支持 DeepSeek-V4 与 Kimi-K3 超显存运行
- 推荐动作：值得精读。这是 SGLang 首次把 MoE 路由专家整体下沉到 SSD 的成熟实现，ExpertPackStore 的缓存 / 预取设计与完整 fail-closed 校验体系（identity 三元组、manifest 对账、stats 健康信号）有很高参考价值；review 中 BBuf 对 " 共享路径行为变化 "" 参数校验聚合 "" 校验成本与安全 " 的追问也值得学习。建议关注 pack 格式的版本演进、gguf.py 行为变化对非 expert-pack 模型的影响，以及 #36803 之后 deepseek_v4.py 的深层状态。

功能与动机

PR body 明确写道："It allows these MoE models to run when their complete weights are larger than GPU VRAM and host RAM combined"，目标是在单张 RTX 5090、32 GB VRAM 与 32 GB host RAM 的消费级硬件上运行权重以 TB 计的 MoE 模型。设计上强调 " 不剪枝、不合并、不跳过、不替换、不复量化专家，也不降低 Expert Top-K"，以保精度为第一原则；且 SSD 执行只在显式传入 --load-format expert_pack 时启用，避免污染标准加载路径。

实现拆解

实现按以下 5 步拆解：

1. 定义 Expert Pack 二进制格式与构建工具链（tools/expert_pack/） - format.py 定义 SGLANG-EXPERTPACK-v1 格式：magic、HEADER_STRUCT、ENTRY_STRUCT、expert-major 的 identity triplet 布局与 4096 对齐约束。 - build.py 从 GGUF 构建 pack：create_inventory 先对每个源张量做 payload 级 SHA-256 全量哈希，copy_range 按 16 MiB 块拷贝专家切片，generation 依据 model/source digest 派生代数。 - validate.py 提供离线全量校验（--full-pack-hash、--full-pack-entry-hashes），逐条目核对 index 覆盖、对象对齐、stride 与身份变换。 - prepare_deepseek_pack.py / prepare_kimi_pack.py / prepare_kimi_manifest.py 包装具体模型的构建流程；kimi_ggml.py 扫描 38 个 GGUF 分片并生成 SGLANG-KIMI-GGMLMOEPACK-ADAPTER-v1 格式的 adapter manifest。
2. 建立模型侧加载器与名称映射契约（python/sglang/srt/model_loader/） - deepseek4_gguf.py 的 build_deepseek4_checkpoint_name_map 基于 gguf 包的 DEEPSEEK2 名称表叠加 V4-only 显式映射（compressor、mHC、indexer、exp_probs_b），采用 fail-closed 策略：任何未映射或冲突张量直接抛错。 - expert_pack_loader.py 新增

ExpertPackModelLoader，校验 Kimi/DeepSeek 两种模型不变量（TP/DP/EP=1、disable_cuda_graph、disable_shared_experts_fusion），并构造 ExpertPackStore 或 KimiGGMLExpertPackStore。 - kimi_k3_gguf.py 的 kimi_k3_nonexpert_weights_iterator 按 manifest 分片流式产出非路由权重：逐分片核对 inventory、跳过路由专家、还原 A_log 变换（_kda_a_log_target_value）并处理 residual 双目标张量。

3. 实现运行时缓存与 I/O（python/sglang/srt/layers/moe/expert_pack.py） - ExpertPackStore 打开 pack 后立即校验 header（magic、版本、struct 尺寸、index_count、alignment、identity 标志）。 - _initialize_runtime_state 建立 staging slot、LFU/LRU 缓存、CUDA 事件同步与 stats 统计；direct_io 依赖 os.O_DIRECT，非 Linux 平台直接拒绝。 - 推理时按 router 选择的 (layer, expert) 键读 pack：多 split 并行预读、pinned 缓冲、异步 H2D，缓存按字节预算淘汰，fallback_count 与 io_errors 作为健康信号暴露在 stats 中。
4. 集成进 ServerArgs 与启动流程（server_args.py、arg_groups/expert_pack_hook.py、model_loader/expert_pack_runtime.py） - --load-format expert_pack 进入公共 choices；handle_expert_pack 在启动期聚合校验互斥选项并自动注入 --disable-cuda-graph 等依赖项。 - prepare_raw_kimi_server_args 对裸 GGUF 自动构建 / 复用 Kimi 资产（tokenizer 元数据、pack、manifest），使用 fcntl.flock 保证并发安全，产物按源文件指纹缓存到 ~/.cache/sglang-expert-pack/。 - DeepSeek 侧通过 _deepseek_model_config_from_gguf 从 GGUF 元数据重建 HF config，并推导 source_sha256 / model_identity_sha256 / config_sha256 完整性三元组。
5. 测试与示例配套 - test/registered/expert_pack/ 新增运行时测试（LFU/LRU victim 选择、zero staging 拒绝、split read 重构、全量校验 opt-in）与 MXFP4 kernel 参考实现测试（marlin nibble、scale perm）。 - examples/runtime/deepseek_v4/ 与 examples/runtime/kimi_k3/ 各提供一个一键 benchmark 脚本：自动定位仓库、检测 RTX 5090、构建 / 复用 pack、启动 server、流式生成 200 token 并汇报 prefill/decode 速率。

关键文件：

- python/sglang/srt/layers/moe/expert_pack.py（模块 专家缓存；类别 source；类型 core-logic；符号 _fixed_string, _sha256_file, ExpertPackHeader, read）：ExpertPackStore 运行时核心：pack header/entry 二进制解析、staging slot、LFU/LRU 缓存、direct I/O 与 stats 统计，931 行，是整条 SSD 推理路径的心脏。
- python/sglang/srt/model_loader/expert_pack_loader.py（模块 模型加载；类别 source；类型 data-contract；符号 _bf16_tensor, _compressor_component, _fused_compressor_name, deepseek4_nonexpert_weights_iterator）：ExpertPackModelLoader 是唯一入口，串起模型类型校验、运行时不变量检查、两种 store 构造与流式权重加载，并执行 MXFP4 扩展预热。
- python/sglang/srt/model_loader/kimi_k3_gguf.py（模块 权重映射；类别 source；类型 data-contract；符号 routed_expert_tensor, kimi_k3_checkpoint_targets, _runtime_name, _residual_target_value）：Kimi-K3 非路由权重流式加载器，靠 manifest 逐分片对账并处理 GGUF 到 SGLang 参数的精确映射，是数据契约最复杂的文件之一。

- python/sglang/srt/model_loader/expert_pack_runtime.py（模块 启动准备；类别 source；类型 data-contract；符号 cache_root, artifact_dir_for_source, _tokenizer_candidate, resolve_kimi_tokenizer）：启动期资产准备：从裸 GGUF 自动构建 / 复用 Kimi 资产、重建 DeepSeek HF config，并用 flock 保证并发安全，是 " 一行命令启动 " 的关键。
- python/sglang/srt/model_loader/deepseek4_gguf.py（模块 权重映射；类别 source；类型 data-contract；符号 routed_expert_tensor, _split_suffix, _v4_checkpoint_name, _candidate_score）：DeepSeek-V4 GGUF 张量名到 checkpoint 名的精确映射，覆盖 compressor、mHC、indexer 等 V4 独有张量，fail-closed 防止静默错载。
- python/sglang/srt/models/deepseek_v4.py（模块 模型定义；类别 source；类型 data-contract；符号 _is_fused_mhc_post_pre_enabled_xpu, _apply_gguf_grouped_wo_a, _prepare_deepseek_v4_weights, _fuse_deepseek_v4_wqkv_a_pair）：模型文件被修改以适配 expert_pack 权重加载路径，但 merge main 时意外回退了 #32166 的 XPU MHC dispatch，是 review 中发现的跨 PR 风险样本。
- tools/expert_pack/build.py（模块 构建工具；类别 source；类型 core-logic；符号 now, write_json_atomic, hash_range, copy_range）：pack 构建主入口：全量 inventory 哈希、分块拷贝、身份三元组与 generation 派生，是保证 pack 可审计性的核心工具。
- test/registered/expert_pack/test_expert_pack_runtime.py（模块 测试覆盖；类别 test；类型 test-coverage；符号 _sha256, _make_pack, _refresh_manifest_hash, TestExpertPackRuntime）：运行时核心行为测试：LFU/LRU victim 选择、zero staging 拒绝、split read 重构与全量校验 opt-in，是缓存策略正确性的主要保障。

关键符号：ExpertPackModelLoader.load_model, deepseek4_nonexpert_weights_iterator, kimi_k3_nonexpert_weights_iterator, build_deepseek4_checkpoint_name_map, kimi_k3_checkpoint_targets, ExpertPackStore._initialize_runtime_state, ensure_kimi_assets, prepare_raw_kimi_server_args, handle_expert_pack

关键源码片段

python/sglang/srt/layers/moe/expert_pack.py

ExpertPackStore 运行时核心：pack header/entry 二进制解析、staging slot、LFU/LRU 缓存、direct I/O 与 stats 统计，931 行，是整条 SSD 推理路径的心脏。

```
@dataclass
class _CacheSlot:
    # 每个 slot 记录专家身份、使用频率与 CUDA 事件，供 LFU/LRU 决策与就绪同步使用
    key: tuple[int, int] | None = None
    generation: int = 0
    frequency: int = 0
    last_use: torch.cuda.Event | None = None
    ready: torch.cuda.Event | None = None


def _initialize_runtime_state(
```

```
    store,
    *,
    cache_vram_mib: int,
    cache_vram_reserve_mib: int,
    stage_slots: int,
    read_splits: int,
    direct_io: bool,
    stats_flush_interval: int,
    stats_path: str | os.PathLike[str] | None,
) -> None:
    # 所有预算都必须为正；direct I/O 依赖 os.O_DIRECT，非 Linux 平台直接拒绝
    store.cache_vram_mib = int(cache_vram_mib)
    store.cache_vram_reserve_mib = int(cache_vram_reserve_mib)
    store.kernel_backend = "custom"
    store.stage_slot_count = int(stage_slots)
    store.read_splits = int(read_splits)
    store.direct_io = bool(direct_io)
    store.stats_flush_interval = int(stats_flush_interval)
    if (
        store.cache_vram_mib <= 0
        or store.cache_vram_reserve_mib <= 0
        or store.stage_slot_count <= 0
        or store.read_splits <= 0
    ):
        raise ValueError(
            "expert cache and staging budgets, and read splits, must be positive"
        )
    if store.direct_io and not hasattr(os, "O_DIRECT"):
        raise ValueError("expert-pack direct I/O is unavailable on this platform")

    # pack 文件全程只读打开；staging 用固定 slot 数做预读与 H2D 重叠
    open_flags = os.O_RDONLY | (os.O_DIRECT if store.direct_io else 0)
    store._fd = os.open(store.path, open_flags)
    store._lock = threading.RLock()
    store._cache = None
    store._cache_slots = []
    store._key_to_slot = {}
    store._key_frequency = {}
    store._lru = OrderedDict()
    store._staging = []
    store._stage_events = []
    store._stage_cursor = 0
    store._transfer_stream = None
    store._read_executor = None
    store._active_keys = set()
    store._route_calls_by_layer = [0] * store.header.num_layers
    store._route_tokens_by_layer = [0] * store.header.num_layers
    store.stats_path = Path(stats_path).resolve() if stats_path else None
    store._last_stats_flush_calls = 0
```

```
    # stats 覆盖 I/O 字节、缓存命中 / 驱逐、fallback 与 io_errors，
    # 是发现 pack 损坏等静默问题的唯一观测面
    store.stats = {
        "pack_path": str(store.path),
        "pack_entries": len(store.entries),
        "pack_reads": 0,
        "pack_read_bytes": 0,
        "pack_read_ns": 0,
        "read_splits": store.read_splits,
        "direct_io": store.direct_io,
        "cache_hits": 0,
        "cache_misses": 0,
        "cache_evictions": 0,
        "cache_policy": "reuse-lfu-lru-v2",
        "kernel_backend": "custom",
        "resident_experts": 0,
        "resident_bytes": 0,
        "h2d_bytes": 0,
        "cache_vram_reserve_mib": store.cache_vram_reserve_mib,
        "fallback_count": 0,
        "io_errors": 0,
    }
    atexit.register(store.close)
```

python/sglang/srt/model_loader/kimi_k3_gguf.py

Kimi-K3 非路由权重流式加载器，靠 manifest 逐分片对账并处理 GGUF 到 SGLang 参数的精确映射，是数据契约最复杂的文件之一。

```
def kimi_k3_nonexpert_weights_iterator(
    manifest_path: str | os.PathLike[str],
) -> Generator[tuple[str, torch.Tensor], None, None]:
    """按 manifest 分片流式产出非路由权重，跳过路由专家 payload。"""

    import gguf

    manifest_file = Path(manifest_path).resolve()
    manifest = json.loads(manifest_file.read_text(encoding="utf-8"))
    if manifest.get("format") != "SGLANG-KIMI-GGMLMOEPACK-ADAPTER-v1":
        raise ValueError("Kimi-K3 manifest format is unsupported")
    if not manifest.get("complete"):
        raise ValueError("Kimi-K3 manifest is incomplete")

    # 按 shard 索引聚合记录，便于逐分片核对 inventory
    records_by_shard: dict[int, list[dict]] = defaultdict(list)
    for record in manifest["source"]["tensors"]:
        records_by_shard[int(record["shard_index"])].append(record)

    emitted: set[str] = set()
    for shard in manifest["source"]["shards"]:
```

```
shard_index = int(shard["index"])
shard_path = Path(shard["path"]).resolve()
# 启动时校验分片存在性与字节数，防止源文件被替换
if not shard_path.is_file() or shard_path.stat().st_size != int(shard["size"]):
    raise FileNotFoundError(
        f"Kimi-K3 GGUF shard is missing or changed: {shard_path}"
    )
reader = gguf.GGUFReader(str(shard_path), mode="r")
tensors = {tensor.name: tensor for tensor in reader.tensors}
expected = {record["name"]: record for record in records_by_shard[shard_index]}
if set(tensors) != set(expected):
    raise ValueError(f"Kimi-K3 GGUF shard inventory changed: {shard_path}")

for source_name, tensor in tensors.items():
    record = expected[source_name]
    if tensor.tensor_type.name != record["dtype"]:
        raise ValueError(f"Kimi-K3 GGUF tensor type changed: {source_name}")
    # 路由专家完全跳过，由 ExpertPackStore 在推理期按需加载
    if routed_expert_tensor(source_name):
        continue

    targets = kimi_k3_checkpoint_targets(source_name)
    quantized = tensor.tensor_type.name not in ("F32", "F16", "BF16")
    raw = torch.tensor(tensor.data)
    for target_index, checkpoint_name in enumerate(targets):
        if source_name.endswith(".ssm_a"):
            # llama.cpp 在 GGUF 转换期做了 A_log -> -exp(A_log)，这里还原
            value = _kda_a_log_target_value(raw)
        elif len(targets) == 2:
            value = _residual_target_value(raw, target_index)
        else:
            value = raw
        runtime_name = _runtime_name(checkpoint_name, quantized)
        if runtime_name in emitted:
            raise ValueError(
                f"duplicate Kimi-K3 target parameter: {runtime_name}"
            )
        if quantized:
            # 先发 qweight_type，量化方法必须先知道类型再收到原始 qweight
            type_name = runtime_name.removesuffix("qweight") + "qweight_type"
            if type_name in emitted:
                raise ValueError(
                    f"duplicate Kimi-K3 target parameter: {type_name}"
                )
            emitted.add(type_name)
            yield type_name, torch.tensor(
                int(tensor.tensor_type), dtype=torch.uint8
            )
        emitted.add(runtime_name)
```

```
            yield runtime_name, value
```

评论区精华

Review 核心交锋集中在 6 个线程：

1. Kimi-K3 官方 checkpoint 的嵌套 config 校验：BBuf 指出官方 moonshotai/Kimi-K3 顶层 model_type 是 kimi_k3，kimi_linear 只出现在嵌套 text_config，当前校验会给出误导性错误。beyondHJM 澄清该 loader 只面向去 vision 后扁平化的量化 GGUF，官方 checkpoint 走 auto/safetensors 路径，不会触发该分支。
2. attn_residual SM120 门控修复应拆分：BBuf 建议把 major >= 10 的修复拆成独立 PR 并补充日志。beyondHJM 改为 major >= 10 and major != 12 并创建 #35361。
3. gguf.py shard 排序修复影响所有 GGUF 模型：BBuf 指出旧的 pass-through 行为被硬失败替代，QKV 只含 {q,k} 的融合层会直接报错，建议拆分并回归验证。beyondHJM 在多模型（Qwen、Llama 族）上验证无输出与性能差异后保留在原 PR。
4. --load-format expert_pack 参数校验聚合：BBuf 认为该 flag 单独无法工作，建议从公共 choices 移除或在 __post_init__ 聚合校验。beyondHJM 采用后者：启动时自动准备资产并一次性聚合报错。
5. verify_pack_sha256 默认值不一致：BBuf 指出 Kimi 默认 False、DeepSeek 默认 True 的不一致，且 pack 损坏会表现为错误专家。beyondHJM 将全量哈希改为离线校验，启动时执行结构校验（manifest、pack 大小、header、identity digests、entries、对象布局、索引 SHA-256、采样 payload）。
6. 顶层目录风格与意外回退：merrymercy 要求 tools/expert_pack 不能作为顶层目录；cyxlily 指出 merge main 时意外回退了 #32166 的 XPU MHC dispatch，需 #36803 恢复。

- Kimi-K3 官方 checkpoint 的嵌套 text_config 校验 (correctness): beyondHJM 澄清 loader 只面向去 vision 后扁平化的量化 GGUF，官方 checkpoint 走 auto/safetensors 路径不会触发该分支；PR body 补充说明仅支持文本版 Kimi-K3。
- attn_residual SM120 门控修复应拆分 (design): beyondHJM 改为 major >= 10 and major != 12 并拆分为 #35361 独立落地，但作者自述拆分 PR 的 CI 尚未通过。
- gguf.py shard 排序修复影响所有 GGUF 模型 (correctness): beyondHJM 在 Qwen、Llama 族多模型上验证无输出与性能差异，认为修复与 expert_pack 强相关而保留在本 PR。
- --load-format expert_pack 的启动期参数校验 (design): beyondHJM 采用集中处理：启动时自动准备 / 解析资产、自动禁用 CUDA graph 与 shared-expert fusion，并以单个聚合错误汇报所有不兼容项。
- verify_pack_sha256 默认值不一致与全量校验成本 (security): beyondHJM 将全量哈希移出 serving 配置，启动执行结构校验与采样 payload 校验，全量 sha256 保留为离线工具；并报告两个模型 20-token 冒烟均无 fallback 与 I/O 错误。
- tools/expert_pack 顶层目录风格 (style): 材料中未见最终迁移结果，PR 已关闭，需在合入后确认实际目录位置。
- XPU MHC dispatch 被意外回退 (correctness): 已通过 #36803 恢复；该事件成为大 PR 高频 merge 的警示样本。

风险与影响

- 风险：
  1. 共享 GGUF 路径回归风险：python/sglang/srt/layers/quantization/gguf.py 新增的 _ordered_gguf_shard_ids 把之前的 pass-through 改为硬失败，影响所有使用 GGUF 加载的模型。作者已在 Qwen/Llama 族验证，但审查中 BBuf 仍建议拆分独立回归。
  2. XPU 路径意外回退：python/sglang/srt/models/deepseek_v4.py 在 merge main 时回退了 #32166 的 XPU MHC dispatch，说明大 PR 高频 merge 存在 silent revert 风险，已由 #36803 恢复。
  3. 完整性校验成本与盲区：Kimi 路径启动时只做结构校验与 6 个 payload 抽样，全量 Sha256 校验被移出 serving 配置；pack 静默损坏可能仅在推理期以 fallback_count 升高形式暴露。
  4. 平台强绑定：direct_io 依赖 os.O_DIRECT，MXFP4 kernel 与 TileLang indexer 面向 SM120（RTX 5090）验证；非支持平台或其它量化变体 GGUF 不在宣称范围内。
  5. 精度验证缺失：BBuf 明确指出 K3 数据来自第三方 abliterated Q2_K GGUF，" matched exactly" 是自比；PR 无 GSM8K 等标准精度数据支撑。
  6. 存储与性能不确定性：Kimi 38 分片加 pack 共约 1.814 TiB，要求 4 TB SSD；decode 速率高度依赖专家缓存命中率与 SSD 带宽，多并发场景未验证。 - 影响：用户侧：需自备 RTX 5090、NVMe SSD（建议 4 TB）与 32 GB host RAM，首次启动需 8-45 分钟构建 pack；仅支持 PR 列明的两个量化 GGUF 输入（DeepSeek MXFP4 单文件、Kimi Q2_K/Q3_K 38 分片），其它量化变体不在支持范围。系统侧：新增约 8300 行代码、46 个文件，新增一种公共 load format 与独立工具链目录，加载器、量化层、CUDA kernel 与模型文件多处联动。团队侧：格式契约（header/index/manifest/inventory）的版本演进与共享 gguf.py 行为变化需要持续维护；CI 需要覆盖 pack 构建、运行时缓存与 MXFP4 kernel 的回归测试。 - 风险标记：共享 GGUF 路径行为变化，缺少标准精度验证（GSM8K），XPU dispatch 意外回退，平台 / 硬件强绑定（O_DIRECT、SM120），大数据契约维护成本高

关联脉络

- PR #35361 attn_residual SM120 TMA gate 独立修复：review 中 BBuf 建议拆分 attn_residual 后端选择修复，作者据此创建独立 PR（仅改动约四行）。
- PR #36803 恢复 XPU MHC dispatch: Issue 评论中 cyxlily 指出本 PR merge main 时意外回退了该功能对应代码，需此 PR 恢复。
- PR #32166 XPU MHC dispatch 原始实现：本 PR 意外回退的源头 PR，说明大 PR 与 main 高频合并存在 silent revert 风险。