PR #34274 完整报告

sgl-project/sglang

[kernel] Content-addressed JIT build cache, generated from our own ninja

合并时间：2026-08-14 13:41

原文链接：http://prhub.com.cn/sgl-project/sglang/pull/34274

执行摘要

- 一句话：JIT 缓存改内容寻址双 key，修复陈旧复用
- 推荐动作：值得精读。设计上有三个可借鉴点：双 key 分离“编译前可知 / 编译后可知”信息；leaf 名 = 依赖状态哈希的内容寻址自验证，无共享 manifest；快速路径无锁 + 锁内重查 + 原子 rename 发布。测试上 41 个用例专门 pin 静默失败模式（如 test_no_unordered_container_reaches_the_key）。建议阅读重点：cache.py 的 key 推导与缓存布局、ninja.py 的 generate 与 depfile 处理。

功能与动机

PR 的 Problem 部分给出量化依据：旧实现 _local_jit_source_hash 用两个正则手工模拟预处理器，CUTLASS kernel 遍历只覆盖 8 个文件，而编译器实际读取 1112 个；30 个引号 include 无法解析（CUTLASS 走 -I），648 个尖括号 include 被跳过（含 92 次 <tvm/ffi/container/tensor.h>）。每个漏掉的 include 都隐藏其整棵头文件子树，因此“升级 flashinfer 或编辑 CUTLASS 头文件会复用陈旧二进制”；编译器与包版本也不在 key 中，且 load_jit 总是传入显式 build_directory，导致 tvm-ffi 自带的 flag 感知哈希从未运行，extra_cuda_cflags 只有在调用方手工编码进模块名时才生效（如 activation vs rounded_activation）。ROCm 侧更严重：tvm-ffi 的 HIP 命令声明了 depfile = $out.d 却从不生成，头文件依赖完全缺失。

实现拆解

实现按 5 步拆解：

1. 单文件拆包为 compile/ 包：python/sglang/kernels/jit/utils/compile.py（删除 341 行）按依赖顺序拆成 compile/ 包——paths.py（_resolve_kernel_path、KERNEL_PATH、默认 flags）、cpp_args.py（CPPArgList、make_cpp_args）、spec.py（不可变 BuildSpec 与 TranslationUnit，作为 cache 与 ninja 之间的唯一交接对象）、toolchain.py（把 tvm-ffi 隐式 flags 显式化）、ninja.py、cache.py、loader.py。arch.py 不动，__init__.py 做重导出兼容，paths.py 依赖 RFC #29630 确立的目录结构。
2. 双 key 内容寻址缓存：cache.py 用 build_key 覆盖编译前已知的一切（生成的 ninja 文本全文、wrapper 源码、直接源码内容、目标 arch、双编译器指纹、依赖包版本），选择 build-<build_key>/ 目录；deps_key 从构建后留下的 depfile 读回传递闭包，选择 deps-<deps_key>/ leaf。每个 leaf 携带自己的 sgl_deps.json，命中条件 = 重新哈希当前文件能复现 leaf 名，数据自验证而非被信任，无共享可变 manifest。_DepEntry 与

_anchor_roots 把路径归一化为 anchor token + 相对路径，第二个 clone 不同位置、全新 mtime 也能复用缓存。

3. 自持有 build.ninja：ninja.py 的 generate 直接生成完整构建描述，缓存 key 取文件全文而不是近似值；刻意不发射 deps = gcc 以保留 .d 文件；设备规则显式 -MD -MF，修复 HIP 不写 depfile（MI350X 实测 depfile 从 NONE 变为 21307 字节）。路径处理先 shlex.quote 再转义 $，解决含空格目录被拆成多参数的问题。
4. 并发构建与原子发布：loader.py 的 _build_lock 按 module variant 用文件锁串行化冷构建，抢到锁后重查缓存，8 rank 并发只编译一次；构建在 .staging-<uuid> 中进行，完成后原子 rename 发布，快速路径无锁读；损坏 leaf 在第二次加载失败后被替换。load_jit 用 torch.compiler.disable 隔离 Dynamo 追踪。
5. 配套改动：load_jit 删除 build_directory 与 external_cpp_files / external_cuda_files（绝对路径等价表达）；environ.py 注册 SGLANG_JIT_CACHE_DIR / SGLANG_JIT_CACHE_DEBUG / SGLANG_JIT_CACHE_KEEP；新增 test/registered/kernels/test_jit_cache.py 41 个 CPU-only 用例；适配 trtllm-gen 的 runtime 组装源码与 inventory 检查；CI 超时经历 30→45→60 后再 revert 回 30。

关键文件：

- python/sglang/kernels/jit/utils/compile/cache.py（模块 JIT 缓存；类别 source；类型 core-logic；符号 _DepEntry, _package_dir, _anchor_roots, _normalize_path）：新增 522 行，是双 key 内容寻址缓存的核心：build_key / deps_key 推导、anchor 归一化、leaf 自验证与原子发布。
- python/sglang/kernels/jit/utils/compile/ninja.py（模块 构建脚本；类别 source；类型 core-logic；符号 _escape, _arg, _quote_path_flags, generate）：新增 228 行，自持有 build.ninja 的生成与执行：depfile 显式 -MD -MF、路径 shell 引号、depfile 解析回读传递闭包。
- python/sglang/kernels/jit/utils/compile.py（模块 JIT 编译；类别 source；类型 file-removal；符号 _local_jit_source_hash, _resolve_kernel_path, CPPArgList, make_cpp_args）：删除 341 行的旧单文件实现，是本次拆包重构的起点；手写正则的 _local_jit_source_hash 正是陈旧缓存复用的根源。
- python/sglang/kernels/jit/utils/compile/loader.py（模块 加载器；类别 source；类型 core-logic；符号 load_jit, _build_lock, _load）：新增 200 行，load_jit 新入口：文件锁串行化并发构建、锁内重查、原子发布与损坏 leaf 替换。
- python/sglang/kernels/jit/utils/compile/spec.py（模块 构建规格；类别 source；类型 core-logic；符号 TranslationUnit, stem, BuildSpec, module_name）：新增 134 行，BuildSpec 作为 cache 与 ninja 之间的唯一交接对象，明确新字段必须同步进入 build_key。
- python/sglang/kernels/jit/utils/compile/toolchain.py（模块 工具链；类别 source；类型 configuration；符号 cuda_home, rocm_home, device_compiler_path, host_compiler_path）：新增 168 行，把 tvm-ffi 隐式提供的编译器、include、target flags 显式化并纳入 key；ROCm 用 gcnArchName 区分 gfx940/941/942。
- test/registered/kernels/test_jit_cache.py（模块 缓存测试；类别 test；类型 test-coverage；符号 _fresh_digests, _write, _spec, _build_key）：新增 532 行，41 个 CPU-only 用例覆盖 anchor 归一化、key 分离、leaf 自验证、ninja 生成与 depfile 解析，

pin 住静默失败模式。

- python/sglang/kernels/jit/utils/compile/cpp_args.py（模块 模板参数；类别 source；类型 dependency-wiring；符号 CPPArgList, str, make_cpp_args, _convert）：新增 50 行，从旧 compile.py 迁移 CPPArgList 与 make_cpp_args，负责把 Python 值渲染为 C++ 模板参数。
- python/sglang/kernels/jit/utils/compile/paths.py（模块 路径解析；类别 source；类型 dependency-wiring；符号 _resolve_kernel_path）：新增 32 行，以 importlib find_spec 定位内核源码树，提供 KERNEL_PATH 与默认 flags，是 anchor 归一化的基准。
- python/sglang/kernels/jit/utils/compile/__init__.py（模块 包入口；类别 source；类型 entrypoint）：新增 44 行，包重导出与兼容层，承接旧 compile.py 的公开符号，降低调用方迁移成本。
- python/sglang/srt/environ.py（模块 环境变量；类别 source；类型 configuration）：修改 +9，注册 SGLANG_JIT_CACHE_DIR / SGLANG_JIT_CACHE_DEBUG / SGLANG_JIT_CACHE_KEEP 三个环境变量声明。
- test/registered/kernels/test_kernel_inventory.py（模块 内核清单；类别 test；类型 test-coverage）：修改 +9，适配外部源码折叠进 cpp_files 后的 inventory 检查：新增对 runtime 组装源码的豁免与校验。

关键符号：load_jit, _load, _build_lock, _DepEntry, _anchor_roots, _normalize_path, _resolve_path, _file_digest, _hash_parts, BuildSpec, translation_units, generate, build, scan_dependencies, _parse_depfile, gpu_arch_name, target_flags, make_cpp_args

关键源码片段

python/sglang/kernels/jit/utils/compile/cache.py

新增 522 行，是双 key 内容寻址缓存的核心：build_key / deps_key 推导、anchor 归一化、leaf 自验证与原子发布。

```
# content-addressed JIT 构建缓存：key 推导、目录布局、原子发布。
# 每次 load_jit 只回答一个问题：是否存在一个已构建的 .so，与现在立刻构建
# 出来的产物保证一致？因为完整答案在首次编译前无法计算，这里拆成两个 key：
# build_key —— 编译前已知的一切（模块参数、调用方 flags、wrapper 导出、
# 编译目标、直接源文件内容），用来选择目录；
# deps_key —— 传递依赖闭包的内容，只有编译器能枚举，用来选择 leaf。

# 每个 leaf 携带自己的依赖列表，发布后不再修改。只有重新哈希当前文件
# 得到的名字与 leaf 名一致才算命中，因此记录的数据是自验证的：截断、
# 篡改或格式不同的列表都无法复现 leaf 名，也就不会污染本机缓存。
# 这里刻意不提供共享的可变 manifest，避免多写者合并问题。

class _DepEntry(msgspec.Struct, frozen=True, array_like=True):
    """一条依赖，按与安装位置无关的方式存储。

    root 是 anchor 锚点 token，对应当前环境解析；
    这样一份列表由某个 clone 写出后，另一个 clone 也能读懂。
    """
```

```
    root: str # anchor token，如 kernels / tvm_ffi / toolkit
    relpath: str # 相对 anchor 的路径
    digest: str # 文件内容摘要


# 锚点根按"最具体优先"排序，保证最长匹配胜出。候选包括：内核源码树、
# tvm_ffi、flashinfer、deep_gemm、nvidia 包、CUDA/ROCm 工具链根、
# site-packages 与系统 /usr。路径在写入时被解析成 anchor + 相对路径，
# 缓存命中时再按当前环境的 anchor 反解回真实路径——
# 这正是第二个 clone 哪怕位于不同路径、mtime 全新也能复用缓存的原因。
@cache_once
def _anchor_roots() -> Tuple[Tuple[str, pathlib.Path], ...]:
    candidates: List[Tuple[str, Optional[pathlib.Path]]] = [
        ("kernels", KERNEL_PATH),
        ("tvm_ffi", _package_dir("tvm_ffi")),
        ("pkg:flashinfer", _package_dir("flashinfer")),
        ("pkg:deep_gemm", _package_dir("deep_gemm")),
        ("pkg:nvidia", _package_dir("nvidia")),
        ("toolkit", toolchain.toolkit_home()),
        ("sitepkgs", pathlib.Path(sysconfig.get_paths()["purelib"])),
        ("sys", pathlib.Path("/usr")),
    ]
    # 返回前再做 resolve，确保映射稳定、不随调用位置漂移。
```

python/sglang/kernels/jit/utils/compile/toolchain.py

新增 168 行，把 tvm-ffi 隐式提供的编译器、include、target flags 显式化并纳入 key；ROCm 用 gcnArchName 区分 gfx940/941/942。

```
# 工具链解析：sglang 自己生成 build.ninja，不再走 tvm_ffi.cpp.load_inline，
# 因此 tvm-ffi 隐式提供的 flags 必须在这里显式声明——这正是关键：到达编译器的
# 每个 flag 都在一处可见、可哈希进 build_key，而不是藏在一个只能靠版本号
# 近似其默认行为的外部依赖里。仍然消费 tvm-ffi 的只有位置信息（头文件、
# 共享库）与 load_module。

@cache_once
def gpu_arch_name() -> str:
    """编译目标，按厂商自己的名字表达。

    ROCm 上是 gcnArchName（如 gfx942:sramecc+:xnack-），而不是
    CUDA 风格的 (major, minor)：后者会把 gfx940/gfx941/gfx942 都映射成
    同一个 9.4，但这三者是三个不同的编译目标。
    """
    if not is_hip_runtime():
        return get_jit_cuda_arch().target_name
    try:
        device = torch.cuda.current_device()
        return str(torch.cuda.get_device_properties(device).gcnArchName)
    except Exception:
        logger.warning("Cannot detect ROCm gcnArchName; the JIT cache target degrades.")
```

```
        return "unknown"


def target_flags() -> List[str]:
    """固定到当前 GPU 的设备编译 flags。

    直接由 sglang 已检测到的架构生成，而不是交给编译器驱动去探测：
    该值属于缓存 key 的一部分，必须在编译前决定，不能让编译器在
    构建时才重新发现。
    """
    if is_hip_runtime():
        return [f"--offload-arch={gpu_arch_name()}"]
    arch = get_jit_cuda_arch()
    target = f"{arch.major}{arch.minor}{arch.suffix}"
    return [f"-gencode=arch=compute_{target},code=sm_{target}"]
```

评论区精华

BBuf 的 review 提出三个缓存边界问题，全部在提交 74c8e381 中处理：

> BBuf（ninja.py）：These paths are Ninja-escaped but not shell-quoted. For example, -I/tmp/dir$ with$ spaces expands to -I/tmp/dir with spaces, so the compiler gets three args. Could we quote compiler/include/library paths and add a path-with-spaces test?
>
> BBuf（cache.py）：Could this mtime update be best-effort? A read-only cache hit raises here, and a concurrent prune between is_file() and utime() can raise FileNotFoundError. Catching OSError would let _load() handle the latter.
>
> BBuf（cache.py:518）：If an existing .so fails to load, the rebuild gets the same deps key. This branch then keeps the broken leaf, so every new process rebuilds it again. Can we quarantine or replace the leaf after the second load failure?

三个发现都被确认为真实问题：_arg 先 shlex.quote 再转义 $、_quote_path_flags 统一包裹 -I / -L 并补含空格路径测试；mtime 更新改为 best-effort 捕获 OSError；损坏 leaf 第二次加载失败后被替换。随后 BBuf APPROVED。

- 路径 shell 引号问题 (correctness): 已处理：新增 _arg 先 shlex.quote 再转义 $，_quote_path_flags 统一包裹 -I / -L 目录，并在测试中覆盖含空格路径。
- 缓存命中时 mtime 更新需 best-effort (correctness): 提交 74c8e381 已按 best-effort 处理，捕获 OSError。
- 损坏 leaf 的隔离与替换 (correctness): 已处理：第二次加载失败后替换损坏 leaf，避免反复重建。

风险与影响

- 风险：

1. 缓存 key 完备性成为长期维护约束：spec.py 明确要求“新增影响生成代码的字段必须同步进入 cache.compute_build_key”，未来扩展若遗漏会错误命中陈旧产物，这是本设计最主要的回归面。
2. 已知盲区 __has_include：依赖图开关基于文件存在性而非内容，靠编译器与包版本兜底（ccache 同款缺口）。
3. 共享 / 只读缓存边界：SGLANG_JIT_CACHE_DIR 指向只读挂载或并发 prune 时，mtime 更新等写路径虽已 best-effort，仍属行为边界；跨机器共享时 _anchor_roots 依赖 importlib.find_spec 与系统目录布局，不同发行版可能导致 key 不命中（安全失败）。
4. API 破坏：load_jit 删除 build_directory / external_cpp_files / external_cuda_files 对树外自定义内核是 breaking change；旧 compile.py 的直接导入路径被包重导出取代，依赖内部符号的插件需更新。
5. 测试局限：41 个用例均为 CPU-only，不覆盖真实 nvcc / hipcc 编译路径，CUDA / ROCm 行为依赖人工验证；CI 冷缓存下超时波动仍存在，历史上 30→45→60 再 revert 回 30。 - 影响：性能影响显著：CUDA 热加载 5.26 s → 0.02 s，ROCm 2.89 s → 0.08 s；touch 全部头文件不重建；编辑传递头文件后精确重建并点名文件；revert 编辑立即复用旧 leaf；8 rank 冷启动从 8 次编译降为 1 次；不同路径的第二个 clone 直接命中缓存。对用户部署，默认缓存位于 ~/.cache/sglang/jit，可指向持久挂载跨 CI job 共享。对开发体验，新内核作者不再需要手工把 flag 编码进模块名，wrapper 宏头文件显式包含避免侥幸编译。对团队维护，缓存逻辑从单文件 341 行拆为 7 模块包，错误模式从静默陈旧复用变为明确定位与自验证。 - 风险标记：核心路径变更：所有 load_jit 内核加载，缓存 key 完备性依赖新字段同步，并发共享缓存与只读挂载边界，已知 __has_include 盲区，CI 冷缓存超时反复调整

关联脉络

- PR #33997 Bump FlashInfer to 0.6.17 and remove Kimi K3 workarounds: 该 PR 移除了 trtllm_gen_moe.py——本分支 load_jit 的 build_directory 参数唯一调用方与 jit_module_name 唯一消费者，使本 PR 能顺势去掉 build_directory 逃生口并清理相关兼容逻辑。