PR #51911 完整报告

vllm-project/vllm

[CI] Add registry layer cache to x86 CPU image build

合并时间：2026-08-12 17:58

原文链接：http://prhub.com.cn/vllm-project/vllm/pull/51911

执行摘要

- 一句话：CPU CI 镜像构建接入 buildx 注册表缓存与 sccache
- 推荐动作：推荐负责 CI 与镜像构建的工程师精读本 PR。值得学习的点：① buildx type=registry 缓存与 BuildKit cache mount 的边界——cache mount 不被 registry 缓存导出，跨 builder 命中必须依赖 sccache 这类远端缓存；② docker-container builder 的网络隔离对 IMDS/S3 的影响及 network=host 的权衡；③ 缓存 key 的分支 /PR 回退设计（CACHE_FROM_BASE_BRANCH、CACHE_FROM_MAIN 去重与兜底）。后续可考虑将 prepare_cache_tags 抽象为多平台共享脚本，并补充 arm64 构建验证。

功能与动机

PR body 明确描述问题：CPU CI 镜像构建 “used classic docker build with no layer cache, so every build recompiled csrc/rust from scratch”，而 CUDA 镜像构建（image_build.sh）已经具备 registry 层缓存，CPU 路径是明显缺口。实现过程中又发现仅 registry 层缓存不够——rust 与 C++ 编译使用 BuildKit cache mount（ccache、cargo registry）作为对象存储，--cache-to type=registry 不会导出 cache mount，换一台构建机即失效，因此同步引入 S3 后端 sccache（USE_SCCACHE=1）保证跨 builder 命中。相关 PR #46711 只覆盖了 x86 CUDA release 镜像构建，本 PR 恰好补齐 CPU CI（非 release）镜像构建的空缺。

实现拆解

1. 缓存 key 解析（脚本层）：在 image_build_cpu.sh 移植 clean_docker_tag() 与 prepare_cache_tags()，按 BUILDKITE_PULL_REQUEST 与 BUILDKITE_BRANCH 区分 main 分支、feature 分支 push、PR 合入 main、PR 合入非 main 分支四种场景，得到 CACHE_TO / CACHE_FROM / CACHE_FROM_BASE_BRANCH / CACHE_FROM_MAIN，并对 cache-from ref 去重。
2. ECR 复用与登录：缓存写入 vllm-ci-test-cache / vllm-ci-postmerge-cache 私有 ECR，所有 tag 统一带 -x86_cpu 后缀；脚本新增 aws ecr get-login-password 私有 ECR 登录。
3. 构建命令迁移：docker build 换为 docker buildx build --push，--cache-to type=registry,mode=max，--cache-from 依次回退到 PR 专属缓存、base 分支缓存、main 缓存；创建 docker-container 驱动的 builder vllm-cpu-builder，并用 --driver-opt network=host 保证 sccache 可访问 EC2 实例元数据服务（IMDS）。
4. Dockerfile 阶段重构：新增 base 阶段安装公共包与可选 sccache；base-common 与 rust-build 以 FROM base 并行；为三个编译阶段注入 USE_SCCACHE、

SCCACHE_ENDPOINT、SCCACHE_BUCKET 等 ARG/ENV，rust 用 RUSTC_WRAPPER=sccache，triton-cpu 用 CMAKE_C/CXX_COMPILER_LAUNCHER=sccache，并让 rust-build 的 SCCACHE_SERVER_PORT=4227 避免与主构建阶段端口冲突。

5. 验证与配套：bash -n 语法检查、prepare_cache_tags() 四场景推演、本地 registry:2 干跑验证缓存 round-trip；真实 ECR 路径通过多轮 /ci run 在 Buildkite 上验证，期间的临时验证 commit 全部 revert。无运行时、测试或模型配套改动。

关键文件：

- .buildkite/image_build/image_build_cpu.sh（模块 构建脚本；类别 infra；类型 core-logic；符号 clean_docker_tag, prepare_cache_tags）：CPU CI 镜像构建入口脚本，从经典 docker build 切换为 buildx registry 缓存构建，新增 clean_docker_tag / prepare_cache_tags 缓存 key 解析、私有 ECR 登录与 sccache 参数传递，是本 PR 的核心控制逻辑。
- docker/Dockerfile.cpu（模块 镜像构建；类别 infra；类型 infrastructure）：CPU 镜像的 Dockerfile，新增公共 base 阶段、将 rust-build 与 base-common 并行化，并在三个编译阶段注入 sccache，是跨 builder 编译缓存能否命中的关键。

关键符号：clean_docker_tag, prepare_cache_tags

关键源码片段

.buildkite/image_build/image_build_cpu.sh

CPU CI 镜像构建入口脚本，从经典 docker build 切换为 buildx registry 缓存构建，新增 clean_docker_tag / prepare_cache_tags 缓存 key 解析、私有 ECR 登录与 sccache 参数传递，是本 PR 的核心控制逻辑。

```
# clean_docker_tag: 将任意字符串规整为合法 Docker tag,
# 非 [a-zA-Z0-9._-] 字符替换为下划线，并截断到 128 字符上限
clean_docker_tag() {
  local input="$1"
  echo "$input" | sed 's/[^a-zA-Z0-9._-]/_/g' | cut -c1-128
}

# prepare_cache_tags: 解析并设置 CACHE_TO / CACHE_FROM 等全局缓存引用,
# 复用 CUDA 镜像构建的 ECR 缓存仓库，所有 tag 加 -x86_cpu 后缀与 CUDA 隔离
prepare_cache_tags() {
  TEST_CACHE_ECR="936637512419.dkr.ecr.us-east-1.amazonaws.com/vllm-ci-test-cache"
  MAIN_CACHE_ECR="936637512419.dkr.ecr.us-east-1.amazonaws.com/vllm-ci-postmerge-
  cache"

  if [[ "${BUILDKITE_PULL_REQUEST:-false}" == "false" ]]; then
    if [[ "${BUILDKITE_BRANCH:-}" == "main" ]]; then
      # main 分支直接命中 postmerge 长期缓存
      cache="${MAIN_CACHE_ECR}:latest-x86_cpu"
    else
      # 非 main 分支 push 时用清理后的分支名作为缓存 key
      clean_branch=$(clean_docker_tag "${BUILDKITE_BRANCH:-unknown}")
```

```
    cache="${TEST_CACHE_ECR}:${clean_branch}-x86_cpu"
  fi
  CACHE_TO="$cache"
  CACHE_FROM="$cache"
  CACHE_FROM_BASE_BRANCH="$cache"
else
  # PR 构建：写入 PR 专属缓存，回退链为 base 分支 → main
  CACHE_TO="${TEST_CACHE_ECR}:pr-${BUILDKITE_PULL_REQUEST}-x86_cpu"
  CACHE_FROM="${TEST_CACHE_ECR}:pr-${BUILDKITE_PULL_REQUEST}-x86_cpu"
  if [[ "${BUILDKITE_PULL_REQUEST_BASE_BRANCH:-main}" == "main" ]]; then
    CACHE_FROM_BASE_BRANCH="${MAIN_CACHE_ECR}:latest-x86_cpu"
  else
    clean_base=$(clean_docker_tag "${BUILDKITE_PULL_REQUEST_BASE_BRANCH}")
    CACHE_FROM_BASE_BRANCH="${TEST_CACHE_ECR}:${clean_base}-x86_cpu"
  fi
fi
# main 兜底，保证任何构建都能从最近一次 main 缓存开始
CACHE_FROM_MAIN="${MAIN_CACHE_ECR}:latest-x86_cpu"
}
```

docker/Dockerfile.cpu

CPU 镜像的 Dockerfile，新增公共 base 阶段、将 rust-build 与 base-common 并行化，并在三个编译阶段注入 sccache，是跨 builder 编译缓存能否命中的关键。

```
# 公共 base 阶段：安装所有子阶段共享的 apt 包与可选 sccache。
# rust-build 刻意不 FROM base-common，而是 FROM base，保持最小依赖，
# 以便与 vllm-build 并行构建；sccache 的 S3 后端让编译缓存跨 BuildKit builder 存活。
FROM ubuntu:22.04 AS base

RUN --mount=type=cache,target=/var/cache/apt,sharing=locked \
    --mount=type=cache,target=/var/lib/apt,sharing=locked \
    apt-get update -y \
    && apt-get install -y --no-install-recommends ca-certificates curl git

ARG TARGETARCH
ARG USE_SCCACHE
ARG SCCACHE_DOWNLOAD_URL

# 按目标架构下载 sccache 二进制（amd64 用 x86_64，arm64 用 aarch64）
RUN if [ "$USE_SCCACHE" = "1" ]; then \
      echo "Installing sccache..." \
      && case "${TARGETARCH}" in \
        arm64) SCCACHE_ARCH="aarch64" ;; \
        amd64) SCCACHE_ARCH="x86_64" ;; \
        *) echo "Unsupported TARGETARCH for sccache: ${TARGETARCH}" >&2; exit 1 ;; \
      esac \
      && export SCCACHE_DOWNLOAD_URL="${SCCACHE_DOWNLOAD_URL:-https://github.com/
      mozilla/sccache/releases/download/v0.8.1/sccache-v0.8.1-${SCCACHE_ARCH}-unknown-
      linux-musl.tar.gz}" \
```

```
      && curl -L -o sccache.tar.gz "${SCCACHE_DOWNLOAD_URL}" \
      && tar -xzf sccache.tar.gz \
      && mv sccache-v0.8.1-${SCCACHE_ARCH}-unknown-linux-musl/sccache /usr/bin/sccache \
      && rm -rf sccache.tar.gz sccache-v0.8.1-${SCCACHE_ARCH}-unknown-linux-musl; \
    fi

# rust-build 段的 sccache 接线：通过 ARG 传入 S3 bucket 与 endpoint，
# 用 RUSTC_WRAPPER 包装 cargo 编译，使 rust 产物缓存落在 S3 而非本地 cache mount。
FROM base AS rust-build
ARG USE_SCCACHE
ARG SCCACHE_ENDPOINT
ARG SCCACHE_BUCKET_NAME=vllm-build-sccache
ARG SCCACHE_REGION_NAME=us-west-2

ENV SCCACHE_BUCKET=${USE_SCCACHE:+${SCCACHE_BUCKET_NAME}}
ENV SCCACHE_REGION=${USE_SCCACHE:+${SCCACHE_REGION_NAME}}
# 与 vllm-build 阶段的 sccache 守护进程错开端口，避免同一构建机上的端口冲突
ENV SCCACHE_SERVER_PORT=4227

RUN --mount=type=cache,target=/root/.cargo/registry,sharing=locked \
    --mount=type=cache,target=/root/.cargo/git,sharing=locked \
    --mount=type=secret,id=aws-credentials,target=/root/.aws/credentials,required=false \
    if [ "$USE_SCCACHE" = "1" ]; then \
      export RUSTC_WRAPPER=sccache; \
      if [ -n "${SCCACHE_ENDPOINT}" ]; then export SCCACHE_ENDPOINT="${SCCACHE_
      ENDPOINT}"; fi; \
      sccache --show-stats; \
    fi && \
    bash build_rust.sh && \
    if [ "$USE_SCCACHE" = "1" ]; then sccache --show-stats; fi
```

评论区精华

该 PR 没有人类 review 评论线程：reviewer jikunshang 直接 approve（无评论），claude[bot] 因 PR 来自 fork 而跳过自动 review。值得注意的"讨论"发生在开发过程中并已通过 commit 自述固化：① 空 SCCACHE_ENDPOINT 被无条件导出，导致 sccache 以非法自定义 S3 endpoint URI 启动崩溃，修复为仅非空时导出；② docker-container builder 处于独立网络命名空间，无法访问 EC2 IMDS 获取 S3 凭证，每次编译都写缓存失败，最终用 network=host 解决；③ 早期使用本地 --mount=type=cache（ccache/cargo registry），发现其不会被 --cache-to type=registry 导出，因此必须引入 S3 后端 sccache 才能跨 builder 命中。这些属于值得复用的排障经验。

- 暂无高价值评论线程

风险与影响

- 风险：主要风险集中在构建链路本身：

    - 构建脚本行为变更：image_build_cpu.sh 由 docker build 改为 docker buildx build，且每次运行会 docker buildx rm vllm-cpu-builder 重建 builder；若 buildx 初始化或 registry 缓存读写失败（如 ECR 凭证过期），CPU 镜像构建会直接失败，影响 CI 可用性。
    - 缓存隔离依赖命名约定：CPU 与 CUDA 共用同一批 ECR 仓库，仅靠 -x86_cpu tag 后缀隔离，若后续其他平台（如 XPU/HPU）也复用，需要保证后缀唯一；clean_docker_tag 把非法字符替换为 _，分支名碰撞可能让不同分支共享同一缓存 key，存在轻微互相污染风险。
    - sccache 的 S3 依赖：Dockerfile.cpu 新增 SCCACHE_BUCKET/SCCACHE_REGION 等环境变量与 aws-credentials secret mount，构建机必须能访问 IMDS 与 S3；在非 AWS 或网络受限环境（如本地调试、部分社区 CI）会出现缓存写失败，虽然 commit 中已做降级（required=false），但仍增加隐式外部依赖。
    - 多架构影响：Dockerfile.cpu 的改动涉及 TARGETARCH 分支（arm64 交叉编译路径），base 阶段对 sccache 的架构选择若出错会直接 exit 1，需要 arm64 CPU 构建 CI 覆盖确认。
    - 不涉及运行时与模型代码，无推理延迟 / 精度回归风险。
- 影响：影响范围限于 CI/ 镜像构建：
    - CI 耗时：CPU 镜像构建首次推送缓存后，后续构建的 csrc/rust/triton-cpu 编译层可跨 builder 复用，显著缩短 PR 验证等待时间（PR body 描述原先每次构建都全量重编）。
    - 基础设施：新增对私有 ECR 缓存仓库的读写权限与 vllm-build-sccache S3 bucket 依赖；ECR 存储随 mode=max 缓存层增长。
    - 团队：Intel/CPU CI 维护者受益最大，后续可参照相同模式为 XPU/HPU 或 release 镜像补齐缓存；对普通用户无感。
    - 风险标记：CI 关键路径变更，新增私有 ECR/S3 依赖，缓存隔离依赖 tag 约定，多架构路径需验证

关联脉络

- PR #46711 [CI] Add registry caching to x86 CUDA release image builds: PR body 明确提及 #46711 已为 x86 CUDA release 镜像构建加入 registry 缓存，并把 CPU/ROCm 划为 unchanged；本 PR 正是补齐 x86 CPU CI 镜像构建这个缺口，且复用了同套 ECR 缓存仓库与 tag 解析设计。
- PR #51905 [XPU][CI]Change to use global VLLM_DISABLE_COMPILE_CACHE=1 in Intel GPU CI: 同属 Intel 维护者推动的 CI 构建基础设施调整（Intel GPU CI 侧），与本 PR 一起反映 Intel 对硬件 CI 构建配置的系统性收紧。