第 25 章　Profiling 与 Runtime 调度

> **本章实测环境**：1 × NVIDIA H100 80GB；vLLM 0.17.0 / SGLang 0.5.6 / Nsight Systems 2025.6.1（下称 nsys，GPU 时间线工具）/ Nsight Compute 2025.2.1（下称 ncu，单 kernel 硬件计数器工具）。正文各项目的基线记录、复测数与产物均来自这一套环境的真实运行，产物按项目分根存放（见 0.2 节）。 **命令、参数名与依赖版本按 0.3 节的锁定版本书写**（vLLM 0.26.0 / SGLang 0.5.16），与上面的实测版本不同。两者的差异影响绝对数值，不影响判读方法与结论方向；参数名在版本间有变动的地方，正文与各项目的排错表已逐处注明现行写法与查证命令。**任何一处参数名报未知，以你所装版本的 `--help` 输出为准**，不要照抄。

这一章在整本教材里的位置：全书 26 章，本章是第 25 章。前面各章已经分别讲过 GPU 执行模型与硬件底座、自回归解码的瓶颈、KV Cache 与 PagedAttention、Flash Attention 与 decode 算子、Continuous Batching、各类并行策略与通信原语、投机解码、量化、前缀缓存、调度策略、分离式架构、框架对比与图编译；第 26 章讲工程化与生产部署。本章的位置在它们之后：前面各章告诉你有哪些机制和参数，本章讲的是拿到一个具体负载时，**怎么量出瓶颈在哪、怎么验证一次改动到底有没有用**。所以本章可以独立读完，不需要先读前 24 章；但本章用到的概念（KV Cache 分页、Continuous Batching、CUDA Graph）在各自章节里有完整推导，这里只做必要复述。

本章三个项目：

- **项目一**：用 `torch.profiler` 把一次推理请求拆成 prefill 与 decode 两段，定位各自耗时最高的算子。**回答**：这次推理的时间花在哪，换一个更快的注意力实现值不值得。
- **项目二**：对常驻的 vLLM / SGLang 服务做 Nsight Systems + Nsight Compute 端到端分析，从 GPU 利用率、Occupancy、Warp 停顿三层往下定位，并验证一项优化。**回答**：服务吞吐上不去时，瓶颈在算力还是在显存带宽，该调哪一个调度参数。
- **项目三**：读懂两个引擎的 CUDA Graph 捕获实现，在一个简化 Runtime 里复现捕获与回放，量化它相对 eager 模式在 decode 上的收益与代价。**回答**：CUDA Graph 省的是哪一层开销，代价有多大，什么场景下不该开。

三个项目都不是照步骤跑通就结束。每个项目必须走完同一个闭环：**先测出基线 → 用 profiler 定位 → 只改一处 → 同口径复测验证**。只跑通不定位，等于没做。

文中三个纯 CPU 脚本（`trace_top.py` / `ncu_top.py` / `capture_sizes.py`）贴的是真实输出；GPU 侧的基线记录与复测数来自上面所说的实测环境。

第零节 三个项目共用的环境与版本基线

0.1 硬件与系统

- 1 × NVIDIA H100 80GB
- RHEL 9 系（Rocky Linux 9 / AlmaLinux 9 / CentOS Stream 9），包管理器 `dnf`
- 全章单机单卡跑通，不需要多机、不需要多卡

没有 H100 也能做完全章。 三个项目里真正吃显存的只有加载 7B 权重（bf16 约 15 GB）与 KV Cache，本章的方法与结论都不依赖 80 GB 这个具体容量。按你手上的卡调整这几个参数：

显存	模型	`--max-model-len`	`--gpu-memory-utilization`	压测 `--num-prompts` / `--max-concurrency`
80 GB（本章）	Qwen2.5-7B	8192	0.88	256 / 32
48 GB（A6000、L40S）	Qwen2.5-7B	4096	0.90	128 / 16
24 GB（4090、A10）	Qwen2.5-1.5B	2048	0.90	64 / 8

哪些结论会变、哪些不会：绝对吞吐、TTFT、每个 kernel 的耗时都会变，**不要拿本章的数字对标**。判读方法、瓶颈类型（decode 主路径受限于显存带宽）、CUDA Graph 只对 decode 有整图收益这些结论方向不变。换小模型后层数与 hidden 维度都会变，`capture_sizes.py` 与手写 Runtime 的参数要跟着改（脚本都有对应的命令行开关）。

换卡之后必须自己重测一遍再写进简历，不能把本章的数字当成自己的实测结果。

0.2 目录约定：全章相对路径

所有命令都在同一个工作根目录下执行。先固定它：

```
mkdir -p ch25_prof && cd ch25_prof
export CH25_ROOT="$(pwd)"
mkdir -p "$CH25_ROOT"/{models,bench,profile,graph,reports}
echo "$CH25_ROOT"
```

之后一律用 `"$CH25_ROOT/..."` 引用，不写 `~/`、`/root/`、`/home/` 这类机器专属路径。换一台机器只需重新 `cd` 再 `export`，全章命令一字不用改。

本章要同时开 3 到 4 个终端（服务一个、压测一个、采集一个）。`export` 只在当前终端生效，**每开一个新终端都要先重跑这两行**，否则后面所有带 `$CH25_ROOT` 的路径都会变成空串。嫌麻烦可以写进 `~/.bashrc`：

```
echo "export CH25_ROOT=\"$CH25_ROOT\"" >> ~/.bashrc
```

实测产物按项目分根存放：`$CH25_ROOT/projects/pr1`、`pr2`、`pr3` 分别对应项目一、二、三（pr 即 project），每个根下的子目录与上面相同。各项目第九节的本步产生的文件表里的路径都按这套布局写。

关于这些产物文件要先说清楚。 本文档给出的是方法、完整命令、可直接运行的脚本，以及本章实测跑出来的读数；各项目第九节反复引用的那些原始产物本身（`.nsys-rep`、`.ncu-rep`、chrome trace、各步的 JSON 与人写的读数笔记）体积较大，**随课程发放，不包含在这份文档里**。它们的用途是让你把自己采到的结果与本章的逐项对照，看差异出在哪一步。只看这份文档，你需要在自己的机器上跑出属于自己的那一份产物，第九节的表格就是产出清单。

0.3 版本锁定表

组件	锁定版本	来源
vLLM	0.26.0	https://github.com/vllm-project/vllm/releases/tag/v0.26.0
SGLang	0.5.16	https://github.com/sgl-project/sglang/releases/tag/v0.5.16
torch	2.11.0	https://pypi.org/project/torch/2.11.0/
transformers	5.14.1	https://pypi.org/project/transformers/5.14.1/
flashinfer-python	0.6.14	两个框架的 PyPI `requires_dist` 都锁这个版本
CUDA	13	torch 2.11.0 的 `requires_dist` 拉的是 `nvidia-cudnn-cu13` / `nvidia-nccl-cu13`
Python	3.12	四个包 `requires_python` 的交集
huggingface_hub	1.26.0	https://pypi.org/project/huggingface-hub/
modelscope / modelscope-hub	1.39.0 / 0.1.8	https://pypi.org/project/modelscope/
Nsight Systems	2026.4.1	https://developer.nvidia.com/w/devtools/repos/rhel9/x86_64/
Nsight Compute	2026.2.1	同上
uv	0.12.0	https://pypi.org/project/uv/

撰写时上游已有 torch 2.13.0、flashinfer-python 0.6.16.post1、modelscope 1.39.1，本课统一锁上表这一套以保证全书可复现。

必须说清的一点：两个框架的官方安装命令**本身都是裸装、不带版本号**。上表的 `==` 是本章额外加的约束，不是官方写法。锁版本是为了让几百个人跑出可比的结果。profiling 尤其如此：换一个 torch 小版本，kernel 选型就可能变，算子排行榜会完全不同。

0.4 三个隔离 venv

本章要装三套环境。**vLLM 和 SGLang 不能装进同一个 venv**：两边有五处精确 pin 直接冲突（`numba` 0.65.1 vs 0.65.0、`tilelang` 0.1.11 vs 0.1.9、`apache-tvm-ffi` 0.1.11 vs 0.1.10、`torchcodec` ==0.11.1 vs >=0.14、`outlines_core` 0.1.26 vs 0.2.14），pip / uv 的解析器会直接判 `ResolutionImpossible`。项目一只用裸 torch + transformers，再单独开一个，避免两个框架的依赖树互相牵制。

RHEL 9 默认的 `python3` 是 3.9，低于要求的 `>=3.10`，先显式装 3.12：

```
sudo dnf install -y python3.12 python3.12-pip
python3.12 --version
```

RHEL 9 通过 AppStream 提供 python3.12，与默认的 3.9 并存；装不上时以 `dnf search python3.1` 的实际输出为准。

三个 venv：

```
# 一、裸 torch + transformers，项目一与项目三的简化 Runtime 用
python3.12 -m venv "$CH25_ROOT/.venv-torch"
source "$CH25_ROOT/.venv-torch/bin/activate"
pip install --upgrade pip
pip install torch==2.11.0 transformers==5.14.1
python -c "import torch, transformers as tf; \
    print(torch.__version__, torch.version.cuda, tf.__version__)"
deactivate

# 二、vLLM
python3.12 -m venv "$CH25_ROOT/.venv-vllm"
source "$CH25_ROOT/.venv-vllm/bin/activate"
pip install --upgrade pip && pip install vllm==0.26.0
python -c "import vllm, torch; print(vllm.__version__, torch.__version__, torch.version.cuda)"
deactivate

# 三、SGLang
python3.12 -m venv "$CH25_ROOT/.venv-sglang"
source "$CH25_ROOT/.venv-sglang/bin/activate"
pip install --upgrade pip && pip install uv==0.12.0
uv pip install --prerelease=allow sglang==0.5.16
python -c "import sglang; print(sglang.__version__)"
deactivate
```

三条检查命令预期分别打印 `2.11.0 13.x 5.14.1`、`0.26.0 2.11.0 13.x`、`0.5.16`。

三点说明：

- **不要给 vLLM 加 `--extra-index-url .../cu129`**。v0.26.0 的默认 wheel 由 CUDA 13 构建，torch 2.11.0 的依赖链拉的也是 `nvidia-cudnn-cu13` / `nvidia-nccl-cu13`（PyPI 上 torch 2.11.0 的 `requires_dist` 可直接核对）。照抄 cu129 会把 torch 换成 cu129 版，和 CUDA 13 构建的 vllm 二进制混用。
- `--prerelease=allow` 不能省：SGLang 的依赖里有预发布版本的 `flash-attn-4`，uv 默认拒绝。
- `pip install torch==2.11.0` 装的是 PyPI 默认 wheel。它绑的 CUDA 运行时必须和机器上的驱动匹配，`torch.version.cuda` 打不出 13.x 或跑起来报驱动版本不足时，去 `download.pytorch.org` 取对应 CUDA 版本的 wheel，不要在 venv 里混装。

0.5 Nsight Systems 与 Nsight Compute

本章两个工具都要用。走 NVIDIA devtools 仓库，服务器无桌面，装 CLI 版：

```
CUDA_REPO=https://developer.download.nvidia.com/compute/cuda/repos
sudo rpm --import "$CUDA_REPO/ubuntu1804/x86_64/7fa2af80.pub"
sudo dnf install -y 'dnf-command(config-manager)'
RHEL_MAJOR=$(source /etc/os-release; echo ${VERSION_ID%%.*})
ARCH=$(rpm --eval '%{_arch}' | sed s/aarch/arm/)
DEVTOOLS_REPO=https://developer.download.nvidia.com/devtools/repos
sudo dnf config-manager --add-repo "$DEVTOOLS_REPO/rhel$RHEL_MAJOR/$ARCH/"
sudo dnf install -y nsight-systems-cli nsight-compute
nsys --version && ncu --version
```

GPG key 路径官方写的就是 `ubuntu1804` 下那个 `7fa2af80.pub`，不是笔误。这个仓库的目录索引里实际存在 `nsight-systems-cli-2026.4.1.191-1.x86_64.rpm` 与 `nsight-compute-2026.2.1.5-1.x86_64.rpm`。不要走 CUDA 主仓库装 nsight，那里 `cuda-nsight-*` 系列的包实测拿不到（确切文件名返回 404）。

两处权限必须提前放开，否则数据缺失或直接报错。

nsys 采 CPU 侧线程调度数据需要放开内核限制（缺了不报错，只是这部分数据没有，CUDA kernel 时间线不受影响）。先看当前值，`2` 或更小才够用：

```
cat /proc/sys/kernel/perf_event_paranoid   # 大于 2 就要改
sudo sh -c 'echo kernel.perf_event_paranoid=2 > /etc/sysctl.d/local.conf'
sudo sysctl --system
```

ncu 读硬件性能计数器需要额外授权，非 root 跑会报 `ERR_NVGPUCTRPERM`：

```
echo 'options nvidia NVreg_RestrictProfilingToAdminUsers=0' \
  | sudo tee /etc/modprobe.d/nsight.conf
sudo dracut --regenerate-all -f
sudo reboot
```

第二条是因为 nvidia 内核模块被打包在 initramfs 里，不重建 initramfs，新的模块参数不会在下次启动时生效。

重启后这样核对，输出里这一项为 `0` 才算放开：

```
grep RestrictProfiling /proc/driver/nvidia/params   # 期望 RestrictProfilingToAdminUsers: 0
```

拿不到 root 时怎么办。 这一步要写 `/etc/modprobe.d` 并重启，租用的共享实例通常给不了。先把这件事问清楚再开始项目二，不要做到一半才发现。

- 租卡时**优先选明确写了支持 GPU profiling 的实例**，或直接向管理员申请开启 `NVreg_RestrictProfilingToAdminUsers=0`。
- 完全拿不到权限时，项目二仍可交付**除 ncu 之外的全部内容**：压测基线、nsys 时间线与 kernel 汇总、按耗时占比定位、改一处并复测、按判据决定采纳与否，都不需要计数器权限。缺的只是四象限判读那一步的 Compute 与 Memory 吞吐，对应地把结论口径收敛成“我定位到了耗时最高的 kernel，但没有计数器权限，无法判定它是算力受限还是访存受限”。**这样讲不会有风险，拿本章的数字冒充自己的实测才会。**NVIDIA 官方已预告：从驱动 R610 起改用 capabilities 授权，这条 regkey“将在未来版本移除”。驱动较新时改用 `nvidia-modprobe -f`

`/proc/driver/nvidia/capabilities/profiler-device`。**没有其他可选路径**：不放权就采不到计数器，ncu 只能出时长不出任何 Section。

0.6 模型下载：双轨

用 **Qwen/Qwen2.5-7B-Instruct**。HF 与 ModelScope 的 repo id 同名。

主轨，ModelScope（国内直连）：

```
source "$CH25_ROOT/.venv-torch/bin/activate"
pip install modelscope==1.39.0 modelscope-hub==0.1.8
ms download Qwen/Qwen2.5-7B-Instruct --local-dir "$CH25_ROOT/models/Qwen2.5-7B-Instruct"
```

`ms download` 的 repo id 是**位置参数**，目录参数是 `--local-dir`（连字符）。旧写法 `modelscope download --model X --local_dir Y` 仍能跑，但会打 DeprecationWarning。

备轨，hf-mirror 镜像：

```
pip install "huggingface_hub==1.26.0"
export HF_ENDPOINT=https://hf-mirror.com
hf download Qwen/Qwen2.5-7B-Instruct --local-dir "$CH25_ROOT/models/Qwen2.5-7B-Instruct"
```

这里有一处必然出错的地方：hf-mirror.com 首页的示例至今还写 `huggingface-cli download`，而 `huggingface-cli` 这个命令在 huggingface_hub v1.0.0 已被**彻底删除**（不是改名兼容），照抄必然 `command not found`。现在只有 `hf`。第三轨是 HuggingFace 官方（需科学上网），去掉 `HF_ENDPOINT` 那行即可。三轨都要注意 Xet：huggingface_hub v1.x 默认走 Xet 传输，企业网络下常见卡住或校验失败，卡住时设 `HF_HUB_DISABLE_XET=1` 退回普通 HTTP。

核对一下模型结构参数，后面三个项目都要用：

```
python - <<'PY'
import json, os
p = os.path.join(os.environ["CH25_ROOT"], "models/Qwen2.5-7B-Instruct/config.json")
c = json.load(open(p))
keys = ["num_hidden_layers", "hidden_size", "num_attention_heads",
        "num_key_value_heads", "intermediate_size", "vocab_size", "torch_dtype"]
print({k: c[k] for k in keys})
PY
```

预期：28 层 / hidden 3584 / 28 个 Q 头 / **4 个 KV 头** / intermediate 18944 / vocab 152064 / bfloat16。`head_dim` 这个字段在 Qwen2.5 的 config.json 里**根本不存在**，128 是 3584÷28 算出来的；项目三搭简化 Runtime 时要用到这一组形状。

项目一　用 torch.profiler 定位推理热点算子

这个项目要回答的问题：一次推理请求慢，时间到底花在哪？是注意力实现不够快，还是别的地方？

本项目的结论：在 7B 模型、1024 输入、batch=1 这个形状下，把 `--attn` 从 eager 换成 sdpa 只让 GPU 时间下降 **6.6%**（289→270 ms），换完之后注意力算子（`aten::_flash_attention_forward`）只占 GPU 时间的 **3.24%**。占比最高的是 GEMM（通用矩阵乘，`aten::mm` 一项就是 **56.11%**；`aten::` 是 PyTorch 底层算子的命名前缀）与每步固定发出的那几百次 kernel launch（CPU 把一个 GPU 函数提交给驱动的动作，单次几微秒）。**换一个更快的注意力实现，在这个形状下的收益上限本来就很低**，这个方向不该是第一选择。

本次的墙钟反而从 **1.008 s** 上升到 **1.331 s**。核对与 `--attn` 无关的算子之后确认，上升来自两次运行之间主机侧执行速度的变化，不是 sdpa 造成的：单轮对照不足以支撑墙钟侧的结论。这条核对方法（改一个变量之后，先看不该变的算子有没有变）比这个项目的任何一个数字都更常用。

读完能讲清：

1. prefill 与 decode 的算子构成为什么必须分开看；
2. Self 与 total 两列的区别，以及用错列会得出什么错误结论；
3. 怎么判断一次对照实验的测量环境是否可比。

一、目标与交付物

对一次真实的推理请求做 `torch.profiler` 采集，**分别**给出 prefill 与 decode 两段的算子排行，指出耗时最高的算子，给出优化方向，改一处再复测。

交付物三样：

1. `prof_infer.py`：采集脚本，把 prefill 与 decode 用 `record_function` 标注开，导出算子表 + chrome trace；
2. `trace_top.py`：纯标准库解析 trace，按算子汇总 GPU 侧耗时、算 GPU 占空比与 launch 密度；
3. 一份走完基线 → 定位 → 改一处 → 复测这一完整闭环的记录。

做到什么算完成（四条缺任何一条，这个项目就没走完闭环）：

1. 用 `trace_top.py --annotation` 切出 prefill 与 decode 两段，各自给出前三名算子与占比；
2. GPU 占空比与 launch 密度有实测数；
3. 改动前后有同口径的两组数（墙钟由 `prof_infer.py` 落进 `bench/*.json`，算子表自己 `tee` 留存）；
4. 能说清这次改动为什么被采纳或被否决，以及测量环境是否可比。

二、开始前必须清楚的三件事

第一，profiler 采的是两路数据。 `ProfilerActivity.CPU` 采 PyTorch 在算子入口埋的桩（`aten::linear` 这些）；`ProfilerActivity.CUDA` 采 GPU 侧活动（kernel、memcpy、memset），数据来自 NVIDIA 的 CUPTI 采集接口。两路数据由 profiler 自动对应起来。**只开 CPU 那一路，看到的全是 CPU 提交这些算子花了多久，和 GPU 实际算了多久无关。**

第二，prefill 和 decode 的热点必然不同，混着统计等于没统计。 prefill 一次算完整个 prompt，矩阵乘两个维度都很大，算力密集；decode 每步只算 1 个 token，batch=1 时 GEMM 退化成 GEMV（矩阵乘向量；服务态合批后是 M 维等于 batch 的扁 GEMM），算力利用率低，瓶颈转到读权重和读 KV Cache 的带宽上。这里的算术很直观：7B 模型 bf16 权重约 15 GB，**每一步 decode 都要把它完整读一遍**，而实际参与计算的只有 1 行输入，而每步要发的 kernel 数量不变，launch 开销的占比被放大。

第三，CUDA 是异步的。 `model(...)` 返回时 GPU 可能一个 kernel 都没跑完。**自己用 `time.perf_counter` 计时而前面不加 `torch.cuda.synchronize()`，量到的是 launch 时间不是执行时间**，通常小一个量级。profiler 报的 GPU 时间来自 CUPTI 的设备侧时间戳，不受影响。

三、采集脚本

新建 `$CH25_ROOT/profile/prof_infer.py`：

```
"""对一次推理请求做 torch.profiler 采集，把 prefill 与 decode 分段标注后导出 trace，
同时记录墙钟耗时与实际生成的 token 数，落盘成 JSON 供前后两次对照。
用法: python prof_infer.py --model "$CH25_ROOT/models/Qwen2.5-7B-Instruct" \
          --input-len 1024 --decode-steps 31 --attn sdpa --outdir "$CH25_ROOT/profile"
"""
import argparse, json, os, time

import torch
from torch.profiler import ProfilerActivity, profile, record_function
from transformers import AutoModelForCausalLM, AutoTokenizer


def build_inputs(tok, batch, seq_len, device):
    """用词表内的随机 token 造输入，不依赖任何外部数据集。"""
    g = torch.Generator().manual_seed(0)
    ids = torch.randint(0, tok.vocab_size, (batch, seq_len), generator=g)
    return ids.to(device)


def main():
    ap = argparse.ArgumentParser()
    ap.add_argument("--model", required=True)
    ap.add_argument("--batch", type=int, default=1)
    ap.add_argument("--input-len", type=int, default=1024)
    ap.add_argument("--decode-steps", type=int, default=31)
    ap.add_argument("--attn", default="sdpa",
                    help="eager / sdpa / flash_attention_2 / flex_attention")
    ap.add_argument("--outdir", default=".")
    ap.add_argument("--benchdir", default=None,
                    help="wall/tokens 的 JSON 落盘目录，默认与 --outdir 同级的 bench/")
    ap.add_argument("--row-limit", type=int, default=15)
    a = ap.parse_args()

    assert torch.cuda.is_available(), "本脚本要在 GPU 机器上跑"
    dev = "cuda"
    tok = AutoTokenizer.from_pretrained(a.model)
    # transformers v5：参数名是 dtype，torch_dtype 已弃用；不显式写时默认 "auto"（按权重文件里存的
    # 精度加载），与 v4 默认 fp32 的行为不同。跨版本比 profiling 结果前先确认这一项。
    model = AutoModelForCausalLM.from_pretrained(
        a.model, dtype=torch.bfloat16, attn_implementation=a.attn).to(dev).eval()
    print(f"attn_implementation = {model.config._attn_implementation}  dtype = {model.dtype}")

    ids = build_inputs(tok, a.batch, a.input_len, dev)

    # 预热：第一次前向要做 kernel 选型、JIT、显存池扩张。不预热，trace 里全是一次性开销。
    with torch.inference_mode():
        out = model(input_ids=ids, use_cache=True)
        nxt = out.logits[:, -1].argmax(-1, keepdim=True)
        model(input_ids=nxt, past_key_values=out.past_key_values, use_cache=True)
    torch.cuda.synchronize()

    os.makedirs(a.outdir, exist_ok=True)
```

```
    benchdir = a.benchdir or os.path.join(
        os.path.dirname(os.path.abspath(a.outdir)), "bench")
    os.makedirs(benchdir, exist_ok=True)

    # 墙钟必须把 synchronize 放在计时区间内部：CUDA 异步，不同步就量成了 launch 时间。
    torch.cuda.synchronize()
    t0 = time.perf_counter()
    with torch.inference_mode(), profile(
            activities=[ProfilerActivity.CPU, ProfilerActivity.CUDA],
            record_shapes=True, with_stack=False) as prof:
        with record_function("prefill"):
            out = model(input_ids=ids, use_cache=True)
            past = out.past_key_values
            nxt = out.logits[:, -1].argmax(-1, keepdim=True)
        torch.cuda.synchronize()

        for _ in range(a.decode_steps):
            with record_function("decode"):
                out = model(input_ids=nxt, past_key_values=past, use_cache=True)
                past = out.past_key_values
                nxt = out.logits[:, -1].argmax(-1, keepdim=True)
        torch.cuda.synchronize()
    wall_s = time.perf_counter() - t0
    tokens = 1 + a.decode_steps   # prefill 出 1 个，之后每步 1 个

    # 排序键写 self_device_time_total。self_cuda_time_total 这个属性在 torch 2.11 已标 @deprecated
    #  （torch/autograd/profiler_util.py，FutureWarning）；作为排序字符串它仍能用，是因为 _build_table
    # 里做了 sort_by.replace("cuda", "device")，但别再往新代码里写它。
    print(prof.key_averages().table(sort_by="self_device_time_total", row_limit=a.row_limit))
    trace = os.path.join(a.outdir, f"trace_{a.attn}_in{a.input_len}.json")
    prof.export_chrome_trace(trace)

    meta = {"attn": a.attn, "wall_s": wall_s, "tokens": tokens,
            "input_len": a.input_len, "batch": a.batch}
    bench_json = os.path.join(benchdir, f"{a.attn}.json")
    with open(bench_json, "w", encoding="utf-8") as f:
        json.dump(meta, f, ensure_ascii=False, indent=2)

    print(f"\nwall_s = {wall_s:.4f} s   tokens = {tokens}")
    print(f"trace 已导出: {trace}（用 https://ui.perfetto.dev/ 打开，或喂给 trace_top.py）")
    print(f"墙钟与 token 数已落盘: {bench_json}")


if __name__ == "__main__":
    main()
```

五处依据说明：

写法	依据
`dtype=torch.bfloat16` 而不是 `torch_dtype=`	transformers v5.14.1 的 `src/transformers/modeling_utils.py` 里有 `logger.warning_once("` torch_dtype `is deprecated! Use` dtype `instead!")`
必须显式写 dtype	同文件 `from_pretrained` 里 `if dtype is None: dtype = "auto"`。v5 默认按权重文件里存的精度加载，v4 默认 fp32；同一段代码跨大版本会加载出不同精度，profiling 结果不可比
`attn_implementation` 的合法取值	同文件 `get_correct_attn_implementation` 的报错串里列出：`eager`、`sdpa`、`flash_attention_2` / `_3`、`flex_attention`，以及各自的 `paged|` 前缀变体
`sort_by="self_device_time_total"`	`torch/autograd/profiler_util.py` 里 `self_cuda_time_total` 带 `@deprecated("... Use` self_device_time_total `instead.")`；同文件 `_build_table` 对排序键做 `sort_by.replace("cuda", "device")`，所以旧字符串不报错，但属性访问会 FutureWarning
不加 `schedule`	`torch.profiler.schedule(wait, warmup, active, repeat, skip_first)` 是给**长期运行的循环**设定采集窗口用的：wait 阶段完全不采集（零开销）、warmup 采但丢弃、active 才留下。本脚本跑的是有界的 1 次 prefill + N 次 decode，已经在 profiler 外面手动预热过，再套 schedule 反而会把 prefill 那一步吃进 warmup 丢掉

`prof.key_averages()` 支持 `group_by_input_shape=True`：配合 `record_shapes=True`，能把同一个算子按输入形状拆开。decode 阶段这一项很有用：同名的 `aten::linear` 在 prefill 是 `[1024, 3584] × [3584, 18944]`，在 decode 是 `[1, 3584] × [3584, 18944]`，性质完全不同。

四、基线：跑一次并读表

```
source "$CH25_ROOT/.venv-torch/bin/activate"
cd "$CH25_ROOT/profile"
python prof_infer.py --model "$CH25_ROOT/models/Qwen2.5-7B-Instruct" \
    --input-len 1024 --decode-steps 31 --attn eager --outdir "$CH25_ROOT/profile"
```

`--decode-steps 31` 对应 1 次 prefill 加 31 次 decode，共 32 个 token，与第九节实测记录里的 `tokens=32` 一致；改成别的值，wall 就不能与本章的基线记录比较。32 这个值是折中：再少，一次性开销（kernel 选型、显存池扩张）在总时间里的占比就压不下去；再多，trace 体积和采集时长都会明显上升。

脚本会打印算子表，并把墙钟与 token 数写进 `$CH25_ROOT/bench/eager.json`。**算子表要自己重定向留存**，第九节的对照表用的就是这两份文件：

```
python prof_infer.py --model "$CH25_ROOT/models/Qwen2.5-7B-Instruct" \
    --input-len 1024 --decode-steps 31 --attn eager --outdir "$CH25_ROOT/profile" \
    | tee "$CH25_ROOT/profile/top_eager.txt"
```

输出表的列名是固定的：`Name` / `Self CPU %` / `Self CPU` / `CPU total %` / `CPU total` / `CPU time avg` / `Self CUDA` / `Self CUDA %` / `CUDA total` / `CUDA time avg` / `# of Calls`。设备侧那四列的标题由 `f"Self {device_name}"` 这类模

板生成，`device_name` 取的是 `use_device.upper()`（`torch/autograd/profiler_util.py`），所以在 CUDA 机器上就是 `CUDA`。

Self 和 total 是两回事，这是读这张表最容易出错的地方。 `self_device_time_total` 的语义是“Device time excluding child operations”（同文件字段文档串），即扣掉子算子的净耗时；`device_time_total` 含子算子。`aten::linear` 会把时间记在自己名下**并且**记在它调用的 `aten::addmm` 名下，所以 total 列相加会远超窗口时长。**同一个算子的不同层级之间，排序和归因看 Self 列。**

但 Self 列也有一处不能跨的界，这是这张表第二个容易出错的地方。 表里同时有两类行：

1. **CPU 算子行**，名字以 `aten::` 开头，例如 `aten::mm`。它的 Self CUDA 是这个算子直接拉起的那些 kernel 的设备时间合计。
2. **kernel 行**，名字是 `nvjet_tst_*`、`void at::native::*` 这类，就是 GPU 上真正执行的那个 kernel。

`aten::mm` 拉起的正是 `nvjet_tst_*` 这一批 GEMM kernel，**同一段 GPU 时间在两类行里各记了一次**。本章实测那张表里，`aten::mm` 占 56.11%、`nvjet_tst_256x8_64x6_4x1_v_bz_TNT` 占 31.39%、`nvjet_tst_64x8_64x16_2x1_v_bz_splitK_TNT` 占 14.43%，三行相加已经超过 100%；整张表按两类分开算，CPU 算子行合计 93.7%、kernel 行合计 76.1%，混加是 169.8%。

每一行的百分比单独看都是对的，因为分母统一是表尾那个 `Self CUDA time total`，它只累加 kernel 行的设备时间。`aten::mm` 的 56.11% 读作"这个算子拉起的 kernel 占全部 GPU kernel 时间的 56.11%"，`nvjet_tst_256x8` 的 31.39% 读作"这一个 kernel 占 56.11% 里的多少"。**两个数都对，就是不能相加。**

规则：排序和求和都要在同一类行里做。想知道 GEMM 这一类占多少，加 kernel 行里的 `nvjet_tst_*`；想知道哪个 PyTorch 算子最贵，加 `aten::` 那一类。跨类相加得到的数没有意义。

基线记录（实测；算子明细以 trace 为准，`reports/torch_prof.md`）

段	指标	eager 基线	sdpa 复测（第七步回填）
end-to-end	wall_s（32 tokens）	**1.008**	**1.331**
prefill	Self CUDA 第 1 名算子 / 占比	（见 trace）	（见 trace）
prefill	前三名算子累计占比	（见 trace）	（见 trace）
decode	Self CUDA 第 1 名算子 / 占比	（见 trace）	（见 trace）
decode	前三名算子累计占比	（见 trace）	（见 trace）
decode	GPU 占空比	（见 `trace_top.py`）	（见 `trace_top.py`）

五、把 trace 汇成可比的数

`key_averages()` 那张表按算子名聚合，但它不回答两个关键问题：GPU 有多少时间是空着的？launch 发得有多密？这两个要从 trace 的时间戳里算。

新建 `$CH25_ROOT/profile/trace_top.py`：

```
"""解析 torch.profiler 导出的 chrome trace，按算子汇总 GPU 侧耗时并算 GPU 占空比。纯标准库。
用法: python trace_top.py --trace prefill.json --top 12 [--annotation decode]
只做汇总，不重算任何 profiler 已经算好的量。
"""
import argparse, gzip, json, os

DEVICE_CATS = {"kernel", "gpu_memcpy", "gpu_memset"}
LAUNCH_NAMES = ("cudaLaunchKernel", "cudaGraphLaunch", "cuLaunchKernel")


def load_events(path):
    op = gzip.open if path.endswith(".gz") else open
    with op(path, "rt", encoding="utf-8") as f:
        doc = json.load(f)
    evs = doc["traceEvents"] if isinstance(doc, dict) else doc
    return [e for e in evs if e.get("ph") == "X" and "dur" in e]


def annotation_window(events, name):
    """取名字匹配的 user_annotation 区间（CPU 侧时间轴），用于把 trace 切成 prefill / decode 两段。"""
    hits = [e for e in events
            if e.get("cat") in ("user_annotation", "cpu_op") and e.get("name") == name]
    if not hits:
        return None
    lo = min(e["ts"] for e in hits)
    hi = max(e["ts"] + e["dur"] for e in hits)
    return lo, hi


def main():
    ap = argparse.ArgumentParser()
    ap.add_argument("--trace", required=True, help="export_chrome_trace 落盘的 json 或 json.gz")
    ap.add_argument("--top", type=int, default=12)
    ap.add_argument("--annotation", default=None,
                    help="只统计落在该 record_function 区间内的 GPU 事件")
    a = ap.parse_args()

    events = load_events(a.trace)
    win = annotation_window(events, a.annotation) if a.annotation else None
    if a.annotation and win is None:
        raise SystemExit(f"trace 里没有名为 {a.annotation!r} 的标注，先在脚本里加 record_function")

    dev = [e for e in events if e.get("cat") in DEVICE_CATS]
    if win:
        dev = [e for e in dev if win[0] <= e["ts"] <= win[1]]
    if not dev:
        raise SystemExit("没有 GPU 侧事件。确认 activities 里带了 "
                         "ProfilerActivity.CUDA，且采集窗口内真的有 kernel")

    agg = {}
    for e in dev:
        r = agg.setdefault(e["name"], [0.0, 0])
```

```
        r[0] += e["dur"]
        r[1] += 1

    span_lo = min(e["ts"] for e in dev)
    span_hi = max(e["ts"] + e["dur"] for e in dev)
    span = span_hi - span_lo
    # 繁忙时间取所有设备侧事件区间的并集，不是时长直接相加：
    # 多流并发时同一时刻可能有多个 kernel 在执行，直接相加会重复计入，占空比可能超过 100%。
    ivs = sorted((e["ts"], e["ts"] + e["dur"]) for e in dev)
    busy = 0.0
    cur_lo, cur_hi = ivs[0]
    for lo, hi in ivs[1:]:
        if lo > cur_hi:
            busy += cur_hi - cur_lo
            cur_lo, cur_hi = lo, hi
        else:
            cur_hi = max(cur_hi, hi)
    busy += cur_hi - cur_lo
    launches = sum(1 for e in events if e.get("name") in LAUNCH_NAMES
                   and (not win or win[0] <= e["ts"] <= win[1]))

    tag = f"[{a.annotation}] " if a.annotation else ""
    print("=" * 78)
    print(f"{tag}trace={os.path.basename(a.trace)}  GPU 事件 {len(dev)} 条  唯一算子 {len(agg)} 个")
    print(f"GPU 时间跨度 = {span/1000:.2f} ms   GPU 繁忙（区间并集）= {busy/1000:.2f} ms")
    print(f"GPU 占空比 = {busy/span*100:.1f}%   (明显偏低时瓶颈多半不在算力)")
    if launches:
        print(f"kernel launch 调用 = {launches} 次   平均每次 launch 换来 GPU 时间 = "
              f"{busy/launches:.1f} us  (这个值越小，launch 开销占比越重)")
    else:
        print("kernel launch 调用 = 0 次（没采 CPU 活动，或 kernel 全在 CUDA Graph 里被一次性 replay）")
    print("-" * 78)
    print(f"{'算子':<44}{'总时长(ms)':>12}{'次数':>8}{'均值(us)':>11}{'占比':>7}")
    for name, (tot, cnt) in sorted(agg.items(), key=lambda kv: -kv[1][0])[:a.top]:
        short = name if len(name) <= 42 else name[:39] + "..."
        print(f"{short:<44}{tot/1000:>12.3f}{cnt:>8}{tot/cnt:>11.1f}{tot/busy*100:>6.1f}%")
    print("-" * 78)
    top3 = sum(v[0] for v in sorted(agg.values(), key=lambda x: -x[0])[:3])
    print(f"排前三的算子占了 {top3/busy*100:.1f}% 的 GPU 时间")


if __name__ == "__main__":
    main()
```

```
python trace_top.py --trace trace_eager_in1024.json --annotation prefill --top 12
python trace_top.py --trace trace_eager_in1024.json --annotation decode  --top 12
```

用一份构造的 trace 验证解析、按阶段切分与汇总路径，输出如下：

```
================================================================================
[decode] trace=fixture_decode.json  GPU 事件 288 条  唯一算子 6 个
GPU 时间跨度 = 6.06 ms   GPU 繁忙（区间并集）= 4.18 ms
GPU 占空比 = 69.0%   (明显偏低时瓶颈多半不在算力)
kernel launch 调用 = 288 次   平均每次 launch 换来 GPU 时间 = 14.5 us  (这个值越小，launch 开销占比越重)
--------------------------------------------------------------------------------
算子                                                总时长(ms)       次数      均值(us)      占比
nvjet_tst_128x16_64x8_4x1_v_bz_TNT                   1.952      32        61.0  46.7%
void cutlass::device_kernel<flash::Flash...          1.216      32        38.0  29.1%
void at::native::vectorized_elementwise_...          0.416      64         6.5   9.9%
void at::native::elementwise_kernel<128...           0.307      96         3.2   7.3%
void at::native::reduce_kernel<256, 4, a...          0.160      32         5.0   3.8%
void at::native::unrolled_elementwise_ke...          0.131      32         4.1   3.1%
--------------------------------------------------------------------------------
排前三的算子占了 85.7% 的 GPU 时间
```

（上面的数值来自构造的 fixture，只用于验证脚本能正确按阶段切分与汇总，**不是任何机器上的实测性能**。真实数字要用第四步的命令跑出来。

顺带给一个量级参照，方便你判断自己跑出来的数合不合理：28 层的 7B 模型，一步 decode 大约会发出 280 到 560 个 kernel（每层十几个算子），其中 GEMM 类约 4 × 28 = 112 个；单步 decode 的 GPU 时间下限由权重读取决定，bf16 权重约 15 GB、H100 的 HBM 带宽约 3.35 TB/s，所以一步不会低于 4.5 ms 左右。**如果你采了 32 步却只看到几百个事件，或者 32 步的总时间只有几毫秒，那是采集窗口没设对，不是模型快。** 上面 fixture 那一组数的量级对应的是单步 decode，不是 32 步。kernel 名按 transformers + PyTorch 这条路径上会出现的样子写：GEMM 是 cuBLASLt 的 `nvjet_*`，注意力是 cutlass 的 FlashAttention，其余是 `at::native::*` 的逐元素与归约算子。**不会出现 `vllm::` 或 `flashinfer::` 前缀**，那是推理引擎自带的算子，项目二才会看到。）

这个脚本的一处近似必须说清：GPU 事件的时间戳在设备时间轴上，而 `record_function` 的区间在 CPU 时间轴上，kernel 总是在 launch 之后才执行。这里按时间窗口做近似归属，落在边界上的 kernel 可能归错段。要精确归属得跟 trace 里的 flow 事件（`"ph": "s"` / `"ph": "f"`），那是自选扩展，不在主线。

六、定位：按顺序问三个问题

1. **GPU 占空比是多少？** 参考线：**85% 以上算忙**，60%～85% 需要结合第 2、3 问一起判，**60% 以下基本可以断定 GPU 在等 CPU**。占空比偏低说明 GPU 有大量空闲，瓶颈不在算力，往下看第 3 问。
2. **排名靠前的算子里，GEMM 类和 attention 类各占多少？** 两个阶段 GEMM 占大头都是常态：prefill 的 GEMM 算力密集；decode 每步都要重读全部权重，GEMM 的时间占比同样高，但这不能推出 batch 小。batch=1 时 GEMM 退化为 GEMV、算力利用率极低，方向是增大 batch；服务态多请求合批时 M 维等于 batch，是扁 GEMM 而不是 GEMV，仍受限于显存带宽。区分两者要看 ncu 的 Compute 与 Memory 吞吐（项目二）。
3. **平均每次 launch 换来多少 GPU 时间？** 这个值越小，说明单个 kernel 干的活越少、而提交它的固定成本占比越重。**它是个参考量，不是判据**，因为 launch 与 GPU 执行是重叠的，提交慢不慢要看 GPU 有没有因此空等。真正的判据仍然是第 1 问的占空比和 kernel 之间的间隙分布：**间隙密集且每个都很短，才是 CPU 追不上队列**。本章这个值只有个位数微秒，与第 1 问的低占空比同向，两者合起来才支持 launch 受限的判断，项目三整章都在处理这个问题。

定位结论（实测）：eager wall **1.008** s vs sdpa wall **1.331** s（同 32 tokens，`reports/torch_prof.md`）。GPU 侧 sdpa 有小幅收益（Self CUDA 289→270 ms）；wall 的上升经核对来自主机侧漂移，与 `--attn` 无关，见第九节步骤 5。

七、改一处，然后复测

只改一个变量。基线用的是 `--attn eager`（纯 PyTorch 实现的注意力，会显式物化出 `[heads, q_len, kv_len]` 的注意力分数矩阵），复测换成 `sdpa`（走 `torch.nn.functional.scaled_dot_product_attention`，由后端自行选择融合实现）：

```
python prof_infer.py --model "$CH25_ROOT/models/Qwen2.5-7B-Instruct" \
    --input-len 1024 --decode-steps 31 --attn sdpa --outdir "$CH25_ROOT/profile"
python trace_top.py --trace trace_sdpa_in1024.json --annotation prefill --top 12
python trace_top.py --trace trace_sdpa_in1024.json --annotation decode  --top 12
```

复测这一组同样要留存算子表，文件名换成 `top_sdpa.txt`：

```
python prof_infer.py --model "$CH25_ROOT/models/Qwen2.5-7B-Instruct" \
    --input-len 1024 --decode-steps 31 --attn sdpa --outdir "$CH25_ROOT/profile" \
    | tee "$CH25_ROOT/profile/top_sdpa.txt"
```

除 `--attn` 外所有参数逐字相同，这是复测能比较的前提。

预期方向：prefill 段的注意力相关算子应该合并成更少、更大的 kernel，softmax 与 masked_fill 之类的中间算子消失；decode 段变化会小得多：decode 的 q_len 只有 1，注意力本来就不是热点。**如果 decode 段的数字几乎没动，那不是实验失败，那正是结论**：decode 的优化方向不在注意力实现上，在 batch 和 launch 开销上。

`model.set_attn_implementation("sdpa")` 可以不重新加载模型直接切（transformers v5.14.1 的 `modeling_utils.py` 里有这个方法），但本项目坚持重跑整个脚本，是为了让两次采集的预热、显存状态、随机种子完全一致。**判定标准预先登记**：同一段（prefill 或 decode）的 GPU kernel 累计时间下降，且前三名算子构成发生可解释的变化，才算这次改动被观测到了。

八、必做核心与自选扩展

必做核心是第三到第七步这五步：写出带 `record_function` 分段的采集脚本、跑出 eager 基线的两段算子表、用 `trace_top.py` 算出 GPU 占空比与 launch 密度、按三问定位、换 `--attn sdpa` 复测并填表。少任何一步，闭环就断了。

自选扩展往两个方向做，两个方向产出的东西完全不同：

1. **精确归属**：用 trace 里的 flow 事件把每个 GPU kernel 挂回发起它的 CPU 算子，替换掉第五步的时间窗口近似，顺带就能算出每个 PyTorch 算子的真实 GPU 成本。
2. **批量维度**：把 `--batch` 从 1 扫到 32，观察 decode 段每次 launch 平均换来的 GPU 时间如何随 batch 增长，画出 launch 开销占比的衰减曲线。这条曲线直接决定了项目三里 CUDA Graph 是否值得启用。

九、逐步证据链（面试可复述）

> 把测什么、看到什么、为何如此、改哪一处、复测收益这五步整理成可在面试中复述的证据链。数字来自 `projects/pr1/reports/torch_prof.md` 、 `projects/pr1/bench/` 与 `projects/pr1/profile/`（ `gpu_run_complete` ）。本项目用的是 **torch.profiler**（Chrome trace + 算子表），目录里**没有** nsys / ncu 采集文件。

本节术语

术语（全称）	缩写	含义
torch.profiler	—	PyTorch 自带的采集器。实测产出 `trace_eager_in1024.json` / `trace_sdpa_in1024.json`（Chrome trace）和 `top_*.txt` 算子表。
Self CUDA / Self CPU	—	扣掉子算子之后的净耗时。归因只看 Self：eager 的 Self CUDA **289.1** ms、Self CPU **728.7** ms；sdpa 是 **269.9** / **990.4**。
eager / sdpa	—	两种注意力实现。`--attn eager` 是手写路径；`sdpa` 走 `aten::_flash_attention_forward`。本微基准不能默认 sdpa 更快。
墙钟 wall_s	—	主机侧停表，含 CPU 派活。GPU Self 降了但 wall 涨了，就是 host/同步把收益抵消了。
nsys / ncu	—	本章项目二才用。本项目**没有** `.nsys-rep` / `.ncu-rep`，不要把项目二的 kernel 表抄过来。

步骤 1｜锁定口径（同一长度，只换注意力实现）

这一步在干什么：微基准若一边 32 token、一边 128 token，墙钟差没有意义。实测两组都是 tokens=32。

关于分段数据要说清楚一件事： `prof_infer.py` 打印的 `top_*.txt` 是整段算子表，prefill 与 decode 混在一起；**分段要用第五节的 `trace_top.py --annotation prefill|decode` 从 trace 里切**，这一步你自己跑得出来，交付物里的两段算子排行指的就是它。本章保留下来的实测记录只有整段表与墙钟，没有保存切分后的分段毫秒数，所以下面各步的表里不会出现分段数字。**这不是你不用做，是本章没有可以给你对照的分段基准**：你跑出来的分段结果以自己的机器为准。

- 动作：基线 `--attn eager`，复测 `sdpa`；同一 decode 步数。只信 `torch_prof.md`、`meta_*.json`、`top_*.txt`、`trace_*.json`。

本步产生的文件

文件	来源	看什么	得到什么
`projects/pr1/reports/torch_prof.md`	汇总	`eager.tokens` / `sdpa.tokens`	两组都是 **32**
`projects/pr1/profile/meta_eager.json` / `meta_sdpa.json`	采集脚本	`attn` / `tokens`	eager / sdpa 各一份元数据
`projects/pr1/profile/trace_eager_in1024.json` / `trace_sdpa_in1024.json`	torch.profiler Chrome trace	文件在不在	两份都在（体量大，用 Chrome 打开，不在报告里重算）
（无 nsys / 无 ncu）	目录里没有	—	本项目不采集 Nsight

- 读数：对照变量将是 `--attn`；长度对齐。
- 为什么能推出下一步：统一口径后，先采 eager 墙钟当锚。

步骤 2｜基线（eager 墙钟 + Self 表）

这一步在干什么：先测出当前的端到端耗时。墙钟含 host，必须同时看 Self CUDA，否则无法区分是 GPU 在算还是 CPU 在派活。eager 的 Self CPU（729 ms）已经大于 Self CUDA（289 ms），这条微基准本身就偏 host。

- 动作：eager 微基准，落 JSON 与 top 表。

本步产生的文件

文件	来源	看什么	得到什么
`projects/pr1/bench/eager.json`	微基准	`wall_s` / `tokens`	wall **1.0077** s（报告四舍五入 **1.008**）、tokens **32**
`projects/pr1/profile/top_eager.txt`	profiler 表	Self CUDA 第 1 行 + 表尾合计	`aten::mm` **56.11%**（162.2 ms，3616 次）；Self CUDA total **289.078** ms；Self CPU total **728.677** ms
`projects/pr1/profile/trace_eager_in1024.json`	Chrome trace	存在即可	给步骤 3 对照用
（无 nsys）	未采	—	kernel 名来自 profiler 表，不是 `cuda_gpu_kern_sum`

- 读数：热点在 `aten::mm` / `nvjet_*` GEMM，不在 attention 名。墙钟 1.01 s 是锚。
- 为什么能推出下一步：有了 Self 表，才能看 sdpa 是 GPU 变慢还是 CPU 变慢。

步骤 3｜剖析（GPU 侧略有下降，墙钟反而上升）

这一步在干什么：切 sdpa 之后，若只看 wall 1.33 s，会判定 sdpa 更慢。打开 top 表：Self CUDA **289.1→269.9** ms（下降 6.6%），Self CPU **728.7→990.4** ms（上升 36%）。`aten::_flash_attention_forward` 出现了，但只占 Self CUDA **3.24%**（8.7 ms）。GPU 侧的结论已经清楚：sdpa 使 GPU 时间小幅下降，且注意力在这个形状下不是热点。Self CPU 的上升归属于谁，要到步骤 5 用未改动的算子核对之后才能判定，此处不能直接归给 sdpa。

- 动作：同一 32 token 跑 sdpa，对照两份 top 表。

本步产生的文件

文件	来源	看什么	得到什么
`projects/pr1/bench/sdpa.json`	微基准	`wall_s`	**1.331** s（同 32 tokens）
`projects/pr1/profile/top_sdpa.txt`	profiler 表	`aten::mm` / `_flash_attention_forward` / 表尾	mm **59.51%**（160.6 ms）；flash **3.24%**（8.7 ms，896 次）；Self CUDA **269.906** ms；Self CPU **990.377** ms
`projects/pr1/profile/trace_sdpa_in1024.json`	Chrome trace	存在即可	可与 eager trace 并排看，报告未切出分段毫秒
（无 nsys / 无 ncu）	未采	—	没有 Occupancy / 带宽计数器；四象限不能编

- 读数：GPU Self 下降 6.6%，wall 反而上升 32%，两者方向相反，差异在 host 侧。
- 为什么能推出下一步：两组 tokens 相同，差异不来自解码步数；host 侧的差异能否归因到 `--attn`，要在步骤 5 用未改动的算子核对。

步骤 4｜只改一处（只改 --attn）

这一步在干什么：两组 tokens 都是 32，唯一差别是 eager→sdpa。这样 wall +0.32 s 不能归因于解码步数不同。

- 动作：只改 `--attn`。

本步产生的文件

文件	来源	看什么	得到什么
`projects/pr1/profile/meta_eager.json` / `meta_sdpa.json`	元数据	`attn` 字段	`eager` vs `sdpa`，`tokens` 都是 32
`projects/pr1/bench/closed_loop_summary.json`	摘要	两组结构	只有 eager / sdpa，没有第三处改动
（无 nsys）	未采	—	不靠 Nsight 证明单变量

- 读数：只改了一个参数，单变量成立。
- 为什么能推出下一步：参数只改了一处，不等于测量环境没有变；按墙钟验收之前，先核对 host 侧的变化是否来自 `--attn`。

步骤 5｜复测与收益判定（GPU 侧成立，wall 侧本次不成立）

这一步在干什么：预期是切换 sdpa 后墙钟与 GPU Self 时间同时下降。实测 GPU Self 下降（289→270 ms），wall 却上升（1.008→1.331 s）。判定之前先做一项核对：**与 `--attn` 无关的算子，两组之间的单次 CPU 时间是否相同**。矩阵乘、张量拼接、求均值三类算子不经过注意力实现，换 `--attn` 不应改变它们的 CPU 开销。

算子（与 `--attn` 无关）	Self CPU eager→sdpa	变化	Self CUDA eager→sdpa	变化
`aten::mm`	47.5→81.3 ms	+71%	162.2→160.6 ms	−1.0%
`aten::addmm`	44.0→73.6 ms	+67%	28.3→28.3 ms	−0.1%
`aten::cat`	22.1→36.8 ms	+67%	11.6→11.5 ms	−0.4%
`aten::mean`	13.5→24.3 ms	+79%	6.6→6.4 ms	−2.4%

来源：`top_eager.txt` / `top_sdpa.txt` 对应行；四个算子的调用次数两组相同。

读法：GPU 侧变化都在 2.4% 以内（四项中三项在 1% 以内），说明两次采集的 GPU 工作量基本一致；CPU 侧却整体上升 67%～79%。换注意力实现不会让矩阵乘的单次 CPU 时间增加 70%，所以 Self CPU 的上升来自两次运行之间主机侧执行速度的变化，与 `--attn` 无关，本次未能定位具体原因。结论分两层：

- **GPU 侧成立**：sdpa 使 Self CUDA 下降 6.6%（289→270 ms）。换完之后注意力算子只占 3.24%，加上总量只降了 6.6%，两个数合起来说明在 7B、1024 输入、batch=1 这个形状下，换注意力实现的收益上限本来就低。**注意 3.24% 是改动之后的占比，不能反过来当成"改之前就知道不该改"的依据**；改之前 eager 组的注意力分散在 `aten::bmm`、softmax、mask 几行里，要自己加起来才有可比的数。
- **wall 侧本次不成立**：wall 的上升由主机侧变化主导，单轮对照无法把它与 `--attn` 的影响分离。要下 wall 侧的结论，必须 eager / sdpa 交替各跑 3～5 轮，报区间不报单点。

这项核对本身是一条通用方法：**改动一个变量之后，先看不该变的算子有没有变**。它们变了，说明测量环境变了，结论要先搁置。

- 动作：用 wall + Self 两列判定，并用未改动算子核对漂移；不补 nsys。

本步产生的文件

文件	来源	看什么	得到什么
`projects/pr1/reports/torch_prof.md`	汇总	两组 `wall_s`	1.008 vs 1.331
`projects/pr1/profile/top_eager.txt` / `top_sdpa.txt`	算子表	Self CPU / Self CUDA 合计，以及 `mm` / `addmm` / `cat` / `mean` 四行	CPU 729→990；CUDA 289→270；四个未改动算子的 Self CPU 上升 67%～79%
`projects/pr1/bench/closed_loop_summary.json`	摘要	`status`	`gpu_run_complete`
（无 nsys / 无 ncu）	未采	—	收益判定不依赖 Nsight

- 读数：GPU 侧 sdpa 有小幅收益；wall 侧的差异来自主机侧漂移，本次不能作为 sdpa 更慢的依据。
- 为什么到这里可以停：面试能复述：先看 wall，再拆 Self CPU / CUDA，再用未改动算子核对两次运行是否可比。本例 GPU 侧可比、host 侧不可比，所以 wall 侧结论保留到多轮复测之后。

一句话总结：torch.profiler 闭环里 sdpa 使 Self CUDA 下降 6.6%（289→270 ms），注意力算子只占 3.24%；wall 1.01→1.33 s 的上升由主机侧漂移主导（未改动算子的 Self CPU 同步上升 67%～79%），单轮对照不能把它归给 sdpa；本项目没有 nsys/ncu。

十、排错表

现象	原因	处理
表里没有 Self CUDA 那几列	`activities` 里漏了 `ProfilerActivity.CUDA`，或采集窗口内没有 GPU 活动	补上 CUDA；确认预热之后才进 profiler
打出 `torch_dtype is deprecated`	transformers v5 已改名	改用 `dtype=`
不写 dtype，显存直接翻倍	v5 的 `from_pretrained` 默认 `dtype="auto"`，按权重文件精度加载；有的仓库存的是 fp32	显式写 `dtype=torch.bfloat16`
`self_cuda_time_total` 打 FutureWarning	该属性在 torch 2.11 已 `@deprecated`	换成 `self_device_time_total`；作为 sort_by 字符串两者等价
total 列相加远超采集窗口	total 含子算子，会重复计入	归因一律看 Self 列
trace 文件几百 MB 打不开	采了太多步，或开了 `with_stack=True`	减少 `--decode-steps`；长期循环改用 `schedule(wait, warmup, active, repeat)` 设定采集窗口
`trace_top.py` 报没有名为 decode 的标注	采集脚本里没加 `record_function`，或名字对不上	名字要逐字一致
自己加的计时比 profiler 报的小一个量级	没 `torch.cuda.synchronize()` 就停表，量到的是 launch 时间	计时前后都同步；或只采信 profiler 的设备侧时间

十一、面试深挖点

下面的参考答法里带着本章的实测数。 这些数是在 H100 上跑出来的，换卡、换负载都会变；照搬去讲，对方一追问负载口径就答不上来。哪些能讲、哪些要自己跑出来，见全章末节“把这三个项目写进简历之前”。

Q1 | `torch.profiler` 表里的 Self CUDA 和 CUDA total，有什么区别?

考察点：有没有真的读过这张表。只跑过一次的人会把两列当同一个东西。

参考答法：Self 是扣掉子算子之后的净耗时，total 含子算子。PyTorch 的算子是嵌套的，`aten::linear` 内部调 `aten::addmm`，同一段 GPU 时间会同时记在两者的 total 里，所以 total 列相加可以远超采集窗口的实际长度。**同一个算子的不同层级之间，归因和排序用 Self 列**；total 列的用处是看某个高层算子的整体成本，比如整个 attention 模块的总耗时。

还要说清 Self 列自己的边界：这张表里 `aten::` 开头的 CPU 算子行和 `nvjet_tst_*` 这类 kernel 行是混排的，CPU 算子行的 Self CUDA 是它拉起的 kernel 的时间，两类行之间同样重复计数。本章实测表里两类分别合计 93.7% 与 76.1%，混加超过 100%。**单行的百分比都对，分母是表尾那个只累加 kernel 行的 total；要相加就必须在同一类行里加。** 另外要说清这两列都是 GPU 侧时间，和 CPU 侧那几列不是一回事。要判断瓶颈是否在 CPU 侧，先算 GPU 繁忙时间占墙钟的比例（本例 Self CUDA 合计 289 ms 对 wall 1.008 s，约 29%），比例低才说明 GPU 在等 CPU；只比较 Self CPU 与 Self CUDA 两列的大小不够，CUDA 是异步的，两者本来就重叠。

Q2｜prefill 和 decode 为什么必须分开 profile?

考察点：对推理两阶段特性的理解，而不是背“prefill 是计算密集、decode 是访存密集”这句话。

参考答法：算子构成、kernel 形状、瓶颈类型都不同。prefill 一次处理整个 prompt，矩阵乘两个维度都很大，算力利用率高，热点在 GEMM 和注意力；decode 每步只出一个 token，同样的权重矩阵要重新读一遍；batch=1 时只乘一行，GEMM 退化成 GEMV，算力用不到几个百分点，服务态合批后是 M 维等于 batch 的扁 GEMM，仍然受限于显存带宽；而每步要发的 kernel 数量和 prefill 一样多，launch 开销的相对占比被放大数倍。混在一起统计，长的那一段会把另一段完全淹没：本例 1024 token 的 prefill 配 31 步 decode，decode 每步都要重读全部权重，31 步累计的 GPU 时间与 prefill 同一量级甚至更高，短的那一段就被掩盖了。分段手段就是 `record_function` 加标注，再按标注切 trace。

Q3｜采集窗口怎么选？为什么不能从头采到尾?

考察点：有没有处理过 trace 体积失控。

参考答法：三个理由。

1. 第一次前向包含 kernel 选型、JIT 编译、显存池扩张这些一次性开销，不预热就采，排行榜前几名全是一次性成本。
2. trace 体积随采集时长线性涨，几百 MB 的 trace 前端根本打不开，`with_stack=True` 还会再放大一个量级。
3. profiler 本身有开销，采得越久扰动越大。

做法分两种：短的有界任务，手动预热几轮再进 profiler；长期运行的服务或训练循环，用 `torch.profiler.schedule` 设定采集窗口，wait 阶段完全不采集是零开销的，warmup 采但丢弃，active 才留下，repeat 控制循环几轮。

Q4｜你定位到 decode 阶段 GPU 占空比只有 40%，接下来怎么办?

考察点：能不能从一个观测量走到具体动作，而不是停在 GPU 利用率不足。

参考答法：先分清空闲是哪一种。看 kernel 之间的间隙分布：间隙密集且每个都很短，是 launch 受限，CPU 来不及发指令，解法是减少 kernel 数量（算子融合）或消除 launch 开销（CUDA Graph）；间隙集中在少数几个大缺口，多半是同步点或 CPU 侧的数据准备阻塞，解法是找到那次同步。再看单步的绝对耗时和 batch，这里要分开两个维度：batch=1 时单个 GEMV kernel 占不满 SM，这是空间维的利用率不足，它可能仍然接近显存带宽上限、GPU 在时间维上是忙的；占空比低说的是时间维，是 kernel 之间有空隙。前者的动作是提高并发，后者的动作是减少 launch 次数或找到同步点。**顺序不能反**：先确认是没有 kernel 在执行，还是 kernel 执行得慢，再决定是调度问题还是 kernel 问题。

Q5｜（陷阱题）我用 profiler 测出 `aten::linear` 的 CUDA total 占了 80%，所以瓶颈在矩阵乘，对吗?

考察点：会不会被一个看起来很大的百分比带走。题面用的是 total 列，而这一列的语义决定了这个 80% 说明不了它想说明的事。

参考答法：不能这么下结论，问题出在用错了列。CUDA total 含子算子，`aten::linear` 名下的时间包含它调用的 `aten::addmm` 乃至更下层的时间，同一段 GPU 时间被重复计入了多层，所以各行 total 相加会超过 100%，80% 这个数本身不代表它占了总时间的八成。要判断矩阵乘是不是瓶颈，应该看 Self CUDA 列，但有一个前提要一起说出来：**这张表里 CPU 算子行与 kernel 行混排，两类之间 Self 列同样重复计数**，`aten::mm` 的 Self CUDA 就是它拉起的那批 `nvjet_tst_*` kernel 的时间。所以单行百分比可以用，跨类相加不行；本章实测表两类分别合计 93.7% 与 76.1%。最干净的做法是直接看 GPU 侧的 kernel 汇总，只在 kernel 这一类行里求和；kernel 之间不存在嵌套，同一条流上加起来就是真实的 GPU 繁忙时间，跨流并发要按时间轴取并集。而且即使 Self 列确认了 GEMM 占比最高，也还要再问一句：这是 prefill 段还是 decode 段？decode 段的 GEMM 占比高是常态，每步都要重读全部权重，它不能推出 batch 小；要判断是不是并行度不足，看 ncu 的 Compute 与 Memory 吞吐和 GEMM 的 M 维 tile。这两步都跳过，就会得出更换更快的 GEMM 实现这种方向完全错误的结论。

项目二　对 vLLM / SGLang 生产服务做 Nsight 端到端 Profiling

这个项目要回答的问题：一个已经在跑的推理服务，吞吐上不去，该调哪个参数？

本项目的结论：并发 32 这个负载下，vLLM 与 SGLang 的主 GEMM 指标几乎一致（Memory 约 **86%**、Compute 只有 **12%**），两边都受限于显存带宽，不是算力不够。真正形成约束的也不是并发上限：默认 `max_num_seqs=1024` 远大于实际并发，调它不会改变一步里入批的条数；形成约束的是单步 token 预算。但把这个预算从 8192 翻到 16384 之后，vLLM 吞吐没有提升（**−0.7%**，单轮，在噪声范围内）、首包 P99 差 **5.7%**；SGLang 吞吐降 **5.4%**、首包 P99 差 **56%**。**按预先登记的判据，两边都不采纳。**

一个改动被否决，和一个改动被采纳，是同样完整的闭环。能把“我改了、按同口径测了、按事先定好的判据否掉了”讲清楚，比“我调了个参数快了百分之几”更经得起追问。

读完能讲清：

1. nsys 与 ncu 分别回答什么问题，为什么顺序不能反；
2. 常驻服务为什么必须先设采集窗口再采；
3. 怎么从 kernel 的 Compute 与 Memory 吞吐判断瓶颈类型；
4. 怎么在动手改之前就把判定标准定下来。

一、目标与交付物

对一个**常驻的**推理服务做端到端 GPU 性能分析：先用 Nsight Systems 看时间线定位开销最高的几个 kernel，再用 Nsight Compute 对准这几个 kernel 看 Occupancy 与 Warp 停顿，最后根据结论改一处并复测验证。

交付物三样：

1. 一份 nsys 采集记录（含 kernel 汇总 CSV）；
2. `ncu_top.py`：把 ncu 的竖表 CSV 透视成一行一个 kernel 的横表；
3. 走完压测基线 → nsys 定位 → ncu 下钻 → 改一处 → 复测这一完整闭环的记录。

做到什么算完成（五条缺任何一条，这个项目就没走完闭环）：

1. 基线是同口径、可复现的一组数；
2. nsys 汇总能指出累计耗时前几名的 kernel；
3. ncu 对这几个 kernel 给出 Compute 与 Memory 吞吐，并据此判定瓶颈类型；
4. 改动前后有同口径的两组端到端数；
5. 判定标准在改动之前就已写下。

二、两个工具分别回答什么

nsys 是时间线，ncu 是显微镜。 nsys 采集的是 kernel 的时间线信息：起止时刻、所在的流、kernel 名与持续时长，能回答 GPU 有没有空闲、哪个 kernel 累计耗时最高、CPU 与 GPU 的等待关系；它**不采集**硬件性能计数器，回答不了“这个 kernel

为什么慢”。ncu 采单个 kernel 的硬件计数器（SM 吞吐、显存吞吐、Occupancy、Warp 停顿原因），代价是要把 kernel 反复重放以采全计数器，**不可能对整个服务跑一遍**。

顺序不能反：先 nsys 拿到哪几个 kernel 耗时占比最高，再用 ncu 只针对这几个采。另外，**常驻服务不能从头采到尾**：服务不退出，报告会大到不可用，node 级的 CUDA Graph 追踪还会进一步放大，必须设定采集窗口。

三、压测基线

先有一个可比的数，否则后面改一处没有参照。

起 vLLM：

```
source "$CH25_ROOT/.venv-vllm/bin/activate"
vllm serve "$CH25_ROOT/models/Qwen2.5-7B-Instruct" \
    --served-model-name Qwen2.5-7B-Instruct \
    --host 0.0.0.0 --port 8000 \
    --max-model-len 8192 --gpu-memory-utilization 0.88 \
    2>&1 | tee "$CH25_ROOT/bench/serve_baseline.log"
```

另一个终端压：

```
source "$CH25_ROOT/.venv-vllm/bin/activate"
cd "$CH25_ROOT/bench"
vllm bench serve --model Qwen2.5-7B-Instruct --base-url http://127.0.0.1:8000 \
    --dataset-name random --random-input-len 1024 --random-output-len 128 \
    --num-prompts 256 --max-concurrency 32 --request-rate inf \
    --seed 0 --ignore-eos --percentile-metrics ttft,tpot,itl,e2el \
    --metric-percentiles 50,95,99 --save-result --result-filename baseline.json
```

`--gpu-memory-utilization` 在 v0.26.0 的默认值是 **0.92**，这里写 0.88 是给 profiler 留余量，nsys 和 ncu 自己也要占显存。`--dataset-name random` 是纯合成的，不下载任何文件，断网也能跑。`--ignore-eos` 让每条请求都生成满 128 个 token，否则输出长度不一，TPOT 没法横向比。

基线记录（实测，`reports/nsys_vllm.md`）

指标	基线（默认 btok=8192）	复测 `--max-num-batched-tokens 16384`
输出吞吐 tok/s	**2604.7**	**2585.8**（−0.7%）
TTFT P99 (ms)	**838.9**	**886.3**（+5.7%）
TPOT mean (ms)	**8.77**	**8.68**
ITL P99 (ms)	**114.0**	**62.4**

> 早先一次缩水负载（prompts=32 / conc=8 / in512）的结果，且改一处那一步误用了 `--enforce-eager`，不作基线。本表以同口径网格为准。采纳判据是吞吐提升、且 TTFT P99 劣化不超过 10%，两条同时满足才采纳。本次两条都没满足，**不采纳** btok×2。

SGLang 同口径（实测版本 0.5.6；`reports/nsys_sglang.md`）

指标	基线（默认 chunked_prefill=8192）	复测 `--chunked-prefill-size 16384`
输出吞吐 tok/s	**2515.2**	**2380.2**（−5.4%）
TTFT P99 (ms)	**801.2**	**1249.8**（+56.0%）
TPOT mean (ms)	**9.28**	**9.10**
ITL P99 (ms)	**21.4**	**27.7**

> 参数名不同，机制相同：H100 默认 chunked_prefill=8192，conc=32 时 `max_running_requests=4096` 不是约束。翻倍之后否决更明确（首包延迟大幅劣化）。`/start_profile + profile_by_stage` 已落到 DECODE / EXTEND 两份 torch trace。

四、用 nsys 设定采集窗口采一次

vLLM V1 是多进程架构，服务常驻不退出。官方给的做法是让**客户端**来触发采集窗口。

先停掉第三节那个压测用的服务（在它的终端里 `Ctrl-C`），否则 8000 端口被占。然后服务端重起一次，带上 profiler 配置：

```
source "$CH25_ROOT/.venv-vllm/bin/activate"
cd "$CH25_ROOT/profile"
nsys profile \
    --trace-fork-before-exec=true \
    --cuda-graph-trace=node \
    --capture-range=cudaProfilerApi \
    --capture-range-end repeat \
    --gpu-metrics-devices cuda-visible \
    -o "$CH25_ROOT/profile/vllm_baseline" \
    vllm serve "$CH25_ROOT/models/Qwen2.5-7B-Instruct" \
        --served-model-name Qwen2.5-7B-Instruct \
        --host 0.0.0.0 --port 8000 --max-model-len 8192 \
        --gpu-memory-utilization 0.88 \
        --profiler-config '{"profiler": "cuda"}' \
    2>&1 | tee "$CH25_ROOT/bench/serve_baseline.log"
```

所有 `vllm serve` 都建议这样接一个 `tee`。 第七节要回到启动日志里读引擎实际用的 `max_num_batched_tokens`，项目三要读 `Graph capturing finished` 那一行，日志没留就得重启一次服务重采。

客户端只发 2 条请求，够抓一个窗口就行（**新终端，记得先 `cd` + `export` + 激活 venv**）：

```
source "$CH25_ROOT/.venv-vllm/bin/activate"
vllm bench serve --model Qwen2.5-7B-Instruct --base-url http://127.0.0.1:8000 \
    --dataset-name random --random-input-len 1024 --random-output-len 128 \
    --num-prompts 2 --profile
```

客户端会打印 `Starting profiler...` / `Profiler started` 与 `Stopping profiler...` / `Profiler stopped`。

这时报告还没有落盘。 nsys 包着的是一个不退出的常驻服务，`.nsys-rep` 要等被包的进程结束才会写出来。看到 `Profiler stopped` 之后，回到服务端那个终端按一次 `Ctrl-C` 停掉服务，nsys 会打印 `Generating '...nsys-rep' file` 并落盘，之后才能做下一步的导出。直接去读 `.nsys-rep` 会发现文件不存在，这不是采集失败。

五个开关各自在解决什么：

开关	解决什么
`--capture-range=cudaProfilerApi`	服务常驻，不能全程采集。让应用内的 `cudaProfilerStart/Stop` 来设定采集窗口。`--profiler-config '{"profiler": "cuda"}'` 就是让 vLLM 去调这一对 API，客户端的 `--profile` 是触发器
`--capture-range-end repeat`	一个窗口结束后继续等下一个，而不是直接收工。取值还有 `none` / `stop` / `stop-shutdown`（默认）
`--cuda-graph-trace=node`	decode 走 CUDA Graph。默认的 `graph` 粒度在时间线上只能看到整图一条，看不到图里每个 kernel；`node` 才能看到，代价是开销明显变大
`--trace-fork-before-exec=true`	官方 vLLM profiling 文档里带的。它**不是**采集子进程的开关：nsys 默认 `--sample=process-tree` 本来就覆盖进程树；它抓的是 fork 到 exec 之间那一小段，官方明确警告这依赖未定义行为、可能让应用崩溃或死锁。运行不稳定就先去掉
`--gpu-metrics-devices cuda-visible`	**这个默认值是 `none`**（官方 CLI 手册），不加就采不到 SM active、DRAM 带宽这些采样型 GPU 指标，时间线上只有 kernel 条。取值还有 `all` / `<id,…>` / `help`

`--profiler-config` 的字段来自 `vllm/config/profiler.py` 的 `ProfilerConfig`：`profiler` 取值是 `"torch"` 或 `"cuda"` 两选一（`ProfilerKind = Literal["torch", "cuda"]`），默认 `None`。要出 torch trace 而不是给 nsys 设定采集窗口，就换成：

```
--profiler-config '{"profiler": "torch", "torch_profiler_dir": "'"$CH25_ROOT"'/profile"}'
```

三处易错：`torch_profiler_dir` **只在** `profiler` 是 `"torch"` 时可用，配 `"cuda"` 会在校验阶段直接 raise（同文件 `_validate_profiler_config`）；反过来 `profiler` 是 `"torch"` 而没给目录也会 raise。目录的文档串写着必须是绝对路径，但校验函数实际是对非 URI 路径调 `os.path.abspath` 做转换而不是拒绝：**相对路径不会报错，会按启动进程的工作目录展开**，多进程下很容易落到意料之外的地方，所以还是显式传绝对路径。`$CH25_ROOT` 展开后就是绝对路径，既满足要求，文档里又没写死任何机器专属路径。

另外，**老教程里的 `VLLM_TORCH_PROFILER_DIR` 环境变量已经不存在了**：v0.26.0 的 `vllm/envs.py` 全文检索 `TORCH_PROFILER` 零匹配。

`/start_profile` 与 `/stop_profile` 这两个端点定义在 `vllm/entrypoints/serve/profile/api_router.py`，**只在 `profiler_config.profiler` 不为 None 时才挂载**，并会打一条一次性警告 "Profiler with mode '%s' is enabled in the API server. This should ONLY be used for local development!"。也就是说：不带 `--profiler-config` 启动的服务，这两个端点是 404，客户端的 `--profile` 会失败。这一点客户端的帮助文本也写了：“Use vLLM Profiling. --profiler-config must be provided on the server.”

出 kernel 汇总：

```
nsys stats --report cuda_gpu_kern_sum --format csv \
    --output "$CH25_ROOT/profile/kern_baseline" \
    "$CH25_ROOT/profile/vllm_baseline.nsys-rep"
head -15 "$CH25_ROOT/profile/kern_baseline_cuda_gpu_kern_sum.csv"
```

报告名以所装版本 `nsys stats --help-reports` 的实际输出为准。当前官方文档列出的是下划线形式（`cuda_gpu_kern_sum` / `cuda_gpu_trace` / `cuda_api_sum`）；老的 `gpukernsum` 这类写法在现役文档里已经查不到，但官方也没有明说移除，所以别直接断言它失效。

SGLang 侧的等价做法（换成 `.venv-sglang`）：

```
source "$CH25_ROOT/.venv-sglang/bin/activate"
export SGLANG_TORCH_PROFILER_DIR="$CH25_ROOT/profile"
python -m sglang.launch_server \
    --model-path "$CH25_ROOT/models/Qwen2.5-7B-Instruct" \
    --host 0.0.0.0 --port 30000 --context-length 8192 \
    --mem-fraction-static 0.85 --enable-metrics \
    --enable-layerwise-nvtx-marker
```

```
curl -X POST http://127.0.0.1:30000/start_profile -H "Content-Type: application/json" \
  -d '{"num_steps": 10, "start_step": 5,
       "profile_by_stage": true, "activities": ["CPU","GPU"]}'
# 期间发几条请求。给了 num_steps 会自己停，下面这条可以不调
curl -X POST http://127.0.0.1:30000/stop_profile
```

请求体对应 v0.5.16 `python/sglang/srt/managers/io_struct.py` 的 `ProfileReq`。这一章真正要用的只有四个字段，其余保持默认即可：

字段	本例取值	干什么
`profile_by_stage`	`true`	**本章最关键的开关。** 把 prefill 和 decode 分成两份 trace 落盘，项目一里手动按阶段切分那一步可以省掉
`num_steps`	`10`	采满 10 个 scheduler step 就自动停，不必再调 `/stop_profile`
`start_step`	`5`	跳过前 5 步（预热 / 编译），从稳态开始采
`activities`	`["CPU","GPU"]`	采哪些活动。还能加 `"MEM"`、`"RPD"`，本章用不到

其余字段（`output_dir` / `with_stack` / `record_shapes` / `profile_id` / `merge_profiles` / `profile_prefix` / `profile_stages`）是落盘路径和 torch profiler 细节，不改也能跑通；要对齐目录时再设 `output_dir`，否则走环境变量 `SGLANG_TORCH_PROFILER_DIR`。

`--enable-layerwise-nvtx-marker` 的帮助文本是 “Enable layerwise NVTX profiling annotations for the model.”（`server_args.py`），打开后时间线上每一层会有 NVTX 区间，配合 ncu 的 `--nvtx-include` 就能只采某一层的 kernel。`--enable-metrics` 默认 **False**，不加这个 flag，`/metrics` 端点根本不会挂载。

五、用 ncu 对准开销最高的几个 kernel

从 nsys 的 kernel 汇总里挑出累计时间前 2～3 名，只对它们采：

```
source "$CH25_ROOT/.venv-vllm/bin/activate"
cd "$CH25_ROOT/profile"
ncu --target-processes all \
    --graph-profiling node \
    --kernel-name "regex:nvjet|FlashAttnFwd" \
    --launch-skip 200 --launch-count 20 \
    --section SpeedOfLight --section Occupancy \
    --section WarpStateStats --section LaunchStats \
    -o "$CH25_ROOT/profile/ncu_baseline" \
    vllm serve "$CH25_ROOT/models/Qwen2.5-7B-Instruct" \
        --served-model-name Qwen2.5-7B-Instruct \
        --host 0.0.0.0 --port 8000 --max-model-len 8192 \
        --gpu-memory-utilization 0.80
```

起来之后另开终端发请求，采集满 20 次 launch 后回到 ncu 那个终端 `Ctrl-C` 结束：

```
source "$CH25_ROOT/.venv-vllm/bin/activate"
vllm bench serve --model Qwen2.5-7B-Instruct --base-url http://127.0.0.1:8000 \
    --dataset-name random --random-input-len 1024 --random-output-len 128 \
    --num-prompts 32 --max-concurrency 8
```

ncu 挂载下服务会慢几十倍，这是正常的：它要把每个目标 kernel 重放多次以采全计数器。服务长时间没有响应不等于挂掉了，先看 ncu 终端有没有在滚 kernel 名。

`--kernel-name` 的正则必须来自你自己第四步 `cuda_gpu_kern_sum.csv` 第一列里的真实名字。本章实测栈（CUDA 13、cuBLASLt、FlashAttention 3）上 GEMM 是 `nvjet_tst_*`，attention 是 `FlashAttnFwdSm90`；旧栈上常见的 `gemm`、`BatchDecodeWithPagedKV` 在本章版本的两份 kernel 汇总里一个都不会命中，照抄会让 ncu 等到超时也采不到任何 kernel。

本章实测的 ncu 数据没有挂在常驻服务上采，而是用单批离线复现同样的 kernel 形状后采集：vLLM 侧 eager latency、SGLang 侧 `bench_one_batch`，均为 batch=32 / in1024，`--launch-skip 2500 --launch-count 24`。原因是 ncu 的 kernel 重放与常驻服务的请求调度互相干扰，采集窗口难以稳定。**两份数据的 kernel 属于同一族，但不是同一个 kernel，这一点要说准。** 服务态 nsys 汇总（`kern_baseline_cuda_gpu_kern_sum.csv`）的 `Name` 列第一行是 `nvjet_tst_192x16_...`（Time 21.2%、Instances 3444）；单批离线采到的主 GEMM 是 `nvjet_tst_192x32_...`。两者都是 cuBLASLt 的 nvjet GEMM，差在 tile 的第二个数：离线跑固定 batch 32，服务态是连续批处理下的实际合批规模。注意力那一侧同理，两边都是 `FlashAttnFwdSm90`。

能对应回服务态的是瓶颈类型，不是具体数值：两份都落在 Memory 远高于 Compute 的同一区间，说明 decode 主路径受限于显存带宽这个结论在服务态成立；但单批那份的 Duration、Instances 不能当作服务态的数来引用。上面挂在 `vllm serve` 上的命令是标准路径。离线单批那条的命令如下，它不需要起服务、也不受请求调度干扰，**拿不到稳定采集窗口时优先用这条**：

```
# vLLM 侧：单批 latency，关掉 CUDA Graph 以便逐 kernel 采
source "$CH25_ROOT/.venv-vllm/bin/activate"
ncu --target-processes all \
    --kernel-name "regex:nvjet|FlashAttnFwd" \
    --launch-skip 2500 --launch-count 24 \
    --section SpeedOfLight --section Occupancy \
    --section WarpStateStats --section LaunchStats \
    -o "$CH25_ROOT/profile/ncu_baseline" \
    vllm bench latency --model "$CH25_ROOT/models/Qwen2.5-7B-Instruct" \
        --input-len 1024 --output-len 8 --batch-size 32 \
        --num-iters-warmup 2 --num-iters 5 --enforce-eager
```

本章实测记录了 batch 32、输入 1024、eager、`--launch-skip 2500 --launch-count 24` 这几项，**没有记录当时用的 `--kernel-name` 过滤条件，也没有记录 `--output-len`**；上面命令里的正则与 `--output-len 8` 是按本章实测栈的 kernel 名重新给出的可用写法，不是实测当时的原始命令。对采集 GEMM 与 attention kernel 的 SpeedOfLight 指标来说，输出长度只影响采到的 decode 步数，不影响单个 kernel 的吞吐读数。

两条路径按自己的环境选一条：能拿到稳定窗口就挂服务采，拿不到就用单批离线采。

六个开关的依据（全部出自 Nsight Compute CLI 官方手册）：

开关	说明
`--target-processes all`	取值只有 `application-only` 和 `all`。vLLM 是多进程，模型跑在 worker 子进程里，不写 `all` 就一个 kernel 也采不到
`--graph-profiling node`	取值 `node`（逐个 kernel 节点采）或 `graph`（整张图当一个 workload）。decode 走 CUDA Graph，要看单 kernel 必须用 `node`
`--kernel-name "regex:..."`	支持精确名或 `regex:` 前缀的部分匹配。不加过滤，ncu 会去采服务启动时的每一个 kernel，实际上会长时间无输出
`--launch-skip` / `--launch-count`	前者跳过前 N 次 launch（默认 0），后者限制采集次数。跳过前面是为了避开预热和图捕获阶段
`--section`	可多次给。四个 Section 的官方说明分别是：SpeedOfLight “High-level overview of the throughput for compute and memory resources”、Occupancy “活跃 warp 与理论最大 warp 之比”、WarpStateStats “Analysis of the states in which all warps spent cycles”、LaunchStats “Summary of the configuration used to launch the kernel”
（未用）`--set full`	`--set` 取值含 `basic`（默认）与 `full`。`full` 采全部 Section，开销大得多，只在已经锁定单个 kernel 之后才值得用

指标名不要背，两套命名不通用。 `--page details` 导出的 CSV 用显示名称（`Duration`、`Compute (SM) Throughput`、`Memory Throughput`、`Achieved Occupancy`）；`--page raw` 与 `--query-metrics` 列出的是内部指标名

（`sm__throughput.avg.pct_of_peak_sustained_elapsed` 这一类）。本章按 Section 采、按 `--page details` 导出，`--show` 的子串要按显示名称写；先用 `head -30` 查看 CSV 的 `Metric Name` 列再定。本章正文只按 Section 采，不硬编码指标字符串，就是为了避免文档上抄来的指标名在这台卡上并不存在。

导出成 CSV 再透视：

```
ncu --import "$CH25_ROOT/profile/ncu_baseline.ncu-rep" --csv --page details \
    > "$CH25_ROOT/profile/ncu_baseline.csv"
```

这份 CSV 是竖表，一个 kernel 的每个指标各占一行，直接看很费劲。下面这个脚本把它透视成一行一个 kernel 的横表；**先建脚本，再执行透视**。

新建 `$CH25_ROOT/profile/ncu_top.py`：

```
"""把 ncu 的 CSV 输出竖表转成"一行一个 kernel"的横表，按指定指标排序。纯标准库。
用法: ncu --import prof.ncu-rep --csv --page details > ncu.csv
      python ncu_top.py --csv ncu.csv --sort duration --top 10
不重算任何指标，只做透视与排序：ncu 已经算好的值原样转述。
"""
import argparse, csv, sys


def sniff(header):
    """ncu 各版本列名有出入，按子串定位，定位不到就把表头打出来让人自己核。"""
    idx = {}
    for want, keys in (("kernel", ("kernel name",)),
                       ("metric", ("metric name",)),
                       ("value", ("metric value",)),
                       ("unit", ("metric unit",))):
        for i, col in enumerate(header):
            if col.strip().lower() in keys:
                idx[want] = i
                break
    missing = [k for k in ("kernel", "metric", "value") if k not in idx]
    if missing:
        print("认不出这份 CSV 的列名，缺少: " + ", ".join(missing), file=sys.stderr)
        print("实际表头: " + " | ".join(header), file=sys.stderr)
        print("用 `head -3 ncu.csv` 看一眼，再把 sniff() 里的关键字改成你这版 ncu 的列名。",
              file=sys.stderr)
        raise SystemExit(2)
    return idx


def to_float(s):
    s = (s or "").replace(",", "").strip()
    try:
        return float(s)
    except ValueError:
        return None


def main():
    ap = argparse.ArgumentParser()
    ap.add_argument("--csv", required=True)
    ap.add_argument("--top", type=int, default=10)
    ap.add_argument("--sort", default="duration",
                    help="用于排序的指标名的子串，默认 duration")
    ap.add_argument("--show",
                    default="duration,compute (sm),memory throughput,dram throughput",
                    help="要展示的指标名子串，逗号分隔；子串要和真实指标名对得上")
    a = ap.parse_args()

    with open(a.csv, newline="", encoding="utf-8") as f:
        rows = list(csv.reader(f))
    rows = [r for r in rows if r and any(c.strip() for c in r)]
    if not rows:
```

```
        raise SystemExit("CSV 是空的")
    idx = sniff(rows[0])

    table = {}
    for r in rows[1:]:
        if len(r) <= max(idx.values()):
            continue
        k, m, v = r[idx["kernel"]], r[idx["metric"]], to_float(r[idx["value"]])
        if v is None:
            continue
        table.setdefault(k, {})[m] = v

    if not table:
        raise SystemExit("没解析出任何数值型指标，确认导出时带了 --page details")

    wants = [w.strip().lower() for w in a.show.split(",") if w.strip()]

    def pick(metrics, sub):
        for name, val in metrics.items():
            if sub in name.lower():
                return name, val
        return None, None

    def sort_key(item):
        _, val = pick(item[1], a.sort.lower())
        return -(val or 0.0)

    print(f"kernel 数 {len(table)}，按含 {a.sort!r} 的指标降序取前 {a.top}\n")
    for kname, metrics in sorted(table.items(), key=sort_key)[:a.top]:
        short = kname if len(kname) <= 66 else kname[:63] + "..."
        print(short)
        for w in wants:
            mn, mv = pick(metrics, w)
            if mn is None:
                print(f"    {w:<18} —（这份 CSV 里没有含该子串的指标）")
            else:
                print(f"    {w:<18} {mv:>12.2f}   [{mn}]")
        print()


if __name__ == "__main__":
    main()
```

脚本建好之后再执行透视：

```
python ncu_top.py --csv "$CH25_ROOT/profile/ncu_baseline.csv" --top 8
```

对本章实测的 `ncu_baseline.csv` 透视，输出如下（kernel 名超长时脚本自行截断）：

```
kernel 数 11，按含 'duration' 的指标降序取前 8

nvjet_tst_192x32_64x7_4x1_v_bz_TNT
    duration                      96.10   [Duration]
    compute (sm)                  11.67   [Compute (SM) Throughput]
    memory throughput             85.83   [Memory Throughput]
    dram throughput               85.83   [DRAM Throughput]

nvjet_tst_64x32_64x16_4x1_v_bz_splitK_TNT
    duration                      52.74   [Duration]
    compute (sm)                  10.47   [Compute (SM) Throughput]
    memory throughput             78.98   [Memory Throughput]
    dram throughput               78.98   [DRAM Throughput]

void cutlass::device_kernel<flash::enable_sm90_or_later<flash::FlashAttnFwdSm...
    duration                      38.02   [Duration]
    compute (sm)                  16.13   [Compute (SM) Throughput]
    memory throughput             55.68   [Memory Throughput]
    dram throughput               55.68   [DRAM Throughput]

nvjet_tst_64x32_64x16_4x1_v_bz_bias_TNT
    duration                      17.57   [Duration]
    compute (sm)                   7.60   [Compute (SM) Throughput]
    memory throughput             58.68   [Memory Throughput]
    dram throughput               58.68   [DRAM Throughput]

nvjet_tst_64x16_64x16_4x2_h_bz_TNT
    duration                      14.56   [Duration]
    compute (sm)                  12.78   [Compute (SM) Throughput]
    memory throughput             53.51   [Memory Throughput]
    dram throughput               53.51   [DRAM Throughput]

void vllm::act_and_mul_kernel<c10::BFloat16, __nv_bfloat162, &vllm::silu_kern...
    duration                       8.51   [Duration]
    compute (sm)                   6.72   [Compute (SM) Throughput]
    memory throughput              8.55   [Memory Throughput]
    dram throughput                8.55   [DRAM Throughput]

void vllm::rotary_embedding_kernel<c10::BFloat16, 1>(const long *, T1 *, T1 *...
    duration                       7.10   [Duration]
    compute (sm)                   3.08   [Compute (SM) Throughput]
    memory throughput              5.67   [Memory Throughput]
    dram throughput                1.14   [DRAM Throughput]

void vllm::reshape_and_cache_flash_kernel<__nv_bfloat16, __nv_bfloat16, 0>(co...
    duration                       6.91   [Duration]
    compute (sm)                   1.20   [Compute (SM) Throughput]
    memory throughput              5.03   [Memory Throughput]
    dram throughput                0.40   [DRAM Throughput]
```

（以上是本章实测 `ncu_baseline.csv` 的透视结果，与第九节步骤 3 的读数一致；方括号里是 CSV 中的原始指标名。）

六、定位：四象限判读

拿到 SM 吞吐、显存吞吐两个数之后，按这张表读。**高与低的界线直接用 ncu 自带规则的阈值**，这两句规则文本会随 `--page details` 一起导出到你的 CSV 里：

1. 任一项达到 **80%** 以上，判为受该资源限制。规则原文："This workload is utilizing greater than 80.0% of the available compute or memory performance of the device."
2. 两项都低于 **60%**，判为延迟受限。规则原文："Achieved compute throughput and/or memory bandwidth below 60.0% of peak typically indicate latency issues."
3. 落在 60%～80% 之间的，结合该 kernel 在 nsys 汇总里的耗时占比一起判，占比小的先不处理。

Occupancy 是第二步才看的量，见表后说明：

SM 吞吐	显存吞吐	性质	该做什么
高	低	算力受限	kernel 本身已接近上限，往上只能换更快的卡或降精度
低	高	访存受限	decode 的 attention 与 GEMM 是典型。方向是提高数据复用：增大 batch、启用前缀缓存、量化权重减少读取量
低	低	延迟受限	先看 Achieved Occupancy 与 Theoretical Occupancy 的差距：差距大是并行度不足（batch=1 的 GEMV 占不满 SM，方向是增大 batch 或合并小 kernel）；已贴近理论值则看 WarpStateStats 的停顿原因分布，多半是指令依赖或同步

Occupancy 不进判别列，原因有两个：

1. **它要和该 kernel 自己的 Theoretical Occupancy 比，不是和 100% 比。** 本章实测主 GEMM 的 Achieved Occupancy **14.2%**、Theoretical Occupancy **18.8%**（`ncu_baseline.csv` 的 Occupancy Section），比值 0.76。参考线取 0.7：achieved 除以 theoretical 超过 0.7 就算已经贴近该 kernel 自身的上限，这个上限由 tile 与寄存器配置决定，不是可调量。
2. **算力受限的 kernel 常常主动压低 Occupancy**，换取每线程更多寄存器，Occupancy 低不等于有问题。

读这两个指标之前，先分清 ncu 里两个容易混淆的名字。 `Memory Throughput` 是整个存储子系统（L1/TEX、L2、DRAM 等多条管线）里最吃紧那一条的峰值占比，`DRAM Throughput` 才专指 HBM。本章主 GEMM 上两者恰好都是 85.83%，所以判为受 HBM 限制没有问题；但不能把"Memory 高就是 HBM 受限"当成通则。反例就在你自己的 CSV 里：`rotary_embedding_kernel` 的 Memory 是 **5.71%**、DRAM 只有 **1.15%**；RMSNorm 那个 kernel（CSV 里是 `std::enable_if<...vllm::_typeConvert...>` 这样的模板展开名）是 Memory **7.81%** 对 DRAM **2.73%**，而它的 L1/TEX 有 **29.84%**。这两个 kernel 吃紧的是片上管线，不是显存。**判访存受限时把两个数一起看，只有 DRAM 也跟着高才是 HBM 的问题。**

另一个最容易犯的错，是看见 `nvidia-smi` 显示 100% 就判定 GPU 已被充分利用。 该指标只回答这段时间内是否有 kernel 在执行。一个只用了几个 SM 的小 kernel，只要一直占着卡，利用率也能显示 100%，它不区分有效计算与等待访存。要判断算力是否已达上限，只能看 ncu 的 SpeedOfLight：同一段时间里，Compute 和 Memory 各自摸到了峰值的百分之几。

实测结论明确。vLLM 和 SGLang 的主 GEMM 指标几乎一致：Memory 大约 **86%**，Compute 只有 **12%**。对照上表，Memory 过了 80% 这条线，两边的主 GEMM 都判为访存受限：瓶颈不在算力，而在等待 HBM 把权重和 KV 读入。

FlashAttention 那一行要按同一套规则判，不能跟着主 GEMM 说。 它的 Memory 55.7%、Compute 16.1%（SGLang 侧 56.2% / 33.0%），两项都不到 60%，按上表判的是**延迟受限**，不是访存受限。它在 nsys 汇总里只占 5.0%，不是本负载的主路径，所以本项目不对它下钻；真要优化它，方向是 WarpStateStats 的停顿原因，而不是提高数据复用。这一处是按规则判、不按看上去像判的例子：Memory 比 Compute 大不等于访存受限，要看它有没有过 80% 那条线。nsys 时间线里还有一件事要说清楚，**它也是这一步最容易过度解读的地方**：同一个窗口里既有启动次数极多的扁 tile，又有次数很少、两边都大的 tile。

这不足以证明 Continuous Batching 在混合调度。 本章的 nsys 是用 `--num-prompts 2` 采的，窗口里只有两条请求。把汇总表的 `Instances` 除以模型层数 28 就能看出这个窗口的结构：大 tile `nvjet_tst_192x192_*` 是 56 次，56 ÷ 28 = **2**，正好是那两条请求各自的 prefill；扁 tile `nvjet_tst_192x16_*` 是 3444 次，3444 ÷ 28 = **123**，是它们的 decode 步数。先做两次 prefill 再做 123 步 decode，同样会得到这个分布。

要证明引擎在同一个调度步里混合 prefill 与 decode（Continuous Batching：引擎每个迭代步都重新组批，已在解码的请求与新到的 prefill 请求会出现在同一步里），需要的是**逐 kernel 时间戳**：导 `cuda_gpu_trace` 看两类 tile 是否交错出现在同一个 step 区间内，或者直接看引擎的调度日志。汇总表只有次数和总时长，没有时间戳，回答不了这个问题。本章没有采时间戳，所以这里只能说窗口同时覆盖了 prefill 与 decode 两种工作。

顺带记住这个除法： `Instances ÷ 模型层数` 能直接读出采集窗口里跑了多少次前向，是核对采集口径最快的手段。

那下一步该调整哪个参数？conc=32 时，vLLM 默认 `max_num_seqs=1024`、SGLang 默认 `max_running_requests=4096`，都远大于并发，再翻倍请求数不会多纳入任何一条请求。真正形成约束的是单步 token 预算：两边 H100 默认都是 **8192**（vLLM 是 `max_num_batched_tokens`，SGLang 是 `chunked_prefill_size`；两者作用相近但语义不等价，vLLM 的预算含正在 decode 的 token，SGLang 的只约束 prefill 分块，conc=32 时 decode 只占 32 个 token，差异可忽略）。把它翻到 16384 之后：vLLM 吞吐 **2605→2586 tok/s（−0.7%，单轮，在噪声范围内）**，TTFT P99 +5.7%；SGLang 更明显，**2515→2380（−5.4%）**，TTFT P99 从 801 ms 升至 1250 ms。预先登记的判定标准是吞吐要涨、且首包劣化不超过 10%，**两边都不采纳**。主路径已经受限于访存带宽，再加大单步预算，只会让一步里同时做的 prefill 条数变多、单步变长，排在同一步里的 decode 和后面等待的请求都被拉长。关 CUDA Graph 省 launch 开销，是项目三的事，不要混入这一处对照。数字和 CSV 在 `reports/nsys_vllm.md`、`reports/nsys_sglang.md`。

七、改一处，然后复测

只改一个变量。根据第六步的结论选：

第六步的判定	改哪一个	本章是否命中
并发已经贴近 `max_num_seqs`，请求排不进批	调大 `--max-num-seqs`，把更多请求并入同一批，提高每次读 KV 的复用	**否**：默认 seqs=1024，实际并发只有 32
GPU 有空闲、队列里一直有等待请求	调大 `--max-num-batched-tokens`	**是**：单步 token 预算才是本负载的约束
GPU 利用率接近上限但 TTFT P99 很高	调**小** `--max-num-batched-tokens`，让 prefill 分块更细，给 decode 让路	否

先对着这张表判一次，再动手。 本章命中的是第二行，所以下面改的是 `--max-num-batched-tokens`，不是 `--max-num-seqs`：第六步已经算过，conc=32 时 seqs 默认 1024 远大于并发，调它不会改变一步里入批的条数。`--max-num-seqs` 和

`--max-num-batched-tokens` 在 EngineArgs 层都是 `None`，真实取值由引擎按场景推导：**不要背数字，去启动日志里看引擎实际用了多少**，然后在此基础上改：

```
vllm serve "$CH25_ROOT/models/Qwen2.5-7B-Instruct" \
    --served-model-name Qwen2.5-7B-Instruct \
    --host 0.0.0.0 --port 8000 --max-model-len 8192 \
    --gpu-memory-utilization 0.88 \
    --max-num-batched-tokens 16384
```

16384 是基线值 8192 的两倍。基线值不要背，从启动日志里读：

```
grep -E "max_num_batched_tokens|max_num_seqs" "$CH25_ROOT"/bench/serve_baseline.log
```

复测：命令与第三步**逐字相同**，只换输出文件名：

```
vllm bench serve --model Qwen2.5-7B-Instruct --base-url http://127.0.0.1:8000 \
    --dataset-name random --random-input-len 1024 --random-output-len 128 \
    --num-prompts 256 --max-concurrency 32 --request-rate inf \
    --seed 0 --ignore-eos --percentile-metrics ttft,tpot,itl,e2el \
    --metric-percentiles 50,95,99 --save-result --result-filename tuned.json
```

然后**重新采一次 ncu**，确认那个 kernel 的 Occupancy 或显存吞吐真的动了。**只看端到端吞吐涨了就宣布优化成功，是这一行最常见的误判**：吞吐可能因为完全无关的原因波动。判定标准预先登记：目标 kernel 的目标指标朝预期方向变化，且端到端吞吐提升、TTFT P99 不劣化超过 10%，两条同时成立才算数。

复测结果：conc=32 时默认 seqs / max_running_requests 都不是约束，对着预算把单步 token 预算从 8192 改到 16384。vLLM btok：吞吐 **2605→2586 tok/s（−0.7%）**，TTFT P99 **839→886 ms（+5.7%）**，两项都是单轮、都在噪声范围内，按判据不能认定有收益。SGLang chunked_prefill：吞吐 **2515→2380 tok/s（−5.4%）**，TTFT P99 **801→1250 ms（+56%）**。ncu 主 GEMM 两边都是 Mem ≫ Comp。按判据 **都不采纳**。完整证据链见第九节。

八、必做核心与自选扩展

必做核心是四步：压出一个可复现的基线、用 `--capture-range=cudaProfilerApi` 给常驻服务设定采集窗口采一次 nsys 并出 kernel 汇总、用 ncu 只对前几名 kernel 采四个 Section、按四象限判读后改一处并同时复测端到端指标和目标 kernel 的指标。只做 nsys 不做 ncu，就只能说出哪个 kernel 耗时高，说不出它为什么慢。

自选扩展往两个方向做，前者面向选型决策，后者面向 kernel 调优：

1. **跨引擎归因**：用同一套采集参数对 SGLang 采一次，配合 `--enable-layerwise-nvtx-marker` 和 `profile_by_stage`，比较两个引擎在同一模型上的 attention kernel 选型差异，产出一份说明性能差异来源的对比报告。
2. **层级下钻**：用 NVTX 区间（`--nvtx --nvtx-include`）把采集限制到单独一层，在层内对比不同 attention backend 的 SpeedOfLight，产出一份 backend 选型依据。

九、逐步证据链（面试可复述）

这段要能在面试里连贯讲完：常驻服务必须设定采集窗口再采，Nsight Systems（nsys，时间线工具）只回答哪个 kernel 累计耗时最高，Nsight Compute（ncu，单 kernel 计数器）才回答它为什么慢；调整调度参数之前先确认哪一个真正形成约束，最后按预先登记的判定标准决定采不采纳。数字来自 `projects/pr2/reports/nsys_vllm.md` 与 `projects/pr2/reports/nsys_sglang.md`。关 CUDA Graph（把一串 kernel 提交录成一张图、之后一次回放）的 A/B 在项目三，不并入本项目的改一处。

本节术语

术语	全称	含义
nsys	Nsight Systems	画一张各 kernel 起止时刻与持续时间的时间线，用来定位开销最高的 kernel
ncu	Nsight Compute	针对单个 kernel，读取它的硬件计数器，用来回答它为什么慢
CUDA Graph	CUDA Graph	把一串 kernel 提交录成一张图，之后一次 `cudaGraphLaunch` 整张回放
launch	kernel launch	CPU 把一个 kernel 交给 GPU 的那一次提交，单次只要几微秒，但 decode 每步要发几百次
ITL	Inter-Token Latency	相邻两个输出 token 之间隔了多久，流式对话最敏感的那个数
TPOT	Time Per Output Token	平均每个输出 token 花多久，稳态 decode 快慢的尺子
eager	eager mode	不开 CUDA Graph，每个 kernel 单独 launch；本项目的 `eager.json` 是旧口径，不当锚

步骤 1｜先确定对比口径

这一步在干什么：vLLM V1 是常驻不退出的服务，如果你把 nsys 从启动一直开到结束，启动期编译、捕获图、预热和稳态 decode 会混进同一份报告，后面读 Time(%) 就会把耗时排行读错。所以必须先约定：只采稳态窗口、端到端看哪几个数、怎样才算改动成功。

动作：服务端加 `--profiler-config '{"profiler":"cuda"}'`，让应用在窗口里自己调 `cudaProfilerStart/Stop`；客户端压测加 `--profile` 当触发器。端到端同时记吞吐、TTFT P99、TPOT（Time Per Output Token，平均每输出 token 耗时）；ncu 只对排行榜前几名下钻四个 Section。判据预先登记：目标 kernel 的目标指标朝预期方向走，**且**端到端吞吐提升、TTFT P99 劣化不超过 10%，两条同时成立才算数。

本步产生的文件

路径	谁写的	看哪列	得到哪个数
`projects/pr2/reports/nsys_vllm.md`	人写的读数笔记	判据（预先登记）那一段	吞吐必须升，且 TTFT P99 劣化 ≤10%
`projects/pr2/bench/closed_loop_summary.json`	闭环脚本汇总	`load` / `change` / `adopt`	负载 in1024/out128、prompts=256、conc=32；改的是 btok 8192→16384；`adopt=false`
`projects/pr2/bench/eager.json`	vLLM bench（早先一次缩水负载）	`num_prompts` / `max_concurrency` / `output_throughput`	prompts=32、conc=8，缩水负载，**不当本项目锚**

读数：统一口径为 in1024 / out128、prompts=256、conc=32、rate=inf；改一处只动 `--max-num-batched-tokens`。`eager.json` 是早先一次用 `--enforce-eager` 关 CUDA Graph 的产物，负载更小，CUDA Graph 的对比留给项目三。

为什么能推下一步：口径和判据预先登记之后，才能采一条保持默认 CUDA Graph、尚未调整调度参数的基线；否则后面 2604 和 2586 对不上号。

步骤 2｜基线：conc=32 下两个引擎都已受限于显存带宽

这一步在干什么：先测出当前吞吐，不先改任何调度参数。如果基线本身还没有到达任何一种资源上限，后面将 token 预算翻倍后变快，就无法区分是基线本身有余量，还是调整确实带来了收益。

动作：用默认路径（CUDA Graph 开着、btok=8192）跑同口径压测，把 JSON 落盘。

本步产生的文件

路径	谁写的	看哪列	得到哪个数
`projects/pr2/bench/baseline.json`	vLLM bench	`output_throughput` / `p99_ttft_ms` / `mean_tpot_ms` / `p99_itl_ms`	**2604.7 tok/s**、TTFT P99 **838.9 ms**、TPOT mean **8.77 ms**、ITL P99 **114.0 ms**
`projects/pr2/bench/sgl_baseline.json`	SGLang 同口径 bench	同上四个字段	**2515.2 tok/s**、TTFT P99 **801.2 ms**、TPOT mean **9.28 ms**、ITL P99 **21.4 ms**
`projects/pr2/bench/closed_loop_summary.json`	闭环脚本汇总	`baseline.*`	把上面 vLLM 四个数收成一张表，方便和复测相减
`projects/pr2/reports/nsys_vllm.md`	人写的读数笔记	端到端对照的基线列	与 JSON 对齐的四舍五入数

读数：vLLM 默认路径 2604.7 tok/s，SGLang 同口径 2515.2 tok/s，两边差约 3.4%，只说明两者在该负载下受同一个瓶颈约束（步骤 3 的 ncu 证明是显存带宽，主 GEMM 的 Compute 只有 11.7%），不能据此宣称谁更快。早先一次缩水负载（prompts=32 / conc=8 / in512）的数字不作基线，两套数不能混比。

为什么能推下一步：基线已经受限于带宽，下一步不应假设增大 token 预算就能线性提升吞吐，而要问 GPU 时间到底花在谁身上、那个 kernel 受限于算力还是受限于带宽。

步骤 3｜剖析：瓶颈是访存带宽，不是单步 token 预算不足

这一步在干什么：nsys 给出哪个 kernel 累计耗时最高，ncu 给出它为什么慢。只有两边对上，你才知道该不该去换 attention、该不该禁用 CUDA Graph、还是该调整调度参数。

动作：对常驻服务设定采集窗口采 nsys（`--cuda-graph-trace=node`，否则整张 CUDA Graph 在时间线上只是一条），导出 `cuda_gpu_kern_sum`；再对排行榜前几名用 ncu 采 SpeedOfLight 等四个 Section。SGLang 半边同样做一遍。

本步产生的文件

路径	谁写的	看哪列	得到哪个数
采集 `projects/pr2/profile/vllm_baseline.nsys-rep` → 导出 `projects/pr2/profile/kern_baseline_cuda_gpu_kern_sum.csv`	nsys `stats --report cuda_gpu_kern_sum`（`nsys_stats.log` 记载采集名；目录里读导出 csv）	`Time (%)` + `Name` + `Instances`	第 1 行 **21.2%** `nvjet_tst_192x16_*`（Inst 3444，decode 扁 GEMM）；第 3 行 **11.1%** `nvjet_tst_192x192_*`（Inst 仅 56，大 tile / prefill-like）；第 6 行 **5.0%** `FlashAttnFwdSm90`（Inst 3976）
采集 `projects/pr2/profile/sglang_baseline.nsys-rep` → 导出 `projects/pr2/profile/kern_sglang_cuda_gpu_kern_sum.csv`	同上	`Time (%)` + `Name` + `Instances`	第 1 行 **8.1%** `nvjet_192x8`（Inst 532）；第 2 行 **7.6%** `FlashAttnFwdSm90`（Inst 2268）；第 3 行 **4.9%** `nvjet_192x128`（Inst 280，大 tile）
采集 `ncu_baseline.ncu-rep` → 导出 `projects/pr2/profile/ncu_baseline.csv`；人读摘要 `ncu_top.txt`	ncu（`--launch-skip 2500 --launch-count 24`）	`Kernel Name` + `Metric Name` = `Memory Throughput` / `Compute (SM) Throughput` + `Metric Value`	主 GEMM `nvjet_192x32`：**Mem 85.8% / Comp 11.7%**；FA3：**Mem 55.7% / Comp 16.1%**
采集 `ncu_sglang.ncu-rep` → 导出 `projects/pr2/profile/ncu_sglang.csv`；人读摘要 `ncu_sglang_top.txt`	ncu 同口径	同上	主 GEMM **Mem 86.2% / Comp 11.8%**；FA3 **Mem 56.2% / Comp 33.0%**（仍是 Mem 压过 Comp）
`projects/pr2/reports/nsys_vllm.md` / `nsys_sglang.md`	人写的读数笔记	nsys kernel 汇总与 ncu 四象限两节	把 csv 前几行转成短表，避免在超长 Name 中难以定位

先学会读 kernel 名，否则这一列只是一串无法辨识的字符。 以第一名 `nvjet_tst_192x16_64x8_4x1_v_bz_TNT` 为例：`nvjet` 是 cuBLASLt 在 Hopper 上的 GEMM 实现族，紧跟其后的 `192x16` 是这个 kernel 一次处理的输出块形状（tile）。nvjet 的命名规则没有公开文档，后面几段数字无需逐段解读；**只看第一对数字就够**：它是这个 kernel 一次处理的输出块形状，两个数里小的那一个对应 M 方向，也就是 token 行数那一维。decode 每步每条请求只有 1 行，即使合批也只有几行到几

十行，所以落在 `192x16` 这类一边大一边小的扁 tile 上；prefill 一次进来上千行，所以落在 `192x192`、`256x160` 这类两边都大的方 tile 上。本章数据与此一致：扁 tile 的 `Instances` 是 3444，方 tile 只有 56。

但不要把 tile 的第二个数当成实际 batch 读。 M 方向的 tile 尺寸是有档位的，batch 1、2、8 很可能都落到同一档，`192x16` 里的 16 是这一档的块高，不是"这一步合批了 16 条"。本章的 nsys 窗口实际只有 2 条请求，照样出现 `192x16`。tile 形状只能用来区分扁与方、也就是区分 decode 与 prefill，读不出具体批量。把 tile 形状和 `Instances`（启动次数）合起来看：扁 tile 且次数极多是 decode，方 tile 且次数很少是 prefill。

读数：同一窗口里既有 Inst 极多的扁 tile，也有 Inst 很少、但单次很长的大 tile，说明这个窗口同时覆盖了 prefill 与 decode 两种工作（`Instances ÷ 28 层` 可读出是 2 次 prefill 加 123 步 decode）。**注意这一步的采集窗口只有 2 条请求，它的 Time% 排行不能代表并发 32 的压测负载，也不足以证明两类请求被排进了同一个调度步。**ncu 进一步确定瓶颈类型：主路径 Memory Throughput 八十多、Compute 十个点上下，四象限落在访存受限。SGLang 那边时间占比被大量不同 tile 拆碎，没有单一 kernel 占到 20%+，优化单个 kernel 更不可能拉动端到端。

为什么能推下一步：主路径已经受限于访存带宽，下一步不该去换一个更快的 attention kernel，也不该把 `--enforce-eager` 拿来当本项目的对照（那是项目三要量化的 launch 开销）。该问的是：当前并发下，两个调度上限参数究竟哪一个在形成约束。

步骤 4｜只改一处：对着真正的约束将 btok 翻倍

这一步在干什么：H100 serve 默认有两个调度上限参数：`max_num_seqs`（一步最多几条请求）和 `max_num_batched_tokens`（一步最多几个 token，简称 btok）。conc=32 时 seqs 默认 1024，远大于并发，再翻倍 seqs 不会改变一步里实际入批的条数。一步预算 8192 token，而单条 prompt 是 1024 token。**这里要分清 btok 限的是什么**：它限的是一个调度步里所有请求加起来最多多少 token，不是单条请求的分块大小。1024 小于 8192，所以单条 prompt **一步就能做完、不会被切块**；8192 这个预算的实际含义是"一步最多拼 8 条 prefill"。32 条请求要排 4 个步才能全部 prefill 完，对着这个约束改的只能是 btok。

翻到 16384 之后变化的是"一步能拼 16 条"，不是"每块变大"。这个区别直接决定了下面复测结果该怎么解释。

动作：只把 vLLM 的 `--max-num-batched-tokens` 从 8192 改到 16384，压测命令逐字相同。SGLang 半边只把 `--chunked-prefill-size` 从 8192 改到 16384（它与 vLLM 的 btok 作用相近但语义不等价：vLLM 的预算含正在 decode 的 token，SGLang 的只约束 prefill 分块；conc=32 时差异可忽略）。要验证的命题很具体：放大单步 token 预算，能不能让已经受限于访存带宽的主路径再多承载一些。

本步产生的文件

路径	谁写的	看哪列	得到哪个数
`projects/pr2/bench/engine_cfg_serve_tuned.txt` / `sgl_serve_tuned.log`	服务启动日志	搜 `max_num_batched_tokens` / `chunked_prefill_size`	翻完后确认为 16384，证明改动真的生效了
`projects/pr2/reports/nsys_vllm.md`	人写的读数笔记	为什么改 btok 而不是 seqs 那一段	默认 seqs=1024、btok=8192；conc=32 时 seqs 不是约束
—	—	—	**不采集 nsys**：这一步只改参数、不重采时间线，证据在启动日志和下一步的 bench JSON

读数：启动日志原文是 `Chunked prefill is enabled with max_num_batched_tokens=8192`，翻完变成 16384。SGLang 基线日志是 `chunked_prefill_size=8192, max_running_requests=4096`，conc=32 时 running-requests 也远不是约束。

为什么能推下一步：改动已经落到服务配置上，而且只动了一处。下一步用同一条压测命令复测，数字一变就能归因到单步 token 预算这一处改动。

步骤 5｜复测：两边改动都不成立

这一步在干什么：把该参数翻倍后再跑一遍完全相同的压测，拿预先登记的判定标准做决定。ITL（Inter-Token Latency，相邻两个输出 token 的间隔）P99 从 114 降到 62，方向与 TTFT 相反；SGLang 侧则是 21→28，与 vLLM 又相反。**两边方向都对不上同一个机制，说明单轮数据不足以支撑任何机制叙述。** 本项目按预先登记的判据否决这次改动，但不声称知道 ITL 为什么这样动；要解释它，至少需要重复 3 到 5 轮看这个方向稳不稳定，以及看 ITL 的中位数与 P99.9 而不只是 P99。

动作：vLLM / SGLang 各跑一轮 tuned，和基线逐字段相减。

本步产生的文件

路径	谁写的	看哪列	得到哪个数
`projects/pr2/bench/tuned.json`	vLLM bench	`output_throughput` / `p99_ttft_ms`	**2585.8 tok/s**、TTFT P99 **886.3 ms**
`projects/pr2/bench/sgl_tuned.json`	SGLang bench	同上	**2380.2 tok/s**、TTFT P99 **1249.8 ms**
`projects/pr2/bench/closed_loop_summary.json`	闭环脚本汇总	`tuned_btok_16384.*` / `delta_pct` / `adopt`	吞吐 **−0.7%**、TTFT P99 **+5.7%**、`adopt=false`
`projects/pr2/reports/nsys_vllm.md` / `nsys_sglang.md`	人写的读数笔记	端到端对照的复测列	vLLM −0.7% / +5.7%；SGLang **−5.4%** / **+56%**
—	—	—	**不重采 nsys/ncu**：复测看的是端到端 JSON；微观侧沿用步骤 3 的 Mem ≫ Comp，两边同向

读数：vLLM 2605→2586 tok/s（−0.7%），TTFT P99 839→886 ms（+5.7%），单轮、在噪声范围内，判据要求吞吐提升，未满足。SGLang 2515→2380（−5.4%），TTFT P99 801→1250（+56%），变化明确。按事先判据 **两边都不采纳**。SGLang 否决更干净：一步里拼进更多 EXTEND（SGLang 对 prefill / chunked-prefill 的阶段名）直接使首包延迟大幅劣化。ncu 主 GEMM 两边都是 Mem ≫ Comp，和端到端同向：已经受限于显存带宽，再加大预算只会让一步里拼进更多 prefill、单步变长，decode 的间隔被拉开。

为什么能推到结论：吞吐没升、首包更差，判据两条都没满足；项目一里靠 `/start_profile` + `profile_by_stage` 落到 DECODE / EXTEND 两份 trace 的那一步，本项目可以省掉。禁用 CUDA Graph 的 launch 开销对比留给项目三。

一句话：常驻服务设定采集窗口采 nsys，ncu 确认两边主 GEMM 均为访存受限（Mem ~86% / Comp ~12%）；conc=32 时真正的约束是单步 token 预算（vLLM btok / SGLang chunked_prefill），翻到 16384 后吞吐未提升（vLLM 侧在噪声内，SGLang 侧下降 5.4%）、首包更差，按判据否决这次改动。CUDA Graph 的对比留给项目三。

十、排错表

现象	原因	处理
客户端 `--profile` 报 404	服务端没带 `--profiler-config`，两个端点没挂载	启动时加上；帮助文本里已写明这是前提
服务端配了 `profiler: "cuda"` 还传了 `torch_profiler_dir`	校验函数只允许 `profiler == "torch"` 时给目录	二选一
`torch_profiler_dir` 传相对路径，trace 找不到	不会报错，会按启动进程的工作目录 `abspath`	显式传 `"$CH25_ROOT/profile"`
设了 `VLLM_TORCH_PROFILER_DIR` 完全没反应	v0.26.0 的 `envs.py` 里已无此变量	改用 `--profiler-config`
nsys 报告里只有一条 CUDA Graph 时间线	用了默认的 `--cuda-graph-trace=graph`	换成 `node`
nsys 时间线上没有 SM active / DRAM 行	`--gpu-metrics-devices` 默认是 `none`	加 `cuda-visible` 或 `all`
nsys 抓不到线程调度数据	`perf_event_paranoid` 未放开	见 0.5 节。注意这不会报错，只是数据缺失
`ncu` 报 `ERR_NVGPUCTRPERM`	计数器权限未放开	见 0.5 节。新驱动（R610+）改用 capabilities 授权
ncu 一个 kernel 都没采到	默认只跟根进程	加 `--target-processes all`
ncu 启动后长时间无输出	没加 kernel 过滤，正在采集服务启动阶段的每一个 kernel；或正则匹配不到任何 kernel，一直在等	加 `--kernel-name regex:` + `--launch-skip` + `--launch-count`；正则从 nsys 汇总第一列抄
ncu 采到 0 个 kernel	`--kernel-name` 的正则与实际 kernel 名不匹配	回 `cuda_gpu_kern_sum.csv` 第一列抄真实名字，本章实测栈上是 `nvjet` 与 `FlashAttnFwd`
ncu 报告里 CUDA Graph 内的 kernel 全并成一条	默认按整图当一个 workload	加 `--graph-profiling node`
`--show` 里的指标一行都没匹配上	`--page details` 用显示名称，`--page raw` 与 `--query-metrics` 用内部指标名，两套不通用	先看 CSV 的 `Metric Name` 列，按显示名称写子串
SGLang 的 `/metrics` 返回 404	`--enable-metrics` 默认 False	加上该 flag 重启

十一、面试深挖点

下面的参考答法里带着本章的实测数。 这些数是在 H100 上跑出来的，换卡、换负载都会变；照搬去讲，对方一追问负载口径就答不上来。哪些能讲、哪些要自己跑出来，见全章末节“把这三个项目写进简历之前”。

Q1｜nsys 和 ncu 有什么区别？什么时候用哪个？

考察点：分不分得清时间线工具和计数器工具。这是这一行的入门筛子。

参考答法：nsys 是时间线，采 kernel 什么时候在哪个流上跑了多久，加上 CPU 侧的 API 调用和线程状态，用于回答三类问题：CPU 与 GPU 的等待关系、哪个 kernel 累计耗时最高、GPU 是否存在空闲。ncu 是单 kernel 的硬件计数器采集，回答“这一个 kernel 为什么慢”：SM 吞吐、显存吞吐、Occupancy、warp 停在什么原因上。**顺序必须是先 nsys 后 ncu**：ncu 要把 kernel 重放多次以采全计数器，不可能对整个服务采一遍，必须先用 nsys 圈出目标，再用 `--kernel-name` 加 `--launch-skip` / `--launch-count` 精确采那几个。

Q2｜nsys 怎么抓一个常驻不退出的推理服务？

考察点：有没有真的抓过线上服务。只抓过 demo 脚本的人答不出采集窗口的问题。

参考答法：不能全程采集，报告会大到不可用，node 级的 CUDA Graph 追踪开销也很大，必须设定采集窗口。官方路径有两条：

1. `--capture-range=cudaProfilerApi` 配合应用内的 `cudaProfilerStart/Stop`。vLLM 侧是服务端配 `--profiler-config '{"profiler":"cuda"}'`、客户端压测时加 `--profile` 当触发器；SGLang 侧是 POST `/start_profile`。
2. `nsys launch` 起进程后用 `nsys start/stop --session` 控制，或者用 `--delay` / `--duration` 按时间划。

没有 `nsys attach` 这个子命令，不能对一个任意的已运行 PID 事后附加。另外抓 decode 必须加 `--cuda-graph-trace=node`，否则整张 CUDA Graph 在时间线上只是一条。

Q3｜Occupancy 低就一定是问题吗？

考察点：会不会把 Occupancy 当成越高越好的分数。这是 GPU 性能分析里最常见的误解。

参考答法：不一定。Occupancy 衡量的是每个 SM 上活跃 warp 数与硬件上限之比，它的作用是**掩盖延迟**：warp 数量足够时，一个 warp 等访存时可以切到别的 warp。所以只有在延迟受限的场景下提高 Occupancy 才有收益。如果一个 kernel 已经达到显存带宽峰值的 80%，那它是访存受限，再多 warp 也只是排更长的队；如果它 SM 吞吐已经很高，那它是算力受限，Occupancy 低反而可能是刻意的：每个线程多用寄存器换更少的访存，是常见的优化手法。判断顺序应该是：先看 SpeedOfLight 里 SM 和显存两个吞吐谁高，确定受限类型，再决定 Occupancy 这个数有没有意义。**孤立地看 Occupancy 得不出任何结论。**

Q4｜你怎么保证改了一处之后变快了这个结论是真的？

考察点：实验设计里的变量控制。

参考答法：三层。

1. 一次只改一个变量，复测命令与基线逐字相同，包括 seed、并发、输入输出长度、`--ignore-eos`。
2. **端到端指标和微观指标必须同时验证**：只看吞吐涨了不算，还要回去用 ncu 确认目标 kernel 的目标指标真的朝预期方向动了；两者对不上说明变化另有原因。
3. 判定门槛预先登记并留余量：同配置重复三到五轮，报区间不报单点，门槛要显著大于轮间方差。

第三条本章自己没做到：项目二的 btok 对照只跑了单轮，vLLM 侧 −0.7% 落在噪声里，所以那半边我只能说“没有观测到收益”，不能说“变差了”。面试里主动说清这个边界，比把单轮结果讲成结论安全得多。还要主动说清 profiling 本身的扰动：开着 node 级图追踪时的绝对性能不能和不开 profiler 的压测数直接比，两组各自内部比较才有意义。

Q5｜（陷阱题）我用 ncu 把一个 kernel 优化快了 30%，端到端却几乎没变，是不是测错了？

考察点：会不会先怀疑测量，而不是先算一遍这个 kernel 在整体里占多少。题面预设了“kernel 快 30% 就该端到端快”，这个前提本身不成立。

参考答法：大概率没测错，是这个 kernel 的占比太小。先回到 nsys 的 kernel 汇总做一次估算：如果它只占 GPU 繁忙时间的 10%，快 30% 最多带来 3% 的 GPU 时间下降；而如果 GPU 占空比本身只有 50%，这 3% 还会被空闲时间抵消大半，端到端看不出变化完全正常。**这就是为什么必须先 nsys 定位再 ncu 下钻**：优化对象要按累计占比选，不是按主观判断选。接下来该做三件事：

1. 确认这个 kernel 的累计占比到底多少；
2. 看 GPU 占空比，占空比低就说明瓶颈根本不在任何单个 kernel 上，而在调度或 launch 开销上；
3. 检查这个 kernel 是否在关键路径上，被其它流上的操作掩盖掉的 kernel，再快也不影响端到端。

项目三　复现 CUDA Graph 捕获机制并量化 Decode 加速收益

这个项目要回答的问题：CUDA Graph 该不该开？它省的到底是哪一层开销，代价有多大？

本项目的结论：它省的是 CPU 侧提交 kernel 的开销，GPU 侧的计算一点没变。在本项目的负载上（输入 64、输出 256、并发 16，decode 占主导），默认路径相对 `--enforce-eager` 的吞吐是 **1.91×**、TPOT 是 **0.52×**；**这是 torch.compile 与 CUDA Graph 两项合计**，因为 `--enforce-eager` 会把两者一起关掉，本章没有跑只关图的第三组。代价是启动时 **3 秒**捕获、**0.01 GiB** 显存，都远小于常见的估计。收益随负载形态变化：换成项目二那种 1024 输入的负载，prefill 走分段图且本身耗时长，收益比例会明显小于这个数。

手写的最小 Runtime 在四档 batch 上量到 **1.16×～1.18×**，省下的绝对时间随 batch 从 950.9 us 升到 1289.1 us，但占 eager 的比例始终停在 **13%～16%**，没有出现常被引用的“比例随 batch 迅速下降”。原因是这个微基准的单个 kernel 本来就足够长，launch 开销一开始就不是主导项。**一个与预期不符的实测结果，要先解释它为什么不符，而不是改掉它。**

读完能讲清：

1. CUDA Graph 省的是哪一层开销，为什么只有 decode 走整图捕获；
2. 捕获有哪五条硬约束，违反后是什么报错；
3. 捕获尺寸阶梯怎么算，图的显存为什么不等于张数乘以单张激活；
4. 一次开关图的 A/B 里除了图本身还同时变了什么，以及怎么用量级核对排除它。

一、目标与交付物

先读懂两个引擎里 CUDA Graph 的捕获与回放是怎么实现的，再在一个简化 Runtime 里把这套机制复现出来，量化它相对 eager 模式在 decode 上的 kernel launch 开销节省与加速比，同时把代价（显存、启动时间、padding 浪费）算清楚。

交付物三样：

1. `capture_sizes.py`：按引擎源码里的阶梯公式复算捕获尺寸表，给出图数量、padding 浪费与显存成本；
2. `mini_graph_runtime.py`：最小可运行的捕获/回放 Runtime，含一个主动触发捕获失败的开关；

3. 引擎侧 `--enforce-eager` A/B 的端到端复测记录。

做到什么算完成（四条缺任何一条，这个项目就没走完闭环）：

1. 能在本版本源码里指出捕获模式、默认值与参数名；
2. 捕获尺寸表与图数量是算出来的，能与启动日志对上；
3. 手写 Runtime 能跑通捕获与回放，并能主动触发一次捕获失败；
4. 引擎侧 A/B 有同口径的两组端到端数，且收益与代价两侧都有实测背书。

二、CUDA Graph 到底在省什么

三句话讲清前置概念。

它省的是 CPU 侧的提交开销，不是 GPU 侧的计算。 正常执行时，每发一个 kernel，CPU 都要走一遍完整的提交路径：Python 侧的循环与函数调用、PyTorch 的算子分发、显存分配器、参数打包、驱动调用、提交到流。CUDA Graph 把一整串操作记录成一张图，之后一次 `cudaGraphLaunch` 就把整张图交给 GPU，省掉的是这中间几百次往返。**注意省下的不只是驱动层**：驱动侧单次 launch 通常在亚微秒到 1.5 us，而实测每步省下近 1 ms，主体来自它上面的 Python 与分发开销。

代价是图里的一切都必须固定。 张量的地址、形状、控制流路径在回放时和捕获时必须完全一致：因为图记录的是具体地址上的操作序列，不是按变量名记的操作语义。

decode 恰好满足这个条件，prefill 不满足。 decode 每步的形状只随 batch 大小变（序列长度的变化被 KV Cache 的分页机制吸收在 attention kernel 内部），可以按 batch 尺寸预先捕获若干张图；prefill 的输入长度千变万化，捕获成本不划算。所以**引擎的整图（FULL）捕获只针对 decode；prefill 与混合批在 vLLM 里走分段（PIECEWISE）捕获，只把形状固定的算子段录进图**。项目一第六步里“平均每次 launch 只换来个位数微秒 GPU 时间”那个观测，就是这一步值得做的直接证据。

三、读现役实现：vLLM v0.26.0

三个文件构成完整链路。

模式定义在 `vllm/config/compilation.py` 的 `CUDAGraphMode` 枚举，五个取值及其文档串原意：

模式	含义
`NONE`	不捕获任何图
`PIECEWISE`	只建分段图，把和图不兼容的算子（主要是 attention）留在图外 eager 执行
`FULL`	对所有 batch 捕获整图。文档串明写 “not supported by many backends”
`FULL_DECODE_ONLY`	只对 uniform decode batch 捕整图，混合 prefill-decode 批不走 CUDA Graph。适合 P/D 分离里的 decode 实例，省显存
`FULL_AND_PIECEWISE`	decode 走整图、prefill 与混合批走分段图。文档串原文 “This is the most performant mode for most models and is the default”

默认就是开的，这一点决定了整个实验的方向：**不存在打开 CUDA Graph 这个动作，A/B 只能反着做**：基线是开着的，实验组用 `--enforce-eager` 或 `--compilation-config '{"cudagraph_mode": "NONE"}'` 关掉。`--enforce-eager` 的官方帮

助文本是 "Whether to always use eager-mode PyTorch. If True, we will disable CUDA graph and always execute the model in eager mode."，默认 `False`。

捕获尺寸同样在这个文件里，`max_cudagraph_capture_size` 的文档串给了完整公式：

```
[1, 2, 4] + list(range(8, 256, 8)) + list(range(256, max_cudagraph_capture_size + 1, 16))
```

并写明"If not specified, max_cudagraph_capture_size is set to min(max_num_seqs*2, 512) by default"，理由是避免在 `max_num_seqs` 很小时 OOM，以及避免捕获大量 512 以上的大图使启动时间延长。相关字段还有 `cudagraph_capture_sizes`（默认 `None`，显式指定时以指定值为准）、`cudagraph_num_of_warmups`（源码默认 `0`，但未强制 eager 时引擎会在配置阶段把它改成 `1`，所以启动日志里看到的是 1）、`cudagraph_copy_inputs`（默认 `False`，且文档串注明“only effective when cudagraph_mode is PIECEWISE”）。

捕获执行在 `vllm/v1/worker/gpu_model_runner.py` 的 `capture_model()`：它在 `self._freeze_gc()` 和 `graph_capture(device=...)` 两个上下文里遍历所有待捕获的 batch 描述符逐个捕获，结束时记下前后可用显存之差，打这条日志：

```
Graph capturing finished in %.0f secs, took %.2f GiB
```

这条日志就是本项目要的两个成本数字：捕获耗时进了服务冷启动时间，捕获显存从可用显存里扣掉。同一个文件里还有 `profile_cudagraph_memory()`，配合默认为 `True` 的环境变量 `VLLM_MEMORY_PROFILER_ESTIMATE_CUDAGRAPHS`（`vllm/envs.py`，注释 "enable CUDA graph memory estimation during memory profiling. This profiles CUDA graph memory usage to provide more accurate KV cache memory allocation"）。也就是说，图占的显存在算 KV Cache 池大小之前就被预留了，不是事后挤占。

包装层在 `vllm/compilation/cuda_graph.py`：`CUDAGraphWrapper` 包住一个 runnable，第一次遇到某个 batch 描述符就捕获成一个 `CUDAGraphEntry`（里面存着 `torch.cuda.CUDAGraph` 对象），之后同样的描述符直接回放。类文档串里有一句关键说明：**wrapper 本身不管持久 buffer，也不把运行时输入拷进去，这件事必须由外部保证**。这正是下一步简化 Runtime 里静态 buffer 那部分要做的事。

四、读现役实现：SGLang v0.5.16 的参数已经全换了

这是本章最容易出错的一处：网上几乎所有 SGLang 教程写的 `--disable-cuda-graph` 和 `--cuda-graph-max-bs`，在 0.5.16 上已经是 Deprecated 参数。它们仍能启动（`server_args.py` 里分别注册为 `DeprecatedStoreTrueAction` 与 `DeprecatedAliasStoreAction`），但会打警告并自动映射到下表的新参数；新写的命令不要再用旧名，面试里也不要说它们已被删除。

依据是 `python/sglang/srt/server_args.py`。0.5.16 把 CUDA Graph 配置改成了**按阶段（prefill / decode）分开**的一组字段：

现役参数	说明（取自源码里的 help 串）
`--cuda-graph-config`	收 JSON，形如 `'{"decode":{"backend":"full","max_bs":256},"prefill":{"backend":"tc_piecewise"}}'` 。help 里明写 "JSON wins over the per-phase --cuda-graph-* convenience flags and over legacy flags"
`--cuda-graph-backend-decode` / `--cuda-graph-backend-prefill`	取值 `full` / `breakable` / `tc_piecewise` / `disabled` ，其中 `full` 只对 decode 有效
`--cuda-graph-max-bs-decode` / `--cuda-graph-max-bs-prefill`	“Maximum batch size captured for the ... cuda graph.” 默认 `None`
`--cuda-graph-bs-decode` / `--cuda-graph-bs-prefill`	“Explicit list of batch sizes to capture”
`--disable-decode-cuda-graph` / `--disable-prefill-cuda-graph`	help 原文 “Convenience for --cuda-graph-backend-decode=disabled”
`--enable-profile-cuda-graph`	“Enable profiling of cuda graph capture.”
`--debug-cuda-graph`	用 breakable 图插入 graph break，让每个算子仍走 eager 但保留捕获/回放路径，专门用来定位捕获失败

`disable_cuda_graph` 这个**字段**在 0.5.16 里被标成了 `Arg(no_cli=True)` ，不再由字段自动生成命令行参数；但 `--disable-cuda-graph` 这个**命令行 flag** 在 `add_cli_args` 里单独注册为 Deprecated 动作（ `new_flag="--cuda-graph-backend-{decode,prefill}=disabled"` ），仍能识别。 `--cuda-graph-max-bs` 同理，注册为 `--cuda-graph-max-bs-decode` 的 Deprecated 别名。**字段被隐藏与 flag 被删除是两回事**，读源码时要分清。

这条差异的教训比这条差异本身重要：引擎的参数名会随版本改，而文档和教程的更新总是滞后。判断一个 flag 在你锁定的版本上到底存不存在，唯一可靠的办法是去看那个 tag 的源码，或者跑一次 `--help` 抓一下：

```
source "$CH25_ROOT/.venv-sglang/bin/activate"
python -m sglang.launch_server --help 2>&1 | grep -i "cuda-graph"
```

这条命令的输出就是所装版本的权威名单，比任何文档都准。

五、复算捕获尺寸与成本

新建 `$CH25_ROOT/graph/capture_sizes.py` ：

```
"""按 vLLM v0.26.0 源码里的阶梯公式复算 CUDA Graph 捕获尺寸表，并给出 padding 浪费与显存成本。纯标准库。
公式出处: vllm/config/compilation.py 中 max_cudagraph_capture_size 的文档串:
    [1, 2, 4] + list(range(8, 256, 8)) + list(range(256, max_cudagraph_capture_size + 1, 16))
默认 max_cudagraph_capture_size = min(max_num_seqs * 2, 512)
用法: python capture_sizes.py --max-num-seqs 128 [--capture-gib 1.83]
      [--eager-us 12800 --graph-us 9600]
"""
import argparse


def ladder(max_size):
    sizes = [1, 2, 4] + list(range(8, 256, 8)) + list(range(256, max_size + 1, 16))
    return [s for s in sizes if s <= max_size]


def pad_stats(sizes, upper):
    """batch size 在 1..upper 上均匀分布时，被向上取整到最近捕获尺寸带来的平均浪费。"""
    tot_real = tot_pad = 0
    miss = 0
    for b in range(1, upper + 1):
        cand = [s for s in sizes if s >= b]
        if not cand:
            miss += 1
            continue
        tot_real += b
        tot_pad += min(cand)
    return tot_real, tot_pad, miss


def main():
    ap = argparse.ArgumentParser()
    ap.add_argument("--max-num-seqs", type=int, default=128)
    ap.add_argument("--max-capture-size", type=int, default=None,
                    help="不填则按源码默认 min(max_num_seqs*2, 512)")
    ap.add_argument("--capture-gib", type=float, default=None,
                    help="启动日志里 'Graph capturing finished ... took X GiB' 的 X")
    ap.add_argument("--eager-us", type=float, default=None, help="实测 eager 单步 decode 耗时(us)")
    ap.add_argument("--graph-us", type=float, default=None, help="实测 graph 单步 decode 耗时(us)")
    ap.add_argument("--capture-secs", type=float, default=None, help="实测捕获耗时(s)")
    a = ap.parse_args()

    mx = a.max_capture_size or min(a.max_num_seqs * 2, 512)
    sizes = ladder(mx)

    print("=" * 70)
    print(f"max_num_seqs={a.max_num_seqs}  ->  max_cudagraph_capture_size={mx}"
          + ("（源码默认式 min(max_num_seqs*2, 512)）" if a.max_capture_size is None else "（手动指定）"))
    print(f"捕获尺寸个数 = {len(sizes)}   最小 {sizes[0]}   最大 {sizes[-1]}")
    print(f"前 12 个: {sizes[:12]}")
    print(f"后 8 个 : {sizes[-8:]}")
    print("-" * 70)
```

```
    real, pad, miss = pad_stats(sizes, min(mx, a.max_num_seqs))
    print(f"batch size 均匀落在 1..{min(mx, a.max_num_seqs)} 时，向上取整到捕获尺寸：")
    print(f"  实际 token 行数合计 {real}，padding 后 {pad}，平均多算 {(pad/real-1)*100:.1f}%")
    if miss:
        print(f"  有 {miss} 个 batch size 超出最大捕获尺寸，这些请求走非图路径")
    print("  说明：这只刻画 batch 维的取整浪费，不含 attention 那部分随序列长度变化的成本。")

    if a.capture_gib is not None:
        print("-" * 70)
        print(f"捕获显存 {a.capture_gib:.2f} GiB / {len(sizes)} 个捕获尺寸 "
              f"= 平均每个尺寸 {a.capture_gib * 1024 / len(sizes):.1f} MiB")
        if a.capture_secs is not None:
            print(f"捕获耗时 {a.capture_secs:.0f} s = 平均每个尺寸 "
                  f"{a.capture_secs * 1000 / len(sizes):.0f} ms（这部分计入服务冷启动时间）")

    if a.eager_us is not None and a.graph_us is not None:
        print("-" * 70)
        save = a.eager_us - a.graph_us
        print(f"实测单步 decode：eager {a.eager_us:.2f} us -> graph {a.graph_us:.2f} us，"
              f"每步省 {save:.2f} us，加速比 {a.eager_us / a.graph_us:.2f}x")
        if save > 0:
            print(f"生成 512 个 token 的单请求累计省 {save * 512 / 1000:.1f} ms")
        else:
            print("图模式没有更快：先确认捕获真的生效（看启动日志），再确认 batch 落在捕获尺寸内。")
    else:
        print("-" * 70)
        print("未提供 --eager-us / --graph-us：收益部分到此为止。")
        print("kernel launch 开销必须实测，不能用经验值代入：它随模型层数、kernel 数量、")
        print("CPU 主频、驱动版本变化，换一台机器就是另一个数。")


if __name__ == "__main__":
    main()
```

以下为实测输出：

```
python capture_sizes.py --max-num-seqs 128
```

```
========================================================================
max_num_seqs=128  ->  max_cudagraph_capture_size=256（源码默认式 min(max_num_seqs*2, 512)）
捕获尺寸个数 = 35   最小 1   最大 256
前 12 个: [1, 2, 4, 8, 16, 24, 32, 40, 48, 56, 64, 72]
后 8 个 : [200, 208, 216, 224, 232, 240, 248, 256]
------------------------------------------------------------------------
batch size 均匀落在 1..128 时，向上取整到捕获尺寸：
  实际 token 行数合计 8256，padding 后 8683，平均多算 5.2%
  说明：这只刻画 batch 维的取整浪费，不含 attention 那部分随序列长度变化的成本。
------------------------------------------------------------------------
未提供 --eager-us / --graph-us：收益部分到此为止。
kernel launch 开销必须实测，不能用经验值代入：它随模型层数、kernel 数量、
CPU 主频、驱动版本变化，换一台机器就是另一个数。
```

脚本在这里主动停住，是刻意设计的。 尺寸表能纯算，收益不能：launch 开销随模型层数、kernel 数量、CPU 主频、驱动版本变化，没有任何“经验值”能代入。

这个 35 是个有用的数，但要看清它的前提：`--max-num-seqs 128` 时算出 35 个捕获尺寸（`min(128*2, 512) = 256`），`FULL_AND_PIECEWISE` 模式下每个尺寸要捕两轮（PIECEWISE 与 FULL），每一轮都要占显存、都要花时间。`max_num_seqs` 提到 256 或更大，上限就顶到 512，尺寸个数涨到 **51**。

本章实测服务上是 51 不是 35：实测版本在 H100 上 serve 时 `max_num_seqs` 取到 1024，`min(1024*2, 512)` 被 512 截住，启动日志里 `max_cudagraph_capture_size: 512`、`cudagraph_capture_sizes` 正好 51 个元素。锁定版本 0.26.0 的源码默认值是 128（`config/scheduler.py` 的 `DEFAULT_MAX_NUM_SEQS`），跑出来会是 35。**这个数不要背，去自己的启动日志里数**：

```
grep -o 'cudagraph_capture_sizes[^]]*]' "$CH25_ROOT"/bench/serve_baseline.log
```

捕获成本随尺寸个数上升，这是调大并发时一个容易被忽略的隐性代价。

六、简化 Runtime：把捕获与回放自行写一遍

新建 `$CH25_ROOT/graph/mini_graph_runtime.py`：

```
"""一个最小的 CUDA Graph decode runtime：静态 buffer + 侧流预热 + 捕获 + 回放，
对比 eager 与 graph 两种执行方式在同一段 decode 计算上的单步耗时。
用法: python mini_graph_runtime.py --layers 28 --hidden 3584 --batch 16 --iters 200
"""
import argparse

import torch
import torch.nn as nn


class DecodeBlock(nn.Module):
    """按 Qwen2.5-7B 的形状搭一层 decode 计算的骨架：RMSNorm + QKV/O 投影 + SwiGLU。
    这里只关心 kernel 的数量与形状，不追求和真实模型数值一致。"""

    def __init__(self, hidden, q_heads, kv_heads, head_dim, inter):
        super().__init__()
        self.w_q = nn.Linear(hidden, q_heads * head_dim, bias=True)
        self.w_kv = nn.Linear(hidden, 2 * kv_heads * head_dim, bias=True)
        self.w_o = nn.Linear(q_heads * head_dim, hidden, bias=False)
        self.w_gate = nn.Linear(hidden, inter, bias=False)
        self.w_up = nn.Linear(hidden, inter, bias=False)
        self.w_down = nn.Linear(inter, hidden, bias=False)
        self.g1 = nn.Parameter(torch.ones(hidden))
        self.g2 = nn.Parameter(torch.ones(hidden))

    @staticmethod
    def _rms(x, g):
        return x * torch.rsqrt(x.float().pow(2).mean(-1, keepdim=True) + 1e-6).to(x.dtype) * g

    def forward(self, x):
        h = self._rms(x, self.g1)
        q = self.w_q(h)
        _ = self.w_kv(h)                      # 站位：真实实现这里写 KV Cache
        x = x + self.w_o(q)
        h = self._rms(x, self.g2)
        return x + self.w_down(torch.nn.functional.silu(self.w_gate(h)) * self.w_up(h))


def timed(fn, iters, warmup=20):
    for _ in range(warmup):
        fn()
    torch.cuda.synchronize()
    s, e = torch.cuda.Event(enable_timing=True), torch.cuda.Event(enable_timing=True)
    s.record()
    for _ in range(iters):
        fn()
    e.record()
    torch.cuda.synchronize()
    return s.elapsed_time(e) * 1000.0 / iters          # ms -> us/步


def main():
```

```
    ap = argparse.ArgumentParser()
    ap.add_argument("--layers", type=int, default=28)
    ap.add_argument("--hidden", type=int, default=3584)
    ap.add_argument("--q-heads", type=int, default=28)
    ap.add_argument("--kv-heads", type=int, default=4)
    ap.add_argument("--inter", type=int, default=18944)
    ap.add_argument("--batch", type=int, default=16)
    ap.add_argument("--iters", type=int, default=200)
    ap.add_argument("--break-capture", action="store_true",
                    help="主动在捕获区里做一次 CPU 同步，看捕获怎么失败")
    a = ap.parse_args()

    assert torch.cuda.is_available(), "本脚本要在 GPU 机器上跑"
    torch.manual_seed(0)
    head_dim = a.hidden // a.q_heads
    dev, dt = "cuda", torch.bfloat16
    layers = nn.ModuleList([
        DecodeBlock(a.hidden, a.q_heads, a.kv_heads, head_dim, a.inter).to(dev, dt)
        for _ in range(a.layers)]).eval()

    def step(x):
        for blk in layers:
            x = blk(x)
        return x

    # ---- 静态输入输出 buffer：图回放时地址必须不变，所以先固定下来 ----
    static_in = torch.randn(a.batch, a.hidden, device=dev, dtype=dt)

    with torch.no_grad():
        eager_us = timed(lambda: step(static_in), a.iters)

    # ---- 预热必须放在侧流上 ----
    # 官方要求捕获发生在非默认流上；预热同样走侧流，是为了让预热期分配的显存块落进
    # 捕获要用的那个内存池，避免回放时地址对不上。
    s = torch.cuda.Stream()
    s.wait_stream(torch.cuda.current_stream())
    with torch.cuda.stream(s), torch.no_grad():
        for _ in range(3):
            static_out = step(static_in)
    torch.cuda.current_stream().wait_stream(s)

    # ---- 捕获 ----
    g = torch.cuda.CUDAGraph()
    try:
        with torch.no_grad(), torch.cuda.graph(g):
            static_out = step(static_in)
            if a.break_capture:
                # 捕获区内禁止 CPU-GPU 同步；.item() 会把 GPU 结果拉回 CPU
                print(float(static_out.sum().item()))
    except RuntimeError as e:
        print("捕获失败：", e)
        print("这正是 CUDA Graph 的硬约束：捕获区内不能同步 CPU、不能有依赖数据的分支、"
              "不能有变化的形状。去掉 --break-capture 再跑。")
```

```
        return

    def replay():
        g.replay()

    graph_us = timed(replay, a.iters)

    print("=" * 66)
    print(f"层数 {a.layers}  hidden {a.hidden}  batch {a.batch}  dtype {dt}")
    print(f"eager  单步 {eager_us:9.2f} us")
    print(f"graph  单步 {graph_us:9.2f} us")
    print(f"每步省 {eager_us - graph_us:.2f} us，加速比 {eager_us / graph_us:.2f}x")
    print("-" * 66)
    print("回放只做一次 cudaGraphLaunch；eager 每步要走一遍完整的提交路径。")
    print("差值就是 launch 开销的上界估计。它占 eager 的比例随 batch 如何变化，")
    print("取决于单个 kernel 的耗时是否远大于 launch 开销：kernel 短时比例随 batch 下降，")
    print("kernel 已经足够长时比例基本持平（本章实测四档都在 13%～16%）。")
    # 校验必须换一批新输入，否则测不出任何东西：
    # static_out 在捕获那一刻就已经用旧的 static_in 算过一遍，
    # 如果不换输入，g.replay() 生不生效，diff 都会是 0。
    print("验证输出一致（换一批新输入，回放写进同一块 static_out）：")
    new_in = torch.randn_like(static_in)
    with torch.no_grad():
        ref = step(new_in)                # eager 对新输入的结果
    static_in.copy_(new_in)               # 新数据必须拷进捕获时那块地址
    g.replay()                            # 回放写进 static_out
    torch.cuda.synchronize()
    diff = (static_out.float() - ref.float()).abs().max().item()
    print(f"  max|graph - eager| = {diff:.3e}")


if __name__ == "__main__":
    main()
```

```
export CUDA_VISIBLE_DEVICES=0
cd "$CH25_ROOT/graph"
for b in 1 4 16 64; do python mini_graph_runtime.py --batch $b --iters 200; done
python mini_graph_runtime.py --batch 16 --break-capture
```

实测（H100，torch 2.9.1+cu128，脚本在 `projects/pr3/graph/mini_graph_runtime.py`）：四档数值来自 `projects/pr3/graph/capture_scan.txt`，`--break-capture` 按预期在捕获区因 `.item()` 同步失败。

捕获扫描记录（实测）

batch	eager 单步 (us)	graph 单步 (us)	每步省 (us)	加速比	省下部分占 eager 的比例
1	7019.3	6068.4	950.9	1.16×	**13.5%**
4	7371.3	6240.4	1130.9	1.18×	15.3%
16	7993.0	6786.2	1206.8	1.18×	15.1%
64	8281.3	6992.2	1289.1	1.18×	15.6%

`--break-capture` 真实报错：`CUDA error: operation failed due to a previous error during capture`（`cudaErrorStreamCaptureInvalidated`）。捕获区内 `.item()` 把 GPU 结果拉回 CPU，捕获立即失效。

先说清这个差值里装了什么。 把每步省下的时间直接叫作 kernel launch 开销是不准确的：28 层模型一步大约发 280 到 560 个 kernel，950 us 除下来每次要 1.7 到 3.4 us，而驱动侧 `cudaLaunchKernel` 本身通常在亚微秒到 1.5 us 量级，并且与 GPU 执行重叠。这个差值的主体是**整条 CPU 侧提交路径**：Python 循环、PyTorch 的算子分发、显存分配器、每层里临时张量的创建与释放，CUDA Graph 把这一整条路径都省掉了，kernel launch 只是其中一部分。脚本打印的措辞是"上界估计"，就是这个意思。

这张表要看两件事，不要只看加速比。 每步省下的绝对微秒在 0.95–1.29 ms 之间波动，没有随 batch 线性增长：launch 次数不随 batch 变，与预期一致。但“占 eager 的比例随 batch 迅速下降”在实测中**没有出现**：四档都停在 13–16%。不是测错了，是这张 H100 上 Qwen2.5-7B 形状的 Linear（hidden 3584、inter 18944、28 层）在 batch=1 时已经不是大量微秒级小 kernel：eager 从 7.0 ms 只涨到 8.3 ms，kernel 自身耗时已经足够长，launch 开销被摊薄。图仍然稳定快出 **1.16–1.18×**，只是在大显存、大层数的配置上没有那么显著。教材中的结论“CUDA Graph 在小 batch 下收益最大”，要在 kernel 真的很短的时候才看得清楚；要把收益放到真实调度里，看项目三引擎侧 A/B（2522 vs 1320 tok/s）。原始输出：`projects/pr3/graph/capture_scan.txt`。

三处写法上的依据，都来自 PyTorch 官方 CUDA 语义文档的 CUDA Graphs 一节：

- **预热必须放在侧流上**，官方示例就是 `s = torch.cuda.Stream(); s.wait_stream(torch.cuda.current_stream()); with torch.cuda.stream(s): ...` 这个形状；官方同时写明 “Capture must occur on a non-default stream”，`torch.cuda.graph` 这个上下文管理器内部已经替你处理了捕获流。
- **捕获区的硬约束**官方列了五条：不能同步 CPU（"Ops that synchronize the CPU with the GPU (e.g., `.item()` calls) are prohibited"）、每次回放读写同一批虚拟地址、禁止依赖数据的动态控制流、禁止动态形状、同一进程内同时只能有一次捕获在进行。`--break-capture` 就是把第一条主动违反一次，让人亲眼看到报错长什么样。
- `torch.cuda.graph` 在 torch 2.11 的签名是 `torch.cuda.graph(cuda_graph, pool=None, stream=None, capture_error_mode='global')`，`capture_error_mode` 可取 `global` / `thread_local` / `relaxed`。**脚本里一个额外参数都没传**，是为了让代码在锁定版本上一定能跑，新版本加的参数在 2.11 上不存在。

七、回到引擎：A/B 量化端到端收益

简化 Runtime 量的是纯 launch 开销的上界。真实引擎里还有调度、采样、KV 管理，收益会被稀释。用 `--enforce-eager` 反着做一次 A/B。

本项目的 A/B 是独立的一组压测，负载与项目二不同，见下方口径表。基线是默认启动（图开启、不加任何 CUDA Graph 相关参数），日志落到 `serve_graph.log`；实验组只多一个 `--enforce-eager`。

先说清楚这个开关到底关了什么。 `--enforce-eager` 不只关 CUDA Graph，它同时把 torch.compile 也关掉了。vLLM 的启动日志会明说这件事：

```
Enforce eager set, disabling torch.compile and CUDAGraphs.
This is equivalent to setting -cc.mode=none -cc.cudagraph_mode=none
```

源码在 `vllm/config/vllm.py`，`enforce_eager` 为真时把 `compilation_config.mode` 与 `cudagraph_mode` 一起置成 `NONE`。**所以下面这组 A/B 量到的是这两项加起来的收益，不是 CUDA Graph 一项的收益。**要单独量 CUDA Graph，实验组得换成只关图、留着编译的写法：

```
--compilation-config '{"cudagraph_mode": "NONE"}'
```

本章没有跑这一组，所以本项目的数字按两项合计报。这条留作自选扩展，做完就能把两项拆开。

```
source "$CH25_ROOT/.venv-vllm/bin/activate"
vllm serve "$CH25_ROOT/models/Qwen2.5-7B-Instruct" \
    --served-model-name Qwen2.5-7B-Instruct \
    --host 0.0.0.0 --port 8000 --max-model-len 4096 \
    --gpu-memory-utilization 0.85 \
    --enforce-eager \
    2>&1 | tee "$CH25_ROOT/bench/serve_eager.log"
```

两组压测命令相同，只换 `--result-filename`（基线组 `cg_graph.json`，实验组 `cg_eager.json`）：

```
vllm bench serve --model Qwen2.5-7B-Instruct --base-url http://127.0.0.1:8000 \
    --dataset-name random --random-input-len 64 --random-output-len 256 \
    --num-prompts 64 --max-concurrency 16 --request-rate inf \
    --seed 0 --ignore-eos --percentile-metrics ttft,tpot,itl,e2el \
    --metric-percentiles 50,95,99 --save-result --result-filename cg_eager.json
```

负载口径与项目二的区别

	项目二（调度参数 A/B）	本项目（torch.compile + CUDA Graph A/B）
输入 / 输出长度	1024 / 128	64 / 256
请求数 / 并发	256 / 32	64 / 16
`--max-model-len`	8192	4096
`--gpu-memory-utilization`	0.88	0.85

本项目的负载 decode 占主导（输出长度是输入的 4 倍、并发小），这是 CUDA Graph 收益最能显现的形态，也是隔离 launch 开销这一个变量的合理选择。换成项目二那种 1024 输入的负载，prefill 走分段图且本身耗时大，收益比例会小于本项目的数字。两组数字不可横向比较。

启动时从两份日志里各抓一次：

```
grep -E "Graph capturing finished" "$CH25_ROOT"/bench/serve_*.log
```

基线日志里应该有 `Graph capturing finished in <秒> secs, took <GiB> GiB`（`gpu_model_runner.py::capture_model` 里的 `logger.info_once`），`--enforce-eager` 那份不该有：它会走 `capture_model()` 开头那个 `cudagraph_mode == NONE` 的分支直接返回 0，并打一条 “Skipping CUDA graph capture” 的 warning。

把数字填回测算脚本：

```
python capture_sizes.py --max-num-seqs <启动日志里的实际值> \
    --capture-gib <日志里的 GiB> --capture-secs <日志里的秒数> \
    --eager-us <eager 单步> --graph-us <graph 单步>
```

输出形如（下面四个数是示意值，不是任何机器上的实测值，替换为你的日志里读到的数）：

```
--------------------------------------------------------------------------
捕获显存 1.83 GiB / 51 个捕获尺寸 = 平均每个尺寸 36.7 MiB
捕获耗时 42 s = 平均每个尺寸 824 ms（这部分计入服务冷启动时间）
--------------------------------------------------------------------------
实测单步 decode：eager 12800.00 us -> graph 9600.00 us，每步省 3200.00 us，加速比 1.33x
生成 512 个 token 的单请求累计省 1638.4 ms
```

收益与代价对照（实测）

	图开启（默认）`cg_graph`	**`--enforce-eager` `cg_eager`**
输出吞吐 tok/s	**2522.4**	**1319.6**
mean TPOT (ms)	**6.06**	**11.55**
ITL P99 (ms)	**8.13**	**20.42**
mean TTFT (ms)	**73.8**	**152.9**
服务冷启动捕获耗时 (s)	**3**	0（不捕获）
捕获显存 (GiB)	**0.01**	0

来源：`reports/cudagraph.md`；捕获两行来自启动日志 `bench/serve_graph.log`（`Graph capturing finished in 3 secs, took 0.01 GiB`；本次 51 个捕获尺寸，PIECEWISE 与 FULL 各捕获一轮，日志里两条进度条各自 51/51）。吞吐约 **1.91×**，TPOT 约 **0.52×**。

这个 1.91× 是 torch.compile 与 CUDA Graph 两项合计的收益，因为 `--enforce-eager` 把两者一起关掉了（见本节开头）。面试里要照这个口径说；说成 CUDA Graph 单独带来 1.91×，对方追问一句实验组关了什么就穿了。

这张表是本项目的结论：CUDA Graph 不是无代价的收益，它用启动时的捕获耗时和一块预留显存换 decode 的每步延迟。**本次实测的代价很小：3 秒、0.01 GiB**，因为 vLLM 默认的 `FULL_AND_PIECEWISE` 模式让两轮捕获的全部图共享同一个内存池。

`took 0.01 GiB` 这个数的语义要说准。 引擎在内存 profiling 阶段就已经估算过图的显存并从 KV 池预算里扣掉了（`VLLM_MEMORY_PROFILER_ESTIMATE_CUDAGRAPHS` 默认为 `True`），所以启动日志里这个 0.01 GiB 是**捕获前后可用显存的差值**，也就是预留之外的那一点增量，不是图的总显存成本。想知道总成本，要对比开图与关图两组启动日志里的 `Available KV cache memory`，差值才是图实际占走的部分。本章只留了开图那一组的日志，所以这个数没有拿到。**但代价随捕获尺寸数量与模型规模增长**：捕获尺寸列表越长、模型越大、走整图捕获的比例越高，这两个数都会上升，必须在你自己的配置上从启动日志读，不能照搬本表。什么时候不该开，答案就在这张表里：显存已经紧张、KV 池是瓶颈的部署，要求秒级冷启动的弹性伸缩场景，以及 batch 一直很大的离线批处理，三种情况下这次权衡都可能不划算。

一个必须澄清的误解：捕获占的显存不是事后从 KV Cache 中挤占的。`VLLM_MEMORY_PROFILER_ESTIMATE_CUDAGRAPHS` 默认为 `True`，引擎在内存 profiling 阶段就会先估算图的显存开销（`gpu_model_runner.py::profile_cudagraph_memory`），再据此算 KV Cache 池大小。所以关掉图之后 KV 池会变大、可承载并发上升。**A/B 的时候这两个变化是同时发生的，不能把吞吐差异全算到 launch 开销头上**。要隔离，得用 `--num-gpu-blocks-override` 把两组的 KV 池强制对齐。

本次 A/B 里这个变量的量级可以直接算出来。 启动日志给出 KV 池 50.64 GiB（948,256 token），图显存 0.01 GiB，占 KV 池的 0.02%；负载侧并发 16、每条请求最长 64 + 256 = 320 token，KV 工作集不超过 5,120 token，占池容量的 0.5%。**两组的 KV 池都远没有用满**，所以即使关图释放出的显存比 0.01 GiB 多得多，也没有并发可以被它多装下去，这条路径在本次测量里不会产生吞吐差异。这个结论依赖的是"工作集只占池容量 0.5%"这个量级，不依赖 0.01 GiB 那个数本身。这项核对每次 A/B 都要做，不能默认第二变量可忽略，也不能默认它不可忽略。

八、必做核心与自选扩展

必做核心是四步：读源码确认本版本的模式、默认值与参数名（尤其是 SGLang 那组已经改名的 flag），用 `capture_sizes.py` 复算出捕获尺寸表与图数量，在 `mini_graph_runtime.py` 里跑通四个 batch 的 eager/graph 对比并跑一次 `--break-capture` 看约束，最后用 `--enforce-eager` 做引擎侧 A/B 并把收益代价表填满。少了最后一步，就只证明了 CUDA Graph 能降低 launch 开销，没有证明它在真实服务里的实际收益。

自选扩展往两个方向做，前者面向显存预算，后者面向后端选型：

1. **内存池共享**：给 `torch.cuda.graph` 传同一个 `pool=torch.cuda.graph_pool_handle()` 捕获多个 batch 尺寸的图，量化共享内存池能省下多少显存。这正是引擎捕获几十张图却没有几十倍显存开销的原因。
2. **分段图**：把简化 Runtime 里的某一个算子（模拟 attention）主动留在图外 eager 执行，其余进图，复现 `PIECEWISE` 模式，量化整图与分段图的收益差。这个差值决定了一个引擎在 attention 后端不支持整图捕获时还剩多少收益。
3. **把 torch.compile 与 CUDA Graph 拆开**：实验组改用 `--compilation-config '{"cudagraph_mode": "NONE"}'`，只关图、留着编译，再跑一次同口径压测。它与本节那组 `--enforce-eager` 的差值，就是 torch.compile 单独贡献的那一份。做完这一组，1.91× 才能拆成两个数。

九、逐步证据链（面试可复述）

> 把测什么、看到什么、为何如此、改哪一处、复测收益这五步整理成可在面试中复述的证据链。数字来自 `projects/pr3/reports/` 。项目二已经用 Nsight Systems（nsys，时间线）/ Nsight Compute（ncu，单 kernel 计数器）证明稳态主路径受限于显存带宽、调度参数翻倍无效；本项目把 **CUDA Graph（把一串 kernel 提交录成一张图、之后一次回放）的 launch（CPU 提交 kernel）收益**从时间线印象落成 eager（eager mode，不启用 CUDA Graph、每个 kernel 单独提交）↔ graph 端到端表。本文件后文没有单独的手写 runtime 逻辑分析一节，手写最小 runtime 的读数收在本节最后一小节。

本节术语

术语	全称	含义
CUDA Graph	CUDA Graph	把一串 kernel 提交录成一张图，之后一次 `cudaGraphLaunch` 整张回放，省的是 CPU 往返，不是 GPU 算力
launch	kernel launch	CPU 把一个 kernel 交给 GPU 的那一次提交；decode 每步要发几百次，单次虽短但会叠起来
eager	eager mode	不启用 CUDA Graph，每个 kernel 单独 launch；本项目用 `--enforce-eager` 强制走这条路
TTFT	Time To First Token	从发出请求到收到第一个 token 的时间，即首包延迟
ITL	Inter-Token Latency	相邻两个输出 token 之间隔了多久
E2EL	End-to-End Latency	单条请求从发出到全部收完的总时间
TPOT	Time Per Output Token	平均每个输出 token 花多久
Event 计时	`torch.cuda.Event`	在 GPU 时间线上打两个点再相减，比 `time.time()` 干净，手写 runtime 用它

步骤 1｜锁定口径

这一步在干什么：CUDA Graph 省的是 CPU 侧的 launch，不是 GEMM 算力。所以验收不能只看单个 kernel 的耗时变化，而要看端到端的 tok/s、TPOT（Time Per Output Token）、ITL（Inter-Token Latency）。还要事先承认一件事：关掉图之后，引擎原来预留给图的那块显存会回到 KV Cache 池，吞吐差里可能混入了第二个变量。

动作：引擎侧做 A/B：`cg_eager.json` 用 `--enforce-eager` 禁用 CUDA Graph，`cg_graph.json` 走默认图；同模型、同 bench 命令。本步不采集 nsys：项目二已经把耗时排行与瓶颈类型确认为访存受限，本项目要的是 launch 合并后的端到端数据。

本步产生的文件

路径	谁写的	看哪列	得到哪个数
`projects/pr3/reports/cudagraph.md`	人写的读数笔记（内嵌 JSON）	`project` / `rows[].file`	项目名 `ch25_pr3_cudagraph`；两组文件名 `cg_eager.json` / `cg_graph.json`
`projects/pr3/bench/closed_loop_summary.json`	闭环脚本汇总	与上一行相同的 `rows`	`status=gpu_run_complete`
—	—	—	**不采集 nsys**，证据在上面两份 JSON 和后面的 bench 文件

读数：统一口径为仅切换 CUDA Graph 开关。面试时必须主动说：未对齐 KV 池时，要先核对图显存与 KV 工作集相对池容量的量级；本次两者都不到 1%，第二变量可以排除。

为什么能推下一步：对照基准还没有，先跑 eager 组，才能知道逐 kernel 提交的开销有多大。

步骤 2｜基线：eager 给出对照基准

这一步在干什么：eager 是逐个 kernel 提交的上限成本。没有这个对照基准，后面 graph 的 2522 tok/s 只是一个孤立的数值，无法说明快了多少、快在哪。

动作：`--enforce-eager` 起服务，跑与 graph 组逐字相同的压测。

本步产生的文件

路径	谁写的	看哪列	得到哪个数
`projects/pr3/bench/cg_eager.json`	vLLM bench	`output_throughput` / `mean_tpot_ms` / `p99_itl_ms` / `mean_ttft_ms`	**1319.6 tok/s**、mean TPOT **11.55 ms**、P99 ITL **20.42 ms**、mean TTFT **152.9 ms**
`projects/pr3/reports/cudagraph.md`	人写的读数笔记	`rows` 里 `file=cg_eager.json` 那一行	与 JSON 对齐的四个数
`projects/pr3/bench/serve_eager.log`	服务启动日志	搜 `enforce-eager` / Graph capturing	确认图没被捕获，这一组真的是 eager
—	—	—	**不采集 nsys**，证据在 `cg_eager.json`

读数：1319.6 tok/s、TPOT 11.55 ms。这就是每步 decode 都由 CPU 逐个提交 kernel 的端到端代价。

为什么能推下一步：对照基准有了，下一步用默认图复测，差值才能解释成 launch 被合并。

步骤 3｜剖析：graph 组，看 launch 合并长什么样

这一步在干什么：默认路径本来就开着 CUDA Graph（vLLM 默认 `FULL_AND_PIECEWISE`），所以实验组不是启用 CUDA Graph，而是保持默认不关闭。如果图真的在吃 launch 开销，你应该同时看到吞吐上去、TPOT / ITL 下来，因为每步少了几百次 CPU 往返。

动作：关掉 `--enforce-eager`，默认图复测。

本步产生的文件

路径	谁写的	看哪列	得到哪个数
`projects/pr3/bench/cg_graph.json`	vLLM bench	`output_throughput` / `mean_tpot_ms` / `p99_itl_ms` / `mean_ttft_ms`	**2522.4 tok/s**（≈**1.91×**）、mean TPOT **6.06 ms**（≈**0.52×**）、P99 ITL **8.13 ms**、mean TTFT **73.8 ms**
`projects/pr3/reports/cudagraph.md`	人写的读数笔记	`rows` 里 `file=cg_graph.json` 那一行	与 JSON 对齐的四个数
`projects/pr3/bench/serve_graph.log`	服务启动日志	搜 `Graph capturing finished`	捕获耗时和占用 GiB（代价侧，稳态 bench 不含这两项）
—	—	—	**不采集 nsys**，证据在 `cg_graph.json`；项目二的 ncu 已经证明 kernel **内部**受限于显存带宽

读数：2522 vs 1320 tok/s，TPOT 6.06 vs 11.55 ms。ITL / TPOT 一起下降，符合逐个 kernel 提交被合并的解释，而不是 kernel 自身变快。注意实验组同时关掉了 torch.compile，所以这个差值里也含编译带来的那一份。这与项目二不矛盾：项目二证明 **kernel 内部**已经受限于访存带宽，本项目证明 **kernel 之间**的 CPU 提交开销仍然可以被 CUDA Graph 合并掉，两层瓶颈可以同时存在。

为什么能推下一步：收益方向对了，但只改一处这一条还没有论证充分；下一步要点名第二变量并核对它的量级。

步骤 4｜只改一处，并点名第二变量

这一步在干什么：这组 A/B 仅切换 CUDA Graph 开关。禁用 CUDA Graph 会让预留的图显存回到 KV 池，能同时在途的请求数可能变多，吞吐差里就混入了第二件事。彻底隔离要用 `--num-gpu-blocks-override` 把两组 KV 池写成同一个数；本次不必对齐也能排除：图显存 0.01 GiB 只占 KV 池 50.64 GiB 的 0.02%，负载的 KV 工作集只占池容量的 0.5%，两组的 KV 池都远未用满。

动作：本项目的稳态数字来自未对齐 KV 的两组；面试必须主动说出第二变量，并给出它在本次负载下可忽略的量级依据。

本步产生的文件

路径	谁写的	看哪列	得到哪个数
`projects/pr3/bench/cg_eager.json` / `cg_graph.json`	vLLM bench	四个端到端字段并排	没有第三份 KV 池对齐后复测的 JSON
`projects/pr3/bench/closed_loop_summary.json`	闭环脚本汇总	`rows` 只有两行	确认只做了 eager↔graph，没有第三组
—	—	—	**不采集 nsys**，证据仍在这两份 bench

读数：没有对齐 KV 的第三组，但量级核对已排除 KV 池变化的贡献，2522 vs 1320 可以读成 launch 合并的收益；它不是算力提升，GPU 侧每个 kernel 的耗时没有变。

为什么能推下一步：变量说清了，就可以按事先口径判定该收益代价表在本负载下是否成立，并补上手写 runtime 的微观证据。

步骤 5｜复测与收益判定

这一步在干什么：graph 组在吞吐、TPOT、ITL、TTFT 上全面胜出。代价（冷启动捕获、图显存）要在启动日志另计，本项目以稳态 bench 为主。反转条件也要说在前面：显存不足导致 KV 不足，或弹性扩容无法承受这段捕获耗时（本次 3 秒，捕获尺寸多、模型大时会显著上升），这时可以强制 eager。

动作：把两组四个数收进 `closed_loop_summary.json`，和 `cudagraph.md` 对拍。

本步产生的文件

路径	谁写的	看哪列	得到哪个数
`projects/pr3/bench/closed_loop_summary.json`	闭环脚本汇总	`rows[0]` vs `rows[1]`	eager 1319.6 / 11.55 ms；graph 2522.4 / 6.06 ms
`projects/pr3/reports/cudagraph.md`	人写的读数笔记	同上	与 summary 逐字段一致
—	—	—	**不采集 nsys**，判定看端到端 JSON，不看时间线

读数：Graph **2522** vs eager **1320** tok/s，mean TPOT **6.06** vs **11.55** ms。收益幅度与负载形态有关：本项目输出长、并发小，decode 占主导；换成项目二那种 1024 输入的负载，prefill 走分段图且本身耗时大，收益比例会小于 1.91×。链式结论：项目二证明 kernel 内部受限于访存带宽、调度参数翻倍无效 → 本项目保持默认路径、用 `--enforce-eager` 反着做 A/B → 复测 tok/s 与 TPOT 如本表，收益按 torch.compile 与 CUDA Graph 两项合计报。

为什么能推到结论：端到端数据已经足以判定本负载下启用 CUDA Graph 有收益；还缺一层图的捕获过程与失败形态，那是手写最小 runtime 的事。

步骤 6｜手写最小 runtime：看哪个文件

这一步在干什么：引擎侧 A/B 只能告诉你启用 CUDA Graph 更快，不能让你实际触发一次捕获约束。`mini_graph_runtime.py` 用 Qwen2.5-7B 的形状（28 层 / hidden 3584 / inter 18944）搭一层 decode 骨架，静态 buffer + 侧流预热 + 捕获 + 回放，用 `torch.cuda.Event` 计时。本文件没有单独的手写 runtime 逻辑分析一节，读数都在这里。

动作：对 batch ∈ {1, 4, 16, 64} 各跑 200 iter；再加一次 `--break-capture`，主动在捕获区里 `.item()`，看它怎么失败。

本步产生的文件

路径	谁写的	看哪列	得到哪个数
`projects/pr3/graph/mini_graph_runtime.py`	手写脚本	`timed()` 里的 `torch.cuda.Event`；`--break-capture` 分支	计时口径是 Event 相减后的 us/步；捕获区 `.item()` 会触发失败
`projects/pr3/graph/capture_scan.txt`	脚本 stdout 落盘	`eager 单步` / `graph 单步` / `每步省` / `加速比`	batch 1：**7019.3 vs 6068.4 us**（省 950.9，1.16×）；batch 16：**7993.0 vs 6786.2 us**（省 1206.8，1.18×）；batch 64：**8281.3 vs 6992.2 us**（1.18×）
`projects/pr3/reports/capture_scan.md`	人写的读数笔记	四档表 + `--break-capture` 行	与 txt 对齐；失败信息含 `cudaErrorStreamCaptureInvalidated`
—	—	—	**不采集 nsys**，证据在 Event 计时产物 `capture_scan.txt` / `capture_scan.md`

读数：四档加速比都在 1.16–1.18×，绝对节省 0.95–1.29 ms，不随 batch 线性涨：这张卡上该结构在 batch=1 时单步耗时已经足够长，launch 占比没有出现“迅速下降”。`--break-capture` 按预期在捕获区因 `.item()` 同步失败。`max|graph - eager| = 0`，说明回放的结果与 eager 逐字相同。

这个校验必须换一批新输入才有意义，脚本里 `new_in` 与 `static_in.copy_(new_in)` 那两行不能省。原因是 `static_out` 在捕获那一刻就已经用当时的 `static_in` 算过一遍并写好了：如果校验时不换输入，那么无论 `g.replay()` 有没有真的执行，`static_out` 里都是那次结果，diff 恒等于 0。一个恒等于 0 的校验测不出任何东西。判断自己写的校验是否有效，有个简单办法：**把 `g.replay()` 那一行注释掉再跑一次，diff 必须变大**；如果还是 0，说明这个校验是空的。

为什么能推到结论：手写 runtime 把 launch 开销的节省路径与捕获区的同步约束变成可复现的脚本和四档表；引擎侧 2522 vs 1320 是同一件事在真实服务里的放大版。

一句话总结：默认路径 vs `--enforce-eager`（同时关掉 torch.compile 与 CUDA Graph）：2522 vs 1320 tok/s、TPOT 6.06 vs 11.55 ms；kernel 内部受限于访存带宽，不妨碍 CUDA Graph 进一步降低 launch 开销。手写证据看 `projects/pr3/graph/mini_graph_runtime.py` 与 `projects/pr3/reports/capture_scan.md`。

十、排错表

现象	原因	处理
`--disable-cuda-graph` 启动时打 Deprecated 警告（0.5.16）	旧 flag 仍注册但已废弃，自动映射到 `--cuda-graph-backend-{decode,prefill}=disabled`	用 `--disable-decode-cuda-graph`，或 `--cuda-graph-backend-decode disabled`
`--cuda-graph-max-bs` 启动时打 Deprecated 警告（0.5.16）	旧 flag 仍注册但已废弃，是 `--cuda-graph-max-bs-decode` 的别名；0.5.16 把它拆成了 `-decode` / `-prefill` 两个	用 `--cuda-graph-max-bs-decode`
找不到启用 CUDA Graph 的开关	v1 下默认就是 `FULL_AND_PIECEWISE`，本来就开着	A/B 反着做，用 `--enforce-eager` 或 `--compilation-config '{"cudagraph_mode":"NONE"}'`
捕获时报 `.item()` 相关错误	捕获区内做了 CPU-GPU 同步	把同步移出捕获区；官方明确列为禁止操作
捕获成功但回放结果不对	输入没写进静态 buffer，或换了新张量	每次回放前 `static_in.copy_(new_data)`，地址不能变
换一个 batch size 就不再走 CUDA Graph 路径	该尺寸不在捕获列表里	看 `capture_sizes.py` 算出的阶梯；或用 `cudagraph_capture_sizes` 显式指定
服务启动变慢	图捕获在启动阶段完成	看 `Graph capturing finished in ... secs` 那行确认耗时（本章实测 3 秒，51 个尺寸两轮捕获）；不能接受就降 `--max-num-seqs` 缩短捕获尺寸列表，或禁用 CUDA Graph
关掉图之后可用 KV 变大、吞吐反而涨	图显存在内存 profiling 阶段就被预留了	正常。要隔离变量得用 `--num-gpu-blocks-override` 对齐两组 KV 池
nsys 时间线上图里的 kernel 看不见	默认 `--cuda-graph-trace=graph`	换 `node`
ncu 把整张图当成一个 workload	默认 `--graph-profiling graph`	换 `node`

十一、面试深挖点

下面的参考答法里带着本章的实测数。 这些数是在 H100 上跑出来的，换卡、换负载都会变；照搬去讲，对方一追问负载口径就答不上来。哪些能讲、哪些要自己跑出来，见全章末节“把这三个项目写进简历之前”。

Q1｜CUDA Graph 到底省了什么？为什么只用在 decode 上？

考察点：知不知道省的是 CPU 侧开销，还是以为它能加速计算。

参考答法：省的是 CPU 侧提交 kernel 的开销。正常执行时每个 kernel 都要 CPU 走一遍参数打包和驱动调用，一个 28 层的模型一步 decode 要发几百个；CUDA Graph 把这一串记录成一张图，之后一次提交就够。GPU 侧的计算一点没变，所以只在 CPU 提交速率成为瓶颈时才有收益：decode 恰好是这种场景。prefill 不走整图捕获，原因有两个：

1. 它的 kernel 本身跑得久，launch 开销占比低；
2. 它的输入长度千变万化，而图要求形状固定，为每种长度捕一张不现实。

所以引擎的做法是 decode 捕整图、prefill 与混合批走分段图，只把形状固定的算子段录进去。

Q2｜捕获一张图有哪些硬性约束？

考察点：有没有真的写过捕获代码。只用过引擎的人只能背概念。

参考答法：核心是回放与捕获的执行上下文必须完全一致。官方列的五条硬约束：

1. 捕获区内不能有 CPU-GPU 同步，`.item()`、`.cpu()`、`print(tensor)` 都会让捕获失败；
2. 每次回放读写同一批虚拟地址，输入必须先 `copy_` 进静态 buffer，不能换新张量；
3. 不能有依赖数据的动态控制流；
4. 不能有动态形状；
5. 同一进程内同时只能有一次捕获在进行。

另有两条工程上的：捕获必须在非默认流上；预热也要走侧流，否则预热期分配的显存块不在捕获要用的池里，回放时地址对不上。`torch.cuda.graph` 会替你处理捕获流，但预热那一段得自己写。

Q3｜引擎要捕获几十张图，为什么显存没有溢出？

考察点：对捕获尺寸阶梯和内存池共享的理解。

参考答法（先讲这两句主干）：

1. 图的数量有上限，不是每个 batch size 一张，引擎按一个阶梯只捕几十个尺寸，落在两档之间的 batch 向上取整；
2. 这些图共享同一个内存池（`graph_pool_handle`），中间激活值的显存是复用的，几十张图并不等于几十份激活。

被追问细节时再展开：vLLM 的尺寸表是 `[1,2,4] + range(8,256,8) + range(256, max+1, 16)`，最大值默认取 `min(max_num_seqs*2, 512)`，`max_num_seqs=128` 时算下来 35 个尺寸；`FULL_AND_PIECEWISE` 模式下每个尺寸捕两轮。向上取整的无效计算在 batch 均匀分布下大约 5%。所以真实开销远小于张数乘以单张激活，本章实测只有 0.01 GiB（`Graph capturing finished in 3 secs, took 0.01 GiB`，51 个捕获尺寸、`FULL_AND_PIECEWISE` 模式两轮捕获）；具体数值随捕获尺寸数量、模型规模与捕获模式变化，要在自己的启动日志里读。引擎在内存 profiling 阶段就把它估算并预留掉了，不会事后挤占 KV Cache。

Q4｜你怎么向业务方解释是否启用 CUDA Graph 这个决定？

考察点：能不能把工程数据翻译成一个有条件的取舍，而不是因为更快就启用。

参考答法：这是一次权衡，不是无代价的收益。买到的是 decode 每步的延迟下降，付出的是启动时的图捕获耗时、一块预留显存、以及 batch 向上取整带来的少量无效计算。本章实测是 3 秒与 0.01 GiB，两者都随捕获尺寸数量和模型规模上升，要在自己的配置上读启动日志。什么时候值：以小 batch 为主的在线服务、对 ITL 敏感的流式对话产品、长期常驻很少重启的服务。什么时候不值：显存不足、KV 池是瓶颈时，那块图显存换成 KV 可能更划算；要求秒级弹性伸缩时，捕获耗时直接体现在扩容速度上；batch 一直很大的离线批处理，launch 开销占比本来就低。所以答法是给一张收益代价表加一句在当前流量画像下如何取舍，而不是一个是或否。

Q5｜（陷阱题）我们关掉 CUDA Graph 之后吞吐反而涨了 8%，是不是说明 CUDA Graph 在我们这儿是负优化？

考察点：会不会漏掉 A/B 里那个同时变化的第二个变量。题面预设了“禁用 CUDA Graph 只改了一件事”，而这个前提不成立。

参考答法：先做量级核对，再下结论。看启动日志里 `took X GiB` 与 `Available KV cache memory` 两个数，再估算负载的 KV 工作集：图显存占 KV 池不到 1%、工作集远小于池容量时，第二变量可以排除；否则这个 A/B 大概率没控住变量。禁用 CUDA Graph 之后同时发生了两件事：

1. 每步 decode 少了图回放、多了逐个 kernel launch；
2. **原本预留给图的那块显存回到了 KV Cache 池**：引擎在内存 profiling 阶段就会估算图的显存并据此算 KV 池大小，禁用之后 KV 池随之变大，能同时在途的请求数上升，吞吐自然可能涨。

这 8% 里有多少来自第二件事，光看吞吐分不出来。要彻底隔离，得把两组的 KV 池强制对齐（用 `--num-gpu-blocks-override` 写死块数）再比一次。**我没有做这一步复测，我做的是量级核对**：启动日志里图显存 0.01 GiB、KV 池 50.64 GiB，图只占千分之零点二；负载并发 16、单请求最长 320 token，KV 工作集占池容量约 0.5%。两组的 KV 池都远没有用满，所以这条路径在本次测量里贡献可以忽略。同时应该看的指标不是聚合吞吐而是**低并发下的 TPOT 和 ITL**，那才是 launch 开销直接作用的地方。如果对齐 KV 池之后禁用 CUDA Graph 仍然更快，那才轮到怀疑图本身，届时要查的是 batch 是否长期落在捕获尺寸之间导致 padding 浪费过大，或者是否根本没有走到 decode 整图那条路径。

把这三个项目写进简历之前

这一节不讲技术，讲这份文档能支撑到哪里。

一、没在自己机器上跑过，不要写本章的数字

本章的每一个数都出自 H100 上的一次实测，各自带着自己的负载口径：项目二是输入 1024、输出 128、并发 32，项目三是输入 64、输出 256、并发 16。**换一张卡、换一个负载，这些数全都会变。**

面试官问的第一个问题通常是"你的卡是什么型号、你跑出来多少"。把本章的 1.91× 或 85.8% 直接报出去，接下来两三个追问就会到不了底：你的 KV 池多大、你的采集窗口跑了几条请求、你那个 kernel 的 Achieved Occupancy 和 Theoretical Occupancy 差多少。这些数只有自己跑过才有。

二、跑完之后，手里会多出这些东西

这些是文档里没有、只能自己跑出来的，也是面试里真正能撑住深挖的部分：

1. **分段的算子表**：用 `trace_top.py --annotation` 切出的 prefill 段与 decode 段各自的前三名与占比。本章只保留了整段表。
2. **你自己的 GPU 占空比与 launch 密度**，以及它们随 batch 怎么变。
3. **完整的 `ncu_baseline.csv`**，不只是 `ncu_top.py` 摘出来的那几行。里面还有 Grid Size 与 SM 数的关系、每线程寄存器数、warp 停顿原因的分布，这几项本章都没有引用，但面试官会问。
4. **开图与关图两组的 `Available KV cache memory`**，这是量化图显存成本唯一可靠的办法。
5. **你自己那份 `capture_scan.txt`**，以及四档 batch 上加速比的实际走向。

三、简历该怎么写

可以写的是方法与结论的结构，例如：在 H100 上对 vLLM 与 SGLang 做端到端 profiling，用 nsys 定位耗时最高的 kernel、用 ncu 判定瓶颈类型为显存带宽受限，据此调整单步 token 预算并做同口径复测，按预先登记的判据否决了这次改动。

一个被否决的改动同样是完整的闭环，而且比"我调了个参数快了百分之几"更经得起追问，因为它需要你说清判据是什么、什么时候定的、为什么不满足。

不建议写的是脱离口径的加速比。如果要写数字，把负载口径一起写上，并且确保那是你自己机器上的数。

四、本章明确没做的事

这几处本章自己也没有做，如果被问到，照实说比编强：

- 项目一的墙钟差异来自主机侧漂移，本章没有定位到具体原因，也没有交替跑多轮取区间。
- 项目二的调度参数对照只跑了单轮，vLLM 侧 −0.7% 落在噪声范围内，只能说没有观测到收益。
- 项目三没有跑只关闭 CUDA Graph、保留 torch.compile 的第三组，所以 1.91× 是两项合计。
- 项目二的 ncu 数据来自单批离线复现，不是常驻服务态，两者的算子路径并不完全相同。

本文档为课程工程文档的完整样例。文中全部实测数据来自页首所述环境的真实运行，产物文件按 0.2 节的目录布局存放。