机器学习系统 2026春

第九章 机器学习图优化

张燕咏 讲席教授 yanyongz@ustc.edu.cn
张午阳 特任教授 wuyangz@ustc.edu.cn

中国科学技术大学
University of Science and Technology of China

第一节课

推理图优化: 基本概念

计算图是什么, 四种浪费, 四个核心方法, LLM 新挑战

第一部分

计算图是什么

从 PyTorch 代码到 GPU 执行: 计算图在中间扮演什么角色

本节课内容 (进度)

1.1 一个神经网络究竟在算什么

任何神经网络都可以拆成一连串"算子" (operator): 把它们顺序连起来, 就是计算图。

Computation graph: ops are nodes, tensors are edges

Edge label = tensor shape & dtype. The graph is the OBJECT we optimize.

计算图的基本要素

节点 (node) = 算子: matmul, add, ReLU, softmax 这类基本运算
边 (edge) = 张量: 每条边上流动的数据, 带 shape 和 dtype
有向无环: 数据从输入向输出单向流动, 不会绕回去

为什么用图来表示

看得到全貌, 而不是只见单一算子
便于做"整体性"的优化 (后面几部分要展开的核心)
便于在不同硬件之间迁移 (图的语义与硬件无关)

计算图是"把一个模型完整画出来"的标准表示; 后面所有优化都作用在这个图上。

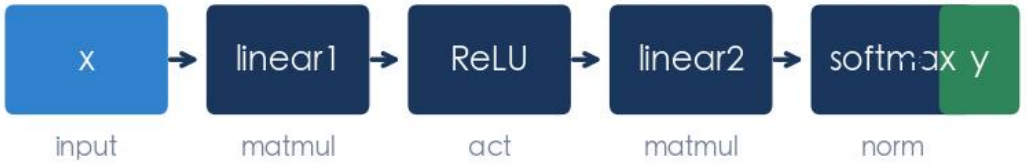
1.2 计算图长什么样: 从代码到图

同一段 PyTorch 代码, 框架会自动把它建成一张图; 我们看到的就是这张图。

PyTorch 代码

```
# A two-layer MLP
h1 = linear1(x)        # matmul
h2 = relu(h1)          # activation
h3 = linear2(h2)       # matmul
y  = softmax(h3)       # normalize
```

对应的计算图



节点 = 算子 (op)　边 = 张量 (tensor, 含 shape/dtype)

一一对应的关系

代码里写 `linear1(x)`, 图里就有一个 `matmul` 节点, 输入边是 `x`, 输出边是 `h1`
节点带 shape 信息 (`x: [B, D_in]`, `h1: [B, D_hidden]`), 优化时按 shape 决策
PyTorch 默认是 "边写边跑" (eager), 不主动建图; torch.compile 才把它捕获成完整图

计算图既能反过来看代码, 也能往下交给编译器: 它是"代码"和"机器执行"之间的中间表示。

1.3 什么时候做图优化: 推理管线中的位置

图优化不是一种独立工具, 而是从代码到 kernel 之间一段必经的处理。

图优化在推理管线中的位置

图优化的输入和输出

输入: 框架捕获到的原始计算图 (节点很多, 没合并)
输出: 节点更少、调度更优、内存更省的等价图
保证: 数学语义不变 (容差范围内), 只是"执行方式"更优

三种典型时机

离线编译 (TensorRT-LLM): 部署前一次编译, 运行时不再变
JIT 编译 (torch.compile): 第一次跑到时编译, 之后缓存复用
运行时 (vLLM): 部分模型片段动态选择不同优化路径

图优化是"在你的代码和 GPU 之间, 自动地把图改得更省力"; 本节后续讲的就是这一步具体怎么做。

1.4 三种图形态: 静态 / 动态 / 折中

"何时建图" 决定了能做多少优化: 越早建越好优化, 但越难写。

静态图

何时建图: 跑之前先建好整张图
优点: 全图可见, 可深度优化; 部署期无 Python
缺点: 控制流 (if/while) 难写; 调试不直观
代表: TensorFlow 1.x, ONNX, TensorRT-LLM
适用: 工业部署, 模型结构稳定

动态图 (eager)

何时建图: 不主动建图, 逐 op 执行
优点: 写起来像普通 Python; 调试方便; 研究友好
缺点: 框架看不到全图, 难做整体优化
代表: PyTorch (默认), JAX 部分模式
适用: 研究迭代, 快速原型

折中 (torch.compile)

何时建图: 第一次跑到时, 在运行时建图
优点: 不改用户代码; 既能像 eager 写, 又能像静态图优化
缺点: 第一次有编译开销; 控制流要分段
代表: PyTorch 2.x torch.compile
适用: 大多数 PyTorch 项目的默认选择

本节其余内容默认在 "折中" 这条路线上展开 (torch.compile 与同类工具)。

Ansel et al., PyTorch 2, ASPLOS 2024 (DOI:10.1145/3620665.3640366)

第二部分

不优化, GPU 在浪费什么

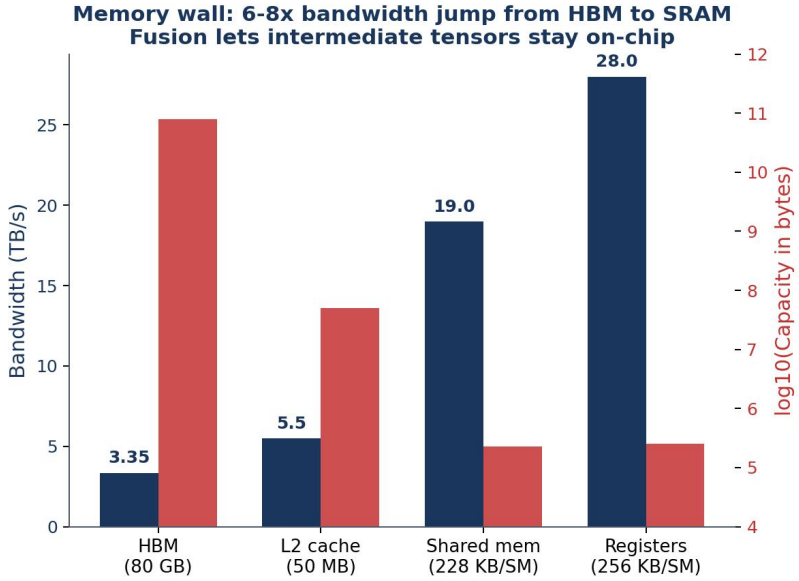
四种典型浪费: 反复读 HBM, 启动开销, 串行执行, 多余计算

本节课内容 (进度)

√ 第一部分	▶ 第二部分	第三部分	第四部分	第五部分	第六部分
计算图是什么	不优化, 会浪费什么	四个核心优化方法	LLM 推理图的新挑战	LLM 中的代表性融合	主流工具栈速览

2.1 浪费一: 中间张量反复进出 HBM

GPU 内, HBM 是大但慢的内存, SRAM 是小但极快; 算子之间反复经 HBM 中转, 等于浪费速度差。



一个具体场景

Q·K^T 算完, 中间矩阵写回 HBM, 占内存且占带宽
下一步 softmax 又从 HBM 读出来, 再写回去
最后 matmul V 再读一次: 一份中间值往返 3 次 HBM
实际 HBM 带宽 ≈ 3 TB/s, SRAM ≈ 19 TB/s, 差 6 倍

本质

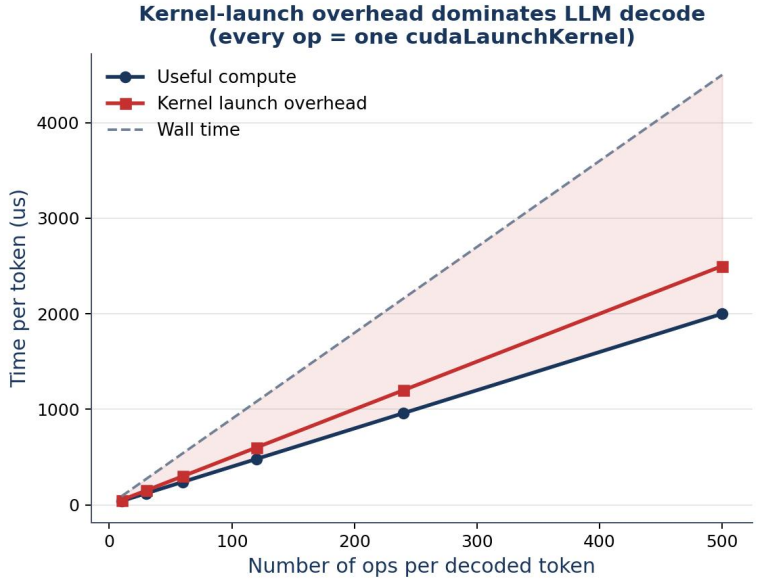
中间张量没必要落 HBM, 留在芯片上即可
把多步运算合在一个 kernel 里, 中间值"借住"SRAM
这就是后面 3.1 "算子融合"要解决的浪费
FlashAttention 是把这个思想做到极致的例子

HBM 是 GPU 的最大瓶颈; 减少中间张量进出 HBM, 直接换来线性加速。

NVIDIA H100 datasheet; Dao et al., FlashAttention, NeurIPS 2022 (arxiv:2205.14135)

2.2 浪费二: 每个算子要发起一次 kernel launch

每发起一个 CUDA kernel 都有约 5 微秒的固定开销; 算子越细碎, 累积越严重。



一个具体数字

LLaMA-7B 一层 ≈ 30 算子, 32 层 = 960 次 launch
每次 ~5 μs, 一次 forward 仅 launch 就花 ~4.8 ms
Decode 单 token 计算才 3-4 ms, **launch 比算还多**
Prefill 算子大, launch 占比小; decode 单 token 算子小, 占比放大

本质

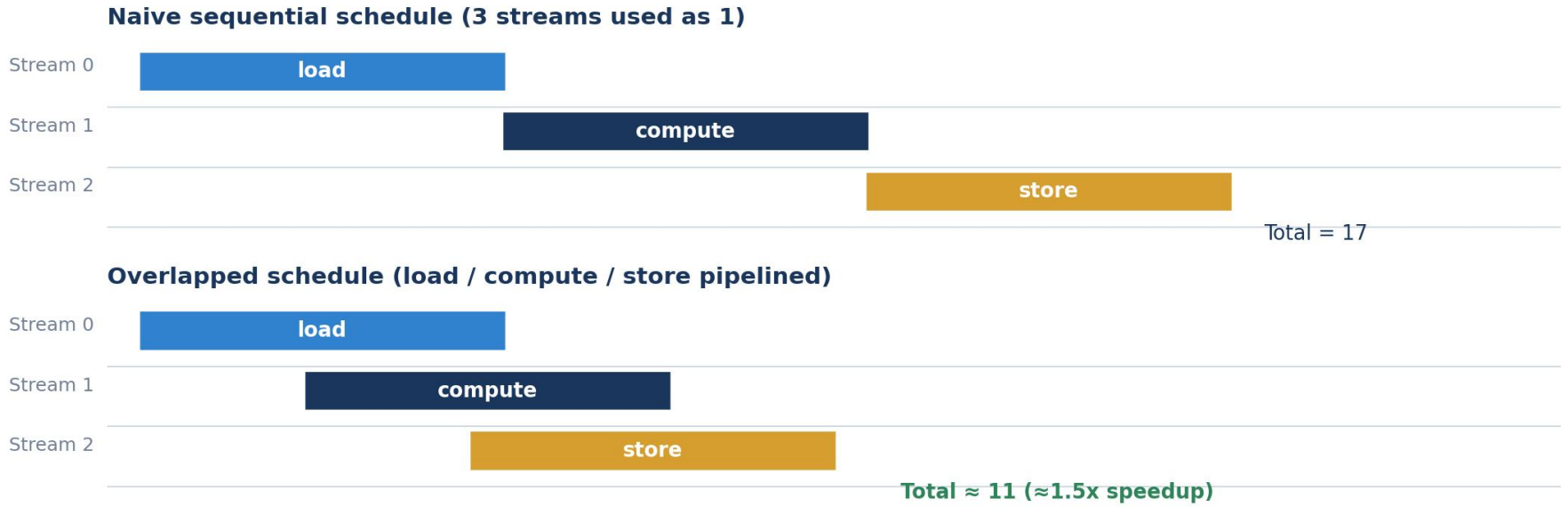
算子合并 = 一次 launch 干完更多事, 直接减少次数
CUDA Graph = 把固定序列"录像"成一次性回放
两条对策都在后面的方法里出现

decode 阶段 launch 开销可以占到 50%+ 的总延迟, 是 "算子合并" 的另一动力。

NVIDIA CUDA Graphs blog (Alan Gray, 2019); vLLM blog (2025-08-20)

2.3 浪费三: 算子排成一队, 没用上 GPU 的并行

GPU 内部可同时跑多条 stream, 也能让计算和数据搬运重叠; 串行排队就浪费了这些可能。



哪些可以同时做

不互相依赖的两个算子, 可以丢到不同 stream
数据从 CPU 搬到 GPU, 同时 GPU 在算别的
多卡推理时, all-reduce 通信和下一层计算可重叠

本质

不是"算得更快", 而是"同一时刻让更多硬件忙起来"
图视角才看得到"哪些算子互不依赖"
对应后面 3.2 "调度并行"方法

调度的关键不是更快地算一步, 而是同一时刻让更多 SM / 设备 同时干活。

NVIDIA CUDA Programming Guide, Streams & Concurrency

2.4 浪费四: 同样的中间值算了两次, 或留得太久

图里常有冗余: 同一个子表达式被计算两次; 训练用的辅助节点在推理时没用; 中间张量留到不再需要还占着内存。

① 重复算同一个值

图里同一个 `norm(x)` 出现在两个分支
常量 `2π/d_h` 编译期就能算出, 没必要每步算
对策: 算一次复用 (CSE)
对策: 编译期算成常量 (const fold)

② 训练痕迹遗留

推理图里仍有 dropout (推理时其实是 identity)
训练用的 backward 算子被一同保留
`x * 1` / `x + 0` 这种恒等运算
对策: 死代码删除 (DCE)

③ 中间张量占用过久

算完没用了, 还占用显存
多个临时张量, 可以共用同一块 buffer
对策: 编译期做 lifetime 分析, buffer 复用

这一类浪费由编译器经典 pass 处理, 几乎零运行时代价, 但所有框架都会做。

ONNX Runtime Graph Optimizations (官方文档); TensorRT Best Practices

第三部分

四个核心优化方法

算子融合 / 调度并行 / 图化简 / 内存规划

本节课内容 (进度)

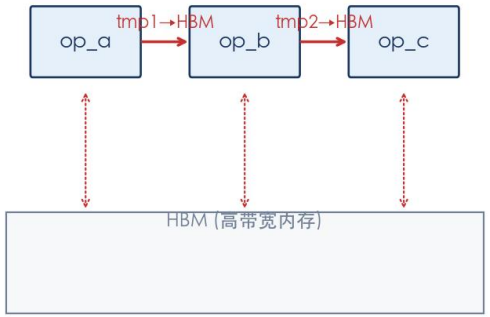
√ 第一部分	√ 第二部分	▶ 第三部分	第四部分	第五部分	第六部分
计算图是什么	不优化, 会浪费什么	四个核心优化方法	LLM 推理图的新挑战	LLM 中的代表性融合	主流工具栈速览

3.1 方法一: 算子融合: 把相邻算子合并为一个 kernel

针对浪费 ① + ②: 减少 HBM 读写, 同时减少 kernel launch 次数; 收益最直接。

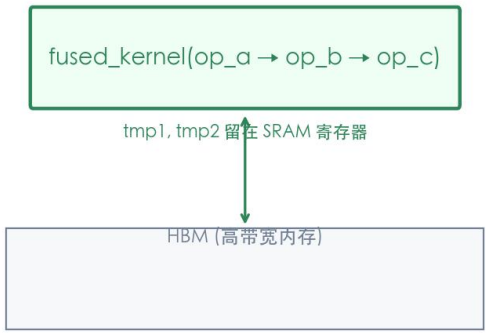
垂直融合: 串行算子合一, 中间张量不落 HBM

融合前: 三次 HBM 来回



HBM 读写次数: 3 次 (启动 3 次 kernel)

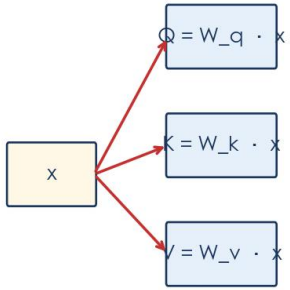
垂直融合后: 锁在 SRAM



HBM 读写次数: 1 次 (单次 kernel)

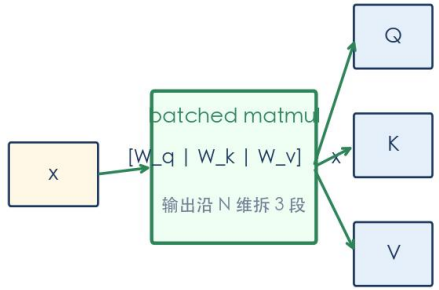
水平融合: 同型并行算子合一, 输入只读一次

融合前: 三次同型 matmul, 输入 x 重复读



x 读 3 次

水平融合后: 一次 batched matmul



x 读 1 次, SM 利用率 ↑

垂直融合: 串行算子合一

依赖链上的算子 (LayerNorm → matmul → activation) 合成一个
中间张量留在 SRAM, 不再落 HBM
典型: Norm + 矩阵乘 + 激活, FlashAttention 内部

水平融合: 同型并行算子合一

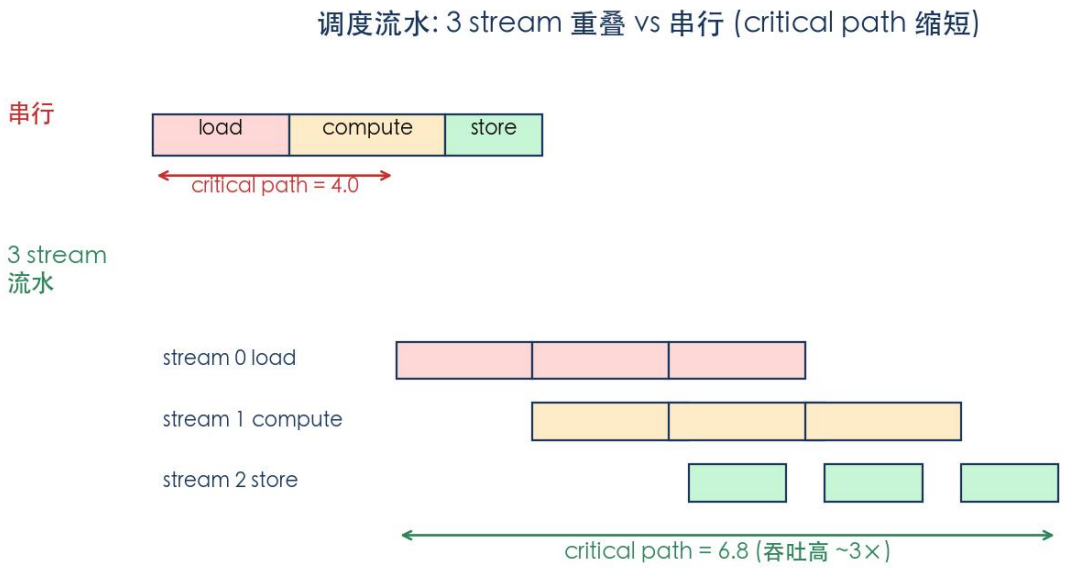
输入相同的并行 matmul 合成一个大 matmul
输入只读一次, 输出按段拆开, SM 利用率更高
典型: QKV 三投影 → batched matmul

融合是 LLM 图优化中最主要的手段; 几乎每个推理框架都把它作为核心 pass。

Dao et al., FlashAttention, NeurIPS 2022 (arxiv:2205.14135); FasterTransformer 设计文档

3.2 方法二: 调度并行: 让 GPU 同一时刻多干一些

针对浪费 ③: 不改变要算什么, 只改变什么时候算, 让 GPU 不再排队空闲。



单卡内能做什么

多个 stream 同时跑互不依赖的算子
CUDA Graph 把固定一段"录像"成一次性回放, 省 launch
数据搬运和计算重叠 (HBM 读 + SM 算 同时进行)

多卡 / 跨设备

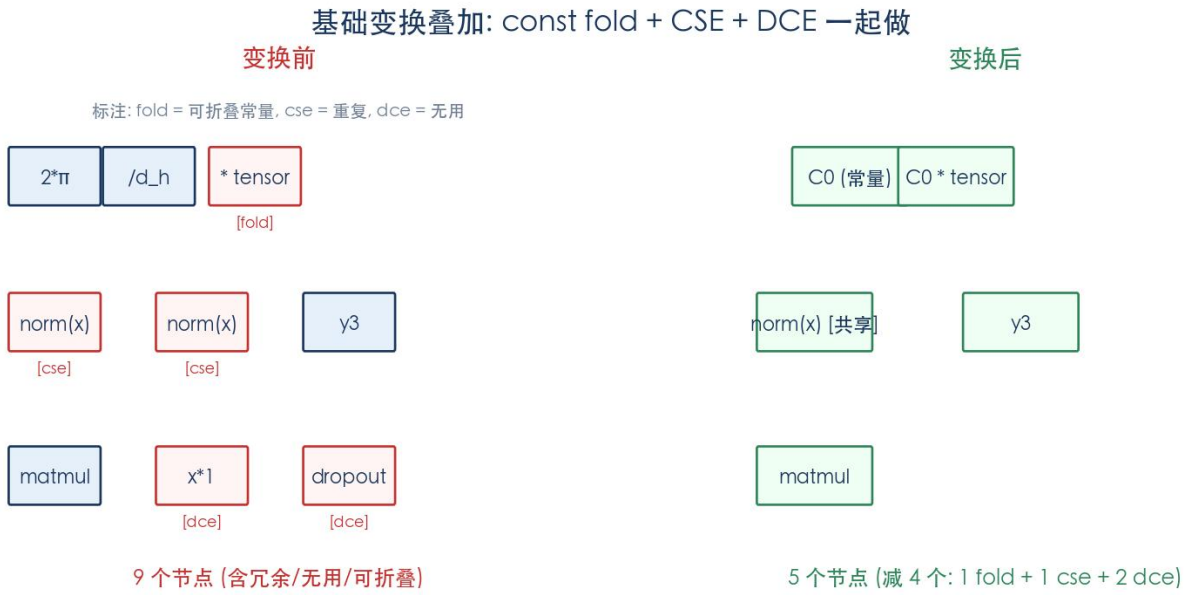
TP 推理: 通信 (all-reduce) 与下一层计算重叠
CPU ↔ GPU: tokenize / sampling 异步, 不阻塞 GPU
vLLM, TRT-LLM 都把这些作为默认行为

图视角才能看到"哪些算子可以并行"; 调度是可落地的关键。

NVIDIA CUDA Programming Guide, Streams & Concurrency

3.3 方法三: 图化简: 删冗余, 把图改得更小更稳

针对浪费 ④: 几乎零代价的清理, 所有框架都默认做; 是后续融合的前置准备。



四种典型化简

常量折叠: 编译期能算的就别留到运行时 (例: rotary 的 cos/sin 表)
公共子表达式: 同一个 `norm(x)` 出现两次, 只算一次
死代码删除: 删 dropout、 删 `x*1`, 删训练残留
代数等价改写: softmax 减最大值 (防 FP16 溢出, 数学完全等价)

为什么是"前置准备"

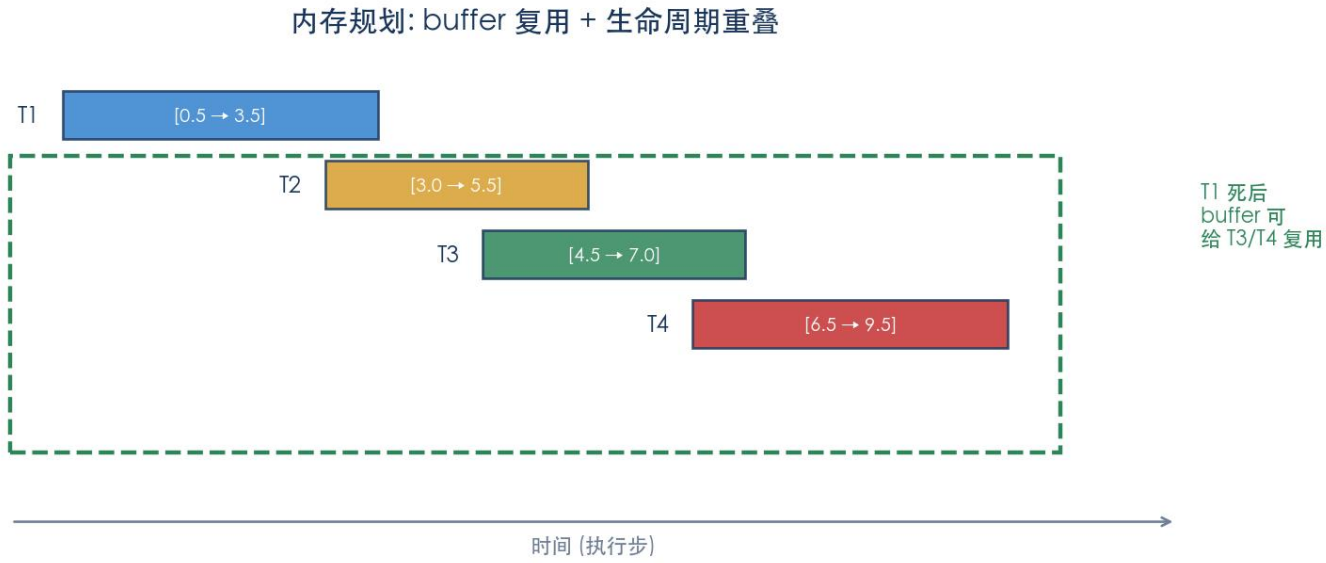
化简后图变小, 后续融合更容易识别相邻可合并的节点
数值稳定改写也避免融合后中间值溢出
ONNX Runtime / torch.compile / TensorRT 默认都做这步

图化简虽然简单, 但是后面所有重型优化能跑得顺的前提条件。

ONNX Runtime Graph Optimizations (官方文档); TensorRT Best Practices

3.4 方法四: 内存规划: 让大模型挤进有限显存

不抢算力, 抢显存: 让能放进 GPU 的模型更大, 让长序列推理仍然能跑。



两种主要做法

Buffer 复用: 同一块显存先后承载不同的中间张量
(编译期做 lifetime 分析, 谁不再用就让出来)
重计算 vs 存储: 短计算 (norm, rotary) 不存了, 用时重算
在显存紧张时换来空间, 代价是多一点计算

LLM 场景里特别有用

long context attention 的中间张量极大
KV cache 需要按需分配显存 (PagedAttention 的动机)
训练里的 gradient checkpointing 也是这个思想

内存规划决定"装不装得下"; 在 long context / 大模型场景里, 比加速更要紧。

Kwon et al., PagedAttention/vLLM, SOSP 2023 (arxiv:2309.06180); TensorRT Developer Guide

第四部分

LLM 推理图为什么特别需要图优化

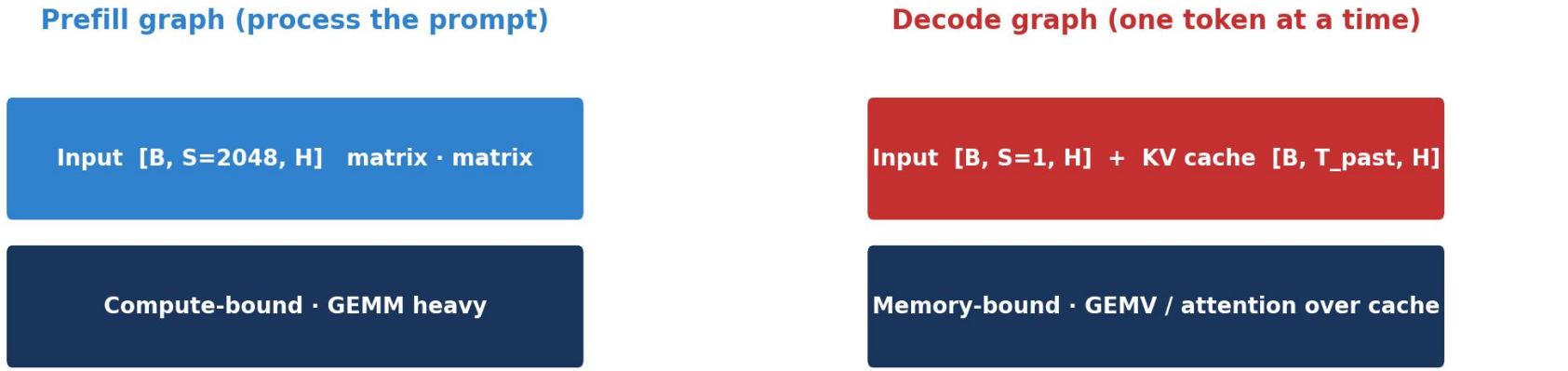
Prefill/decode 双形态, KV state, 动态 shape: 用 第二部分 的 4 机会重新审视 LLM

本节课内容 (进度)

√ 第一部分	√ 第二部分	√ 第三部分	▶ 第四部分	第五部分	第六部分
计算图是什么	不优化, 会浪费什么	四个核心优化方法	LLM 推理图的新挑战	LLM 中的代表性融合	主流工具栈速览

4.1 LLM 推理的双形态: prefill 与 decode

LLM 与传统 CV 模型的根本差异: 一次推理对应两张本质不同的计算图。



Prefill 阶段

- 输入: 整段 prompt (S = 几百 ~ 几千 token)
- 算子形状: (B, S, H), S 大
- 算术强度高, **compute-bound**
- 主要瓶颈是 SM 算力 + attention 复杂度

Decode 阶段

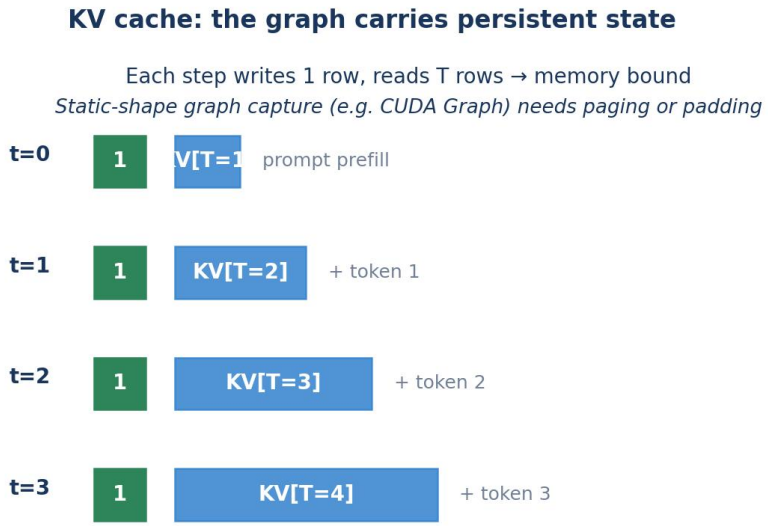
- 输入: 单 token (S=1)
- 算子形状: (B, 1, H)
- 算术强度极低, **memory-bound**
- 主要瓶颈是 HBM 带宽 + KV cache 读取

Prefill 和 decode 的优化目标完全不同: 前者吃算力, 后者吃带宽。

Pope et al., Efficiently Scaling Transformer Inference, MLSys 2023 (arxiv:2211.05102)

4.2 KV cache: 让计算图带 "记忆"

经典推理算完即弃; LLM decode 每一步要读、要写一个会变长的 cache。



对照看: 传统 vs LLM

传统 CNN: `y = model(x)`, 每次独立, 没有跨调用的存储
LLM decode: 每生成 1 token, 把新算出的 K, V **追加**到 cache
下一步生成时, 又要从 cache **读出**全部历史 K, V
cache 是一份**跨多次 forward 持续存在、不断变长**的张量

对图优化带来的三个具体问题

经典编译器假设算子是纯函数 (输入决定输出, 不改变外部状态)
本次 forward 没人读 cache 写入, 死代码删除 (DCE) pass 可能误判为冗余, 把写删掉, 下一步 decode 会拿不到正确的历史 K, V
服务化时每个用户一份独立 cache; 调度器要算每份 cache 还剩多少显存, 才能决定能否接下一个请求

计算图一旦带状态, 经典优化 pass 要全部重新审视; 第三节会看到 PagedAttention 怎么管这份 cache。

Kwon et al., PagedAttention/vLLM, SOSP 2023 (arxiv:2309.06180)

4.3 输入 shape 一直在变: 静态编译的难题

编译器擅长把固定 shape 优化到极致; 但 LLM 推理几乎每一步 shape 都不一样。

shape 为什么会变 (3 个来源)

prompt 长度: 一句 "你好" 32 tokens, 一段摘要 4096 tokens
decode 进展: 第 1 步 KV 长度 = 0, 第 200 步 KV 长度 = 200
同 batch 不同请求长度都不同 (continuous batching)

为什么 "shape 在变" 是个麻烦

静态编译会把 "S=128" 当成常量写进机器代码
shape 一变就要重新编译, 一次重编 ≈ 数秒到数分钟
若退回 eager, 又拿不到融合 / 调度的优化

三种工程上的折中

桶化 + padding: 只编译 S=128/512/2048 三档, 短输入补 0 进桶, 实现简单但有浪费
dynamic 编译 (`torch.compile(model, dynamic=True)`): 编译一份支持变长的 kernel, 通用但 kernel 慢一些
piecewise 编译 (vLLM 做法): attention 那段留 eager 处理变长, 其它部分静态化 — 当前主流折中

动态 shape 是 LLM 时代图优化最大的新挑战; 后两节会看到具体的折中方案。

torch.compile dynamic docs; vLLM blog (2025-08-20)

第五部分

算子融合: LLM 图优化的主要方法

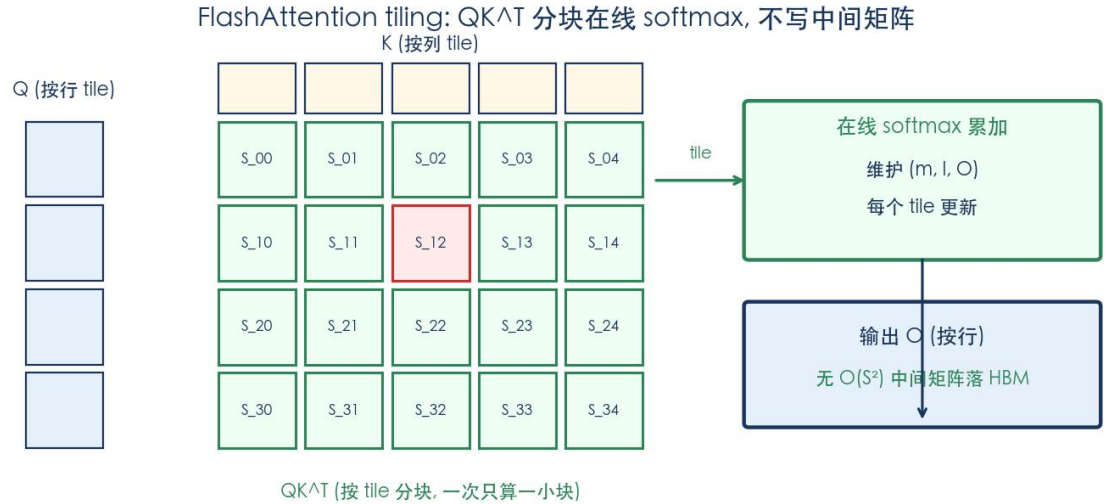
FlashAttention 1/2/3, fused MLP, RMSNorm 融合, KV append 融合: 收益, 代价, 边界

本节课内容 (进度)

√ 第一部分	√ 第二部分	√ 第三部分	√ 第四部分	▶ 第五部分	第六部分
计算图是什么	不优化, 会浪费什么	四个核心优化方法	LLM 推理图的新挑战	LLM 中的代表性融合	主流工具栈速览

5.1 FlashAttention: 把整个 attention 写成一个 kernel

传统做法: matmul → softmax → matmul 三个独立算子, 中间 S×S 矩阵反复进出 HBM。**FlashAttention**: 三步合一, 中间矩阵 "从未真正生成过"。



HBM 流量: O(S²) → O(S); 原本 N×N 的 score 矩阵从未存在过

两个关键技巧

tiling: 把 S×S 切成小块, 一块块算, 算完即走, 不写 HBM
在线 softmax: 一块块累加归一化项, 数学上完全等价原 softmax

效果

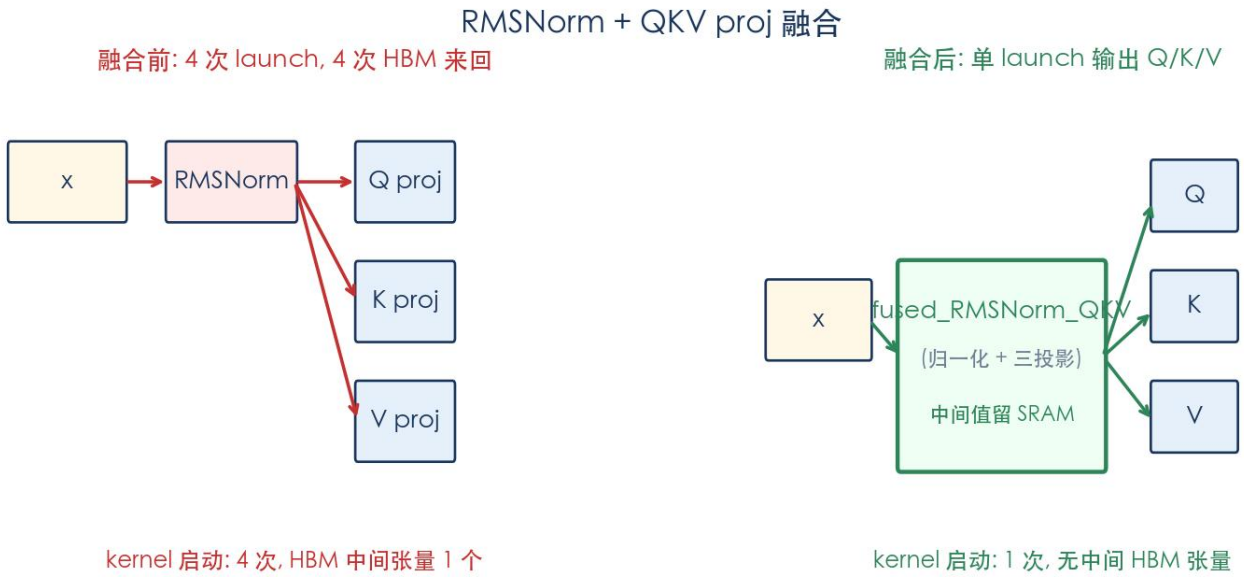
HBM 读写量 O(S²) → O(S), 序列越长越受益
几乎所有 LLM 推理框架的默认实现 (PyTorch SDPA / vLLM / TRT)

FlashAttention 是把 "算子融合" 和 "省 HBM 流量" 思想做到极致的代表; 第二节会看到 Inductor 自动 emit 类似 kernel。

Dao et al., FlashAttention, NeurIPS 2022 (arxiv:2205.14135); FA-2 (arxiv:2307.08691); FA-3 (arxiv:2407.08608)

5.2 RMSNorm + QKV proj 融合: 4 op → 1 kernel

思路: 归一化的中间值不写 HBM, 直接进行三投影, 单 launch 输出 Q/K/V。



收益

- launch 次数 4 → 1
- 归一化中间张量不落 HBM
- 同时对应空间 ① 与 ②

实现要点

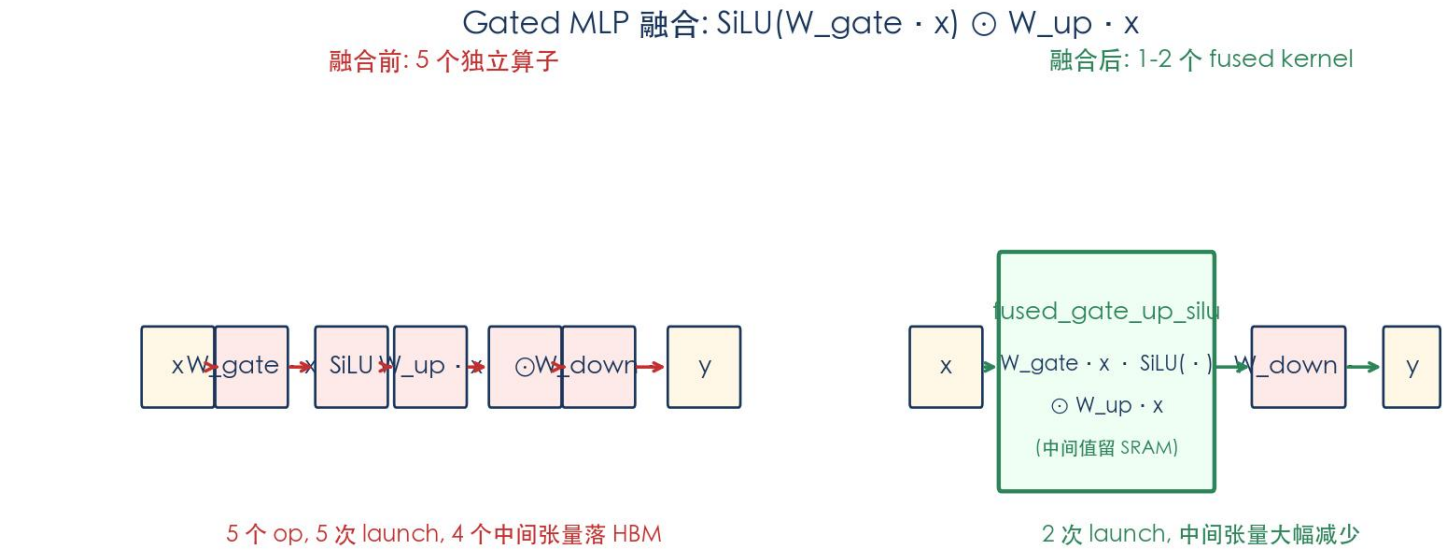
- 三个权重在 SM 内并发使用, 输入只读一次
- 适合 LLaMA / GPT 系列每层入口
- vLLM, FasterTransformer 默认实现

RMSNorm + QKV 是 LLM 每一层入口的标准融合, 收益叠加在所有 token 上。

FasterTransformer 设计文档; vLLM 源码 fused_rms_norm

5.3 Gated MLP 融合: 5 op → 1-2 kernel

思路: SiLU(W_gate·x) ⊙ W_up·x → W_down 的中间张量都锁在 SRAM。



收益

- launch 5 → 1-2
- 4 个中间张量从 HBM 中消除
- HBM 流量节省约 50%

实现要点

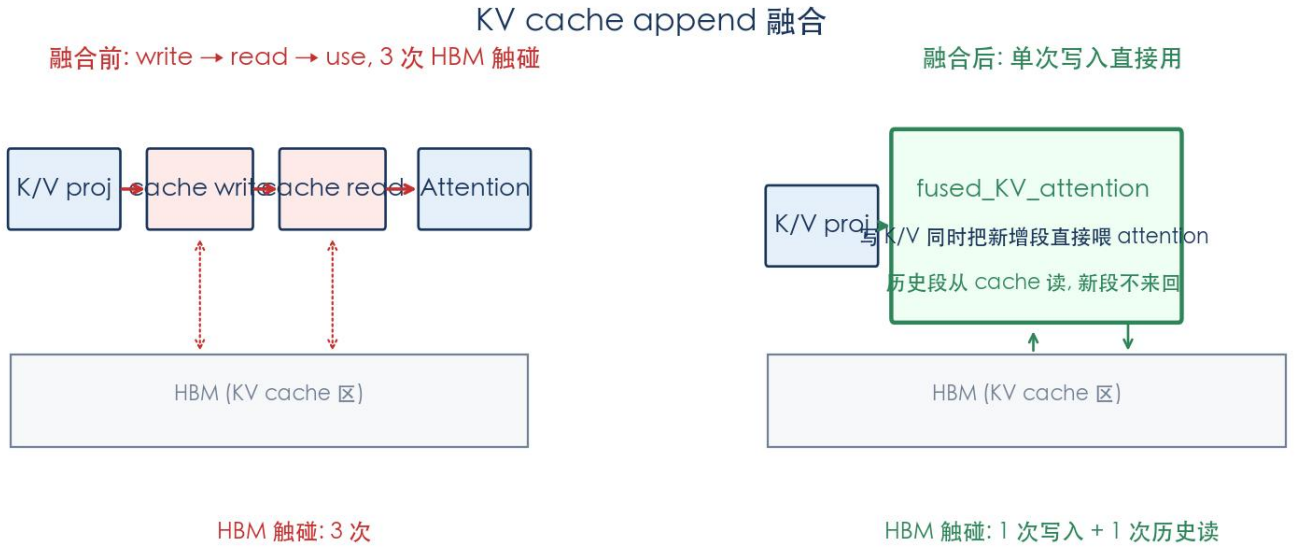
- gate 与 up 投影做水平融合
- SiLU 与逐元素乘做垂直融合
- LLaMA / Mistral / Qwen 默认结构

Gated MLP 在 LLaMA 系模型中占解码时间的 30-40%, 融合收益直接显现。

Touvron et al., LLaMA; vLLM 源码 fused_silu_and_mul

5.4 KV cache append 融合: 写入直接给 attention

思路: K/V proj → cache write → cache read → attention 合并; 新增段不来回 HBM。



收益

- HBM 触碰 3 → 1+1 (写新段, 读历史)
- decode 步关键路径缩短
- 与 paged KV 配合更佳

实现要点

- 新段 K/V 直接传给 attention 当前块
- 历史段从 cache 按 block 读
- vLLM, SGLang 都做这个融合

KV append 融合是把 "图状态写入" 与 "消费" 合一, 直击 decode 关键路径。

Kwon et al., PagedAttention/vLLM, SOSP 2023 (arxiv:2309.06180)

5.5 融合的代价: 不是免费午餐

收益越大的融合, 越要为它付出四种成本: 开发, 数值, 灵活性, 维护。

① 开发成本

手写 fused kernel = 写一份新的 GPU 程序, 工作量大
FlashAttention 从 v1 到 v3 业界打磨了约 2 年
Triton 降低门槛, 但每代新硬件 (H100, B200) 还要重写

② 数值精度的代价 (举例)

未融合: 每个 op 之间结果以 FP16 存回 HBM, 自动"四舍五入"一次
融合后: 中间结果留在寄存器里, 全程 FP32 累加, 不再 round
看起来更精确, 实际与未融合版有差异 (通常 1e-3 量级)
在 FP8 (FA3) 下还要加 scaling 校准, 否则会数值溢出

③ 灵活性下降

原本 4 个独立 op 可以单独换 (例如换 dropout 概率)
融合后变成一个不可分的 kernel: 加 alibi/sliding window 都要重写

④ 维护负担

框架升级要保证 fused kernel 仍兼容
调 bug 时要先判: 是上层算法错, 还是 kernel 实现错

融合越深, 收益与代价都越大; 工程上需要按场景判断哪些 pattern 值得做。

Dao et al., FlashAttention; vLLM Maintainers, GitHub issues

5.6 哪些算子不能 (或不该) 融合

三类硬边界: 形状不固定, 有 if/while 控制流, 跨多张 GPU 的通信 — 都让融合做不下去。

① 形状不固定

典型场景: prompt 长度可变, 编译期不知道 S=128 还是 4096

tile 大小、block 数都依赖 shape, 没法编一份代码通用

应对: 桶化预编几份, 或允许 attention 段留 eager

② 控制流 (if / while)

典型场景: speculative decoding 里 `if accept then X else Y`

分支让图不再是单一 DAG, 不同分支需要不同 kernel

应对: 在分支处切图 (graph break), 各段分别融合

③ 跨 GPU / 跨设备

典型场景: all-reduce 把 8 张 GPU 的中间值同步, 中间是 NCCL 调用

融合 kernel 只能在单卡内, 跨卡的通信跳不过去

应对: 通信和计算 overlap, 而不是融合成一个 op

融合不是万能; 这三条边界是后两节里反复出现的 "工程妥协点"。

Ansel et al., PyTorch 2, ASPLOS 2024; vLLM blog (2025-08-20)

第六部分

主流工具栈速览

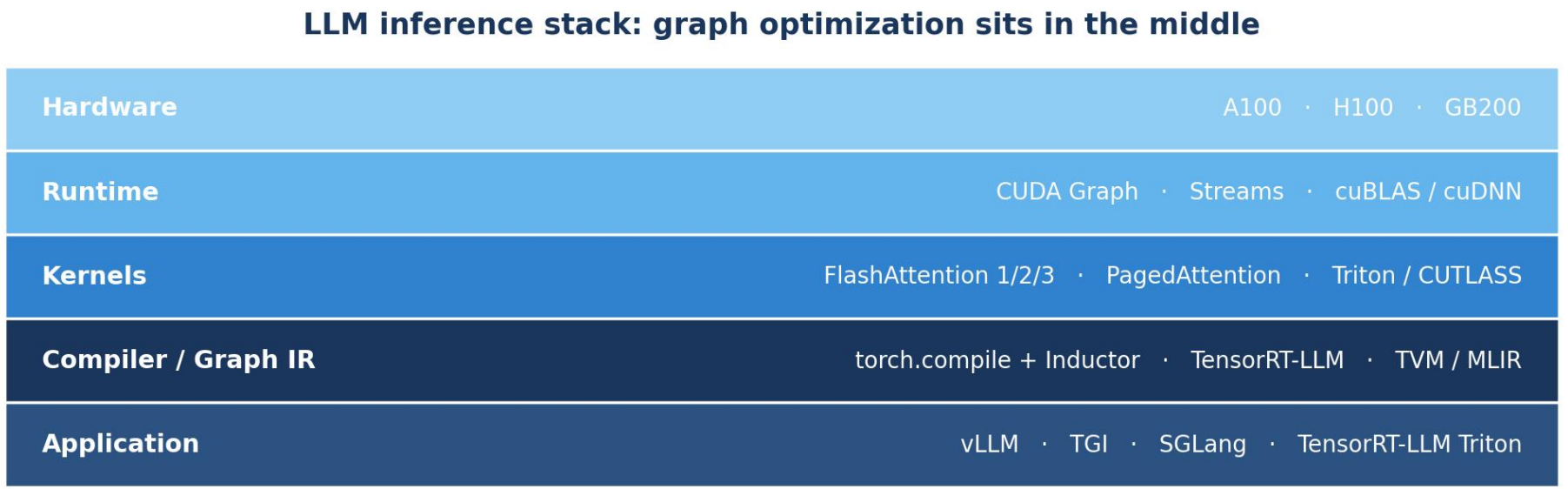
torch.compile · TensorRT-LLM · vLLM · SGLang: 不同工具如何把 6 类手段串起来

本节课内容 (进度)

√ 第一部分	√ 第二部分	√ 第三部分	√ 第四部分	√ 第五部分	▶ 第六部分
计算图是什么	不优化, 会浪费什么	四个核心优化方法	LLM 推理图的新挑战	LLM 中的代表性融合	主流工具栈速览

6.1 推理图优化工具栈: 一座金字塔

结构: 越往下越接近硬件; 越往上越接近用户; 每层都做一部分图优化。



自底向上

- 硬件: NVIDIA / AMD / Ascend
- Kernel: CUDA, ROCm, Triton, CUTLASS
- 编译器: NVCC, LLVM, MLIR, TVM
- 框架编译: torch.compile, ONNX Runtime
- 服务层: vLLM, TGI, SGLang, TRT-LLM

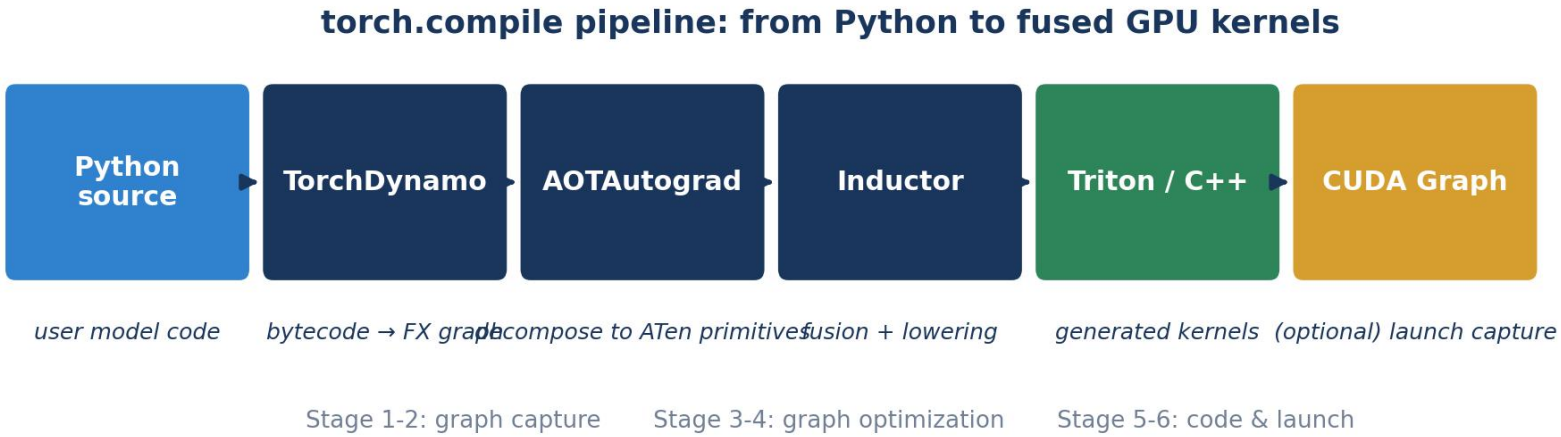
每层做的图优化

- Kernel 层: 单算子高性能
- 编译器层: 算子融合, 调度, layout
- 框架层: 全图 pass, autograd 处理
- 服务层: dynamic batching, KV 调度

图优化不是一个工具的事, 而是从硬件到服务层的多级协作。

6.2 torch.compile + Inductor: PyTorch 2 的核心

用户代码完全不改; 框架在运行时把 Python 函数翻译成融合后的 GPU 程序。



四个连贯的步骤 (输入 → 输出, 一气呵成)

第一步 TorchDynamo: 把 Python 字节码翻成中间表示 FX graph (节点 = PyTorch op, 边 = tensor)
第二步 AOTAutograd: 把 FX 里的高级 op (如 `linear`) 拆成更底层的原语 (`mm`, `add`), 方便后续融合
第三步 Inductor: 在原语级别做融合 / 调度 / 内存规划, 输出 Triton 程序模板
第四步 代码生成: 把 Triton 模板填上具体 shape, 编出可执行 GPU kernel, 缓存以便后续复用

四步走完, 用户 Python 代码就跑在融合 kernel 上, 全过程对用户透明; 第二节会逐步打开每一步看产物。

Ansel et al., PyTorch 2, ASPLOS 2024 (DOI:10.1145/3620665.3640366)

6.3 TensorRT-LLM: NVIDIA 的工业级方案

思路: 离线编译为 engine, 编译期把所有 pattern 优化到位; 运行时只跑 engine。

编译流水

Pattern matcher
- 识别 attention, RMSNorm, gated MLP 等典型子图
- 替换为预先优化好的 plugin

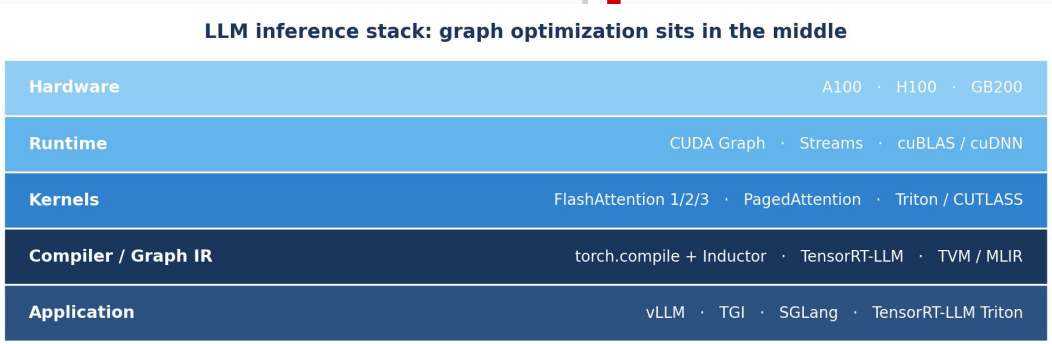
Plugin (手写 kernel)
- FlashAttention v2/v3 内置
- KV cache, rotary, fused MLP 都是 plugin
- FP8 / NVFP4 量化路径

Engine 序列化
- 编译期固化所有 shape 桶
- 运行时无 Python, 直接执行
- 适合稳定模型 + 高吞吐场景

对应本节哪些手段

- 算子融合 (第三部分 ②): plugin 全做
- 基础 pass (第三部分 ③): 编译期完成
- 调度并行 (第三部分 ⑤): TP / PP 调度内置
- 内存规划 (第三部分 ⑥): KV pool, workspace 编译期分配

对比 torch.compile
- 静态化更彻底, 性能更高
- 灵活性更差, 不适合频繁迭代
- 主要用于稳定服务化场景



TensorRT-LLM 把 6 类手段做到工业级稳定, 是高吞吐生产线的常见选择。

TensorRT-LLM 官方文档; NVIDIA H100/H200 推理 best practices

6.4 vLLM piecewise + SGLang RadixAttention

新思路: 不追求全图静态化, 而是 "该静态的部分静态, 动态的部分留 eager"。

vLLM piecewise compilation

核心做法

- Attention 留 eager (动态 shape)
- 其余 (norm + MLP + projection) 进 CUDA Graph
- 减少 launch overhead 同时保留灵活性

典型收益

- Decode launch 占比 50%+ → 个位数
- Continuous batching 兼容动态 shape
- 是 2024-2025 主流推理服务的默认

与本节手段对应

- 主要机会 ② (launch) + ④ (调度)

SGLang RadixAttention

核心做法

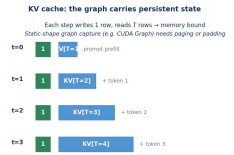
- 把 KV cache 组织成前缀树 (radix tree)
- 多 request 共享 prefix 的 KV
- Long context + 多轮对话场景命中率高

典型收益

- prompt 重复段不再重复 prefill
- 多模型 / agent 场景吞吐显著提升

为什么放在这里

- 是 "图状态 + 内存规划" 的极致案例
- 下节课 L3 会继续展开



vLLM 与 SGLang 表明: LLM 时代图优化要兼顾静态化 + 状态管理两条线。

vLLM blog (2025-08-20); Zheng et al., SGLang, NeurIPS 2024 (arxiv:2312.07104)

第二节课

工具介绍: 怎么用 + 期望效果

torch.compile / TensorRT-LLM / vLLM 选型与上手

本节目标: 学会用工具, 不必手写 GPU 程序

承接 L1: L1 讲清楚了图优化的机会与方法; 本节回答 "我应该用什么工具落地"。

为什么不再写 kernel

- 自己写 CUDA/Triton 门槛高, 调优周期长
- 主流工具栈已经覆盖 90% 常见模型与场景
- 多数加速 (fusion, cudagraph, paged KV) 已工具化
- 学习目标: 选对工具 + 用对模式
- 本节按 “什么时候用 / 为什么用 / 怎么用 / 能拿到什么” 介绍 4 个核心工具
- 不深入编译器内部, 只看用户接口与典型效果

本节覆盖的 4 个工具

- **torch.compile**: PyTorch 项目的默认入口
- **TensorRT-LLM**: NVIDIA 工业级离线编译
- **vLLM**: 在线 LLM 服务的引擎
- **ONNX Runtime / SGLang**: 分支
- 每个工具讲 What / When / How / Effect 四件事
- 最后给一张选型表 + 三条建议

本节是工具介绍课: 知道什么时候选哪个, 会用最少的代码拿到加速。

第一部分

推理图优化的工具

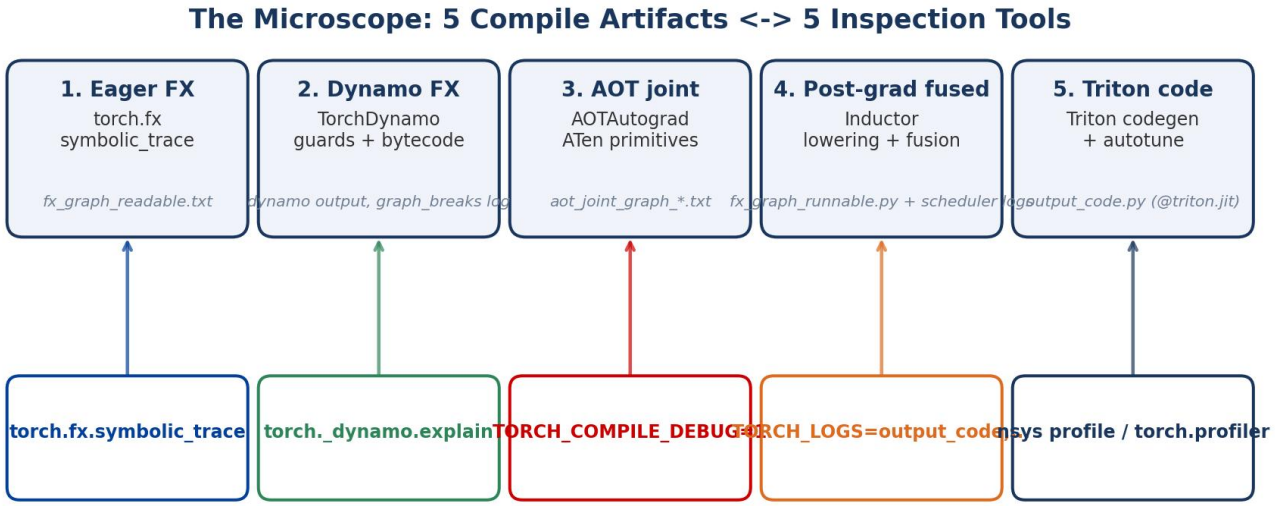
主流工具 + 选型决策

本课内容安排 (进度)

▶ 第一部分	第二部分	第三部分	第四部分	第五部分
工具栈	torch.compile	TensorRT-LLM	vLLM	实战经验与选型

1.1 主流工具家族: 五个最常被提到的名字

一句话定位: 不同工具面向不同场景, 没有 "最好", 只有 "最合适"。



Each stage emits a file; each tool reads one. Master the mapping = master compile.

PyTorch 阵营 (科研友好)

- **torch.compile**: PyTorch 2 自带, 一行启用
- **ONNX Runtime**: 多框架互通, 跨硬件部署
- 适合: 研究, 单机推理, 模型对外发布

生产部署阵营 (高吞吐 / 低延迟)

- **TensorRT-LLM**: NVIDIA 离线编译成 engine
- **vLLM**: 在线服务, PagedAttention + 连续 batching
- **SGLang**: 兼具服务与结构化输出, 增长很快

记住 5 个名字 + 各自定位; 选型就是 "哪个最贴合你的场景"。

PyTorch 2 (ASPLOS 2024); TensorRT-LLM docs; vLLM (SOSP 2023); SGLang (NeurIPS 2024)

1.2 选型决策: 你的场景 -> 推荐工具

先看场景, 再选工具: 大多数时候 torch.compile 就够用, 上线再换。

推理优化工具选型决策树

你的场景是?

研究 / 单卡训练实验 -> torch.compile (default)
本地推理脚本 -> torch.compile (reduce-overhead)
在线 LLM 服务 -> vLLM (OpenAI 兼容 + 高吞吐)
工业级离线部署 -> TensorRT-LLM (engine 模式)
多框架模型互通 -> ONNX Runtime

通用建议: 先 torch.compile, 满足需求就停。

选择的 3 个关键问题

Q1: 你要服务还是脚本?
- 在线服务 -> vLLM / SGLang
- 一次性脚本 -> torch.compile

Q2: 形状是否固定?
- 形状固定 + 离线编译可接受 -> TensorRT-LLM
- 形状变化大 -> vLLM 或 torch.compile (dynamic)

Q3: 是否依赖 PyTorch 生态?
- 强依赖 -> torch.compile
- 跨框架 / 跨硬件 -> ONNX Runtime

选型不是技术决策, 是场景决策; 三个问题就能筛掉 80% 候选。

第二部分

torch.compile: PyTorch 项目的默认入口

一行调用就能拿到 fusion + 调度 + 内存规划

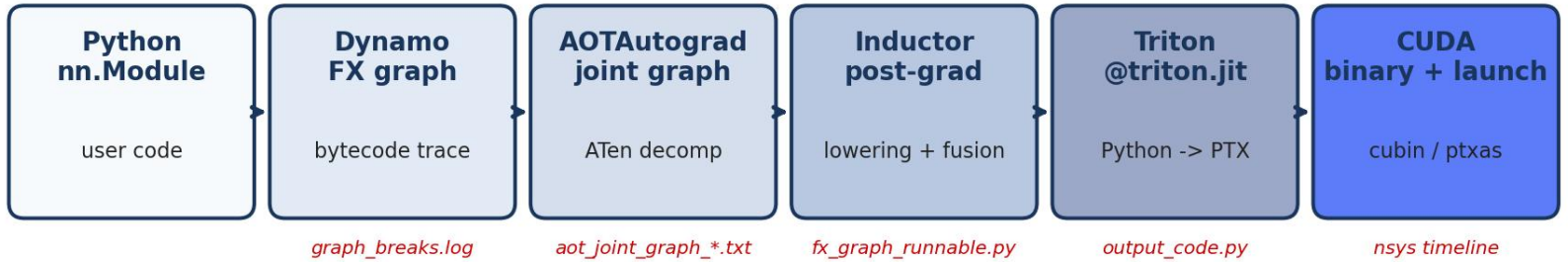
本课内容安排 (进度)

√ 第一部分	▶ 第二部分	第三部分	第四部分	第五部分
工具栈	torch.compile	TensorRT-LLM	vLLM	实战经验与选型

2.1 What: torch.compile 是什么

一行调用: `model = torch.compile(model)`, 不改你的模型代码。

torch.compile Pipeline: Python -> Dynamo -> AOT -> Inductor -> Triton -> CUDA



Each arrow drops a dump file (red label). Lecture 2 walks through every drop.

自动做的三件事

- **融合**: 把多个 element-wise 算子合并成一个 kernel
- **调度**: 选择 Triton 自动生成或调用 cuBLAS
- **内存规划**: 复用中间 buffer, 减少 HBM 访问

底层流程 (大概知道即可)

- TorchDynamo 拦截 Python 字节码
- AOTAutograd 把高层算子展开成 ATen 原语
- TorchInductor 调度 + 生成 Triton 代码
- 用户看到的只有 "快了" 而已

你不需要懂内部; 只需要记得它把 融合 / 调度 / 内存这三件麻烦事代办了。

Ansel et al., PyTorch 2, ASPLOS 2024 (arxiv:2403.07641)

2.2 When/Why: 什么时候应该首选 torch.compile

经验 在 PyTorch 写模型, 想要点免费加速, 就先试它。

适合的场景

- 你在 PyTorch 写模型, 不想换框架
- 想要 "开箱即用" 的加速, 不想手写 kernel
- 训练循环或单机推理脚本
- 模型仍在迭代, 不愿意做离线编译
- 想试 cudagraph 但不想自己 capture
- 典型加速 1.3-2x, 个别 workload 可达 4x+

不太适合的场景

- 在线 LLM 服务 (用 vLLM/SGLang 更直接)
- 形状超动态 (会反复重编译, 不如 eager)
- 模型里大量 Python 控制流
- 想要工业级稳定的多卡部署 (用 TensorRT-LLM)
- 注意: HF generate() 默认有 graph break, 收益受限

默认入口: 任何 PyTorch 项目都值得先 compile 一次试试, 五分钟见效。

2.3 How: 一行调用 + 三个 mode 选项

调用模式: 同样一行 compile, 用不同 mode 控制 "编译多深"。

torch.compile: 一行启用图优化

```
import torch
from my_model import MyModel

model = MyModel().cuda().eval()

# 一行启用: 自动 fusion + 调度 + 内存规划
fast_model = torch.compile(model, mode="reduce-overhead")

out = fast_model(x)  # 首次较慢 (编译), 之后很快
```

三种 mode

default
- 全套 fusion + Inductor
- 编译 ~30s, 收益 ~1.5x
- 日常实验首选

reduce-overhead
- 在 default 基础上加 CUDA Graph
- 适合形状固定的推理 (例如 decode)
- 启动开销趋近 0

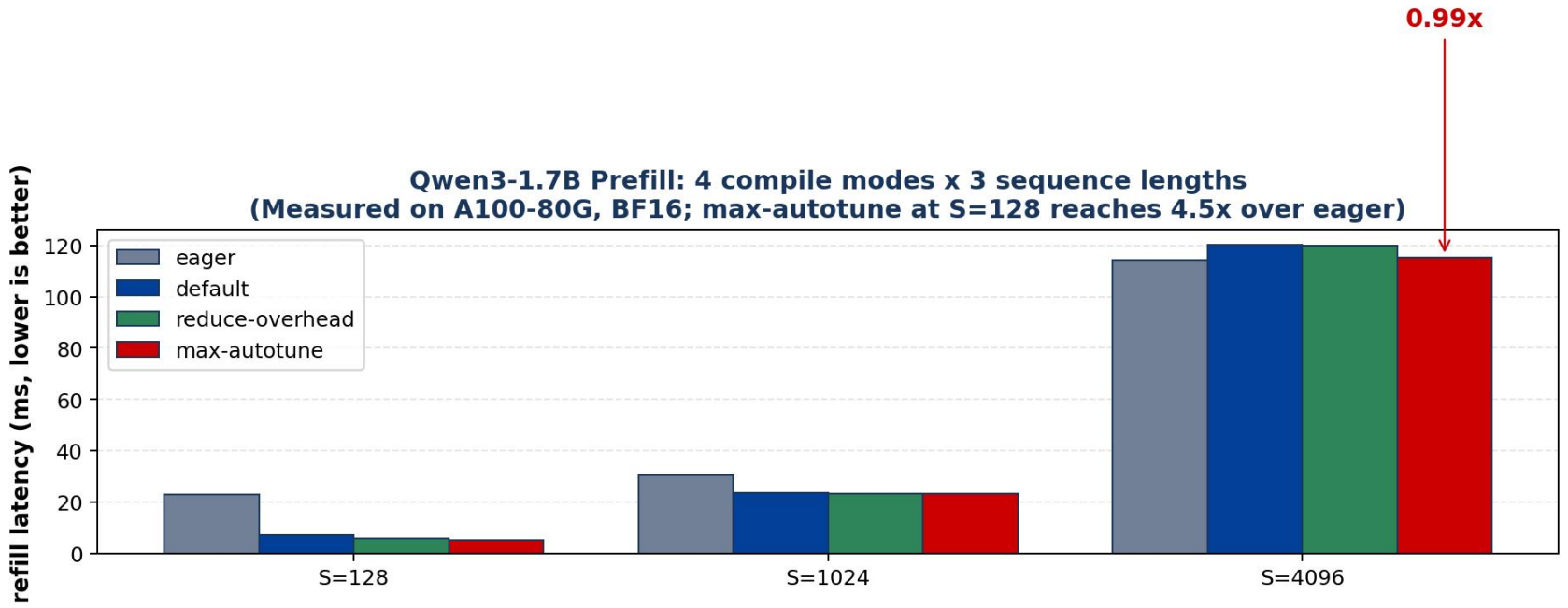
max-autotune
- 加上 Triton GEMM 自动调优
- 编译可能 5-10 min, 之后最快
- 适合部署前一次性

记住三档差别: 编译时间换稳态收益; 上线前再切 max-autotune。

PyTorch docs: torch.compile mode (2024)

2.4 Effect: Qwen3-1.7B 四档实测, 短 prefill 加速 4.49x

实测: A100-80G + BF16, prefill S=128 时 max-autotune 相对 eager 提速 4.49x。



读图发现

- 短序列 S=128: max-autotune **4.49x** eager (22.9 -> 5.1 ms)
- 中序列 S=1024: 加速降到 ~1.32x
- 长序列 S=4096: 几乎追平 (compute-bound, 已逼近峰值)

结论

- 短序列 / 小 batch / decode 收益最大
- 长序列大 GEMM 已饱和, compile 收益边际递减
- 实际项目: 先看你处的工作点, 再决定值不值得 max-autotune

torch.compile 不是处处都快; 越是 launch-bound / BW-bound 的工作点, 收益越大。

Qwen3 (Yang et al., arxiv:2505.09388); A100-80G + BF16 实测 (2026-05).

第三部分

TensorRT-LLM: NVIDIA 工业级离线编译

把整模型烧成 engine, 部署时只跑 engine, 适合大规模线上

本课内容安排 (进度)

√ 第一部分	√ 第二部分	▶ 第三部分	第四部分	第五部分
工具栈鸟瞰	torch.compile	TensorRT-LLM	vLLM	实战经验与选型

3.1 What/When: TensorRT-LLM 是什么, 什么时候选它

离线编译: 编译期把模型 + 权重打包成 engine; 运行时只跑 engine。

What

- NVIDIA 官方面向 LLM 的推理编译器
- 构建期把模型转成 .engine 文件 (固定 shape 范围)
- 内置: kernel fusion, KV cache 优化, FP8/INT8/INT4 量化
- 内置 in-flight batching (类似 vLLM 的连续 batching)
- 与 Triton Inference Server 集成做部署
- 主要在 NVIDIA GPU 上跑 (H100 / A100 优化最好)

When

- 模型已定稿, 上线规模 8+ GPU
- 输入形状有界 (例: max_input=4K, max_batch=32)
- 对延迟和吞吐都很敏感 (例: 客服 / 搜索 / 推荐)
- 不适合: 模型仍在日常改, 形状动态

把 TensorRT-LLM 当 “上线版编译器”; 形状稳定后变成 engine, 之后只管推理。

NVIDIA TensorRT-LLM documentation (2024-2025)

3.2 How: 三步把模型变成 engine

典型流程: 转权重 -> 构建 engine -> 加载 engine 推理。

TensorRT-LLM: 构建一次, 离线烧成 engine

```
# 1) 把 HF 权重转成 TensorRT-LLM 格式
$ python convert_checkpoint.py --model_dir Qwen3-1.7B \
    --output_dir ./ckpt --dtype bfloat16

# 2) 编译成 engine (固定 batch / 最大长度)
$ trtllm-build --checkpoint_dir ./ckpt \
    --output_dir ./engine --max_batch_size 32 \
    --max_input_len 4096 --gemm_plugin bfloat16

# 3) 部署时只加载 engine, 无需 PyTorch
```

关键参数

max_batch_size
- engine 支持的最大并发请求数

max_input_len / max_output_len
- 输入 / 输出 token 上限
- 超出就必须重新 build

dtype
- bfloat16 (默认) / fp8 / int8 weight-only

plugin
- gemm_plugin: 选 cuBLAS 或定制 GEMM
- gpt_attention_plugin: paged KV

engine 一旦 build 就 "冻" 在 shape 范围内; 超界就必须重新 build。

TensorRT-LLM examples/qwen/README.md (2025)

3.3 Effect: 比 PyTorch eager 提升 3-5x, 但 build 要几分钟

典型收益: 同等硬件下 throughput 比原生 PyTorch eager 提升 3-5x。

收益侧

- 单卡吞吐相对 PyTorch eager 提升 **3-5x** (LLM 推理典型)
- 启动延迟更稳定 (engine 内部 launch 已优化)
- 在 H100 + FP8 上可再 2x (受益硬件)
- 内置 in-flight batching, 不必另写 server
- 多卡 TP/PP 自动切分, 不必手写通信

代价侧

- 单次 build 5-30 分钟 (模型越大越久)
- shape 范围一旦改, 必须重新 build
- 模型结构改一行也要重新 build + 测试
- 部署要带 TensorRT-LLM runtime, 镜像变大
- 仅 NVIDIA GPU; 跨厂商需要其他方案

TensorRT-LLM 是 "工程成本换长期性能"; 单条服务每天处理百万级请求才划算。

NVIDIA TensorRT-LLM perf docs; vLLM v0.6 vs TRT-LLM benchmark (公开报告)

第四部分

vLLM: 在线 LLM 推理引擎

OpenAI 兼容 HTTP + PagedAttention + 连续 batching, 一行 serve 即可

本课内容安排 (进度)

√ 第一部分	√ 第二部分	√ 第三部分	▶ 第四部分	第五部分
工具栈	torch.compile	TensorRT-LLM	vLLM	实战经验与选型

4.1 What/When: vLLM 是什么, 什么时候选它

在线服务专用: 想直接搭一个 LLM 服务就用它, 几乎零配置。

What

- UC Berkeley 团队开源, SOSP 2023 发表
- 提供 OpenAI 兼容的 HTTP 服务 (/v1/chat/completions)
- 核心机制 **PagedAttention**: KV cache 分页管理
- 核心机制 **continuous batching**: 不等 batch 凑齐
- 内置 prefix caching, speculative decoding 等
- 支持 HF 上 100+ 模型, 加载即用

When

- 想直接搭一个 LLM HTTP 服务
- 多用户并发, 请求长度差异大
- 想要 prefix cache (例: system prompt 复用)
- 希望快速换模型
- 不适合: 单机本地脚本 (用 torch.compile)
- 不适合: 嵌入式 / 边缘部署 (太重)

vLLM 是 "LLM 服务的默认起点"; 没有特殊理由不会一上来就选 TRT-LLM。

Kwon et al., vLLM, SOSP 2023 (arxiv:2309.06180)

4.2 How: 一行命令起服务, 或者用 Python API

两种入口: 命令行起 HTTP 服务最常用; Python API 适合脚本批处理。

vLLM: 一行起一个 OpenAI 兼容服务

```
# 方式 1: 命令行 (推荐, 直接起 HTTP 服务)
$ vllm serve Qwen/Qwen3-1.7B \
    --max-model-len 4096 --gpu-memory-utilization 0.9

# 方式 2: Python API
from vllm import LLM, SamplingParams
llm = LLM(model="Qwen/Qwen3-1.7B")
out = llm.generate(prompts,
    SamplingParams(temperature=0.7, max_tokens=256))
```

常用参数

--max-model-len
- 上下文上限, 决定 KV cache 占用

--gpu-memory-utilization
- 占多少显存做 KV cache (默认 0.9)

--tensor-parallel-size
- 多卡 TP 分片数

--enable-prefix-caching
- 复用 system prompt 等公共前缀

--quantization
- awq / gptq / fp8 等

默认参数已经足够; 只在显存 / 并发量 / 上下文长度上做微调。

vLLM docs: serving (2024-2025)

4.3 Effect: PagedAttention + continuous batching = 高吞吐

三个核心机制

PagedAttention

- KV cache 按页分配, 解决碎片化
- 显存利用率从 ~40% 提升到 ~90%

Continuous batching

- 新请求随时加入, 不等当前 batch 结束
- 吞吐相对静态 batching 提升 2-4x

Prefix caching

- 公共前缀 (system prompt) KV 复用
- 短回复场景 prefill 几乎免费

效果与限制

- 高并发场景: 吞吐比裸 HF transformers 高 5-10x
- 单用户低延迟场景: vLLM 与 TRT-LLM 接近
- KV cache 大模型: PagedAttention 收益最大
- 内置 OpenAI 兼容接口, 现有客户端代码零改
- 模型 / 量化更新很快, 跟得上社区前沿
- 限制: 极致低延迟场景 (< 50ms) 仍可能不及 TRT-LLM

vLLM 适合大多数在线 LLM 场景; 等真的卡延迟瓶颈了再考虑 TRT-LLM。

Kwon et al., vLLM, SOSP 2023 (arxiv:2309.06180); vLLM v0.6 release notes (2024)

第五部分

工程上的常见坑与选型

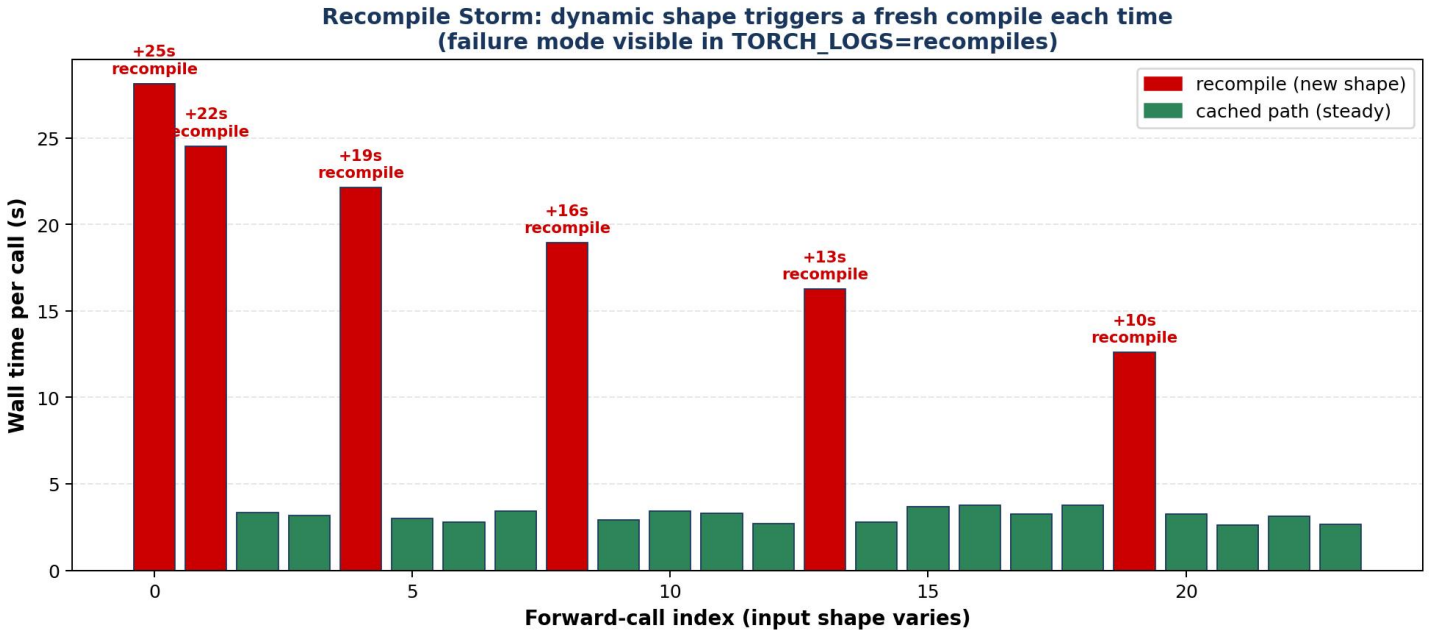
三个常见坑 + 工具对比表 + 给学生的选型建议

本课内容安排 (进度)

√ 第一部分	√ 第二部分	√ 第三部分	√ 第四部分	▶ 第五部分
工具栈	torch.compile	TensorRT-LLM	vLLM	实战经验与选型

5.1 三个常见坑: 用 torch.compile 时最容易踩

踩坑预警: 这三种情况会让 compile 比 eager 还慢。



三个常见坑

1. 动态 shape 反复重编译

- 表现: TORCH_LOGS=recompiles 大量行
- 解法: `compile(dynamic=True)` 或 padding 到固定桶

2. graph break 切断 fusion

- 来源: print, .item(), 列表赋值, custom op
- 解法: `torch._dynamo.explain` 找 break 位置

3. 编译时间过长

- max-autotune 单次 5-10 min
- 解法: 设 TORCHINDUCTOR_CACHE_DIR 缓存

compile 不是 "加上就快"; 踩到这三种情况就要回头排查。

PyTorch troubleshooting guide: torch.compile (2024)

5.3 选型建议: 四个场景四个推荐

建议: 按场景选, 不要追新; 多数情况两个工具组合就够。

按场景看

1. 研究 / 训练实验
- 首选: torch.compile (default 即可)
- 不必上 cudagraph, 训练 shape 变化常触发重编译

2. 单机推理脚本
- 首选: torch.compile + reduce-overhead (cudagraph)
- 配合 BF16 / FP16, 多数模型 1.5-2x 起

3. 工业部署 (形状稳定, 大规模)
- 首选: TensorRT-LLM 烧 engine
- 上线前 build 一次, 之后只跑 engine

4. 在线 LLM 服务
- 首选: vLLM (OpenAI 兼容 + PagedAttention)
- 极致低延迟才考虑 TRT-LLM 替换

三条经验法则

经验 1: 从最简单的开始
- 先 torch.compile 试 default, 不满足再升级
- 工具复杂度与收益不成正比

经验2: 把工具贴近场景
- 在线 LLM 几乎一定是 vLLM 起步
- 训练几乎一定是 torch.compile
- 不要用 TRT-LLM 跑日常改的研究模型

经验3: 关注稳定性, 不只是峰值
- 上线前测 p50 / p99 / 重编译触发频率
- 一次性 benchmark 不代表生产表现

工具是手段, 场景是目的; 选错工具比不优化更耗时。

第三节课

前沿研究 (2023-2026)

超优化器, 可编程 attention, PD 解耦, KV cache 即图状态

第一部分

超优化器: 图重写从规则到搜索

PET · Welder · Mirage μGraph · Hidet (核心问题: 编译器能自己写 FlashAttention 吗?)

本节课内容 (进度)

▶ 一、	二、	三、	四、	五、
超优化器	可编程 Attention	PD 解耦	KV 即图状态	推测 + 稀疏

1.1 问题: 人写 fusion pattern 写不完

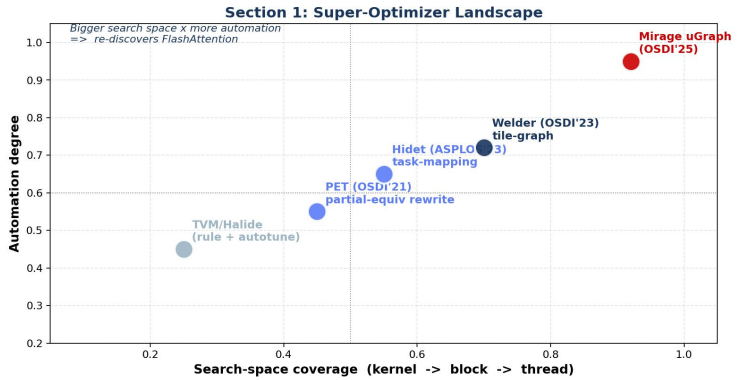
本部分 4 个工作逐步演化: PET 用搜索+修正首次自动找等价图; Welder 把搜索扩到 block tile 级; Mirage 进一步扩到 kernel/block/thread 三级并自动重发现 FlashAttention; Hidet 用 task mapping 给搜索一个更好的表达。

L1/L2 范式的边界

- Fusion pattern 由人在 IR 中匹配
- TVM/Halide schedule 模板由人定义
- 模板覆盖不到的算子组合就漏掉
- FA 等系统级 kernel 不在搜索空间内
- 每出新 attention 变体都要人重写

超优化器要做的事

- 搜索空间: 不只是 schedule, 还包括 algorithm
- 等价验证: 自动 (近似) 等价证明
- 重写粒度: kernel → block → thread
- 目标: 自动重新发现 FA 这类 kernel
- 代价: 搜索时间长, 验证难



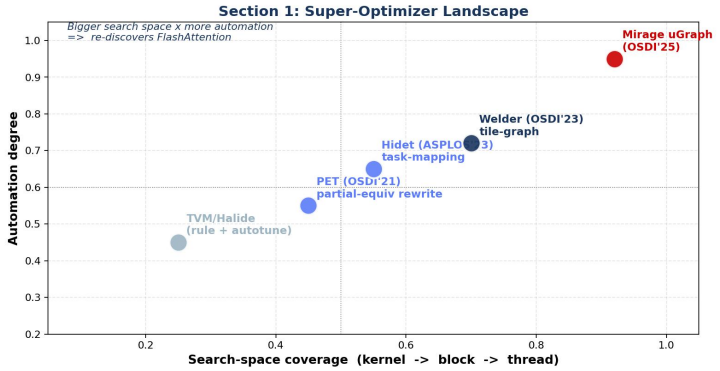
本节第一个主线: 把"写 fusion 规则"变成"搜索 + 验证"的自动化问题。

1.2 PET: 允许中间值有一点偏差, 末尾自动修正 (OSDI 2021)

经典图重写要求处处严格相等, 搜得到的方案少。PET 放松要求, 让搜索能找到更多融合机会。

PET 四点速览

- **问题**: 经典图重写要求每个中间张量都严格等价, 大量"几乎等价"的有效融合被错过
- **创新点**: 允许中间值有少量偏差, 再在末尾加一个修正算子, 让最终输出仍然等价
- **核心方法**: 用 MCMC 等搜索算法枚举重写方案, 自动算出需要的修正项
- **主要结果**: ResNet / BERT 上比 TASO / MetaFlow 找到更多有效融合, 端到端加速 **1.2-2.5×**



PET 让 "等价" 从严格变成 "近似 + 修正"; 但搜索仍局限在算子级, Welder 把它扩到更细的块级。

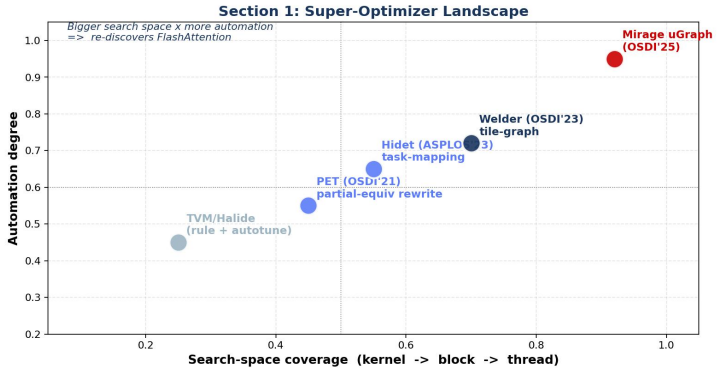
Wang et al., PET, OSDI 2021 (arxiv:2101.01332)

1.3 Welder: 把图的最小节点从 "算子" 换成 "小块" (OSDI 2023)

传统融合一遇算子边界就停; Welder 把整张图重切成更小的数据块 (tile), 融合自然跨过原本的边界。

Welder 四点速览

- **问题**: 经典融合只在相邻算子合并, 相隔几个算子的中间张量仍要反复进出 HBM
- **创新点**: 不再以算子为节点, 把每个算子拆成多个 GPU 一次能处理的 "小块" (tile), 整张图重新连成 tile 图
- **核心方法**: 在 tile 图上自动决定每块多大、按什么顺序算、数据怎么流, 再统一生成 CUDA 代码
- **主要结果**: BERT / GPT-J 等长依赖模型对比 TensorRT 加速 **1.5-2.8×**



把融合的基本单位从算子换成更小的块, 跨算子的融合自动出现; 但仍只在 "块" 一个层次, Mirage 继续扩到 GPU 的多个硬件层级。

Shi et al., Welder, OSDI 2023

1.4 Mirage: 让编译器自己 "搜" 出 FlashAttention (OSDI 2025)

FlashAttention 由 Tri Dao 手写出来; Mirage 把同样的搜索过程自动化, 编译器自己重发现它。

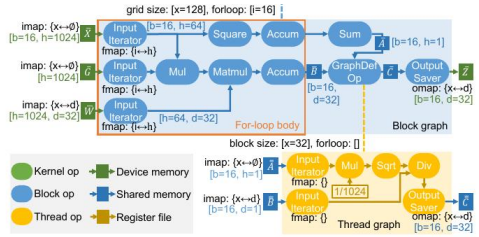
Figure 2: GPU compute and memory hierarchy.

and heavily optimized by existing systems, such as the group-query attention used in LLMs [41], Mirage still outperforms current approaches by up to 3.3× by exploiting subtle custom kernels and optimizations missing in existing systems.

2 Multi-Level Graph Representation

Mirage uses a μGraph to specify the execution of a tensor program on GPUs. A μGraph contains hierarchical graphs at multiple levels to represent computation at the kernel, block, and thread levels[2]. This section first describes the GPU hierarchy and uses Figure 3 as a running example to introduce the key components of a μGraph.

GPU hierarchy. Figure 2 shows the hierarchy of today's GPUs. Computations on GPUs are organized as *kernels*, each of which is a function executed simultaneously on multiple GPU cores in a single-program-multiple-data (SPMD) fashion. A kernel includes a grid of *thread blocks*, each of which is executed on one GPU streaming multiprocessor and includes



(b) The best μGraph discovered by Mirage.

Figure 3: Figure 3a is the computation graph for RMSNorm and MatMul. Figure 3b shows the best μGraph discovered by Mirage for computing RMSNorm and MatMul, which fuses the computation in a single kernel to reduce device memory access and kernel launch overhead, outperforms existing approaches by 1.9×. Numbers in brackets indicate tensor shapes, and numbers in braces show the *imap*, *omap*, or *fmap* for the corresponding operators.

block) graph. As an example, the kernel operator in Figure 3b is a graph-defined operator specified by a block graph.

Block graph. A *block* graph specifies computation associated with a thread block[3], where each node denotes a *block*

Mirage 四点速览

- **问题**: 之前的自动搜索还做不到 FlashAttention 这种 "整层 attention 写进一个 kernel" 的复杂融合
- **创新点**: 同时在 GPU 的三个硬件层次描述计算 — 整个 GPU (kernel) / 一组线程 (block) / 单条线程 (thread)
- **核心方法**: 在三层表示空间里枚举可能的重写, 用 "符号验证 + 数值采样" 混合判断结果是否等价
- **主要结果**: 自动重新发现 FlashAttention; 还找到 GQA+RoPE 等新融合, 对比手写实现 **1.1-3.5×**

编译器开始能完成 "人类研究员才能想到的优化"; 但 GPU 硬件层级的表达仍粗糙, Hidet 给一个更接近真实硬件的写法。

Wu et al., Mirage μGraph, OSDI 2025 (arxiv:2405.05751)

第二部分

可编程 Attention: 编译器的一等公民

FA-1/2/3 · FlexAttention · Ring Attention (核心问题: attention 变体能否一个 DSL 写完?)

本节课内容 (进度)

√ 一、	▶ 二、	三、	四、	五、
超优化器	可编程 Attention	PD 解耦	KV 即图状态	推测 + 稀疏

2.1 问题: 每个 attention 变体都要重写一个 kernel?

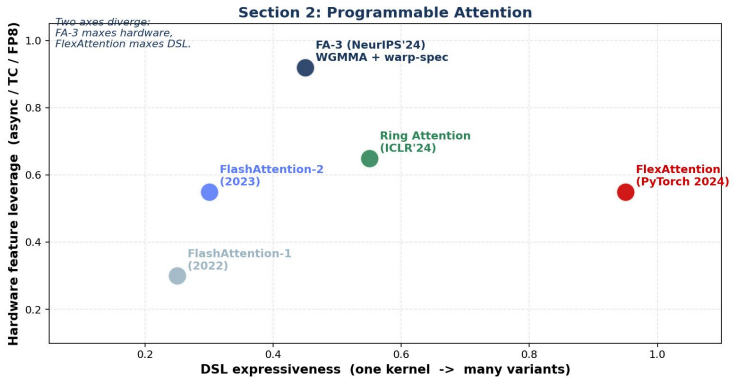
本部分 4 个工作分两条线: FA-2/FA-3 把硬件打满 (异步、WGMMA、FP8); FlexAttention 用 DSL 让一份代码覆盖几十种变体; Ring Attention 把单卡 attention 扩展到跨设备长 context。

重复造轮子的现实

- FA 系列只覆盖少数 mask
- Document mask, ALiBi, soft-cap 都要改 kernel
- 每改一次都要重新 fuse + 调 tile
- 业务方等不起两周写 kernel
- 长 context 还要跨设备分摊

两条解法

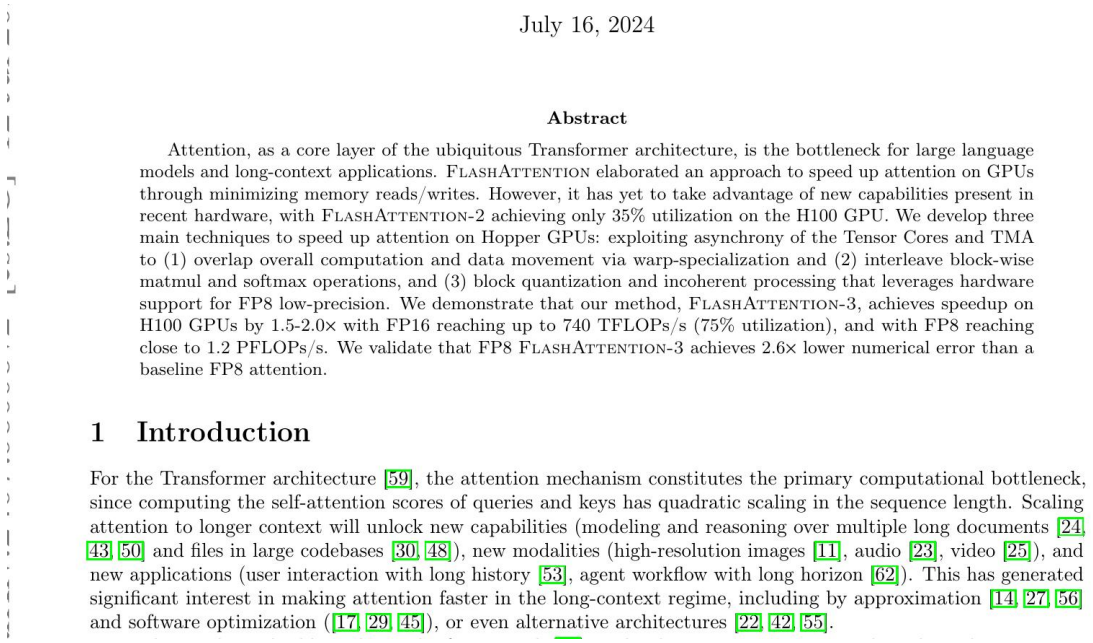
- 解法 A: 一个 DSL 描述变体, 编译器生成 kernel
  → FlexAttention 走这条路
- 解法 B: 把硬件特性榨到极限
  → FA-3 用 WGMMA + warp-spec + FP8
- 解法 C: 跨设备并行 attention
  → Ring Attention 用 P2P 环



本部分主线: DSL 表达力 (FlexAttn) × 硬件 leverage (FA-3) × 跨设备 (Ring)。

2.2 FlashAttention 简史: FA-1 → FA-2 → FA-3 性能曲线

三代 FA 的核心进步: tile + online softmax → 减少非矩阵 op → 异步 + FP8。



July 16, 2024

Abstract

Attention, as a core layer of the ubiquitous Transformer architecture, is the bottleneck for large language models and long-context applications. FlashAttention elaborated an approach to speed up attention on GPUs through minimizing memory reads/writes. However, it has yet to take advantage of new capabilities present in recent hardware, with FlashAttention-2 achieving only 35% utilization on the H100 GPU. We develop three main techniques to speed up attention on Hopper GPUs: exploiting asynchrony of the Tensor Cores and TMA to (1) overlap overall computation and data movement via warp-specialization and (2) interleave block-wise matmul and softmax operations, and (3) block quantization and incoherent processing that leverages hardware support for FP8 low-precision. We demonstrate that our method, FlashAttention-3, achieves speedup on H100 GPUs by 1.5-2.0× with FP16 reaching up to 740 TFLOPs/s (75% utilization), and with FP8 reaching close to 1.2 PFLOPs/s. We validate that FP8 FlashAttention-3 achieves 2.6× lower numerical error than a baseline FP8 attention.

1 Introduction

For the Transformer architecture [59], the attention mechanism constitutes the primary computational bottleneck, since computing the self-attention scores of queries and keys has quadratic scaling in the sequence length. Scaling attention to longer context will unlock new capabilities (modeling and reasoning over multiple long documents [24, 43, 50] and files in large codebases [30, 48]), new modalities (high-resolution images [11], audio [23], video [25]), and new applications (user interaction with long history [53], agent workflow with long horizon [62]). This has generated significant interest in making attention faster in the long-context regime, including by approximation [14, 27, 56] and software optimization ([17, 29, 45]), or even alternative architectures [22, 42, 55].

FA-1/2/3 四点速览

- **问题**: 标准 attention 把整张 N×N 中间矩阵写显存, 长序列下 IO 远超算力
- **创新点**: 三代沿一条线推进: tile+online softmax (FA-1) → 减非 GEMM 调度 (FA-2) → 吃满硬件异步 (FA-3)
- **核心方法**: 每代靠新硬件特性, FA-2 调 work partition, FA-3 用 WGMMA + warp-specialization + FP8
- **主要结果**: FA-3 vs FA-2 在 H100 上 **1.5-2.0×**, FP8 再翻倍, 接近 H100 75% 理论峰值

FA 三步走全靠硬件感知: I/O → 非矩阵 → 异步+低精度 → 接下来 FA-3 详细看 H100 异步流水怎么吃。

Dao et al., FA-1 (arxiv:2205.14135); FA-2 (arxiv:2307.08691); Shah et al., FA-3, NeurIPS 2024 (arxiv:2407.08608)

2.3 FlashAttention-3: warp-specialization + WGMMA + FP8

FA-3 把 H100 的异步特性吃满: producer-consumer 流水 + WGMMA + FP8。

computation. For example, the WGMMA instruction can target the FP8 Tensor Cores on Hopper to deliver 2x the throughput per SM when compared to FP16 or BF16.

However, correctly invoking FP8 WGMMA entails understanding the layout constraints on its operands. Given a GEMM call to multiply $A \times B^\top$ for an $M \times K$-matrix A and an $N \times K$-matrix B, we say that the A or B operand is *mn-major* if it is contiguous in the outer M or N dimension, and *k-major* if is instead contiguous in the inner K-dimension. Then for FP16 WGMMA, both mn-major and k-major input operands are accepted for operands in SMEM, but for FP8 WGMMA, only the k-major format is supported. Moreover, in situations such as attention where one wants to fuse back-to-back GEMMs in a single kernel, clashing FP32 accumulator and FP8 operand layouts pose an obstacle to invoking dependent FP8 WGMMAs.

In the context of attention, these layout restrictions entail certain modifications to the design of an FP8 algorithm, which we describe in §3.3.

2.3 Standard Attention and Flash Attention

Following Dao et al. [17], we let **standard attention** denote an implementation of attention on the GPU that materializes the intermediate matrices **S** and **P** to HBM. The main idea of FlashAttention was to leverage a local version of the softmax reduction to avoid these expensive intermediate reads/writes and fuse attention into a single kernel. Local softmax corresponds to lines 18-19 of the consumer mainloop in Algorithm 1 together with the rescalings of blocks of **O**. The simple derivation that this procedure indeed computes **O** can be found in [15, §2.3.1].

3 FlashAttention-3: Algorithm

In this section, we describe the FlashAttention-3 algorithm. For simplicity, we focus on the forward pass, with

FA-3 四点速览

- **问题**: H100 引入异步 GEMM (WGMMA) 和 TMA, 旧 FA-2 kernel 用不到这些新特性
- **创新点**: 把 attention 改成异步生产者-消费者流水, 同时引入 FP8 量化路径
- **核心方法**: warp-specialization 让部分 warp 取数、另一部分算 WGMMA, 双缓冲 + FP8 block-quant
- **主要结果**: H100 上 FA-2 → FA-3 提速 **1.5-2.0×**, FP8 再翻倍, 接近 75% 理论峰值

FA-3 把硬件吃到极致, 但每加一个 attention 变体仍要重写 kernel → FlexAttention 走另一条路: 用 DSL 让编译器自动生成。

Shah et al., FA-3, NeurIPS 2024 (arxiv:2407.08608)

2.4 FlexAttention: score_mod / mask_mod 一统 attention 变体

PyTorch 把 attention 变成 DSL: 写一个函数, 自动编译成 fused kernel。

FlexAttention is a general programming model for attention that can be used to define many different attention flavors. We observe that many attention variants can be defined as a score modification applied on the intermediate score matrix before conducting softmax (Eq. 1). Additionally, a good portion of these modifications take the form of masks, which set part of the score matrix to `-inf`.

$$\text{Attention}(\mathbf{Q},\mathbf{K},\mathbf{V}) = \text{softmax}(\frac{\mathbf{QK}^T}{\sqrt{d_k}})\mathbf{V}$$

$$\text{FlexAttention}(\mathbf{Q},\mathbf{K},\mathbf{V}) = \text{softmax}(\text{mod}(\frac{\mathbf{QK}^T}{\sqrt{d_k}}))\mathbf{V} \quad (1)$$

With these insights, we build *FlexAttention*, which takes a score modification callable (`score_mod`) and an attention mask callable (`mask_mod`) in addition to tensor inputs. It enables users to implement a new attention variant by defining how to update scores or calculate boolean masks based on positional information. We demonstrate that many existing attention variants (e.g. Alibi, Document Masking, PagedAttention, etc.) can be implemented via *FlexAttention*. In addition, our programming model allows for easy composition of attention variants via nested `score_mods`

to elementwisely mask out score scalars, thus maintaining the semantics. Additionally, `BlockMask` is implemented as a index vector which can also serve as the mapping used in PagedAttention(Kwon et al., 2023a) (subsection 5.1).

We evaluate *FlexAttention* on 7 popular attention variants and compare its performance against today's most commonly used infrastructures: PyTorch SPDA, and handwritten kernels from FlashAttention (FAv2 (Dao, 2024), FAv3 (Shah et al., 2024) and FAKV(Dao et al., 2023)). We show that *FlexAttention* delivers 0.68x-1.43x the performance of FAv2 and 0.93x-1.45x of FAKV for decoding on conventional attention variants supported by FlashAttention. *FlexAttention* works well with existing ML infrastructure and improves end-to-end performance for inference by 2.04x in gpt-fast for 16k context length and training by 2.4x in torchtune. Additionally, we show that *FlexAttention* supports paged attention at negligible overhead.

2 Background

2.1 Attention Variants

The attention mechanism plays a critical role at the core of Transformers. Each attention layer takes three input

FlexAttention 四点速览

- **问题**: causal/sliding/ALiBi/document mask 等几十种变体, 每加一个就要重写 FA kernel
- **创新点**: 把 attention 变体抽象成两个用户函数: score_mod (改分数) 和 mask_mod (改 mask)
- **核心方法**: torch.compile 把用户函数 inline 到 FA kernel 模板里, 一份代码生成所有变体
- **主要结果**: 比手写 SDPA 快 **5-10×**, 接近 FA-2 性能; 与 torch.compile 无缝集成

FlexAttention 把 attention 升级成可编程一等公民 → 但仍是单卡, 长 context 装不下时就要 Ring Attention 切到多卡。

Dong et al., FlexAttention (arxiv:2412.05496); PyTorch blog 2024-08

2.5 Ring Attention: 把超长序列切到多张卡上算 (ICLR 2024)

百万 token 上下文一张卡放不下 K/V; Ring Attention 让多张卡各存一段 K/V, 像传接力棒一样轮流传, 每张卡边算边等下一段。

much greater than the capacity of contemporary GPUs and TPUs, which typically have less than 100GB of high-bandwidth memory (HBM).

To tackle this challenge, we make a key observation: by performing self-attention and feedforward network computations in a blockwise fashion [23], we can distribute sequence dimensions across multiple devices, allowing concurrent computation and communication. This insight stems from the fact that when we compute the attention on a block-by-block basis, the results are invariant to the ordering of these blockwise computations. Our method distributes the outer loop of computing blockwise attention among hosts, with each device managing its respective input block. For the inner loop, every device computes blockwise attention and feedforward operations specific to its designated input block. Host devices form a conceptual ring, where during the inner loop, each device sends a copy of its key-value blocks being used for blockwise computation to the next device in the ring, while simultaneously receiving key-value blocks from the previous one. As long as block computations take longer than block transfers, overlapping these processes results in no added overhead compared to standard transformers. The use of a ring topology for computing self-attention has also been studied in prior work [21] but it incurs non-overlapped communication overheads similar to sequence parallelism, making it infeasible for large context sizes. Our work utilizes blockwise parallel transformers [23] to substantially reduce memory costs, enabling zero-overhead scaling of context size across tens of millions of tokens during both training and inference, and allowing for the use of an arbitrarily large

Figure 1: Maximum context length under end-to-end large-scale training on TPUv4-1024. Baselines are vanilla transformers [37], memory efficient transformers [30], and memory efficient attention and feedforward (blockwise parallel transformers) [23]. Our proposed approach Ring Attention allows training up to device count times longer sequence than baselines and enables the training of sequences that exceed millions in length without making approximations nor adding any overheads to communication and computation.

Ring Attention 四点速览

- **问题**: context 长到 100 万 token 时, K/V 张量超出单卡显存; 但 attention 要让 Q 看到所有 K/V, 不能简单切片
- **创新点**: 把序列切成 N 段, 每张 GPU 只存其中一段的 K/V; N 张卡围成一个环, 计算时每张卡把手里的 K/V 传给下一卡
- **核心方法**: 每张卡边算 Q × 当前手里的 K/V, 边接收上一卡传来的下一段 K/V; 通信和计算同时进行, 网络延迟被算时间掩盖
- **主要结果**: 8×A100 上跑通 **100 万 token** 上下文; 已被 Megatron / DeepSpeed 等并行框架采用

把单卡装不下的超长上下文分摊到多张卡, 再让传输与计算重叠几乎不增加延迟; 但这只解决单次推理, 接下来的第三部分讨论一台集群里多个请求怎么协同。

Liu et al., Ring Attention, ICLR 2024 (arxiv:2310.01889)

第三部分

Prefill/Decode 解耦

DistServe · Splitwise · Mooncake (核心问题: prefill 和 decode 该跑在一台机吗?)

本节课内容 (进度)

√ 一、	√ 二、	▶ 三、	四、	五、
超优化器	可编程 Attention	PD 解耦	KV 即图状态	推测 + 稀疏

3.1 动机: prefill compute-bound, decode memory-bound

本部分 3 个工作沿一条线: DistServe 首次把 prefill/decode 分到不同节点并用 IB 传 KV; Splitwise 从数据中心视角看异构集群; Mooncake 进一步把 KV 抽出来当一等存储, 跨节点共享。

words, we want the serving system to maximize the requests served per second adhering to the SLO attainment goal for each GPU provisioned – *maximizing per-GPU goodput*. Next, we detail the LLM inference computation (§2.1) and discuss existing optimizations for LLM serving (§2.2).

2.1 LLM Inference

Modern LLMs [37, 51] predict the next token given an input sequence. This prediction involves computing a hidden representation for each token within the sequence. An LLM can take a variable number of input tokens and compute their hidden representations in parallel, and its computation workload increases superlinearly with the number of tokens processed in parallel. Regardless of the input token count, the computation demands substantial I/O to move LLM weights and intermediate states from the GPU's HBM to SRAM. This process is consistent across varying input sizes.

The prefill step deals with a new sequence, often comprising many tokens, and processes these tokens concurrently. Unlike prefill, each decoding step only processes one new token generated by the previous step. This leads to significant computational differences between the two phases. When dealing with user prompts that are not brief, the prefill step tends to be compute-bound. For instance, for a 13B LLM

summary, batching prefill and decoding invariably leads to compromises in either TTFT or TPOT.

Model parallelism. In LLM serving, model parallelism is generally divided as intra- and inter-operator parallelisms [33, 46, 59]. Both can be used to support larger models but may impact serving performance differently. Intra-operator parallelism partitions computationally intensive operators, such as matrix multiplications, across multiple GPUs, accelerating computation but causing substantial communication. It reduces the execution time[3], hence latency, particularly for TTFT of the prefill phase, but requires high bandwidth connectivity between GPUs (e.g., NVLINK). Inter-operator parallelism organizes LLM layers into stages, each running on a GPU to form pipelines. It moderately increases execution time due to inter-stage communication, but linearly scales the system's rate capacity with each added GPU. In this paper, we reveal an additional benefit of model parallelism: reduced queuing delay of both prefill and decoding phases, steaming from shorter execution time. We delve into this further in §3. Besides model parallelism, replicating a model instance, irrespective of its model parallelism configurations, linearly scales the system's rate capacity.

These parallelism strategies create a complex space of op-

算术强度差异

- Prefill: 长序列 GEMM, 算术强度高 → compute-bound
- Decode: 单 token attention, 算术强度低 → memory-bound
- 同机 batch 起来反而互相干扰

解耦动机

- Prefill 适合大 batch, 短延迟
- Decode 适合小 batch, 长持续
- SLO: TTFT 与 TPOT 不同优化目标
- 自然结论: 分开跑, KV 传过去

PD 解耦的根本原因: 算术强度差 → roofline 不同 → 该用不同硬件配置。

Patel et al., Splitwise, ISCA 2024 (arxiv:2311.18677); Zhong et al., DistServe, OSDI 2024

3.2 DistServe: 异构集群 + IB 传 KV (OSDI 2024)

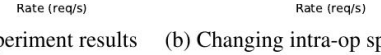
Prefill 和 decode 分别用不同 GPU 配置, KV 通过 InfiniBand 传过去。

(a) Prefill phase (b) Decoding phase

Figure 3: Throughput for two phases with different batch sizes and input lengths when serving an LLM with 13B parameters.

delaying all included requests. Hence, for prefill instances, it is necessary to profile the specific LLM and GPUs in advance to identify a critical input length threshold, denoted as L_m, beyond which the prefill phase becomes compute-bound. Batching more requests should only be considered when the input length of the scheduled request is below L_m. In practice, user prompts typically average over hundreds of tokens [8]. Batch sizes for the prefill instance are generally kept small.

Parallelism plan. To study the parallelism preferences for prefill-only instances, we serve a 66B LLM on two A100 GPUs with inter-op or intra-op parallelism strategy. To simplify the problem, we assume uniform requests input lengths of 512 tokens and a Poisson arrival process. We compare the resulting average TTFT at various arrival rates in Figure 4(a): intra-op parallelism is more efficient at lower arrival rates, while inter-op parallelism gains superiority as the rate increases. Disaggregation enables the prefill phase to function



(a) Real experiment results (b) Changing intra-op speedup

Figure 4: Average TTFT when serving an LLM with 66B parameters using different parallelism on two A100 GPUs.

For intra-op parallelism, we introduce a speedup coefficient K, where $1 < K < 2$, reflecting the imperfect speedup caused by high communication overheads of intra-op parallelism. With the execution time $D_s = \frac{D}{K}$, the average TTFT for 2-degree intra-op parallelism is:

$$Avg\_TTFT_{intra} = \frac{D}{K} + \frac{RD^2}{2K(K-RD)}. \quad (3)$$

Comparing Eq. 2 and Eq. 3: at lower rates, where execution time (first term) is the primary factor, intra-op parallelism's reduction in execution time makes it more efficient. As the rate increases and the queuing delay (second term) becomes more significant, inter-op parallelism becomes advantageous, concurred with Figure 4(a).

The prefill phase's preference for parallelism is also influenced by TTFT SLO and the speedup coefficient K. Seen from Figure 4(a): A more stringent SLO will make intra-op parallelism more advantageous, due to its ability to reduce

DistServe 四点速览

- **问题**: 同机跑 prefill+decode 会互相干扰: TTFT 和 TPOT 不能同时满足 SLO
- **创新点**: 首次提出 prefill / decode 物理解耦, 分别用不同 GPU 配置和并行策略
- **核心方法**: prefill 节点高 TP, decode 节点强带宽, KV 通过 InfiniBand 一次性传; placement 由 SLO 反推
- **主要结果**: 同等硬件预算下 **goodput 4-7×** 于 vLLM, TTFT/TPOT 同时改善

DistServe 开启 PD 解耦研究 → 但只关注单 query 算法, Splitwise 接着从数据中心采购角度看代价。

Zhong et al., DistServe, OSDI 2024 (arxiv:2401.09670)

3.3 Splitwise: 集群级别的 PD 解耦设计 (ISCA 2024)

Microsoft 从数据中心视角设计 PD 解耦, 关注硬件采购与功耗。

ping it with the computation in the prompt phase. As each layer in the LLM gets calculated in the prompt machine, the KV cache corresponding to that layer is also generated. At the end of each layer, we trigger an asynchronous transfer of the KV-cache for that layer while the prompt computation continues to the next layer. Figure 11b shows this asynchronous transfer which reduces the transfer overheads. Layer-wise transfer also enables other optimizations, such as earlier start of the token phase in the token machines, as well as earlier release of KV-cache memory on the prompt machines.

Layer-wise KV-cache transfer happens in parallel with the prompt computation for the next layer. This requires fine-grained synchronization per layer for correctness. Thus, it is possible to incur performance interference and increase the TTFT, especially for smaller prompts. However, for small prompts the total KV-cache size is small and does not need the layer-wise transfer to hide the latency. Since the number of tokens in a batch is already known at the start of computation, Splitwise picks the best technique for KV-cache transfer. It uses serialized KV-cache transfer for smaller prompts and layer-wise transfer and for larger prompts. We show that the overall transfer and interference overheads are relatively small in Section VI-A.

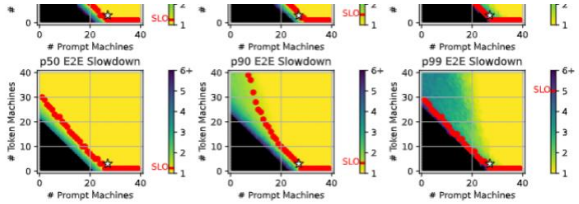


Fig. 12: Design space for provisioning a Splitwise-HH cluster. Cluster configurations targets a peak throughput of 70 RPS. The cost-optimal Splitwise-HH configuration is marked with ⋆ (27 prompt and 3 token machines).

50% of the power. We propose this design based on Figure 9 and Insight VII (*i.e.*, the prompts phase is impacted by power caps while token has no performance impact with 50% lower power cap per GPU).

Number of machines. The LLM inference cluster deployment must be sized with the appropriate number of prompt and token machines. Our methodology involves searching the design space using our event-driven cluster simulator, which is described in detail in Section V. We need to provide as input: (1) the target

Splitwise 四点速览

- **问题**: 数据中心怎么为 LLM 推理混合采购 GPU 才省钱省电?
- **创新点**: 把 PD 解耦推到集群粒度, prompt/token pool 用不同代 GPU, 异构调度
- **核心方法**: prompt pool 用 H100 高算力, token pool 用 A100 高带宽, mixed pool 处理峰值, 调度器路由请求
- **主要结果**: 同等吞吐下功耗 −20%, 成本 −15%; 指出 KV transfer 是核心瓶颈

Splitwise 证明 PD 解耦能省钱省电 → 但 KV 仍只是请求内传一次, Mooncake 把 KV 抽出来当一等存储跨请求共享。

Patel et al., Splitwise, ISCA 2024 (arxiv:2311.18677)

3.4 Mooncake: KV cache 中心化的解耦架构 (FAST 2025 Best Paper)

Moonshot AI 把 KV cache 抽出来当一等存储, 跨节点共享与分层。

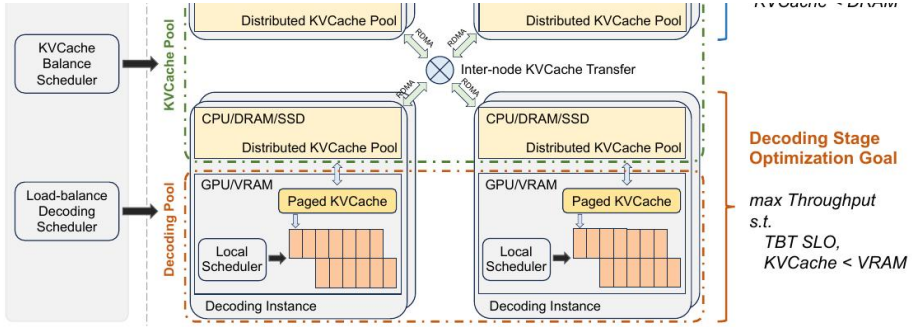


Figure 1: Mooncake Architecture.

remote location will prolong the TTFT, and a large batch size will lead to a larger TBT. Thus, the utilization of both these throughput-oriented optimizations may lead to violations of latency-related SLOs.

According to the above guidelines, we propose a disaggregated design that is centered around KVCache for scheduling and optimization. Figure 1 presents our current **KVCache-centric disaggregated architecture** for LLM serving, named Mooncake. For each request, the global scheduler (Conductor) needs to select a pair of prefill and decoding instances and schedule the request in the

Mooncake 四点速览

- **问题**: PD 解耦后, KV 还只是请求私有的临时数据, 跨请求重复 prefill 浪费严重
- **创新点**: 把 KV cache 抽出来当系统一等存储, GPU/CPU/SSD 分层, 跨节点跨请求共享
- **核心方法**: 全局 KV pool, prefill 写入并按 prefix hash 索引, decode 节点按 hash 拉取, agent 场景前缀命中率高
- **主要结果**: Kimi 生产环境, 同 SLO 下比 vLLM 多 **75% 吞吐**; FAST 2025 Best Paper

Qin et al., Mooncake, FAST 2025 Best Paper (arxiv:2407.00079)

第四部分

KV cache 即图状态: 不是临时变量

vLLM PagedAttention · SGLang RadixAttention · vAttention (核心问题: KV 该用什么数据结构?)

本节课内容 (进度)

√ 一、	√ 二、	√ 三、	▶ 四、	五、
超优化器	可编程 Attention	PD 解耦	KV 即图状态	推测 + 稀疏

4.1 问题: KV cache 不是临时变量, 是图状态

本部分 3 个工作沿抽象层级演化: vLLM 在软件层把 KV 切成 block 做分页; SGLang 在 vLLM 之上加 radix tree 跨请求共享前缀; vAttention 把分页下沉到 CUDA 虚拟内存, 让 vanilla FA 直接跑。

KV cache 的尺度震撼

- LLaMA-70B + 128k context: KV 约 40 GB
- batch=8 时 320 GB, 单 H100 装不下
- 占据推理内存的 30-60%
- 决定 batch size 上限和延迟
- 是 long-context 的核心瓶颈

三种抽象路线

- vLLM PagedAttention: 软件分页
- SGLang RadixAttention: 共享前缀树
- vAttention: CUDA 虚拟内存
- 共同问题: 灵活性 × kernel 兼容性
- 共同启发: OS 思想搬到 ML

KV-cache abstractions: vLLM vs SGLang vs vAttention

	vLLM PagedAttention	SGLang RadixAttention	vAttention (ASPLOS'25)
Abstraction	Logical block table (software paging)	Radix tree of shared prefixes	CUDA Virtual MM (physical paging)
Sharing	Per-request blocks	Cross-request prefix reuse	Per-request, but zero copy
Kernel impact	Custom paged kernel (rewrite needed)	Same as vLLM plus tree mgmt	Vanilla FlashAttn works unchanged
Best fit	General serving, long outputs	Agents, multi-turn shared prefix	Drop-in for any attention kernel

KV cache 不是局部变量, 是"图状态"; 它的数据结构决定推理系统的天花板。

4.2 vLLM PagedAttention: OS paging 类比 (SOSP 2023)

核心想法: 把 KV cache 切成固定大小 block, 用 page table 索引。

Figure 6. Block table translation in vLLM.

divides it into *physical KV blocks* (this is also done on CPU RAM for swapping; see §4.5). The *KV block manager* also maintains *block tables*—the mapping between logical and physical KV blocks of each request. Each block table entry records the corresponding physical blocks of a logical block and the number of filled positions. Separating logical and physical KV blocks allows vLLM to dynamically grow the KV cache memory without reserving it for all positions in advance, which eliminates most memory waste in existing systems, as in Fig. 2.

4.3 Decoding with PagedAttention and vLLM

Next, we walk through an example, as in Fig. 6, to demonstrate how vLLM executes PagedAttention and manages the memory during the decoding process of a single input sequence: ① As in OS's virtual memory, vLLM does not require reserving the memory for the maximum possible generated sequence length initially. Instead, it reserves only the nec-

Figure 7. Storing the KV cache of two requests at the same time in vLLM.

requests and the latest tokens for generation phase requests) as one sequence and feeds it into the LLM. During LLM's computation, vLLM uses the PagedAttention kernel to access the previous KV cache stored in the form of logical KV blocks and saves the newly generated KV cache into the physical KV blocks. Storing multiple tokens within a KV block (block size > 1) enables the PagedAttention kernel to process the KV cache across more positions in parallel, thus increasing the hardware utilization and reducing latency. However, a larger block size also increases memory fragmentation. We study the effect of block size in §7.2.

Again, vLLM dynamically assigns new physical blocks to logical blocks as more tokens and their KV cache are generated. As all the blocks are filled from left to right and a new physical block is only allocated when all previous blocks are full, vLLM limits all the memory wastes for a request within one block, so it can effectively utilize all the memory, as shown in Fig. 2. This allows more requests to fit into mem-

vLLM PagedAttention 四点速览

- **问题**: KV cache 用连续显存预分配, 内存碎片严重 (利用率 ~60%), batch size 上不去
- **创新点**: 把 OS 分页搬到 KV cache, 切成固定大小 block, 用 page table 间接索引
- **核心方法**: KV 切 16-token 物理 block, 每请求一张 logical→physical 表, 配套 paged attention kernel
- **主要结果**: 内存利用率 60% → 96%, 服务吞吐 **2-4×**, 已成事实标准

PagedAttention 把 OS paging 思想 ML 化, 解决单请求碎片问题 → 但多请求间的共享前缀仍重复计算, SGLang 用 radix tree 解决。

Kwon et al., vLLM, SOSP 2023 (arxiv:2309.06180)

4.3 SGLang RadixAttention: 跨请求前缀共享 (NeurIPS 2024)

Agent 场景大量 prompt 共享前缀, 用 radix tree 自动合并 KV。

DSPy) and low-level (e.g., LMQL, Guidance, SGLang). High-level systems provide predefined or auto-generated prompts, such as DSPy's prompt optimizer. Low-level systems typically do not alter prompts but allow direct manipulation of prompts and primitives. SGLang is a low-level system similar to LMQL and Guidance. Table 1 compares their features. SGLang focuses more on runtime efficiency and comes with its own co-designed runtime, allowing for novel optimizations introduced later. High-level languages (e.g., DSPy) can be compiled to low-level languages (e.g., SGLang). We demonstrate the integration of SGLang as a backend in DSPy for better runtime efficiency in Sec. 6.

Runtime optimizations. Fig. 2 shows three runtime optimization opportunities: KV cache reuse, fast constrained decoding, API speculative execution. We will discuss them in the following sections.

3 Efficient KV Cache Reuse with RadixAttention

SGLang programs can chain multiple generation calls and create parallel copies with the "`fork`" primitive. Additionally, different program instances often share some common parts (e.g., system prompts). These scenarios create many shared prompt prefixes during execution, leading to numerous opportunities for reusing the KV cache. During LLM inference, the KV cache stores intermediate tensors from the forward pass, reused for decoding future tokens. They are named after key-value pairs in the self-attention mechanism [51]. KV cache computation depends only on prefix tokens. Therefore, requests with the same prompt prefix can reuse the KV cache, reducing redundant computation and memory usage. More background and some examples are provided in Appendix A.

Given the KV cache reuse opportunity, a key challenge in optimizing SGLang programs is reusing the KV cache across multiple calls and instances. While some systems explore certain KV cache reuse cases [23, 58, 18, 12], they often need manual configurations and cannot handle all reuse patterns

SGLang RadixAttention 四点速览

- **问题**: agent/多轮对话有大量共享 prompt 前缀, vLLM 每请求各自重算 KV
- **创新点**: 在 vLLM block 之上加 radix tree, 跨请求自动复用相同前缀的 KV
- **核心方法**: 用 radix tree 索引前缀 KV, 新请求 prefix-hit 时跳过 prefill, LRU 替换冷前缀
- **主要结果**: 多轮对话前缀复用率 70%+, 端到端吞吐 **2-5×** vs vLLM, agent/RAG 场景收益最大

RadixAttention 在软件层做跨请求共享 → 但 paged kernel 仍要为 vLLM 改写, 新 attention kernel 不兼容, vAttention 把分页下沉到硬件层。

Zheng et al., SGLang, NeurIPS 2024 (arxiv:2312.07104)

4.4 vAttention: CUDA 虚拟内存解耦虚实 (ASPLOS 2025)

让 vanilla FlashAttention 直接跑在分页 KV 上, 不用改 kernel。

Figure 1. PagedAttention involves two layers of memory management: one in user space and one in OS kernel space.

and competitive advantage. To provide an example, Table 7 shows that the paged kernel of vLLM is already up to 2.8× slower than the FlashAttention-2 kernel [37].

3.2 Adds Redundancy in the Serving Framework

PagedAttention makes an LLM serving system responsible for managing the mappings between KV cache and dynamically allocated memory blocks. For example, consider a request that allocates four KV cache blocks over time (left half of Figure 1). These blocks are usually non-contiguous in virtual memory. During the computation of Equation 2, PagedAttention kernel needs to access all the elements of the four KV cache blocks. To facilitate this, the serving system needs to track the virtual memory addresses of KV cache blocks and pass them to the attention kernel at runtime. This approach effectively requires duplicating what the operating system already does for enabling virtual-to-physical address translation (right half in Figure 1).

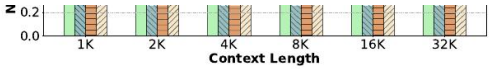


Figure 2. Overhead of PagedAttention in prefill kernels (model: Llama-3-8B, one A100 GPU). Numbers on top show overhead over the corresponding non-paged implementation of FlashAttention-2 (FA2) and FlashInfer (FI).

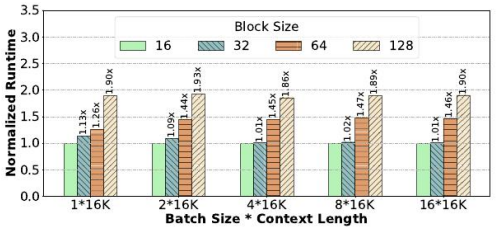


Figure 3. Latency of vLLM's paged decode kernel is sensitive to block size (model: Llama-3-8B, one A100 GPU).

blocks have a higher memory bandwidth utilization due to

vAttention 四点速览

- **问题**: vLLM 的软件分页要为每个新 attention kernel (如 FA-3) 重写一遍, 维护成本高
- **创新点**: 把分页下沉到 CUDA Virtual Memory Management, 由硬件提供虚拟连续
- **核心方法**: KV 在虚拟地址连续、物理可分散; kernel 看到的是普通连续 tensor, 无需 paged 改写
- **主要结果**: 兼容任意 attention kernel (FA-3 直接跑); 比 vLLM 快 **1.2-1.97×** in 多场景

vAttention 让 kernel 写法回归 vanilla, 工程极优雅 → 第四部分的三种抽象层级各有位置, 下一张总结对比。

Prabhu et al., vAttention, ASPLOS 2025 (arxiv:2405.04437)

4.5 vLLM / SGLang / vAttention 三方对比

三种抽象层级一图概括: 哪一层做分页, 决定了 kernel 兼容性。

KV-cache abstractions: vLLM vs SGLang vs vAttention

	vLLM PagedAttention	SGLang RadixAttention	vAttention (ASPLOS'25)
Abstraction	Logical block table (software paging)	Radix tree of shared prefixes	CUDA Virtual MM (physical paging)
Sharing	Per-request blocks	Cross-request prefix reuse	Per-request, but zero copy
Kernel impact	Custom paged kernel (rewrite needed)	Same as vLLM plus tree mgmt	Vanilla FlashAttn works unchanged
Best fit	General serving, long outputs	Agents, multi-turn shared prefix	Drop-in for any attention kernel

选型建议

- **通用 serving**: vLLM PagedAttention (最成熟)
- **Agent/RAG 多轮**: SGLang RadixAttention (前缀复用)
- **要用最新 kernel**: vAttention (FA-3 直接跑)
- 三者可组合: vAttention 底座 + Radix 上层共享

第四部分核心: KV 是图状态, 不同抽象层级 (软件页/前缀树/CUDA VMM) 各有其位 → 第五部分换问题: 不改主模型也不改 KV 结构, 还能从算法层挤出多少加速?

第五部分

推测解码 + 稀疏化: 图变换两条新轴

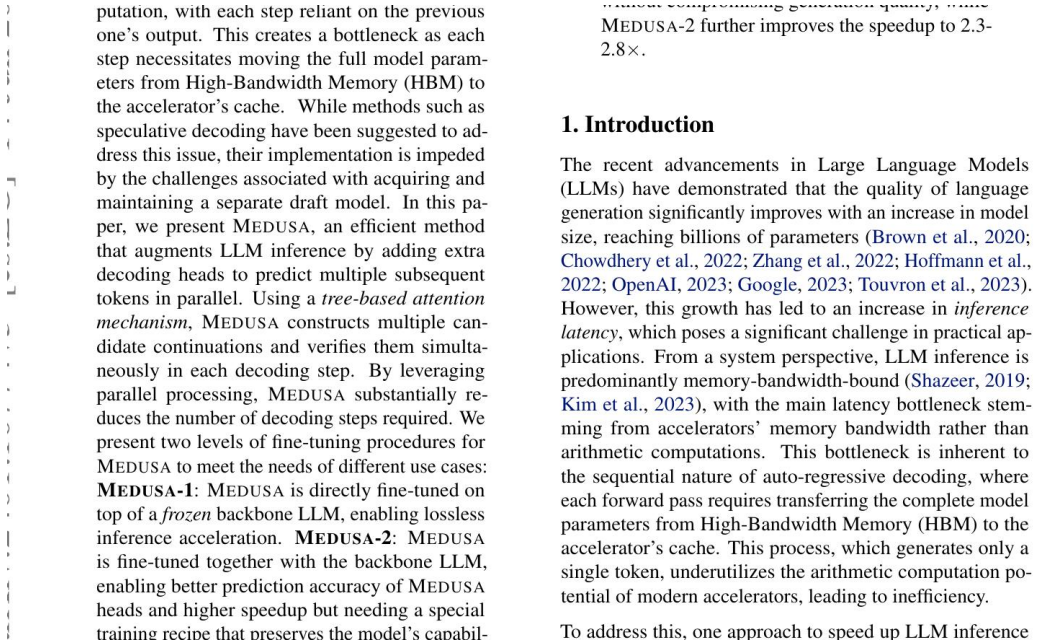
Medusa · EAGLE · Lookahead · H2O · Quest (核心问题: 不改主模型还能再加速多少?)

本节课内容 (进度)

√ 一、	√ 二、	√ 三、	√ 四、	▶ 五、
超优化器	可编程 Attention	PD 解耦	KV 即图状态	推测 + 稀疏

5.1 Medusa: 多头并行预测 + tree attention

本部分 5 个工作分两条线: 投机解码 (Medusa 加 head, EAGLE 用 feature, Lookahead 用 Jacobi) 和 KV 稀疏 (H2O 留 heavy hitter, Quest 按 query 选 page)。



putation, with each step reliant on the previous one's output. This creates a bottleneck as each step necessitates moving the full model parameters from High-Bandwidth Memory (HBM) to the accelerator's cache. While methods such as speculative decoding have been suggested to address this issue, their implementation is impeded by the challenges associated with acquiring and maintaining a separate draft model. In this paper, we present MEDUSA, an efficient method that augments LLM inference by adding extra decoding heads to predict multiple subsequent tokens in parallel. Using a *tree-based attention mechanism*, MEDUSA constructs multiple candidate continuations and verifies them simultaneously in each decoding step. By leveraging parallel processing, MEDUSA substantially reduces the number of decoding steps required. We present two levels of fine-tuning procedures for MEDUSA to meet the needs of different use cases: **MEDUSA-1**: MEDUSA is directly fine-tuned on top of a *frozen* backbone LLM, enabling lossless inference acceleration. **MEDUSA-2**: MEDUSA is fine-tuned together with the backbone LLM, enabling better prediction accuracy of MEDUSA heads and higher speedup but needing a special training recipe that preserves the model's capabil-

MEDUSA-2 further improves the speedup to 2.3-2.8×.

1. Introduction

The recent advancements in Large Language Models (LLMs) have demonstrated that the quality of language generation significantly improves with an increase in model size, reaching billions of parameters (Brown et al., 2020; Chowdhery et al., 2022; Zhang et al., 2022; Hoffmann et al., 2022; OpenAI, 2023; Google, 2023; Touvron et al., 2023). However, this growth has led to an increase in *inference latency*, which poses a significant challenge in practical applications. From a system perspective, LLM inference is predominantly memory-bandwidth-bound (Shazeer, 2019; Kim et al., 2023), with the main latency bottleneck stemming from accelerators' memory bandwidth rather than arithmetic computations. This bottleneck is inherent to the sequential nature of auto-regressive decoding, where each forward pass requires transferring the complete model parameters from High-Bandwidth Memory (HBM) to the accelerator's cache. This process, which generates only a single token, underutilizes the arithmetic computation potential of modern accelerators, leading to inefficiency.

To address this, one approach to speed up LLM inference

Medusa 四点速览

- **问题**: 标准 autoregressive decode 一次只出一个 token, 利用率极低
- **创新点**: 不引入独立 draft 模型, 直接给主模型加几个并行预测 head + tree attention 一次 verify
- **核心方法**: 4-5 个 Medusa head 各预测 t+1/t+2/..., 候选 token 拼成 tree, 用 tree attention 一次 verify
- **主要结果**: Vicuna-7B/13B 上 **2.2-2.8×** speedup, 训练 head 即可不动主模型, TGI/vLLM 已集成

Medusa 不用 draft 模型就能加速, 但 head 预测准确率有限 → EAGLE 把 draft 抬到 feature 级, 命中率更高。

Cai et al., Medusa (arxiv:2401.10774)

5.2 EAGLE: feature-level 自回归 draft (ICML 2024)

在 hidden feature 上跑小自回归 draft, 比 token 级 draft 准很多。

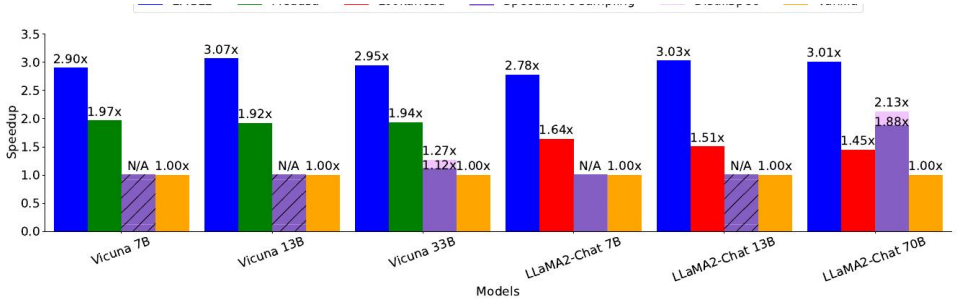


Figure 1: Speedup ratio of Vicuna and LLaMA2-Chat inference latency on the MT-bench for greedy (temperature=0) settings. Speedup ratio of Medusa and Lookahead are copied from their original technical reports. With speculative sampling, there is a lack of suitable draft models to accelerate the 7B model. Employing a 7B model as the draft model for a 13B model results in slow speeds due to the high overhead of the 7B model, rendering it less efficient than vanilla autoregressive decoding. These scenarios are marked as N/A. *In this paper, we only compare with speculative sampling based methods that do not need to finetune the backbone models, ensuring the output text distribution remains constant.*

Abstract

Autoregressive decoding makes the inference of Large Language Models (LLMs) time-consuming. In this paper, we reconsider speculative sampling

of EAGLE, including all models from the Vicuna and LLaMA2-Chat series, the MoE model Mixtral 8x7B Instruct, and tasks in dialogue, code generation, mathematical reasoning, and instruction following. For LLaMA2-Chat 70B, EAGLE

EAGLE 四点速览

- **问题**: Medusa 的并行 head 命中率有限; 传统 token 级 draft 模型信息太少, 抓不到主模型的真实分布
- **创新点**: 在主模型 hidden feature 上跑一个小自回归 draft, 同时预测 feature 和 token
- **核心方法**: Draft 网络以主模型 hidden state 为输入做 feature-level 自回归, 再投影回 token verify
- **主要结果**: LLaMA 上 **3-4×** wall-clock speedup, 命中率超过 Medusa, 与 vLLM 集成顺畅

EAGLE 把 draft 抬到 feature 级, 是投机解码 SOTA → 但仍需训练 draft 模型, Lookahead 走另一条路: 完全 training-free。

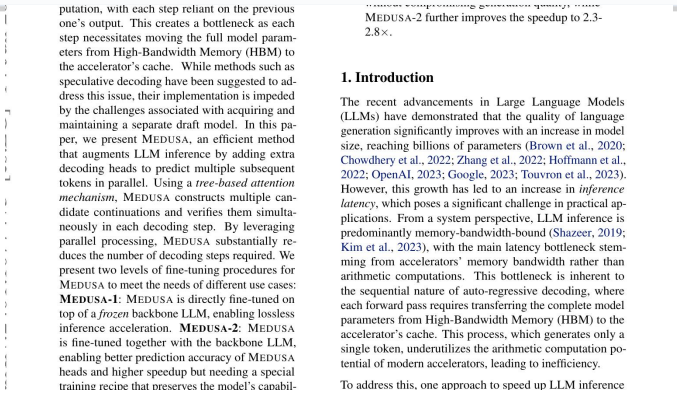
Li et al., EAGLE, ICML 2024 (arxiv:2401.15077)

5.3 Lookahead Decoding: Jacobi 并行解码 (ICML 2024)

用 Jacobi 迭代把 decode 写成并行不动点求解, 不需要 draft 模型。

Lookahead 四点速览

- **问题**: Medusa/EAGLE 都要训练额外 head 或 draft 模型, 工程接入门槛仍存在
- **创新点**: 把若干步 decode 写成一个方程组, 用 Jacobi 迭代并行求解不动点, 完全 training-free
- **核心方法**: 多步 token 候选并行 forward, N-gram pool 缓存历史命中, 与原模型输出严格等价
- **主要结果**: LLaMA-2-7B 上 **1.5-2.3×** 加速, 即插即用, vLLM 已支持, 长 context 收益更明显



putation, with each step reliant on the previous one's output. This creates a bottleneck as each step necessitates moving the full model parameters from High-Bandwidth Memory (HBM) to the accelerator's cache. While methods such as speculative decoding have been suggested to address this issue, their implementation is impeded by the challenges associated with acquiring and maintaining a separate draft model. In this paper, we present MEDUSA, an efficient method that augments LLM inference by adding extra decoding heads to predict multiple subsequent tokens in parallel. Using a *tree-based attention mechanism*, MEDUSA constructs multiple candidate continuations and verifies them simultaneously in each decoding step. By leveraging parallel processing, MEDUSA substantially reduces the number of decoding steps required. We present two levels of fine-tuning procedures for MEDUSA to meet the needs of different use cases: **MEDUSA-1**: MEDUSA is directly fine-tuned on top of a *frozen* backbone LLM, enabling lossless inference acceleration. **MEDUSA-2**: MEDUSA is fine-tuned together with the backbone LLM, enabling better prediction accuracy of MEDUSA heads and higher speedup but needing a special training recipe that preserves the model's capabil-

MEDUSA-2 further improves the speedup to 2.3-2.8×.

1. Introduction

The recent advancements in Large Language Models (LLMs) have demonstrated that the quality of language generation significantly improves with an increase in model size, reaching billions of parameters (Brown et al., 2020; Chowdhery et al., 2022; Zhang et al., 2022; Hoffmann et al., 2022; OpenAI, 2023; Google, 2023; Touvron et al., 2023). However, this growth has led to an increase in *inference latency*, which poses a significant challenge in practical applications. From a system perspective, LLM inference is predominantly memory-bandwidth-bound (Shazeer, 2019; Kim et al., 2023), with the main latency bottleneck stemming from accelerators' memory bandwidth rather than arithmetic computations. This bottleneck is inherent to the sequential nature of auto-regressive decoding, where each forward pass requires transferring the complete model parameters from High-Bandwidth Memory (HBM) to the accelerator's cache. This process, which generates only a single token, underutilizes the arithmetic computation potential of modern accelerators, leading to inefficiency.

To address this, one approach to speed up LLM inference

Lookahead 把投机解码做成纯算法变换, 工程极简 → 但仍是"算更多"的思路; 反过来"算更少"就是 KV 稀疏, H2O 开始。

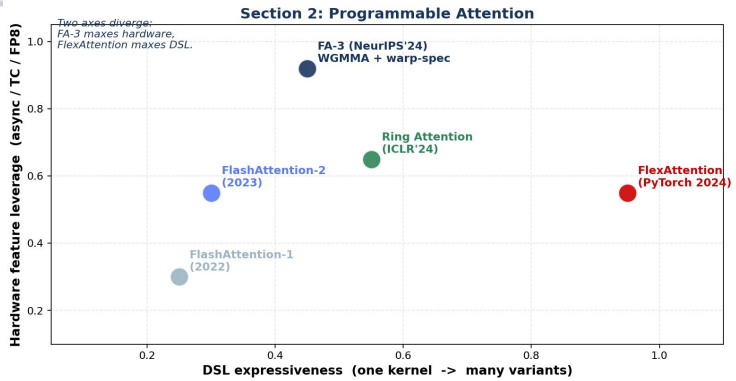
Fu et al., Lookahead Decoding, ICML 2024 (arxiv:2402.02057)

5.4 H2O: heavy-hitter 稀疏 KV (NeurIPS 2023)

少数 token 主导 attention, 只保留这些 token 的 KV: 给图加一个稀疏化算子。

H2O 四点速览

- **问题**: 长 context 下 KV cache 占主导, 但实际 attention 集中在少数 token 上
- **创新点**: 提出 heavy-hitter 观察, 用固定预算只保留高累计分数的 KV, 在线驱逐其余
- **核心方法**: 累计每 token 的 attention score, 每步留 top-k heavy hitter + 最近 R 个, 其余永久淘汰
- **主要结果**: KV 预算 20% 几乎无损, OPT-30B 长 context **5.5×** 加速, 开创 eviction-based 剪枝



H2O 用永久淘汰换显存, 但被淘汰的 token 可能在远后又被需要 → Quest 不淘汰, 改成按 query 动态选 page。

Zhang et al., H2O, NeurIPS 2023 (arxiv:2306.14048)

5.5 Quest: query-aware 页选择 + 第五部分小结

Quest 每步按 query 选 KV page, 不淘汰只重选: 算法 + 系统协同设计。

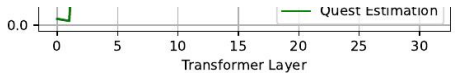


Figure 3. The query aware sparsity for each layer in LongChat-7B model. We measure the sparsity by eliminating KV cache tokens while making sure the perplexity on PG19 increases less than 0.01. For the first two layers, the sparsity is below 10%, while for the rest of the layers, the sparsity is larger than 90%, showing great potential for optimization. Quest closely aligns with the oracle.

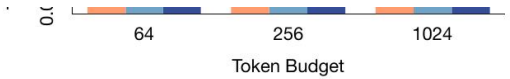


Figure 4. Recall rate of tokens with Top-10 attention scores. Results are profiled with LongChat-7b-v1.5-32k model in passkey retrieval test of 10K context length. Recall rate is the ratio of tokens selected by different attention methods to tokens selected by the full attention in each round of decoding. The average rate is shown in the figure, with various token budgets assigned.

We then present the design of Quest and discuss its benefits.

3.1. Long-context Inference Is Costly

LLM inference contains two stages, namely, the prefill stage and the decode stage. In the prefill stage, all the input tokens are transformed into embeddings and generate the Key (K), Query(Q), and Value(V) vectors. Both the Key and the Value vectors are saved in the KV cache for future use. The rest of the prefill stage includes self-attention and feed-forward network (FFN) layers, which produce the first response token.

time in a decode stage. Therefore, optimizing self-attention becomes a must for efficient long-context inference.

3.2. Self-Attention Operation Features High Sparsity

Luckily, previous research has highlighted the inherent sparsity in self-attention (Zhang et al., 2023b; Ge et al., 2024). Due to this property of self-attention, a small portion of tokens in the KV cache, called critical tokens, can accumulate sufficient attention scores, capturing the most important inter-token relationships. For example, as shown in Fig. 3, apart from the first two layers, less than 10% of the tokens are needed to achieve similar accuracy, which makes the

Quest 四点速览

- **问题**: H2O 永久淘汰 KV, 但某些 token 后期可能又重要
- **创新点**: 不淘汰, 每步按当前 query 动态选 KV page (与 vLLM 分页天然兼容)
- **核心方法**: KV 按 16-token page 切, 每 query 估 page 重要性, 只对 top-k page 跑 attention

第五部分小结

- 投机解码 (Medusa→EAGLE→Lookahead) 一路从加 head 到 feature 到 training-free
- KV 稀疏 (H2O→Quest) 从 eviction 改成 query-aware 动态选 page
- 共同特点: 不改主模型, 与第四部分 vLLM/SGLang 天然组合

第五部分: 不改主模型, 算法层投机+稀疏还能再换 2-7× token/s, 与前四部分正交可叠加。

Tang et al., Quest, ICML 2024 (arxiv:2406.10774)