UIUC CS 498 机器学习系统 学习笔记

Machine Learning Systems — Fall 2026, University of Illinois Urbana-Champaign

课程：CS 498 · Machine Learning Systems（F'26），UIUC Siebel School of Computing and Data Science **讲师**：Fan Lai（赖凡，Assistant Professor，2024 起；PhD'23 "Minimalist Systems for Pervasive Machine Learning"；研究云侧 ML / 端侧 ML / ML4Sys / ML 效率算法；曾任 Visiting Faculty@Google 2023—2024、Meta AI 2022） **助教**：Banruo Liu（banruol2）、Tony Hong（tonyhong）；TA Office Hour：周五 14:00—15:00（Zoom，作业截止日可线下） **上课时间与地点**：周一、周三 09:30—10:45，1310 Digital Computer Laboratory；Instructor Office Hour：3128 Siebel Center，周一 14:00—15:00 **课程主页**：https://github.com/fanlai0990/cs498（分支 `f26-mlsys`）；所有沟通走 Canvas（course id 73559） **材料来源**：本地抓取的课程仓库快照 `/tmp/cs498/`，包含 `README.md`（大纲/日程/评分/必读论文）、`slides/L1_overview.pdf` … `L18_RAG.pdf` 共 18 份讲义及其 `pdftotext -layout` 文本、`hw/README.md`（HW1 编码说明）、`hw/Assignment_1.pdf`（7 页，79 分）、`hw/q2.py` / `q3.py` / `q4.py` / `run.py` / `glooHelper.sh` / `grade_public.py`、`hw/HW2.ipynb`（50 分）、`hw/HW3_AWQ.ipynb`（50 分）、`hw/cloudlab.md` **访问日期**：2026-11（材料快照时间） **版权声明**：讲义、作业、README 的版权归 Fan Lai 与 UIUC 课程团队所有；本笔记为个人学习用途的二次整理、重组与批注，不替代原始材料。笔记中的公式、账本与数字尽量对齐讲义原口径；讲义只有图、文字极少的地方，用 README 的必读论文列表与公开常识补写，并以 `【补充】` 标注；基于材料做的推理以 `【推断】` 标注；只保证数量级的数字以 `【量级】` 标注。若与课程当前仓库或官方文档冲突，以课程仓库与官方文档为准。 **一个重要口径**：本课程讲义（尤其 L6—L8、L14—L18）大量是示意图与动画帧，`pdftotext` 抽出的文字经常只剩标题与几个标注。因此本笔记对这几讲的"公式层"内容做了补写，凡属补写均在段落内标明 `【补充】`；而"课程讲了什么、讲了哪些图"的描述严格来自讲义文本。

1.1 课程定位：不是讲算法，是讲"怎么把算法跑起来"
1.2 这门课的组织形态
1.3 评分结构（合计 100%）
1.4 18 讲地图
1.5 与 AIInfraGuide M2/M3 的分工
1.6 怎么读这套讲义（本笔记的诚实边界）
2.1 五个阶段，五种资源压力
2.2 云侧的一次"台阶式"跃迁
2.3 一台"简化 AI 服务器"的带宽账本
2.4 计算—通信—显存三角
2.5 roofline 视角：算术强度是这门课的"通用货币"
2.5.1 定义与玩具例子
2.5.2 Attention 的 AI 推导（L3 原文）
2.5.3 MLP 的 AI（L3 原文）
2.5.4 这个对照的两个直接推论
2.6 三个"效率/效果/易用"的目标
3.1 为什么是 Transformer：三个系统层面的动机
3.2 位置编码：从绝对位置到 RoPE
3.3 KV Cache：本课程第一个"账本"
3.3.1 KV Cache 的显存公式
3.3.2 为什么 KV cache 让生成变成 I/O 密集型
3.4 FlashAttention：tiling + online softmax
3.4.1 朴素实现的三个问题
3.4.2 两个关键想法
3.4.3 online softmax：为什么需要"运行时修正"
3.5 MHA / MQA / GQA：同一笔账的三种记法

3.6 NSA：把稀疏性做成"可训练"的
3.7 一份完整的参数/算力账本（HW1 Q5 的解法）
4.1 先把问题写成数学
4.2 为什么必须分布式：一张"装不下"的表
4.3 两条历史路线：PS 与 AllReduce
4.3.1 Parameter Server 的编程模型
4.3.2 AllReduce 的路线
4.4 一致性谱系：BSP → 异步 → 有界一致性
4.5 集合通信的词汇表
4.6 Ring AllReduce 的复杂度：2N(P−1)/P
4.7 NCCL：把 ring 做进 kernel
5.1 DP 的两种实现与它们的共同假设
5.2 通信量分析：为什么 DP 的通信"看起来免费"
5.3 梯度累积：用时间换 batch
5.4 L5 结尾的两条"延伸"
5.5 ZeRO 的思路（从 L10 的 panel 页反推）
6.1 为什么 DP 之后还要 TP
6.2 层间 vs 层内：两个正交的切法
6.3 矩阵切分的两种方式：列切与行切
6.4 Transformer 的 TP 方案：一个 AllReduce 都不多
6.5 通信次数：1 个 forward + backward 共 4 次
6.6 一个完整的 TP 通信量算例（HW2 1.3 的解法）
6.7 Megatron 的实验结论
7.1 起点：TP 是通信密集的
7.2 朴素层间并行：串行执行，全程 idle
7.3 Pipeline Model Parallelism：micro-batch 是关键
7.4 气泡的定量：把 L7 的图变成公式
7.5 1F1B：把峰值显存压下来
7.6 Interleaved 1F1B：用通信换气泡
7.7 L7 的三条总结
8.1 3D 并行的拼装方式
8.2 组合并行的性能模型
8.3 TP vs PP 的实测：为什么峰值在 t = p = 8
8.4 DP 与模型并行的对照
8.5 经验法则与它背后的三个约束
8.6 Alpa：把"选并行策略"变成优化问题
9.1 硬件前提：Tensor Core
9.2 为什么不能"纯 FP16"
9.3 三个机制：master weights、loss scaling、算子分级
9.3.1 Master weights（权重主副本）
9.3.2 Loss scaling（损失缩放）
9.3.3 算子分级：哪些能 FP16，哪些不能
9.4 收益与代价的实测
9.5 与并行策略的耦合（课程里没明说，但会影响你的配置）
10.1 问题定义
10.2 动态分配 vs 静态分配
10.3 三个图级优化
10.3.1 In-place（原地写）
10.3.2 Memory sharing（内存复用）
10.3.3 收益的天花板
10.4 Activation Checkpointing：用计算换显存
10.4.1 算法
10.4.2 次线性内存：为什么是 $O(\sqrt{n})$
10.4.3 Selective checkpointing：只丢便宜的
10.5 这一讲与 ZeRO 的分工
11.1 从"全参微调"到 PEFT
11.2 LoRA 的一页纸
11.3 设计选择：为什么加在 attention 上
11.4 服务很多 LoRA：S-LoRA
11.5 dLoRA：从静态放置到动态编排
11.6 课程没讲但必须知道的两件事
12.1 五个场景，五种 SLA
12.2 三条扩展律：为什么推理需求不会下降
12.3 训练 vs 推理：一张对照表
12.4 三个挑战
12.4.1 挑战一：延迟违反 SLA
12.4.2 挑战二：小 batch 限制了并行度
12.4.3 挑战三：显存推高成本
12.5 三层优化栈
12.6 算法层：压缩四件套
12.6.1 稀疏的三种形态（讲义原文对照）
12.6.2 量化的两个层次
12.7 四个指标的严格定义

13.1 从"请求级调度"到"迭代级调度"
13.2 调度器的四条规则
13.3 Continuous Batching 的收益
13.4 Chunked Prefill：把长 prefill 打碎
13.5 Speculative Decoding：为何放在 panel 而不是正文
14.1 学习目标
14.2 从 KV caching 到"内存不够"
14.2.1 prefill 与 decode 的形状差异
14.2.2 不缓存的代价
14.2.3 复杂度降阶
14.2.4 显存是并发的上限
14.3 三种浪费与"预留最大值"的愚蠢
14.3.1 朴素做法
14.3.2 三种浪费（原文定义）
14.4 PagedAttention：把 OS 的分页搬过来
14.4.1 核心思想
14.4.2 分页不改数学
14.4.3 block table 的四步走查（block size = 4）
14.4.4 两个"白送"的能力
14.4.5 内部碎片被"上界化"
14.5 代价：天下没有免费的午餐
14.6 四条 takeaways 与 DistServe 的预告
15.1 分页解决了碎片，但没解决"线性增长"
15.2 三个约束：容量、质量、策略开销
15.3 观察：attention 是稀疏的
15.4 第一个困难：稀疏 ≠ 驱逐策略
15.5 H2O：用累计注意力做保留信号
15.5.1 三条观察
15.5.2 打分规则
15.5.3 保留策略：heavy hitters + recent window
15.5.4 一个完整的算例（讲义给的玩具例子）
15.5.5 H2O 的三个风险
15.6 StreamingLLM：为什么滑窗会崩
15.6.1 现象
15.6.2 原因：attention sink
15.6.3 保留策略：固定 sink + 滚动窗口
15.7 两种策略的正面比较
15.8 开放问题与 SGLang 的接口
16.1 量化的位置：它是 KV 管理的"补集"
16.1.1 省多少：一个 7B 的账
16.2 三种"量化什么"的组合
16.3 均匀仿射量化的两个参数
16.3.1 一个完整的 INT4 算例（讲义原文）
16.3.2 对称 vs 非对称
16.3.3 粒度：一个 scale 还是很多个
16.4 真正的敌人：激活离群值
16.5 SmoothQuant：把难度从激活搬到权重
16.5.1 恒等变换
16.5.2 怎么选 s：α 迁移公式
16.5.3 讲义给的两通道算例
16.5.4 α 的取舍曲线
16.5.5 工程落地：离线准备 + 在线服务
16.5.6 硬件视角：只量化"贵的算子"
16.5.7 效果与局限
16.6 讲义没展开、但作业要考的三块
16.6.1 GPTQ：把量化变成逐列的最小二乘
16.6.2 AWQ：保护 1% 的显著通道
16.6.3 FP8 与 KV 量化
16.7 一个完整的量化吞吐算例（HW3 2.6 的解法）
16.8 L16 的三条总结
17.1 稠密扩展撞墙
17.2 架构：把 FFN 换成"路由的多个 FFN"
17.2.1 路由的数学
17.2.2 容量 ≠ 计算：Mixtral 的例子
17.3 四类系统代价
17.3.1 演进史（讲义时间轴）
17.3.2 四个设计旋钮
17.3.3 两个具体架构
17.4 负载不均：路由的"原罪"
17.4.1 Top-k 不等于均匀
17.4.2 一个课堂小测
17.4.3 Token-choice vs Expert-choice
17.4.4 训练目标的四件套
17.5 专家并行（EP）：把路由变成通信问题

17.5.1 三种可切分的东西
17.5.2 两次 All-to-All
17.5.3 一个可用的延迟模型
17.5.4 拓扑：节点内 vs 节点间
17.5.5 四种并行组合的适用条件
17.6 Tutel：让并行度"随负载变"
17.7 怎么评价一个 MoE 系统
17.8 L17 的全课程串联页
18.1 为什么需要 RAG
18.1.1 流程
18.1.2 检索：一个内积
18.1.3 一个具体的 prompt 构造例子
18.2 RAG 的延迟分解
18.3 前缀复用的边界：SGLang 能做到哪一步
18.4 为什么"独立编码的 chunk KV"不能直接拼
18.5 三条路线
18.6 CacheBlend：选择性重算
18.6.1 核心问题
18.6.2 判据：KV deviation（KV 偏差）
18.6.3 关键假设：偏差排名跨层稳定
18.6.4 渐进式过滤
18.6.5 与 KV 加载 overlap
18.6.6 "One Slide" 总结
18.6.7 收益
18.6.8 什么时候不灵
18.7 Mooncake：把 KV 变成系统的中心
18.8 四个 KV 决策（L18 的收束页）
19.1 AIInfra M2（分布式训练）：公式在那边，直觉在这边
19.2 AIInfra M3（推理优化）：实现在那，演进史在这
19.3 与 CS336、CS144、CS149 的关系
19.4 三门课的"同一道题"对照
20.1 逐讲"能否复述"清单
20.2 自测题（15 题，附答案）
20.3 五个"如果只记五件事"
21.1 课程材料（本地快照，访问日期 2026-11）
21.2 README 列出的必读论文（10 篇）
21.3 讲义正文引用的论文与技术报告
21.4 关于本笔记的三点声明

第 1 章 课程概览：一门"系统视角"的 GenAI 课

1.1 课程定位：不是讲算法，是讲"怎么把算法跑起来"

README 里对定位写得非常直白，这句话值得先抄下来：

> Note that this course is **NOT focused on AI methods**. Instead, we will *focus on how one can build software systems* so that existing AI methods can be used in practice and new AI methods can emerge.

也就是说：这门课不教你设计新的 attention 变体，而是教你**当模型已经给定、硬件已经给定、SLA 已经给定的时候，怎么让这套东西在真实集群上跑得起、跑得快、跑得省**。课程的三条 Learning Objectives 也是按系统口径写的：

#	目标	落到本笔记的对应章节
1	Understand and critique the design principles behind state-of-the-art ML systems, from model architecture to system-level considerations	第 3、6、7、8、14–18 章
2	Develop and utilize tools to profile, monitor, and optimize the performance of ML systems	第 2、12、13 章
3	Explore and conduct research in topics related to the practical deployment and optimization of ML systems	第 20 章（复习与自测）、项目部分

先修要求同样说明这一点：**必须**修过至少一门系统课（OS / 数据库 / 分布式系统 / 网络），**ML/AI 背景 helpful but not required**，但要有"good programming skills（例如 PyTorch）"，因为要做课程项目。

1.2 这门课的组织形态

课程结构是四件套：

1. **Lectures**（含 guest lectures）——18 讲主课 + 若干项目反馈/客座
2. **Lab assignments**——2–3 个（4 学分学生 3 个），主题是分布式训练与推理优化
3. **Reading summaries**——10 篇必读论文里自选 8 篇，每篇 2%
4. **Semester-long project**——4–5 人一组，占 45%（15% 报告演示 + 30% 报告）

有几条"非技术但影响你时间分配"的硬规则，先记住：

规则	内容
出勤	要求全勤，最多 2 次合理缺勤；课堂讨论参与计入 15%
Summary 截止	上课前一天 23:59 前上传 PDF 到 Canvas；**逾期不接受，直接 0 分**
Summary 篇幅	4–5 段，按 P1–P5 模板（问题/影响、主要想法、组件理解、优缺点、改进空间）
Summary 评分	每段 2 分制：2=清晰理解、1=没抓住重点、0=缺失；每篇 10 分
分组	4–5 人，9 月 5 日前登记（表单链接在 README）
Panel Discussion	每组至少各当一次 "Companion（作者方）" 与 "Reviewer（审稿方）"，共 2 次，各 4%
AI 工具	允许做语法检查与头脑风暴；**最终 review 与代码必须自己写**
项目红线	纯综述不允许；必须包含背景与相关工作综述，但主体要有原创贡献

1.3 评分结构（合计 100%）

项目	权重	说明
Attendance（参与）	15%	最多 2 次缺勤
Panel Discussion	8%（4% + 4%）	两次，作者方 + 审稿方
Lab assignments	16%	2–3 个作业（4 学分 3 个）
Reading summary	16%	10 选 8，每篇 2%
Final project presentation	15%	课堂展示，教师 50% + 同学 50% 打分
Project report	30%（5% + 10% + 15%）	Proposal 5% + Mid-term 10% + Final 15%

注意这里有个容易误读的点：**"Final project 占 45%"** 是 L1 讲义的说法，它把 presentation 15% 与 report 30% 加在一起算；README 的表格是拆开列的。两者不矛盾，但引用时要说清楚是"演示+报告"合计 45%。L1 还给了一张"成绩cutoff"表（讲义原文，属于教师自述的评分口径，可能每年调整）：

Grade	Absolute Cutoff (≥)	Relative Bin (≥)
A+	95	Top 5%
⩾ A−	88	Top 35%
⩾ B−	75	Top 80%
⩾ C−	60	—

讲义还补了一句"grade is the better of the two"（绝对值与相对排名取更优者），以及"上一门课 300+ 学生里只有 5 人拿到 ⩽ B"。这条对复习策略有直接影响：**作业与 summary 是保底项，项目是拉开差距项。**

1.4 18 讲地图

README 的 schedule 是"tentative and subject to changes due to the breakneck pace of AI"，下表按快照原样整理，日期为 2026 年秋季学期：

#	日期	主题	讲师	讲义	配套必读 / 里程碑
L1	Aug 24	Course Introduction and Logistics	Fan Lai	`L1_overview.pdf`	—
—	Aug 26	No Class （Project Kickoff & Team Formation）	Fan Lai	—	—

#	日期	主题	讲师	讲义	配套必读 / 里程碑
L2	Aug 31	Transformers	Fan Lai	`L2_Transformers.pdf`	—
L3	Sept 2	Transformers Deep Dive	Fan Lai	`L3_Transformers_Deep.pdf`	首次 paper summary（Sept 9 前）
—	Sept 7	No Class （Labor Day）	—	—	—
L4	Sept 9	Distributed Training Overview	Fan Lai	`L4_distributed_training_overview.pdf`	BLASST（arXiv 2512.12087）
L5	Sept 14	Data Parallelism	Fan Lai	`L5_data_parallelism.pdf`	DeepSeek-V3 Report（Sec 1–3）
L6	Sept 16	Tensor Parallelism	Fan Lai	`L6_tensor_parallelism.pdf`	—
L7	Sept 21	Pipeline Parallelism	Fan Lai	`L7_pipeline_parallelism.pdf`	LlamaRL；**Assignment 1 released**
—	Sept 23	No Class （refine project ideas）	Fan Lai	—	—
L8	Sept 28	Multi-Dimensional Parallelism	Fan Lai	`L8_Multi_Parallelism.pdf`	Alpa（OSDI'22）
L9	Sept 30	Mixed Precision Training	Fan Lai	`L9_mixed_precision_training.pdf`	Project Proposal Due
—	Oct 5 / Oct 7	Project Proposal Feedback I / II	Fan Lai	—	—
L10	Oct 12	Memory Optimization	Fan Lai	`L10_training_memory_opt.pdf`	ZeRO（arXiv 1910.02054）
L11	Oct 14	Finetuning Techniques	Fan Lai	`L11_fine_tuning_techniques.pdf`	**Assignment 1 Due**
L12	Oct 19	Inference Overview	Fan Lai	`L12_inference_overview.pdf`	—

#	日期	主题	讲师	讲义	配套必读 / 里程碑
L13	Oct 21	Batch Serving Techniques	Fan Lai	`L13_inference_scheduling.pdf`	Speculative Decoding；**Assignment 2 released**
L14	Oct 26	Paged Attention	Fan Lai	`L14-Paged-Attention.pdf`	DistServe（OSDI'24）
—	Oct 28	Advanced Topics: Agentic AI Scheduling	Wei Zhang, Zhiyu Wu	—	—
L15	Nov 2	Adaptive KV	Fan Lai	`L15_adaptive_kv.pdf`	SGLang（NeurIPS'24）
L16	Nov 4	Quantization	Fan Lai	`L16_quantization.pdf`	Murakkab（OSDI'26）；Mid-semester Report Due
L17	Nov 9	MoE Efficiency	Fan Lai	`L17_MoE.pdf`	**Assignment 2 Due**
—	Nov 11	Project Check-in & Feedback	Fan Lai	—	—
L18	Nov 18	Advanced Topics: RAG Systems	Fan Lai	`L18_RAG.pdf`	MoonCake（FAST'25）；**Assignment 3 released**（仅 4 学分学生必做）
—	Nov 16	Project Check-in & Feedback	Fan Lai	—	Assignment 3 released
—	Nov 21–29	Fall Break — No Class	—	—	—
—	Nov 30	Guest Lecture	Chen Wang（IBM）	—	—
—	Dec 2 / 7 / 9	Final Presentations	—	—	Assignment 3 Due（Dec 9）

#	日期	主题	讲师	讲义	配套必读 / 里程碑
—	Dec 18	—	—	—	Final Report Due

> 说明：README 表格里 Nov 16 与 Nov 18 两行顺序有交叉（Nov 16 写的是 Check-in + Assignment 3 released，Nov 18 是 RAG），本表按日期排序整理。L14 讲义第 2 页提醒里写的 "Assignment 2 Due Mon, Nov 9" 与 README 一致。

如果把 18 讲按"资源维度"重排，会得到一张更有用的认知地图（这张表是本笔记的整理，不是讲义原表）：

主线	讲次	一句话
模型基础	L2、L3	Transformer 的算力/显存账本、FlashAttention、MHA/MQA/GQA、NSA
训练并行	L4–L8	DP / PS / AllReduce → TP → PP → 3D 并行
训练效率	L9、L10	混合精度（省显存+提吞吐）、激活检查点与显存规划
后训练	L11	LoRA / PEFT / 多 LoRA 服务（S-LoRA、dLoRA）
推理基础	L12、L13	Prefill/Decode、TTFT/TPOT、continuous batching、chunked prefill
推理显存	L14、L15	PagedAttention（分页）、H2O/StreamingLLM（该留哪些 token）
推理压缩	L16、L17	量化（SmoothQuant/AWQ）、MoE（EP + All-to-All）
推理系统	L18	RAG 的检索、CacheBlend 选择性重算、Mooncake KV 池

L1 讲义里给过一张 "System for X (#Lectures)" 的课时分配图，数字是：**ML Basics (2)、Memory Optimization (1)、Opt. Serving Throughput (4)、Guest Lectures (3)、Distributed ML/parallelism (5)、Fine-Tuning / Alignment (1)、Opt. Serving User Experience (3)**（图上另标 Concept / Practice(Agents) 两栏）。这张图解释了为什么课程后半段几乎全在讲推理：**serving 相关的课时（4+3）超过了训练并行（5）之外的所有单项**。

1.5 与 AIInfraGuide M2/M3 的分工

我已经写过《AIInfra M2 分布式训练》与《AIInfra M3 推理优化》两份笔记（同目录），这里必须把分工说清楚，否则会重复劳动：

主题	AIInfraGuide M2/M3 的侧重	CS 498 的侧重	建议读法
集合通信	原语语义、通信量公式推导、代码级 `dist.all_reduce` 用法	**手写** ring all-reduce（isend/irecv），只给复杂度结论	先读 M2 第 2 章建立直觉，再用 CS 498 HW1 Q3 落到代码

主题	AllnfraGuide M2/M3 的侧重	CS 498 的侧重	建议读法
DP / ZeRO	16Ψ/12Ψ 显存账本、ZeRO-1/2/3、FSDP	PS 与 AllReduce 两条历史路线、BSP/异步/有界一致性（staleness）	M2 补公式，CS 498 L4/L5 补"为什么当年先有 PS"
TP / PP	公式化（bubble = (p−1)/(m+p−1)、1F1B、Interleaved）	图示为主 + Megatron 实测（t=p=8 峰值）	CS 498 给直觉与实验，M2 给推导
显存优化	激活重算、offload、序列并行公式	计算图视角：in-place、memory sharing、liveness 分析、checkpointing 的次线性内存	两者互补：CS 498 L10 是"编译/图优化"视角，M2 是"分布式"视角
推理	vLLM/nano-vllm 源码、PD 解耦、量化、speculative decoding 完整推导	Orca 式调度演进、PagedAttention 的 OS 类比、H2O/StreamingLLM、SmoothQuant、CacheBlend	M3 读实现，CS 498 读"设计动机与演进链条"
量化	GPTQ/AWQ/FP8 与 kernel	从 scale/zero-point 讲到 SmoothQuant 的 α 迁移，AWQ 作为 panel 讨论	CS 498 L16 是最清晰的"从零讲量化"材料

一句话总结分工：**M2/M3 是"公式与实现手册"，CS 498 是"系统演进史与设计取舍课"**。CS 498 独有的、M2/M3 里没有的内容至少有五块：① PS 的一致性/陈旧度谱系；② 计算图级显存规划（liveness、in-place、memory sharing）；③ 多 LoRA 服务（S-LoRA / dLoRA）；④ H2O / StreamingLLM 的 KV 保留策略；⑤ CacheBlend + Mooncake 的 RAG 服务栈。

1.6 怎么读这套讲义（本笔记的诚实边界）

必须提醒：18 份讲义里，L1–L5、L9–L13 的文字量是够的（每份 12–40 KB 抽文本），但 **L6、L8 抽出来只有 11 KB 左右，L7 的 bubble 图、L14–L18 的多数图表都只剩标题**。所以：

- 讲"课程讲了哪些图、哪些结论、哪些数字"——严格照讲义文本；
- 讲"这张图背后的公式"——标 `【补充】`，来源是 README 列的必读论文 + 公开常识；
- 讲"我自己的推论"——标 `【推断】`；
- 讲数字——能算的当场算给你看（比如 KV cache 的 GiB、TP 通信的 MiB），算不出的标 `【量级】`。

第 2 章 ML 系统全景：生命周期、三角约束与 roofline

2.1 五个阶段，五种资源压力

L1 用一张"GenAI Lifecycle"图把整门课串起来，图上的五个阶段各自标注了规模量级：

阶段	讲义标注的规模	主要资源压力	对应讲次
Data（Globally generated）	全球产生的数据	存储、清洗吞吐、数据加载成为训练瓶颈	L1、L4
Training（预训练）	Thousands of GPUs	算力 + 显存 + 集群内通信	L4—L10
Tuning / Alignment	Hundreds of GPUs	显存（优化器状态）+ 多租户调度	L9—L11
Grounding（RAG / 工具 / 检索）	Millions of users	prefill 计算、KV 复用、检索延迟	L18
Inference Serving	O(1M) requests	显存带宽、KV cache 容量、调度	L12—L17

讲义在这张图上叠了五个"系统工程手段"的标签，顺序很有讲究，基本就是课程的技术目录：

1. **Model Auto-Sharding**（自动切分）——L6—L8
2. **Adaptive Runtime**（运行时根据负载动态激活组件，例如 MoE、稀疏）——L17
3. **Multi-job Scheduling**（多任务争抢共享资源）——L1、L18
4. **Model Quantization**（CPU+GPU 混合精度/低比特）——L16
5. **Low-rank Adaptation**（小矩阵微调）——L11
6. **Load Balancing**（O(1M) 请求 → Request Scheduler → Model 1..N，图上画了个 ❌ 404 表示不均衡会掉请求）——L13、L17
7. **Inference Support**（把上面这些服务于推理）——L12—L18

2.2 云侧的一次"台阶式"跃迁

L1 的第一张技术图是 "Cloud Enabled Last Leap in ML"，把 ML 性能的跃迁归因于三股力量的叠加：

```
        硬件资源（GPU/TPU 集群）
                  ↘
        大数据（PB 级日摄入）   →  ML Performance  →  ML-based *
                  ↗
        ML 理论（Transformer 等）
```

讲义引用的两个数字（出自 Meta 的 *Sustainable AI*, MLSys'22）：

- **模型规模每两年翻一倍**（Model size doubled every two years）；
- **每天 PB 级的数据摄入**（Petabytes of data ingestion every day）。

这两条合起来就是本课程存在的理由：**模型翻倍 → 单卡装不下 → 必须切分；数据 PB 级 → 单卡算不完 → 必须并行。**

2.3 一台"简化 AI 服务器"的带宽账本

L1 画了一台服务器的框图，标注了各层带宽，这是全课程最基础的一张"数字表"，后面所有 roofline 分析都建立在它上面：

链路	讲义标注带宽	备注
DDR5（CPU 内存）	64 GB/s	每通道
PCIe v5（CPU↔GPU）	4 GB/s	每通道（讲义标注）
Ethernet（跨机）	12 GB/s	简化图，不含 RDMA
NVMe SSD	7.5 GB/s	—
SATA SSD	600 MB/s	—

讲义自己声明了两处简化：**不含更快的网络（RDMA）**，**不含专用 GPU 互连（NVLink）**。但正是这两个"被省略的东西"决定了并行策略的选择——第 8 章会看到 Megatron 的实验结论：TP 只在一个节点内（8 卡 NVLink）划算，跨节点就必须换 PP。

【补充】把这张表换算成"数量级感受"，会更直观（以 H100 级别硬件为例，仅作量级参考）：

层级	带宽量级	相对 GPU HBM 的比例
GPU HBM	~3 TB/s	1×
NVLink（节点内）	~900 GB/s	~0.3×
InfiniBand / RoCE（节点间）	~50–400 GB/s	~0.02–0.13×
PCIe v5 x16	~64 GB/s	~0.02×
DDR5	~0.5–1 TB/s（整机）	—
NVMe	~7 GB/s	~0.002×

这张表解释了三件事：① 为什么 TP 必须待在节点内（每层两次 AllReduce 走 NVLink 才不亏）；② 为什么 ZeRO-Offload 把优化器状态丢到 CPU 内存是"用带宽换算力"；③ 为什么 RAG 的 KV 复用（CacheBlend/Mooncake）要把 KV 放在 DRAM/SSD 上还能赚——因为省下的是 prefill 的**算力**，付出的是**带宽**。

2.4 计算—通信—显存三角

课程反复回到同一个三元权衡。我在 M2 笔记里把它叫做"三角"，CS 498 没有给这个名字，但 L6、L7、L8 的 summary 页几乎逐字在讲这三角：

维度	手段	代价	讲义出处
装不下（显存）	切模型：TP / PP / ZeRO	TP 通信密集；PP 有 bubble	L6 总结页："Tensor parallelism (TP)：split matrix row-wise or column-wise"、"Reduce synchronization cost is crucial as TP is communication-intensive"
跑不快（通信）	换拓扑：ring / tree、NVLink、overlap	树形 AllReduce 对 straggler 更敏感；overlap 需要切分通信	L5："Collectives are implemented using many P2P"、"Collectives are not fault-tolerant"
算不久（算力）	混合精度、稀疏、MoE、量化	精度/质量风险；MoE 引入 All-to-All	L9、L16、L17

L8 的总结页给了这门课最重要的一句"经验法则"（原文）：

> **Rule of thumb**: – Use tensor parallelism **within a node** and pipeline model parallelism to scale to **multiple nodes** – Decide tensor-parallel size and pipeline-parallel size based on **GPU memory size** – Data parallelism can be used to scale to **more GPUs**

这三句就是 3D 并行的配置口诀，第 8 章会展开它的量化依据。

2.5 roofline 视角：算术强度是这门课的"通用货币"

L3 用整整一节课讲 Arithmetic Intensity（AI = #ops / #bytes），这是本课程唯一一个能贯穿训练与推理的分析工具。

2.5.1 定义与玩具例子

系统性能被两个硬件上限夹住：**峰值算力（FLOPS/s）**与**内存带宽（GB/s）**。两者的比值就是硬件的"脊点"（ridge point）：

$$\text{ridge} = \frac{\text{peak FLOPS}}{\text{bandwidth}} \ (\text{ops/byte})$$

L3 的玩具例子：`C[i] = A[i] + B[i]`，两次 load、一次 store、一次加法 → AI = 1 op / 3 × 4 bytes（FP32）≈ 0.083 ops/byte，或者按"每次访存 1 op"记作 1/3。**AI 远低于脊点 → 内存 bound**，加再多算力也没用。

2.5.2 Attention 的 AI 推导（L3 原文）

记号：h = QKV 隐藏维（常 4096），N = 输入长度（如 4096 tokens）。Attention 分三步：

步骤	计算	FLOPs
Step 1: $QK^\top$	$(N \times h) \times (h \times N)$	$2hN^2$

步骤	计算	FLOPs
Step 2：softmax	对 $N \times N$ 逐元素	$3N^2$
Step 3：$\times V$	$(N \times N) \times (N \times h)$	$2N^2h$
合计		$(4h+3)N^2$

访存（FP16，2 bytes/element）：

$$\text{Memory IO} = 8N^2 + 8Nh \text{ bytes}$$

于是：

$$\text{AI}_{\text{attn}} = \frac{(4h+3)N^2}{8N^2+8Nh} \xrightarrow{N\text{大}} \frac{4h+3}{8} \approx \frac{h}{2}$$

这条式子说明：**N 越大，attention 的算术强度越高，越可能从 memory-bound 翻到 compute-bound**。讲义的原话是"Arithmetic intensity increases as the input length N grows"、"Attention can become compute-bound as N grows, if its AI crosses the hardware's roofline ridge point"。

2.5.3 MLP 的 AI（L3 原文）

MLP 本质是 GEMM：$X(N,K) \times W(K,M)$。

$$\text{Compute} = 2NKM, \qquad \text{Memory} = 2NK + 2KM + 2NM \ (\text{FP16})$$

取 $N = K = M = 4096$：compute $= 2 \times 4096^3 = 1.374 \times 10^{11}$ FLOPs，memory $= 6 \times 4096^2 = 1.007 \times 10^8$ bytes，比值：

$$\text{AI}_{\text{MLP}} = \frac{2NKM}{2NK+2KM+2NM} = \frac{4096^3}{3 \times 4096^2} \approx 1365 \text{ ops/byte}$$

L3 用它对照 A10：**125 TFLOPS / 600 GB/s = 208.3 ops/byte** 的脊点。1365 ≫ 208.3 → **MLP 是 compute-bound**。

2.5.4 这个对照的两个直接推论

1. **训练时 MLP 是算力瓶颈，attention 在短序列下是带宽瓶颈** → 所以 FlashAttention 做的是"减少 HBM 往返"而不是"减少 FLOPs"（第 3 章）。
2. **推理 decode 时 N=1** → 所有 GEMM 退化成 GEMV，AI 掉到 ~1 ops/byte 量级，**整个 decode 阶段都是 memory-bound**。L12 用这张图画得很清楚：

```
训练（N=2K/4K…）：  Input (N × 8192) × Weight (8192 × 4096)   → AI 高
推理（N=1）       ：  Input (1 × 8192) × Weight (8192 × 4096)   → AI ≈ 0.5 ops/byte
```

L12 的原文结论是 "Small batch size → Low latency yet low data reuse (arithmetic intensity)"。这一条是理解后面所有推理优化（batching、量化、speculative decoding、KV 复用）的总纲：**推理优化几乎全部在**

解决"权重读一次只算了一个 token"这件事。

2.6 三个"效率/效果/易用"的目标

L1 用三句话定义"什么叫支持好 ML"：

目标	讲义原文	度量
Efficiency	Efficient use of hardware for high scalability and throughput	吞吐、MFU、扩展效率
Effectiveness	High accuracy and fast convergence	精度、收敛步数
Easy to use	Improve development productivity for developers and users	API 复杂度、调试成本

这三者经常互相拉扯：量化提效率、掉效果；ZeRO 提容量、增通信；continuous batching 提吞吐、涨 TPOT。第 12、13 章会把这张三角具体化成"SLA 下的 goodput 最大化"。

【推断】我认为这门课真正想训练的能力是：**拿到一个新系统/新论文时，能立刻问出"它动了三角的哪条边、代价记在谁头上"**。L14 的 PagedAttention 是最标准的例子——它不减少一个字节的 KV，只是把"分配浪费"消掉，就把吞吐提上去了（第 14 章）。

第 3 章 Transformer 的系统视角（L2 / L3）

3.1 为什么是 Transformer：三个系统层面的动机

L2 从 RNN 的痛点讲起，但它的三条理由全是"系统理由"而不是"建模理由"：

RNN 的问题	系统后果
Sequential computation prevents parallelization	时间步串行 → GPU 的几千个 SM 大部分闲置，无法用 batching 之外的维度填满
Vanishing / exploding gradients（长程依赖）	只能靠门控（LSTM/GRU）缓解，模型变复杂但并行度不变
不增加每层计算复杂度的前提下无法避免梯度消失	于是需要一种"每层复杂度不变、但可以全并行"的结构

Transformer 的答案是把"沿时间递归"换成"沿序列做一次全对全的加权求和"：**用 $O(N^2)$ 的计算换 $O(1)$ 的串行深度**。这笔交易在系统上极其划算——因为 N^2 的计算可以切成矩阵乘法喂满 tensor core，而串行深度直接决定墙钟时间。

L2 列出的组件清单（原样）：Tokenization、Input embeddings、Position encodings、Query/Key/Value、Self-attention、Multi-head attention、Feed forward、Add & norm、Residual、Masked attention、

Causal attention、Linear、Softmax、Encoders、Decoder、Encoder–decoder。

3.2 位置编码：从绝对位置到 RoPE

L2 讲了两代位置编码：

1. **Absolute positional encoding**（原始 Transformer）：token embedding 与绝对位置 embedding **逐元素相加**。
2. **RoPE**（RoFormer, 2021）：把 token 的 embedding **按其位置旋转一个角度**，从而把"相对位置"编码进内积。

讲义对 RoPE 的描述是三段式动画（同一页重复出现三次），核心信息是：

- 旋转角度由 **token 的位置 p** 决定；
- 旋转的"档位"由一组**预先确定的角频率** $\theta_1, \theta_2, \ldots$ 给出（讲义举的例子是 $\theta_1 = 1$、$\theta_2 = 0.01$）；
- 对 query 向量（例如 d=1024 维）按维度两两成对做旋转。

【补充】RoPE 的标准形式（讲义只有图）：对第 i 对维度，旋转角为 $p \cdot \theta_i$，其中 $\theta_i = 10000^{-2i/d}$。旋转后 $q_p^\top k_m$ 只依赖 $(p - m)$，这正是"相对位置"的代数来源。系统含义有两个：① 位置信息不占额外显存（不需要 position embedding 表）；② **KV cache 里的 K 已经带位置信息，所以缓存的是"旋转后的 K"**，这也是为什么长上下文外推（context extension）必须动 θ 的基频而不是重算 cache。

3.3 KV Cache：本课程第一个"账本"

L2 用一个问号把 KV cache 的必要性问出来了：

> **Question: why there is no such Q cache?**

答案（讲义未写，但它是本课程后续一切推理优化的起点）：**Q 只用一次**。当前 token 的 query 与历史 K/V 做一次点积就算完了，之后再也不需要它；而 K/V 会被**后续每一个 token** 反复读取。所以缓存的是"被复用的东西"，这是缓存设计的第一性原理。

【补充】严格说，KV cache 带来的是复杂度降阶：

方式	生成第 t 个 token 的 attention 计算量	说明
无 cache（重算全前缀）	$O(t^2 \cdot H)$	前面所有 query 都要重算（L14 原文口径）
有 cache（增量 decode）	$O(t \cdot H)$	一条 query 行对 t 个缓存 K/V

L14 把这句写成了两行对照表："Recompute full prefix → $O(t^2 \cdot H)$，Processes all earlier queries again" / "Incremental KV decode → $O(t \cdot H)$，One new query attends to cached K/V"，并给了一句总结：**"KV caching exchanges repeated computation for persistent GPU memory."**

3.3.1 KV Cache 的显存公式

【补充】单个请求的 KV cache 字节数：

$$\text{KV bytes} = 2 \times L \times H_{kv} \times D \times T \times b$$

其中 L = 层数，H_{kv} = KV 头数（GQA 下小于 query 头数），D = head dim，T = 序列长度（已缓存 token 数），b = 每元素字节数（BF16 = 2），系数 2 是 K 和 V 各一份。

L15 用同一公式给了三个具体数字（讲义标注为示意算术，配置是 **32 层、8 个 KV 头、head dim 128、BF16**）：

序列长度 T	每请求 KV 大小	计算过程
8k	1.0 GiB	$2 \times 32 \times 8 \times 128 \times 8192 \times 2 = 1.074 \times 10^9$ B $= 1$ GiB
32k	4.0 GiB	线性增长 ×4
128k	16.0 GiB	线性增长 ×16

这个算例的价值在于：**每 token 是 128 KiB**（$2 \times 32 \times 8 \times 128 \times 2 = 131072$ B）。记住这个数，任何 GQA 模型的 KV 预算都能心算：`每 token 128 KiB × 并发序列数 × 平均长度`。L12 说 GPT-3 175B 时代"per-request key-value cache takes 3GB GPU memory"，量级上对得上（那是 MHA、96 层、长度 2048 的配置）。

3.3.2 为什么 KV cache 让生成变成 I/O 密集型

L2 的推演链条非常干净：

```
KV cache 存在 HBM（不是 SRAM）
   → 每个 decode step 都要把整个 KV 从 HBM 读一遍
   → 生成 1 个 token 的时间 ≈ KV 字节数 / HBM 带宽
   → 序列越长、并发越多，读的量越大
   → decode 阶段是 memory-bound（与 2.5 节的 roofline 结论一致）
```

讲义列了内存层级三句话："GPU SRAM is small, and supports the fastest access"、"GPU HBM is larger but with much slower access"、"CPU DRAM is huge, but the slowest of all"。这就是第 14、15、18 章所有 KV 管理技术的物理基础。

3.4 FlashAttention：tiling + online softmax

L2 用几乎一半的篇幅讲 FlashAttention，因为它同时示范了"如何用 AI 分析找瓶颈"和"如何用系统手段消除瓶颈"。

3.4.1 朴素实现的三个问题

讲义的原文（含被图遮住的部分）：

- **Repeated reads/writes from GPU HBM**：$N \times N$ 的 score 矩阵要写回 HBM 再读回来做 softmax；
- **Large intermediate results**：$O(N^2)$ 的中间矩阵；
- **Hardly scale to long sequences due to $O(N^2)$ intermediate results**。

3.4.2 两个关键想法

想法	讲义原文	系统效果
Tiling	"compute the attention block by block so that we don't have to load everything into SRAM at once"	HBM 往返从 $O(N^2)$ 降到 $O(N^2/M)$ 级别（M 为 SRAM 块大小）
Operator fusion	"Use a single kernel that includes all operators during computation to avoid kernel launching overhead and intermediate data movement"	消除 kernel launch 与中间张量落地

tiling 的三步循环（讲义逐帧给出）：① 从 HBM 按块 load 到 SRAM；② 在片上计算该块的 attention 输出；③ 按块更新 HBM 里的输出（"Update output in HBM by scaling"）。

3.4.3 online softmax：为什么需要"运行时修正"

softmax 的数值问题讲义讲得很直白：$e^{1000} \approx 10^{434}$，FP32 装不下 → INF。标准解法是减最大值（stable softmax）：先求 $m(x) = \max_i x_i$，再算 $e^{x_i - m}$，最后除以 $\sum e^{x_j - m}$。

但在 tiling 下有个致命问题：

> **"But in Attention, we don't know m(x) in advance?"**

FlashAttention 的答案（讲义原文）：

> **Key Idea: When we find a new max in x, we correct the running sum by multiplying by $e^{old_max - new_max}$**

【补充】把它写成可实现的递推（对每个 query 行，逐块扫描 key 块 j）：

```
初始化:  m = -inf,  l = 0,  acc = 0        # m: running max, l: running sum, acc: 未归一化输出
对每个 key 块 j:
    s    = q @ K_j^T / sqrt(d)              # 块内 score
    m_new = max(m, rowmax(s))
    p     = exp(s - m_new)                  # 用新 max 稳定
    corr  = exp(m - m_new)                  # 修正因子
    l     = l * corr + rowsum(p)
    acc   = acc * corr + p @ V_j            # 关键：旧的 acc 也要乘修正因子
    m     = m_new
输出:  acc / l
```

这三行（`corr`、`l` 修正、`acc` 修正）是 FlashAttention 反向传播之外最容易写错的地方——**旧累加器必须整体乘 $e^{m_{old}-m_{new}}$，不只是分母**。

讲义最后留了一个开放问题："**Question: limitations of FlashAttention?**" 没有给答案。本笔记的答案（`【推断】`）：① 它省的是 HBM 往返，**FLOPs 一个没省**，所以对 compute-bound 的长序列场景帮助有限；② 反向需要重算或保存中间量，训练时仍有额外开销；③ 它假设 attention 是稠密的，稀疏/滑窗/检索式 attention 需要专门的 kernel；④ 分块尺寸与 SRAM 容量强耦合，换硬件要重调 tile。

3.5 MHA / MQA / GQA：同一笔账的三种记法

L3 用一页三栏图讲清了这条演进线（这张图在 L4 又复习了一次）：

变体	KV 头数	每 token 每层 KV 元素	质量	显存
MHA	$H_{kv} = H_q$，每个 head 独立吃 QKV	$2H_qD$	基线最好	最大
MQA	$H_{kv} = 1$，所有 query 头共享一组 KV	$2D$	可能下降	最小
GQA	$H_{kv} < H_q$，每组 $R = H_q/H_{kv}$ 个 query 头共享一组 KV	$2H_{kv}D$	接近 MHA	按 $1/R$ 缩减

讲义原文的评语是 "Better compute, smaller memory footprint, but quality may drop"——注意它把 compute 与 memory 分开说：**MQA/GQA 省的主要是显存与带宽，FLOPs 只省一点点**（attention 的 $QK^\top$ 计算量不变，省的是 K/V 的读取量与投影量）。

【补充】GQA 被 Llama 2 70B、Llama 3、Mistral 7B、Qwen2 采用（这句来自 HW2 的 1.2 节原文，不是我的推断）。工程上 GQA 还有一个隐性好处：**它把 KV cache 缩小了 R 倍，等价于把可并发请求数放大了 R 倍**，这对 serving 的吞吐是直接乘数。

一个具体账本（用 HW1 Q1–Q6 的口径算，见 3.7 节）：L=61 层、h=128 head、d_k=128 时，MHA 每 token 缓存 $2 \times 61 \times 128 \times 128 = 1{,}998{,}848$ 个标量；MQA 只要 $2 \times 61 \times 1 \times 128 = 15{,}616$ 个，**相差 128 倍**（正好等于 head 数）。

3.6 NSA：把稀疏性做成"可训练"的

L3 最后一节讲 DeepSeek 的 **Native Sparse Attention**（ACL'25 Best Paper），文字极少，只有三条：

- **Key insight**：我们可以先"摘要"长上下文，然后识别并聚焦关键词（"we can summarize long-context input, identify and focus on key words"）；
- 效果："Better quality, compute, and memory footprint"；
- 结论：Summarize（压缩块级摘要）+ focus on key tokens + neighboring tokens。

L4 的 panel discussion 页给了一句更技术化的补充（这是 panel 的讨论要点，不是论文正文）：

> **Key insight: leverage the block-wise context sparsity to reduce the computation** – If $q \times K$ is small, then skip computing that block – Reduce computation and I/O (skip loading V vector)

这句话点出了稀疏 attention 的系统本质：**跳过块不仅省 FLOPs，还省 V 的加载**——所以它是同时打"算力"和"带宽"两个瓶颈的少数手段之一。这也是第 15 章 H2O 的前置动机（"attention is sparse"）。

3.7 一份完整的参数/算力账本（HW1 Q5 的解法）

作业题是最好的账本练习。HW1 Q5 给定：$L = 64$ 层，$d_{model} = 8192$，$h = 32$ 头，$d_k = 256$，$V = 65536$，MLP 上投影 4 倍，权重 bf16。要求只算 embedding、Q/K/V/O 投影、MLP，不算最终输出投影、bias、LayerNorm、位置编码。

每层参数：

矩阵	形状	参数
W_Q	$8192 \times (32 \times 256)$	$8192 \times 8192 = 67{,}108{,}864$
W_K	8192×8192	$67{,}108{,}864$
W_V	8192×8192	$67{,}108{,}864$
W_O	8192×8192	$67{,}108{,}864$
MLP up	8192×32768	$268{,}435{,}456$
MLP down	32768×8192	$268{,}435{,}456$
每层合计		$805{,}306{,}368$

全局：

$$64 \times 805{,}306{,}368 = 51{,}539{,}607{,}552$$

$$+ \text{embedding } (65536 \times 8192 = 536{,}870{,}912)$$

$$= 52{,}076{,}478{,}464 \approx 52.08 \text{ B 参数}$$

显存：$52{,}076{,}478{,}464 \times 2 = 104{,}152{,}956{,}928$ B，除以 2^{30}：

$$\frac{104{,}152{,}956{,}928}{1{,}073{,}741{,}824} = 97.0 \text{ GiB}$$

注意这里有两个"陷阱"：① embedding 是 $V \times d_{model}$（不是 $V\times$ 别的），且题目明确要求计入；② 单位是 **GiB（2^{30}）不是 GB（10^9）**，差 7.4%，正好是很多人算错的地方。

【推断】这道题的参数结构（QKV+O 占 $4d^2$、MLP 占 $8d^2$）解释了为什么 `params ≈ 12 L d^2`：$4d^2 + 8d^2 = 12d^2$。这是所有"这个模型多大"心算的源头，也是第 10 章显存账本的起点。

第 4 章 分布式训练总览（L4）

4.1 先把问题写成数学

L4 开头复习了训练的目标函数（原文）：

$$\min \frac{1}{N}\sum_{i=1}^{N}\text{loss}\big(f(x_i),y_i\big)$$

常见损失：cross-entropy、MSE。求解手段：Adam 类自适应学习率优化器，或 SGD。更新式：

$$\Delta w=\eta\cdot\frac{1}{N}\sum_{i=1}^{N}\nabla\text{loss}\big(f_w(x_i),y_i\big),\qquad w\leftarrow w-\Delta w$$

到这里一切正常。问题是：**当我们把 batch 从 8 提到 8×P 时，单卡的 η 与吞吐同时撞墙**。讲义的原话："Increase the batch size increases throughput but eventually bounded by single GPU compute"。

4.2 为什么必须分布式：一张"装不下"的表

L4 给了一张模型规模演进表，这张表是全课程"显存焦虑"的起点：

指标	BERT-Large	GPT-2	Turing NLG 17.2B	GPT-3
Parameters	0.32B	1.5B	17.2B	175B
Layers	24	48	78	96
Hidden Dimension	1024	1600	4256	12288
Relative Computation	1×	4.7×	54×	547×
Memory Footprint	5.12 GB	24 GB	275 GB	2800 GB

然后接一句最关键的对照：

> NVIDIA V100 GPU memory capacity: **16G / 32G** NVIDIA A100 GPU memory capacity: **40G / 80G**

2800 GB vs 80 GB——这就是"为什么需要并行"的全部理由。注意这里的 memory footprint 是**训练**口径（含优化器状态与激活），不是推理。

4.3 两条历史路线：PS 与 AllReduce

L4 的标题页把答案先给了：

> Two Solutions – **(Centralized) Parameter Server** – **(Decentralized) AllReduce** – **Key assumption**: The model can fit into a (GPU) worker memory so we can create many model replicas

最后那句 Key assumption 是整个第 4–5 章的边界条件：**DP 类方案都要求"单卡装得下整个模型"**。一旦装不下，就必须切模型（第 6 章 TP、第 7 章 PP）。

4.3.1 Parameter Server 的编程模型

L4 把 PS 归结为四步循环：

```
1. Partition the training data
2. Parallel training on different machines
3. Synchronize the local updates
4. Refresh local model with new parameters, then go to 2
```

编程接口（讲义原文）：

角色	接口
Client (workers)	`Pull()`：a new iteration begins；`Compute()`；`Push()`：an iteration finishes
Server (aggregator)	`Update()`：aggregate model updates

讲义特意点了一句："Very similar to the spirit of **MapReduce**"，以及"Flexible and can be used in various ML algorithms, not just DL"。这句话解释了 PS 为什么是 2014–2017 年的主流：它不只服务 DL，还服务 LDA、矩阵分解、LR 等传统模型。

紧接着讲义抛出了问题："**How to ensure correctness?**"——这就是 4.4 节的一致性谱系。

4.3.2 AllReduce 的路线

L4 末尾给出另一条路：**每张卡都有完整模型 + 完整 batch 的一个分片，梯度用 AllReduce 求平均**。讲义在图上标了 "Fast connections: NVLink, NVSwitch"，并直接写出了更新式：

$$\Delta w = \eta \cdot \frac{1}{N} \sum_{i=1}^{N} \nabla \text{loss}(f_w(x_i), y_i)$$

【补充】这就是今天 PyTorch DDP 的原型：`loss.backward()` 之后，所有 rank 的梯度做一次 SUM（再除以 world_size，或先除后加），然后各自 `opt.step()`。它与 PS 的本质差别是：**没有中心瓶颈，但通信是"全对全"的，且对最慢的那个 rank 极度敏感**。

4.4 一致性谱系：BSP → 异步 → 有界一致性

L4 用四页图讲了这条谱系，每页的标题就是结论：

模式	讲义标题	机制	后果
BSP（Bulk Synchronous Parallel）	"Bulk Synchronous Parallel: Consistency"	每轮所有 worker 同步一次（barrier），再进下一轮	语义等价于大 batch SGD；**被最慢者拖住**
同步的敌人	"Stragglers"	某台机器慢（GC、网络抖动、异构硬件）	整轮等待，利用率塌陷
异步	"Asynchronous Communication (No Consistency)"	谁先算完谁就 push/pull，不等	无等待，但读到的是**陈旧参数**
有界一致性	"Bounded Consistency"	限制 staleness 上界（如 $s \leq 2$）	用可控的统计效率换系统效率
权衡	"Impacts of Consistency/Staleness: Unbounded Staleness"	—	讲义原话：**"Too small → poor sys efficiency; Too large → poor stats efficiency"**

这段是 CS 498 独有的内容（M2/M3 笔记里没有）。它的价值在于给出一条判断标准：**任何"异步训练"论文，本质都在回答"我用多大的 staleness 换多少系统效率，以及这个交换在统计上是否还收敛"**。

【补充】几个经典结论供对照：异步 SGD（Hogwild 风格）在稀疏问题上收敛性有保证，但在稠密深度网络上 stale 梯度会显著劣化；Stale Synchronous Parallel（SSP）给出的是"最快与最慢的时钟差不超过 s"的保证；今天的大模型预训练**几乎全部回到 BSP**（因为要复现、要对齐 loss 曲线），异步主要出现在 RL/推荐/联邦学习场景。L5 结尾的图也印证了这一点——它把 "Federated Learning" 标注为 "~ PS-based Data Parallelism"。

4.5 集合通信的词汇表

L5 用三页图建立术语（讲义文字极少，以下定义属 `【补充】`，但术语本身来自讲义标题）：

原语	语义	在训练中的用途
Broadcast	1 → 全部	分发初始权重、分发超参
Scatter	1 → 各一份分片	分发不同 batch 分片
Gather	各一份分片 → 1	收集指标/评估结果
Reduce	各一份 → 1（做 op）	汇总 loss/统计量
AllReduce	各一份 → 全部得到归约结果	**DP 的梯度同步**
ReduceScatter	归约后各留一片	ZeRO/FSDP 的梯度分片
AllGather	各一片 → 全部拼齐	参数/激活的收集
All-to-All	每对 rank 互换分片	**MoE 的 token 分发**（第 17 章）

L5 给了四条关于 collectives 的判断，条条都是工程经验：

1. "Collectives are implemented using many P2P"——集合通信不是魔法，是 P2P 的组合；
2. "Collectives are much more expensive than P2P"；
3. "Collectives are highly optimized throughout the past 20 years"（找 `X`CCL 库：NCCL、RCCL、MSCCL…）；
4. "**Collectives are not fault-tolerant**"——一个 rank 挂掉，整个 job 挂掉。

第 4 条是"为什么大集群训练要靠 checkpoint 而不是靠容错通信"的答案。

4.6 Ring AllReduce 的复杂度：2N(P−1)/P

L5 把 ring AllReduce 拆成两阶段逐帧讲解，并留下三个课堂提问，答案都在页面上：

阶段	做什么	传输量	讲义给的答案
Phase 1: **Reduce-Scatter**	数据切 P 块，绕环 P−1 步，边传边加	每 rank 发 N/P 块 × (P−1) 次	$O\left(\frac{N}{P}(P-1)\right)$
Phase 2: **AllGather**	只搬运不再归约，绕环 P−1 步	同上	$O\left(\frac{N}{P}(P-1)\right)$
合计		$\frac{2N(P-1)}{P}$	"largely independent of P"

最后那句 "largely independent of P" 是 ring 的精髓：当 P 很大时 $\frac{P-1}{P} \to 1$，每个 rank 的通信量趋近 $2N$——**与 P 无关**。这就是 ring 能扩展到上千卡的原因，也是为什么它打败了"中心 reduce"（后者每个 rank 要发 N、中心要收 PN）。

L5 还对比了两种实现范式（"Two AllReduce Paradigms"）：**Ring-based** 与 **Tree-based**。L5 用四页讲 "Some Transformations"，标注了三个参数：**α = round-trip time latency（往返延迟）、β = network bandwidth（带宽）、γ = compute overhead（计算开销）**，并引了一篇 ASPLOS'24 最佳论文 *Centauri: Enabling Efficient Scheduling for Communication–Computation Overlap in Large Model Training via Communication Partitioning*。

【补充】这套 α/β/γ 记号是 Hockney 模型的通信时间公式：$T = \alpha + \frac{N}{\beta}$。它解释了两件事：① 小消息（N 小）时延迟 α 主导 → 应该**合并小消息**（gradient fusion / bucket）；② 大消息时带宽 β 主导 → 应该**切分大消息**以便与计算 overlap（Centauri 与 ByteScheduler 的思路）。L5 最后一页 "Further Comm. Optimizations for Model Parallelism (ByteScheduler)" 的结论正是这句话：**"Partitioning and reordering communication allows better overlap"**。

4.7 NCCL：把 ring 做进 kernel

L5 给了 NCCL 的三条特征（原文）：

- Use unified GPU direct memory accessing（GPU Direct）；

- **Each GPU launches a working kernel, cooperate with each other to do ring-based reduction**（每张卡发一个 kernel，互相协作做 ring 归约）；
- A common backend for popular DL frameworks such as PyTorch。

第三条是关键：PyTorch 的 `gloo`（CPU）/ `nccl`（GPU）后端就是这套东西。而**本课程的三个作业全部用 CPU + gloo**，正是为了让"手写 ring all-reduce"变得可运行、可调试（见速查文档）。

第 5 章 数据并行（L5）

5.1 DP 的两种实现与它们的共同假设

L5 开头复习了 L4 的两条路线，并再次强调 Key assumption：**模型能装进一个 worker 的显存**。在这个假设下，DP 的两种实现：

```
Option 1: Parameter Server
   worker_i --grad--> server --param--> worker_i        （星形，中心瓶颈）

Option 2: AllReduce
   worker_i <--allreduce(sum/gather)--> worker_j         （环形/树形，无中心）
```

L5 在第 7 页图上写了一句话压住整节课的重点："Optimizing communication efficiency is the key"。

5.2 通信量分析：为什么 DP 的通信"看起来免费"

设模型参数量为 Ψ（个元素），每元素 b 字节（BF16 = 2）。一个训练 step 里：

量	表达式	GPT-3 175B（$\Psi = 175 \times 10^9$, bf16）
梯度总量	Ψb	350 GB
Ring AllReduce 每 rank 发送量	$2\frac{P-1}{P}\Psi b \approx 2\Psi b$	700 GB
单 step 计算量（约）	$6\Psi\times$ tokens	与 tokens 成正比

关键比值是**通信/计算比**：

$$\frac{\text{comm per rank}}{\text{compute per rank}} = \frac{2\Psi b}{6\Psi \cdot m \cdot t_{\text{seq}}} = \frac{b}{3mt_{\text{seq}}}$$

其中 m 是每卡 micro-batch 大小，t_{seq} 是序列长度。

【推断】这条式子的三个推论，恰好解释了 L8 的实测结论：

1. m **越大，通信/计算比越小** → 大 batch 时 DP 几乎免费，这就是 L8 "Why does larger batch size scale better?" 的答案；
2. **序列越长，通信越不划算**（相对计算而言通信占比下降，但 KV/激活显存上升）；
3. Ψ **越大，单步通信绝对值越大**，但比值不变 → 所以 DP 的扩展效率与模型大小基本无关，只与 batch 有关。

5.3 梯度累积：用时间换 batch

L5 的 "How to run large training batch size?" 一问，标准答案是梯度累积：

```
accum_steps = 4          # 累积 4 个 micro-batch 才同步一次梯度
opt.zero_grad(set_to_none=True)
for i, batch in enumerate(loader):
    loss = model(batch).loss / accum_steps      # 注意：要除以累积步数
    loss.backward()                             # 梯度在 p.grad 上累加
    if (i + 1) % accum_steps == 0:
        allreduce_grads(model)                  # 一次集合通信
        opt.step()
        opt.zero_grad(set_to_none=True)
```

三个容易踩的点（`【补充】`，但在 HW1 的 `run.py` 里能直接看到对应写法）：

1. **loss 必须除以累积步数**，否则等价于把学习率放大了 accum_steps 倍；
2. **`zero_grad(set_to_none=True)` 比 `zero_grad()` 省一次写全零**（PyTorch 会把 `.grad` 置为 `None`，反向时再分配）；
3. **BN/BatchNorm 在累积下语义会变**（每个 micro-batch 单独统计），LLM 用 RMSNorm/LayerNorm 所以不受影响——这也是 LLM 训练能自由梯度累积的原因之一。

5.4 L5 结尾的两条"延伸"

L5 的总结页写了两句，都很重要：

> - AllReduce-based Data Parallelism
> - Workers exchange updates in a ring/tree topology
> - Communication traffic: $2N/P \cdot (P-1)$ (N: data size; P: # of workers)
> - **Widely-used (e.g., NCCL) yet sensitive to stragglers**
> - Federated Learning ≈ **PS-based Data Parallelism**

第二句把联邦学习定位成"PS 架构 + 隐私约束"，这是很好的类比：**FedAvg 的 server 就是 PS 的 aggregator，client 就是 worker**，区别只在于数据不出本地、参与方高度异构。这也解释了为什么 FL 里

staleness 与 straggler 问题比数据中心更严重。

5.5 ZeRO 的思路（从 L10 的 panel 页反推）

ZeRO 在 README 里是 L10 的必读论文（"ZeRO-style Data Parallelism"，arXiv 1910.02054），而 L10 的 panel 页给了一句话总结：

> **ZeRO partitions optimizer states and model weights across machines to reduce peak memory demands at the cost of higher communication.**

【补充】把这句话展开成三段（这是理解 ZeRO 的最短路径）：

阶段	切什么	每卡显存（相对 16Ψ）	每 step 额外通信
基线 DP	不切	$\mathbf{16\Psi}$	$\mathbf{2\Psi}$（一次 AllReduce）
ZeRO-1	优化器状态（fp32 的 m、v）	$\approx \mathbf{4\Psi} + \mathbf{12\Psi}/P$	同基线
ZeRO-2	+ 梯度	$\approx \mathbf{2\Psi} + \mathbf{14\Psi}/P$	ReduceScatter + AllGather
ZeRO-3	+ 参数	$\approx \mathbf{16\Psi}/P$	每层 AllGather（前向+反向）

那 16 的来历：bf16 参数 2 + bf16 梯度 2 + fp32 master 权重 4 + fp32 Adam m 4 + fp32 Adam v 4 = 16 bytes/param。这套账本在 M2 笔记里有完整推导，此处只做索引——**CS 498 不推导它，只在 panel 里给结论**，这是两门材料分工的典型例子。

第 6 章 张量并行（L6）

> **讲义文字警告**：L6 的 `pdftotext` 输出只有 11.8 KB，几乎是纯图。本章的公式部分（分块矩阵、通信插入点、复杂度）以 `【补充】` 标注，来源是讲义标题、L6/L7/L8 的图注、README 指向的 Megatron-LM 论文，以及 HW1 Q6 / HW2 1.3 的题面（这两处题面本身就是课程给出的权威口径）。

6.1 为什么 DP 之后还要 TP

L6 开头两页把逻辑链摆得很清楚：

- "Recall: Data Parallelism —— Partitions a training minibatch across multiple devices，**Linearly scalable**"
- 紧接着一页并列 PS 与 AllReduce 两张图，然后一句判决：**"Data parallelism does not help with excessive model size"**。

下一节标题是 "Large Models Require Large Memory"，列了五条"为什么训练比推理更吃显存"：

因素	方向
Training > Inference	训练要存激活与梯度
Adam > SGD（due to optimizer state）	每个参数多 8 字节（fp32 的 m 与 v）
Larger minibatch	激活随 batch 线性涨
More parameters	权重与优化器状态
…	—

然后是那句课堂提问："**Question: How do we enable or even speed up training?**"

6.2 层间 vs 层内：两个正交的切法

L6 用两页图区分了两种 model parallelism（这段文字在 L6 和 L7 里都出现了，是课程的标准定义）：

类型	讲义定义	切的对象	通信特征
Inter-layer (Pipeline) parallelism	"Split **sets of layers** across multiple workers；Layer 0,1,2 和 Layer 3,4,5 在不同设备"	深度方向	只在 stage 边界传激活，**低频**
Intra-layer (Tensor) parallelism	"Split **individual layers** across multiple devices；两个设备都计算 layer 0..5 的不同部分"	宽度方向	每层内部多次集合通信，**高频**

L6 有一页标题就叫 "Complementary Types of Model Parallelism"（Inter + Intra Layer Parallelism），说明这两者是**互补**而非替代——这正是第 8 章 3D 并行的起点。

6.3 矩阵切分的两种方式：列切与行切

L6 用 8 页（第 10–17 页）逐帧演示 GEMM 的切分，但每页只留一个标题词："Row-wise sharding (of A)"/"Column-wise sharding (of A)"。第 17 页给了唯一一句判断：

> **"Column- or row-wise highly depends on tensor operations"**

【补充】把这句判断展开，就是 Megatron 的两条铁律。**先约定 PyTorch 的 `nn.Linear` 权重形状为** $[out, in]$，前向是 $Y = XW^\top$（HW2 与 HW1 Q6 都用 $Y = XW_1$ 的写法，即 W_1 形状 $[H, F]$，此时前向是 XW_1，注意区分）。

（a）列并行（Column Parallel）：把 W 的**输出维**切开。

$$W = [W_1 \mid W_2], \quad Y = XW^\top = [XW_1^\top \mid XW_2^\top]$$

- 每卡输入 $\boldsymbol{X}$ 完整，输出是 $\boldsymbol{Y}$ 的一个列块；
- **前向不需要通信**（各卡独立算出一段输出）；
- 反向：$\partial L/\partial X = GW$，每卡只有自己那一段 $\boldsymbol{W}$，所以**需要对 dX 做 AllReduce（求和）**。

（b）行并行（Row Parallel）：把 $\boldsymbol{W}$ 的**输入维**切开。

$$W = \begin{bmatrix} W_1 \\ W_2 \end{bmatrix}, \quad Y = XW^\top = X_1 W_1^\top + X_2 W_2^\top$$

- 每卡输入是 $\boldsymbol{X}$ 的一个列块，输出是**部分和**；
- **前向需要 AllReduce** 才能得到完整 $\boldsymbol{Y}$；
- 反向：$dW_i = G^\top X_i$ 本地可算，$dX_i = GW_i$ 本地可算；但要把 dX 的列块拼起来才能给上一层（其实上一层也是切好的，见下）。

6.4 Transformer 的 TP 方案：一个 AllReduce 都不多

Megatron 的核心贡献是**把两种切法串起来，使前向与反向各只需要一次 AllReduce**。L6 的 MLP 四页图把这个过程画成了：先切第一层 → "Requires AllReduce before GeLU" → "Can run in parallel now!"。

【补充】正确的串法（也是 HW1 Q6 的考点）：

```
输入 X（完整，[B,S,H]）
  │
  ├─ W1 按【列】切 → 每卡 [H, F/P]，本地 Z_r = X W1_r   （无通信）
  │    注意：GeLU 是逐元素非线性，可以在本地做 → A_r = GeLU(Z_r)
  │
  ├─ W2 按【行】切 → 每卡 [F/P, H]，本地 Y_r = A_r W2_r   （部分和，无通信）
  │
  └─ AllReduce(SUM) over Y_r  →  Y（完整，[B,S,H]）    ← 前向唯一一次通信
```

这里有两个必须记住的"为什么"：

1. **GeLU 必须在 AllReduce 之前、在本地做**。因为 GeLU 非线性，$\mathbf{GeLU}(Z_1 + Z_2) \neq \mathbf{GeLU}(Z_1) + \mathbf{GeLU}(Z_2)$。HW1 Q6(b) 就是问这个：如果 $\boldsymbol{W_1}$ 按行切、各卡算部分积，**必须先求和再 GeLU**。
2. **$\boldsymbol{W_2}$ 必须按行切**，才能让"部分积求和"对应到完整的 $\boldsymbol{Y}$。若 $\boldsymbol{W_2}$ 也按列切，就需要两次 AllReduce。

Attention 的切法同理（L6 第 23–25 页三帧图）：

```
QKV 投影：按 head 切（列并行，每卡拿自己那几个 head 的 Q/K/V）   → 无通信
Attention：每卡对自己的 head 做 attention（head 之间本来就独立）   → 无通信
输出投影 W_O：按【行】切（因为输入维就是 head 维）                → AllReduce(SUM)
```

一个 head 内不能切，因为 softmax 需要整行 score。这就是 HW2 TP-1 的答案：每卡持有 $\frac{H_q}{P}$ 个 query head 与 $\frac{H_{kv}}{P}$ 个 KV head，**投影与 attention 都不需要通信，只有 $\boldsymbol{W_O}$ 之后需要一次 all-reduce**。L6 也留

了同一类提问："**Question: What if we have more GPUs than the number of heads?**"（答案见 6.6 节）

6.5 通信次数：1 个 forward + backward 共 4 次

L6 第 26 页只有一行字：

> **"4 total communication operations in 1 forward + backward pass"**

【补充】这四次的来历（每层）：

时机	操作	张量	作用
Forward	AllReduce（MLP）	$[B, S, H]$	合并 W_2 的部分和
Forward	AllReduce（Attention）	$[B, S, H]$	合并 W_O 的部分和
Backward	AllReduce（Attention，对 dX）	$[B, S, H]$	列并行层的输入梯度
Backward	AllReduce（MLP，对 dX）	$[B, S, H]$	列并行层的输入梯度

关键观察：通信量与 H（隐藏维）成正比，与切分数 P 无关。这是 TP 的致命弱点：P 越大，每卡的计算量变成 $1/P$，但通信量不变 → **通信/计算比随 P 线性恶化**。这就是 L6 总结页那句 "Reduce synchronization cost is crucial as TP is communication-intensive" 的定量原因。

L6 第 33–34 页标题是 "Complexity Analysis of Tensor Parallelism"，标注 "(P: # of workers)"，但公式被图吃掉了。【补充】标准结论是：TP 的通信时间 $\propto \frac{2(P-1)}{P} \cdot \frac{BSHb}{\beta}$，计算时间 $\propto \frac{1}{P} \cdot (\text{每层 FLOPs})$，所以加速比上界受 $\frac{P}{1+c \cdot P}$ 形式约束——**存在最优 P，超过就负收益**。

6.6 一个完整的 TP 通信量算例（HW2 1.3 的解法）

HW2 1.3 给的是 Llama 3 8B 的 attention 尺寸，要求算 GQA 层的 TP 通信量。题面参数：$H_q = 32$、$H_{kv} = 8$（$R = 4$）、$D = 128$、$C = H_q D = 4096$、$P = 4$、BF16（2 B/elem）、ring all-reduce（每阶段 $P - 1$ 步）。

（1）TP-1 头分配：每卡 $32/4 = 8$ 个 query head、$8/4 = 2$ 个 KV head。因为 head 之间独立，**Q/K/V 投影与 attention 都不需要通信**；只有 W_O（按行切）之后需要一次 all-reduce。

（2）TP-2 Prefill：$B = 8$ 请求 × $L = 512$ prompt = 4096 个 token 一次性前向。

- 每卡部分输出形状：$[4096, 4096]$（token 数 × C），BF16 大小 $= 4096 \times 4096 \times 2 = 33{,}554{,}432$ B $= 32$ MiB。
- Ring all-reduce 每卡发送量 $= 2 \cdot \frac{P-1}{P} \cdot N = 2 \times \frac{3}{4} \times 32 = 48$ MiB。
- 四卡总量 $= 4 \times 48 = 192$ MiB。

（3）TP-3 Decode：8 个请求各有 2048 缓存 token，本步每请求只处理 1 个新 token → 8 个 token。

- 每卡部分输出：$[8, 4096]$，$8 \times 4096 \times 2 = 65{,}536$ B $= 64$ KiB。
- 每卡发送量 $= 1.5 \times 64 = 96$ KiB；四卡合计 384 KiB。
- **2048 token 的缓存上下文不改变这个量**：all-reduce 只作用于"本步新算出的激活"，历史 KV 留在各卡本地、不参与集合通信。
- **GQA 是否把通信量降到 MHA 的 1/4？不。** all-reduce 的张量是 W_O 的输出（$[N, C]$），而 $C = H_q D$ 与 H_{kv} 无关。GQA 省的是 **KV cache 显存与 KV 读取带宽**（缩小 $R = 4$ 倍），**不省 TP 的激活通信**。这是本题最大的"陷阱"，也是理解"GQA 优化什么、不优化什么"的关键。

（4）prefill 与 decode 的通信量对比：$192\ \text{MiB} / 384\ \text{KiB} = 512$ 倍，正好等于 token 数之比 $4096/8$。**结论：TP 的通信成本几乎全部记在 prefill（以及训练）头上，decode 阶段 TP 的通信相对便宜**——这解释了 PD 解耦（DistServe）为什么要按阶段配不同的并行度。

6.7 Megatron 的实验结论

L6 最后两页标题是 "Experiments: Scalability"，图上标了 "Weak-scaling: the effective execution time"。【补充】Megatron-LM（2019）与它的续作（SC'21）给出的结论可以概括成三条，第 8 章会用它们做定量对照：

1. **8 卡以内 TP 的扩展效率接近线性**（P100 上 8 卡到 74% 的模型 FLOPs 利用率）；
2. **跨节点 TP 迅速崩坏**（因为每次 AllReduce 都要穿 IB）；
3. **TP 与 PP 组合后最优配置倾向于 $t = p = 8$**（见 8.4 节的实测解释）。

第 7 章 流水线并行（L7）

7.1 起点：TP 是通信密集的

L7 的第一页标题就是 "Recall: Tensor Parallelism is Communication-heavy"，配图是 Intra-layer 的定义与行切/列切两栏。这句话是 PP 的存在理由：**既然层内切分通信太贵，那就改成层间切分——只在 stage 边界传一次激活**。

7.2 朴素层间并行：串行执行，全程 idle

L7 用 8 页画了一个"最笨"的 PP：每个设备负责几层，一个 mini-batch 从头走到尾，然后反向走回来。图上的标注有两处关键信息：

- "**DNN training involves a bi-directional execution**：前向从输入层开始，反向在输入层结束"；
- "**BW is often two times longer than FW**"（反向通常是前向的两倍耗时）。

结论页写得很直接：

> Issues with Simple Inter-Layer Parallelism – Under-utilization of compute resources – Only one device is computing at a time and others are idling

图上有三个 "Idle" 标注——这就是后来说的 **bubble（气泡）**。

7.3 Pipeline Model Parallelism：micro-batch 是关键

L7 给的解法是经典的 GPipe 思路，四句话（原文）：

概念	讲义定义
Mini-batch	the number of samples processed in each iteration
切分	"Divide a mini-batch into multiple smaller **micro-batches**（e.g., 128 = 4×32）"
Pipeline execution	"Micro-batches flow through the pipeline from one stage to the next."
交替	"As soon as a stage completes its work, it passes the micro-batch to the next stage and **starts working on the next micro-batch**"

于是时间轴变成一条斜线：stage 0 先算 mb0，算完立刻传给 stage 1，自己接着算 mb1；stage 1 收到 mb0 立刻算……**稳态时所有 stage 同时在忙**，只有开头（fill）与结尾（drain）有空闲。

7.4 气泡的定量：把 L7 的图变成公式

L7 第 21–25 页用同一组参数逐层叠加时间轴，参数写在页面上：**m = #microbatches (8)，p = pipeline stages (4)，tf = time of forward，tb = time of backward**。三段时间的标注依次出现：

$$m \cdot t_f \qquad (p-1) \cdot (t_f + t_b) \qquad m \cdot t_b$$

并配了一句解释："**Each device needs to wait for (p−1) devices to finish their forward and backward passes**"。

【补充】把这三段拼起来就是总步时与气泡率（这是 L7 图的直接代数化，不是外部资料）：

$$T_{\text{step}} = \underbrace{mt_f}_{\text{fill}} + \underbrace{(p-1)(t_f + t_b)}_{\text{bubble}} + \underbrace{mt_b}_{\text{drain}}$$

$$\text{Bubble fraction} = \frac{(p-1)(t_f + t_b)}{mt_f + (p-1)(t_f + t_b) + mt_b}$$

取 $t_f = t_b = t$（讲义说反向常是两倍，这里先取对称做估算）：

$$\text{Bubble} = \frac{2(p-1)t}{2mt + 2(p-1)t} = \frac{p-1}{m+p-1}$$

这个式子给出三条工程直觉：

1. **m 越大，气泡越小**（m=8、p=4 时 =3/11≈27%；m=32 时 =3/35≈8.6%）；
2. **p 越大，气泡越大**（p=64、m=32 时 =63/95≈66%）→ **PP 不能无限加深**；
3. 但讲义同时给了 caveat：

> **Caveat: Increasing m reduces the micro-batch size, leaving insufficient work to saturate GPUs; moreover, m is capped by the mini-batch size.**

也就是说 m 不能随便加：m 大 → 每个 micro-batch 变小 → 单次 GEMM 的 M 维变小 → 算术强度下降（回到 2.5 节）→ GPU 反而跑不满。**这是一个双重约束：气泡要求 m 大，效率要求 m 小。**

7.5 1F1B：把峰值显存压下来

L7 第 29 页把问题从"效率"转向"显存"，标题是 "Improving Memory Efficiency for Pipeline Parallelism"，提问："**Can we improve the pipeline schedule to reduce memory requirements?**"

答案是 **1F1B**（One Forward One Backward），来源标注为 PipeDream。讲义用的三帧图给出三个状态：

状态	内容
Warmup	先连续做若干 forward，填满流水线
Steady state	每做一个 forward 就做一个 backward（1F1B），保持流水线满载
（Cooldown）	收尾阶段连续做 backward

【补充】1F1B 与 GPipe 的差别，一句话：**GPipe 是"所有 forward 全做完再所有 backward"**（激活要一直留着，峰值显存 $\propto m$），**1F1B 是"边进边出"**（每个 stage 同时持有的 in-flight micro-batch 数被限制在 p 个以内，峰值显存 $\propto p$ 而非 m）。这就是 L8 开头复习的那句 "1 Forward 1 Backward (1F1B)"。

代价是：1F1B 的调度实现复杂得多（要保证 backward 的依赖满足），且**它并不减小气泡**（气泡仍是 $(p-1)$ 个 stage 的填/排空）。**GPipe 优化吞吐，1F1B 优化显存**，这是两条正交的收益。

7.6 Interleaved 1F1B：用通信换气泡

L7 最后四页讲 **Interleaved 1F1B**（来源标注 Megatron-LM, SC'21），并给了具体例子：

```
原始：Layers [1–4] [5–8] [9–12] [13–16]      （4 个 stage，每 stage 4 层）
交错：GPU0: [1,2],[9,10]   GPU1: [3,4],[11,12]
      GPU2: [5,6],[13,14]  GPU3: [7,8],[15,16]
```

图注是："**Each device is assigned two chunks (subset of noncontiguous layers)**"，深色是第一块、浅色是第二块。文字规则三条（原文）：

1. "**Further divide each stage into v sub-stages**"
2. "**The forward (backward) time of each sub-stage is t/v**"
3. "**Reduce bubble time (increasing v) at the cost of increased communication**"

【补充】第三条的"代价"很具体：本来一个 micro-batch 在一台设备上连续过 4 层、只做 1 次 stage 间发送；交错后要过 2 个 chunk，发送次数翻倍（v 倍）。所以交错 1F1B 的最优 v 由"气泡节省"与"通信增量"的平衡点决定，Megatron 的实测建议是 v 取小值（2 左右），且要求 micro-batch 数 m 是 v 的整数倍。气泡率的通用形式可写作 $\frac{p-1}{mv+p-1}$【补充，系数以 Megatron SC'21 原文为准】。

7.7 L7 的三条总结

讲义总结页把三种并行放在一句话里对照（这三句是第 8 章的引子）：

> - Data parallelism (DP) **can not host a large model within a worker**
> - Tensor parallelism (TP) is often **communication-intensive**
> - Pipeline parallelism (PP) introduces **execution bubbles**

三种方案，三种病：DP 装不下、TP 通信贵、PP 有空泡。第 8 章的 3D 并行不是"三个都用"，而是"**用 PP 解决装不下、用 TP 换掉一部分 PP 的气泡、用 DP 把剩下的卡都用上**"。

第 8 章 多维并行（L8）

8.1 3D 并行的拼装方式

L8 第 5–6 页给出 "Combining Model Parallelism" 与 "3D Parallelism" 两张图，文字标注只有两项："**Tensor-slicing parallelism**" 与 "**Pipeline parallelism**"，再叠加 DP。加上讲义抛的课堂问题：

> **Question: How do we know which parallelism to choose?**

【补充】标准拼法（Megatron 的 rank 编排，也是第 9 章"通信域"的物理基础）：

```
总卡数 n = t × p × d
  t = tensor-parallel size（节点内，NVLink）
  p = pipeline-parallel size（跨节点，IB）
  d = data-parallel size（剩余的卡，全部用于复制）

rank 编号（Megatron 习惯，t 最快变化）：
  rank = ((d_idx * p + p_idx) * t + t_idx)
  → 同一个 TP 组内的 t 张卡编号连续 → 落在同一节点内 → 走 NVLink
  → 同一个 PP 组内的 p 张卡编号跨 t 步 → 落在不同节点 → 走 IB
```

这个编号顺序不是小事：**它决定了哪种通信走哪张网**。如果 TP 组跨了节点，性能会立刻崩掉（见 8.4 节）。

8.2 组合并行的性能模型

L8 第 7–18 页连续 12 页都在推同一个模型，页面上留下的碎片很有价值：

页面标注	含义
"(Bubble Fraction)"	要算的核心指标
"Assume d = 1, n = p * t"	先看不用 DP 的纯模型并行情形
"m = B / (d * b) = b' / d"	micro-batch 数 = 全局 batch / (DP 度 × 每卡 micro-batch)
"B / (d * b)"、"B / b"	两种分母写法
"n: # of GPUs, b': ~ # of micro-batches"	符号定义

【补充】这套记号对应 Megatron SC'21 的 bubble 公式：

$$\text{Bubble fraction} = \frac{p-1}{m} \quad (\text{当 } m \gg p), \qquad m = \frac{B}{d \cdot b}$$

代入 $m = B/(db)$：

$$\text{Bubble} \approx \frac{(p-1)\,d\,b}{B}$$

这条式子的工程含义极强：**气泡率与 p、d、b 成正比，与全局 batch B 成反比**。所以"想用大 p 就必须用大 B"，而大 B 又受收敛性约束（L8 第 22 页的提问 "Why does larger batch size scale better?" 正是这个逻辑的另一面）。

8.3 TP vs PP 的实测：为什么峰值在 t = p = 8

L8 第 12–13 页给了一张 Megatron 的实测图，图注信息量很大：

> 8 Nvidia 80GB A100 cards per node，8 nodes are connected through fat-tree topology

以及页面上的两句诊断：

区域	讲义解释
$t > 8$	expensive cross-server allreduce
$t < 8$	bubble fraction increase

这就是"最优并行度由**拓扑**决定"的最干净例证：

- 一个节点 8 卡、走 NVLink → $t \leq 8$ 时 all-reduce 便宜；
- 一旦 $t = 16$，TP 组跨节点 → 每层两次 all-reduce 都要穿 IB → 通信爆炸；
- 但 t 太小，模型切得不够细，就需要更大的 p 才能装下 → 气泡变大。

最优解刚好落在"节点边界"上：$t = 8$（吃满 NVLink）、$p = 8$（8 个节点）、d 用来扩展更多卡。

8.4 DP 与模型并行的对照

L8 第 19–23 页做了一组"DP vs Model Parallelism"的对照，页面文字留下两条关键诊断：

> **Limitations of tensor-parallelism:** 1. **More frequent Allreduce** 2. **Allreduce is on critical path**

"在关键路径上"这五个字非常重要：TP 的 AllReduce **不能被后续计算遮盖**（因为下一层要用它的结果），所以它是**纯延迟**，不是可以被 overlap 掉的带宽成本。而 DP 的梯度 AllReduce 发生在**反向的最后**，可以和下一轮的前向 overlap（这就是 PyTorch DDP 的 `no_sync` 与 bucket 机制、以及 ByteScheduler 的思路）。

L8 第 24 页的总结页给了 DP 两侧的病症对照（原文）：

方案	病灶
Parameter Server	**Overwhelm the centralized server**（压垮中心）
AllReduce	**Amortize aggregation but face stragglers**（聚合被摊薄，但怕掉队者）

8.5 经验法则与它背后的三个约束

L8 总结页的三条 rule of thumb（4.4 节已引），这里给出它们的定量依据：

规则	依据
TP 在节点内，PP 跨节点	$t > 8$ 时 TP 的 all-reduce 穿 IB，延迟进关键路径（8.3 节实测）
TP/PP 的 size 由显存决定	每卡权重 = $\Psi b/(tp)$，要装下"权重+梯度+优化器状态+激活"；给定模型与卡容量，tp 的最小值被锁死

规则	依据
DP 用来扩到更多卡	DP 通信/计算比 $\propto 1/(mt_{seq})$，只要 batch 够大就近似免费（5.2 节）

【推断】这三条其实是一条：**先按显存定 $t \cdot p$，再按拓扑定 t 的上界，剩下的全给 d**。这就是为什么今天所有大模型训练的配置看起来都像 "TP=8, PP=4~16, DP=其余"。

8.6 Alpa：把"选并行策略"变成优化问题

L8 最后一页是 Alpa 的 panel 讨论预告，三句话概括了 Alpa 的方法：

> - **Inter-operator design**: dynamic programming to minimize end-to-end training latency（算子间：用动态规划做切分与流水线划分）
> - **Intra-operator design**: linear programming to decide tensor-GPU mappings（算子内：用线性规划决定张量怎么落到 GPU 上）
> - **Assumes homogeneous GPUs!**（假设同构 GPU）

【补充】Alpa 的两级抽象值得记住，因为它把"自动并行"这个模糊问题拆成了两个可解的子问题：

1. **Inter-operator**：把计算图切成 stage、决定每个 stage 放哪些算子、用什么并行度 → **DP（动态规划）**，目标是最小化端到端延迟；
2. **Intra-operator**：给定一个算子（如 attention），决定它的 sharding spec（怎么切）与 device mesh（落到哪些卡）→ **ILP（整数线性规划）**；
3. 两级之间用"cost model + 通信代价"耦合。

最后那句 "Assumes homogeneous GPUs!" 是 Alpa 的最大限制，也是后续工作（异构集群上的自动并行）的入口。**课程把它放在 panel 讨论而不是正文，说明重点是"理解问题结构"，不是"背算法"**。

第 9 章 混合精度训练（L9）

9.1 硬件前提：Tensor Core

L9 用两页讲 Tensor Core（原文）：

> Tensor cores are: – Special hardware execution units – Execute matrix multiply operations in parallel – **Fused multiply-add on small matrices in one instruction cycle** – Built to accelerate deep learning
>
> Different flavors: – (V100) Volta Tensor Cores **FP16** – (T4) Turing Tensor Cores **FP16/INT8/INT4**

关键数字出现在 "Mixed Precision Training Benefits" 页：

收益	讲义原文
加速	"**Tensor cores are 8x faster than FP32**"
省带宽	"FP16 halves memory traffic compared to FP32"
省显存	"FP16 halves the size of activation and gradient tensors"、"Enables larger models, mini batches or inputs"

9.2 为什么不能"纯 FP16"

L9 用一页专门回答 "Why Not Use Pure FP16?"，三条理由：

问题	讲义表述
动态范围窄	"FP16 has a narrower dynamic range than FP32；May cause **underflow/overflow** issues"
权重更新被吞	"Optimizer takes very small increments when search narrows into a solution；**Late updates often cannot be represented in FP16**"
归约误差	"Large sums of values … can be too big for FP16；**Adding small values to a large sum can lead to rounding errors**"

第二条是本质：当 $w \gg \alpha\nabla w$ 时，$w - \alpha\nabla w$ 在 FP16 的尾数精度下**四舍五入回 w 本身**，更新被静默丢弃。L9 把它写成一行不等式：

$$w \gg \alpha\nabla \implies \text{weight update gets clipped to zero}$$

9.3 三个机制：master weights、loss scaling、算子分级

9.3.1 Master weights（权重主副本）

L9 的解法（原文）："**keep master copy of weights and optimizer state in FP32** so small updates can accumulate"。图上还标了 "x learning_rate"。

【补充】完整的数据流是：

```
FP32 master w  ──cast──►  FP16 w  ──forward──►  FP16 loss
     ▲                                              │
     │                                        backward (FP16)
     │                                              ▼
     └── opt.step() ◄── unscale ◄── FP16 grads ──cast──► FP32 grads
```

注意三个细节：① 前向用的是 **cast 出来的 FP16 副本**，不是 master；② 反向得到的梯度是 FP16，要 **cast 回 FP32** 再交给优化器；③ 优化器状态（Adam 的 m/v）也是 FP32。

9.3.2 Loss scaling（损失缩放）

L9 的论证链（原文）：

- "**Gradients are small**：If cast to FP16, some lost to zero；**Much of fp16 representable ranges was left unused**"
- 解法："**Scaling gradients during backpropagation to prevent underflow**；Gradients are scaled **before** backpropagation begins and **rescaled before** updating weights"

为什么"缩放 loss"能缩放所有梯度？因为链式法则：$\nabla(\lambda L) = \lambda \nabla L$，**常数因子会原样传到每一个梯度上**。所以 scale loss 是"给所有梯度乘同一个数"的最便宜实现。

L9 还给了**自动 loss scaling** 的三条规则（原文，来自 NVIDIA 文档）：

步骤	规则
1	"Start with very large scale factor（**e.g., 2^24**）"
2	"If gradients overflows（with Inf or a NaN）: **decrease the scale by 2 and skip the update**"
3	"If no overflows have occurred for some time（**e.g., 2000 iterations**）: **increase the scale by 2**"

"skip the update"是很多人忽略的一步：一旦出现 Inf/NaN，这一步的梯度已经污染，**必须丢掉这一步**（否则 NaN 会进权重，之后永远 NaN）。

9.3.3 算子分级：哪些能 FP16，哪些不能

L9 给了"保守建议"表，这是本课程最实用的一张表：

类别	算子	精度要求	原因
可以用 FP16	**矩阵乘法**、多数逐元素运算（relu、tanh、add、sub）	FP16	误差被平均、且 Tensor Core 只在这个精度上快
需要 FP32 **尾数**	**归约运算**（softmax、layernorm）	FP32 accumulate	累加 10^4 级别的和，FP16 尾数只有 10 位，误差累积
需要 FP32 **范围**	逐元素函数（$\|f(x)\| \gg \|x\|$，如 exp、log、pow）、**损失函数**	FP32	中间值可能超出 FP16 范围（e^{1000}）

注意表头两个词的区别：**尾数（mantissa）** vs **范围（range）**。softmax 的问题是"要精度"，exp 的问题是"要量程"，两者不是同一类问题——这个区分在 L2 的 online softmax 里也出现过。

【补充】BF16 的出现正是为了"范围"这一类：BF16 有 8 位指数（与 FP32 同范围）、7 位尾数（比 FP16 的 10 位少）。所以：

格式	符号	指数位	尾数位	字节	动态范围	精度
FP32	1	8	23	4	$\sim 10^{\pm 38}$	高
TF32	1	8	10	4（存）	同 FP32	中
BF16	1	**8**	**7**	2	同 FP32	低
FP16	1	5	10	2	$\sim 10^{\pm 5}$	中
FP8 (E4M3)	1	4	3	1	小	很低
FP8 (E5M2)	1	5	2	1	中	极低

这张表解释了为什么 **LLM 训练普遍用 BF16 而不是 FP16**：LLM 的激活里经常有量级 10^2–10^3 的离群值（L16 会专门讲 activation outlier），FP16 会溢出，BF16 不会；代价是尾数少 3 位，需要更小心地做归约（这就是为什么 BF16 训练里 layernorm/softmax 仍然用 FP32 累加）。

9.4 收益与代价的实测

L9 给了一个完整的实验（原文）：

> **Mixed Precision Training Example** – ResNet–50 training for ImageNet classification – 8 GPUs on DGX–1 – Comparing to FP32 training – **~3x speedup** – **Equal accuracy** – **No hyperparameters changed**

"3x 而不是 8x"这个差距本身就是课程想传达的信息：Tensor Core 的 8× 是**矩阵乘峰值**，端到端还有 softmax、layernorm、数据加载、通信在 FP32 上跑，Amdahl 定律立刻生效。总结页也承认这一点：

> - Mixed precision **improves efficiency but may hurt quality**（FP16 vs. INT8）
> - **Automatic Scaling at runtime may help quality**

9.5 与并行策略的耦合（课程里没明说，但会影响你的配置）

【推断】混合精度不是独立的一层优化，它改变了第 5–8 章所有账本里的系数：

场景	FP32	BF16 混合
每参数显存（Adam）	16 B	2+2+4+4+4 = **16 B**（没变！）
每参数梯度通信量	4 B	**2 B**（减半）
每参数计算吞吐	1×	8×（峰值）
激活显存	1×	~0.5×

第一行最容易误解：**混合精度并不减少"每参数 16 字节"的训练状态**（因为 master weight 与优化器状态仍是 FP32）。它省的是**激活**与**通信**。真正把 16Ψ 降下来的是 ZeRO（第 5.5 节）或低精度优化器状态（8-bit Adam 之类，课程未涉及）。

第 10 章 训练显存优化（L10）

> L10 是课程里"编译/运行时视角"最强的一讲：它不讨论"要不要切模型"，而是讨论**单卡内部**怎么把显存压下来。文字量充足（22.6 KB），是 L6–L8 之后难得的一份"读得懂"的讲义。

10.1 问题定义

L10 把任务写死在一句话里：

> - **Task**: Reduce the memory consumption to train a DL model **on single device**
> - **Main challenge**: Cost to store **intermediate results and gradients**
> - **Key Ideas**:
> - Computation graph memory optimization（**In-place Storing, Memory Sharing**）
> - Save memory by **re-computing**（**Activation checkpointing**）

注意 "on single device" 这个限定词：这一讲与并行无关，是"**在给定的一卡显存里塞下更多东西**"。它与 ZeRO/并行是正交的两条路。

10.2 动态分配 vs 静态分配

L10 用四页画了"给定计算图，构建 executor"的过程，然后对比两种内存管理：

维度	Dynamic Memory Allocation	Static Memory Allocation
决策时机	运行时	**编译期**（determined at compile time）
灵活性	高，能更有效地用内存	低，可能浪费
运行时开销	有（分配器）	**无**
主要问题	**fragmentation（碎片）**	**wasted memory（预留浪费）**
代表实现	PyTorch eager	**TensorFlow 1.x、PyTorch 2.0（static graph mode）**

这张表与第 14 章的 PagedAttention 是**同一个问题的两个尺度**：训练图的静态内存规划 vs 推理 KV 池的动态分页。课程把它们放在 L10 与 L14，中间隔了三讲，但思想同源。

10.3 三个图级优化

10.3.1 In-place（原地写）

L10 的定义与约束（原文）：

> - Operations that modify data **directly in the memory location where it is stored**
> - Examples: In-place activation functions, in-place BatchNorm
> - Benefits: Reduce memory footprint；Improve computation efficiency
>
> **We can only do in-place computation if the result op is the only consumer of the current value**（即没有别的算子还依赖这个值）

课堂提问是 **"What operation cannot be done in-place?"**。【补充】标准答案是**任何"输出形状/索引依赖输入"的算子**，以及**被多个下游消费的张量**：`reshape` / `transpose`（改变 stride 视图）、`softmax`（需要先求 max/sum，若覆盖输入就没法回代）、残差连接的输入（`x + f(x)` 里 `x` 还被 add 用着）。所以在 Transformer 里，能安全 in-place 的通常只有**激活函数与 dropout 的输出**。

10.3.2 Memory sharing（内存复用）

L10 的定义（原文）：

> - Memory used by intermediate results that are **no longer needed** can be recycled and used in another node
> - **Shared memory between the nodes whose lifetime do not overlap**

关键概念是 **liveness（活跃期）**：

> - **Static Memory Planning**: plan for reuse ahead of time
> - **Liveness counters**: Track the number of operations that need to be fulfilled before memory can be reused
> - **Dependencies**: Important to declare minimum dependencies to optimize memory usage

【补充】这套算法就是编译器里的**寄存器分配**：把每个张量的活跃区间看成一条线段，重叠的不能共用同一块内存，不重叠的可以。最优解是区间图着色（区间图是完美图，贪心按起点排序即可最优）；工程实现通常用"按大小分桶 + 首次适配"的近似。L10 说了一句很关键的限定：

> (DL frameworks already handle this for common operators)

也就是说：**这一层优化已经在 PyTorch/TF 里做掉了，你不需要手写**。这也是为什么课程把它作为"理解原理"而不是"作业"。

10.3.3 收益的天花板

L10 用一句话点出了 memory sharing 的局限：

> **We can save memory usage by a constant factor using memory sharing for training, how can we do better?**

"constant factor"（常数倍）——这就是瓶颈。因为无论怎么复用，**反向传播需要的激活必须全部活着**（否则算不出梯度）。于是内存仍然是 $O(n)$（n = 层数），只是系数小了点。要突破这个天花板，必须换思路：**不存，重算**。

10.4 Activation Checkpointing：用计算换显存

10.4.1 算法

L10 给的算法只有三步（原文）：

```
1. Divide the network into several segments;
2. Store the output of each segment and drop all the intermediate results within each segme
3. Recompute dropped results at segment-level during back-propagation
```

一句话版本（原文）："**Checkpointing**: storing only a few intermediate results during the forward pass. **Recomputation**: Recomputing the necessary values during the backward for gradient calculation"。

10.4.2 次线性内存：为什么是 $O(\sqrt{n})$

L10 用两页推这个结论，页面上的文字是：

> Assume we divide the n network into **k segments**, the activation memory to train the network: （公式被图遮挡） Assume segments are equally divided, setting （$k = \sqrt{n}$）, we get （$O(\sqrt{n})$） with **only one additional forward pass**.

【补充】把推导补全（这是 L10 那张图的代数化）：

- 每个 segment 的**内部**激活可以全部丢掉，只留 segment 的输出 → 常驻内存 = k 份（每个 segment 一份输出）；

- 反向到某个 segment 时，需要重算该 segment 的内部激活 → 临时内存 = 单个 segment 的大小 = n/k 份；
- 总峰值 $\approx k + \frac{n}{k}$；
- 对 k 求极小：$\frac{d}{dk}(k + n/k) = 1 - n/k^2 = 0 \Rightarrow k = \sqrt{n}$；
- 最小值 $= 2\sqrt{n}$，即 $O(\sqrt{n})$。

代价是**额外一次前向**（严格说：每个 segment 重算一次，总量等于一次完整前向），所以训练时间约增加 33%（前向:反向 ≈ 1:2，多一次前向 → $3 \to 4$，$4/3 \approx 1.33$）。

L10 的实测页印证了这个估算：

> Running different schemes on ResNet and LSTM. **Training time roughly increases 30%.**

30% ≈ 33%，理论与实测对得上。

10.4.3 Selective checkpointing：只丢便宜的

L10 给了一个更细的策略（原文）：

> **Selective Activation Checkpointing**: Drop the results of **low compute cost** operations, e.g., Conv–BatchNorm–Activation, drop results of the batch norm, activation, and pooling, but **keep the results of convolution**

这条规则的逻辑是**性价比**：重算一个 Conv 很贵（FLOPs 高），重算一个 BN/ReLU 很便宜（逐元素）。所以要丢的是"**占显存多、重算便宜**"的那些。

【补充】把它写成可操作的判据：对每个张量，计算 $\frac{\text{bytes saved}}{\text{recompute FLOPs}}$，优先丢这个比值高的。在现代 Transformer 里，"便宜"的算子恰恰是**占激活大头**的：attention 的 $N \times N$ 中间矩阵、LayerNorm 的输入、dropout 的输出。这也是 FlashAttention 的反向能"不存 $N \times N$ 矩阵"的原因——它把重算做进了 kernel。

10.5 这一讲与 ZeRO 的分工

L10 的最后一页是 panel 讨论的引子，一句话把边界划清了：

> **ZeRO partitions optimizer states and model weights across machines to reduce peak memory demands at the cost of higher communication.**

对照来看：

手段	作用域	省的什么	代价
In-place / memory sharing	**单卡、图内**	中间结果的峰值	无（编译器自动）
Activation checkpointing	**单卡、时间维**	激活的 $O(n) \to O(\sqrt{n})$	+30% 训练时间

手段	作用域	省的什么	代价
ZeRO	**跨卡**	优化器状态/梯度/参数	通信量增加
混合精度	单卡	激活与通信字节数	数值风险（第 9 章）

四者**可以叠加**，且叠加顺序在工程上有讲究：先混合精度（免费的一半），再 checkpointing（用时间换），最后才是 ZeRO（用通信换）。

第 11 章 微调技术（L11）

11.1 从"全参微调"到 PEFT

L11 的三段式动机（原文）：

概念	讲义定义
Full-parameter Fine-Tuning	"to accommodate new tasks/datasets；**Updating all model parameters is resource-intensive**"
Parameter-Efficient Fine-Tuning (PEFT)	"update a **small subset** of parameters；Tradeoff between efficiency and model quality"
Low-rank Adaptation (LoRA)	"Pre-trained LLMs have a **'low intrinsic dimension'**；LLMs can still learn efficiently despite a **low-dim reparametrization**"

"low intrinsic dimension" 是 LoRA 的立论基础：**任务适配所需的参数变化，落在一个很低维的子空间里。**L11 后面还用一张图反问 "What is the optimal rank for LoRA?"，答案是：

> "**increasing rank does not cover more meaningful subspaces**, which suggests that a low-rank adaptation matrix is sufficient."

11.2 LoRA 的一页纸

L11 的 "LoRA in One Slide"：

> **Insight**: insert small trainable weight matrix (as the **side path**) while keeping the **base model frozen**

【补充】数学形式：

$$h = W_0 x + \Delta W x = W_0 x + \frac{\alpha}{r} B A x, \qquad B \in \mathbb{R}^{d \times r},\ A \in \mathbb{R}^{r \times k},\ r \ll \min(d, k)$$

前向时 W_0 冻结，只训练 A、B。初始化通常是 $A \sim \mathcal{N}(0, \sigma^2)$、$B = 0$（保证训练开始时 $\Delta W = 0$，不破坏原模型）。α/r 是缩放因子，用来在改 r 时保持有效学习率不变。

L11 给的**反向传播**关键句（原文）：

> The gradients of the loss L with respect to A and B can be derived using the chain rule … （W_0 部分标注为）**constant "W/o needing to compute the gradients for the base model (W_0)!"**

这是 LoRA 省显存的真正来源，值得展开（【补充】）：

$$\frac{\partial L}{\partial B} = \frac{\alpha}{r} \frac{\partial L}{\partial h} (Ax)^\top, \qquad \frac{\partial L}{\partial A} = \frac{\alpha}{r} \left(B^\top \frac{\partial L}{\partial h} \right) x^\top$$

两式里都**没有 W_0 的梯度**，也没有对 W_0 的反向。所以：

项	全参微调	LoRA
可训练参数	Ψ	$2rd$ 每层（通常 <1%）
优化器状态显存	12Ψ 字节	$12 \times 2rd$
梯度	全量 Ψ	只有 A, B
前向	$W_0 x$	$W_0 x + BAx$（多一次小 matmul）

L11 的 "Benefits of LoRA" 页把最后一行写成："**Negligible inference latency as r << d**"——因为可以把 BA 合并回 W_0（$W = W_0 + BA$），推理时**零额外开销**。这是 LoRA 相比 adapter（串行插入、必增延迟）的关键优势。

11.3 设计选择：为什么加在 attention 上

L11 有一页标题是 "Design Choice of LoRA"，页面上只有一句问答：

> **LoRAs are mostly applied to the attention layer, why?** → "LoRAs are mostly applied to the attention layer **to capture token interactions**"

【补充】原论文的经验是：把 LoRA 加在 W_q, W_v 上性价比最高，加在 MLP 上收益递减，加在 W_k, W_o 上几乎无用；同时**增大 r 不如增大"加哪些矩阵"的覆盖**。这与"低内在维度"的假设一致：真正需要适配的是"如何看上下文"，不是"如何做逐 token 变换"。

11.4 服务很多 LoRA：S-LoRA

L11 的第二部分把视角从"训练"切到"**服务**"，这是 CS 498 独有的视角（M2/M3 都没有）。

问题设定（原文）：

> - Cloud providers may host **many adapters** for an LLM；Different users may use different adapters for different scenarios
> - "**Setting up model replicas for each user (LoRA) under-utilizes GPUs**"
> - "We can **co-locate multiple LoRAs while sharing one base model**"

S-LoRA（MLSys'24）的核心设计（原文列点）：

机制	内容
存放位置	"Store adapter weights in **main memory (e.g., DRAM)**；GPU memory too small to store there"
朴素做法的两个问题	"**Dynamic fetching** leads to **memory fragmentation and I/O overhead**"、"Hard to **batch computation of adapters in non-contiguous memory**"
Unified Paging	"KV cache and adapter weights **share same memory pool**；Both allocated on request input and deallocated on request completion；**Interleaved storage** to reduce fragmentation"
Prefetching / overlapping	"Predict adapter weights needed for **next batch** when decoding current batch"
Heterogenous Batching	"Batching to allow concurrency of **different adapters**"
S-LoRA Tensor Parallelism	"Split up adapter weights for inference across GPUs"；"Act very similar to single-model parallelism as **LoRA r << d**"

三个总结（原文）：

> - **Unified Paging**: Unified memory pool to reduce fragmentation, increase batch size
> - **Heterogenous Batching**: Batching to allow concurrency of different adapters
> - **S-LoRA Tensor Parallelism**: Split up adapter weights for inference across GPUs

【推断】S-LoRA 最值得学的是"**把 KV cache 与 adapter 权重放进同一个分页池**"这一招。它把两个生命周期相似（都是 request 来则分配、request 完则释放）但**类型完全不同**的对象统一管理，从而让碎片率由"两个池各自的碎片"降为"一个池的碎片"。这与第 14 章 PagedAttention 是同一套思路的复用，只不过对象从"一个请求的 KV"扩展到"一个租户的 adapter"。

11.5 dLoRA：从静态放置到动态编排

L11 用两页讲 dLoRA（OSDI'24），起点是 S-LoRA 的一个假设：

> **S-LoRA assumes all-or-nothing serving**（use that LoRA or not） "The burst of **variable requests** leads to **severe load imbalance** under **static LoRA placement**"

dLoRA 的三条关键洞察（原文）：

> - Set up **multiple model replicas** and **replicate hot LoRAs**
> - Considering **both adapter loading time and queuing delay** to balance the load
> - **Reactively migrate** workload to balance the load

总结页把三者串成一句话：

> **dLoRA = S-LoRA + (reactive) load balance + (proactive) migration**

【补充】这三条对应三个经典分布式问题：① **复制热对象**（hot replica，缓存/CDN 里的老问题）；② **把"加载成本"计入调度代价**（否则会做出"迁移很便宜"的错误决策）；③ **反应式迁移 + 主动预测**（reactive + proactive）。课程把它放在 L11 结尾，其实是在为 L13 的调度与 L15 的 KV 复用做铺垫：**"什么东西放在哪一层、什么时候搬"是同一个问题的不同实例**。

11.6 课程没讲但必须知道的两件事

作业与项目里一定会遇到，这里补两句（`【补充】`）：

技术	一句话	与 LoRA 的关系
QLoRA	把 base model 量化到 4-bit（NF4）+ 双重量化 + paged optimizer，反传穿过冻结的量化权重到 LoRA	显存再降 ~4×，质量接近 16-bit LoRA；本质是 LoRA + 第 16 章的量化
RLHF / DPO	用人类偏好训练奖励模型，再用 PPO/DPO 优化策略	与 LoRA 正交，可组合（LoRA + PPO 是常见省钱做法）；README 的 grading 里 "Tuning/Alignment" 只有 1 讲，所以课程只到"alignment 属于 lifecycle 的一环"这个颗粒度

L11 的总结页留了一个开放问题，很能代表这门课的风格：

> **Open problems?**（e.g., **how to cache LoRA and KVCache**）

"LoRA 与 KV cache 如何共享缓存"——这个问题在 L14（分页）、L15（保留策略）、L18（CacheBlend）里被反复触碰，是贯穿后半程的一条暗线。

第 12 章 推理总览（L12）

12.1 五个场景，五种 SLA

L12 用五页列了推理的典型场景，这一页一页看过去其实是在建立"**SLA 是分类型的**"这个认知：

场景	讲义举例	关键约束
对话/搜索	ChatGPT / SearchGPT / Copilot	TTFT 极敏感（人在等）
在线图像/视频生成	DALL·E 3、Sora2	单次生成时间以秒~分钟计，可批处理
监控系统	Surveillance System	持续吞吐、成本敏感
移动 App	Llama 3.2 量化模型 + ExecuTorch + KleidiAI	端侧显存/能耗
AR/VR	Augmented and Virtual Reality	延迟硬约束（帧级）

12.2 三条扩展律：为什么推理需求不会下降

L12 引用 Jensen Huang 在 CES 2025 / GTC 2025 的两张图，把 scaling 分成三类（讲义原文）：

#	名称	含义
1️⃣	**Pretraining scaling**	"bigger model and data, better performance"
2️⃣	**Post-training scaling**	"fine-tuning for precision"
3️⃣	**Test-time scaling（long thinking）**	"**multi-pass reasoning** for complex problems / try it many times"

结论页的标题就是判决："**Inference Time Scaling Requires More Compute**"。

【推断】这一页是本课程"为什么后半程全讲推理"的答案：**训练是一次性的资本支出，推理是持续的经营支出**，而 test-time scaling 让推理的 token 数随问题难度线性甚至超线性增长。所以"每 token 成本降 10×"会被"每请求 token 数涨 10×"吃掉——这是所有推理优化的"红皇后"困境。

12.3 训练 vs 推理：一张对照表

L12 逐帧给出了这张表（原文）：

维度	Training	Inference
Runtime	Weeks or months	**Milliseconds or seconds**
Challenges	Speed, Scale, Data	**SLA（Service-level agreement）、TCO（total cost of ownership）**
重要因素	—	**Speed（LLM: token rates）**、**Model size**（Parameter volume；LLM: **Context length**）

这张表最值得注意的是最后一格：**推理侧的"模型大小"由两部分组成——参数量与上下文长度**。而这正是第 14、15 章的全部战场（KV cache 随上下文线性增长）。

12.4 三个挑战

12.4.1 挑战一：延迟违反 SLA

L12 给了一张非常具体的表（原文），它把"延迟"从形容词变成了数字：

DL 场景	原始延迟	延迟目标
Deep Query Document Similarity	10~12 ms for [query, 1 doc] × **33 docs**	**< 6 ms**
Travel Planning	10 ms for [query, 1 passage] × **150 passages**	**< 5 ms**
Ads seq2seq model for query rewriting	**~51 ms**	**< 5 ms**

这张表的可怕之处在于：**原始延迟与目标差 10 倍量级，而且原始延迟本身还随检索文档数线性增长**。广告 query rewriting 那行（51 ms → 5 ms）需要 **10× 加速**，这不是靠"换个更快的 kernel"能解决的——必须动模型（蒸馏/量化）或动系统（并行/缓存）。

12.4.2 挑战二：小 batch 限制了并行度

L12 用一张对照图说明：

```
Training :  Input (N × 8192)  × Weight (8192 × 4096)   N = 2K, 4K, ...
Inference:  Input (1 × 8192)  × Weight (8192 × 4096)   batch = 1
```

讲义结论："**Small batch size → Low latency yet low data reuse (arithmetic intensity)**"、"**Autoregressive generation → Generate tokens one-by-one**"。

【补充】把它写成数字会更震撼：一个 8192×4096 的 bf16 权重矩阵是 64 MiB。batch=1 时，读 64 MiB 只做了 $2 \times 8192 \times 4096 = 6.7 \times 10^7$ FLOPs，AI ≈ 1 FLOP/byte；在 3 TB/s、1000 TFLOPS 的 GPU 上，ridge ≈ 333 ops/byte。**离脊点差 300 倍** → 这张卡在这一步只发挥了 ~0.3% 的算力。

12.4.3 挑战三：显存推高成本

L12 引 PagedAttention 论文（Kwon et al., 2023）的图，标注 "**LLaMA-13B on A100 40 GB**"，并列出显存的三个占用者：

占用者	随什么增长
Model parameters	# Layers、# Hidden dim
KV cache（in attention layer）	**Batch size**、**Sequence length**、# Layers、# Hidden
Activation and others	运行时缓冲

【补充】算一下这张图的张力：LLaMA-13B 的 FP16 权重 = 26 GB，A100 40 GB 只剩 14 GB 给 KV 与激活。若每 token KV 约 800 KiB（40 层、40 头 MHA、d=128、bf16：$2 \times 40 \times 40 \times 128 \times 2 = 819{,}200$ B），则 14 GB 只够 **约 17k 个 token**——即一条 4k 上下文的请求只能并发 4 条左右。**这就是"显存决定并发、并发决定吞吐、吞吐决定成本"的完整链条。**

12.5 三层优化栈

L12 用一张"down-to-top view"把优化手段分了层（原文）：

层	讲义描述	具体技术	对应讲次
Execution layer	how to execute efficiently	定制 kernel（FlashAttention）、模型并行（AlpaServe）	L2、L6—L8、L12
Planning layer	how to schedule resources	Orca（iteration-level scheduling）、Sarathi-Serve（chunked prefill）、PagedAttention / SGLang（KV 管理）	L13—L15
Algorithm layer	how to change model	压缩（剪枝/量化/低秩/蒸馏）、MoE、speculative decoding	L16、L17

这张三层图是整门课推理部分的地图：**L13 在 planning 层，L14/L15 在 planning+execution 之间，L16/L17 在 algorithm 层，L18 横跨三层。**

12.6 算法层：压缩四件套

L12 列了四类压缩（原文标题）：

技术	讲义要点
Pruning / Sparsification	三类稀疏的对照（见下）

技术	讲义要点
Quantization	"Reduce the bits per weight, saving memory consumption；Accelerate inference speed on supporting hardware"
Low-rank decomposition	只列名
Distillation	只列名（引 *Distilling the Knowledge in a Neural Network*, 2015；以及 IC-Cache, SOSP'25）

12.6.1 稀疏的三种形态（讲义原文对照）

形态	精度	性能
Unstructured（connection）Sparsity	"High accuracy"	"**No performance improvement or performance regression**"
Structured Sparsity	"Large accuracy degradation"	"**High performance scalability**"
N:M Semi-Structured（如 4:2）	"High accuracy"	"**High performance improvement**"

这张表是"稀疏为什么难落地"的最短回答：**非结构化稀疏精度好但硬件不认（不规则访存 + 索引开销），结构化稀疏硬件认但精度崩，N:M 是唯一的中间地带**（因为 2:4 稀疏可以映射到 Tensor Core 的稀疏 MMA 指令，理论上 2× 吞吐）。

12.6.2 量化的两个层次

L12 用四页讲量化的"为什么难"：

问题	讲义内容
标准量化会崩	"Standard quantization strategy leads to **catastrophic accuracy drop**"（引 LLM.int8(), 2023）
粒度权衡	Per-tensor："Low accuracy / Fast to quantize/dequantize"；Per-token/channel："**High accuracy / Slower / Custom kernels required**"（引 ZeroQuant, NeurIPS 2022）
混合精度	"Weights follow **Gaussian distribution**；**Outliers remain in original form**, quantize the rest；Different bits for different layers"（引 GOBO, MICRO 2020）

"Outliers remain in original form" 就是 LLM.int8() 的混合精度分解（离群值走 FP16，其余走 INT8）。这条思路在 L16 会被 SmoothQuant 用另一种方式（**把离群值从激活迁移到权重**）取代——**L12 提问题，L16 给答案**，这是课程的一条暗线。

12.7 四个指标的严格定义

L12 给了推理指标的权威定义（原文，这是全课程最该背下来的一页）：

指标	讲义定义
TTFT（Time to First Token）	"Measures how quickly users begin to see the model output token after submitting a query；**Critical for real-time interactions**；Driven by **prompt processing time** and the generation of the first token"
TPOT（Time per Output Token）	"Time taken to generate each output token；Impacts user perception of speed（e.g., **100ms/token = 10 tokens/second**）"
E2E Latency	"**TTFT + TPOT × (the number of generated tokens − 1)**"
Throughput	"Number of tokens generated per second **across all requests** by the inference server"

注意 E2E 公式里的 **−1**：第一个 token 的时间已经算在 TTFT 里了，不能重复计。这个小细节是很多"延迟模型"出错的地方。

以及那句核心权衡（原文）：

> **Throughput vs. TPOT Tradeoff**: "Processing multiple queries concurrently **increases throughput** [but] **extends TPOT** for each user."

【补充】把这句话写成公式，就得到 serving 的基本优化目标：

$$\max \text{ goodput} = \frac{\#\text{requests meeting } (\text{TTFT} \le S_{ttft}) \wedge (\text{TPOT} \le S_{tpot})}{\text{second}}$$

注意是 **goodput 不是 throughput**——超 SLA 的吞吐是无效吞吐。L14 的 panel（DistServe）与 L18 的 Mooncake 都在优化这个量，而不是原始 tokens/s。

第 13 章 批处理与调度（L13）

13.1 从"请求级调度"到"迭代级调度"

L13 用一个五帧动画讲了静态 batching 的两个病，这是理解 continuous batching 的最短路径：

问题	讲义原文	后果
早完成的请求出不去	"Early finished requests **cannot return** to the Inference Server"	Latency increase（x2 已经生成完，但要等 x1）

问题	讲义原文	后果
晚到的请求进不来	"Late join requests need to wait until engine finishes execution"	Latency increase（x3 要等整个 batch）

根因是**调度粒度太粗**：一次调度决定了"接下来几百个 step 谁在算"，而 LLM 的 decode 是**每个 step 只产一个 token**，每个 step 的完成情况都不同。

Orca（OSDI'22）的解法叫 **iteration-level scheduling**（迭代级调度），讲义用 8 帧动画演示：**每个 iteration 结束后重新选一次 batch**。图上的文字标注是 "schedule one iter" 与 "select requests"，x2 生成 `<EOS>` 后立刻返回，x3/x4/x5 立刻补位。

13.2 调度器的四条规则

L13 的 "LLM Inference Scheduler" 页给了 Orca 式调度器的实现要点（原文）：

> - Enforces **iteration-level first-come-first-served (FCFS)** property
> - **Maximum batch size → Throughput vs. Latency control knob**
> - Keep track of **number of reserved slots to avoid deadlock**
> - **Slot := memory required for storing an Attention key and value for a single token**
> - **Reserves max_tokens memory slots per request**

这四条里，前两条是策略，后两条是**正确性**：

问题	若不处理会怎样
slot 计数（避免死锁）	若每个请求都"先占位再算"，可能出现"所有请求都占了一部分、谁也无法完成"——经典的分段分配死锁
预留 `max_tokens`	若不预留，一个请求可能在 decode 到一半时发现 KV 池不够，只能 preempt 或 OOM

【补充】"slot"这个概念在第 14 章会升级成 "block"，但本质相同：**把 KV 显存抽象成可计数的单位，调度器才有可量化的约束**。HW2 的 `PagedKVCache.reserve()` 与 `ContinuousBatcher._try_reserve()` 就是这两个概念的代码化（见速查文档第 4 节）。

13.3 Continuous Batching 的收益

L13 的总结页（原文）：

> - Continuous batching handles **early-finished and late-arrived** requests more efficiently
> - **Fills GPU capacity after each token generation**
> - **As variance in sequence length increases, continuous batching increases GPU utilization**

第三条最容易被忽略：**收益随"序列长度方差"增大而增大**。如果所有请求长度都一样，静态 batching 的浪费本来就小；而真实负载（聊天、RAG、agent）的长度分布是长尾的，所以收益巨大。

【补充】Anyscale 那篇经典博客（L13 引了链接）给的量级是 **23× 吞吐**（相对 naive static batching，在特定负载下）。但要记住这是"特定负载 + 特定实现"的数字，**不能当成普适常数**——它取决于长度方差、到达率、SLA 松紧。

13.4 Chunked Prefill：把长 prefill 打碎

L13 最后一节只有一页，但信息密度很高：

> **Chunked-Prefill in One Slide** Core idea: **partition prefilling input into multiple chunks (e.g., 256 tokens/chunk) and mix with decoding requests**

以及一个反问："**How about using a larger batch size?**"（意思是没有 chunking 时，只能靠"更大的 batch 摊薄长 prefill 的抖动"，但那会更严重地伤害其他请求的 TPOT。）

【补充】为什么需要 chunked prefill？因为 prefill 与 decode 的计算特征完全不同：

阶段	一次前向的 token 数	算术强度	对同 batch 其他请求的影响
Prefill（L 长）	L（可上千）	高（compute-bound）	**一个长 prefill 会把整个 batch 的 TPOT 拉爆**
Decode	1	低（memory-bound）	平稳

chunked prefill 把"一次 8k 的 prefill"拆成 32 次 256 token 的 prefill，与 decode 混在一个 batch 里，使 **每次 iteration 的 token 总量可控**（这就是 Sarathi-Serve 的 "stall-free batching" 思路）。代价是：① 首 token 延迟变长（要多走几轮）；② 实现上要处理"部分 prefill 的 KV 已在 cache 里"的状态。

L13 的讲义明确说明这是一个**取舍**而不是免费午餐——它优化的是 TPOT 的稳定性（P99），付出的是 TTFT。

13.5 Speculative Decoding：为何放在 panel 而不是正文

L13 的 panel 讨论题目是 *Fast Inference from Transformers via Speculative Decoding*（2023）。正文里没有展开，但 HW3 Part 1（32 分）把它讲透了。这里先给系统视角的动机，细节见第 16 章与速查文档第 5 节。

【补充】speculative decoding 的系统逻辑只有一句：**decode 是 memory-bound，读一次权重只算一个 token 太亏；那就让一个小模型先猜 K 个 token，大模型一次前向并行验证 K 个**。这样"读一次权重"摊到了多个 token 上，算术强度提升约 K 倍。

但要注意它的三个限制条件（【推断】）：

1. **必须有 draft 模型**，且 draft 的质量（acceptance rate）决定收益；
2. **batch 越大收益越小**——因为大 batch 本身已经把权重读取摊薄了，speculation 的边际收益下降（HW3 1.1 节的 "Systems Perspective" 页原文就写了这一点）；
3. **验证是并行的，接受是串行的**——一旦第 i 个候选被拒，后面全部作废，所以期望接受长度是收益的核心。

HW3 里那句原文很好，直接抄下来：

> "At small batch sizes, autoregressive (AR) decoding is often **memory-bandwidth bound** … Verification evaluates several candidate positions together and **reuses weights across them**. This increases **arithmetic intensity** … **At high concurrency, ordinary batching already improves utilization, leaving less room for speculation to help.**"

第 14 章 PagedAttention（L14）

> L14 的讲义是 18 份里排版最"现代"的一份（有章节导航条、learning objectives、takeaways），文字信息量大。本章尽量保留原文表述。

14.1 学习目标

L14 开篇给了三条 learning objectives（原文），这三条正好对应本章的三节：

目标	对应节
Explain how KV caching reduces decoding compute while increasing memory demand	14.2
Apply a logical-to-physical block table across multiple decode steps	14.3
Evaluate fragmentation, sharing, overhead, and serving throughput tradeoffs	14.4

14.2 从 KV caching 到"内存不够"

14.2.1 prefill 与 decode 的形状差异

L14 用一页讲清了两者的形状（这是理解后面所有优化的前提）：

阶段	Q	K / V	注意力形状
PREFILL	S tokens	S tokens	$S \times S$ causal attention
DECODE	**1 token**	all retained tokens	$1 \times S$ **attention row**

配的那句因果性说明很关键："**Causality means a new token never changes the old K/V projections.**"——这就是 KV 可以被缓存的根本原因（如果新 token 会改变旧 K/V，缓存就失效了）。

14.2.2 不缓存的代价

L14 用三帧图演示"重算全前缀"：

```
Step 1  [1 2 3 4]              → 重算 4 个 token 的 K/V
Step 2  [1 2 3 4 5]            → 重算 5 个 token 的 K/V
Step 3  [1 2 3 4 5 6]          → 重算 6 个 token 的 K/V
```

配文："The cost of earlier tokens is **paid again and again.**"

14.2.3 复杂度降阶

方式	位置 t 的 attention 计算量	配文
Recompute full prefix	$O(t^2 \cdot H)$	"Processes all earlier queries again"
Incremental KV decode	$O(t \cdot H)$	"One new query attends to cached K/V"

总结句（原文）："**KV caching exchanges repeated computation for persistent GPU memory.**"

14.2.4 显存是并发的上限

L14 用一页图说明"权重、KV、运行时缓冲争抢同一块显存"，并给了一个具体数字：

> **Illustration: FP16 13B parameters ≈ 26 GB before caches and buffers.** "The practical KV budget is what remains after **weight and runtime allocations.**"

然后一句因果链："**Less memory waste → more live sequences → larger effective batch.**"

14.3 三种浪费与"预留最大值"的愚蠢

14.3.1 朴素做法

L14 的图：每个请求按**最长可能输出**预留一块**连续**显存。图上标了三个请求的利用率：**3/12、7/12、5/12**。配文："Output lengths are **uncertain** yet pre-reserving the maximum **wastes capacity**."

14.3.2 三种浪费（原文定义）

#	名称	定义
01	**Reservation**	"Memory reserved for tokens the request **may never generate**"
02	**Internal fragmentation**	"Unused space **within the last allocated storage unit**"
03	**External fragmentation**	"Free space **spread across gaps**, blocking a large contiguous request"

以及一个引用数字：

> **Orca allocators use only about 20–40% for actual token states in these cases.**

【补充】这意味着 **60–80% 的 KV 显存被浪费**。换算成吞吐：如果能把这 70% 收回来，并发数理论上能到 3–5 倍。这就是 vLLM 论文里"2–4× 吞吐提升"的来源。

14.4 PagedAttention：把 OS 的分页搬过来

14.4.1 核心思想

L14 的标题页就点明了："**The idea comes from OS memory paging**"，配文："A logical sequence **does not need to be a physically contiguous allocation**."

三个概念（原文）：

概念	定义
Logical blocks	"Ordered by position within a request"
Physical blocks	"Placed **anywhere** in a GPU KV block pool"
Block table	"**Maps logical block IDs to physical block IDs**"

以及一句系统层面的提醒："**The attention kernel must follow the block table when reading K/V.**"——分页不是免费的，它把"连续访存"变成了"**间接寻址**"，kernel 必须重写。

14.4.2 分页不改数学

L14 特意用一页澄清概念（这页很重要，因为很多人误以为 PagedAttention 是"稀疏化"）：

> "Block indirection **changes the read pattern, not the attention equation**." "What changes? **KV-address translation and block-aware memory access.**" "**Paging is a memory-management mechanism, not token eviction.**"

最后一句是它与第 15 章（H2O/StreamingLLM 的驱逐）的分界线：**L14 一个 token 都不丢，L15 才开始丢 token**。

14.4.3 block table 的四步走查（block size = 4）

L14 用四页逐帧演示了一个 4-token block 的完整生命周期，这是本章最该动手复现的部分：

步骤	事件	block table 变化	关键点
Step 1	prefill 7 个 token（A–G）	0→3, 1→6	"**Only the final block may be partially occupied**"（逻辑块 1 只用了 3 个槽）
Step 2	decode 写 H	不变	"Token H fills an **already allocated** physical block; **no new block needed**"
Step 3	decode 写 I（跨块边界）	新增 2→1	"A new block is allocated **only when the active logical block becomes full**"
Step 4	decode 写 J	不变	"**Existing blocks do not move**"

这四步对应 HW2 的 KV-1（reserve）、KV-2（slot_mapping）、KV-3（append/read）三个考点：**reserve 只在跨块时分配，slot_mapping 做"逻辑块→物理块→扁平槽位"的两级映射**。

14.4.4 两个"白送"的能力

分页带来两个**不用额外设计就得到**的能力：

能力	讲义表述	机制
跨请求共享块池	"A block returned by one request can be **reused by another**"	池化的自然结果
Copy-on-Write 共享前缀	"Parallel outputs can share identical prefix KV until their histories diverge"、"Prefix K/V: **reference-counted blocks**"	引用计数 + 写时复制

第二个是 PagedAttention 论文里"parallel sampling / beam search 省 30–50% 显存"的来源：

```
共享前缀 [Sys][Few][Shot][Q]  → 引用计数 1 的若干 block（只读）
        │
        ├─ Branch A: 写到新 block（COW 触发）
        └─ Branch B: 写到新 block（COW 触发）
```

讲义配了一句限定："**Shared blocks save memory. Writes to a diverging block require isolation.**"——COW 的经典约束。

14.4.5 内部碎片被"上界化"

L14 给了最干脆的一句话证明：

> With B-token blocks, each sequence wastes **fewer than B token slots**. Example: B = 4 tokens per block, so **waste per request ≤ 3 slots**.

对照"预留最大值"（浪费可达上万个 slot），这是**从 $O(T_{max})$ 降到 $O(B)$** 的质变。注意：**分页消灭了 external fragmentation 与 reservation，把 internal fragmentation 压到"最后一块"**，这与 OS 分页的收益结构完全同构。

14.5 代价：天下没有免费的午餐

L14 专门用一页列了 paging 的三项成本（原文）：

#	成本	内容
01	**Block-table metadata**	"Memory indirection and per-request block tracking"
02	**Kernel complexity**	"Efficient attention across **non-contiguous** physical blocks"
03	**Allocator overhead**	"Block allocation, release, **reference counting**, and **copy-on-write**"

配了一句方法论："**Measure net serving throughput — not KV utilization alone.**"

【补充】这句话是评审论文的标准：很多 KV 管理论文只报"KV 利用率从 40% 提到 90%"，但 kernel 慢了 15%、元数据占了几百 MB、分配器加了锁——**净收益可能远小于宣称**。L14 的 vLLM 结果页（"At comparable normalized latency, vLLM supports higher arrival rates"）用的正是"同等延迟下更高的到达率"，这是正确的口径。

14.6 四条 takeaways 与 DistServe 的预告

L14 的总结页（原文，四条）：

> 1. KV caching **saves decode computation**. But all retained tokens require **persistent K/V storage**.
> 2. PagedAttention **eliminates contiguous sequence allocation**. Logical blocks map to a **shared pool** of physical blocks.
> 3. **Memory efficiency enables higher concurrency**. Savings must exceed **bookkeeping and kernel overhead**.
> 4. **Systems connection**: allocation efficiency and runtime scheduling are **tightly coupled**.

最后一条引出 panel：DistServe（OSDI'24）。L14 给了它的核心图与一句对照：

> **DistServe idea**: Separate the two phases — **Prefill GPU pool**（compute-bound、TTFT-sensitive） / **Decode GPU pool**（memory-bandwidth bound、TPOT-sensitive），中间 transfer KV。 "Discuss: Should we optimize TTFT, TPOT, or **service goodput**?" **Connection**: "PagedAttention asks **where KV is stored efficiently**; DistServe asks **where prefill and decode should run**."

【推断】这两句话合起来说明：KV 的位置（PagedAttention）与**计算的位置**（DistServe 的 PD 解耦）是两个正交维度。第 18 章的 Mooncake 把它们合并成一个"KV 中心"的架构——**KV 成了系统的第一等公民，计算围着 KV 转**。

第 15 章 自适应 KV Cache（L15）

15.1 分页解决了碎片，但没解决"线性增长"

L15 的开篇提问是整章的逻辑起点：

> Why does KV memory keep growing after PagedAttention?

答案在两页之后（原文）：

$$\text{KV bytes per request} \approx 2 \times L \times H_{(\#\text{grouped head})} \times D \times T \times \text{bytes/value}$$

配置（讲义示例）：32 层、8 KV 头、head dim 128、BF16，得到三个数字：8k → 1.0 GiB，32k → 4.0 GiB，128k → 16.0 GiB。配文一句判决：

> "Even perfect page packing cannot prevent linear growth in T."

15.2 三个约束：容量、质量、策略开销

L15 把"为什么不干脆少留点"拆成三条（原文）：

#	约束	原文
01	**Capacity**	"A **bounded** number of KV tokens per request"
02	**Model quality**	"**Do not discard history that matters later**"
03	**Policy overhead**	"Eviction should **not cost more than it saves**"

然后是一个反问（本章的核心问题）：

> **"Can attention provide an inexpensive retention signal?"**

15.3 观察：attention 是稀疏的

L15 用一页图给出观察："Many scores are near zero. A small subset receives large attention. **Recent tokens can matter too.**"
配了一句非常关键的警告：

> "**But sparse today does not mean useless tomorrow.**"

以及 H2O 论文的量化（讲义引用）："The paper reports **>95% sparse scores** in most layers under its threshold."

15.4 第一个困难：稀疏 ≠ 驱逐策略

L15 用一页两帧图讲了这个陷阱：

```
当前 token 只 attend 位置 2 和 9          → 若按"当前命中"驱逐，5 会被丢
下一个 token 可能需要位置 5               → 但你已经丢了它
```

配文（原文）：

> "**We do not observe future attention before generating future tokens.**" "Need a retention signal that is **cheap and robust over time**."

【推断】这句话点出了所有 KV 驱逐算法的根本困难：它是一个**在线决策问题**，没有"未来信息"。所以任何驱逐策略都在做"用过去预测未来"的赌博，H2O 与 StreamingLLM 的差别只是**赌的是什么**：H2O 赌"历史累计

注意力的分布是稳定的"，StreamingLLM 赌"位置本身（前几个 + 最近的）就够了"。

15.5 H2O：用累计注意力做保留信号

15.5.1 三条观察

L15 把 H2O（Zhang et al., NeurIPS 2023）的论证拆成三步，每步一页图：

观察	内容
1	**attention is sparse**（>95% 分数接近 0）→ "Full access to all past KV may be **unnecessary**"
2	**token importance is skewed** → "**A small set of tokens repeatedly attracts attention**"（heavy hitters）
3	**heavy hitters affect accuracy** → "**Evicting the wrong KV hurts**"；"Removing heavy hitters reduces accuracy across evaluated reasoning tasks"

15.5.2 打分规则

L15 给的公式（原文）：

$$\text{score}(i, t+1) = \text{score}(i, t) + \text{attention}(t+1 \to i)$$

"新注意力贡献累加到旧分数上"。讲义还给了一个具体例子（5 个 token 的分数更新）：

Token	1	2	3	4	5
原分数	0.8	3.1	0.7	2.4	0.5
本步增量	+0.1	+0.6	+0.0	+0.2	+0.1
新分数	0.9	3.7	0.7	2.6	0.6

配文："**Cheap online scores approximate future value. Recency still needs protection.**"

15.5.3 保留策略：heavy hitters + recent window

L15 的图：在一个固定预算下，**同时保留"历史重要"（heavy hitters）与"最近"（recent window）**，理由各一句：

保留谁	理由
历史重要	"Retain useful **older context**"
最近	"Protect **newly created context**"

15.5.4 一个完整的算例（讲义给的玩具例子）

条件：**12 个已有 token，总共 6 个 slot，其中 3 个留给 recent**。

累积分数（讲义原表）：

Token	1	2	3	4	5	6	7	8	9	10	11	12
Score	2	**10**	3	7	4	**12**	5	1	**9**	2	3	1

- Heavy hitters（分数最高的 3 个，排除 recent 窗口）= **{2, 6, 9}**（10、12、9）
- Recent = **{10, 11, 12}**
- 讲义特意点评："**Token 4 scores relatively well（7）, but loses to 2 / 6 / 9.**"

最终保留 = {2, 6, 9, 10, 11, 12}，共 6 个。

15.5.5 H2O 的三个风险

L15 用一页列了 H2O 会在哪里栽（原文）：

#	风险	原文
01	**Future relevance is unknown**	"A low-scoring token may become **critical after a topic change**"
02	**Eviction is often irreversible**	"Without recomputation or offloading, removed KV is **gone**"
03	**Scores and eviction have cost**	"Attention tracking, ranking, and memory movement **compete with decode**"

第 03 条尤其重要：**打分本身要读 attention 矩阵、要排序**，这与第 14 章"分页要付元数据与 kernel 成本"是同一类账。

15.6 StreamingLLM：为什么滑窗会崩

15.6.1 现象

L15 引用 Xiao et al.（ICLR 2024）的 PPL 曲线，配文两句：

> "**Window-only eviction can destabilize generation even with recent tokens present.**" "（better PPL due to mitigating **lost-in-the-middle** issue）"

15.6.2 原因：attention sink

L15 用一页图解释（原文）：

> **Attention sinks: a few early positions attract attention** "First tokens (sinks) … Recent tokens"
> "**Attention sink ≠ a token is necessarily semantically relevant.**"

机制（三句话）：

#	机制
01	"**Softmax allocates probability mass**；Some attention heads learn to assign excess mass to early positions"
02	"**A pure window deletes those positions**；The resulting attention pattern **differs from training-time behavior**"
03	"**Preserving a few sinks stabilizes decoding**"

【补充】第 01 条背后的数学是 softmax 的"必须归一化"约束：$\sum_j \alpha_j = 1$，所以即使某个 query 对所有 key 都"不想看"，它也必须把概率质量分配到某些位置上。早期位置因为对所有 query 都可见（causal mask 下最早的位置被所有 token 看到），自然成为"垃圾桶"，学成 sink。一旦把这些位置删掉，softmax 被迫把质量分配到语义相关的 token 上，**分布偏移**就发生了。

15.6.3 保留策略：固定 sink + 滚动窗口

L15 给了三帧序列（**2 个 sink + 4 个 recent**）：

时刻	已生成	保留
token 8	1..8	{1, 2, 5, 6, 7, 8}
token 10	1..10	{1, 2, 7, 8, 9, 10}
token 12	1..12	{1, 2, 9, 10, 11, 12}

配文："**Bounded KV memory even as the generated stream grows.**"

15.7 两种策略的正面比较

L15 用一页图把两种策略放在**同一个 12-token、6-slot** 的场景下对照：

策略	保留
H2O	重要历史 + 最后 3 个（{2,6,9} + {10,11,12}）
StreamingLLM	前 2 个 sink + 最后 4 个（{1,2} + {9,10,11,12}）

配文："**Different goals → different retained history.**"
以及一张完整的四维对照表（原文，这是本章最该背的表）：

Property	Full / paged KV	H2O	StreamingLLM
Retained history	All tokens	**Heavy + recent**	**Sink + recent**
Memory grows?	$O(T)$	**Bounded budget**	**Fixed window**
Best strength	Exact history	**Adaptive importance**	**Long stream stability**
Primary risk	Memory pressure	**Future relevance**	**Old facts lost**

15.8 开放问题与 SGLang 的接口

L15 列了四个开放方向（原文）：

方向	问题
Dynamic KV budget	"How much retained history is worth its memory under **tight SLO slack**?"
Tiered KV management	"When to **evict, offload, prefetch, or recompute**?"
Agent context	"Can **future tool calls** tell us which earlier tokens will matter?"
Joint scheduling	"Do KV residency and batching need **one coordinated controller**?"

最后一个 open problem 直接连到下一讲的 panel：

> **SGLang and KV-cache reuse**：Key idea: "**Shared prefix → reuse prefilled KV using RadixAttention**" "Discuss: **Cache-aware scheduling, eviction under memory pressure**, and where H2O differs."

【补充】RadixAttention 的机制值得补一句（讲义只有一句图注）：SGLang 把**所有请求的 KV 组织成一棵 radix tree（前缀树）**，树的每条边对应一段 token 序列，节点持有 KV。新请求来时，沿树匹配最长前缀，**命中部分直接复用**；LRU 式驱逐时按叶子回收。这与 L14 的 COW 分页是同一目标（共享前缀）的两种数据结构：**分页用"块 + 引用计数"，radix tree 用"树 + 前缀匹配"**。

第 16 章 量化（L16）

> L16 是课程里"从零讲量化"讲得最完整的一份讲义：从 scale/zero-point 一路推到 SmoothQuant 的 α 迁移，每一步都有小算例。AWQ 在讲义里只作为 panel 讨论出现，但 HW3 Part 2（18 分）把它讲全了。

16.1 量化的位置：它是 KV 管理的"补集"

L16 开篇用一张图把三种显存压力并列：

压力来源	手段	代表工作
KV cache	retain fewer tokens	H2O / StreamingLLM（L15）
Weights	**use fewer bits**	今天的量化
Activations	use fewer bits	SmoothQuant 的 W8A8

配了一句很重要的限定：

> **"Quantization complements KV-cache management. It does not replace it."**

16.1.1 省多少：一个 7B 的账

L16 的数字（原文）：

格式	7B 权重大小
FP16	**≈ 14 GB**
INT8	**≈ 7 GB**

（讲义注明："Illustrative decimal GB（7×10^9 parameters；storage only）"。）

配文一句诚实的提醒：

> **"Lower precision saves bytes; speedup also depends on hardware kernels and conversion overhead."**

16.2 三种"量化什么"的组合

L16 用一页三栏图区分（原文）：

格式	含义	具体
W16A16	Full / half precision	FP16 weights + FP16 activations
W8A8	**INT8 matrix multiplies**	INT8 weights + INT8 activations
W4A16	**Weight-only low-bit**	INT4 weights + FP16 activations

配文："Quantization format is **not a performance guarantee**: execution kernels, packing, and memory traffic matter." 以及一句分工说明：

> "**SmoothQuant targets W8A8, while AWQ targets low-bit weight-only quantization.**"

【补充】这三种格式的适用场景差别巨大，值得列表：

格式	省显存	省带宽	能用 INT 算力	典型场景
W8A8	2×	2×	✅（INT8 Tensor Core）	数据中心高吞吐服务
W4A16	4×	4×（权重）	❌（要反量化回 FP16 做 GEMM）	**decode 显存/带宽受限**，端侧
W4A8	4×（权重）	4×（权重）	部分硬件	新硬件上的折中

关键认知：**W4A16 不提算力，只提带宽**。而 decode 恰好是 memory-bound（第 2.5、12.4 节），所以 W4A16 在 decode 上收益极大——这就是 AWQ 的战场。

16.3 均匀仿射量化的两个参数

L16 的核心公式（原文）：

$$q = \mathrm{clip}\big(\mathrm{round}(x/S) + Z\big), \qquad \hat{x} = S(q - Z)$$

符号	含义
$S > 0$	scale / quantization step size（步长）
Z	zero point（零点；q 被位宽约束）

配文："Quantization incurs **rounding** and possible **clipping** errors. **A larger S gives coarser steps.**"

16.3.1 一个完整的 INT4 算例（讲义原文）

条件：对称 INT4，有符号范围 $-7 \cdots +7$；$\max|x| = 1.4$。

$$S = \frac{1.4}{7} = 0.2$$

对 $x = 0.68$：

$$q = \mathrm{round}(0.68/0.2) = \mathrm{round}(3.4) = 3$$

$$\hat{x} = 3 \times 0.2 = 0.60, \qquad |\mathrm{error}| = 0.08$$

配文："Even without clipping, rounding to a coarse grid changes the value."

这个 0.08 的相对误差是 $0.08/0.68 \approx 11.8\%$——**这就是低比特量化的"基本盘"误差**，任何算法（SmoothQuant/AWQ/GPTQ）都只能优化它，不能消除它。

16.3.2 对称 vs 非对称

类型	Symmetric（signed INT8）	Asymmetric（unsigned INT8）
Zero point	$Z = 0$	Z can be nonzero
Scale	$S \approx \max\|x\|/127$	$S = (x_{max} - x_{min})/255$
Zero point 计算	—	$Z \approx \text{round}(-x_{min}/S)$
范围	$-127 \cdots +127$	$0 \ldots 255$

配文（原文）：

> "Both need a scale and have rounding error. **Hardware support and tensor distribution guide the choice.**" "**Use symmetric scales for simplicity. Asymmetric scales can represent skewed ranges more efficiently.**"

【补充】这条取舍在 LLM 里几乎总是选**对称**：因为权重分布近似零均值高斯（L12 说 "Weights follow Gaussian distribution"），非对称的 zero point 省不下多少范围，反而多一次加法、破坏 INT8 Tensor Core 的对称乘加。

16.3.3 粒度：一个 scale 还是很多个

L16 用一页四格图 + 一张表讲粒度（原文）：

Scheme	How scales are shared	Typical tradeoff
Per-tensor	one across tensor	"simple, **sensitive to outliers**"
Per-channel	one per weight channel	"good precision"
Per-token	one per activation token	"**adapts to token variation**"
Group-wise	one per **small block**	"**popular for low-bit weights**"

配文两句，都很关键：

> "Finer granularity can **limit outlier impact**, at the cost of **more scales and execution complexity**." "**Per-tensor W8A8 needs careful handling of activation outliers. Granularity alone is not enough.**"

【补充】group size 的选择是"精度–开销"的曲线：AWQ/GPTQ 的默认是 **g=128**，端侧会用 g=32。为什么不是越小越好？因为每组要存一个 FP16 scale（+ 一个 zero point），g=128 时元数据开销是

$16/128 = 0.125$ bit/weight（相对 4 bit 是 3%），g=32 时是 0.5 bit/weight（**12.5%**，相当于把 4 bit 变成 4.5 bit）。

16.4 真正的敌人：激活离群值

L16 用三页建立这个问题：

页	内容
理论	"**Uniform quantization assumes most values fit reasonably well into the same range**"；"A rare large value **expands the scale for everyone else**"；"Most small values then use only a **tiny fraction** of available integer levels"
证据	引 LLM.int8()："**Quantizing activations can cause severe degradation as models grow**"
对比	"**Activations: strong outliers；Weights: comparatively smooth**"（引 SmoothQuant, ICML 2023）

配文一句点睛："**An outlier may dominate the quantization scale while carrying little probability mass.**"

【推断】这句话其实是一个"信息论"论证：离群值占了 scale，导致**大部分正常值被压到少数几个量化格点上**，有效比特数远小于名义比特数。假设 99.9% 的值在 $[-1, 1]$，但有一个 100 的离群值，则 $S = 100/127 \approx 0.787$，$[-1, 1]$ 之间只剩 3 个格点——**INT8 实际只有 1.5 bit 分辨率**。

16.5 SmoothQuant：把难度从激活搬到权重

16.5.1 恒等变换

L16 的核心技巧（原文）：

$$Y = XW = (XD^{-1})(DW)$$

$D = \mathrm{diag}(s)$ 是正对角矩阵。图注："Smoothed activations"（XD^{-1}）与 "Compensated weights"（DW），以及一句关键提示：

> "The factors D^{-1} and D cancel in full precision."

也就是说：**这个变换在数学上什么也没改，只是在量化之前改变了"两个操作数各自的动态范围"**。

16.5.2 怎么选 s：α 迁移公式

L16 给了标定量的定义与公式（原文）：

$$a_j = \max_t |X_{tj}|, \qquad w_j = \max_k |W_{jk}|$$

$$s_j = \frac{a_j^{\alpha}}{w_j^{1-\alpha}}$$

配文两条：

> "a_j … from **calibration** and weights" "**α = 0: favor weight equalization；α = 1: favor activation smoothing**" "**Intermediate α balances the quantization difficulty for the two operands.**"

以及一句防止误解的提醒（原文）：

> "**α is not an optimization learning rate; it controls how aggressively activation range moves into weights.**"

【补充】为什么这个公式能把难度"搬过去"？设迁移后两个操作数的动态范围分别为：

$$a'_j = \frac{a_j}{s_j} = a_j^{1-\alpha} w_j^{\alpha}, \qquad w'_j = w_j s_j = a_j^{\alpha} w_j^{1-\alpha}$$

当 $\alpha = 0.5$ 时，$a'_j = w'_j = \sqrt{a_j w_j}$——**两者的范围被拉到同一个几何平均值上**。这就是"balance"的精确含义。

16.5.3 讲义给的两通道算例

条件：$\alpha = 0.5$，两个通道。

通道	$\max\|X_j\|$	$\max\|W_j\|$	s_j	缩放后 X/s 与 sW
1	2	4	$\sqrt{0.5} \approx 0.71$	2.82 ; 2.84
2	16	1	4	4 ; 4

【补充】手算校验（讲义的数字是四舍五入的结果）：

- 通道 1：$s = \sqrt{2/4} = 0.7071$；$X/s = 2/0.7071 = 2.828$；$sW = 4 \times 0.7071 = 2.828$ → 两者都是 2.83（讲义标 2.82/2.84，属舍入差异）。
- 通道 2：$s = \sqrt{16/1} = 4$；$X/s = 16/4 = 4$；$sW = 4 \times 1 = 4$ → 都是 4。

结论（讲义原话）："**Magnitudes are now more balanced in both X′ and W′.**" 原本 $(2, 4)$ 与 $(16, 1)$ 两组悬殊的范围，变成了 $(2.83, 2.83)$ 与 $(4, 4)$。

16.5.4 α 的取舍曲线

L16 用一页说明："**Increasing α makes activations easier to quantize but can make weights harder.**"（引 SmoothQuant 论文的 OPT-175B α sweep 曲线），配文一句方法论：

> "Use representative calibration data and validate α against model accuracy and runtime."

16.5.5 工程落地：离线准备 + 在线服务

L16 给了完整的流水线（原文）：

```
Offline:  Calibrate X  →  Compute s_j  →  Rescale weights
Online :  Quantize input  →  INT8 GEMM  →  Dequant / other ops
```

配文："**The rescaling can be absorbed into existing layers, limiting online overhead.**"

16.5.6 硬件视角：只量化"贵的算子"

L16 的提醒（原文）：

> "SmoothQuant targets **matrix multiplications**, so not every operation must become INT8."
> "**Compute-heavy GEMMs benefit from INT8 kernels, while normalization and softmax may stay higher precision.**"

【补充】这与第 9 章 L9 的"算子分级"表完全一致：**GEMM 可以低精度，归约（softmax/LN）要 FP32，范围敏感的（exp/log/loss）要 FP32**。两讲在不同章节给了同一张结论表，值得对照记。

16.5.7 效果与局限

页	内容
精度	"**SmoothQuant's W8A8 accuracy stays close to FP16** in the paper's evaluated OPT models"
收益	"W8A8 can improve serving efficiency **when kernels and memory placement exploit the lower precision**"；"Lower precision can change the **required GPU count**"（OPT-66B / 175B 的 batch-size 曲线）

"Where SmoothQuant can struggle" 页列了四条（原文）：

#	局限
01	**Calibration drift**："Future workloads can have different activation outliers"
02	**Low-bit regimes**："W8A8 does not automatically solve W4 / W3 accuracy"
03	**Kernel availability**："Packed formats and quantization overhead may erase theoretical speedup"
04	**End-to-end cost**："**Memory savings do not directly imply lower user-visible latency**"

第 04 条与第 14 章那句 "Measure net serving throughput" 是同一个方法论：**报利用率/报比特数都不算，要报端到端**。

16.6 讲义没展开、但作业要考的三块

HW3 Part 2 覆盖了 L16 之外的三块内容，这里补齐（【补充】，题面口径来自 HW3 notebook 原文）。

16.6.1 GPTQ：把量化变成逐列的最小二乘

核心思想（Optimal Brain Quantization 的近似）：**逐列量化 W，每量化一列就把误差用 Hessian 的逆传播到尚未量化的列上**：

$$\min_{\hat{W}} \|XW - X\hat{W}\|_2^2 \implies \hat{w}_q = \text{quant}(w_q), \quad \delta = \frac{w_q - \hat{w}_q}{[H^{-1}]_{qq}}, \quad w_{j>q} \mathrel{-}= \delta\,[H^{-1}]_{qj}$$

其中 $H = 2XX^\top$ 来自校准数据。要点：① 需要校准集；② 用 Cholesky 分解避免显式求逆；③ 逐层独立，可以很快（7B 模型单卡几十分钟量级）。**GPTQ 与 AWQ 是同一战场的两条路线：GPTQ 靠"误差补偿"，AWQ 靠"保护显著通道"。**

16.6.2 AWQ：保护 1% 的显著通道

HW3 2.2 节给的定义（原文公式）：

$$a_j = \text{mean}_n |X_{nj}|, \qquad u_j = \max(a_j^\alpha, 10^{-4}), \qquad s_j = \frac{u_j}{\sqrt{\max(u)\min(u)}}$$

以及恒等式：

$$XW^\top = (X/s)(W \odot s)^\top$$

步骤	内容
观察	权重里只有约 **1%** 的通道（对应激活的显著通道）重要，按"激活幅度"而不是"权重幅度"来识别
手段	对显著通道**放大权重**（$W \odot s$），使其在分组量化里占更多格点；同时**缩小对应激活**（X/s）保持数学等价
搜索	用**校准集输出 MSE**（不是权重 MSE）搜 α 与 clip 比例 ρ
结果	AWQ INT4 在 Llama-2-7B 上 PPL 5.60，RTN INT4 是 5.73，FP16 是 5.47（HW3 2.6 表）

注意 AWQ 与 SmoothQuant 的**方向相反**：SmoothQuant 把激活的难度搬到权重（为了 W8A8 能算 INT8），AWQ 把权重的精度"预算"往显著通道倾斜（为了 W4A16 不掉点）。**同样的 X/s、$W \odot s$ 恒等式，两种用法。**

16.6.3 FP8 与 KV 量化

技术	一句话	现状【推断】
FP8 (E4M3/E5M2)	8 位浮点，保留指数（范围），精度靠 per-tensor/per-block scale	H100 起有原生 FP8 GEMM；训练与推理都在用，是"比 INT8 更好训"的低比特格式

技术	一句话	现状【推断】
KV cache 量化	把 KV 从 BF16 压到 FP8/INT8/INT4，直接放大并发数	收益直接（KV 是显存大头），但**长上下文下误差累积明显**，通常只量化 K 或只量化 V、或保留 outlier 通道
weight-only INT4 + group 128	目前端侧/单卡部署的主流	AWQ/GPTQ 的事实标准配置

16.7 一个完整的量化吞吐算例（HW3 2.6 的解法）

题面：Llama-2-7B，$N = 7.0 \times 10^9$ 权重，带宽 1.0×10^{12} B/s，每 token 读一遍权重与组元数据，batch=1。

存储公式（每组 g 个权重）：

$$\text{bytes/group} = \frac{g \cdot b}{8} + \underbrace{2}_{\text{fp16 scale}} + \underbrace{\frac{b}{8}}_{\text{b-bit zero point}}$$

逐项计算：

方法	每组字节（g=128）	每权重字节	总存储	吞吐上界	相对 FP16
FP16	—	2.0	14.0 GB	71.4 tok/s	1.00×
RTN INT4	$64 + 2 + 0.5 = 66.5$	0.5195	3.637 GB	275.0 tok/s	3.85×
AWQ INT4	同上	0.5195	3.637 GB	275.0 tok/s	3.85×
AWQ INT3	$48 + 2 + 0.375 = 50.375$	0.39355	2.755 GB	363.0 tok/s	5.08×

三条解释题的标准答案：

1. **AWQ INT4 vs RTN INT4 同存储不同精度**：两者字节数完全相同（都是 4 bit + g=128 元数据），差别**只在量化误差的分布**——AWQ 用通道缩放把格点预算往显著通道倾斜，所以 PPL 5.60 < 5.73。**同成本、更高质量，这才是 AWQ 的价值**，不是"更快"。
2. **AWQ INT4 → INT3 的取舍**：存储从 3.64 GB 降到 2.76 GB（−24%），吞吐上限从 275 涨到 363（+32%），但 **PPL 从 5.60 崩到 6.24**（+11%）。这是典型的"过了一条悬崖"——**3 bit 已经接近该架构的可压缩极限**。
3. **理论上界为何达不到**：① kernel 要把 INT4 反量化成 FP16 再做 GEMM，这部分**不是带宽**，是算力与指令开销；② 组元数据要额外读、要对齐/padding；③ 实际 batch>1 时权重读取被摊薄，瓶颈会转移到别处；④ 显存带宽的"标称值"与"可持续值"差 10–20%；⑤ 采样、KV 读取、kernel launch 都有固定开销。

【推断】最后一条最重要：**这批数字是"上界"，它的用途是判断"值不值得做"，不是预测线上 QPS**。若实测只有上界的 40–60%，属于正常范围。

16.8 L16 的三条总结

讲义总结页（原文）：

> - **Quantization**: Maps continuous values to fewer representable levels; **clipping and rounding matter**.
> - **Granularity**: Controls whether **outliers affect many values or only one group / channel**.
> - **SmoothQuant**: Rescales X and W equivalently to **enable accurate W8A8** inference.

以及 panel 页给的 Murakkab（OSDI 2026）数字，作为"下一层系统问题"的引子：

> "Reported gains: **up to 2.8× less GPU use, 3.7× less energy, and 4.3× lower cost**（vs. LangGraph）." "**Decouple application logic from execution choices**; jointly meet quality and latency SLOs."

第 17 章 MoE 的效率（L17）

17.1 稠密扩展撞墙

L17 用一张"历史成本表"开场（原文，讲义注明为"historical illustrative examples"）：

模型	Model size	Accelerators	Train time
BLOOM	176B	384 A100	115 d
OPT	175B	992 A100	56 d
MT-NLG	530B	**2,200 A100**	60 d
PaLM	540B	**6,144 TPU v4**	57 d

这张表的读法是：**参数量涨 3 倍，卡数涨 3–6 倍，时间不变**。也就是说稠密扩展的成本是**线性甚至超线性**的，而收益（质量）是对数的。MoE 的动机就在这里（原文标题）："**Why do we need sparse computation?**"——"**Dense scaling expands both parameters and computation.**"

17.2 架构：把 FFN 换成"路由的多个 FFN"

L17 的对照（原文）：

	Dense transformer block	Sparse MoE block
结构	一个 Dense FFN	Top-2 router → E1..E4
计算	"**Every token runs the full FFN**"	"**Each token runs selected experts only**"
扩展性	"FLOPs grow with FFN width"	"**Expert count can grow at fixed top-k**"

一句总结："**MoE decouples parameter capacity from per-token FFN FLOPs.**"

17.2.1 路由的数学

L17 给的公式（原文）：

$$y = \sum_e g_e(h) \cdot E_e(h)$$

符号	含义
h	"Hidden state of the current token (input vector)"
$g_e(h)$	"**Gating weight** assigned to Expert e by router"
$E_e(h)$	"Output of Expert e given input h"

配文："**The router introduces dynamic, input-dependent work.**"

17.2.2 容量 ≠ 计算：Mixtral 的例子

L17 给的数字（原文）：

指标	值
Total parameters	**141B**
Activated parameters / token	**39B**
Active fraction	**39/141 ≈ 28%**

配了一句关键提醒："**But the full model weights still require placement, storage, and transfer bandwidth.**"

【推断】这句话是 MoE 系统问题的全部根源：**FLOPs 省了 72%，但显存一个字节都没省**。所以 MoE 的部署成本由"总参数"决定，而计算成本由"激活参数"决定——两者脱钩，导致 MoE 是"显存换质量"的技术，不是"省钱"的技术。

17.3 四类系统代价

L17 用一页列了 MoE 引入的系统复杂度（原文）：

代价	原文
Expert weights	"Many parameters still need memory capacity"
Token routing	"Token destinations differ across requests"
Communication	"**Expert parallelism moves activations between GPUs**"
Imbalance	"**The busiest expert can determine the iteration time**"

以及一句总纲："**Effective throughput depends on memory, networking, load balance, and scheduling.**"

17.3.1 演进史（讲义时间轴）

年份	工作	关键词
2017	Sparsely-gated MoE	Conditional computation
2021	GShard / Switch	Scalable routing and stability
2022	DeepSpeed-MoE	Inference & multidimensional parallelism
2023	Tutel	**Adaptive execution at scale**
2024	DeepSeekMoE	**Fine-grained and shared experts**

配文："**More experts require both better routing and better distributed execution.**"

17.3.2 四个设计旋钮

L17 的表（原文）：

Design knob	Example choices	Systems tradeoff
Expert granularity	Few large / many small	"Compute efficiency vs. specialization"
Shared experts	Always active / only routed	"Common knowledge vs. sparse capacity"
Routing	Top-1 / Top-2 / **expert choice**	"**Quality vs. imbalance**"
Layer placement	Every layer / selected layers	"Compute and network frequency"

17.3.3 两个具体架构

架构	关键设计	数字
DeepSeekMoE（ACL 2024）	细粒度 routed experts + **always-active shared experts**；"Router · top-6"，"**6 of 64 routed**"，另加 "**2 shared experts**"	**16.4B total，~2.8B activated per token**

架构	关键设计	数字
DeepSpeed-MoE（ICML 2022）	**PR-MoE（pyramid）**：早期层少专家、后期层多专家；再加一条 residual dense path	讲义图：early 2 专家 / middle 4 / late 8

配文（DeepSeekMoE）："**Both branches process each token; only the routed branch selects a subset.**"、"Memory capacity and dispatch cost remain critical even when active FLOPs are small."

【推断】DeepSeekMoE 的"细粒度 + 共享专家"组合，本质是把"**通用能力**"与"**专门能力**"在参数上解耦：共享专家永远激活（保证基础能力稳定、且是所有 token 的公共路径），细粒度 routed 专家负责稀疏的专门化。这也解释了为什么它能用 6/64 的激活率还保持质量。

17.4 负载不均：路由的"原罪"

L17 用四页把这个问题讲得很透。

17.4.1 Top-k 不等于均匀

> "**Even with k fixed, the number of tokens per expert is not fixed.**"

图例：4 个专家上 16 个 token-to-expert assignment，实际分布是 **7 / 4 / 2 / 3**。配文一句判决：

> "**Step time often follows the slowest expert, not the average load.**"

17.4.2 一个课堂小测

题目（原文）：8 个 token 用 top-1 路由到 4 个专家，assignment 为 `T1→E1, T2→E1, T3→E2, T4→E1, T5→E3, T6→E1, T7→E4, T8→E2`。三个问题：① 各专家多少 token？② 若每个专家占一张 GPU，哪张是瓶颈？③ 不均衡会怎样影响 TTFT / 每 token 延迟？

答案（讲义下一页）：

专家	Token 数	占比
E1	**4**	4/8
E2	2	2/8
E3	1	1/8
E4	1	1/8

配文："**For fixed expert placement, E1 is the straggler**；capacity limits or routing changes trade flexibility for balance."

【推断】第 ③ 问的答案是"**两者都影响，但机制不同**"：对 TTFT，prefill 阶段所有 token 一起来，E1 的排队时间直接进 TTFT；对 TPOT，decode 阶段每步都有新 token 落到 E1，E1 的负载决定了**每步的最慢路径**，于是 TPOT 的 P99 被 E1 拉长。而 MoE 的 decode 还有个额外麻烦：**每步的 token 数太少，expert 的 GEMM 很小，算术强度低**（回到 2.5 节）。

17.4.3 Token-choice vs Expert-choice

L17 用一页图对照两种路由范式（同样 6 tokens、4 experts、12 个 assignment）：

范式	规则	各专家负载	各 token 得到的专家数
Token-choice（top-2 / token）	每个 token 选 2 个专家	**6 / 4 / 2 / 0**	每个 token 恰好 2 个
Expert-choice（top-3 / expert）	每个专家选 3 个 token	**3 / 3 / 3 / 3**	T1 得到 4 个；T4、T6 只得到 1 个

配文（原文）：

> "**Fixed experts/token can overload GPUs; fixed tokens/expert makes load predictable.**"
> （expert-choice 来源：Zhou et al., NeurIPS 2022）

这是本章最漂亮的一组对照：**两种范式的"总 assignment 数"完全相同（都是 12），但一个把不均衡放在专家侧，另一个放在 token 侧**。Expert-choice 让**系统负载完全可预测**（每个专家固定 3 个 token，可以精确规划 batch），代价是**不同 token 得到的算力不同**（T1 四个专家、T4 一个）——而且这个差异是**不可微的**，训练时需要额外技巧。

17.4.4 训练目标的四件套

L17 的表（原文）：

目标	作用
Task loss	"Optimize predictions, not only uniform workloads"
Auxiliary load balance	"Encourage more even expert utilization"
Router z-loss	"**Regularize large routing logits for stability**"
Capacity/drop policy	"Decide what happens when an expert overflows"

配文："**Balancing losses encourage better routing; real batch sizes and hardware still vary.**"

【补充】router z-loss 的形式是 $\frac{1}{B}\sum_i \left(\log \sum_j e^{z_{ij}}\right)^2$，惩罚 logits 的绝对值过大（防止 softmax 饱和导致梯度消失/数值不稳）。capacity factor 则是"每个专家最多收 $\mathbf{capacity} = \mathbf{capacity_factor} \times \frac{\mathbf{tokens}}{\mathbf{experts}}$ 个 token"，超出的 token **被丢弃**（dropped token 直接走残差），这是"训练效率 vs 质量"的又一处硬取舍。

17.5 专家并行（EP）：把路由变成通信问题

17.5.1 三种可切分的东西

L17 的图（原文标注）：

方案	每个 GPU 上放什么	得到什么
Data parallelism	"Replica of **all experts** on each GPU"	"More independent batches"
Tensor parallelism	"**Partition one expert** across GPUs"（E1–A/E1–B/E1–C）	"Parallelize large expert matmuls"
Expert parallelism	"**Place different experts on different GPUs**"（E1/E2/E3）	"**Route tokens across devices**"

配文："**EP saves per–GPU weight memory but adds token exchanges.**"

17.5.2 两次 All–to–All

L17 的经典 EP 流水（原文五步）：

```
1. Route & group tokens
2. All-to-all dispatch      ← 第一次集合通信
3. Expert FFN compute
4. All-to-all combine       ← 第二次集合通信
5. Restore token order
```

配文："**Forward requires one dispatch collective and one return collective.**"、"**Backward training carries corresponding gradient communication.**"（所以训练时是 **4 次** All–to–All：前向 2 次 + 反向 2 次）。

以及一句效率警告："**All–to–all traffic, packing/unpacking, and synchronization can dominate for small experts.**"

17.5.3 一个可用的延迟模型

L17 给了一个非常实用的成本分解（原文）：

$$T_{\text{layer}} \approx T_{\text{route}} + T_{\text{pack}} + T_{\text{dispatch}} + \max_{e}\left(T_{\text{expert},e}\right) + T_{\text{combine}}$$

项	归属	备注
$T_{\text{route}}, T_{\text{pack}}$	"CPU/GPU launch + layout work"	常被忽略，但小 batch 时是主导
$T_{\text{dispatch}}, T_{\text{combine}}$	"Bandwidth + message latency"	All–to–All
$\max_e T_{\text{expert},e}$	"Load imbalance and kernel efficiency"	用 max 不是 mean

配文一句诊断方法："**If the collective dominates, making FFN faster may barely improve end-to-end latency.**"

【推断】这个模型的价值在于**先定位再优化**：如果 T_{dispatch} 占 60%，那优化 expert 的 GEMM 毫无意义；如果 $\max_e T_{\text{expert},e}$ 是均值的 3 倍，那应该先做负载均衡（EPLB/Tutel）。这与第 14 章"先算碎片账再谈吞吐"是同一种工程纪律。

17.5.4 拓扑：节点内 vs 节点间

L17 的图把两个节点画出来，标注："**An inter-node token exchange is not the same as an intra-node exchange**"，以及 "fast local GPU interconnect" 与 "inter-node fabric" 的对比。配文：

> "**Co-locate frequently communicating experts when routing patterns justify it.**"

17.5.5 四种并行组合的适用条件

L17 的表（原文）：

Strategy	When plausible	Primary cost
EP-heavy	"Experts fit on one GPU"	"Low per-GPU memory; **token all-to-all**"
TP + EP	"Expert matmuls are large"	"Add intra-expert collectives"
DP + EP	"Many requests / high batch"	"Replicas improve throughput at memory cost"
Pipeline + EP	"Deep execution graph"	"Higher utilization, more stage coordination"

配文："**Expert size, batch size, and interconnect bandwidth determine the best layout.**"

17.6 Tutel：让并行度"随负载变"

L17 用三页讲 Tutel（MLSys 2023），核心观察（原文）：

> "**Tutel observation: expert loads change over time**" "The number of tokens an expert needs to process is **not stationary**." 变化来源：Token distribution、Expert capacity / batch、Layer-specific routing "**A static layout may be good now and bad later.**"

解决方案是**运行时重配置**：

情形	策略
Case A: small per-expert batch	"Favor layouts with **less communication and launch overhead**"
Case B: large per-expert batch	"**Exploit more parallel GPU compute** when enough tokens accumulate"

以及**通信–计算 overlap**：把 token 切成 chunk，让一个 chunk 在算、另一个在传：

```
Without overlap:  Dispatch → Compute → Combine
With overlap:     D1 D2 C1 R1 C2 R2 ...       (D: dispatch, C: compute, R: return)
```

配文："**The best pipeline depth depends on batch size, expert load, and communication.**"

【推断】Tutel 与 dLoRA（第 11 章）是同一个思想的两次出现：**"静态放置 + 动态负载"必然失配，解法是让放置/并行度可运行时调整**。这也解释了为什么 MoE 的 serving 系统比稠密模型复杂一个量级——它同时有"路由的动态性"与"并行度的动态性"两层不确定性。

17.7 怎么评价一个 MoE 系统

L17 给了五个维度（原文），这张表可以直接当评审 checklist：

维度	看什么
Quality	"task accuracy / validation loss"
Latency	"prefill, decode and **end-to-end tails**"
Throughput	"tokens/s and requests/s"
Resource	"**peak HBM and interconnect traffic**"
Balance	"**per-expert load variance and stragglers**"

配文："**Optimize service goodput under quality and SLO constraints.**"

17.8 L17 的全课程串联页

L17 末尾有一页 "Connections across CS 498"，把后半程五个问题并列，这页值得原样记下来：

讲	问题
L14 · PagedAttention	Where should **KV be placed** to avoid memory waste?
L15 · Adaptive KV	Which **token states should stay resident**?
L16 · Quantization	Which computations/weights can use **lower precision**?
L17 · MoE	**Which experts execute and on which GPUs**?
L18 · RAG Systems	How do **retrieved contexts and cached states** interact?

配文："**Model architecture and resource orchestration must be co-designed.**"

第 18 章 RAG 系统（L18）

> L18 是课程的最后一讲，结构是"RAG 基础 → CacheBlend（EuroSys'25 Best Paper）→ Mooncake（FAST'25 Best Paper）panel"。它把前四讲的 KV 问题（放哪、留谁、几位、谁算）收束成一个"KV 中心"的系统观。

18.1 为什么需要 RAG

L18 用一页列出三条理由（原文）：

理由	原文
Freshness	"Weights may not reflect a **new Canvas announcement**"
Provenance	"We need a **verifiable knowledge source**"
Modularity	"Retrieve updates **without retraining** the LLM"

配了一句非常克制的限定：

> "**RAG makes information modular, but does not guarantee correctness.**"

18.1.1 流程

```
User query → Retriever → Top-k passages → LLM prefill → Answer
                ↑
          Knowledge corpus
```

18.1.2 检索：一个内积

L18 的两步（原文）：

1. "**Encode the query and knowledge chunks**"：query text → Encoder（e.g., BERT）→ $q = [0.12, -0.04, \ldots, 0.75]$
2. "**Compare document embeddings**"：$\text{score}(q, d_i) = q^\top d_i$

【补充】实际系统是在做 **ANN（approximate nearest neighbor）**检索，而不是全量内积：HNSW/IVF/PQ 之类的索引把 $O(N)$ 的扫描降到 $O(\log N)$ 或 $O(\sqrt{N})$。这一层在 CS 498 里只给了"内积"这一句，因为课程关心的是**LLM 侧的 prefill 成本**，不是向量检索本身的算法（那属于信息检索课）。

18.1.3 一个具体的 prompt 构造例子

L18 给的例子（原文）：

文档候选	Score
Prior CS 498 course overview	0.61
Fall 2026 assignment announcement	**0.91**
General project guidelines	0.28

拼出的 prompt：

```
SYSTEM:  Answer using the retrieved course page.
USER:    When is Assignment 3 due?
RETRIEVED PASSAGE: Assignment 3 is due Dec 7. Only 4-credit students submit it.
LLM RESPONSE: December 7, for 4-credit students.
```

配文："The retriever **adds evidence to the prompt**. It **does not change the model weights**."

【推断】这个例子同时暴露了 RAG 的成本结构：**retrieved passage 是要 prefill 的**。检索回来的 5 篇文档 × 1000 token = 5000 token 的 prefill，比用户问题（10 token）贵 500 倍。**RAG 的成本几乎全在"把检索到的文档喂进去"这一步**——这就是 CacheBlend 存在的理由。

18.2 RAG 的延迟分解

L18 用一页把 RAG 的时间轴拆开（原文）：

```
Retrieve → Prompt assembly → Prefill → Decode
```

配文两句：

> "**Time-To-First-Token (TTFT) includes retrieval, queueing, prompt processing, and the first decode step.**" "**Long retrieved contexts often make prefill more expensive.**"

以及本章的 thrust："**How to improve the latency when retrieving many documents?**"

【补充】把它写成式子：

$$\mathrm{TTFT}_{RAG} = T_{retrieve} + T_{queue} + T_{prefill}(\mathrm{system} + \sum_i \mathrm{doc}_i + q) + T^{(1)}_{decode}$$

四个可优化项对应四种技术：① $T_{retrieve}$ → ANN 索引、缓存热门 query；② T_{queue} → 调度（L13）；③ $T_{prefill}$ → **KV 复用（本节）**；④ $T^{(1)}_{decode}$ → 模型/kernel 优化。CacheBlend 动的是第 ③ 项，而且它是**唯一一项与"文档数"成正比**的，所以收益最大。

18.3 前缀复用的边界：SGLang 能做到哪一步

L18 用一页 SGLang 的例子说明"共享前缀"的局限（原文）：

```
Request 1:  System | Document A | Document B | Question
Request 2:  System | Document B | Document A | Question
```

配文："**Challenge: After these requests diverge, which KV states can still be shared safely?**"

【推断】这个例子非常精准地打在了 radix tree 的痛点上：两个请求共享 `System`（可复用），但从 `Document A` vs `Document B` 开始分叉，**分叉之后的所有 KV 都不能直接复用**。而 RAG 的真实负载恰恰是"**同一批文档、不同顺序、不同子集**"——所以前缀复用只能吃到 `System` 那一小段。

18.4 为什么"独立编码的 chunk KV"不能直接拼

这是全章最关键的一页。L18 用两行对照图说明（原文）：

```
Full prefill:        A → B → Q        (A、B、Q 之间有 cross-chunk attention)
Cached separately:   A | B | Q        (cross-chunk attention 缺失)
```

配文：

> "A chunk encoded alone **does not attend to earlier chunks.**" "**Blind concatenation can be fast but produce incorrect attention states (poor accuracy).**"

【推断】把这件事讲透：Transformer 的第 l 层输出依赖第 $l-1$ 层的**全部**上下文。如果 chunk B 的 KV 是在"没有 A"的情况下算出来的，那么 B 的每一层隐状态都缺少 A 的信息；把这些 KV 拼到 A 后面做 attention，**得到的不是"full prefill 的近似"，而是一个"从未存在过的状态"**。误差会随层数累积（每层都错一点），所以**盲目拼接的精度损失可能非常大**——这是 CacheBlend 论文的核心 motivation。

18.5 三条路线

L18 的对照表（原文）：

路线	做法	代价
Full recompute	"Prefill every input token for the full context at a **higher compute cost**"	慢（但正确）
Full KV reuse	"Reuse independent chunk KV **quickly**, but **miss cross-chunk dependencies**"	快（但错）
CacheBlend	"Reuse most chunk KV and **recompute a small selected set** to recover quality"	折中

这是一条典型的"**正确性–延迟**"帕累托前沿，CacheBlend 的贡献是**找到了前沿上的一个好点**。

18.6 CacheBlend：选择性重算

18.6.1 核心问题

L18 的图注（原文）：

> "Each passage (e.g., document) can have a **precomputed KV cache**." "**Challenge: Recompute only the tokens whose cached KV is most context-sensitive.**"

18.6.2 判据：KV deviation（KV 偏差）

L18 用一页说明"哪些 token 要重算"（原文）：

> "**Which cached tokens would differ most from a full prefill?**" 图例：10 个 chunk token，其中若干被标为 "**High-deviation tokens**" "KV(full context) differs substantially from KV(cached chunk)."

【推断】这个判据是可计算的：对候选 token i，比较"用完整上下文算出的 K_i, V_i"与"缓存里的 K_i, V_i"，偏差大的就是要重算的。但注意——**要算"完整上下文的 KV"就已经等于做了一次 prefill**，所以不能直接算。CacheBlend 的巧妙之处在下一条。

18.6.3 关键假设：偏差排名跨层稳定

L18 有一页专门讲这个 hypothesis（原文）：

> "**Hypothesis: High-deviation token rankings persist across layers**" "**Observation**: High-deviation token rankings are **strongly correlated across nearby layers**." "**Why it matters**: CacheBlend can **reuse earlier importance estimates** to progressively filter tokens for selective recomputation."

这页是全章的"技术支点"：**因为排名稳定，所以可以在浅层用较少的信息估计出"哪些 token 重要"，然后在深层只重算这些小集合。**

18.6.4 渐进式过滤

L18 用两页图演示（原文）：

```
Layer 1:     1 2 3 4 5 6 7 8 9 10      ← 重算较大的候选集（橙色多）
Layer 2:     1 2 3 4 5 6 7 8 9 10      ← 集合收窄
Layer 3:     1 2 3 4 5 6 7 8 9 10      ← 继续收窄
Later:       1 2 3 4 5 6 7 8 9 10      ← 只剩少数几个
```

配文："**Recompute a larger candidate set in early layers, then filter the high-deviation tokens across layers.**"、"**Selective recompute reduces work while respecting cross-chunk dependencies.**"

18.6.5 与 KV 加载 overlap

L18 还给了一个系统层的优化（原文）：

> "**How much to recompute? — Overlapping KV loading with recomp.**" "**Load cached KVs from a lower memory tier while the GPU recomputes the selected token states.**" 图注："**Transfer hidden by useful computation**"

【推断】这条是"用计算遮盖通信"的又一次出现（第 5 章的 Centauri、第 17 章的 Tutel overlap 都是同一招）。它的前提是**重算与加载可以并行**——因为它们操作的是不同的 token 集合，没有数据依赖。

18.6.6 "One Slide" 总结

L18 的图（原文）：

行	内容
Cached K/V	1..8 全有（Light = cached）
Refresh KV	只刷新 2、5、7（Orange = refreshed）
Merged K/V	1..8 全在（Dark = reused），"**Attention sees all K/V**"

配文："Across layers, CacheBlend **narrows the high-deviation candidate set**."

18.6.7 收益

L18 的结果页（原文）：

> "**2.2–3.3× lower TTFT and 2.8–5× higher throughput** vs full KV recomputation."

18.6.8 什么时候不灵

L18 用一页列了四条（原文），这是很有价值的"失败条件"清单：

条件	原文
Low reuse	"Few chunks recur, so precomputed KV offers little benefit"
High dependency	"More cross-chunk dependencies **increase the amount of recomputation**"
Cache misses	"**Moving cached KVs from slower tiers can dominate TTFT**"
Quality risk	"**Approximate fusion must be validated on downstream tasks**"

【推断】第四条最值得警惕：CacheBlend 是**近似**方法（它算出的 attention 状态与 full prefill 不同），所以"TTFT 降 3×"必须与"任务精度掉了多少"一起报。这也是所有 KV 复用/驱逐类工作（L15 的 H2O、L14 的 COW）的共同评审标准。

18.7 Mooncake：把 KV 变成系统的中心

L18 的 panel 页给了 Mooncake（FAST'25 Best Paper）的架构（原文标注）：

组件	作用
Conductor	负责三件事：**Cache-aware prefill placement**、**KV-cache balancing**、**Load-aware decode placement**
Prefill GPU pool	"Prefill computes prompt states"，受 **TTFT constraint** 约束
Decode GPU pool	"Decode generates output tokens"，受 **TBT / TPOT constraint** 约束
Mooncake Store	"**distributed KV**"，跨 **CPU DRAM · SSD · RDMA** 的传输

配文一句总纲："**Trade more storage for less repeated prefill computation.**"

以及一个开放问题（starter question）：

> "**How should a cache-aware scheduler handle dependent agent steps (e.g., multi-agent)?**"

【推断】Mooncake 与 DistServe（L14 panel）的差别值得说清：

系统	切分维度	核心资源
DistServe	按**阶段**切（prefill 池 / decode 池）	GPU 算力
Mooncake	按阶段切 **+ KV 分层存储**（DRAM/SSD/RDMA）	**KV 存储与搬运**

Mooncake 的"KVCache-centric"意味着：**调度器的第一目标不是"哪张卡空"，而是"这个请求的 KV 在哪一层、搬过来要多久"**。这把 L14（KV 放哪）、L15（留哪些）、L18（哪些要重算）三个问题合并成了一个**全局放置问题**——也就是 L18 最后那张"四个 KV 决策"表的收束。

18.8 四个 KV 决策（L18 的收束页）

L18 的 "Bridge to panel: four KV-cache decisions"（原文）：

讲	问题
L14 · PagedAttention	**Where** should each KV block **live on the GPU**?
L15 · Adaptive KV	**Which** past KV tokens should be **retained**?
L18 · CacheBlend	**Which cached knowledge tokens should be recomputed**?
Mooncake · FAST'25	**Which GPU / cache tier** should serve the next request?

配文："KV placement, retention, fusion, and scheduling should be co-designed."

这张表就是整门课推理部分的"四问"，也是我认为最值得背下来的一页：**位置（where）、保留（which）、融合（recompute）、调度（who serves）**。

第 19 章 与已有笔记的衔接

19.1 AIInfra M2（分布式训练）：公式在那边，直觉在这边

主题	M2 有什么	CS 498 补什么	用法
集合通信	原语语义、`dist.all_reduce` 代码、通信量推导	ring 的**手写实现**（HW1 Q3）、α/β/γ 模型、Centauri 的 overlap	先用 M2 建立公式，再用 HW1 落到代码
DP / ZeRO	16Ψ 账本、ZeRO-1/2/3、FSDP	PS 的历史地位、BSP/异步/有界一致性	M2 讲"今天怎么做"，CS 498 讲"为什么曾经那样做"
TP	Megatron 分块公式	HW2 1.3 的通信量算例、prefill/decode 的 512× 差异	CS 498 的算例比 M2 的公式更适合做面试题
PP	bubble 公式、1F1B、Interleaved	L7 的逐帧时间轴、m 的双重约束	两边对照读，公式与直觉互补
MoE	All-to-All、aux-loss-free	L17 的 T_{layer} 分解模型、Tutel	CS 498 的成本模型更工程化

19.2 AIInfra M3（推理优化）：实现在那，演进史在这

主题	M3 有什么	CS 498 补什么
PagedAttention	vLLM 源码级调度、block manager	**OS 分页的类比**、三种浪费的定义、COW 的来源
Continuous batching	调度器实现、chunked prefill 代码	**Orca 的演进动机**（早完成/晚到达两个病）
量化	GPTQ/AWQ/FP8 kernel	**从 scale/zero-point 起讲的完整推导**（L16 是更好的教材）
Speculative decoding	接受率公式、tree attention	HW3 的三种 drafter（EAGLE-3/DFlash/DSpark）与置信度前缀
KV 压缩	H2O/StreamingLLM/SnapKV 的实现	**attention sink 的机制解释**、四维对照表
PD 解耦	DistServe 的 goodput 模型	L14/L18 把它与 KV 放置统一成"四问"

19.3 与 CS336、CS144、CS149 的关系

课程	它教什么	对 CS 498 的作用	典型接口
CS336（从零构建 LM）	手写 attention、tokenizer、训练循环、并行原语	提供"每个算子长什么样"的底层能力；CS 498 则问"这个算子该跑在哪张卡上、值多少钱"	CS336 的 `flash_attention` 实现 ↔ CS 498 L2 的 AI 分析
CS144（计算机网络）	拥塞控制、可靠传输、RPC	解释"为什么集合通信不是免费的"：IB 的 MTU、队头阻塞、incast	CS144 的 RTT/带宽 ↔ CS 498 的 α/β 模型
CS149（并行计算）	GPU 架构、tiling、数据并行思维、roofline	**直接的前置**：CS 498 的 2.5 节几乎就是 CS149 的 roofline 应用	CS149 的 `attention_fused.py` ↔ L2 的 FlashAttention
CS425（分布式系统）	一致性、复制、故障模型	解释 PS 的 staleness 谱系与"collectives are not fault-tolerant"	CS425 的 BSP ↔ CS 498 L4

一句话总结：**CS149 给你"一块卡怎么跑满"，CS336 给你"一个模型怎么搭"，CS425/CS144 给你"多台机器怎么说话"，CS 498 把三者拼成"一个系统怎么挣钱"**。

19.4 三门课的"同一道题"对照

以"KV cache 太大怎么办"为例，看三份材料的回答方式差异：

材料	回答方式
AIInfra M3	给 vLLM 的 block manager 代码、`gpu_memory_utilization` 参数、PagedAttention kernel 的读法
CS336	给"KV cache 的 shape 是 `[B, H_kv, T, D]`"、手工实现 paged 索引的可能写法
CS 498	给"三种浪费 → 分页 → 引用计数 → H2O/StreamingLLM → CacheBlend → Mooncake"的**演进链**，每一步都问"它动了三角的哪条边"

这也解释了我为什么要把三份笔记分开写：**它们回答的是不同层次的问题**，混在一起会互相干扰。

第 20 章 复习清单与自测题

20.1 逐讲"能否复述"清单

打勾标准：能**不看笔记**讲 90 秒，且能被追问两次不崩。

讲	必须能复述的三个点
L1	课程定位（NOT AI methods）；评分结构 15/8/16/16/15/30；efficiency/effectiveness/easy-to-use 三角
L2	RNN 的三个系统问题；KV cache 为什么没有 Q cache；FlashAttention 的 tiling + online softmax 修正因子
L3	attention 的 $(4h+3)N^2$ 与 $8N^2+8Nh$；MLP 的 1365 ops/byte 与 A10 的 208 脊点；MHA/MQA/GQA 的 KV 账本
L4	GPT-3 的 2800 GB vs 80 GB；PS 的 pull/compute/push/update；BSP→异步→有界一致性的 staleness 谱系
L5	ring AllReduce 的 $2N(P-1)/P$ 与"largely independent of P"；PS vs AllReduce 的病灶；α/β/γ
L6	列并行无前向通信、行并行要 AllReduce；GeLU 必须在 AllReduce 之前；4 次通信/层/step
L7	bubble 的 $mt_f+(p-1)(t_f+t_b)+mt_b$；1F1B 省显存不省气泡；Interleaved 用通信换气泡
L8	$n=t\times p\times d$ 与 rank 编排；$t=p=8$ 峰值的拓扑解释；三条 rule of thumb
L9	为什么不能纯 FP16（$w \gg \alpha\nabla$）；loss scaling 的 2^24/减半/2000 步加倍；算子分级表
L10	dynamic vs static 分配；in-place 的"唯一消费者"条件；checkpointing 的 $O(\sqrt{n})$ 与 +30% 时间
L11	LoRA 的反向不含 W_0；为什么加在 attention；S-LoRA 的统一分页；dLoRA = S-LoRA + 负载均衡 + 迁移
L12	TTFT/TPOT/E2E 的定义（含 −1）；三个挑战；三层优化栈
L13	静态 batching 的两个病；iteration-level scheduling；chunked prefill 用 TTFT 换 TPOT
L14	三种浪费；logical/physical/block table；COW；分页的三项成本
L15	KV 公式（每 token 128 KiB 的算例）；H2O 的 score 累加与 {2,6,9}+{10,11,12}；StreamingLLM 的 sink 机制
L16	仿射量化的两个参数；INT4 的 0.08 误差算例；SmoothQuant 的 $s_j=a_j^{\alpha}/w_j^{1-\alpha}$ 与两通道算例
L17	$y=\sum g_e E_e$；token-choice vs expert-choice；两次 All-to-All；T_{layer} 五项分解
L18	RAG 的 TTFT 分解；为什么 chunk KV 不能直接拼；CacheBlend 的偏差排名跨层稳定；四个 KV 决策

20.2 自测题（15 题，附答案）

Q1. 一个 transformer 的 hidden dim 是 768、12 个等大 head，每个 head 维度多少？若序列长 1024、16 个 head，显式存满 $N\times N$ score 矩阵（含 mask 位），共多少元素？序列翻倍后这部分显存涨几倍？

A1. $768/12=64$。$16\times1024\times1024=16{,}777{,}216$ 个元素。翻倍后 $\propto N^2$，涨 **4 倍**。

Q2. $L=61$、$h=128$、$d_k=128$，MHA 与 MQA 每 token 缓存的 K/V 标量数各是多少？比值？

A2. MHA：$2\times61\times128\times128=1{,}998{,}848$；MQA：$2\times61\times1\times128=15{,}616$；比值 **128×**（恰好等于 head 数）。

Q3. 一个 32 层、8 KV 头、head dim 128、BF16 的模型，每 token 的 KV 是多少？128k 上下文单请求要多少 GiB？

A3. $2 \times 32 \times 8 \times 128 \times 2 = 131{,}072$ B $= 128$ KiB/token；$131072 \times 131072 = 1.718 \times 10^{10}$ B $= 16$ GiB。

Q4. 推导 attention 的算术强度，并说明为什么 N 增大时它上升。

A4. compute $= (4h+3)N^2$，IO $= 8N^2 + 8Nh$，故 AI $= \frac{(4h+3)N^2}{8N^2+8Nh}$。$N \to \infty$ 时 $\to \frac{4h+3}{8} \approx \frac{h}{2}$；而 N 小时分母的 $8Nh$ 项（读 QKV）占主导，AI 低。**分子是 N^2、分母含 N 的一次项，所以 AI 随 N 单调上升。**

Q5. Ring AllReduce 的每卡通信量是多少？为什么说它"largely independent of P"？

A5. 两阶段各 $\frac{N}{P}(P-1)$，合计 $\frac{2N(P-1)}{P}$。$P \to \infty$ 时 $\frac{P-1}{P} \to 1$，趋近 $2N$——只与数据量有关，与卡数无关。

Q6. TP 的 MLP 里，为什么 GeLU 必须在 AllReduce 之前做？如果 W_1 按行切会怎样？

A6. GeLU 非线性，$\mathrm{GeLU}(Z_1 + Z_2) \neq \mathrm{GeLU}(Z_1) + \mathrm{GeLU}(Z_2)$。列切时每卡拿到的是 Z 的**不同列**（完整的一部分），可以本地过 GeLU；若 W_1 按行切，每卡算的是**同一批列的部分和**，必须先 AllReduce 求和再 GeLU，否则结果错。这正是 HW1 Q6(b)。

Q7. 8 卡一节点、8 节点 fat-tree，为什么 Megatron 实测峰值在 $t = p = 8$？

A7. $t \leq 8$ 时 TP 组不跨节点，all-reduce 走 NVLink（便宜）；$t > 8$ 时 TP 跨节点，每层两次 all-reduce 穿 IB 且**在关键路径上**，延迟无法被遮盖；$t < 8$ 则模型切得不够细，需要更大的 p 才能装下 → bubble 变大。最优解落在节点边界。

Q8. GPipe 式调度下，$m = 8$、$p = 4$、$t_f = t_b = t$，气泡率多少？若 m 提到 32 呢？这个结论有什么现实约束？

A8. $T = mt_f + (p-1)(t_f + t_b) + mt_b$，气泡率 $= \frac{(p-1)}{m+p-1}$。$m = 8$：$3/11 \approx 27.3\%$；$m = 32$：$3/35 \approx 8.6\%$。约束：m 变大会让每个 micro-batch 变小，GEMM 的 M 维下降、AI 下降，且 m 受 mini-batch 上限约束。

Q9. 为什么纯 FP16 训练会让后期权重更新"消失"？两个标准解法是什么？

A9. 当 $w \gg \alpha \nabla w$ 时，$w - \alpha \nabla w$ 在 FP16 的 10 位尾数下会四舍五入回 w，更新被静默丢弃。解法：① **FP32 master weights**（+ FP32 优化器状态），让微小更新在 FP32 上累积；② **loss scaling**（自动：从 2^{24} 起，溢出则减半并跳过该步，2000 步无溢出则加倍）。

Q10. Activation checkpointing 为什么能得到 $O(\sqrt{n})$ 的激活显存？代价是什么？

A10. 分 k 段，常驻 k 份段输出 + 反向时单段重算的 n/k 份 → 峰值 $\approx k + n/k$，在 $k = \sqrt{n}$ 处取极小 $2\sqrt{n}$。代价是**多一次前向**（每个段重算一次），实测训练时间约 +30%。

Q11. 用 12 个 token、6 个 slot（其中 3 个 recent），累积分数为 `2,10,3,7,4,12,5,1,9,2,3,1`，H2O 与 StreamingLLM（2 sink + 4 recent）分别保留哪些？

A11. H2O：heavy hitters 是分数最高的 3 个（排除 recent 窗口）= $\{2, 6, 9\}$，recent = $\{10, 11, 12\}$ → {2,6,9,10,11,12}。StreamingLLM：sink = $\{1, 2\}$，recent 4 个 = $\{9, 10, 11, 12\}$ → {1,2,9,10,11,12}。注意两者都在 slot 6 的预算下，但保留的"历史"完全不同。

Q12. 对称 INT4 量化，$\max|x| = 1.4$，$x = 0.68$，写出 S、q、$\hat{x}$ 与误差。

A12. 有符号范围 $-7..7 \to S = 1.4/7 = 0.2$；$q = \text{round}(0.68/0.2) = \text{round}(3.4) = 3$；$\hat{x} = 3 \times 0.2 = 0.60$；误差 0.08（相对 11.8%）。

Q13. SmoothQuant 中 $\alpha = 0.5$、通道 1 的 $a = 2$、$w = 4$，s 是多少？缩放后两个操作数的范围各是多少？

A13. $s = \sqrt{a/w} = \sqrt{2/4} = 0.7071$；$X/s = 2/0.7071 = 2.83$，$sW = 4 \times 0.7071 = 2.83$。原来 $(2, 4)$ 的悬殊被拉平到 $(2.83, 2.83)$。

Q14. MoE 层为什么需要两次 All-to-All？如果 $T_{dispatch}$ 占端到端 60%，应该优先优化什么？

A14. 第一次（dispatch）把 token 按路由送到持有目标专家的 GPU，第二次（combine）把专家输出送回原 GPU。若 dispatch 占 60%，应优先优化**通信**（更好的拓扑/专家放置、更大 batch 摊薄、overlap 流水），而不是优化 expert 的 GEMM——因为 $T_{layer} = \max_e T_{expert,e}$ 只占小头。

Q15. RAG 里为什么"独立编码的 chunk KV 直接拼接"会掉精度？CacheBlend 靠什么假设把它救回来？

A15. 因为 chunk B 的 KV 是在"没有 A"的上下文里算出来的，缺少 A 的 cross-chunk attention，拼到 A 后面得到的是"从未存在过的状态"，误差逐层累积。CacheBlend 依赖"**high-deviation token 的排名在相邻层之间强相关**"这一假设：在浅层用大候选集估计、逐层收窄，只重算少数对上下文敏感的 token，从而以很小的重算量接近 full prefill 的质量。

20.3 五个"如果只记五件事"

1. **三角**：装不下（切模型）、跑不快（通信）、算不久（精度）——任何优化都在动其中一条边，并付出另外两条的代价。
2. **AI 是通用货币**：MLP 1365 ops/byte 是 compute-bound，decode 的 GEMV ~1 op/byte 是 memory-bound。**推理优化几乎全在解决后者。**
3. **KV 是推理系统的第一等公民**：位置（分页）、保留（H2O/Streaming）、融合（CacheBlend）、调度（Mooncake）——四问四答。
4. **通信要么便宜要么被藏起来**：节点内（NVLink）做 TP，跨节点做 PP，剩下的给 DP；藏不住的关键路径延迟（TP 的 all-reduce）必须靠减小 P 解决。
5. **报端到端，不报利用率**：KV 利用率、比特数、bubble 率都是中间量，最终只有一个指标——**满足 SLO 的 goodput**。

第 21 章 参考来源

21.1 课程材料（本地快照，访问日期 2026-11）

文件	内容
`README.md`	课程大纲、18 讲日程、评分结构、必读论文列表、项目里程碑、CloudLab 说明
`slides/L1_overview.pdf` — `L18_RAG.pdf`（18 份）	讲义；本地另有 `pdftotext -layout` 抽出的 `.txt`
`hw/README.md`	HW1 三个编码题的说明、CloudLab 三机部署、`glooHelper.sh` 用法、30/70 评分
`hw/Assignment_1.pdf`（7 页）	HW1 题面：简答 14 分 + Q2 10 + Q3 30 + Q4 25 = 79 分
`hw/q2.py` / `q3.py` / `q4.py`	HW1 起始代码（含函数签名与待实现标记）
`hw/run.py`	官方 runner：Qwen3-0.6B + WikiText 训练（Q2/Q3）与 TP 检查（Q4）
`hw/grade_public.py`	公开评分器：3 个 public case/题，与 7 个 hidden case 同构
`hw/glooHelper.sh`	多机启动脚本（`-n -P -r -q -s -e`）
`hw/HW2.ipynb`（50 分）	MHA / GQA / TP 通信 / Paged KV / Continuous Batching / MoE 简答
`hw/HW3_AWQ.ipynb`（50 分）	Speculative Decoding（32）+ 量化与 AWQ（18）
`hw/cloudlab.md`	CloudLab 注册、开实验、预留硬件与使用政策

课程主页：https://github.com/fanlai0990/cs498（分支 `f26-mlsys`）。若与本笔记冲突，**以课程当前仓库为准**。

21.2 README 列出的必读论文（10 篇）

讲	论文	链接
L4	BLASST	https://arxiv.org/abs/2512.12087
L5	DeepSeek-V3 Technical Report（Sec 1–3）	https://arxiv.org/abs/2412.19437
L7	LlamaRL	https://arxiv.org/abs/2505.24034
L8	Alpa（OSDI'22）	https://www.usenix.org/system/files/osdi22-zheng-lianmin.pdf
L10	ZeRO（arXiv 1910.02054）	https://arxiv.org/abs/1910.02054

讲	论文	链接
L13	Speculative Decoding（Leviathan et al., 2023）	https://arxiv.org/abs/2211.17192
L14	DistServe（OSDI'24）	https://arxiv.org/abs/2401.09670
L15	SGLang（NeurIPS'24）	https://arxiv.org/abs/2312.07104
L16	Murakkab（OSDI'26）	https://www.usenix.org/system/files/osdi26-chaudhry.pdf
L18	Mooncake（FAST'25）	https://www.usenix.org/conference/fast25/presentation/qin

21.3 讲义正文引用的论文与技术报告

讲	文献	用途
L2/L3	Vaswani et al., *Attention Is All You Need*, NeurIPS 2017	Transformer 原始结构
L2/L3	Dao et al., *FlashAttention*, NeurIPS 2022	tiling + online softmax
L2	Su et al., *RoFormer / RoPE*, 2021	旋转位置编码
L3	*Native Sparse Attention*（DeepSeek）, ACL 2025	块级稀疏 + 可训练
L6/L7	Shoeybi et al., *Megatron-LM*, 2019	TP 的列/行切方案
L7/L8	Narayanan et al., *Efficient Large-Scale LM Training on GPU Clusters Using Megatron-LM*, SC'21	1F1B 与 Interleaved、$t = p = 8$ 实测
L7	*PipeDream*	1F1B 调度
L5	*Centauri*, ASPLOS'24 Best Paper	通信切分与 overlap
L5	*ByteScheduler*	通信重排与 overlap
L9	*Mixed Precision Training*（Micikevicius et al.）+ NVIDIA《Train With Mixed Precision》	loss scaling 规则
L10	*Activation checkpointing*（Chen et al., 2016）	$O(\sqrt{n})$ 内存
L11	Hu et al., *LoRA*；*S-LoRA*（MLSys'24）；*dLoRA*（OSDI'24）	低秩适配与多 LoRA 服务
L12	*LLM.int8()*, 2023；*ZeroQuant*, NeurIPS 2022；*GOBO*, MICRO 2020；*SliceGPT*, ICLR 2024；*DejaVu*, 2023；*IC-Cache*, SOSP'25	压缩四件套
L13	Yu et al., *Orca*, OSDI 2022；*Sarathi-Serve*, OSDI 2024	continuous batching 与 chunked prefill
L14	Kwon et al., *PagedAttention / vLLM*, SOSP 2023	分页 KV
L15	Zhang et al., *H2O*, NeurIPS 2023；Xiao et al., *StreamingLLM*, ICLR 2024	KV 保留策略

讲	文献	用途
L16	Xiao et al., *SmoothQuant*, ICML 2023；Lin et al., *AWQ*, MLSys 2024 Best Paper	W8A8 与 weight-only
L17	Shazeer et al. 2017；*GShard* 2021；*Switch* 2022；*DeepSpeed-MoE*, ICML 2022；*Tutel*, MLSys 2023；*DeepSeekMoE*, ACL 2024；Mixtral 8×22B 文档	MoE 架构与系统
L18	Yao et al., *CacheBlend*, EuroSys 2025 Best Paper	RAG 的选择性重算
HW3	*EAGLE-3*（2025）、*DFlash*（2026）、*DSpark*、DeepSpec 源码；AWQ 官方 quantizer / auto_scale / auto_clip	drafting 与量化实现参考

21.4 关于本笔记的三点声明

1. **讲义以图为主的地方**（L6、L8 抽文本仅 11 KB 左右），公式部分统一标 `【补充】`，其来源是 README 指向的论文与公开常识；**课程本身是否讲过该公式，本笔记不做断言**。
2. **所有数字都做了可复算处理**：KV 的 GiB、TP 的 MiB、量化的 tok/s、参数量的 GB/GiB 都在正文给了算式。若某处无法复算，已标 `【量级】`。
3. **课程信息（分数、日期、讲师、题号、函数名）全部来自本地 `README.md`、`hw/README.md`、`Assignment_1.pdf`、`q2.py` / `q3.py` / `q4.py` / `run.py` / `grade_public.py` 与两个 notebook**，没有任何一处凭印象填写；不确定的地方已注明"以课程当前仓库为准"。