Jingfeng



大模型推理加速与 AI 基础设施：从算子到集群

一次推理请求、五层系统与可核验成本模型

Jingfeng · 2026 年 10 月 22 日修订

摘　要

一次推理请求从接入、排队、预填充到逐 token 解码，连接请求与调度、引擎与显存、算子与 kernel、硬件与互联、集群与成本五层系统。局部 kernel 的加速只有在质量、容量、尾延迟和故障约束同时成立时，才会转化为用户可用的吞吐与单位成本下降。本书从 Roofline 和显存账本出发，连接注意力、量化、MoE、并行、KV 管理与引擎实现，再讨论存储、通信、多租户、可靠性和成本。

全书保留十九章与两个附录，以一个明确的 70B GQA 教学配置串起请求、资源和成本。解析值由配套脚本复算；CPU 数学检查和合成请求轨迹已经运行；GPU 服务基准提供可执行方案。论文报告、厂商规格、解析下界、合成教学数据与本次实测分别标注，避免以峰值能力代替服务性能。

源码比较固定到 vLLM v0.11.0 与 SGLang v0.5.3.post3 的提交，检索与修订截至 2026 年 10 月 22 日。涉及价格、产品和未固定文档的内容均有时间与证据边界；未取得足够来源的商业数字不作为结论依据。

版本与使用范围

原稿署名为“AI Infra 研究”，归档于 2026 年 10 月 21 日。本版由 Jingfeng 整理修订，归档源稿与旧 PDF 保留。正文适合具备基础线性代数、概率和编程知识的读者；教学算例不构成硬件采购报价或生产容量承诺。

目录

前言与阅读路线 iii

第 I 部分　推理加速 1
第 1 章　引言：推理是系统问题 2
第 2 章　物理约束：roofline、显存墙与能耗 5
第 3 章　注意力谱系与 IO 优化 12
第 4 章　量化：容量、带宽与计算通路 21
第 5 章　稀疏性与 MoE 28
第 6 章　并行策略与显存账本 35
第 7 章　调度与 KV cache 管理 44
第 8 章　vLLM 与 SGLang：固定版本的源码比较 55

第 II 部分　AI 基础设施技术栈 61
第 9 章　AI 基础设施全景与成本模型 62
第 10 章　算力、存储带宽与硬件选型 65
第 11 章　内存、存储与 KV 缓存基础设施 69
第 12 章　网络、拓扑与集合通信 73
第 13 章　调度、编排与多租户 77
第 14 章　编译器、kernel 与运行时 81
第 15 章　可靠性与容量规划 85

第 III 部分　产业、科学与展望 94
第 16 章　公司与开源生态：七层栈上的十六个观察点 95
第 17 章　单位经济学：从资源账到达标任务成本 100
第 18 章　前沿科学：当推理成为科研仪器 105
第 19 章　开放问题、前沿方向与学习路线 110
附录 A　术语表 118
附录 B　符号表 123
参考文献 128

前言与阅读路线

这本书怎样使用证据

读到一个“加速两倍”的数字时，先问比较对象是否使用同一个模型、精度、长度分布和质量门槛，再问测量覆盖的是单个 kernel、引擎还是包括排队与重试的服务。解析模型给出的理想下界、厂商的峰值规格、论文中的实验和本次本机运行，各自只能支持相应范围的结论。正文中的【自算】是显式假设下的计算，【论文】需要连同论文实验条件阅读，【官方文档】说明接口或规格，并不自动证明某个生产服务达到该性能。

本版采用五层请求视角：L1 请求与调度、L2 引擎与显存、L3 算子与 kernel、L4 硬件与互联、L5 集群与成本。第 1 章给出层的边界。第 9 章的七层产业栈用于描述供给侧分工；两种分层服务不同问题，不能把产业栈第五层的引擎公司等同于请求视角的 L5。

贯穿全文的一次请求

设一个无真实产品名称的稠密 Transformer，参数配置见表 1。“70B”只规定近似权重规模，不能唯一决定层数、KV 头或真实 FLOP；这里把这些变量分别固定。输入已经包括 system prompt 和历史消息，输出长度固定用于复算；实际服务需要长度分布，不能用这个定长算例替代重尾负载。

表 1　全书贯穿算例的合同。全部为教学假设；其它配置的局部算例另行注明。

变量	明确定义与取值
模型	70×10^9 参数；80 层；隐藏维度 8192；64 个 Q 头；8 个 KV 头；每头 128 维
数值格式	BF16 权重与 KV，每元素 2 byte；不含量化 scale、投机分支或多模态编码器
请求	输入 4096 token，输出 512 token；资源预留按 4608 token 保守计算
设备布局	4 卡 TP，单卡 80 GB（十进制）；可分配比例 0.9；另预留每卡 8 GB 运行空间
SLO	客户端 TTFT $\le$ 2 s；每请求最大 ITL $\le$ 50 ms；端到端完成 $\le$ 30 s
成本口径	USD；假想内部资源分摊价 3 USD/(GPU·h)，不含设施电费；并非任何地区报价
能耗与产出	每卡平均 IT 功率 0.35 kW，PUE=1.2，电价 0.10 USD/kWh；合格输出 goodput 假设 480 token/s

请求到达后，网关先记录客户端与服务端的时钟边界，再由调度器检查 token 与显存预算。缓存命中只能复用与模型、适配器、token 序列及隔离域相容的前缀；未命中部分执行 prefill。每轮 decode 分配新 KV 槽位，运行矩阵乘和注意力，再将采样结果送回客户端。如果 prefill/decode 分离，4096 个输入 token 的 KV 需要转移；如果某个 TP rank 失败，同一请求不能把其它 rank 的局部结果当成完整输出。最终按成功、质量和 SLO 三项筛选产出。

表 2 由项目内脚本自动生成，与 JSON 配置和结果一一对应。KV 容量上限 76 路只是“显存允许”，未包含吞吐、排队、通信和故障余量约束，不可宣称该并发一定满足 SLO。预留 512 个输出位置还比最后一次前向实际生成的 KV 略保守；边界差异不会改变本例的结论。

传输下界采用一个共享的 100 Gbit/s 聚合瓶颈，负载效率 0.7；TP 分片总共仍是 1.25 GiB，不能既将总量除以四又仍使用四路带宽之和。传输时间不包括握手、注册、排队与同步。成本小计 7.042 USD/百万合格输出 token 由给定的 goodput 得到，不是引擎实测或市场价格。若使用已经包含电费的租金，应删去额外电费项；按全部输入与输出 token 作分母会得到不同数值，两种口径不得放在同一列比较。

表 2 贯穿算例的脚本复算结果；容量与传输仅是模型界限，吞吐是输入假设。

量	结果
全层 KV 字节数	327,680 byte/token
权重 / 每请求 KV 预留	140 GB / 1.40625 GiB
32 路请求 KV / 容量上限	45 GiB / 76 路
prefill KV 传输 / 解析下界	1.25 GiB / 153.39 ms
资源分摊 / 设施电费	12.000 / 0.168 USD/h
给定输出 goodput / 等长请求率	480 token/s / 0.9375 request/s
每百万达标输出 token 小计	7.042 USD

测量合同与阅读次序

以客户端发出请求的时刻 t_0、首个输出 token 到达的时刻 t_1、最后一个 token 时刻 t_n 和响应完成时刻 t_{done} 定义

$$\text{TTFT} = t_1 - t_0, \qquad \text{E2E} = t_{\text{done}} - t_0, \tag{1}$$

$$\text{ITL}_i = t_i - t_{i-1} \quad (i \geq 2), \qquad \text{TPOT} = \frac{t_n - t_1}{n-1} \quad (n \geq 2). \tag{2}$$

TPOT 是每请求的平均间隔，不能替代该请求最大 ITL；只有一个输出 token 时二者均无定义。流式响应的一条消息可能包含多个 token，其 chunk 间隔也不能当成逐 token ITL。服务端队列时间与设备执行时间用于归因，客户端时间用于本文的 SLO 判定。

本书的输出 token goodput 是固定观测窗口内，同时满足成功、质量与逐请求 SLO 的输出 token 总数除以窗口时长。请求通过率的分母包括失败与超时请求。TTFT/TPOT 的请求分位数和全部 token 间隔的分位数具有不同权重；报告时须写明样本集及分位数算法。窗口末未完成请求应作为删失样本另报，或者在停止发流量后等到全部终止，不能悄悄删除。

第一次阅读可顺着第 1、2、3、7、8 章理解一次请求，再读基础设施和可靠性部分，把局部收益接到集群约束。要选择优化方法，先对照注意力、量化、MoE 与并行章节的假设，再检查硬件、通信和运行时。要复现实验，先运行配套的三个 CPU 检查，理解测量合同，再使用第 19 章的 GPU 基准方案。代码教学、模型基础数学和 Agent 压缩分别属于本目录相邻的 CUDA、大模型数学与 Agent 专题；本书只保留建立系统判断所需的前置解释。

修订说明中的章节审计记录了删除或改写的理由；无法获得充分公开证据的商业细节和未执行的 GPU 比较均明确列为限制。读者可以从这个边界继续研究，而不必猜测书中的数字是否实测。

Part I

推理加速

第 1 章　引言：推理是系统问题

本章结论

(1) 在低并发、权重未驻留片上缓存的 dense decode 中，每步通常要读取几乎全部权重，而每次读取只服务少量 token，因此常受显存带宽限制。长上下文会增加 KV 读取，大 batch、MoE 路由和多卡通信又会改变瓶颈；需要按具体算子与负载计算。

(2) 由此派生出四个相互独立、又相互挤压的杠杆：减少要搬的字节（量化、稀疏、更省 KV 的注意力）、提高每次搬运服务的请求数（批处理、前缀复用）、把搬运与计算重叠起来（kernel 融合、异步、图捕获），以及换一个能搬得更快的物理层（更高的 HBM 带宽、片上 SRAM、多卡并行）。

(3) 工程上真正难的不是找到杠杆，而是在延迟 *SLO* 约束下同时使用它们：增大 batch 提高吞吐，却恶化单请求延迟；分离 prefill 与 decode 改善两者的各自效率，却引入 KV cache 的跨节点传输成本。所有调度策略本质上都在这条约束曲线上选点。

两类负载，两种物理

把一次大模型请求拆成两个阶段，是理解全部推理系统的起点。

Prefill（预填充）处理输入的整个 prompt。此时 n 个 token 同时进入模型，每一层的矩阵乘形状是 $[n, d] \times [d, d']$，GPU 的每个 SM 都能拿到足够多的工作。较长 prompt 的线性层往往更接近算力受限（compute-bound），但短 chunk、注意力、通信与 CPU 开销仍可能限制引擎。需要同时观察输入 token/s 与请求的首 token 延迟。

Decode（解码）每次只前进一个 token。此时矩阵乘退化成 $[1, d] \times [d, d']$，即一个向量与一个矩阵相乘——在 batch size 为 1、权重远大于片上缓存时，它通常受显存带宽限制：必须把 $d \times d'$ 个权重全部读一遍，却只做了 $d \times d'$ 次乘加。算术强度（每字节内存流量对应的浮点运算数）约等于 1，而 H100 的机器平衡点约为 295 FLOP/byte［自算，见第 2 章］。两者的差距不是百分之几十，而是两个数量级。

• 整本书的分析框架

任何一项推理加速技术，都可以用一句话定位：它降低了哪一段内存流量，或者提高了哪一段流量的复用次数，代价是什么。此外还要检查计算量、串行依赖、通信和固定开销；单一字节模型不是所有优化的充分解释。

这一反差解释了为什么低并发生成的许多核心设计围绕内存展开：KV cache 的分页管理（第 7 章）、把 batch 拉大以提高权重复用（第 7 章）、把权重压到 4 bit（第 4 章）、用更省 KV 的注意力结构（第 3 章）——它们减少每个有效输出 token 的字节流量，但质量、延迟和容量的约束不同。

三个瓶颈

表 1.1　推理系统的三个瓶颈及其表现。第三列是本书对应的章节。

瓶颈	物理量	典型表现	章节
带宽墙	HBM 带宽	decode 时 GPU 利用率只有十几个百分点，SM 在等数据	2、4
容量墙	HBM 容量	长上下文或大 batch 时 KV cache 放不下，被迫抢占、重算、降并发	2、7、11
调度墙	请求间的耦合	一个长 prompt 阻塞整批；TTFT 与 ITL 无法同时满足	7、13

表 1.1 里的三堵墙并不是并列关系。带宽墙决定了单请求的下限，容量墙决定了并发数的上限，调度墙决定了在给定并发下能否兑现延迟承诺。一台推理服务器的实际吞吐，是这三者的最小值。

四个杠杆

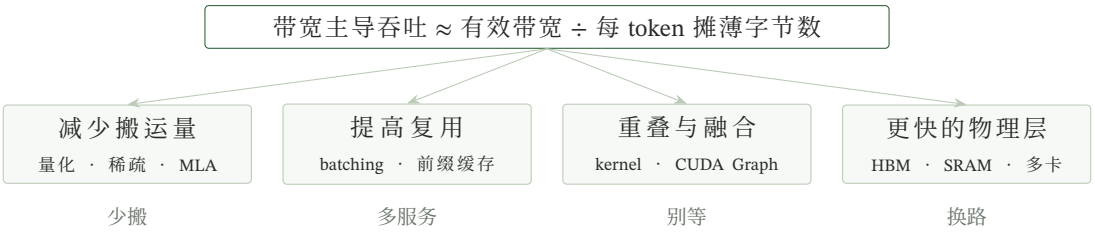


图 1.1 四个杠杆分别影响流量、复用与执行效率。「少搬」减少读取字节，「多服务」通过复用减小每 token 的摊薄字节，「别等」让等待时间被其他工作填满，「换路」提高有效带宽。四者并非可独立叠加——量化会降低 kernel 效率，大 batch 会挤占 KV 空间，这些耦合是第 7 章的主题。

图 1.1 的公式值得写成精确形式。设模型权重字节数为 W，每 token 每层的 KV 字节数为 k，当前可见上下文长度为 ℓ（输入加已生成部分），batch 为 B，总层数为 L，则 decode 单步需读取的字节数约为 $W + B \cdot kL\ell$，而单步产出 B 个 token。于是每 *token* 摊薄字节数为

$$\frac{\text{bytes}}{\text{token}} \approx \frac{W}{B} + kL\ell. \tag{1.1}$$

式 (1.1) 是全书出现频率最高的式子。它立刻给出三个结论。第一，提高 batch 可以让权重项按 $1/B$ 衰减；连续批处理还通过及时补位减少槽位空闲。第二，当 B 增大到 $W/B \ll kL\ell$ 时，KV 项成为主导，继续加 batch 不再划算；此时应检查降低 k（GQA、MLA、量化 KV）、减少可见上下文、改善访问模式或提高带宽的代价。第三，长上下文（ℓ 大）会让 KV 项线性膨胀，这解释了为什么长上下文推理的成本结构与短请求完全不同。

一次请求穿过五层

副标题里的「五层」不是修辞，它有确定的所指：一次请求从进入系统到吐出 token，要依次穿过五个物理上可分别测量的层次。表 1.2 把它们与本书的章节对应起来。

表 1.2 一次请求穿过的五层。第三列列出该层的决策与可观测约束，第四列对应后续章节；层间会相互影响，分层用于定位测量与优化入口。

层	名称	这一层决定什么	对应章节
L1	请求与调度	哪些请求在这一步被服务、以什么顺序、batch 多大	7、13
L2	引擎与显存	KV 放在哪、怎么分配、命中率多少、能否抢占	7、8、11
L3	算子与 kernel	每层的矩阵乘、注意力、量化与通信用什么 kernel 实现	3、4、5、14
L4	硬件与互联	带宽、容量、拓扑、并行度如何切分模型	2、6、10、12
L5	集群与成本	功耗、制冷、故障率、利用率，最终折算成 $/token	9、15、17

这张表有两个用处。**其一，定位问题**。当一次请求变慢时，它必然是这五层中某一层的物理量恶化了：L1 是排队、L2 是显存不足导致的抢占、L3 是 kernel 没打满、L4 是带宽或通信、L5 是资源被别的负载挤占。按层排查比按「感觉」排查有效得多。**其二，定位技术**。第 2 章之后每一章讨论的技术，都可以在表里找到它动的是哪一层的哪个量。

两种分层的区别

第 9 章会给出另一张图：AI Infra 的七层栈（芯片、内存互联、编译 kernel、编排调度、推理引擎、推理云、端侧）。那张图是产业分工视角，回答「谁负责哪一段」；表 1.2 是请求路径视角，回答「一个 token 的时间花在哪」。两者不冲突，但也不能混用：七层栈里的一层（例如「推理引擎」）横跨请求路径的 L1–L3。读本书时若遇到「层」这个词，请先确认是哪一种。

贯穿全书的请求与计量边界

后文使用一个公开、可复算的**教学基线**：70B dense GQA，$L = 80$、$d = 8192$、query 头数 64、KV 头数 8、每头 128 维；BF16 权重按 70×10^9 个参数近似为 $W = 140$ GB，KV 为 $kL = 327{,}680$ byte/token。输入 4096 token，最多输出 512 token。它借用 70B 级公开结构说明机制，不等同于某个 checkpoint 的精确内存快照。单请求在输入末尾已有 1.25 GiB KV，生成到总长 4608 时增为 1.40625 GiB【自算】。第 6 章另保留 32k、32 并发压力例，不能把它的容量预算当成本例的测量。

请求从入口鉴权与分词开始，在 L1 排队，L2 查前缀并分配 KV，L3/L4 执行 prefill，随后重复 decode 与输出，L5 记录故障、功率和账单。本文的客户端时钟从请求发送时刻 t_0 开始，第 i 个有效输出 token 到达为 t_i：

$$\mathrm{TTFT} = t_1 - t_0, \qquad \mathrm{ITL}_i = t_i - t_{i-1}\ (i \geq 2), \qquad \mathrm{TPOT} = \frac{t_o - t_1}{o - 1}\ (o > 1). \tag{1.2}$$

响应完成时刻记为 t_{done}，端到端延迟为 $t_{\mathrm{done}} - t_0$；忽略末 token 之后结束标记与收尾传输时才近似为 $t_o - t_0$。只有一个输出 token 时 TPOT 不定义。TPOT 是请求内平均值，不能代替最大 ITL 或全服务的 ITL 分位数。流式接口一次发送多 token 时还应单列客户端块间隔，不要把不可观测的块内 token 时间当成实测 ITL。

主例要求 **TTFT 不超过 2 s、请求内最大 ITL 不超过 50 ms、端到端不超过 30 s**，并同时满足质量要求。若恰好输出 512 token，前两项给出的充分延迟预算为 $2 + 511 \times 0.05 = 27.55$ s。服务级达成率按所有到达请求统计，失败、超时和拒绝另列计数；goodput 按满足预先约定的质量与延迟条件的完成请求数除以观测时间。论文中的 TPOT 约束、分位数 SLO 和最大 ITL 约束不是同一指标。

单 kernel 计时只能证明 L3 的局部变化；引擎测试还包括组批、缓存、通信；服务测试再包括入口、排队和失败。后文的 roofline 给理想上界，解析账本给条件场景，论文报告给特定实验，三者均不能替代本负载的服务级 goodput 实测。

这不是一本什么样的书

本书不打算成为论文索引。每一章都会回答同一组问题：这项技术动了哪个物理量、在什么条件下才开始赚钱、它的失效边界在哪里、工程落地时最先坏掉的是什么。

本次修订在无目标 GPU 的环境运行解析账本与合成调度核验，并提供真实引擎的复用评测方案。涉及论文加速比时保留原论文条件；涉及本书结果时区分条件估算与本次实测。凡可由解析模型算出的量（显存账本、roofline 上界、成本模型），均给出公式与代入过程，读者可以自行验算 [1, 2]。

关于本书的数字口径

书中出现的性能数字分五类，正文中以方括号标注：【论文】来自同行评议论文或技术报告，附测试条件；【官方文档】来自框架或硬件的官方文档；【厂商宣称】来自产品页、发布会或营销材料，未经独立验证；【第三方复现】来自独立评测；【自算】由本书根据公开参数推导，给出完整公式。不带标注的数字只出现在纯数学推导中。

未解问题

1. **给定 SLO 时，可达吞吐的边界在哪里？**Roofline 给出硬件约束下的理想上界，但没有刻画排队、组批与质量约束。如何把这些因素纳入同一模型，并用真实负载验证，是从局部上界走向服务容量的关键。
2. **请求形状的分布是否稳定？**按历史长度分布调优的策略，需检验估计在当前负载上是否仍然有效。当应用从聊天转向 agent、代码补全或长文档问答时，分布可能变化，使原参数失效。如何在分布漂移下保持调度鲁棒，仍需把排队分析、在线估计与真实工作负载验证结合。

第 2 章　物理约束：roofline、显存墙与能耗

本章结论

(1) Roofline 用计算量与数据流量给出理想性能上界。峰值算力除以显存带宽是机器平衡点，不是实测 kernel 效率；H100 SXM 的 BF16 稠密口径约 295 FLOP/byte。

(2) 线性层 prefill 与 decode 的强度公式相同，区别在可批处理的 token 数与 KV 依赖。短 chunk、长上下文、通信和固定开销会破坏简单的“prefill 算力、decode 带宽”标签。

(3) 容量、流量和能耗是三本账：KV 存得下不代表读得快，TDP 也不是实际功耗。每个估算都必须明确单卡或节点范围，并统一 GB 与 GiB。

2.1　roofline 的推导，而不是背诵

设一个 kernel 的浮点运算量为 F（FLOP），需要从显存搬运的字节数为 Q，加速器的峰值算力为 P（FLOP/s），显存带宽为 B_{mem}（byte/s）。即使允许计算与访存理想重叠，耗时仍不小于

$$T \geq \max\left(\frac{F}{P}, \frac{Q}{B_{\text{mem}}}\right). \tag{2.1}$$

定义算术强度 $I = F/Q$（FLOP/byte），式 (2.1) 可改写为

$$T \geq \frac{F}{P} \cdot \max\left(1, \frac{I^*}{I}\right), \qquad I^* = \frac{P}{B_{\text{mem}}}. \tag{2.2}$$

I^* 是这个理想模型的平衡点：$I > I^*$ 时计算项较大，$I < I^*$ 时访存项较大。实际性能至多为 $\min(P, IB_{\text{mem}})$，未必达到任一峰值；若两项完全不能重叠，时间至少为 $F/P + Q/B_{\text{mem}}$。启动、依赖、通信还会增加时间。Q 须指同一存储层级的实际流量，不能把张量尺寸直接当成 HBM 计数器读数。原始 Roofline 方法的对象是性能上界，不能用它单独证明服务 SLO。

▶ H100 SXM 的机器平衡点

已知 FP16/BF16 稠密算力 $P \approx 989$ TFLOPS $= 9.89 \times 10^{14}$ FLOP/s，HBM3 带宽 $B_{\text{mem}} = 3.35$ TB/s $= 3.35 \times 10^{12}$ byte/s【官方文档】[3]。

$$I^* = \frac{9.89 \times 10^{14}}{3.35 \times 10^{12}} \approx 295 \text{ FLOP/byte}.$$

含义：一个 kernel 每从显存读 1 字节，必须做满 295 次浮点运算才能跑满算力。对纯矩阵乘（$M = N = K = n$、FP16、只计 A 与 B 两块输入）算术强度约为 $n/2$，即忽略输出写入时至少需 $n \geq 590$ 才越过理想拐点；若计入 C 写回则为 $n/3$，门槛约 885。下一节的长方形 GEMM 在权重主导近似下得到 295，差别来自矩阵形状及被计入的流量，不是把同一 GEMM 换一个名字就少读一块输入。
注意 I^* 依赖精度：把 P 换成 FP8 的 1979 TFLOPS，同一个比值变成约 591。本书凡未特别注明，I^* 均指 FP16；第 10 章比较各代硬件时会显式区分精度，因为这两组数字在讨论「换新卡有没有用」时给出的是同一个结论，但推导过程不同。

! 常见误解

在低 batch、权重主导的 decode 中，只提高张量峰值算力通常收益有限。但小 kernel、注意力、通信或 CPU 调度也可能限制时间，不能仅凭低 MFU 判定 SM 在等待 HBM。提高 batch 或启用投机验证之后，计算项也可能进入关键路径。H200 相对 H100 的主要提升正是带宽（HBM3e 约 4.8 TB/s 对 3.35

TB/s），而非算力【官方文档】[4]。

2.2 把 decode 的算术强度算出来

考虑一层前馈网络（FFN），权重矩阵 $W \in \mathbb{R}^{d\times d'}$，batch 为 B，权重每元素 b_w 字节，激活每元素 b_a 字节。FLOP 为 $2Bdd'$，字节为 $b_w dd' + b_a B(d+d')$。于是

$$I_{\text{FFN}} = \frac{2Bdd'}{b_w dd' + b_a B(d+d')} \approx \begin{cases} \dfrac{2B}{b_w}, & B \ll B^* \text{（权重主导）}, \\ \dfrac{2dd'}{b_a(d+d')}, & B \gg B^* \text{（激活主导）}, \end{cases} \qquad B^* \equiv \frac{b_w dd'}{b_a(d+d')}. \tag{2.3}$$

两个区间的分界是「权重读取量等于激活读写量」的那一点。对 $d = d' = 8192$ 的方阵、FP16（$b_w = b_a = 2$），$B^* = d/2 = 4096$，高于不少在线 decode 的批大小。因此不少低 batch decode 线性层位于第一行。两行都只是该 GEMM 的理想流量模型：

• decode 的算术强度就是 batch 本身

在 $B \ll B^*$ 的权重主导区，式 (2.3) 退化为

$$I \approx \frac{2}{b_w} B.$$

FP16 下 $b_w = 2$，于是 $I \approx B$：**算术强度在数值上就等于 batch size**。这就是为什么「加大 batch」是 decode 的第一性优化——它不是「提高利用率」这种模糊说法，而是把工作点沿 roofline 的斜边往上推，每翻一倍 batch 就把强度翻一倍。
代进平衡点 $I^* = 295$：FP16 下需要 $B \approx 295$ 才摸到拐点；换成 INT4（$b_w = 0.5$）斜率变成 4，只需 $B \approx 74$。若仍使用相同 BF16 计算通路，W4 把权重主导区的强度斜率提高 4 倍；缩放元数据、解包和 kernel 效率会降低实际收益，第 4 章分别记账。

对主例 70B GQA，295 个 4096-token 请求的 KV 合计 $295 \times 1.25 = 368.75$ GiB，约 395.94 GB。若教学预算仅留 480 GB 给 KV，它占 82.5%；而在 4k 上下文，KV/权重交叉点为 $140 \times 10^9/(327680 \times 4096) \approx 104$，并非 8k 时的 52【自算】。所以忽略 KV 而追求 $B = 295$ 容易误判容量与带宽，但不能据此断言所有 BF16 decode 永远带宽受限。

2.3 prefill 是另一张 roofline

上一节只算了 decode。prefill 用的是同一个公式，只是那个变量换了名字：把长度为 n 的 prompt 送进权重矩阵 $W \in \mathbb{R}^{d\times d'}$，FLOP 为 $2ndd'$，字节为 $b_w dd' + b_a n(d+d')$（权重整段只读一次，激活按 token 走）：

$$I_{\text{prefill}} = \frac{2ndd'}{b_w dd' + b_a n(d+d')} \xrightarrow[n \ll B^*]{} \frac{2}{b_w} n, \qquad I_{\text{prefill}}\Big|_{n\to\infty} \to \frac{2dd'}{b_a(d+d')} = \frac{d}{b_a} \text{（方阵）}. \tag{2.4}$$

把式 (2.3) 与式 (2.4) 并排看，会发现它们逐字相同，唯一的差别是 B 换成了 n。这不是巧合，而是本章最该记住的一句话：

• prefill 与 decode 的差别不是算术，是那个变量能不能变大

两个阶段的算术强度由同一个函数 $I(x) = 2x/b_w$（$x \ll B^*$ 时）决定，x 对 decode 是 batch、对 prefill 是 prompt 长度。较长 prompt 可使线性层越过 295；decode 并发受容量和 SLO 约束，常停在拐点左侧。这解释了常见趋势，不能替代对短 prefill、注意力或大 batch decode 的逐算子分析。
在这个线性层模型内，增大 x、减少 b_w 或降低 P/B_{mem} 都能改变工作点；前缀缓存则直接省掉已经计

算的 prefill，不等于提高 decode 的 batch。图捕获、通信重叠等优化还会减少模型之外的固定开销。第 3 至第 8 章的每一节，都可以按这三条归位；第 10 章起换到集群视角，但杠杆仍是同一组。

▶ prefill 从哪个长度开始算力受限

令式 (2.4) 等于平衡点 $I^* = 295$（FP16，$b_w = b_a = 2$），解出交叉长度。取 $d = d' = 8192$ 的方阵（FFN 的典型形状）：

$$\frac{2n}{2 + 2n \cdot \frac{2\times 8192}{8192^2}} = 295 \implies \frac{n}{1 + n/4096} = 295 \implies n^* = \frac{295}{1 - 295/4096} \approx 318.$$

含义：本例方阵 GEMM 在约 320 token 时越过理想拐点，不能把它作为所有 prefill 的固定阈值。（用第一行的简化式 $I \approx 2n/b_w = n$ 估，交叉点就是 295，与精确解 318 相差 7%——差距来自分母里被忽略的激活项，prompt 越长差距越大。）
继续把 n 加大，强度沿斜边继续爬：用简化式 $I \approx 2n/b_w = n$ 估，2048 token 时约 2048，是平衡点的 7 倍；把激活项算进去（$d = 8192$ 方阵，$I = dn/(d + 2n)$）降到约 1365，仍是 4.6 倍。再往后增速放缓，向渐近值 $d/b_a = 4096$ 靠拢【自算】。两种估法差的就是被忽略的激活读写：n 越大、d 越小，这项越不可忽略。这两个估法都表明本例 $n = 2048$ 的线性层越过理想拐点。但 TTFT 还含排队、注意力、通信与输出，ITL 也包含调度；不能从一个 GEMM 推广到完整服务。对已算力受限且计算通路不变的线性层，单纯压缩权重的理想收益趋近零；若量化同时启用低精度矩阵乘，或改变驻留与批大小，则结论不同。

▶ 换一张更快的卡，两边各快多少

这是全书最值得亲手算一遍的对照。取 H100 SXM 与 H200 SXM：算力同为 989 TFLOPS（FP16 稠密），HBM 带宽从 3.35 升到 4.8 TB/s【官方文档】[3, 4]。
decode（带宽受限）：时间 $\propto 1/B_{\text{mem}}$，加速比 $= 4.8/3.35 \approx 1.43\times$。
prefill（算力受限）：时间 $\propto 1/P$，加速比 $= 989/989 = 1.00\times$。
1.43 与 1.00 是固定容量、相同效率、忽略其他开销的两项上界比值，不是 H200 端到端实测。H200 更大的容量可能容纳不同 batch 并改变并行度，更高带宽也会帮助 prefill 中的访存算子，必须重新测 TTFT 与 ITL。

反过来看 Blackwell：B200 的 FP4 稠密算力约 9 PFLOPS（FP8 为 4.5 PFLOPS），带宽 8 TB/s，平衡点升到约 562（FP8）——比 *H100* 还高。换言之，新一代卡把「算力/带宽」的比值推得更高，decode 相对而言更带宽受限了，而不是更少。这一点在第 10 章的表 10.1 里逐卡列出。

这里要小心一个常见的混淆：**FP4 的「算力」与 FP4 的「字节数」是两件事**。对 decode 有用的是后者——权重从 8 bit 降到 4 bit 时，纯权重数据字节减半（从 BF16 降则为四分之一），直接作用在 b_w 上；对 prefill 有用的才是前者——真的用 FP4 张量核计算，让算力受限的那一段也变快。把两者混为一谈，就会得出「换了带 FP4 的新卡，decode 就该快一倍」这种错误预期：若量化只把权重压到 4 bit、kernel 仍要反量化成 BF16 再算，decode 拿到的是带宽收益；只有 kernel 真的以 FP4 执行 MMA，prefill 才拿到算力收益。第 4 章会把这两笔收益分开算——它们的适用阶段完全不同，这正是「FP4 对推理的收益大于训练」的准确表述。

2.3.1 第三类负载：embedding 与 reranker

不生成文本的 encoder-only embedding 通常不保留跨请求的自回归 KV，但每层仍需读写激活并执行注意力。设去除 padding 后一个 GEMM 实际处理 T 个 token，不能把全模型参数量放进分子、只把一层激活放进分母。对 $d \times d'$ 线性层，

$$I_{\text{emb,lin}}(T) = \frac{2Tdd'}{b_w dd' + b_a T(d + d')}, \qquad \lim_{T\to\infty} I_{\text{emb,lin}} = \frac{2dd'}{b_a(d + d')}. \tag{2.5}$$

BF16 方阵 $d = 4096$ 时极限为 2048 FLOP/byte，不是 1.7×10^6。$T = 1$ 时强度约 1，$T = 128$ 约 120，$T = 4096$ 约 1365【自算】；因此 embedding 也会受小 batch、短序列、padding 与 kernel 启动限制。长序列的双向注意力仍有二次计算和中间状态，其内存不能按“无 KV”当作零。

Cross-encoder reranker 把每个 query-document 对作为一条序列联合编码，通常有全自注意力，不必使用独立的 cross-attention 层。候选数增加的是序列数，候选长度改变每条序列的注意力成本。按长度分桶和动态组批可以提高利用率，但应把等待时间计入在线 SLO。

混部不是先验禁止项：embedding 可能与 decode 争抢同一带宽，也可能在调度隔离下利用余量；应在固定干扰预算下测两类负载的吞吐和尾延迟。量化是否加速取决于当前瓶颈与实际计算通路。不写持久 KV 只消除一种状态依赖，并不保证恒定算力受限或无限批处理。

2.4 三个区域：谁在限制 decode

上一节只考虑了权重。把 KV 项也放进来，decode 一步的总字节数为

$$Q_{\text{step}} = \underbrace{W}_{\text{权重，}\ W = Nb_w} + \underbrace{B \cdot kL\ell}_{\text{KV 读取}}, \tag{2.6}$$

先用线性层主导代理 $F_{\text{step}} \approx 2NB$（$N$ 为参数量）。它忽略注意力约 $4LBd\ell$ FLOP、KV 写入、激活与通信，仅用于定位数量级。主例 $\ell = 4096$ 时注意力额外约 10.7 GFLOP/请求/步，约为 140 GFLOP 的 7.7%。该代理下的计算/访存比较为

$$\frac{W + BkL\ell}{B_{\text{mem}}} \gtrless \frac{2NB}{P}. \tag{2.7}$$

式 (2.7) 的左边随 B 线性增长（截距为权重量），右边也随 B 线性增长（过原点）。两条直线的斜率之比决定了一切：当 $kL\ell/B_{\text{mem}} < 2N/P$ 时，存在一个有限的交叉点；否则左边永远在上方。

▶ 三个边界，三个区域

取 Llama-3-70B（$N = 70\times10^9$，$W = 140$ GB）、H100 SXM（$P = 989$ TFLOPS，$B_{\text{mem}} = 3.35$ TB/s）、$kL = 320$ KiB/token。

边界一：KV 读取追平计算的上下文长度。令每请求的 KV 读取时间等于计算时间：

$$\ell^* = \frac{2N \cdot B_{\text{mem}}}{kL \cdot P} = \frac{2\times70\times10^9\times3.35\times10^{12}}{327{,}680\times989\times10^{12}} \approx 1{,}448 \text{ token}.$$

边界二：权重读取追平计算的 batch（$\ell \to 0$）。

$$B_{\text{wc}} = \frac{P \cdot W}{2N \cdot B_{\text{mem}}} = \frac{989\times10^{12}\times140\times10^9}{2\times70\times10^9\times3.35\times10^{12}} \approx 295.$$

这个数与式 (2.2) 的机器平衡点 295 完全一致，而这不是巧合。把式 (2.3) 代入 $I = I^*$ 得 $B_{\text{wc}} = I^* b_w/2$，FP16 下 $b_w = 2$，于是 $B_{\text{wc}} = I^* = 295$。**对权重主导区，「摸到平衡点所需的 batch」与「机器平衡点」在数值上相等**（每元素 2 字节时）——这也是为什么 295 这个数在本书里反复出现：它既是 FLOP/byte，也是「70B 模型要多少并发才不浪费算力」。

顺带得到一个可迁移的换算：权重换成 INT4（$b_w = 0.5$）后 $B_{\text{wc}} = 295\times0.25 \approx 74$。量化的作用就是把这条竖线往左推，让同样的硬件在更小的 batch 下进入算力受限区——这是第 4 章的起点，也是容量压缩可能增加可接受并发，但是否改善单请求延迟或吞吐，仍由两项时间及 SLO 共同决定，不能把延迟与吞吐分别归给独立物理量。

边界三：KV 读取追平权重读取的 batch。

$$B_{\text{kv}} = \frac{W}{kL\ell} \xrightarrow{\ell=8\text{k}} \frac{140\times10^9}{2.68\times10^9} \approx 52.$$

这三个边界比较的是不同对象，不能合并成“长上下文必然 KV 主导”的单一标签。在此代理模型内，若 $\ell < \ell^*$，计算/总读取的交叉 batch 为 $B_{\text{comp}} = W/(2N/I^* - kL\ell)$；若 $\ell \geq \ell^*$，该代理的读取项始终高于线性层计算项。**但权重或 KV 谁占主要流量，另由 $B_{\text{kv}} = W/(kL\ell)$ 决定**：即使 32k 上下文，$B = 1$ 时仍是 140 GB 权重对 10.74 GB KV。故量化权重仍有收益，不能称“长上下文量化无效”。

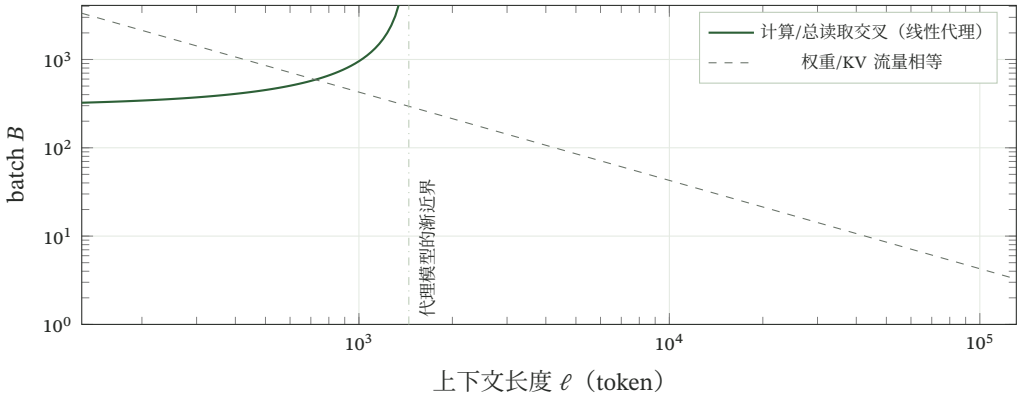


图 2.1 两条不同的交叉线。绿线以上计算项可能更大；虚线以上 KV 流量大于权重，以下则相反。竖线只说明忽略注意力计算的代理模型无法靠增大 batch 越过计算拐点。本图没有计入多卡通信、KV 头复制或 SLO，所有曲线均为解析自算。

! 常见误解

带宽受限不等于加 batch 毫无收益。若 $t = (W + Bq_{\text{KV}})/B_{\text{eff}}$，输出吞吐 $B/t = B_{\text{eff}}/(W/B + q_{\text{KV}})$ 仍随 batch 单调上升，但趋向有限平台，单请求 ITL 则增加。交叉点之后是边际收益递减，而非突然失效。容量、质量和 SLO 才决定是否值得继续增大 batch。

2.5 显存墙：KV cache 的账本

KV cache 的字节数由下式给出：

$$M_{\text{KV}} = 2 \cdot L \cdot n_{\text{kv}} \cdot d_{\text{head}} \cdot \ell \cdot b_{\text{kv}} \cdot B, \tag{2.8}$$

其中 L 为层数，n_{kv} 为 KV 头数（MHA 时等于注意力头数，GQA/MQA 时更小），d_{head} 为每头维度，ℓ 为序列长度（含 prompt 与已生成部分），b_{kv} 为每元素字节数，B 为并发请求数，系数 2 对应 K 与 V 两份。更紧凑地写成「每 token 每层 KV 字节数」：

$$k = 2 \cdot n_{\text{kv}} \cdot d_{\text{head}} \cdot b_{\text{kv}} \quad \text{(byte/token/layer)}. \tag{2.9}$$

▶ Llama-3-70B 的 KV 账本

Llama-3-70B 的公开配置：$L = 80$ 层，注意力头数 64，每头维度 128，GQA 的 KV 头数为 8【官方文档】[5]。FP16 下 $b_{\text{kv}} = 2$。

$$k = 2 \times 8 \times 128 \times 2 = 4096 \text{ byte/token/layer},$$
$$kL = 4096 \times 80 = 327{,}680 \text{ byte/token} \approx 320 \text{ KiB/token}.$$

于是：

- 一个 8k 上下文的请求：320 KiB × 8192 ≈ 2.5 GiB；
- 一个 32k 上下文的请求：≈ 10 GiB；
- 8 张 H100（80GB）：权重 70B × 2B ≈ 140 GB，TP=8 后每卡约 17.5 GB，剩余约 60 GB/卡可用于 KV，共约 480 GB，若教学上预算 480 GB 用于 KV，则为 447 GiB，只能容纳约 44 个 32k 请求（未计页尾）。

若仅按 KV 容量计，FP8 KV（$b_{\text{kv}} = 1$）使可容纳的 token 数近似翻倍，需另计 scale 与质量；若改用 MLA 结构，每 token KV 量可再降一个量级，详见第 3 章。

把这两项随 batch 的变化画在一起，交点标记权重与 KV 流量贡献相等的位置（图 2.2）。

式 (2.8) 与式 (1.1) 合起来，给出了推理系统的「不可能三角」：并发数、上下文长度、模型规模三者不能同时任意大。任何声称打破这个三角的技术，实际上只是在降低 k、降低 b_{kv}，或者把一部分 KV 挪到别的地方（第 11 章的分级存储）。

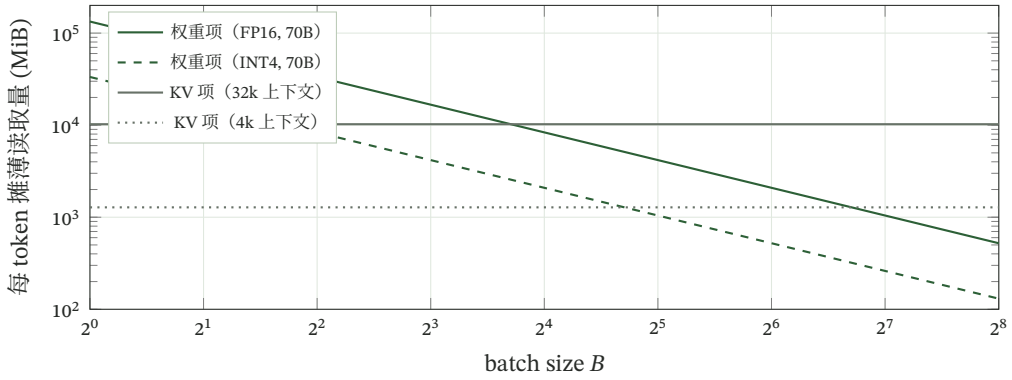


图 2.2 每 token 摊薄读取量的两项随 batch 的变化。权重项按 $1/B$ 下降，KV 项与 batch 无关但随上下文长度线性上升。两条曲线的交点标记权重与 KV 流量相等的位置：FP16 下 32k 上下文的交点约在 $B \approx 13$，而 4k 上下文时推迟到 $B \approx 104$【自算，140 GB/(320 KiB × ℓ)】。含义是：一个 32k 的请求只要凑够 13 个并发，它吃掉的 KV 流量就与全部模型权重相当；此后批处理仍提高吞吐，但增益逐渐趋缓。KV 压缩减少读取量；分级存储主要扩容量，如果每步都需从慢层取回全部 KV，反而可能降低性能，二者不能当作同一种带宽优化。

2.6 能耗与单位成本的条件估算

设每个有效输出 token 摊薄流量为 q，同一节点的聚合有效带宽为 B_{eff}，有效计算量为 f、聚合计算速率为 P_{eff}。如果功率在执行期近似固定为 Φ，则带宽主导的条件估算为

$$E_{1\text{M}} \approx 10^6 \frac{q}{B_{\text{eff}}} \Phi \quad \text{(J)}. \tag{2.10}$$

若 Φ 是有证据支持的最低功率，且 q 为不可消除的流量，可写相应能耗下界；**TDP、额定整机功率和假设有效带宽不能自动组成物理能耗下界**。实际测量应积分 $E = \int \Phi(t)\, dt$，并注明含不含主机、网络和制冷。

▶ 主例的 8 卡节点能耗账

采用第 1 章的 70B、BF16、4096 上下文，固定 $B = 8$。忽略 attention 之外的激活与通信流量，则

$$Q_{\text{step}} = 140 \times 10^9 + 8 \times 327680 \times 4096 = 150.7374 \text{ GB}, \qquad q = Q_{\text{step}}/8 = 18.8422 \text{ GB/token}.$$

教学假设每卡有效带宽 2.5 TB/s、8 卡聚合 20 TB/s，执行期整节点平均功率 8 kW。于是 decode 步时间 150.7374/20000 = 7.537 ms，聚合 token 时间为 0.942 ms。

$$E_{1\text{M}} = 10^6 \times 0.0009421 \times 8000 \approx 7.54 \times 10^6 \text{ J} \approx 2.09 \text{ kWh}.$$

电价取中国大陆教学场景 0.6 元/kWh（假设价，非 2026-10-22 实际报价），得到约 1.26 元/百万输出 token【自算】。这里没有计 prefill、空闲期、PUE、重试与质量失败。b_w、KV 精度、batch、有效带宽或功率改变时都必须重算；不能把这个数当 API 成本底线。

电费占总成本多少取决于购机/租赁、折旧、利用率、电价与 SLO，应与第 9、17 章的同一范围账本相接。GPU utilization、MFU 与 HBM 带宽占用的含义不同；低 MFU 可能正是带宽受限的正常表现，也可能来自 kernel 或 CPU 空闲，排查需要 profiling 而非只看一个利用率百分数。

练习：先检查资源范围

把主例的并发从 8 改为 32，分别计算每步字节、每输出 token 字节与单请求 ITL 的理想值。**提示：**节点每步仍只读一遍 140 GB 权重，但 KV 项乘 4；不要让单卡带宽去除节点总流量。再把上下文改为 4608，检查输出阶段的 KV 增长是否改变你给出的 50 ms 约束判断。这些解析值只用来筛除不可行配置，通过仍需服务级实测。

未解问题

1. **roofline 对多卡还成立吗？**单卡的 roofline 已经很清楚，但一旦引入 TP/EP，通信成为第三个维度，现有的「计算/带宽」二维模型无法解释为什么某些配置下增加并行度反而变慢。目前缺乏一个把通信拓扑、批次大小、模型切分联合起来的统一上界模型。
2. **有效带宽如何预测？**式 (2.10) 里 B_{eff} 的取值高度依赖访问模式（是否合并、是否随机、L2 命中率），不同 kernel 的 B_{eff} 可以相差 2 倍以上。在部署前准确预测 B_{eff}，目前仍依赖对具体 kernel 的实测。

第 3 章　注意力谱系与 IO 优化

本章结论

(1) KV cache 的成本只有一个自变量：每 *token* 每层缓存多少元素。MHA、MQA、GQA 是在 KV 头数上做除法，除数是共享比 g，代价是组内的头被迫共享子空间；MLA 改成在秩上做除法，把 $2n_h d_h$ 压成 $d_c + d_h^R$。按 DeepSeek-V3 的配置每 token 只有 68.6 KiB，约为同组 query 头配置的 MHA 的 1/57；质量比较来自特定训练消融，不能只由缓存比推出。

(2) FlashAttention 不改变注意力的数学。它在实数算术下计算同一个精确注意力函数，浮点结果因累加顺序不同可有差异；注意力矩阵乘的渐近 FLOPs 不变，改变的是读写顺序：把 $N \times N$ 的中间矩阵留在 SRAM 里，在下文注明的 SRAM 模型中降低 HBM 流量的系数，而非消除序列长度的二次项。因此它能治显存墙和带宽墙，治不了长上下文的算力墙。

(3) 稀疏与线性注意力是两种方向相反的减法。稀疏保留 softmax、缩小每个 query 的可见集；线性注意力与 SSM 保留全可见集、把 softmax 换成核函数与固定大小的状态。分界线是检索密集度：逐字召回对固定状态容量与稀疏选择都构成压力，应按目标长度和任务验证。

3.1　KV cache 的通式：先把账本写成结构超参的函数

第 2 章把 KV 显存写成 $2L\, n_{\text{kv}} d_h \ell b_{\text{kv}} B$，那个式子适用于显式缓存 K/V 的 MHA、MQA 与 GQA；MLA 需要另一种布局。真正与结构无关的量是「每 token 每层缓存多少元素」c_{kv}：

$$c_{\text{kv}} = \begin{cases} 2\, n_{\text{kv}}\, d_h, & \text{MHA / MQA / GQA,} \\ d_c + d_h^R, & \text{MLA,} \end{cases} \qquad k = c_{\text{kv}}\, b_{\text{kv}} \quad \text{(byte/token/layer).} \tag{3.1}$$

n_{kv} 是缓存 K/V 的头数：MHA 时等于 n_h，MQA 时为 1，GQA 时为分组数 n_g。系数 2 来自 K 与 V 是两份独立张量。MLA 一行没有系数 2，这是全章最重要的一处差别—— *MLA* 缓存既不是 K 也不是 V，而是一个同时生成两者的潜向量，一份存储供两个用途。

式 (3.1) 把降 KV 的合法手段一次列完：降 n_{kv}（GQA/MQA，最小到 1）、降 b_{kv}（FP8 直接减半，见第 4 章）、让 K/V 共享（MLA，把「头数」与「缓存量」解耦）、降 ℓ（滑窗与驱逐）。减小 d_h 也能降缓存，但会改变模型结构与表达容量，通常需要训练验证。

3.2　从 MHA 到 MQA 再到 GQA：在头数上做除法

把式 (3.1) 第一行的两种结构相除，立即得到压缩比：$k_{\text{MHA}}/k_{\text{GQA}} = n_h/n_g = g$，MQA 则是 $k_{\text{MHA}}/k_{\text{MQA}} = n_h$。这里 $n_g = n_{\text{kv}}$ 是 KV 组数，$g = n_h/n_g$ 是每组共享的 query 头数，二者不能同称“分组数”。在其余结构、精度相同且不复制 KV 时，缓存按 $1/g$ 缩小；$g = 1$ 是 MHA，$g = n_h$ 是 MQA。这个比例不含层数、长度与精度，但工程流量还受布局和复制影响。

MQA 的动机是带宽而不是显存。Shazeer 在 2019 年指出，增量解码时每一步都要把整个 K/V 张量从显存搬到计算单元，搬运量与头数成正比；压成一对 K/V，解码访存量就降为 $1/n_h$【论文】[6]。而 decode 的算术强度只有约 1 – 2 FLOP/byte（第 2 章），离 H100 的平衡点 295 差两个数量级，所以「少读字节」几乎直接等于「少花时间」。

代价落在表达力上。MQA 之后所有 query 头共享同一对 K, V，等价于把 KV 投影矩阵的秩压到 d_h：不同的头再也无法「各看各的子空间」。Shazeer 报告质量只有轻微下降【论文】，GQA 的论文说得更直接—— MQA「可能导致质量退化」，并给出一个实用配方：用原始预训练算力的 5% 把已有 MHA checkpoint *uptrain* 成 GQA【论文】[7]。这个 5% 说明结构改造的代价可以远小于重训。

共享使不同 query 头使用相同 K/V 投影，减少了可独立学习的 KV 子空间。这是一种结构约束，不是质量必然按比例下降的定理；训练可能找到有效的共享表示。原稿所称“梯度汇聚必然放大等效步长”没有控制损失归一化与优化器，不能作为结论。Ainslie 等人的 5% uptraining 配方是其模型和训练设置下的结果，不保证任意 checkpoint 可直接套用。

表 3.1 同一模型规模下四种注意力结构的 KV 账本。超参取 $L = 80$、$n_h = 64$、$d_h = 128$、FP16（$b_{\text{kv}} = 2$），即 Llama-3-70B 的规模【官方文档】[5]；MLA 行取 DeepSeek-V2/V3 的压缩维度 $d_c = 512$、$d_h^R = 64$。所有数值为【自算】。MQA 的 1/64 是用表达力换来的，MLA 的 1/28.4 是用秩换来的，两者的单价不同。

结构	c_{kv}（元素/层）	每层（KiB）	每 token（KiB）	32k 上下文（GiB）	相对 MHA
MHA（$n_{\text{kv}} = 64$）	16384	32	2560	80.0	1
GQA-8	2048	4	320	10.0	1/8
GQA-2	512	1	80	2.50	1/32
MQA	256	0.5	40	1.25	1/64
MLA（$d_c = 512$）	576	1.125	90	2.81	1/28.4

表 3.1 把比例换成了绝对值。把这些数字放到一台 8 卡 H100（640 GB 显存，70B FP16 权重占 140 GB，减去激活与碎片后可留给 KV 的约 420 GiB）上，32k 上下文的并发上限分别是：MHA 约 5 个、GQA-8 约 42 个、GQA-2 约 168 个、MQA 约 336 个、MLA 约 149 个【自算】。MHA 的并发容量显著更低，但是否可提供多租户服务还取决于 QPS 与 SLO；不能仅用并发数把服务判为不可能。但 GQA-8 的 42 个并发在 128k 上下文下会掉到 10 个左右，于是有了下一个问题：能不能在不砍头的前提下继续降 KV。

3.3 MLA：在秩上做除法

GQA 的天花板很清楚：想再降 KV 只能继续加大 g，但共享约束越强越需要重新检验质量。MLA 换了一个除法对象——不再砍头，而是把「每层要缓存的 K/V」看成一个可以被低秩分解的量。这个视角来自一个简单事实：如果从隐状态到 (K, V) 的映射在数值上接近低秩，那么存一个压缩编码、在用时重建，信息损失可以很小；而「让不同的头共享同一对 K/V」是一种结构性约束，可学习的共享表示是否足够，需要受控训练实验判断。

MLA 的具体做法是联合压缩加解耦位置编码：

$$c_t^{KV} = W^{DKV} h_t \in \mathbb{R}^{d_c}, \qquad k_t^C = W^{UK} c_t^{KV}, \qquad v_t^C = W^{UV} c_t^{KV}, \tag{3.2}$$

$$c_t^Q = W^{DQ} h_t, \qquad q_t^C = W^{UQ} c_t^Q, \qquad k_t^R = \text{RoPE}(W^{KR} h_t), \qquad q_t^R = \text{RoPE}(W^{QR} c_t^Q), \tag{3.3}$$

其中 $W^{DKV} \in \mathbb{R}^{d_c \times d}$ 是下投影，$W^{UK}, W^{UV} \in \mathbb{R}^{d_h n_h \times d_c}$ 是上投影，$k_t^R \in \mathbb{R}^{d_h^R}$ 在所有头之间共享。注意力分数由内容段与位置段拼接而成：

分数为 $s_{t,s} = \langle [\, q_t^C ; q_t^R \,], [\, k_s^C ; k_s^R \,] \rangle / \sqrt{d_h + d_h^R}$，两段各管一件事。

关键的一步是吸收（absorption）。把上两行的上投影与 query 投影代入内积：

$$(q_t^C)^\top k_s^C = (W^{UQ} c_t^Q)^\top (W^{UK} c_s^{KV}) = \left(W^{UK\top} W^{UQ} c_t^Q\right)^\top c_s^{KV}. \tag{3.4}$$

右端只含 c_s^{KV}：W^{UK} 被折进了 query 侧的变换。同理 W^{UV} 可以折进输出投影 W^O。于是推理时根本不需要把 k^C, v^C 物化出来，缓存里只有 c_t^{KV} 和 k_t^R，共 $d_c + d_h^R$ 个元素，每个头用自己的一对 $(W_i^{UK\top} W_i^{UQ})$ 去「读」这个共享潜向量。这就是式 (3.1) 第二行的来历，也是 MLA 与 MQA 最本质的区别。

RoPE 是这条路径上需要单独处理的主要障碍；位置编码、量化与布局也会影响吸收实现。RoPE 是位置相关的旋转：若把它直接作用在 k^C 上，W^{UK} 就会与一个随位置 s 变化的旋转矩阵相乘，式 (3.4) 的右端不再成立——吸收要求「位置无关的线性变换」。MLA 的解法是把位置信息拆成独立的一条 64 维通道：它对所有头共享、必须缓存、但不参与吸收。内容走可吸收的低秩路径，位置走不可吸收的小通道。这解释了为什么 $d_c = 512$ 之外还要多存 64 个元素，而不是干脆把这 64 维也压进潜向量。

▶ DeepSeek-V3 的每 token KV

配置：$L = 61$，$n_h = 128$，$d_h = 128$，$d_c = 512$，$d_h^R = 64$，FP16（$b_{\text{kv}} = 2$）【论文】[8]。
每层元素数 $c_{\text{kv}} = 512 + 64 = 576$，于是

$$k = 576 \times 2 = 1152 \text{ byte/token/layer}, \qquad kL = 1152 \times 61 = 70{,}272 \text{ byte/token} \approx 68.6 \text{ KiB/token}.$$

若同一组超参换成 MHA（$2n_h d_h = 32768$ 元素/层）：

$$k_{\text{MHA}} L = 32768 \times 2 \times 61 = 3{,}997{,}696 \text{ byte/token} \approx 3.81 \text{ MiB/token},$$

比值 3,997,696/70,272 ≈ 56.9，与表 3.1 的 1/28.4 不同，是因为那张表的 query 头数为 64、此例为 128；两边相同的层数会在比例里约掉。
一个 128k 上下文的请求：$70{,}272 \times 131{,}072 \approx 9.21 \times 10^9$ byte ≈ 8.6 GiB；同规模 MHA 则是 488 GiB，单机装不下。

为什么低秩共享仍可能保留质量？下面解释结构上的可能性，质量结论仍以训练消融为准。

第一，瓶颈的位置不同。MQA/GQA 的瓶颈是「头与头之间不能有差异」，这是结构性约束；MLA 的瓶颈是「每层 K/V 的秩不超过 d_c」，而 d_c 是可调超参（V2 取 $4d_h = 512$）。对一个固定且可精确低秩分解的线性映射，足够大的 d_c 可以表示它；训练得到的 MLA 并不自动等于原 MHA 的最优低秩近似，不能据此保证无损。

第二，每头仍有独立的读出矩阵。共享的是潜向量，不是最终的 K/V：$k_{t,i} = W_i^{UK} c_t^{KV}$，128 个头各有一个 W_i^{UK}；GQA 的组内头连读出都一样。这也是为什么 V2 论文能给出那句很有分量的话：MLA 的 KV 缓存量只相当于 GQA-2.25，但在同等规模下性能强于 MHA【论文】[9]。同样的 576 个元素，GQA 只能做到 2.25 组，MLA 却超过了 MHA ——差别就在「共享的是基底还是读出」。

第三，位置通道独立、不占内容的秩。

DeepSeek-V2 的整机结果是：相对 DeepSeek 67B，KV cache 减少 93.3%，最大生成吞吐提升到 5.76 倍，训练成本降低 42.5%【论文】（条件：8.1T token 预训练、支持 128K 上下文、对照模型为稠密 67B）。这组数字常被引用成「MLA 让推理快 5.76 倍」，这是误读—— 5.76 倍是最大生成吞吐在特定 batch 与序列长度下的整机结果，其中还包含 MoE 的贡献。

常有人把 MLA 当成「MQA 换了个名字」，但两者的 KV 量虽同量级（每层 0.5 KiB 对 1.125 KiB），代价完全不同：MQA 是 $n_{\text{kv}} = 1$ 的结构共享，丢掉的头间差异不可恢复；MLA 是秩为 d_c 的数值共享，每头保留自己的读出。同样缓存预算下的质量差异应引用具体消融，不能对所有 GQA 与 MLA 作通用排序。

代价也要说清楚。其一，吸收是位置解耦后的线性恒等式，不只在 decode 成立；prefill 或训练是否使用它取决于矩阵形状、重算、激活显存和 kernel 效率，常采用不同路径。其二，低秩瓶颈改变了输出尺度，V2 明确要在压缩潜向量之后加 RMSNorm、在宽度瓶颈处乘缩放因子来稳定训练【论文】。其三，MLA 的 KV 布局与 GQA 不同，不能直接套用现成的 paged attention kernel。

3.4 FlashAttention：把那个二次矩阵从 HBM 里删掉

对单头、无因果掩码的示范，注意力为 $S = QK^{\top}$、$P = \text{softmax}(S)$、$O = PV$。假设朴素实现把 S, P 各写回并读回一次，元素 b 字节，Q/K/V/O 各 $N \times d$，忽略缓存命中，则前向流量为

$$Q_{\text{std}} = 4N^2 b + 4Ndb. \tag{3.5}$$

前一项是 S 与 P 的两写两读，后一项是读 Q/K/V 及写 O；不能在 QK 阶段先计一次 V，又在 PV 阶段重复计一次。$N = 4096, d = 128, b = 2$ 时，中间矩阵流量 134.2 MB，Q/K/V/O 合计 4.19 MB，其中 Q/K/V 为 3.15 MB，约 97%的该流量模型来自中间矩阵【自算】。因果掩码、融合 softmax 或其他基线会改变实际读写次数。

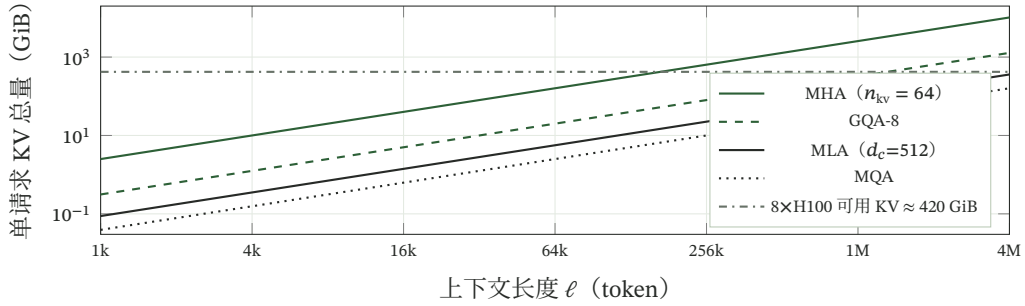


图 3.1 四种结构在 70B 规模下的 KV 总量随上下文长度的变化（双对数）。曲线与水平线的交点就是「这台机器能开多长」。MHA 在 17 万 token 处撞线，GQA-8 在 137 万处，MLA 在 489 万处。这是单请求、假设 420 GiB 全用于 KV 且模型支持该长度的容量估算；实际可能先遇到训练长度、$O(N^2)$ 计算、其他并发或延迟预算，于是问题从「存不下」变成「算不完」，这正是后文稀疏注意力要接手的地方。

FlashAttention 把 Q 按 B_r 行、K/V 按 B_c 行分块，并用在线 softmax 合并各块：

$$m_i^{(j)} = \max\left(m_i^{(j-1)}, \max_k s_{ik}^{(j)}\right), \tag{3.6}$$

$$\ell_i^{(j)} = e^{m_i^{(j-1)} - m_i^{(j)}} \ell_i^{(j-1)} + \sum_k e^{s_{ik}^{(j)} - m_i^{(j)}}, \tag{3.7}$$

$$\tilde{o}_i^{(j)} = e^{m_i^{(j-1)} - m_i^{(j)}} \tilde{o}_i^{(j-1)} + \sum_k e^{s_{ik}^{(j)} - m_i^{(j)}} v_k^{(j)}. \tag{3.8}$$

初始化 $m = -\infty, \ell = 0, \tilde{o} = 0$，最后 $o_i = \tilde{o}_i / \ell_i$。旧状态按新最大值重新缩放，因此在实数算术下等于整体 softmax；实际浮点运算的顺序不同，结果在容差内比较，不能称逐位相同。

若每个 Q 块把整份 K/V 从 HBM 读取一次，并将 Q 及输出各读/写一次，则

$$Q_{\mathrm{FA}} \approx \frac{2N^2 db}{B_r} + 2Ndb. \tag{3.9}$$

这是特定循环与缓存假设的流量模型，不是所有 FA kernel 的必经流量。若 SRAM 预算为 M 字节且块形状可取 $B_r = \Theta(M/(db))$，数量级写成

$$Q_{\mathrm{FA}} = \Theta\left(Ndb + \frac{N^2 d^2 b^2}{M}\right). \tag{3.10}$$

原论文常用元素个数计 SRAM；换成字节必须同时换算 b，不能只写 $\Theta(N^2/M)$ 而隐藏量纲。该结论有 SRAM 容量区间与 tile 可行性的前提，实际还要容纳累加器、softmax 状态与流水缓冲【论文】[1]。

algorithm: tiled exact attention (pseudocode)

```
for each query block i:
    load Qi
    m = -inf; l = 0; O = 0
    for each key/value block j:
        load Kj, Vj
        S = scale * Qi @ Kj.T + mask(i, j)
        m_new = maximum(m, rowmax(S))
        alpha = exp(m - m_new)
        P = exp(S - m_new)
        l = alpha * l + rowsum(P)
        O = alpha * O + P @ Vj
        m = m_new
    write O / l
```

▶ 单头 IO 模型与整 GPU 实测的边界

按 H100 SM 最大共享内存预算 228 KiB 作硬件参照【官方文档】[3]，但不要把它当成一个 kernel 可自由使用的全部 SRAM。仍取 $N = 4096, d = 128, b = 2, B_r = 128$：$Q_{\text{std}} \approx 138.4$ MB，$Q_{\text{FA}} \approx 69.2$ MB，两次矩阵乘共 $4N^2d = 8.59$ GFLOP，模型强度约 62 与 124 FLOP/byte【自算】。它说明避免物化中间矩阵的价值，不能证明 FA 在 H100 上永远带宽受限。L2 复用、寄存器布局、多个头和 CTA 并行会改变有效 HBM 流量；因果注意力又跳过上三角。FA3 的具体性能必须看它自己的 FP16/FP8 测量与相同精度峰值。

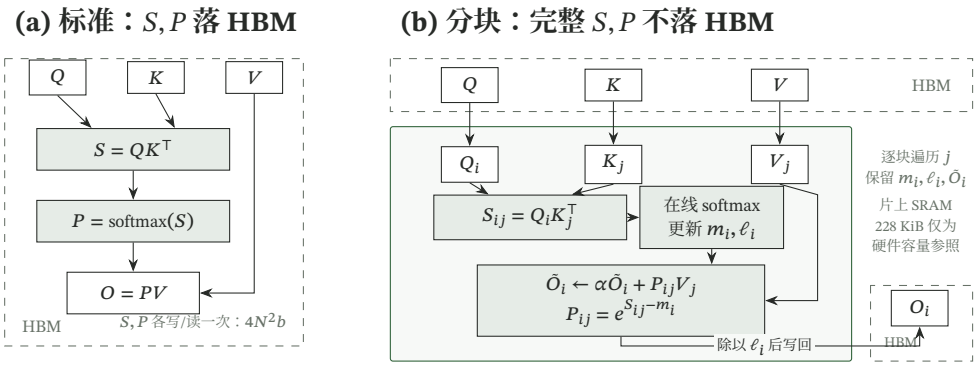


图 3.2 两种实现的访存路径。左图里 $N \times N$ 的 S 与 P 各要经历一次写一次读，折合约 134 MB 流量，而 Q/K/V 合计约 3.15 MB；右图把 S_{ij} 限制在 SRAM 内，K, V 被反复读但每次都拼进同一个累加器，完整 $N \times N$ 中间矩阵不落 HBM。注意右图的读写总量并没有变成与 N 无关：它仍是 N^2 阶，只是系数从 $4b$ 降到 $2d^2b^2/M$，而 K, V 的重读次数由 N/B_r 决定。

图 3.2 把两种实现的访存路径画在一起：左边是标准实现的 S 与 P 各写一遍读一遍，折合 134 MB；右边把 S_{ij} 限制在 SRAM 内，完整 $N \times N$ 中间矩阵不落 HBM。注意右图的读写总量并没有变成与 N 无关——它仍是 N^2 阶，只是系数从 $4b$ 降到 $2d^2b^2/M$，K, V 的重读次数由 N/B_r 决定。这个模型中两种方法的主项均随 N^2 增长，其流量比趋向常数；不能单凭这两个式子推出 FA 加速比必随序列长度变大而收窄。

3.5 FA2 与 FA3：同一个 IO 复杂度，不同的机器利用率

FA1 只解决了「少读」，没解决「读得满」：它在 A100 上只有 25–40% 的峰值利用率【论文】[10]，原因不在算法而在工作划分——并行度只开在 (batch, head) 两维，长序列配小 batch 时大量 SM 空闲，且每块都要做除法与重标定。

FA2 的三处改动都指向同一件事——让 Tensor Core 不空转。其一，减少 non-matmul FLOPs：O 保持未归一化，只在循环结束后除一次 ℓ，重标定合并进矩阵乘的累加。其二，沿序列维并行：即使只有一个头，也把不同 Q 块分给不同 thread block，长序列时 SM 才能填满。其三，warp 级划分：一个 block 内 4 个 warp 各管 32 行 Q，K/V 的循环在 warp 内完成，shared memory 只用于必须交换的部分。

论文报告在 A100 上 attention kernel 达到理论峰值的 50–73%，相对 FA1 约 2 倍；GPT-style 1.3B/2.7B 模型的端到端训练最高为 225 TFLOP/s/卡（72% MFU），相对 FA1 的训练加速最高约 1.3 倍【论文】[10]。两种倍率的测量范围不同。

FA3 假设 IO 已不是主要矛盾，转而榨干 Hopper 的异步能力。异步：TMA 负责 HBM 到 SRAM 的块拷贝、WGMMA 负责矩阵乘，两者都是异步指令，与 CUDA core 上的 softmax 重叠。*warp specialization*：producer warpgroup 只发 TMA，consumer warpgroup 只算 WGMMA，再用两个 consumer warpgroup 做 ping-pong，使「第 j 块 softmax」与「第 $j+1$ 块 GEMM」同时进行。*FP8*：块级量化加 Hadamard 非相干处理，在论文含 outlier 的 FP8 测试中，数值误差约为逐 tensor 量化基线的 1/2.6【论文】，不构成任意输入的误差保证。

在 H100-80GB-SXM5 上，FP16 前向达到 740 TFLOP/s（75% 利用率），FP8 接近 1.2 PFLOP/s；FP16 前向相对 FA2 加速 1.5–2.0 倍，反向为 1.5–1.75 倍。作为对照，论文报告 FA2 在 H100 上约 35% 利用率【论文】[11]。

FA2/FA3 保留精确密集注意力的主要矩阵乘 FLOP 数和 IO 渐近复杂度，但会改变非矩阵乘操作、重算、访存常数和并行划分。更高的实现效率可以降低同负载的墙钟时间和成本，仍不能消除密集注意力随长度的二次增长。

! 常见误解

「FlashAttention 让长上下文变便宜」。它降低的是 HBM 流量，不是 FLOPs；prefill 的计算量仍是 $4N^2d$。当 N 从 4k 涨到 128k，FLOPs 涨 1024 倍，HBM 流量同样涨 1024 倍——两者同阶。FA 之后注意力仍然是长上下文的主要成本，只是从「搬不动」变成了「算不完」。眞正要削掉这个二次项，只有改可见集（稀疏）或改结构（线性 / SSM）。

3.6 稀疏注意力：缩小每个 query 的可见集

统一的写法是：把可见集 $\mathcal{S}_t \subseteq \{1, \dots, t\}$ 从「全部前缀」换成大小为 k 的子集，计算量从 $O(N^2d)$ 变成 $O(Nkd)$，KV 读取量按同比例下降。各种做法的差别在于「怎么选」以及「选完还能不能训练」。

滑窗（sliding window） 取 $\mathcal{S}_t = \{t - W + 1, \dots, t\}$，每层只连前 W 个 token【论文】[12]。牺牲的是单层感受野：跨层堆叠后理论跨度是 $W\times$ 层数，信息传递路径长度却是 $O(N/W)$。Mistral 7B 取 $W = 4096$，32 层给出约 131k 的理论跨度，并用 rolling buffer 把缓存固定为 W，在 32k 序列上把 KV 显存降 8 倍且不损质量【论文】[13]。它在局部性强的负载上才赚：代码补全、流式日志、近场指代。

attention sink Softmax 的每一行必须归一化。当窗口里没有「值得注意」的 token 时，模型会把概率质量倒给序列开头几个 token；StreamingLLM 永远保留最初 4 个 token 的 KV 再拼上滑窗【论文】[14]。它牺牲了中段的一切，没有任何召回能力，但在无限流式输入上很赚：论文报告 Llama-2、MPT、Falcon、Pythia 在 400 万 token 上困惑度保持稳定。它解释了窗口化失效的一个数值/注意力分配机制；丢失远端任务信息仍是独立限制，不能把长文档检索失败全部归给 attention sink。

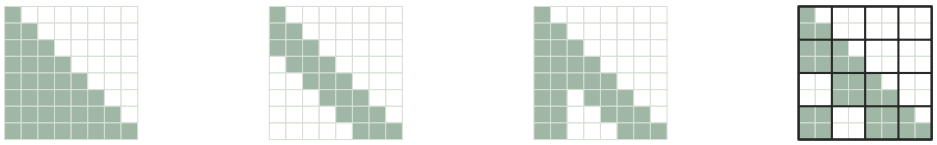
块稀疏（MoBA） 把 KV 按块划分，用一个门控网络给每个 query 对所有块打分，只取 top-k 块做注意力；它遵循「less structure」原则，不预设滑窗或 sink，且可在 full 与 sparse 之间切换【论文】[15]。代价包括块级选择开销及块粒度引入的近似，小模型上收益常被门控开销抵消。它赚在超长且注意力高度集中的负载上，实际产品是否使用何种配置需独立发布证据，本书不据论文名称推断部署。

原生可训练稀疏（NSA） 三条并行分支：压缩分支把块压成粗粒度摘要，选择分支做块级 top-n，滑窗分支保留局部精确；三路输出用学习到的门控加权合并，且都按块对齐以匹配 GPU 的算术强度【论文】[16]。它最关键的主张是从零预训练而非事后套用，因此在 64k 序列上 decode、forward、backward 三个阶段相对全注意力都有数倍加速。它的收益只在长上下文训练与推理这条链上兑现：短序列下门控开销会吃掉省下的算力。代价是必须重训，实现是三套 kernel。

DSA 用一个「闪电索引器」给每个 (t, s) 打分：

$$I_{t,s} = \sum_{j=1}^{H^I} w_{t,j}^I \, \mathrm{ReLU}\big(\langle q_{t,j}^I, \, k_s^I \rangle\big), \tag{3.11}$$

其中索引器头数少、维度小、可用 FP8 计算；取 top-k（训练时 $k = 2048$）后，只在这些位置上做 MLA 的核心注意力。于是核心注意力从 $O(L^2)$ 降到 $O(Lk)$，而索引器仍是 $O(L^2)$，只是常数极小【论文】[17]。它需要继续训练（从 V3.1-Terminus 到 V3.2），而且 KV 容量一点没降——所有 token 的潜向量仍要留着供索引器打分。



(a) 全因果　(b) 滑窗 $W = 3$　(c) sink + 窗口　(d) 块稀疏 top-k

横轴：key 位置；纵轴：query 位置；深色 = 可见

图 3.3 四种可见集模式。从 (a) 到 (d)，被填的格子数从 $N^2/2$ 降到约 $2N$ 量级，softmax 本身没有变；差别只在「哪些格子被跳过」：(b) 靠位置先验，(c) 在位置先验上补一个全局垃圾桶，(d) 让模型自己选块——越往下越贵，也越不依赖任务假设。块粒度不是省事，而是 GPU 只能对连续块高效访存。

五种掩码的几何形状画在图 3.3 里：滑窗是一条带、sink 落在第一列、NSA 与 MoBA 是块状、DSA 是索引选出的散点。形状决定了它们各自能被哪种 kernel 高效实现——带与块能映射到规则的 tile，散点不能。

把五种做法放在一起，有两条共同规律。

第一，稀疏省的是算力和带宽，不是显存。滑窗与 sink 是例外，它们真的丢掉 KV；DSA 的潜向量、NSA 的 K/V 都还在，只是不再全部参与注意力。第二，收益与访存粒度强耦合。decode 时读 2048 个不连续的 token，可能比读 32k 个连续潜向量更慢——不连续访存浪费 32B/128B 的传输粒度，并让 Tensor Core 的块形状失效。因此块稀疏与细粒度索引需要分别评估，不能把 DSA 的 token 选择等同于规则块稀疏。

! 常见误解

「稀疏注意力一定更快」。只有当稀疏模式与 kernel 的块形状对齐、且选择开销被摊薄时，稀疏才更快；否则省下的 FLOPs 会被 gather 与索引开销吃掉。DSA 把索引器做成 FP8 加少头，正是为了让「选择」本身足够便宜；MoBA 用块级门控而不是 token 级门控，是同一个道理。

四种结构在 70B 规模下的 KV 总量随上下文长度的变化见图 3.1：按图中 420 GiB 可用 KV 预算，MHA 约 17 万 token 撞线，GQA-8 约 137 万，MLA 约 489 万。换言之，MLA 之后显存不再是长上下文的第一约束——先撞上的是 $O(N^2)$ 算力与单请求延迟，问题从「存不下」变成「算不完」。

3.7 线性注意力与 SSM：把 softmax 换掉

线性注意力去掉 softmax，把相似度写成核函数内积 $\phi(q)^\top \phi(k)$，于是求和可以重新结合：

$$o_t = \frac{\sum_{s\le t} \phi(q_t)^\top \phi(k_s)\, v_s}{\sum_{s\le t} \phi(q_t)^\top \phi(k_s)} = \frac{\phi(q_t)^\top S_t}{\phi(q_t)^\top z_t}, \qquad S_t = \sum_{s\le t} \phi(k_s) v_s^\top, \quad z_t = \sum_{s\le t} \phi(k_s). \tag{3.12}$$

状态 $S_t \in \mathbb{R}^{d_\phi \times d_v}$ 与 t 无关，所以每 token 的显存是常数，计算是 $O(N)$。代价写在式子里：softmax 被换掉了。

softmax 做了什么？它对分数做指数放大再归一化，等价于一个尖锐的、数据相关的选择：某个 key 的分数高出一截，它的权重就接近 1，其余接近 0。这个「胜者通吃」是非线性的，也是注意力能做精确检索的原因。核特征与状态容量限制可表达的注意力模式；并非所有线性注意力都无法形成尖锐选择，但固定状态无法无限保存独立 key-value 信息。

Mamba 是这条路线最成熟的代表：它让 A, B, C 变成输入的函数（选择机制），从而在保持 $O(N)$ 的同时赢得语言建模的竞争力【论文】[18]；「Transformer 实际上是一个状态数随长度增长的多状态 RNN」这个视角则把两者放进同一个框架里比较【论文】[19]。

Mamba-2 给出了 SSM 与注意力之间的严格联系。SSM 的递推是 $h_t = A_t h_{t-1} + B_t x_t$、$y_t = C_t^\top h_t$，展开得

$$y_t = \sum_{s\le t} C_t^\top \Big(\prod_{i=s+1}^{t} A_i \Big) B_s x_s \quad \Longleftrightarrow \quad Y = \big(L \circ (CB^\top)\big)X, \qquad L_{t,s} = \prod_{i=s+1}^{t} a_i, \tag{3.13}$$

其中 a_i 为标量衰减，L 是下三角的半可分矩阵，$\circ$ 为逐元素乘。右端与注意力同构：注意力是 $Y = \mathrm{softmax}(QK^\top)V$，掩码是「全 1 下三角 + 行归一化」；SSM 是「数据相关的衰减掩码 + 无归一化」。两者都是「掩码矩阵乘」，区别只在掩码的结构——这句话就是「SSM 是结构化掩码矩阵乘」的准确含义。

这个同构带来两个工程后果。其一，L 有 rank-1 的递推结构，既可以用 $O(N)$ 扫描算，也可以用分块矩阵乘算；Mamba-2 的 SSD 算法把序列切成 chunk，chunk 内做 matmul、chunk 间做递推，从而吃到 Tensor Core，核心层比 Mamba-1 快 2–8 倍【论文】[20]。其二，既然 SSM 是「带衰减掩码的注意力」，把它与真正的注意力混在一起就很自然：注意力负责精确召回，SSM 负责廉价的长程聚合。

混合架构的代表是 Jamba【论文】[21]：4 个 Jamba block，每块 8 层，attention:Mamba = 1 : 7，每 2 层插一个 MoE，52B 总参数 / 12B 激活，256K 上下文，以 4-bit 权重量化及指定运行条件才可能放入单张 80GB 卡；BF16 的 52B 权重本身约 104GB。它的消融给出两个有用的结论：1 : 3 与 1 : 7 的 loss 差别很小，而 1 : 7 更省算力；纯 Mamba 在 few-shot 任务上会跑偏输出格式，混合模型即使只有 1/8 的层是注意力也能正常做 in-context learning，作者推测这与 induction head 需要注意力有关【论文】。这一消融只描述 Jamba 的模型与任务，不构成所有模型“1/8 注意力足够”的定律。

(1) **关联召回**。给定 m 个 key-value 对，要求按 key 取回 value（MQAR 类任务）。固定状态 S_t 的容量是 $d_\phi d_v$，超过容量后不同 key 的写入必然互相覆盖。Zoology 用受控合成任务量化了这一点：在同等困惑度下，注意力在召回上系统性优于 SSM，且 SSM 的召回率随状态增大而饱和【论文】[22]。

(2) **逐字复制与长文档抽取**。needle-in-haystack、引用原文、按行号改代码，都需要访问具体 token 而不是摘要。

(3) **分布外长度与短序列**。SSM 的状态是压缩摘要，序列越长早期信息衰减越彻底；而 $O(N)$ 递推在 GPU 上是一条低算术强度的串行链，prefill 并行度不如 matmul，具体交叉长度依赖扫描实现、硬件与形状，必须与优化后的 FlashAttention 实测比较。这是「贵」与「忘」的交换，不是「贵」与「便宜」的交换。

反过来说，线性注意力赢的场合也很清楚：以语义聚合为主、召回需求弱、序列极长（音频、基因、传感时序），且对显存敏感。判断标准不是「序列多长」，而是「答案是否依赖于某个具体位置的字面内容」。

3.8 边界条件：一张表与三条判断

表 3.2 机制的定性对照，比较对象是相同任务中的显式 MHA 与密集注意力。“强/中/弱”仅表示机制改变的方向与潜在程度，不对应固定倍数、质量等级或跨硬件排名；训练与生态字段截至 2026-10-22，实施前须核对具体模型与引擎版本。

方法	降 KV	降算力	降 IO	训练成本	长上下文	生态成熟度
MHA（基线）	无	无	无	低	弱	成熟
MQA	强	中	强	低	中	成熟
GQA	中	中	中	低	中	成熟
MLA	强	无	强	中	强	较成熟
FlashAttention-2	无	无	强	低	强	成熟
FlashAttention-3	无	无	强	低	强	较新
滑窗（SWA）	强	强	中	低	弱	成熟
attention sink	强	强	中	低	弱	较成熟
块稀疏（MoBA）	无	强	中	中	中	新
NSA	无	强	强	高	强	新
DSA	无	强	中	中	需评测	研究/特定实现
线性注意力	强	强	强	中	弱	中
Mamba-2	强	强	强	中	弱	中
混合（Jamba 式）	强	中	中	中	中	较新

读表 3.2 的方法，是依次问三个问题。

第一，省的是容量还是算力。KV 不够（并发上不去、上下文开不长）时，MLA、GQA、量化、分级存储是解；算力不够（长 prefill 慢、训练贵）时，FA、稀疏、线性是解。二者不能互相替代：DSA 省掉一半算力，但 128k 请求的 KV 仍是 8.6 GiB；MLA 把 KV 压到 68.6 KiB/token，也一点没减少 $O(N^2)$ 的 FLOPs。

第二，能不能重训。FA 保留同一注意力函数；滑窗、sink、稀疏选择会改变可见集，即使无需训练也必须验证任务质量。GQA、MLA 与 SSM 改变结构，需要相应 checkpoint 或训练。没有一组可跨模型通用的“免训练最多 2 倍、重训 5—10 倍”收益界限。

第三，召回需求有多强。固定状态、局部窗口与 top-k 选择都会损失某些可访问细节。在目标长度上测多针、组合推理、引用与复制，并与相同预算下的密集基线比较。Jamba 与 Zoology 的受控结果提供研究线索，不能用来断言所有线性模型必然不会输出格式。

练习：验证在线归一化

将任意一行分数分成不同大小的块，分别用整体 softmax 和式 (3.6)—(3.8) 计算输出。**提示：**使用极大正负分数检查稳定性，先用高精度参考，再检查 BF16/FP32 容差；改变分块后逐位不同本身并不说明算法近似。不能把 CPU 小例的时间当成 GPU kernel 吞吐。

未解问题

1. **稀疏的「选择」为什么够用，缺少理论。**DSA 只保留 2048 个 token 就能维持长上下文能力，但没有任何可预测的判据能回答「k 取多少才够」。缺的是被丢弃部分的注意力质量占比与下游准确率之间的定量关系；没有它，k 只能靠扫参，而长上下文扫参极其昂贵。
2. **MLA 不同形状的最优执行路径。**吸收前后的线性代数等价，但 prefill、decode 与训练在重用、反向传播和中间显存上的需求不同。如何在动态形状下选路径并控制浮点误差，需要同时验证数值一致性与系统性能。
3. **线性注意力的状态容量与任务难度之间没有定量关系。**MQAR 给的是合成任务上的下界，真实语料上「需要多少状态位才不丢信息」无法先验估计；混合比例（Jamba 的 1：7）只能靠消融确定，而消融成本随规模上升。

第 4 章　量化：容量、带宽与计算通路

本章结论

(1) 权重、激活与 KV 量化改变不同流量与计算路径。W4A16 主要压缩权重，W8A8 等配置只有在硬件与 kernel 支持下才能使用更高吞吐的低精度矩阵乘。

(2) 位宽不是存储成本的全部：缩放、零点、分组、padding 与解包都要计入。8 bit 也可能需要分块缩放；4 bit 格式的具体块长来自格式定义，而非所有算法的统一要求。

(3) 校准分布、目标长度和任务质量决定可接受精度。理论字节比不等于加速比，ppl 也不能替代代码、数学、检索和长尾任务的配对质量评测。

4.1　三条路径，三个不同的物理量

第 2 章把 decode 的成本写成每 token 摊薄字节数：权重项 W/B 加 KV 项 $kL\ell$。先忽略激活与元数据，可写出如下字节代理：

$$q_{\text{proxy}} \;=\; \frac{Nb_w}{B} \;+\; k_0 L\ell\, b_{\text{kv}}, \qquad k_0 = 2\, n_{\text{kv}} d_{\text{head}}, \tag{4.1}$$

其中 N 是参数量，b_w 是每个权重的字节数，k_0 是每 token 每层的 KV 元素数，b_{kv} 是 KV 每元素的字节数。

这个式子里没有 b_a（激活字节数）的位置，这一点值得停下来看。取一个 $d = d' = 8192$ 的 FFN 层，$B = 32$ 时激活读写为 $2 \times 32 \times 8192 \times 2 \approx 1.05$ MB，而权重是 $8192^2 \times 2 = 134$ MB；把激活从 FP16 压到 INT8 只省 0.5 MB，不到该层总流量的 0.4%【自算】。所以激活量化在 *decode* 阶段几乎不省带宽——它的潜在计算收益来自改变操作数格式与实际 kernel 通路，不是减少 FLOP 个数。反过来，权重量化在 decode 阶段几乎不省算力（FLOP 数一个没少），它省的是带宽与容量。把这两句话记住，本章后面全是它的推论。

表 4.1　三条量化路径各自降低的物理量与收益边界。第三、四列是单独使用时的局部模型，实际时间应重新比较计算、访存与固定开销，不能机械相乘。

路径	降低的物理量	decode 收益	prefill 收益	主要代价
权重及激活 W8A8	容量、带宽、计算通路	$\le 2\times$	$\le 2\times$（峰值翻倍）	缩放因子与校准
权重 FP16→INT4	容量、带宽	$\le 4\times$，随上下文衰减	依赖瓶颈与解包，可能为负	反量化开销、精度
激活（配合权重）FP8/INT8	计算通路、少量带宽	≈ 0（$< 1\%$ 流量）	$\le 2\times$	离群值，需平滑或旋转
KV FP16→FP8	容量、长上下文带宽	见式 (4.2)	≈ 0	attention 分数误差
KV FP16→INT4	容量、带宽	理想纯 W/KV 流量同降 4 倍	≈ 0	K 需 per-channel V 需 per-token

表 4.1 的比值均是匹配硬件通路的局部上限，不是可保证的服务级加速。先推导权重主导的 decode 流量比。先把 decode 的加速比写成闭式：设权重量化把 b_w 压到 b'_w、KV 精度不变，记 ρ_{KV} 为 KV 字节与权重字节之比，则

$$S_W(B,\ell) \;=\; \frac{1+\rho_{\text{KV}}}{b'_w/b_w + \rho_{\text{KV}}}, \qquad \rho_{\text{KV}} \;=\; \frac{B\, k_0 L\ell\, b_{\text{kv}}}{N b_w}. \tag{4.2}$$

$\rho_{\text{KV}} \to 0$ 时 $S_W \to b_w/b'_w$，这就是「理论上限等于位宽比」的严格表述；ρ_{KV} 变大时加速比被 KV 项稀释。对 W4（$b'_w/b_w = 1/4$）有 $S = 4(1+\rho_{\text{KV}})/(1+4\rho_{\text{KV}})$：$\rho_{\text{KV}} = 0.61$ 时只剩 1.87，$\rho_{\text{KV}} = 10$ 时只剩 1.07。第 2 章算过 Llama-3-70B 的 KV 是 320 KiB/token，$B = 32$、$\ell = 8192$ 时共 85.9 GB，正好是 FP16 权重 140 GB 的 0.61 倍【自算】——这就是长上下文服务里「4 bit 权重只换来不到 2 倍」的全部算术。

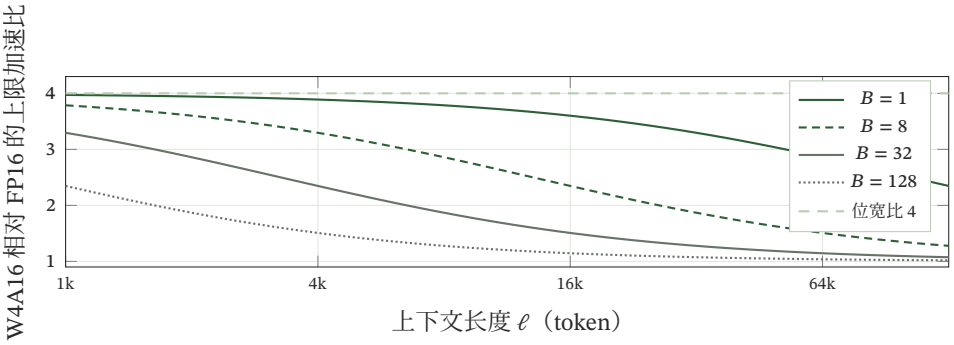


图 4.1 W4A16 的理论上限随上下文长度与并发数衰减，而不是一个常数。权重项被 KV 项摊薄的程度由 ρ_{KV} 决定：$B = 1$ 时 128k 上下文仍有 2.35 倍，$B = 32$ 时 8k 就掉到 1.87 倍、128k 只剩 1.07 倍。曲线全部按式 (4.2) 以 Llama-3-70B 的参数计算，不含任何实测点【自算】。

图 4.1 的要点是形状而不是数字：上限随 ρ_{KV} 单调下降，而 ρ_{KV} 与「并发 × 上下文长度」成正比。所以「4 bit 能快几倍」等价于问「你的平均请求有多长、并发有多高」。

▶ Llama-3-70B 的三档权重账本

取 $N = 70 \times 10^9$，8 张 H100 SXM 做 TP=8（聚合带宽 $8 \times 3.35 = 26.8$ TB/s）【官方文档】[3]，$\ell = 8192$，$B = 32$。KV 沿用第 2 章口径：$k_0 L = 2 \times 8 \times 128 \times 80 = 163{,}840$ 元素/token，FP16 下 320 KiB/token。

(1) 权重字节数 $W = Nb_w$：FP16 = $70 \times 10^9 \times 2 = 140$ GB；W8 = 70 GB；W4 = 35 GB。每卡分别是 17.5 / 8.75 / 4.375 GB。

(2) 每 token 摊薄读取量 按式 (4.1)，$B = 32$、$\ell = 8192$ 时 KV 项 $= 32 \times 327{,}680 \times 8192 = 85.9$ GB：

配置	每步字节 (GB)	每 token 字节 (GB)	每步时间 (ms)	理论上限
FP16	225.9	7.06	8.43	1.00×
W8A16	155.9	4.87	5.82	1.45×
W4A16	120.9	3.78	4.51	1.87×
W4A16+KV4	56.5	1.77	2.11	4.00×

对照 $\ell = 4096$、$B = 1$（KV 只占 1%）：W4A16 的上限是 3.89×。同一种量化，两个场景差一倍，差别不在算法而在 ρ_{KV}。

(3) 从上界到待测参数。令基线与量化配置的有效带宽分别为 $\eta_0 B_{\mathrm{mem}}$、$\eta_q B_{\mathrm{mem}}$，量化额外解包、通信或启动时间为 Δt。若二者都仍带宽受限，则

$$S_{\mathrm{cond}} \approx \frac{225.9/(26.8\eta_0)}{120.9/(26.8\eta_q) + \Delta t} \quad \text{（GB 与 TB/s 换算，时间单位 ms）.}$$

$\eta_0, \eta_q, \Delta t$ 须在同一硬件和形状上测量；本次未做 GPU 实测。QServe 报告的反量化分析适用于其所测基线与 kernel，其他 W4A16 实现需另行测量。

把每步的时间拆成权重、KV 与反量化三块会更清楚：$B = 32$、$\ell = 8192$ 时，权重块从 5.22 ms 压到 1.31 ms，但 KV 的 3.21 ms 一动不动，所以总时间只从 8.43 ms 走到 4.51 ms；再算上 0.39 ms 的反量化开销是 4.90 ms，相比理论值又少一截。未压缩的 KV 限制了纯流量收益，新增解包又增加了时间——这也是「先量化权重、再看 KV」的顺序在长上下文服务里经常出错的原因【自算，额外 0.39 ms 仅为条件假设，并非本次实测】。

现在看 prefill。n 个 token 同时进来时，权重只读一遍就被复用 n 次，算术强度变成

$$I_{\mathrm{prefill}} \approx \frac{2nN}{Nb_w} = \frac{2n}{b_w}, \qquad n^* = \frac{I^* b_w}{2}. \tag{4.3}$$

n^* 是权重读取耗时与计算耗时相等的 token 数。H100 SXM 的 $I^* \approx 295$ FLOP/byte【官方文档】[3]，于是 FP16 的 $n^* = 295$、W8 为 147、W4 为 74【自算】。实际 chunk 需由 SLO 选择；若取 512—8192 token，部分形状会越过该简化交叉点。以 $n = 2048$ 时算术强度约 2048 FLOP/byte，是平衡点的 7 倍，孤立权重读取时间为线性层峰值计算时间的 14.4%（不是端到端时间占比）【自算】。此时把权重再压 4 倍，减少的是

本来就被计算隐藏掉的那 14.4%，若计算与访存理想重叠且其他项不变，关键路径不变。这只是该模型的零收益结论：缓存驻留、激活、通信、kernel 选择和容量变化会改变实际结果。

再叠加反量化。W4A16 的矩阵乘仍要先把 4 bit 解包成 FP16 再送进 Tensor Core，多出来的解包指令与 smem 流量恰好落在同一段关键路径上，所以 prefill 常常是净亏损。QServe 对其所测 INT4 GEMM 基线的分析给出了量级：权重或部分和的反量化带来 20%–90% 的运行时开销【论文】[23]。这不是任意 INT4 kernel 的固定开销，因此必须实测总时间，而不能把字节比当性能承诺。

那 prefill 上怎样才算「省算力」？只能让 F/P 变小，即提高 P。FP8 的 Tensor Core 峰值是 FP16 的两倍（H100 SXM：1979 对 989 TFLOPS）【官方文档】[3]，这是支持低精度计算的一条路径；INT8、FP4 等也可能改变通路，需匹配具体硬件与 kernel。DeepSeek-V3 在 671B MoE 上用 FP8 混合精度完成训练，说明这套数值格式在大规模上已经被验证【论文】[8]。但要拿到这两倍，激活也必须是 *FP8*；只量化权重的 W8A16 和 W4A16 一样，仍然只是在搬字节。工程上还需核对两个操作数的格式、累加精度、缩放布局与实际派发的 kernel。

4.2 数值格式：位宽之外的三个自由度

一个数值格式由三件事决定：总位宽、位宽在指数与尾数之间的切分、以及缩放因子的粒度。前两者决定「能表示多大的数」和「能分辨多小的差」，第三者决定这两件事在张量内部是否均匀。

FP8 的两个变体是这件事最干净的教材。E4M3 用 4 位指数换 3 位尾数，最大可表示 448，最小正规数 2^{-6}；它不表示无穷大，只留一个尾数编码给 NaN，把省下的编码让给动态范围【论文】[24]。E5M2 反过来，5 位指数保住 2^{-14} 到 57344 的范围，但尾数只剩 2 位，相对分辨率约 12.5%（E4M3 是 6.25%）。于是分工很清楚：

- **E4M3 保精度**，用于权重、激活与 KV cache：它们的动态范围相对可控，而精度直接决定输出，值得用 3 位尾数去换。
- **E5M2 保范围**，用于梯度：梯度跨多个数量级，单个梯度的相对误差又会被大量样本平均掉，范围比分辨率值钱。

这也解释了 vLLM 的 KV cache 数据类型里为什么同时有 `fp8_e4m3` 与 `fp8_e5m2`，而默认的 `fp8` 指向 E4M3：KV 要被读回来做点积，精度优先（源码 `vllm/config/cache.py:39` 的 `CacheDType` 列出了全部可选值）【官方文档】。

4 bit 是另一回事。E2M1 只能表示 $\{0, \pm0.5, \pm1, \pm1.5, \pm2, \pm3, \pm4, \pm6\}$ 这 15 个值（含 0），最大 6。用 16 级去覆盖动态范围跨几个数量级的张量，per-tensor 缩放必然失败：

$$\mathrm{SQNR}_{\mathrm{sine}} \approx 6.02\,b + 1.76\ \mathrm{dB}, \qquad \frac{\Delta x}{x} \propto \frac{R}{2^b\,\bar{x}}. \tag{4.4}$$

左式是理想满量程正弦波均匀量化的工程近似，不适用于任意 LLM 张量；右式仅作量级直觉。式 (4.4) 右侧说明相对误差正比于「量程 ÷ 量化级数 ÷ 典型值」。若一个张量的最大值是典型值的 100 倍—— LLM 的激活里这很常见，massive activations 能到 10^5 倍【论文】[25] ——按均匀量化的直觉，典型值的可分辨级数会严重减少；不能把“负有效位宽”当浮点/分块格式的精确误差模型，也不能据此保证 8 bit 可用。这正是 LLM.int8() 必须把离群通道单独拎出来做 FP16 混合分解的原因【论文】[26]。

分块缩放（block scaling）就是把这个「扣除」关进块内：它限制一个离群值影响的元素范围；块内极值比例仍取决于张量分布，不能统一假设为 2—4 倍。表 4.2 把本书讨论的格式按位宽、缩放粒度与硬件通路并列；两种工业 4 bit 格式的差别在块的大小与缩放因子的编码方式：

- **MXFP4**：32 个 E2M1 共享一个 E8M0 缩放因子，而 E8M0 只能表示 2 的整数次幂，平均 $4+8/32 = 4.25$ bit/元素【论文】[27]。好处是解码只需移位，硬件极便宜；代价是幂次缩放限制了局部可表示范围；误差取决于舍入与饱和策略。
- **NVFP4**：16 个 E2M1 共享一个 E4M3 缩放因子，另加一个 per-tensor 的 FP32 二级缩放，平均 $4+8/16 = 4.5$ bit/元素【官方文档】[28]。块小一半、缩放因子带 3 位尾数，代价是元数据从 6.25% 涨到 12.5%。

两级缩放的理由值得说清楚。E4M3 的动态范围只有约 $2^{14.8}$，单靠它覆盖不了整个张量；先用 FP32 把张量整体拉进 E4M3 的甜区，再让每个 16 元素块在甜区里选自己的缩放，两级各管一段。这与浮点本身「指数管范围、尾数管精度」是同一思路，只是把指数的粒度下放到了块。

表 4.2 常用数值格式的对照。有效位宽列出的是把缩放元数据摊到每个元素之后的实际成本；「最大可表示」一列对 MXFP4/NVFP4 指单个元素的绝对值上限。

格式	位宽	缩放粒度	缩放编码	有效位	最大可表示	典型用途
FP16	16	无	—	16	65504	基线
BF16	16	无	—	16	3.4×10^{38}	训练基线
FP8 E4M3	8	per-tensor / 128 方块	FP32	8	448	权重、激活、KV
FP8 E5M2	8	per-tensor	FP32	8	57344	梯度
INT8	8	权重按通道、激活按 token	FP32	8	127	W8A8
MXFP4	4	每 32 元素	E8M0	4.25	6	权重（Blackwell/OCP）
NVFP4	4	每 16 元素 + per-tensor	E4M3 + FP32	4.5	6	权重与激活（Blackwell）

8 bit 同样可以采用 per-token、per-channel 或块缩放，不能仅凭位宽断言无需分块。若一个 $m\times n$ 矩阵使用每行一个 FP32 scale，摊销为 $32/n$ bit/元素：$n=8192$ 时约 0.0039 bit，$n=128$ 时 0.25 bit【自算】。GPTQ/AWQ 的 group size 与 FP8 块缩放都是精度、元数据和 kernel 布局的折中，128 不是所有模型或实现的固定最优值。

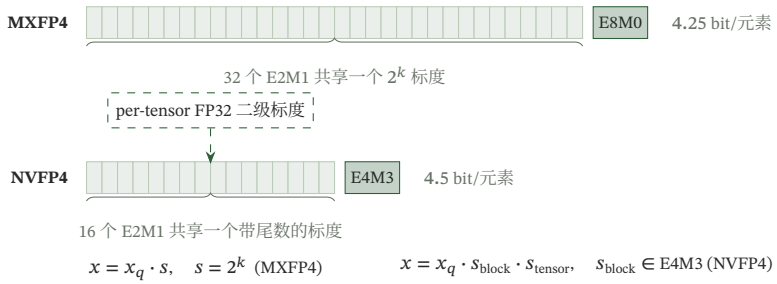


图 4.2 分块缩放把动态范围局部化，两种格式的差别在块大小与标度编码。MXFP4 用 32 元素 + 2 的幂标度，元数据最省、硬件最便宜，但限制了标度的表达能力；NVFP4 把块缩到 16 元素、标度换成带 3 位尾数的 E4M3，再加 per-tensor 的 FP32 二级标度，用 12.5% 的元数据换来更细的拟合。

图 4.2 说明为什么缩放因子的粒度是一个工程折中：粒度越细，离群值的影响越局部、精度越好，但元数据占比越高；粒度越粗，元数据越省，一个离群值却会把整块的可表示范围撑大。MXFP4 取 32 元素一块、NVFP4 取 16 元素一块，都是在这条曲线上选点。

4.3 算法族：四种假设，四种失效

后训练量化（PTQ）没有免费的午餐：在没有梯度的条件下，要决定「哪个权重可以粗一点」，只能靠某个先验。四种主流算法的区别，就是它们各自选了哪条先验，以及那条先验在什么条件下不成立。

GPTQ：逐层二阶误差补偿。它把量化写成逐层的最小二乘问题：

$$\min_{\hat{\mathbf{W}}} \left\| \mathbf{W}\mathbf{X} - \hat{\mathbf{W}}\mathbf{X} \right\|_F^2, \tag{4.5}$$

其中 $\mathbf{X}$ 是该层在校准集上的输入。按列顺序量化，每量化完一列，就用 $\mathbf{H}=2\mathbf{X}\mathbf{X}^\top$ 的逆把这一列产生的误差按比例分摊到尚未量化的列上，使层输出的平方误差最小，而不是让每个权重各自最接近原值。原论文用近似二阶信息把 175B 模型压到 3 – 4 bit，耗时约 4 GPU 小时，并展示低比特推理的部署收益【论文】[29]。

它的假设有三条：层间可解耦（逐层最优近似全局最优）；$\mathbf{H}$ 能被 128 – 1024 条校准样本估准且良态；误差是二次的、可局部补偿。失效正对应这三条：校准集偏移时 $\mathbf{H}$ 估错方向；模型大或上下文长时 $\mathbf{H}$ 病态，逆矩阵放大噪声；MoE 里每个专家分到的校准 token 只有几分之一，$\mathbf{H}$ 的方差大到补偿有害；压到 3 bit 以下时 group 必须缩到 32 以内，元数据开始吃掉压缩收益。

AWQ：保护 1% 的显著通道。它换了一条更便宜的先验：不看 **H**，只看激活的幅度。观察是「只保护 1% 的显著权重，量化误差就大幅下降」，而判断显著要看激活分布而不是权重本身；保护方式不是混合精度（硬件不友好），而是对显著通道做等价放大 s_j，把它们的量化误差压下去，再把缩放折进前一个算子。由于不依赖反向传播或重建，它对校准集的过拟合更轻，在指令微调与多模态模型上迁移更好【论文】[30]。

假设是：显著性可以由离线的激活统计静态确定，且缩放可以数学等价地折叠。在 MoE 上必须按专家检查——路由让每个 token 走不同专家，通道的显著性随专家而变，一份全局显著集可能错配；在长上下文里第二个碎—— massive activations 只出现在极少数 token 上，按 token 平均之后被稀释成噪声；在小 batch 或域漂移时第三个碎——校准时的激活分布与线上不同，s_j 的方向就错了。

SmoothQuant：把难度从激活搬到权重。W8A8 的难点从来不是权重，是激活里的离群通道。SmoothQuant 做一个等价变换 $\mathbf{XW} = (\mathbf{X}/s)(s\mathbf{W})$，取

$$s_j = \frac{\max|\mathbf{X}_j|^{\alpha}}{\max|\mathbf{W}_j|^{1-\alpha}}, \qquad \alpha \approx 0.5, \tag{4.6}$$

把激活的离群按通道压回去、同时把权重的量化难度抬高一点，因为权重比激活更耐压。在 OPT、BLOOM、Llama-1/2、Falcon、Mistral、Mixtral 上做到 W8A8 且精度几乎无损，报告的最高加速 1.56 倍、显存减半，并让 530B 模型在单节点内可服务【论文】[31]。

它的假设非常具体：离群是按通道的，且沿 token 维稳定，所以一个常数 s_j 能对齐整个序列。这条假设在长上下文下松动：massive activations 是逐 token 出现的，一个 per-channel 常数无法同时照顾「有离群的那些 token」和「没有的」。在 MoE 上，不同专家的输入分布不同，如果强行共享一套 s_j 可能失配；可按专家校准，代价是样本覆盖与管理成本。更根本的限制是它只解决 8 bit：α 的迁移把动态范围重新分配，但没有把 4 bit 的 16 级变成够用的分辨率，所以 4 bit 需要的是旋转（把离群摊平到所有通道）或分块，而不是平滑。

QServe：把算法和系统一起设计。它在所测低比特 GEMM 中发现，CUDA core 上的反量化和部分和处理可能成为瓶颈，相关 INT4 基线的运行时开销达 20%–90%【论文】[23]。其 QoQ 算法采用 W4A8KV4：权重和 KV 各 4 bit，线性层激活 8 bit，并使用 INT8 矩阵乘通路。

progressive group quantization 是两阶段量化：先用逐通道 FP16 scale 将权重量化到保护范围 $[-119, 119]$ 内的 INT8，再把这些中间整数逐组量化为非对称 INT4。执行 GEMM 时先由 INT4 恢复到 INT8 中间值，再用 INT8 Tensor Core 计算；保护范围使逐组反量化后的中间值仍落在 INT8 可表示范围，避免溢出。SmoothAttention 则用等价缩放把 K 的逐通道离群迁移到未量化的 Q，减轻 KV4 的误差；寄存器级并行与计算感知的权重重排进一步降低反量化、地址计算和访存开销。这些设计共同作用，不能把系统收益全部归给位宽。

论文 v3 的效率实验使用 A100-80GB-SXM4 与 L40S-48GB，输入 1024、输出 512 token，比较同一显存约束下可达到的最大生成吞吐。相对 TensorRT-LLM 的 FP16、W8A8、W4A16 三种配置中各模型表现最好的配置，Llama-3-8B 的倍率约为 1.20（A100）与 1.39（L40S），Qwen1.5-72B 约为 2.38 与 3.47（原文表 4）【论文】[23]。A100 使用逐通道量化，L40S 使用逐组量化；倍率包含容量允许的 batch、kernel 和硬件差异，并未约束到本书贯穿算例的 TTFT、最大 ITL 与 E2E 合同。

• 净收益必须按时间相除

令 t_0 为基线测量，量化后计算、访存、解包与其他开销分别为 t_c, t_m, t_d, t_f。如果计算与访存可重叠、解包是否能重叠另有证据，可使用 $S \approx t_0/[\max(t_c, t_m) + t_d + t_f]$ 作为条件模型。不能写成“min{ 带宽比、算力比、$1/(1 +$ 开销$)$}”：该表达式会让任何正开销都把加速封到 1 以下，且忽略基线瓶颈。容量带来的并发增益要在相同 SLO 与质量条件下另测。

把四种算法的失效条件并排看，会发现它们指向同一组外部变量：

- **MoE**：专家级分布异质，让「一个校准集、一套 scale」失配；每专家样本变少，让 **H** 的估计方差变

大；路由让激活统计逐 token 变化，让 AWQ 的静态显著集失效。应记录每专家校准样本量和路由覆盖，避免把低覆盖误作算法必然失效。

- **长上下文**：massive activations 只出现在少数 token 上且幅度可达 10^5 倍【论文】[25]，per-channel 平均会把它抹平；校准通常在 2k – 4k 上做，32k 时激活分布已整体平移。
- **小 batch**：激活分布变窄、与校准集偏差变大；反量化的固定开销摊不薄，W4A8 的算力优势也用不上，因为算力不是瓶颈。

! 常见误解

「W4A16 比 FP16 快 4 倍」在 decode、batch=1、短上下文、权重项主导、内核效率打满、无反量化开销这六个条件同时成立时才接近真的。少一个条件就少一截，少三个条件可能一点不剩。看到任何量化加速比，先问三件事：什么 batch、什么上下文长度、基线是什么精度。

4.4 评测：perplexity 不是验收标准

量化论文习惯报 perplexity（ppl），工程团队也习惯用「ppl 只掉了 0.1」来签字。这个签字方式有三个独立的问题。

第一，ppl 是平均量，任务是阈值量。$\text{ppl} = \exp(\overline{\text{NLL}})$，在百万级 token 上平均，少数难 token 的退化会被摊平。ppl 从 6.5 涨到 6.6，对应每 token 平均负对数似然增加 $\ln(6.6/6.5) = 0.0153$ nats【自算】；相对 NLL 变化还需除以原 NLL，不能把 0.0153 直接叫 1.5%。但下游任务是序列级的：一段 500 token 的回答，只要有一个关键 token 被扰动，整段就可能判错，可用一个独立扰动教学模型说明序列敏感性（它不是从 ppl 推出来的概率）：

$$P(\text{整段不被扰动}) \approx (1-\epsilon)^{\ell}. \tag{4.7}$$

取 $\epsilon = 0.1\%$、$\ell = 500$，得 0.606 ——四成的样本至少含一个被扰动的 token，【自算，假设各位置独立且扰动率 0.1%】。这个例子不等于四成任务错误，也不意味着 ppl 一定变化 0.1。ppl 看不见的那部分，恰好是任务看得见的那部分。

第二，抽样的统计噪声大于常见掉点。分类任务的准确率标准误为

$$\text{SE} = \sqrt{\frac{p(1-p)}{n}}. \tag{4.8}$$

$p = 0.7$、$n = 1000$ 时 SE = 1.45%；即使用满 MMLU 的 14042 题，SE = 0.39%【自算】。这些是单个准确率在独立同分布近似下的标准误，不能直接充当两个配置差值的标准误。比较相同题目上的量化与基线，应保留逐题配对结果，使用配对 bootstrap 或二分类的 McNemar 检验，报告差值置信区间。若要声称可接受退化，先给容忍界限与检验功效；未显著不等于无损。

第三，量化误差是系统性的，不是随机的。它偏向特定通道、特定位置、特定 token 类型，因此会改变 greedy 解码的轨迹：误差改变 argmax，输出从这一步起换一条路径，后面的 token 与基线再无对应关系。这解释了一个常见现象—— ppl 几乎不动，但代码生成的 pass@1 掉得很整齐，因为代码对标识符和边界条件零容错。QuaRot 的报告方式值得学：既给 ppl（Llama2-70B 在 WikiText-2 上损失 ≤ 0.47），也给 zero-shot 保留率（99%）【论文】[32]。两个数一起看，才知道掉的是哪一类能力。

那应该怎么评？按「能力类别 × 上下文长度」铺一张矩阵，每一格都有明确的判据：

- **通用知识**（MMLU、ARC-C）：看准确率与置信区间，样本量不少于 2000 题；这一项对 8 bit 几乎不敏感，对 4 bit 敏感。
- **代码**（HumanEval、MBPP）：固定解码、种子与样本数并看 pass@1；greedy 和抽样评测分开；代码对标识符、缩进、边界条件零容错，是 4 bit 最早暴露问题的场景。
- **数学推理**（GSM8K、MATH）：最终答案与中间步骤都要看，量化误差常常只破坏其中一边。
- **多语言**（MGSM 类）：低资源语言的 token 词频低，量化误差的相对影响更大；只测英文会系统性高估。

- **长上下文**（RULER [33]）：按目标长度逐档测，NIAH 这类单针检索只测到表层能力【论文】。
- **长尾**：专有名词、罕见实体、数字与单位；这些 token 的嵌入最容易在量化后失真，而它们恰是 RAG 与 agent 场景的高频输入。

长上下文下的误差累积方式，与短上下文不同，值得单独说。它不是「每多一个 token 就多积一点误差」的线性过程，而是「竞争键变多，检索从确定变成概率」的相变过程。KV 量化给每个 key 的注意力分数带来一个固定量级的扰动；上下文越长，与正确 key 分数接近的干扰 key 越多，正确 key 被挤到第二名的概率就越大。因此某些检索任务可能出现非线性退化：在某个长度之前几乎测不出，越过它之后迅速变差。这也意味着 KV 量化的评测必须在目标长度上做，用 4k 的成绩替 128k 签字是不成立的。工程上的缓解手段是分层：K 按 channel 量化、V 按 token 量化（KIVI 的做法 [34]），或者只对远离当前位置的 KV 降精度，近端保持 FP16。

还有一条容易被忽略：校准集不能泄漏评测答案或直接复用测试样本；可与生产目标域相同，但需独立划分。AWQ 强调「不依赖反向传播，因此不会过拟合校准集」，正是因为重建类方法在 calibration 与 test 同分布时数字偏乐观【论文】[30]。

4.5 决策：先选候选，再验证约束

硬件支持、ρ_{KV} 与任务容忍界限决定候选空间，而不是直接决定生产推荐。下表给机制级候选；具体命令、方法名与 KV 类型以第 8 章固定版本和对应官方文档为准，不把移动的源码行号或未核验字符串当可直接运行的配置。

表 4.3 量化候选与必须补齐的证据。所有性能比较以相同模型、质量、工作负载和 SLO 为条件。

主要约束	候选方向	验证要求
低并发、权重主导 decode	W4A16 或 W8A16	解包 kernel、元数据与实测 ITL；质量不合格退回 BF16
算力主导 prefill	硬件支持的 W8A8/FP8	两操作数与累加格式、矩阵形状、TTFT 和输入吞吐
长上下文、KV 主导	FP8 或更低比特 KV	目标长度的检索/代码质量、实际 attention kernel 与 scale
Blackwell 低精度路径	NVFP4 等支持格式	匹配格式/布局、峰值稀疏口径、质量与实际 goodput
MoE 与 EP	专家/共享/门控分别选格式	每专家校准覆盖、通信量与负载均衡、不同 batch 形状
精度敏感或长尾任务	BF16 或先测 8 bit	配对质量区间、异常样本与长输出；不预设 8 bit 无损

主例在 $B = 32, \ell = 4096$ 时，KV 总流量约 42.95 GB，故 W4 纯流量比为 $(140+42.95)/(35+42.95) \approx 2.35$【自算】。把 ℓ 改为输出末尾 4608 再算，随后测相同到达率下 TTFT、最大 ITL、E2E 和质量。若只是容量足以增加 batch，应报告增大 batch 之后的新延迟，不能把容量收益和原 batch 加速相乘。

练习：格式元数据能否忽略

比较一个 8192×8192 权重矩阵的 BF16、理想 INT4、group-128 且每组 FP16 scale 加 FP16 zero-point 的大小。**提示：**后一种为 $0.5 + 4/128 = 0.53125$ byte/元素，实际压缩比约 3.76 而非 4；再检查行 padding、打包与 kernel 工作区是否占据剩余容量。

未解问题

1. **从格式直接预测任务质量仍有困难**。ppl 变化不足以确定任务的 pass@1；少数离群通道与困难样本可能主导退化。应在独立数据集上复核质量及统计区间，并检验结论能否跨模型、长度与任务迁移。
2. **MoE 的逐专家校准需覆盖路由偏差**。每个专家看到的 token 有限，GPTQ 的 $\mathbf{H}$ 估计和 AWQ 的显著通道可能随路由改变。EP 部署应记录各专家校准覆盖率，核对离线校准与线上分布的一致性。
3. **量化、投机与前缀缓存需联合评测**。量化可能改变 draft/target 分布与接受率，共享 KV 的误差也可能相关。应按目标长度及负载测量质量、接受率与 SLO，不能相乘单项收益或假定误差可平均消除。

第 5 章　稀疏性与 MoE

本章结论

(1) 稀疏能省多少，不由「剪掉多少权重」决定，而由元数据开销与硬件路径决定。2:4 结构化稀疏用 1.78 倍的存储压缩换来 2 倍的 Tensor Core 峰值，端到端由可加速时间占比决定，不能把某个 1.3—1.5 倍场景当普遍范围；非结构化 90% 稀疏的 4 字节列索引把 10 倍的名义收益压到 3.3 倍【自算】，还需计实际索引访问、激活复用与 kernel 效率。

(2) MoE 是目前部署规模最大的稀疏形式，但它省的不是显存而是算力：DeepSeek-V3 的 671B 参数在 BF16 下占 1.34 TB 显存【自算】，每个 token 只激活 37B 参数【论文】。这使 MoE 的第一约束从「算力够不够」变成「显存放不放得下、互联撑不撑得住」，与量化、蒸馏等减显存手段的方向正好相反。

(3) MoE 的瓶颈随 batch、路由与拓扑变化。all-to-all、专家读取、计算与负载不均都可能限制一步；平均字节模型只能筛选配置，不能证明整机“主瓶颈”或 p99。

5.1　三种粒度，三本账

「稀疏」在推理系统里至少指三件不同的事，它们的账本完全不同。

权重稀疏的粒度是单个权重：把 W 中一部分元素置零，用压缩格式存、专门 kernel 算。它的元数据开销与权重个数成正比，剪得不够狠时索引比权重还贵。

激活稀疏（MoE）的粒度是 token 到专家的选择：每个 token 只经过 K 个专家，其余专家的权重根本不参与这次前向。它的元数据只是每个 token 的 K 个专家编号，与权重个数无关。

注意力稀疏的粒度是 KV 条目：每个 query 只与 k 个历史位置做注意力，元数据是每个 query 的 k 个位置，开销与 *query* 数成正比。

稀疏的粒度越细，能省的计算越多，索引也就越贵；工程上真正走通的只有两条路：细粒度交给硬件解压，粗粒度交给路由。表 5.1 是五本账。

表 5.1　五种形式的账本对照（第一行是稠密基准）。关键差别在第三列：元数据开销跟随什么量增长。2:4 与 MoE 的元数据分别按元素、按 token 摊薄，而权重剪枝的索引按非零元摊薄——最后一个在保留率降低时仍保持每个非零元相同的索引/值比例。

形式	粒度	元数据开销	权重显存	单 token 计算	主要约束
稠密（N 参数）	—	0	$2N$ B	$2N$ FLOP	显存带宽
2:4 结构化	4 元素 1 组	0.125 B/元素，硬件解压	1.78× 压缩	0.5×	可剪层比例、N 大小
非结构化 90%	单权重	0.4 B/元素（int32 列索引）	3.3× 名义	0.1×	事务粒度、CUDA core 峰值
MoE（256 选 8）	token	8 个专家号/token	不省，全部驻留	37/671 = 5.5%	显存容量、互联带宽
稀疏注意力（top-2048）	KV 条目	4 B/位置	KV 不省，索引 +11%	$O(L) \to O(k)$	索引器自身计算、缓存一致性

5.2 2:4：硬件怎么在一个 MMA 里跳过零

Ampere 起的 Sparse Tensor Core 采用一种极窄的结构化约束：沿 GEMM 的 K 方向，每 4 个连续元素为一组，只允许保留 2 个非零。这条约束把「哪些位置非零」压缩成每组 4 bit 的元数据—— $C(4,2) = 6$ 种组合，4 bit 够用。

硬件于是可以做一件稠密 MMA 做不到的事：指令形状基本不变，逻辑 K 翻倍。稠密的 `m16n8k16` 对应的稀疏版本是 `m16n8k32`，但物理上只读一半的操作数。数据通路里，每个 K 槽位前有一个由元数据控制的 4 选 2 多路选择器，只把 2 个有效值送进乘法器阵列，累加器照旧按 4 个槽位累加。乘法器数量没变，每个周期处理的逻辑元素翻倍，峰值吞吐随之翻倍【论文】[35]。图 5.1 画出了左侧的稀疏模式与右侧的数据通路。

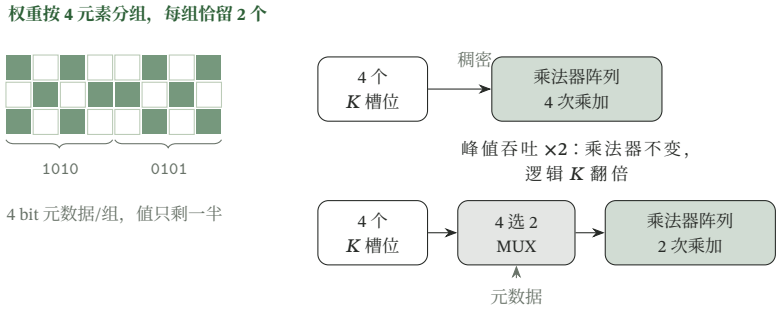


图 5.1 2:4 约束如何变成硬件上的 2 倍吞吐。左边是 2:4 分组与每组 4 bit 元数据；右上是稠密路径（4 个槽位做 4 次乘加），右下是稀疏路径（元数据驱动 4 选 2，同一组乘法器只做 2 次乘加却覆盖 2 倍逻辑 K）。加速只发生在一个操作数上，另一个操作数与输出仍然稠密。

三条硬约束决定了它的适用范围。第一，所讨论的稀疏 MMA 格式只支持一个操作数的指定稀疏布局，另一个操作数与输出仍稠密；但零乘法被跳过的是整个 GEMM 的相应乘积，不能理解成只加速一半 FLOPs。第二，维度必须是 16（16 bit）或 32（8 bit）的倍数，不满足的层只能保持稠密【论文】。第三，压缩布局固定（cuSPARSELt 的 compressed 格式），权重必须在离线阶段按该布局重排。

▶ 2:4 到底省了多少字节

设一组 4 个元素。值：保留 2 个，FP16 每元素 2 B，共 4 B。元数据：每组 4 bit = 0.5 B。合计 4.5 B / 4 元素 = 1.125 B/元素，而稠密是 2 B/元素：

$$\text{压缩比} = \frac{2}{1.125} \approx 1.78.$$

所以 2:4 在存储上是 *1.78* 倍而非 2 倍；2 倍只出现在乘法器吞吐上。作为参照，H100 SXM 的 FP16 稠密算力 989 TFLOPS、稀疏算力 1979 TFLOPS【官方文档】[3]，A100 是 312 与 624 TFLOPS【官方文档】[36]。这四个数都是峰值，且都假设 N 足够大、元数据开销被完全摊薄。

为何局部 2 倍不等于端到端 2 倍

把这个差距全部归因于「实现不好」是懒惰的。它由四个可分离的原因叠加而成，前两个是结构性的，换更好的 kernel 也消不掉。

（一）**N 太小**。语言模型的 GEMM 里 N 通常是 batch 乘序列长。Mishra 等人给出的条件是：序列长 256 时，batch 要到 40，N 才够 10K，稀疏 GEMM 才接近 2 倍【论文】。decode 的 N 是 1 或几十，稀疏 MMA 的 tile 根本填不满，解压指令与元数据读取的固定开销反而占了上风。这与第 2 章是同一件事：小 N 下，一切「提高峰值算力」的手段都被带宽和固定开销吃掉。

（二）**层间不平衡**。能剪的只有 FFN 和部分投影；attention 的 $QK^\top$ 与 PV 要么 K 维不满足 16 的倍数，要么对数值误差敏感。若可加速部分占基线实际时间的比例为 f（不是自动等于 FLOPs 占比），局部 2 倍加速经 Amdahl 折算后端到端只有

$$S_{\text{e2e}} = \frac{1}{(1-f) + f/2}. \tag{5.1}$$

$f = 0.6$ 时 $S_{\text{e2e}} = 1.43$，$f = 0.8$ 时也只有 1.67【自算】。这只是指定 f 与局部速度的条件算例；实际也可能低于 1 或接近 2。真正的 f、局部形状与剪枝后质量都必须实测。

（三）**精度**。2:4 不是「剪到 50%」，而是「每组 4 个里必须丢 2 个」。当某一组里 3 个权重都重要时，这个约束无解，只能牺牲一个。所以 2:4 的精度损失系统性地高于同等稀疏率的非结构化剪枝。

（四）内存侧只省 1.78 倍。decode 是带宽受限的（第 2 章），而 2:4 只把权重字节压到 56%，带宽受限阶段的时间下界也只降到 56%。加速比的天花板因此是 min(2, 1.78)，再乘上 f 与 N 的折损。

纯幅值剪枝的限制

上一代剪枝的标准流程来自 Han 等人：按权重绝对值剪掉小的，再重训恢复【论文】[37]。纯幅值准则在许多 LLM 高稀疏设置上难以保持质量，原因有三层，一层比一层根本。

重训成本不可支付。剪枝后精度必然下降，恢复它需要一次相当于预训练若干比例的训练；当预训练本身是数千卡月时，这个「若干比例」已大过多数团队的全年预算。「剪枝 + 重训」在 LLM 上因此不是技术问题而是经济学问题。

「**小权重**」**在 LLM 里不等于**「**不重要**」。LLM 的权重分布是重尾的，且存在少数幅值极大的离群特征；残差流的尺度由这些特征决定，剪掉它们会改变整层的数值尺度，误差沿几十层累积。一个直接的证据是 Wanda：它把比较对象从 $|w|$ 换成 $|w| \cdot \|x\|$（权重幅值乘以对应输入激活的幅值），在相同稀疏率下显著优于纯幅值准则【论文】[38]。如果「只看权重」够用，就不需要把激活引进来。

没有重建，只有选择。magnitude pruning 本质是「挑一批权重丢掉」，不做任何补偿。SparseGPT 把剪枝写成逐层的最小二乘重建（用 Hessian 的逆近似修正剩余权重），才做到对 OPT-175B 与 BLOOM-176B 一次性剪到 50% 而不重训、60% 稀疏下困惑度仍只有很小上升，且 175B 模型的整个剪枝过程在 4.5 小时内完成【论文】[39]。两条对照说明：能用的剪枝都是「稀疏感知 + 逐层重建」，纯幅值方法缺少对输入分布与误差补偿的建模；差距要按具体模型、稀疏率和质量指标报告。

5.3 非结构化剪枝：索引与计算路径必须同时记账

结构化稀疏能用，是因为它把「稀疏」翻译成了硬件认识的固定模式；非结构化剪枝保留了更高的稀疏率，却把索引变成数据相关的东西。

先把字节账算清楚。设权重 $W \in \mathbb{R}^{d \times d'}$，保留比例 s，值用 FP16（2 B），列索引用 int32（4 B）。稠密需要 $2dd'$ 字节；稀疏需要

$$Q_{\text{sparse}} = s\,dd'\,(b_v + b_i) = 6s\,dd' \quad \text{(byte)}, \tag{5.2}$$

名义压缩比是 $2/(6s) = 1/(3s)$。$s = 0.1$ 时是 3.3 倍而不是 10 倍；$s = 1/3$ 时两者打平；$s > 1/3$ 时稀疏反而更占显存【自算】。即使把索引压到 1 B（块内 8 bit 偏移），也只有 $1/(1.5s) = 6.7$ 倍——「剪掉 90% 就该省 90% 带宽」从第一个字节起就是错的。

存储账不能直接变成运行时账。CSR 的值与列索引可以连续读取；随机的是激活 gather，且相邻行可能在 L1/L2 复用同一激活。不能把一个 gather 的 32 B sector 浪费系数乘到全部压缩权重流量上。一个较诚实的条件模型是

$$t_{\text{sparse}} \gtrsim \max\left(\frac{2Bsdd'}{P_{\text{sparse}}}, \frac{sdd'(b_v + b_i) + Q_{\text{act,eff}}}{B_{\text{mem}}}\right) + t_{\text{idx}}, \tag{5.3}$$

其中 $Q_{\text{act,eff}}$ 由 gather、缓存和事务粒度决定，t_{idx} 为索引/调度开销。不同 SpMM、块稀疏与半结构化 kernel 可能使用不同计算通路，不能宣称所有非结构化计算永远只能走同一个 60 TFLOPS 路径。应测实际 HBM 字节和 kernel 时间，再与优化后的稠密基线比较。

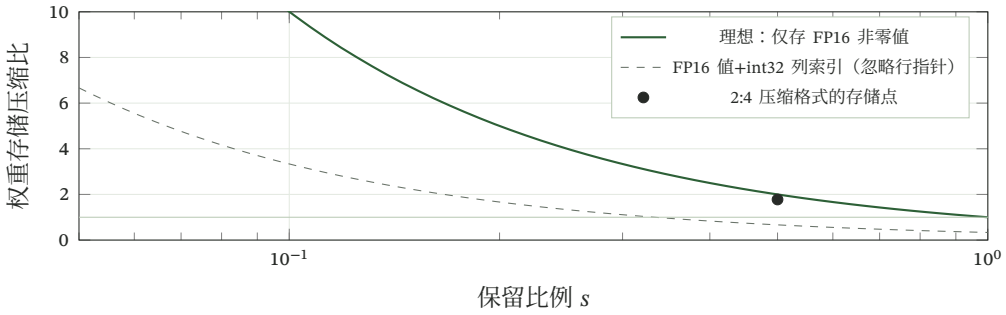


图 5.2 保留比例与存储账本。CSR 示例为 $1/(3s)$，2:4 为约 1.78；图中没有性能实测点，任何曲线都不能直接当作加速比。忽略行指针、padding 及格式头信息，使这些压缩比仍属理想估算。

! 常见误解

稀疏率从 90%增到 99%时，非零计算从原稠密的 10%降到 1%，即相对原剩余工作再降 90%，不是只降 9%。索引总量也下降，单个非零元的索引/值比例不变。性能可能改善或恶化，取决于质量是否可接受、负载平衡与实际 kernel；不能仅从稀疏率高低或一个事务利用率假设作结论。

5.4 MoE：总参数与激活参数分开比较

MoE 把稀疏的作用点从权重换成了 token。门控网络为每个 token 打分，取前 K 个专家，只有这 K 个专家的 FFN 参与该 token 的前向【论文】[40]。它 2017 年就出现了，成为主流靠的是一个反直觉的账本：与相同总参数的稠密计算相比，*MoE 减少每 token 激活的计算与可能读取的权重；全部专家驻留仍占总参数容量。*

「细粒度专家 + 共享专家」这个配置不是一次到位的：DeepSeekMoE 先把专家切细（更多、更小的专家，让激活组合更灵活），再固定若干共享专家吸收通用知识、把路由专家留给专精知识【论文】[41]；V3 的 256 路由 + 1 共享就是这条思路的规模化版本。DeepSpeed-MoE 则从系统侧给出了互补的结论：专家并行下真正吃紧的是**显存**而不是算力，因此需要把专家切到张量维度上、并配合分层通信才能扩展【论文】[42, 43]。

以 DeepSeek-V3 的公开配置为例：671B 总参数、37B 激活参数、61 层、隐维 7168、前 3 层为 dense FFN，后 58 层各 1 个共享专家加 256 个路由专家、每 token 激活 8 个路由专家、专家中间维 2048【论文】[8]。代入两个公式：

$$M_{\text{weight}} = N_{\text{total}} \cdot b_w = 671 \times 10^9 \times 2 = 1.34\ \text{TB}, \tag{5.4}$$

$$F_{\text{token}} = 2N_{\text{active}} = 2 \times 37 \times 10^9 = 74\ \text{GFLOP/token}. \tag{5.5}$$

对照一个稠密 70B 模型（140 GB 权重、140 GFLOP/token）：V3 的权重显存是它的 9.6 倍，单 token 计算量却只有它的一半【自算】。这就是「容量大、算力省」。

• MoE 把压力搬到了哪里

量化、蒸馏、KV 压缩、投机解码——第 4 章及其后的所有手段都在做同一件事：减少每 *token* 要搬的字节或要做的计算。MoE 可以在相近激活计算量下增大总参数；与哪一个稠密基线相比，必须先说明质量与规模。于是它的瓶颈不再是 FLOPS，而是三件别的事——显存容量（1.34 TB 要 17 张 80 GB 卡才装得下理想 BF16 权重，尚未给 KV 与工作区留空间）、显存带宽（无权重复用时，活跃参数按 BF16 约 74 GB/token，实际 batch 还会共享专家读取）、以及互联带宽（token 要在卡间往返）【自算】。把 MoE 当成「更快的稠密模型」来规划容量，一定会同时低估卡数与网络需求。

显存有多紧张，可以再算一笔。MLA 把每 token 每层的 KV 压到 576 个元素（512 维压缩潜向量加 64 维 RoPE），FP16 下 1152 B；MoE 路由层 58 层合计约 65.0 KiB/token（若把全部 61 层都采用该 MLA 缓存，

则为 68.6 KiB/token），32k 上下文的一个请求约 2.1 GiB【自算】。相比 Llama-3-70B 的 320 KiB/token（第 2 章），KV 便宜了 4.7 倍——但权重贵了 9.6 倍。两个数字放在一起，就是 MoE 必须走专家并行的理由：单卡装不下，只能把专家分散到几十上百张卡上，然后为每次前向付一次 all-to-all。

5.5 EP 的 all-to-all：把通信量算出来

设 EP 组大小为 E，一步内该组处理的 token 总数为 T，每 token 选 K 个专家，隐维 h，每元素 b 字节，模型有 L_{MoE} 个路由层。dispatch 阶段要把每个 token 的隐状态送到 K 个专家所在的卡，combine 阶段要把 K 份专家输出送回原卡再按门控加权。于是每层的跨卡字节数是

$$V_{\text{layer}} = 2\,T K h\,b, \tag{5.6}$$

全模型 $V = 2L_{\text{MoE}}TKhb$，表示 dispatch 与 combine 两阶段累计发送的逻辑字节（不再加一遍接收）。假设全部走所比较链路、均匀分布、每卡有效单向发送带宽为 β，带宽时间下界为 $t_{\text{a2a}} \geq 2L_{\text{MoE}}TKhb/(E\beta)$（这正是 DeepEP 这类库的优化对象：FP8 dispatch 把字节减半，IBGDA 降低单次延迟【官方文档】[44]）。这个下界与每卡算力无关，只与 token 数、专家数、隐维和互联带宽有关——这是 MoE 与稠密模型分道扬镳的地方：稠密与 MoE 都可能通过加卡获得容量、带宽与算力，但通信税和权重复用不同。

▶ DeepSeek-V3 规模的 EP 通信量

参数取自公开技术报告【论文】[8]：$L_{\text{MoE}} = 58$，$h = 7168$，$K = 8$（共享专家通常本地计算，不计入路由 top-K），$b = 2$（BF16），decode 部署的 EP 组 $E = 320$。取一步内 $T = 8192$ 个 token（教学总 token 数，不声称复现某个公开部署的精确批量）。

每层每卡：

$$\frac{2 \times 8192 \times 8 \times 7168 \times 2}{320} = 5.87 \times 10^6 \text{ B} \approx 5.6 \text{ MiB},$$

全路由层：5.872 MB × 58 ≈ 340.6 MB，约 324.8 MiB。

按两种互联代入：

- 跨节点 IB 400 Gb/s（50 GB/s）：$340.6 \times 10^6/(50 \times 10^9) \approx 6.81$ ms；
- 节点内 NVLink 400 GB/s：≈ 0.85 ms。

同一步的计算量：8192×74 GFLOP = 606 TFLOP，摊到 320 张卡是每卡 1.89 TFLOP；按 989 TFLOPS 峰值、40% 的 MFU 计，约 4.8 ms。

结论：走 IB 时通信（6.81 ms）约为计算（4.8 ms）的 1.42 倍，走 NVLink 的纯带宽项约 0.18 倍。V3 的技术报告自陈训练时「跨节点 EP 使计算通信比劣化到约 1:1」，只能作为背景对照，训练通信比不能校准 decode 算例。此处仅说明在所设参数下，跨节点 *EP* 的通信时间与专家计算时间同阶。

［口径：跨节点按每卡 400 Gb/s IB、节点内按 400 GB/s NVLink 计，均取公开的 H800 集群规格；MFU 取 40%。这个值对 MoE decode 偏乐观，若实际只有 20%，计算耗时翻倍到 9.6 ms，通信/计算比约降到 0.71 ——但那时整机吞吐也已经减半，瓶颈只是换了个位置。］

把总 token 数换成 1，58 路由层的累计逻辑字节为 $2 \times 58 \times 8 \times 7168 \times 2 \approx 13.30$ MB，相对 74 GB 活跃权重为约 0.018%【自算】。但每层通信启动与同步延迟不会按字节同比缩小，不能因此忽略 batch=1 的 all-to-all。其带宽项随 token 数线性增长，这与权重读取的规律刚好相反（后者随 batch 摊薄）。小 batch 主要检查启动延迟，大 batch 再检查带宽与负载不均。

但它的延迟特性更麻烦：每层两次 all-to-all 都是同步点，完成时间不是平均值的函数，而是最慢接收方的函数。假设某个专家收到 1.6 倍于均值的 token，它所在的卡就是关键路径，其余卡全部空等。58 个 MoE 路由层、每层两次屏障，等于 116 次「等最慢者」的机会，尾延迟逐层放大。这就是为什么 EPLB 的优化目标是 max 而不是 mean，也是为什么 decode 阶段要把专家切到「每卡一个」再用冗余副本摊平热点。

图 5.3 把一次 dispatch/combine 的字节量按专家分布画出来：理想的均匀分布下每卡收发相等，而实际路由的偏斜会让某些卡成为瓶颈——这就是后面 EPLB 要解决的问题。

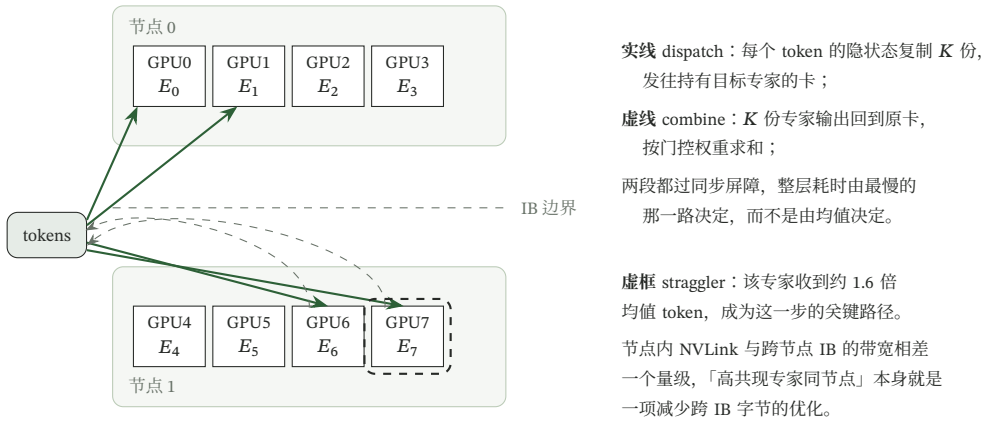


图 5.3 专家并行的 all-to-all 数据流。token 从左侧出发，K 份副本分别落到本节点与其他节点的专家上（实线），专家输出再回到原卡加权求和（虚线）。跨节点的一跳要过 IB 边界，带宽比节点内 NVLink 低一个量级。被虚框圈出的专家是这一步的 straggler：它收到约 1.6 倍均值 token，整层完成时间由它决定，而不是由平均负载决定。

5.6 五种工程手段，四个不同的病因

EP 的性能问题常被笼统说成「通信慢」，但拆开看至少有四个病因：通信量太大、负载不均导致 straggler、*kernel* 效率低（padding 与 launch 开销）、热点专家排队。表 5.2 是它们的对照，下面逐条说明各自治哪一个。

EPLB（专家并行负载均衡）治负载不均。它拿各专家的历史负载（滑动平均）当输入，输出一张「专家副本 → 卡」的映射，目标是让每张卡的负载尽可能相等。DeepSeek 开源的实现给了两套策略：层次均衡用于 prefill（节点数能整除专家组数时，先把专家组按节点打包，再在节点内复制与重排），全局均衡用于 decode（EP 域很大时忽略分组、全局复制）【官方文档】[45]。它同时改善 max 负载与跨节点流量。代价是在线统计与周期性重排，而重排意味着迁移权重——671B 模型上一次全量重排是几百 GB 的显存搬运。

冗余专家治热点。把高负载专家复制一份放到另一张卡上，让两个副本分摊 token。V3 的 prefill 部署设置了 32 个冗余专家，每张卡在原有 8 个专家之外再多放 1 个；decode 部署则用 64 张卡专门承载冗余与共享专家【论文】。代价直接而硬：每个副本一份权重，静态推理权重通常不需逐步同步，但在线版本切换、热更新与副本一致性需要管理。

grouped GEMM 治 kernel 效率，与通信无关。每个专家收到的 token 数不同，朴素做法要么按最大专家 padding（浪费算力），要么逐专家 launch（浪费启动开销）；grouped GEMM 把变长批按 expert 分段，减少逐专家 launch 与 padding；是否完全无 padding 由 kernel 决定。MegaBlocks 更进一步，用 block-sparse 表达整个 MoE 层，既不丢 token 也不 padding，训练端相对 Tutel 最高快 40%【论文】[46]；Tutel 本身用自适应并行与流水把单层 MoE 在 16 张 A100 上加速 4.96 倍【论文】[47]。这两个都是训练口径的数字，推理端的收益取决于 batch 与专家数，不能直接搬。

专家放置治通信量。把经常被同一批 token 选中的专家放在同一节点，跨 IB 的字节就变成节点内 NVLink 的字节，β 直接差一个量级。V3 的办法更彻底：在路由阶段就加约束，保证每个 token 最多被送到 4 个节点【论文】，把「放置」问题前移成了「路由」问题。代价是路由自由度下降，可能轻微损害模型质量。

专家亲和调度治的也是通信量，但作用在请求入口：把路由倾向相近的请求聚到同一组卡，让同一专家的副本尽量服务同一批请求，同时复用它们共享的前缀 KV。它与负载均衡天然冲突——亲和意味着集中，集中就意味着不均，两者要在同一个目标函数里折中。它是少数能在不增加显存的前提下降低 *all-to-all* 字节数的手段，在长上下文、高 QPS 场景里值得优先尝试【自算】。

5.7 稀疏注意力：索引怎么落在分页上

第 3 章讲了稀疏注意力的算法谱系（sliding window、sink、NSA、MoBA、DSA）。这里只讲那边不会展开的一件事：当 *KV* 已经按页管理时，稀疏索引存在哪里、又破坏了什么。

表 5.2 MoE 的五种工程手段及其针对的病因。最后一列是「不做会怎样」，它比「做了会怎样」更能说明每项手段的必要性；四列合起来也解释了为什么没有单一手段能把 EP 调好。

手段	缓解对象	机制	不做的后果
EPLB	负载不均	按历史负载复制专家并重排到卡，最小化 max 卡负载	单卡成瓶颈，其余卡空等
冗余专家	热点专家	复制高负载专家，多副本分摊 token	热点专家排队，尾延迟尖刺
grouped GEMM	kernel 效率	变长专家批合并为一次 launch，减少 padding/launch	逐专家开销或零填充浪费
专家放置	通信量	高共现专家同节点；路由限制跨节点数	跨 IB 字节数翻倍
亲和调度	通信量	请求按路由倾向聚类到同组卡	同专家副本被反复跨节点访问

以 DeepSeek-V3.2 的 DSA 为例。它用一个「闪电索引器」为每个 query 算出它与所有历史位置的分数（少量 head、FP8、ReLU），取 top-k = 2048 个 KV 条目参与注意力，主注意力的复杂度从 $O(L^2)$ 降到 $O(Lk)$【论文】[17]。在 MLA 结构下有一个关键约束：每个 *KV* 条目必须被多个 *query* 共享，否则每个 query head 各选各的，gather 次数要乘以头数。所以 DSA 建在 MLA 的 MQA 模式上，128 个 query head 共享同一份选择集【论文】。

在实现层面，KV 仍然按页存（第 7 章的块表），稀疏索引是叠加在块表之上的一张「选择表」：每个 query 一行，存 k 个位置，每个位置是（页号，页内偏移）。若索引器另存一个 128 维 FP8 key，则每 token 每层额外 128 B，相对于 1152 B 的 MLA 缓存约 11.1%【自算，条件设计账本】。实际 DSA 配置还要计 scale、head 维、路由层数、池布局和 padding。128 B 是上述条件设计账本的输入；发布实现的内存布局须按固定模型与引擎版本核验，不能直接由技术报告的算法描述推出。短序列的选择开销可能超过主注意力节省，需按长度扫描密集/稀疏交叉点。

prefix caching。命中的前缀 token 不再作为新 query 执行，因此其索引计算也可以省去；但新增 query 的选择集依赖它自己的 query，不能直接拿历史选择结果复用。若算法因果、位置与模型状态相同，chunked prefill 仍可与整段执行数学等价：每个 query 都对完整可见历史做选择即可，不要求事后跨 query 或跨 chunk 额外归并 top-k。若实现只在局部块内选择，则是另一种近似，必须明说。

页与索引。选择集可能散落多页，影响 gather 合并、L2 命中和负载平衡。定长 top-k 并不保证内层访问成本一致；应观测实际读字节与页分布，而非只计选择数。

选择本身。单 decode query 的索引评分通常随历史长度线性增长，整段 prefill 可为二次复杂度。常数小不等于可永久忽略；极长上下文下索引器可能成为瓶颈。

练习：通信时间为何不由一个 GB/s 决定

对 58 层、每层两次通信，设每次启动/同步延迟为 5 微秒。即使逻辑发送量趋近零，累计延迟仍约 $58 \times 2 \times 5 = 580$ 微秒（无重叠假设）。再把跨节点字节比例从 1 降为 0.25，比较字节项和延迟项分别改变多少。**提示：**平均带宽下界不包含热点 rank 与拥塞，不能直接用来保证 50 ms 最大 ITL。

未解问题

1. **给定模型，稀疏配置能不能先验预测？**今天选 2:4 还是 4:8、top-2048 还是 top-4096，靠的都是扫点实测，而每次扫点都要重训或至少重评估。缺少一个把权重与激活分布、kernel 实现、互联拓扑联合起来的模型，能直接回答「这个模型在这套硬件上的最优稀疏率是多少」。在此之前，稀疏的调参成本被系统性低估。
2. **EP 的尾延迟难以从平均路由预测**。all-to-all 的完成时间由最慢专家决定，而路由是数据相关的：同一段文本在不同 batch 组成下会得到不同的专家分布。需要结合容量、路由限制、准入与故障假设给出统计保证；不能仅由数据相关路由推出“没有任何上界”。
3. **稀疏注意力与缓存机制如何共存？**选择集依赖 query，prefix cache 只能缓存 KV 不能缓存选择；PD 分离又把 prefill 与 decode 拆到不同机器，选择集要不要跟着 KV 一起搬、搬多少，都没有标准答案。

第 6 章　并行策略与显存账本

本章结论

(1) TP、PP、DP、EP 与 CP 切分不同维度；每种组合需同时检查权重/KV 容量、通信字节与延迟、kernel 形状和服务 SLO。字节少并不保证延迟低，理想分片也未必受引擎支持。

(2) 小消息 TP 常受启动与同步延迟限制，大 batch 时消息字节也会进入关键路径。跨互联域的边界由拓扑、算法与每层工作量共同决定，不能把“TP 只在 8 卡内”当硬定律。

(3) 70B/32k 压力例与 671B MoE 的内存布局不同。前者 KV 大，后者总权重大；显存账须明确 GB/GiB、实际可见容量和工作区假设。多 LoRA 还需按活跃适配器数记额外读取。

6.1　五种切法，五张通信账单

一个推理请求在集群上的分布方式，可以看成对五个维度的切分：请求维（DP）、层维（PP）、层内矩阵维（TP）、专家维（EP）、序列维（CP）。切哪个维度，决定了通信原语，也决定了这种并行对 batch 的敏感程度。图 6.1 把前三种画出来：TP 把同一层的权重竖着劈开，PP 把层横着叠到不同卡上，EP 则让同一个 token 在不同卡之间跳来跳去。

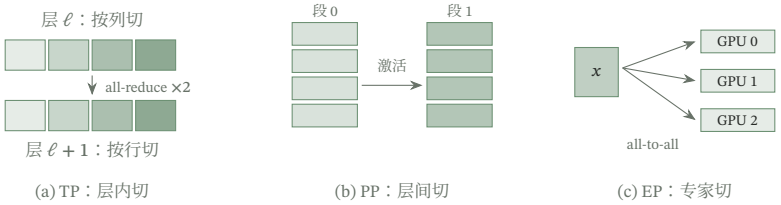


图 6.1　三种切法的拓扑差别。TP 的通信发生在同一层内部，出口是 all-reduce，小消息时延迟项与层数相关，大 batch 时字节项也增长；PP 的通信发生在层与层之间，只传激活，开销小但必须靠多个 micro-batch 才能填满流水；EP 的通信由路由决定，每层两次 all-to-all，字节项随 token 数增长，小 batch 还受启动延迟限制。DP 副本的请求可独立执行（副本内部仍可用 TP/PP），CP 的一种实现用 ring 传 KV，两者不在图中展开。

表 6.1 把五本账并排放在一起。读法是：先看「每卡权重 / KV」，它决定这种并行能否解决「装不下」；再看「通信原语」，它决定固定开销；最后看「batch 要求」，它决定这种并行在延迟敏感的场景里是否可用。

表 6.1　五种并行的通信账。N 是该维度的并行度，p 为 PP 级数，m 为 micro-batch 数，n_{kv} 为 KV 头数。「对 batch 的要求」一列是选型的硬门槛：它是初筛条件，最终要按具体实现的吞吐、容量与延迟验证。

并行	每层通信原语	每卡权重 / KV	对 batch 的要求	适合的规模
TP	常见为 2 次 all-reduce	参数/N；KV 祝头分片	不敏感，$b = 1$ 亦可	优先拓扑紧密互联域
PP	点对点传激活	P/p / KV/p	需 m 个 micro-batch，$m \gg p$	跨机、prefill、训练
DP	副本间推理通常无	每副本全模型；本地请求 KV	$b \geq N_{\text{dp}}$	任意规模，扩吞吐
EP	2 次 all-to-all	$P_e/N_e + P_{\text{ne}}$ / 本地全量	越大越好	多机，MoE
CP	ring 传 KV 块	P / KV/N	长序列、小 batch	长上下文 prefill 与训练

TP 切矩阵，换两次 all-reduce。Megatron-LM 的分法是把注意力与 FFN 的第一个矩阵按列切、第二个按行切，于是每张卡算出的是部分和，需要每层两次 all-reduce 加回去 [48]。每卡权重 P/N，KV 按 KV 头切分，所以 N 实际上被 n_{kv} 卡住了：Llama-3-70B 的 $n_{\text{kv}} = 8$，TP=8 已是 KV 切分的上限，再往上只能复制 KV 头，显存不再下降【自算】。TP 最独特的性质是对 *batch* 不敏感—— batch=1 时它照样有效，这是它与其他四种并行的根本区别，也是它成为低延迟首选的原因。

PP 切层，换点对点。第 i 段持有 L/p 层，段间只传激活，不传梯度也不传 KV [49]。每卡权重与 KV 都降为 $1/p$，看似与 TP 等价，但它对 batch 的要求最苛刻：必须有足够多的 micro-batch 把流水填满，否则后段 GPU 在等前段。这条要求在第 6.3 节会被量化成一条不等式，结论是它在 decode 场景几乎不可满足。

DP 切请求，换零通信。每张卡一份完整模型，各跑各的请求，推理时卡间无通信（训练时才有梯度 all-reduce）[50]。一个副本可以由单卡或 TP/PP 组组成；需要每个副本装得下且上游与负载均衡跟得上，吞吐才可能近似按副本数扩展。所有「加机器就能扩容」的直觉都来自 DP，而所有「加机器反而变慢」的事故都来自把 TP 当成了 DP。

EP 切专家，换 all-to-all。N_e 个专家分到 N 张卡，token 经路由后被送到它选中的专家所在的卡，算完再送回来，所以每层两次 all-to-all[51, 52]。每卡权重是「专家分片 + 完整非专家层」，这个不对称是账本 B 的全部麻烦所在。EP 对 batch 极其友好，但对互联延迟极其敏感。

CP 切序列，换 ring。每卡持有 ℓ/N 个 token 的 KV，注意力需要看到全部 KV，于是用 ring 把 KV 块依次传给下一张卡 [53]。所描述的纯 CP 布局复制权重，但也可与 TP/PP 组合。CP 可减少长序列的状态与计算压力；decode 还可广播 query 并合并部分 softmax 统计，不必每步环传全部 KV。选型取决于实现、长度和延迟预算。

6.2 TP 的边界：一次 all-reduce 的时间账

「decode 每次只算 1 个 token」这句话的真正含义是：计算量小到无法掩盖任何固定开销。设 hidden 维度为 d，激活每元素 b_a 字节，则每层两次 all-reduce 的消息大小是 $M = d\,b_a$，与模型大小无关。70B 模型 $d = 8192$，FP16 下 $M = 16$ KiB ——这是 batch=1 的消息大小；batch 为 B 时消息通常为 Bdb_a，不能总忽略带宽项。

用 α-β 模型写 ring all-reduce 的耗时 [54]：

$$T_{\text{ar}}(N, M) \approx \underbrace{\frac{2(N-1)}{N} \cdot \frac{M}{B_{\text{link}}}}_{\text{传输}} + \underbrace{2(N-1)\tau}_{\text{逐跳}} + \underbrace{c}_{\text{启动}}. \tag{6.1}$$

其中 B_{link} 是单向链路带宽（NVLink 4 每卡 900 GB/s 双向，即 450 GB/s 单向【官方文档】[55]），τ 是每跳延迟，c 是 kernel 启动与同步开销。第一项随消息增大而增大，后两项与消息大小无关——这正是小消息 all-reduce 的本质：它买的是「同步」，不是「带宽」。

单层的 decode 时间因此是「读本卡权重」加「两次 all-reduce」：

$$T_{\text{layer}}(N) = \frac{P_\ell\, b_w}{N B_{\text{mem}}} + 2\,T_{\text{ar}}(N,\, d\,b_a), \qquad T_{\text{tok}} = L \cdot T_{\text{layer}}, \tag{6.2}$$

P_ℓ 是每层参数量，B_{mem} 是 HBM 带宽。把 70B（$L = 80$，$P_\ell \approx 875$ M，FP16）代进去，得到下面的四行账。

▶ TP=8 与 TP=16 的时间账

假设：H100 SXM，$B_{\text{mem}} = 3.35$ TB/s【官方文档】[3]，$B_{\text{link}} = 450$ GB/s；跨机走 400 Gb/s IB，单向 50 GB/s；取 $\tau_{\text{NVLink}} = 0.3\,\mu s$、$\tau_{\text{IB}} = 2\,\mu s$、$c = 3\,\mu s$（NVLink）/ $5\,\mu s$（IB）【教学假设，未在本环境测量 NCCL；不同 collective 算法不能套用同一逐跳参数】。

配置	每层权重读取	T_{ar}	两次 all-reduce	每层合计
TP=1	522.0 μs	—	0	522.0 μs
TP=8（NVLink）	65.3	7.3	14.6	79.9
TP=16（NVLink 假设）	32.7	12.1	24.2	56.9
TP=16（IB NDR）	32.7	65.6	131.2	163.9

代入过程：每层权重读取 $= P_\ell b_w/(NB_{\text{mem}})$，$N = 8$ 时为 $875 \times 10^6 \times 2/(8 \times 3.35 \times 10^{12}) = 65.3\,\mu s$。all-reduce 的传输项 $= 2 \times \frac{7}{8} \times 16\,\text{KiB}/450\,\text{GB/s} = 0.06\,\mu s$，逐跳项 $= 14 \times 0.3 = 4.2\,\mu s$，于是 $T_{\text{ar}} = 7.3\,\mu s$。

换成 16 卡走 IB 时，传输项只有 0.61 μs，逐跳项却涨到 $30 \times 2 = 60\ \mu s$。
结论：TP 从 8 加到 16，每卡权重读取省下 32.6 μs，通信却多付 116.6 μs，每层净亏 84 μs，整体慢 2.05 倍（每 token 从 6.4 ms 涨到 13.1 ms）。

这张表说明在所假设的小消息和 ring 算法下延迟项主导。把消息从 16 KiB 放大到 1 MiB（batch64），NVLink 模型的传输项约 4.1 微秒，IB 模型则约 39.3 微秒，已不能当零；tree、分层与自定义 all-reduce 也会改变结果。所以「跨机 TP 只要带宽够就行」这句话是错的：带宽是乘数，延迟是加数，而 decode 的小消息场景里加数赢。

把「TP 翻倍是否划算」写成不等式，得到本章第一个判据：

$$2\big[T_{\mathrm{ar}}(2N) - T_{\mathrm{ar}}(N)\big] \;<\; \frac{P_\ell\, b_w}{2N\, B_{\mathrm{mem}}}. \tag{6.3}$$

右端是翻倍省下的权重读取时间，不是计算时间。对表中 70B 参数，右端约 32.6 微秒；保留 c 与传输项后，跨域目标模型近似要求 $2[(30\tau + 5 + 0.61) - (4.2 + 3 + 0.06)] < 32.6$，即 $\tau \lesssim 0.60$ 微秒【自算】。这只约束本例参数化 ring 模型，不证明 InfiniBand 物理上不能更低，也不证明 NVLink 域只能 8 卡。现代 scale-up 域与 collective 实现各有边界，第 12 章区分它们。

! 常见误解

这条判据常被误读成「TP 永远不能跨机」。注意右端正比于 P_ℓ：模型越大，单层越重，能容忍的延迟越高。把 70B 换成 1T 稠密模型（每层约 12 B 参数），右端变成 447 μs，在相同近似下约 $\tau < 7.5\ \mu$s，IB 也能满足【自算】。所以准确的说法是：*TP* 能跨多远，取决于单层有多重。本例 70B 跨机 ring 参数下收益为负；千亿级单层极厚的模型，跨机 TP 仍有人做，代价是必须接受拓扑与拥塞带来的长尾。

同样的算式可解释为何小模型 TP 经常不如多副本经济：8B 模型（$L = 32$，$d = 4096$）每层权重读取在 TP=8 下是 18.7 μs，而两次 all-reduce 是 14.4 μs，通信占 43%，8 张卡只换来 4.5 倍加速；同一批卡做 DP=8 则是 8 倍吞吐，且单卡 4.8 ms 的 ITL 早已满足在线 SLO【自算】。这些是忽略 KV 与实际 kernel 的条件估算；若更严格延迟、长上下文或单卡容量不够，小模型也可能需要 TP，不能设一个通用 70B 门槛。

6.3 PP 的气泡与权重重读

把模型切成 p 段，每段前向耗时 t，把一批请求分成 m 个 micro-batch。第 j 个 micro-batch（$j = 0, \dots, m-1$）在时刻 jt 进入第 1 段，第 i 段在 $(i-1+j)t$ 开始处理它。最后一个 micro-batch 离开第 p 段的时刻是 $(p-1)t + (m-1)t + t = (m+p-1)t$，而每段真正忙碌的时间只有 mt。因此气泡占比与有效利用率分别是

$$\beta \;=\; \frac{p-1}{m+p-1}, \qquad U \;=\; 1-\beta \;=\; \frac{m}{m+p-1}. \tag{6.4}$$

这个式子说明气泡不是「通信开销」，而是结构性空闲：它不随硬件变快而消失，只随 m/p 变大而变小。要让 $\beta \le 10\%$，需要 $m \ge 9(p-1)$；$p = 8$ 时就是 63 个 micro-batch。图 6.2 画出四条曲线，可以直接读出每个 p 需要多少 m。

气泡只是 PP 的第一笔账。第二笔是权重重读：每个 micro-batch 都是一次独立前向，都要把本段权重从 HBM 完整读一遍；TP 每步只读一次权重（整个 batch 合成一次 GEMM）。把两笔账写在一起，一步的耗时是

$$T_{\mathrm{step}}(m) \;=\; (m+p-1)\Big(c_0 + \frac{R\,c_1}{m}\Big) \;=\; m\,c_0 + R\,c_1 + (p-1)\Big(c_0 + \frac{R\,c_1}{m}\Big), \tag{6.5}$$

其中 R 是每步的并发请求数，c_0 是一个 micro-batch 通过一段的权重读取时间，c_1 是一个请求通过一段的 KV 读取时间。与同卡数的理想并行相比，耗时比为

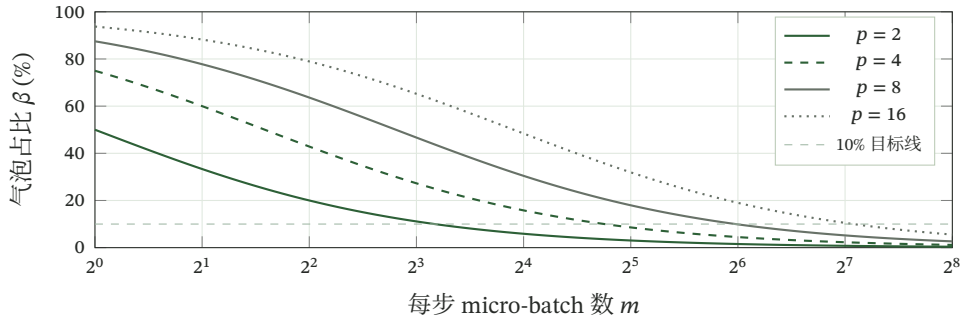


图 6.2 气泡占比随 micro-batch 数的衰减是一个双曲线，不是指数：$p = 8$ 时从 $m = 8$ 到 $m = 16$，气泡从 47% 降到 30%；再从 16 到 32，只降到 18%。要让气泡低于 10%，$p = 2$ 只需 9 个 micro-batch，$p = 16$ 需要 135 个——在本图有限批次排空模型中，较深流水需要很多 micro-batch 才能降低气泡；稳态异步请求流水可采用不同调度，不能套用本图给所有在线 PP 判不可行。

$$\frac{T_{\text{PP}}}{T_{\text{TP}}} = \frac{m+p-1}{m}\left(1+\frac{(m-1)\,c_0}{c_0+R\,c_1}\right), \tag{6.6}$$

第一个因子是气泡，第二个因子是重读。令两个相对额外项各不超过 ϵ，忽略低阶项可得一个较保守的条件

$$m \;\geq\; \frac{p-1}{\epsilon}, \qquad R \;\gtrsim\; \frac{(p-1)\,c_0}{\epsilon^2\,c_1}. \tag{6.7}$$

▶ PP=2 与 PP=8 在 70B / 32k 上的账

取 70B、32k 上下文、TP=8、FP16，$kL = 320$ KiB/token（第 2 章）。$c_0 = \dfrac{W}{pN_{\text{tp}}B_{\text{mem}}}$，$c_1 = \dfrac{kL\ell}{pN_{\text{tp}}B_{\text{mem}}}$。$p = 2$：$c_0 = 141.2\times10^9/(2\times8\times3.35\times10^{12}) = 2.63$ ms，$c_1 = 327{,}680\times32{,}768/(2\times8\times3.35\times10^{12}) = 0.20$ ms。$R = 32$ 时 $Rc_1 = 6.4$ ms，最优 $m^* = \sqrt{(p-1)Rc_1/c_0} = 1.6$，取 $m = 2$：$T_{\text{step}} = 3\times(2.63+3.2) = 17.5$ ms。

对照：同样 32 个 32k 请求，8 卡纯 TP=8 的一步是 $5.27 + 12.8 = 18.1$ ms。PP=2 用了两倍的卡（16 张），一步只快 0.6 ms ——加卡没有换来吞吐。若与「16 卡 TP=16 的理想情形」（$2.63 + 6.4 = 9.0$ ms）比，PP=2 慢 1.94 倍，与式 (6.6) 的预测一致。

再看判据 (6.7)。$c_0/c_1 = W/(kL\ell) = 141.2\times10^9/1.074\times10^{10} \approx 13.2$，与 p 无关。取 $\epsilon = 0.1$：$p = 2$ 需要 $m \geq 10$ 且 $R \geq 1320$ 个 32k 并发；$p = 8$ 需要 $m \geq 70$ 且 $R \geq 9200$ 个 32k 并发。后者约 92,000 GiB KV，说明该假设下的低开销目标代价很大；它不排除其他流水调度或不同硬件资源下的可行配置。

这个算例只说明有限 batch 同步排空、每 micro-batch 重读且没有其他瓶颈时，PP 难以追平理想 TP。稳态异步流水、不同阶段平衡、通信重叠和 TP 跨域成本会改变比较。选择 PP 的常见理由包括容量和拓扑：第一，模型大到 TP 域装不下（405B FP16 需要 810 GB，8 卡放不下）。第二，TP 已经顶到 NVLink 域上限，再扩只能跨机，而跨机 TP 的延迟账（第 6.2 节）已经不划算。第三，prefill 或训练：那里一个 micro-batch 有几千个 token，c_0 被摊薄几千倍，判据 (6.7) 轻松满足。

prefill 与训练的 micro-batch 常包含更多 token，可更好摊薄权重读取；decode 则需要用实际到达流、反馈依赖与流水调度测量，不应把训练的气泡式直接当在线 SLO 预测。

6.4 账本 A：Llama-3-70B，32k 上下文，batch 32

现在把前面的公式落成一张可以拿计算器复算的表。配置：Llama-3-70B（70.6 B 参数，$L = 80$，$d = 8192$，64 头，GQA $n_{\text{kv}} = 8$，$d_h = 128$）【官方文档】[5]，FP16 权重与 KV，32k 上下文，32 并发，TP=8，PP=1。

表 6.2 Llama-3-70B 在 8×H100 80GB 上的单卡显存账本。每一行的计算式都可以独立复算；“碎片”6%、工作区与 graph 均为教学预算假设，并非本次实测。按标称 80GB 换算且仅允许 90%预算，可用约 67.1GiB；实机必须以驱动报告的字节数重算。

项目	计算式	代入	GiB
权重	$Pb_w/(N_{\text{tp}}2^{30})$	$70.6\times10^9\times2/8$	16.4
KV cache	$2Ln_{\text{kv}}d_h\ell Bb_{\text{kv}}/(N_{\text{tp}}2^{30})$	$2\cdot80\cdot8\cdot128\cdot32768\cdot32\cdot2/8$	40.0
激活峰值	见正文，同时存活 3 项	chunk = 8192	1.1
固定开销	CUDA context + NCCL + graph	0.9 + 0.4 + 0.8	2.1
碎片	6%× 前四项	0.06×59.6	3.6
合计			63.2
预算	$0.9\times80\times10^9/2^{30}$		67.1
余量			3.9（约 6%）

▶ **逐项复算与判断**

权重：$70.6\times10^9\times2$ B = 141.2 GB，除以 8 得 17.65 GB = 16.4 GiB。
KV：由第 2 章的式 (2.9)，$k = 2n_{\text{kv}}d_hb = 4096$ B/token/layer，$kL = 320$ KiB/token。32 个请求各 32k token：$320\,\text{KiB}\times32768\times32 = 320$ GiB，除以 8（KV 头被 TP 切开）得 40.0 GiB。校验：每卡每 token 每层 $2\times1\times128\times2 = 512$ B，乘 80 层得 40 KiB/token，乘 32768×32 恰好 40 GiB。
激活峰值：原稿把残差、FFN 与 logits 全部除以 TP 并乘一个经验系数，不能作为精确峰值：残差可能复制，FFN 按矩阵切分，而 logits 可仅在最后位置计算或被收集。本表 1.1 GiB 只作工作区预算，须用固定引擎、chunk、词表与 CUDA Graph 配置的峰值测量替换；没有这项测量就不能声称“8192 chunk 必定放得下”。
固定开销：CUDA context 与 cuBLAS workspace 约 0.9 GiB，NCCL 通信缓冲约 0.4 GiB，CUDA Graph 抓取池约 0.8 GiB【教学预算假设，非本次测量；随驱动、引擎、并行与图捕获设置变化】。
判断：固定假设下 16.4 + 40.0 + 1.1 + 2.1 = 59.6 GiB，加 6%约 63.2 GiB；按标称 80GB 的 90%预算约 67.1 GiB，仅剩约 3.9 GiB。“32k”须含输入与生成；若总长再多 2048，KV 增加 2.5 GiB/卡，总量约 65.8 GiB，余量约 1.3 GiB。模型整数参数、运行时分片与实际可见容量需逐项校准。
隐含性能下界：理想权重读取约 5.3 ms，KV 读取约 12.8 ms，合计约 18.1 ms（两项共用 HBM 带宽）。这是带宽时间下界，不是 ITL 上界；忽略通信、计算、低效访问与调度。对应纯 decode 聚合输出上界约 1.8k token/s，通过该下界不证明 50 ms 最大 ITL 与 2 s TTFT 能达标。

当余量不够时（要支持 34k 总长、或把 batch 提到 48、或不做 prefill 分块），只有三条路，每条都标好代价：

解法 1. 降 batch，32→24。KV 降到 30.0 GiB，按同一预算合计约 52.6 GiB，余量约 14.5 GiB。理想步时间约从 18.1 降至 14.9 ms，而吞吐从约 1768 降至 1615 token/s，并非按 batch 同比下降 25%；排队变化还取决于到达率。

解法 2. KV 降到 FP8。$b_{\text{kv}} = 1$，KV 减半到 20.0 GiB，合计 42.0 GiB，余量约 25.1 GiB。代价是精度风险与 attention kernel 的支持度，第 4 章会给出 FP8 KV 的掉点量级。

解法 3. 加卡上 PP。8 卡扩到 16 卡，TP=8 × PP=2：每卡权重 8.2 GiB、KV 20.0 GiB，合计 32.8 GiB，余量约 34.3 GiB。代价是第 6.3 节算过的气泡与重读，实际 ITL 变化需按流水调度测量，不能从显存表直接断言涨幅。

! **常见误解**

「显存不够就把 TP 换成 PP」是错的。权重和 KV 都按 $N_{\text{tp}}\times N_{\text{pp}}$ 的乘积切分，在同一批 8 张卡上把 TP=8 改成 TP=4×PP=2，乘积仍是 8，理想权重与 KV 项相同，但 KV 头复制、激活和工作区可能不同，性能也随通信/流水变化。当 TP 已出现 KV 头复制时，用 PP 替换部分 TP 也可能减少重复 KV；

应检查布局。在本例理想可均分条件下，不加卡不会自动降低权重与 KV，而激活在这个账本里只占 1.1 GiB。真正在同一批卡上省显存的只有降 batch、降 KV 精度，或者换更省 KV 的注意力结构。

6.5 账本 B：671B MoE 的专家放置

MoE 的账本结构与稠密模型相反：稠密模型的每一份参数都必须被切，MoE 的专家可以切得很碎，但非专家层必须整份复制。于是 EP 的收益随 N_e 增长而递减，成本随卡数线性上升。

DeepSeek-V3 的公开配置是 671 B 总参数、37 B 激活、61 层、$d = 7168$、256 个路由专家加 1 个共享专家、top-8、专家中间维 2048、MLA（KV 压缩维 512 + rope 64），原生 FP8 训练【论文】[8]。据此可以把参数拆开：

▶ **EP=32 与 EP=64 的每卡账**

专家参数：每个专家 $3 \times 7168 \times 2048 = 44.04$ M，每层 256 个即 11.27 B，58 个 MoE 层共 653.7 B。**非专家参数**：671 − 654 ≈ 17 B（注意力、3 个 dense 层、词表），这部分在 EP 下是每卡一份。

每卡占用（GiB）	EP=32	EP=64	说明
专家分片（FP8）	19.0	9.5	$653.7/N_e$ B 参数，1 B/参数
非专家层（FP8）	15.9	15.9	复制，不随 N_e 下降
all-to-all 缓冲	1.3	1.3	式 (6.8)
激活与固定开销	2.7	2.7	0.6 + 2.1
碎片（6%）	2.3	1.8	
合计	41.2	31.2	预算 67.1 GiB
可用于 KV	25.9	35.9	每请求 2.14 GiB
32k 并发数/卡	12	16	

KV：MLA 每 token 每层只存 512 + 64 = 576 个数，BF16 下 $576 \times 61 \times 2 = 70{,}272$ B ≈ 68.6 KiB/token；32k 上下文即 2.14 GiB/请求。由于 DP attention 下每个 rank 只服务自己的请求，这份 KV 不再被切分。**all-to-all 缓冲**：一次 dispatch 的字节数是

$$M_{\text{a2a}} = r_{\text{buf}}(1 + \delta_{\text{load}})skd(b_{\text{dispatch}} + b_{\text{combine}}), \tag{6.8}$$

s 是每 rank 每步 token 数，$k = 8$，$d = 7168$；$r_{\text{buf}} = 1$ 或 2 表示单/双缓冲，δ_{load} 为热点接收余量。表中假设均衡且双缓冲，不含实现的额外路由元数据。prefill 取 $s = 4096$、dispatch FP8（1 B）、combine BF16（2 B）：$4096 \times 8 \times 7168 \times (1 + 2) = 705$ MB，双缓冲后 1.41 GB ≈ 1.3 GiB。decode 取 $s = 128$ 时单缓冲约 22 MB，双缓冲约 44 MB。缓冲要按 prefill 的上界预留，而不是按平均值——这是 MoE 服务里最容易被漏掉的一项。

这张表里最值得记住的是第二行：非专家层不随 *EP* 下降。EP 从 32 加到 64，专家分片省下 9.5 GiB，卡数却翻了一倍，总显存成本翻倍；而非专家层那 15.9 GiB 一动不动，最终成为天花板——当 $N_e \to \infty$ 时，每卡占用收敛到 15.9 GiB 加 KV，EP 的边际收益归零。

非专家层为什么不用 TP 切？因为 MLA 已经把每 token 的 KV 压到 68.6 KiB，通信/计算比比稠密模型更极端：TP 每层两次 all-reduce 的固定延迟不变，而它要摊薄的计算量更小。这正是 DeepSeek 选择 *DP attention* 的原因——注意力与 dense 层每卡一份、各算各的请求，只有 MoE 层做 all-to-all。vLLM 把 DP 与 EP 作为可组合的两个维度暴露出来，也正是为了表达这种布局【官方文档】[56]；低延迟 all-to-all 的实现（如 DeepEP 的 SM 占用与 RDMA 路径）则决定了这套布局在 decode 下能否成立 [44]，第 12 章会展开。

6.6 账本 C：多租户 LoRA 服务

账本 A 是单模型、账本 B 是 MoE。第三种在生产里越来越常见：一个基座模型 + 大量微调适配器。这是企业部署的默认形态——每个客户、每个任务、每个版本都有一个 LoRA，而基座只有一份。

第三种账本单列适配器读取：**KV 不变、基座不变，新增活跃适配器的权重读取。**

6.6.1 适配器有多大

以 70B、全部线性层都挂 LoRA、秩 $r = 16$ 为例。每个被适配的矩阵 $W \in \mathbb{R}^{d\times d'}$ 换成 $W + BA$，其中 $A \in \mathbb{R}^{r\times d}$、$B \in \mathbb{R}^{d'\times r}$，参数量为 $r(d + d')$：

被适配的矩阵	每层个数	每矩阵参数	每层参数
q_{proj}（8192^2）	1	$2 \times 16 \times 8192 = 262$ K	262 K
k_{proj}、v_{proj}（8192×1024，GQA）	2	$16 \times (8192 + 1024) = 147$ K	295 K
o_{proj}（8192^2）	1	262 K	262 K
gate/up/down（8192×28672）	3	$16 \times 36864 = 590$ K	1,770 K
合计（80 层）			**2.589 M × 80 = 207.1 M**

按 FP16 计，**一个适配器约 0.386 GiB** ——不到基座的千分之三（约 0.296%）。这个比例是整个多 LoRA 方案成立的全部理由：个性化只要付千分之三的钱。

6.6.2 读取放大：按「活跃适配器数」而不是按 batch

decode 一步要读的字节数变成

$$Q_{\text{lora}} = \underbrace{W_{\text{base}}}_{\text{所有请求共享}} + \underbrace{n_{\text{adp}} \cdot W_{\text{lora}}}_{\text{每个活跃适配器一遍}} + B\,kL\ell, \tag{6.9}$$

其中 n_{adp} 是本步 batch 里不同适配器的个数。注意它是 n_{adp} 而不是 B：同一适配器的 50 个请求共享同一次读取。代入 $W_{\text{base}} = 140$ GB、$W_{\text{lora}} = 0.386$ GiB：

活跃适配器数 n_{adp}	1	8	32	100
额外字节	0.386 GiB	3.09 GiB	12.35 GiB	38.6 GiB
相对基座的读取放大	+0.296%	+2.37%	+9.47%	+29.6%

这个比例的含义很直接：**多 LoRA 的成本与「租户多样性」成正比，与「请求量」无关**。一个 8 个租户、每租户 100 并发、全部命中同一适配器的服务，纯权重读放大约 2.37%；一个 100 个租户、每租户 1 并发的服务，纯权重读放大约 29.6%——后者仍可共享基座的批处理，不能把租户多样性等同于失去基座复用。

▶ 多 LoRA 与「每租户一个模型」的对比

设 8 个租户，每租户各需一个微调版本，总并发需求 64。
方案一：8 个独立的 70B 模型。权重总量 $8 \times 140 = 1{,}120$ GB。需要 1,120/80 = 14 张卡只为放权重（还没算 KV），且每个模型独立批处理，batch 只有 8 ——每个模型都停在带宽受限区的最左端。
方案二：一个基座 + 8 个 LoRA。权重总量 $140 + 8 \times 0.4142 \approx 143.31$ GB，两张 80 GB 卡放得下（TP=2）；若用满一个 8 卡节点做 TP=8，则权重只占每卡 18 GB，剩下 60 GB 全给 KV。batch 可以开到 64，基座权重被 64 个请求共享。
比值是 1,120/143.31 ≈ 7.8×。但真正的差距不在显存，而在每 *token* 的字节数：方案一每个模型 batch 为 8，所以每输出 token 摊薄权重为 140/8 = 17.5 GB；方案二每步读约 143.31 GB 服务 64 个请求，即 143.31/64 ≈ 2.24 GB/token。纯权重摊薄比约 7.8【自算】，不包含 KV、适配器 kernel 与调度。这不是实测 7.8 倍吞吐；LoRA 与完整微调模型的质量也不能不评测就视为相同。

6.6.3 为什么它必须落到 kernel 层

式 (6.9) 在实现上有一个硬要求：一次 *kernel* 调用要同时处理用到不同适配器的请求。若逐请求串行地跑 LoRA 分支，就退化成了 8 个小模型各自为战，式 (6.9) 的等式仍成立，但 kernel 启动开销与 SM 空闲会把收益吃掉。

工业做法是 *SGMV*（segmented gather matrix-vector）：把 batch 按适配器分段，一次 kernel 里对每段用不同的 A/B 权重，段的边界由一张索引表给出。它与 MoE 的 grouped GEMM 是同一类技术——两者都

是「按索引把一批向量分派到不同的权重矩阵」。区别只在于 MoE 的分派依据是路由分数，多 LoRA 的依据是租户 id。

! 常见误解

「多 LoRA 不影响性能，因为适配器很小」——前半句的「小」是对的，后半句的「不影响」要看租户分布。$n_{\text{adp}} = 100$ 时纯权重读放大约 29.6%，在 decode 上就是 若其他项为零，对应吞吐约变为 1/1.296 = 77.2%，即下降 22.8%；有 KV 与计算项时需重新评估，这已经不是噪声。
还有一个常被忽略的冲突：**适配器会污染前缀缓存**。两个租户的 prompt 前缀逐 token 相同时，它们的 KV 不能共享——因为 LoRA 改变了权重，同一个 token 在不同适配器下的 KV 是不同的。所以缓存键必须包含适配器 id（第 7 章讲的链式哈希把 LoRA id 作为额外键，正是为此）。若这一点被忽略，缓存会串味，表现为「A 租户的回答里出现了 B 租户的语气」——这类 bug 极难定位，因为缓存命中率看起来完全正常。

6.4.4 适配器该放在哪一层介质

0.386 GiB 的适配器不能全放 HBM。100 个该配置适配器约 38.6 GiB，比基座的四分之一还多（约 +29.6%），而其中绝大多数长时间不用。于是它天然适合第 11 章的分级结构：

(1) **热适配器常驻 HBM** ——按近 5 分钟的使用频率选，通常只有几个。
(2) **温适配器放主机内存**——约 39 GiB 在 CPU DRAM 里毫无压力，换入换出走 PCIe；一次换入 0.386 GiB 在 PCIe 4.0 x16 上理想带宽项约 13 ms。
(3) **冷适配器放本地 NVMe 或对象存储**——换入时间从约 13 ms 增加到几百毫秒，但只在租户首次请求时付一次。

判据与第 11 章完全相同：换入的时间是否小于它省下的时间。一个租户若只发一个短请求，把它从 NVMe 换进来（数百毫秒）会增加冷启动延迟；基础模型不能通过“重算”替代缺失的 LoRA 权重。可按产品允许的 SLO 排队、预取或明确拒绝；静默改用基座会改变语义与质量。这给出了一个不那么直觉的产品约束：**长尾小租户在多 LoRA 架构里是有成本的，而且成本可以被精确算出**。

6.5 从场景到配置：一张决策表

把前面四节压缩成一张表。用法是自上而下过三个问题：模型能否装进 TP 域？互联域有多大？SLO 允许的 batch 有多大？三个答案通常能筛掉大多数组合，但仍需实测确认。

表 6.3 并行候选的容量与性能门槛。所有“可用”均须通过目标引擎和 SLO 验证。

场景	候选	需验证的约束
8B、模型可装单卡	TP1 多 DP 副本	单卡实际 ITL、上下文、负载均衡与每副本 batch
主例 70B、4k 输入/512 输出	TP8 或其他可容纳组合	表 1 的质量、TTFT、最大 ITL 与 E2E
70B、32k、32 并发压力例	TP8	账本 A 仅约 3.9GiB 教学余量，需测峰值与 graph 配置
70B、128k	每副本 TP8 加 DP	每请求 KV40GiB，TP8 后 5GiB/卡，不能无依据说仅 1—2 并发
671B、FP8 权重、EP	EP32/64 与 DP attention	专家复制、scale、每 rankKV、热点与 all-to-all 缓冲
405B、BF16 权重	TP8 乘 PP2	810GB/16 约 50.6GB（47.1GiB）/卡，另加 KV 与工作区
长序列 CP	CP 或 TP/CP 组合	独立 TP8 乘 CP8 通常需 64 卡；8 卡不能重复计算为两套独立 8 维度

表 6.3 背后的排序原则只有一条：先让单副本装得下，再让副本数尽量多。因为独立 DP 请求副本通常避免副本间 collective（副本内部仍可能 TP/PP），每多一个副本才可能增加近似线性吞吐；而 TP、PP、EP 都是「为了装得下」而付的税。工程上最常见的错误是把顺序反过来：先按「卡多」去堆 TP，结果通信税吃掉了全部收益。

• 三个可移植的判断

1. 看到一种并行，先问它切的是哪个维度，再问它每层每 token 要付多少字节。把节省时间与新增通信/调度时间放在同一关键路径上比较。
2. TP 需同时看延迟、带宽与消息大小；PP 需同时看流水调度、R, m、阶段平衡与模型容量。把这两个判据代入自己的数，比记结论可靠。
3. 显存账本要按「固定项 + 峰值项 + 碎片」三段写，并且给每一项标上计算式。只报一个总数的账本，在扩容或换模型时一定会翻车。

未解问题

1. **并行组合的代价模型不可移植**。式 (6.1) 里的 τ 随驱动版本、NCCL 算法选择、消息大小与拓扑跳数漂移，同一配置在两个集群上的最优解可能不同。今天选并行度仍靠实测扫描（profile 几十个组合），缺少一个能在部署前给出可信预测的解析模型——尤其是 all-to-all 在拥塞下的尾延迟。
2. **PD 分离后两种并行度如何共存**。prefill 吃算力、偏好大 TP 与大 chunk；decode 吃容量与带宽、偏好小 TP 与大 DP。第 7 章会说明 PD 分离的收益边界，但一个集群里两种并行度如何在请求波动时互相借卡、KV 又如何跨这两组卡迁移，目前只有各家自己的工程方案，没有公认的调度模型。
3. **长上下文 decode 的分片调度**。KV 分片后广播 query、各 rank 计算局部注意力，再用最大值/分母/输出的在线 softmax 统计合并，是可行的算法路径，不能说尚无“切 KV 但不搬 KV”方案。开放问题在动态长度、负载均衡、跨域通信与缓存迁移下怎样稳定实现低尾延迟。

第 7 章　调度与 KV cache 管理

本章结论

(1) 连续批处理及时移出完成请求、插入新请求，在输出长短不均且队列非空时减少空槽。收益取决于到达流、长度分布、prefill 干扰与容量，并非一个固定倍数。

(2) Chunked prefill 在 TTFT 与在跑请求的 ITL 之间重新分配阻塞。线性成本模型可筛选 chunk，实际分位数与最大 ITL 仍要用混合负载测量。

(3) 分页、前缀缓存、投机、抢占和 PD 分离分别改变容量、重复计算、串行步数和资源位置。比较它们时同时计质量、延迟、吞吐、额外状态和故障成本，以有效 goodput 验收。

7.1　continuous batching：省的是空转，不是显存

Orca 在 2022 年给出的诊断至今没有过时：request-level batching 把一批请求绑定在一起，批内任何一个请求结束都不能释放槽位，新到的请求必须等整批跑完【论文】[57]。它的代价有两个独立来源，工程上常被混为一谈。

第一个来源是尾部空闲。设批内有 B 个请求，输出长度为 $o_1, \dots, o_B$，单步 decode 耗时为 t_d。static batching 的批时长由 $\max_i o_i$ 决定，有效工作却只有 $\sum_i o_i$，于是槽位利用率

$$U_{\text{batch}} = \frac{\sum_i o_i}{B \max_i o_i}, \qquad \bar{U}_{\text{static}} = \frac{\mathbb{E}[\sum_i o_i]}{B\mathbb{E}[\max_i o_i]} = \frac{\mathbb{E}[o]}{\mathbb{E}[\max_i o_i]}. \tag{7.1}$$

$\bar{U}$ 是独立批次连续运行的长期“有效槽位时间/总槽位时间”，由更新奖励比值得到，**不等于** $\mathbb{E}[U_{\text{batch}}]$。恒定 t_d、忽略 prefill 且各批长度独立时，指数分布输出的 $\mathbb{E}[\max] = \mu H_B$，故长期效率为 $1/H_B$，$B = 8$ 约 36.8%、$B = 32$ 约 24.6%【自算，合成分布】。不同定义的利用率不能互相替代。

第二个来源是不可插入。若新请求独立观察正在运行的随机批次 T，平均残余时间为 $\mathbb{E}[T^2]/(2\mathbb{E}[T])$，不是普遍的 $\mathbb{E}[T]/2$；只有固定批长、到达相位均匀时才可除 2。这还没有计等待的其他批次。在 TTFT 500 ms 的 SLO 下，这不只是吞吐损失，而是服务不可用——这说明延迟预算必须检查残余批时间；轻载或固定短输出仍可能满足相应 SLO。

iteration-level scheduling（Orca 的命名）把调度粒度从请求改成一次前向：每一步重新决定哪些请求进批，结束的请求立即让出槽位，新请求下一步即可进入。在相同槽位时间口径、队列持续非空且可即时补位的理想条件下，空槽比例趋近零。实际 prefill、CPU 调度和显存不足会阻止补位，不能通过改分母把利用率定义成 1。

▶ static 与 continuous batching 的有效利用率

设输出长度分布为 $o \in \{64, 256, 1024\}$ token，概率 $(0.5, 0.3, 0.2)$，则 $\mathbb{E}[o] = 0.5 \times 64 + 0.3 \times 256 + 0.2 \times 1024 = 313.6$ token。取 $B = 8$，$\max_i o_i$ 的分布为 $\Pr[\max \le 64] = 0.5^8 = 0.0039$、$\Pr[\max \le 256] = 0.8^8 = 0.1678$、$\Pr[\max \le 1024] = 1$，于是

$$\mathbb{E}[\max] = 64(0.0039) + 256(0.1678 - 0.0039) + 1024(1 - 0.1678) = 894.4.$$

于是 $\bar{U}_{\text{static}} = 313.6/894.4 = 35.1\%$。continuous batching 在队列非空时 $\bar{U}_{\text{cont}} = 96\%$（教学假设，非本次引擎实测），同一 B 下吞吐比为 $96/35.1 = 2.7$ 倍。批时长 $= 894.4 \times 7$ ms $= 6.3$ s，独立到达观察的平均残余批时间约 3.46 s；连续批处理只有在槽位、token 预算与队列允许立即插入时，组批等待才可低至一个迭代。

表 7.1 把 B 扫了一遍。两个观察。第一，U_{static} 随 B 下降，所以「把 batch 开大」在 static 下几乎不改

表 7.1 不同 batch 上限下的槽位利用率。输出长度分布为 $p(64)=0.5, p(256)=0.3, p(1024)=0.2$，$\mathbb{E}[o]=313.6$；$\bar{U}_{\text{cont}}$ 假设为 96%，不是测量。

B	$\mathbb{E}[\max_i o_i]$	U_{static}	U_{cont}	吞吐比
2	484.4	64.7%	96%	1.5×
4	697.4	45.0%	96%	2.1×
8	894.4	35.1%	96%	2.7×
16	1002.4	31.3%	96%	3.1×
32	1023.4	30.6%	96%	3.1×

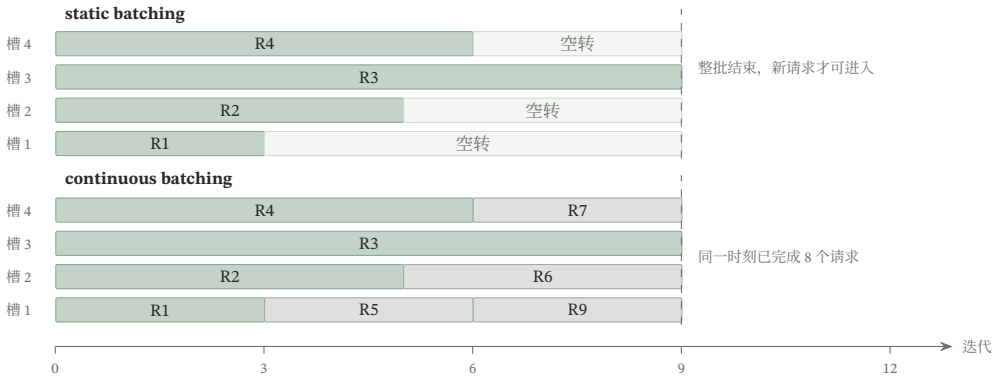


图 7.1 同一时间窗内 static 与 continuous batching 的槽位占用。static 的槽位在最早的请求结束时并不释放，灰色是陪跑的空转；continuous 在请求结束的下一步就把它交给新请求。到 $t=9$ 时两者占用相同的 4 个槽位，却分别完成 4 个和 8 个请求——这正是式 (7.1) 中 $\mathbb{E}[\max_i o_i]$ 与 $\mathbb{E}[o]$ 之差的几何含义。

善利用率，只是把尾巴拉得更长；continuous 的收益因此随 B 先增后饱和，到 $B \approx 16$ 时已逼近 3.1 倍，再加大只是继续拉长尾巴。第二，若输出长度有硬上限 $o_{\max}$（例如 `max_tokens` 截断），$U_{\text{static}} \to \mathbb{E}[o]/o_{\max}$ 而不再趋零：分布越重尾，static 越吃亏；分布越接近定长，两者越接近。

! 常见误解

「continuous batching 就是一次跑更多请求」是流传最广的误读。它没有提高 batch 上限，上限仍由显存容量与 SLO 决定；它提高的是给定上限下的占用率。反过来，如果负载的输出长度几乎相同（定长摘要、分类式生成），continuous 的收益接近零，此时仍可从固定开销、容量和带宽等处优化，不再有消除长短尾的同等收益。Orca 报告在 GPT-3 175B 上相对 FasterTransformer 取得 36.9 倍吞吐（同延迟水平）【论文】[57]，这个数字混合了 iteration-level scheduling、selective batching 与分布式切分；把它整体归因于「连续批处理」会高估这一项技术。

7.2 chunked prefill：把 ITL 尖刺摊成 TTFT

混合批（hybrid batch）里，一个长 prompt 的 prefill 会和已在跑的 decode 挤在同一步。设 prefill 长度为 P，每 token 的 prefill 计算时间为 α（70B、TP=8、有效算力每卡 500 TFLOPS、8 卡合计 4 PFLOPS 时 $\alpha \approx 35\,\mu$s【自算】），先作确定性、串行相加、固定 decode 负载且忽略 attention 增长的教学模型，单步时间与尖刺倍数为

$$t_{\text{step}} = t_d + P\alpha, \qquad \kappa \equiv \frac{t_{\text{step}}}{t_d} = 1 + \frac{P\alpha}{t_d}. \tag{7.2}$$

代入 $P=8192$、$\alpha=35\,\mu$s、$t_d=7$ ms，得 $\kappa \approx 42$：一次 8k prompt 的 prefill 让该步 ITL 从 7 ms 跳到 294 ms。Sarathi-Serve v3 图 9 报告，在 Llama-2-70B、4 张 A100、上下文 4096、decode batch 为 1 的测试点，将完整 prefill 混入后，单步 TBT 为纯 decode 批的 28.3 倍【论文】[58]。这是该模型、硬件与批形状下的测量；上面的 294 ms 则来自本节教学模型，二者分别记账。对最大 ITL SLO 为 50 ms 的教学服务，294

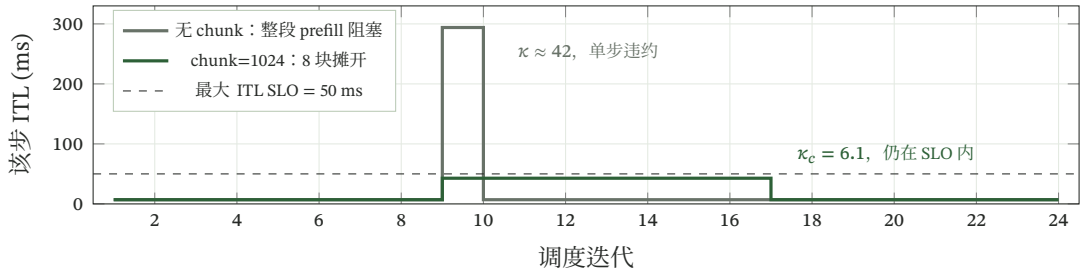


图 7.2 chunked prefill 对 ITL 尖刺的平滑效果。8k token 的 prefill 若整段执行，某一步 ITL 会从 7 ms 跳到 294 ms（$\kappa \approx 42$）；切成 8 个 1024 token 的块后每步 42.8 ms，全部落在 50 ms SLO 以下。代价是这 8 步内其他请求的 ITL 被抬高，且 TTFT 增加约 20%，即式 (7.4) 的那条双曲线。

ms 这一步违约；若约束的是平均 TPOT，不可只看单步判断。而它每来一个长请求就发生一次。后面的图 7.2 给出合成混合批时间线：一个 8k 的 prefill 独占了一步的绝大部分，四个 decode 请求在这一步里完全空等。

chunked prefill 把 P 切成 c 大小的块，每步只放一块，与 decode 混跑。于是 $\kappa_c = 1 + c\alpha/t_d$，而 TTFT 变成

$$\mathrm{TTFT} \;\approx\; \left\lceil \frac{P}{c} \right\rceil (t_d + c\alpha) \;\approx\; P\alpha + \frac{P}{c}\, t_d, \tag{7.3}$$

即「prefill 本身的算力时间」加上「被插入的 P/c 个 decode 步」。忽略整除/末块效应，以 $\mathrm{TTFT}_0 = P\alpha$ 作为不含排队的 prefill 基准，可得到近似关系：

$$\underbrace{\frac{\mathrm{TTFT} - \mathrm{TTFT}_0}{\mathrm{TTFT}_0}}_{\text{TTFT 相对膨胀}} \cdot \underbrace{\kappa_c - 1}_{\text{ITL 相对尖刺}} \;\approx\; 1. \tag{7.4}$$

这是 chunked prefill 的真正含义：它没有消除 *TTFT–ITL* 的冲突，只是把冲突从「不可控的 *42* 倍尖刺」变成一段可选的连续区间（图 7.2）。选 $c = 1024$ 时 $\kappa_c = 6.1$、ITL 为 42.8 ms（落在 50 ms SLO 内），TTFT 从 287 ms 涨到 342 ms，膨胀 19.5%【自算】，恰好等于 $1/(\kappa_c - 1) = 1/5.1$。若把 c 压到 256，ITL 降到 16.0 ms，TTFT 却要膨胀 78%（511 ms）；对 TTFT SLO 更敏感的应用，这个方向是错的。

这不是分位数恒等式。切块降低单步尖峰，但增加受影响的步数；p99 是否下降取决于所有请求与迭代的权重、到达率和并发，不能从一段时间线直接推出。

块大小因此不是「越小越平滑」，而是从 SLO 反解：$c^* = \lfloor (\mathrm{SLO}_{\mathrm{ITL}} - t_d)/\alpha \rfloor$，再向下取整到 tile 的整数倍。上例中 $c^* = \lfloor (50-7)/0.035 \rfloor = 1228$，取 1024；下界则来自效率——若目标实现 profiling 显示 512 token 附近效率足够，则可把 $512 \le c \le 1228$ 作为待测候选区间；512 不是跨 GPU 通用的饱和阈值。Sarathi-Serve 的做法更进一步：token budget（decode 数 + chunk）由一次 profiling 或模拟器（Vidur）在 TBT SLO 下取最大可行值，而不是拍一个常数【论文】[58]。同文第 4.3 节报告，某些测试配置中 chunk 从 256 变成 257 会使 prefill 时间增加 32%，用于说明 tile 量化效应；文中没有把该百分比绑定到一组可直接复算的矩阵形状与硬件参数，因此不能当作所有 kernel 的固定台阶。其第 5.4.1 节与图 14 则测试 Yi-34B、TP=2（两张 A100），prompt 为 2k、4k、8k，以不分块 prefill 为基线；chunk=512 时报告约 25%量级的额外开销。开销可能来自较低 GPU 利用率、重复访问前序 chunk 的 KV，以及 kernel 启动等固定成本；论文未给出各项份额，不能把这一测量主要归因于 KV 重读。引擎默认 token 预算随 release、硬件和 usage context 变化；不能把某文件中的 2048 直接作为所有部署默认值。按第 8 章固定版本核对配置，并用目标负载 profiling 反解预算。

piggyback 与 stall-free 是同一机制的两个排序。Sarathi 的 decode-maximal batching 用「一个 prefill chunk + 尽可能多的 decode」组批，让 decode 搭 prefill 的便车：prefill 本来就吃满算力，decode 的边际成本远低于 decode-only 批；LLaMA-13B 在 A6000 上 decode 吞吐最多提升 10 倍、端到端 1.33 倍【论文】[59]。但它的排序是 prefill 优先，decode 仍可能被挡。Sarathi-Serve 把顺序倒过来：先把所有 running decode 放进批，再用剩余预算放 prefill chunk（stall-free batching），在其调度假设内优先避免 decode 被 prefill 长期

排挤；同时批内计算量均匀，PP 场景下的流水线气泡也随之减少【论文】[58]。区别在目标函数：前者最大化平均吞吐，后者约束尾延迟。工程上选哪个，取决于 SLO 是写在 P50 还是 P99 上。

这些机制可在统一 token 预算下组批：每个请求记录已经计算的位置与待计算位置，调度器分配本步新增计算量。命中前缀的位置应从需要计算的数量中扣除；普通 decode 通常补一个位置，投机验证可能补多个位置。具体计数器、抢占与多模态路径以第 8 章固定 release 的源码为准，不能据一个过时行号推断所有功能均无独立分支。

7.3 投机解码：用额外工作减少串行目标模型步数

先由草稿器提出 γ 个候选，目标模型在一次验证前向中计算相关位置，从左到右接受有效前缀，首个拒绝处恢复采样；若全部接受，可额外产生 bonus token。省下的是多次目标模型串行调用，代价是草稿、验证计算、临时 KV、mask 和调度。权重可在多个候选位置间复用，但“读一遍权重”不等于验证整体免费。

7.3.1 接受长度与净收益

设每个位置在此前均接受条件下的接受概率均为 α，则一次验证期望输出 $E = 1 + \alpha + \cdots + \alpha^{\gamma}$。这是链式草稿的简化模型；真实接受率随位置、请求、draft 与 target 精度变化。以普通目标模型一步时间 t_0 归一化，验证时间为 vt_0，每个草稿 token 为 ct_0，且二者串行，则

$$E = \sum_{i=0}^{\gamma} \alpha^i = \frac{1-\alpha^{\gamma+1}}{1-\alpha}, \qquad S = \frac{E}{v + \gamma c}. \tag{7.5}$$

$\alpha = 1$ 时 $E = \gamma + 1$，用极限处理。只有额外假设 $v = 1$，才得到常见分母 $1 + \gamma c$。盈亏条件是 $E > v + \gamma c$，不是只看接受率。

c（假设 $v = 1$）	$\gamma = 3$	$\gamma = 5$	$\gamma = 8$
0.03	0.083	0.130	0.194
0.10	0.233	0.334	0.445
0.20	0.389	0.509	0.621

表中为 $S = 1$ 的 α 数值根【自算，理想验证时间】，不是某一 EAGLE 或小模型的实测成本。更长草稿的边际接受收益按 α^{γ} 衰减，不能无限延长。$\alpha = 0.7, \gamma = 5, c = 0.1$ 时，$E \approx 2.941$；$v = 1, 2, 3$ 分别得到 $S \approx 1.96, 1.18, 0.84$【自算】。同一接受率可以对应加速或减速，关键是实际验证时间。

7.3.2 为什么长上下文也不能保证不亏

设长历史 KV 在候选位置之间的有效 HBM 加载倍率为 r_{KV}，额外候选写入和中间态为 Q_{new}。一个条件流量模型是

$$Q_{\mathrm{spec}} \approx W + Br_{\mathrm{KV}}kL\ell + Q_{\mathrm{new}}, \qquad F_{\mathrm{spec}} \approx 2NB(\gamma+1) + F_{\mathrm{attn,spec}}. \tag{7.6}$$

理想 tile 能复用历史 KV 时 r_{KV} 可接近 1；若每个候选重读，可能趋近 $\gamma + 1$。L2、分块与 attention 实现决定实际值，新增候选 KV、树节点与通信并非零。忽略这些项并假设 $r_{\mathrm{KV}} = 1$ 时，才有代理强度

$$I_{\mathrm{spec,proxy}} = \frac{2N(\gamma+1)B}{W + BkL\ell}. \tag{7.7}$$

它的计算/读取交叉 batch 为 $B_{\mathrm{comp}} = W/[2N(\gamma+1)/I^* - kL\ell]$（分母为正时）。主例 70B、$\gamma = 5, \ell = 4096$ 时约 93【自算】，只说明此理想模型的计算余量。

! 读取主导不等于投机盈利

在上述代理中，$\ell \geq (\gamma+1)\ell^*$ 会使读取始终高于线性层计算项。这**不能**推出“长上下文、任意 batch 下投机都不亏”：草稿仍付费、接受率可能很低、KV 可能重读、临时状态挤占并发，且不同计算形状

的有效效率不同。是否盈利始终回到式 (7.5) 的实测 v, c, E。

批处理与投机可能竞争同一计算余量，也可能在某些低负载形状中互补。比较时应固定到达率与 SLO：如果首测加速后主动增加并发，需要报告新负载下的 goodput、最大 ITL、显存和质量，不能把单流加速当成可同比扩容。

7.3.3 拒绝采样保的是分布

令目标与草稿分布为 p, q。对草稿 $x \sim q$，以 $\min(1, p(x)/q(x))$ 接受；拒绝后从归一化 $\max(p-q, 0)$ 采样，恢复目标分布 p【论文】[60]。支持集、数值、logits processor、温度与采样规则须匹配算法假设。分布等价**不等于**相同随机种子逐位相同，因为消耗随机数的顺序不同。greedy 在相同目标数值结果、tie-break 及 logits 约束下可逐 token 匹配。如果 target 本身量化，保证针对的是该量化 target 的分布，不自动恢复 BF16 质量。

7.3.4 草稿来源与临时状态

独立 draft 需要额外权重与 KV；EAGLE 类复用特征但需训练并维护草稿头【论文】[61, 62, 63]；Medusa 类多头给出候选树，其采样与质量保证取决于具体接受方案【论文】[64]。n-gram/prompt-lookup 无需训练，但查找、索引与候选缓冲仍耗 CPU/显存，不能写成严格零成本。各实现的模块路径、支持模型、动态草稿能力以固定 release 为准。

树形验证增加候选覆盖，也增加节点 KV 与 mask 成本。若每层最多保留 w 个节点、深度 γ，总节点可为 $O(w\gamma)$；若每节点都分 w 叉、深度 γ，则是 $O(w^\gamma)$，不能把两种“树宽”混为一谈。拒绝节点的状态需回收。动态 mask 不必然禁止 CUDA Graph：固定 shape 与地址、在 buffer 中更新数值可能兼容捕获；动态节点数则可能需要 padding、bucket 或 fallback。调度器必须给验证 token 与临时 KV 预留预算，高负载下可以减少草稿或关闭投机。

▶ 主例的投机验收步骤

固定 70B、4096 输入、512 输出和表 1 的 SLO，先测普通 decode；再按 $\gamma = 1, 3, 5, 8$ 扫描，记录每请求的接受长度、draft 耗时、target 验证耗时、峰值 KV 和客户端输出时间。若多 token 一次返回，按块记录真实到达时间，平均 TPOT 变小不意味着所有块间隔小于 50 ms。本次未运行 GPU 投机实验；接受率表与 v 扫描仅为解析教学例，不报告引擎加速结果。

7.4 PagedAttention 与 prefix caching 的正交性

把 KV cache 的两类浪费分开看，是这一节的起点。第一类是空间浪费：请求长度事先未知，按最大长度预留会留下内部碎片，buddy allocator 会留下外部碎片，共享前缀的相同 KV 被重复持有—— vLLM 把这三类分别称为 reserved、internal fragmentation 与 external fragmentation【论文】[2]。第二类是计算浪费：相同的前缀 token 被不同请求重复做了一遍 prefill。

PagedAttention 只解决第一类。它把每个序列的 KV 切成固定大小的块（默认 16 token），逻辑块到物理块的映射放在一张表里，attention kernel 按块表 gather。块可以不连续，于是分配以块为单位、按需增长；引用计数让多个序列共享同一物理块，写时复制处理分叉。prefix caching 只解决第二类，它要回答三个问题：如何判定「相同前缀」、命中后如何挂接、容量不足时驱逐谁。

分页与前缀缓存可以组合，但索引方式并不只有一种。块哈希通常包含父前缀、当前 token 和影响模型状态的附加键；模型/适配器版本、位置编码、多模态内容与租户隔离约束必须一并处理。可共享物理块时，命中可增加引用计数而不复制数据；若跨设备或慢存储，仍可能付传输成本。第 8 章在固定版本中核对具体函数和路径，本文不沿用漂移行号。

• 分页与前缀复用是不同机制

PagedAttention 让逻辑连续序列引用不连续物理块；前缀缓存复用相同状态的历史计算。二者可以独立实现。vAttention 用虚拟内存映射提供连续虚拟地址，是另一条 KV 管理路径，仍涉及物理页，只是不要求同样的 paged-attention kernel 接口【论文】[65]。整块哈希、radix 匹配与实际 page size 共同决定命中粒度；SGLang 节点可分裂不代表所有配置都能复用任意单 token，需检查发布版本的页对齐约束。

缓存 p 个 token 占用 pkL 字节以及占用持续时间。以一次 HBM 读取时间除以 prefill 时间，得到的 0.003 次“回本复用数”漏掉了容量机会成本，不能证明复用一次必然划算。应比较预期节省的 prefill 时间/成本，与占用 KV 容量造成的准入损失、驱逐后的重算和必要传输成本。命中率、缓存驻留时间、活跃引用和隔离策略都应观测。

无损前缀缓存驱逐通常删除无活跃引用的缓存块；H2O 按注意力质量压缩活跃上下文会改变模型可见信息，属于有质量代价的 KV 选择【论文】[66]。它不是可以直接替换 LRU 的无损跨请求缓存策略。缓存键包含适配器等信息仍不自动消除侧信道或权限泄漏，服务端必须单独定义跨租户共享边界。

7.5 preemption：recompute 还是 swap

被抢占的请求已经付过一次 prefill。恢复它有两条路：重算（recompute），或把 KV 换出到 CPU 再换回（swap）。两条路的成本结构不同：

$$t_{\text{rec}}(n) \;=\; a_w + \alpha n + \beta n^2, \qquad t_{\text{swap}}(n) \;=\; \frac{2\,n\,kL}{B_{\text{pcie}}\,\eta} \;\equiv\; \gamma n. \tag{7.8}$$

$a_w = W_{\text{gpu}}/B_{\text{mem}}$ 是重算时必须把本卡权重读一遍的固定成本；αn 是线性部分的算力时间；βn^2 是 attention 的二次项，$\beta \approx 2Ld/(P_{\text{eff}} \cdot \text{TP})$（因果 attention，跳过上三角）；$\gamma$ 是整节点每 token 换出加换回的时间（$2kL/B_{\text{pcie}}\eta$ 已按节点聚合带宽归一），η 是块粒度导致的 PCIe 有效率。判据是 $t_{\text{rec}} < t_{\text{swap}}$，即

$$\beta n^2 + (\alpha - \gamma)\,n + a_w \;<\; 0 \quad\Longrightarrow\quad n^*_{\pm} = \frac{(\gamma-\alpha) \pm \sqrt{(\gamma-\alpha)^2 - 4\beta a_w}}{2\beta}. \tag{7.9}$$

当 $\gamma > \alpha$ 且判别式为正时，重算只在区间 (n^*_-, n^*_+) 内便宜，两端都该 swap。这个结论值得停下来看一眼：「重算 n 个 *token* 是 $O(n)$」是错的，*dense* 因果 *attention* 仍有 $O(n^2)$ 项，但线性层与固定项可能在目标区间主导，因为重算要对这 n 个 token 做一次 $n \times n$ 的 attention。

▶ recompute 与 swap 的交叉点

基线：70B、TP=8、FP16 KV。由第 2 章，整节点每 token 的 KV 为 kL = 320 KiB（8 卡均分，每卡 40 KiB）。PCIe 4.0 x16 单向理论 31.5 GB/s，但换出是按层切块的小块同步拷贝，块越小有效带宽越低：vLLM 实测 block=16 时 swap 开销极大，远高于重算【论文】[2]。取单卡有效 1.5 GB/s、节点聚合 $B_{\text{pcie}}\eta = 12$ GB/s【教学假设，非本次 PCIe 测量，不从 block 大小直接推带宽】。各常数为

$$a_w = \frac{17.5\ \text{GB}}{3.35\ \text{TB/s}} = 5.2\ \text{ms}, \quad \alpha = \frac{2 \times 70 \times 10^9}{8 \times 5 \times 10^{14}} = 35\ \mu\text{s}, \quad \beta = \frac{2 \times 80 \times 8192}{8 \times 5 \times 10^{14}} = 0.328\ \text{ns},$$

$$\gamma = \frac{2 \times 320\ \text{KiB}}{12\ \text{GB/s}} = 54.6\ \mu\text{s/token}.$$

代入式 (7.9)，两根约为 268 与 59,600 token【自算】。$n = 8192$ 时 $t_{\text{rec}} \approx 314$ ms、$t_{\text{swap}} \approx 447$ ms；$n = 32768$ 时约 1.50 s 对 1.79 s；$n = 131072$ 时约 10.22 s 对 7.16 s。因果 attention 的二次项是原稿全矩阵模型的一半，不能沿用约 30k 的交叉点。这里 t_{rec} 按固定项加计算项的教学代理相加，未建模计算/权重读取重叠，也未实测 kernel；生产交叉点应以恢复路径 profiling 校准。

在这个条件模型内，短端固定项使 swap 较便宜，中段重算较便宜，长端二次 attention 项使 swap 反

表 7.2 交叉点随配置的移动。γ 由 $2kL/(B_{\text{pcie}}\eta)$ 给出，n^* 由式 (7.9) 解出；「—」表示 $\gamma \le \alpha$，重算在任何长度都不便宜（swap 全面占优）。

配置	γ (μs/token)	n^*_-	n^*_+
假设有效 12 GB/s	54.6	0.27k	59.6k
假设有效 6 GB/s	109.2	0.07k	226.5k
假设有效 24 GB/s	27.3	—	—
FP8 KV，仍按 12 GB/s	27.3	—	—

超。这不是任意引擎策略：swap 可能没有实现、主机容量可能不足，恢复还受队列和传输并发影响。抢占不是客户端请求失败，不应仅因短上下文就丢弃请求并要求重试。

表 7.2 展示对有效传输带宽与 KV 精度的敏感性，不证明 block=8/16/256 必然分别对应 6/12/24 GB/s。压缩 KV 可降低 swap 字节，但解包与复制路径也可能变化，必须重新测。调度还应计资源机会成本：重算占计算资源，交换占 PCIe、内存与 copy engine；它们是否空闲由当前负载决定，不能称 decode 阶段两者总“免费”。抢占目标与策略随版本、优先级和路径变化，以第 8 章为准。应记录抢占率、重算 token、交换字节和已运行请求的最大 ITL，通过准入与容量预留降低无必要抢占，而非预设任何负载下最优抢占率严格为零。

7.6 PD 分离：收益公式与传输边界

DistServe 把 prefill 与 decode 放到不同 GPU 上，理由有两条【论文】[67]。其一是干扰：共置时一个 prefill 步会拖慢同批所有 decode，这正是式 (7.2) 的 κ；为守住 TPOT，系统只能压低 batch 或让 prefill 排队，两头都损失。其二是并行配置耦合：prefill 与 decode 对 TP、DP、chunk 及 batch 的最优组合可能不同；较大的 TP 可能帮助 prefill 延迟，也可能因通信而损吞吐，不能规定两个阶段的固定 TP 排序。共置必须选一个折中配置，于是至少一个阶段跑在非最优点上。分离还有一个 Splitwise 强调的好处：两个阶段可以用不同代际、不同功耗的机器，甚至把 decode 放在算力较弱但带宽/容量充足的卡上【论文】[68]。

代价是一项新的、可精确定价的成本：KV 必须从 prefill 节点搬到 decode 节点。

$$t_{\text{xfer}} = \frac{n_{\text{in}}\, kL}{B_{\text{link}}\,\eta}, \qquad \rho_{\text{link}} = \frac{\lambda\, n_{\text{in}}\, kL}{B_{\text{link}}}. \tag{7.10}$$

KV 传输可按层与 prefill、其他请求工作流水重叠；对本请求首个 decode 步，必须拥有所需层的状态，剩余暴露传输时间会增加 TTFT。若 prefill 端已发送首 token，传输也可能表现为首个 ITL 尖刺，需标明首 token 由哪端生成。设分离带来的 per-GPU goodput 增益为 G，基线请求速率为 λ_0，若新方案拟达到 $G\lambda_0$ 的整机吞吐，一个必要的网络条件为：

$$G\,\lambda_0\, n_{\text{in}}\, kL \;<\; B_{\text{link}}. \tag{7.11}$$

代入 70B、$n_{\text{in}} = 2048$、$kL = 320$ KiB：每个请求的 KV 为 0.671 GB。在 100 GbE（12.5 GB/s）上，$t_{\text{xfer}} = 53.7$ ms，占 500 ms TTFT 预算的 11%；DistServe 报告的增益 $G = 2.1$ 下，$\lambda_0 = 4$ req/s 只需 5.6 GB/s（45 Gbps，余量充足），而 $\lambda_0 = 10$ req/s 需要 14.1 GB/s，已经超过 100 GbE 的线速，即 $\rho_{\text{link}} > 1$【自算】。若 prompt 涨到 8k，每请求 2.68 GB，100 GbE 下理想 payload 上限约 4.66 req/s（若仍令 $G = 2.1$，对应基线 $\lambda_0 < 2.22$）；32k 时 100GbE 约 1.16 req/s，是否需要 400GbE 取决于目标 QPS、效率与延迟预算。要注意 100 GbE 是整机的预算，8 卡均分后每卡只有 1.6 GB/s，因此式 (7.11) 里的 λ_0 应当按整机请求率计算。*PD 分离的收益边界由 prompt 长度与网络带宽共同决定，还随模型 KV 结构、精度、并行布局和负载变化*——这是它和 TP/PP 最不一样的地方。

DistServe 在 13B 模型上的算例很能说明问题：共置时单卡 goodput 约 1.6 req/s；分离后 prefill 单卡 5.6 req/s、decode 单卡 10 req/s，配 2 卡 prefill + 1 卡 decode 可得整体 10 req/s，折合每卡 3.3 req/s，是共置的 2.1 倍；在多种模型与 SLO 下，最多可服务 7.4 倍请求或承受 12.6 倍更紧的 SLO（90% 达成率）【论文】[67]。Splitwise 从集群成本角度给出另一组数：吞吐高 1.4 倍且成本低 20%，或在同等成本与功耗预算

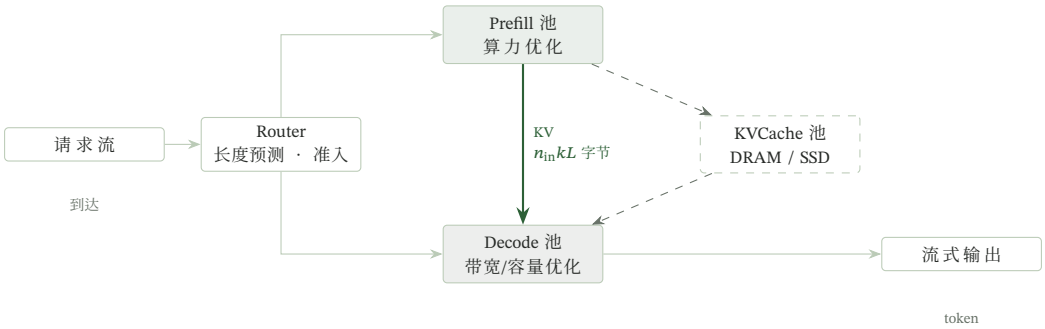


图 7.3 PD 分离的架构与它引入的新成本项。请求先经 prefill 池，KV 经网络送到 decode 池，再由它流式吐 token；KVCache 池是可选的第三层。式 (7.11) 只是必要网络条件；还需核对资源分配、排队、SLO 与故障恢复。

下吞吐高 2.35 倍【论文】[68]。两组数字口径不同，不能混用。

PD 分离的架构与它引入的新成本项见图 7.3：请求先经 prefill 池，KV 经网络送到 decode 池，于是原本在片内的 HBM→SM 搬移变成了跨节点的 HBM→NIC→ 网络 →NIC→HBM 两跳。

Mooncake 把这条边界又推进了一步：它不只分离两个集群，还把 KV cache 做成跨请求的分层池（GPU 显存、CPU DRAM、SSD），调度器以 KVCache 为中心，优先把请求路由到「前缀已经在那儿」的节点，于是「传输」的一部分变成了「命中缓存」【论文】[69]。它在过载场景下还引入基于预测的早拒绝：当排队已无法在 SLO 内消化时，直接拒绝比接受后违约更划算。其报告的数字是：模拟场景下吞吐最多提升 525%，真实负载下 Kimi 多处理 75% 的请求【论文】[69]。这两个数字口径不同，引用时必须区分。

7.7 多模态：先测编码器，再给视觉 token 记账

多模态请求通常经历媒体读取/解码与编码器、LLM prefill、LLM decode 三个阶段，它们的 batch、形状和可抢占粒度不同。编码器可能很小，也可能在高分辨率、多 crop、视频或音频负载下成为瓶颈；没有指定编码器层数、宽度、图像策略与硬件，不能声称“ViT 实测 3—5 ms”或“只占端到端 2%—5%”。

设媒体经处理后向 LLM 注入 n_{img} 个视觉 token。这个数应从实际 *processor* 与模型配置读取，而不是普遍用分辨率除 28。作为纯教学数值，假设 $n_{\text{img}} = 1369$、采用主例的 70B GQA KV 布局，则

$$\Delta M_{\text{KV}} = 1369 \times 327680 = 448{,}593{,}920 \text{ byte} \approx 427.8 \text{ MiB}. \tag{7.12}$$

这约 449 MB，而不是 449 MiB。线性层新增计算代理 $2Nn_{\text{img}} \approx 191.7$ TFLOP，另加视觉/文本之间的 attention；上述数据不是某个多模态模型的实测或通用图像 token 数。十张同等 token 数图像约 4.49 GB KV，若有池化、patch 合并、cross-attention 或不同 KV 结构，必须重算。

7.7.1 编码器缓存与 LLM 前缀缓存

编码器缓存复用媒体特征，省去对应编码器前向；LLM 前缀缓存复用已经建好的 K/V，需要相同模型、适配器、位置与全部可见历史。两种缓存可以叠加，但不能把前者收益叫作省 prefill。媒体哈希还应绑定预处理分辨率、裁剪、processor/encoder 版本，防止相同字节在不同配置下误命中。视觉 token 不一定固定在序列开头，视频分段与交错输入尤其如此；它们也不天然“永不驱逐”，生命周期由活跃引用、缓存策略与质量约束决定。

7.7.2 调度与观测

到达时可能需先读取媒体元数据才能知道 token 数；编码器可按图像、crop、帧或 chunk 调度，是否可切分取决于实现，不能统一当作不可分原子任务。应分别记录媒体预处理时间、encoder 队列/执行、视觉 token 数、prefill、KV 命中与 decode。是否具有独立 encoder 预算、能否跨步执行以及如何共享 GPU，以固定引擎版本为准。优化先根据阶段时间占比和质量选择：减少视觉 token 可能降低 LLM 成本，但也可能丢失细节；编码器优化在其主导时仍有价值，分级 KV 只扩容量不会自动省逐步读取。

多模态、工具调用与模型级联都表明一个请求的资源形状随生命周期变化。工具等待应与 GPU 时间

分开；思考链通常是输出 decode 的一部分，不能因“思考”就把它叫 prefill。拆成多阶段任务后还要处理状态所有权、取消、超时和重试，不能只优化正常路径。

7.8 SLO 感知调度：定义 goodput 的分母与约束

按式 (1.2)，TPOT 是单请求平均 ITL；最大 ITL、所有 token 间隔的 p99、所有请求 TPOT 的 p99 是三个不同统计量。主例使用最大 ITL50 ms；引用 DistServe 等论文的 TPOT 约束时保留其原定义，不与主例直接合并。

固定观测窗 T_{obs}，令 N_{ok} 为满足预先规定质量、TTFT、最大 ITL 与 E2E 的完成请求数，则请求 goodput 为 $N_{\text{ok}}/T_{\text{obs}}$。如果质量无法在线逐条判定，应给出独立质量门槛以及该固定配置的延迟 goodput，不能伪称逐条已验质量。吞吐另报完成请求/s、输入 token/s 与输出 token/s；SLO 达成率分母为所有到达请求，失败、拒绝、超时均独立计数。服务只统计成功子集会隐藏过载。

文献还常定义“达到至少 A 达成率时可支持的最大到达率”【论文】[67]。这是容量边界，与某个固定负载下 $\lambda A(\lambda)$ 的 goodput 数值不是同一个概念；在稳定且观测窗边界处理一致时后者才近似成立。

$$t_{\text{step,proxy}}(B) = \frac{W_{\text{gpu}} + B(kL/\text{TP})\ell}{B_{\text{mem}}} + t_{\text{fix}}. \tag{7.13}$$

这是同长 dense batch 的条件模型，不包含 prefill、通信和低效访问，不能将它当所有请求 TPOT 或最大 ITL 的保证。70B、TP8、$\ell = 8192$ 时理想 HBM 项约 $5.22 + 0.100B$ ms；显存容量也随 $B\ell$ 增长，但计算、NCCL、CUDA Graph 形状、CPU 调度均可能先绑定。“TPOT 几乎从不先绑定”的绝对结论没有服务实测支持。

TTFT 应按真实路径拆解：

$$\text{TTFT} = T_{\text{ingress}} + T_{\text{queue}} + T_{\text{prefill,exposed}} + T_{\text{first output}}. \tag{7.14}$$

排队分析若用 M/G/1，应以一致的请求服务单位测 S，并假设泊松到达、独立服务与稳态，$\mathbb{E}[W_q] = \lambda\mathbb{E}[S^2]/[2(1-\rho)]$。不能用“一个 decode 迭代 10 ms”当作整请求服务时间，就推出利用率 98%仍可达 TTFT。连续批处理是状态依赖服务，长尾与突发到达还需 trace 回放或真实开放环压测。

准入必须同时检查预计 KV、prefill 预算、在跑请求的剩余 SLO 与失败余量。容量模型通过只是必要条件；主例 76 并发容量界限不意味着 76 并发都满足 SLO。预计超出延迟预算时，排队、限流、早拒绝与明确降级都应按产品契约处理；抢占则需保留已输出状态和统计重算成本。

练习：平均 TPOT 为何会隐藏尖刺

构造 512 输出 token 的时间线：一次间隔 200 ms，其余 510 个间隔 20 ms。**提示：**平均 TPOT 约 20.35 ms，但最大 ITL 已违反 50 ms 契约。再给其中一个请求加 2.5 s 队列，检查 TTFT 与 E2E；goodput 应逐条判定，不能先平均请求再判定。

7.9 跨模型调度：级联与路由

前面所有调度都在一个模型内部做决策：哪个请求先跑、KV 放哪、要不要抢占。但生产系统里还有一个更高层的决策，它可能显著影响成本：这个请求该由哪个模型回答。

它同时涉及产品质量契约与基础设施调度，判据还包括准确性、风险和验证可靠性；实现可位于网关、编排器或引擎——它需要在同一个请求生命周期里协调两个不同模型的 KV、token 预算与 SLO。

7.9.1 级联的成本算术

设小模型单位成本为 c_1、大模型为 c_2，记 $r = c_1/c_2$（下表教学假设 0.05—0.2）。级联的做法是：先让小模型答，用一个判据决定是否接受；不接受则升级到大模型。设判据接受小模型结果的比例为 p（通过率，不等于真实正确率），则

$$\frac{c_{\text{casc}}}{c_2} \;=\; r + (1-p)\cdot 1 \;=\; r + 1 - p, \tag{7.15}$$

因为每个请求都付一次 c_1，其中 $1-p$ 的比例再付一次 c_2。这个式子极简，但它的含义不简单：成本对通过率的敏感度恒为 *1*，而 r 只提供一个很小的常数底。

	$p=0.5$	$p=0.7$	$p=0.85$	$p=0.95$
$r=0.05$	1.8×	2.9×	5.0×	10.0×
$r=0.10$	1.7×	2.5×	4.0×	6.7×
$r=0.20$	1.4×	2.0×	2.9×	4.0×

表里是大模型成本除以级联成本，即节省倍率，不是级联成本比例。读法：$p=0.85$、$r=0.1$ 时省 4 倍；$p=0.5$ 时只省 1.7 倍。级联的价值几乎完全由通过率决定，而通过率取决于负载、模型、阈值与可接受质量——这是它最容易被高估的地方。

▶ 级联的延迟是改善还是恶化

直觉上「小模型快，所以级联更快」。这不总对：升级的那部分请求要付两次首 token 时间。设 T_1 为小模型生成并完成接受判定所需总时间（含验证），T_2 为判为升级后大模型首 token 时间；不能只把小模型 TTFT 放进 T_1，则

$$\mathbb{E}[T] \;=\; p\,T_1 + (1-p)(T_1+T_2) \;=\; T_1 + (1-p)\,T_2.$$

代入 $T_1=0.3$ s、$T_2=1.2$ s：

通过率 p	0.70	0.85	0.95
$\mathbb{E}[T]$（s）	0.66	0.48	0.36
相对全用大模型	0.55×	0.40×	0.30×
但升级请求的 TTFT（s）	1.50	1.50	1.50

均值改善了，尾部恶化了。升级请求的 TTFT 是 $T_1+T_2=1.5$ s，比全用大模型的 1.2 s 还高 25%。于是：

- 在本例确定性两点分布且 $p>0.5$ 时，中位数为 T_1；真实分布的 P50 必须另算，均值下降不证明 SLO 达标；
- 若 SLO 写在 **P99**，级联会让尾部变差——除非升级的那部分请求本身占比很小（$1-p<1\%$）。

这条结论可以推广：串行升级通常让升级路径更长，但基线排队、并行验证或提前路由可能改变总体分位数。它是第 7 章「goodput 而不是 throughput」那条原则在多模型上的形态。

7.9.2 判据怎么设：三种可直接落地的方案

「是否接受小模型的回答」这个判断是级联的核心，但成本、质量验证和队列耦合同样决定收益。三种做法：

(1) **用置信度阈值**。取小模型输出分布的某个统计量（首 token 的熵、序列的平均 logprob、top-1 与 top-2 的 margin），低于阈值就升级。实现成本最低，但它校准得很差—— LLM 的 logprob 系统性偏高，同一个阈值在不同任务上的含义不同。

(2) **用可验证性**。若任务有客观校验（代码能否通过测试、JSON 能否解析、数学答案能否代入验证），就直接拿校验结果当判据。这类负载的通过率往往最高（因为「对不对」有客观标准），是级联收益最大的场景。

(3) **用一个小分类器**。训练一个轻量模型预测「小模型能否答对」，输入是 prompt 的特征（长度、领域、是否有代码块）。效果取决于训练与漂移，要额外维护一个模型，且它的漂移会静默地改变成本结构。

第三种方案有一个隐蔽的风险值得单说：分类器的准确率会随时间漂移，而漂移的效果是通过率变化

——成本随之变化，但延迟与质量指标可能看不出异常。于是账单先变，指标后变。这是级联在生产里最常见的失控方式。

7.9.3 路由：不是级联的退化，而是不同的问题

级联是串行的（先小后大），路由是在生成前直接选一个模型，并非必须并行执行多个模型。后者没有「付两次」的问题，但要求在没有答案的情况下判断难度——这是一个更难的预测问题，收益也更不稳定。

两者的适用边界很清楚：

	级联（Cascade）	路由（Routing）
决策时机	拿到小模型答案之后	拿到请求之后、生成之前
判据质量	高（可以看答案，甚至可以验证）	低（只能猜难度）
成本结构	$r + (1-p)$，对 p 极敏感	由误判率决定
延迟影响	升级请求尾部变差	误判到小模型则质量下降
典型场景	有客观校验的任务（代码、结构化输出、数学）	明显的负载分层（简单问答 vs 复杂推理）

工程上更常见的是两者叠加：先用路由把明显简单的请求分给小模型，剩下的走级联。但这会让成本模型变成三段，且每段的通过率都要单独测——这已经进入「需要专门团队维护」的复杂度，是否划算由维护成本、负载规模与质量收益决定。

! 常见误解

「级联能省 5 倍成本，所以我们上一套」——先问三个问题：

(a) 你的负载里，有客观校验的比例是多少？没有校验就只能靠置信度，而置信度阈值需要大量标注才能校准。

(b) 你的 SLO 写在哪个分位？写在 P99 上时级联会让尾部变差（见上面的算式）。

(c) 两个模型的 KV 与显存怎么共存？级联要求两个模型同时在线，于是显存占用是两个之和——省下的成本可能被「少开的 batch」吃掉。

第三条最容易被忽略：级联不是「把大模型换小模型」，而是「两个模型都留着」。在显存受限的部署里，这直接减少了可用于 KV 的容量，从而降低并发——省下的单位成本，可能被降低的利用率抵消。

这个 pitfall 指向一个更一般的判断：**所有跨模型的优化，都要按「总显存」而不是「单位计算成本」来评估**。因为推理的系统约束是显存容量与带宽，而多留一个模型意味着这两者都被分走一份。这也解释了为什么级联在大厂的生产系统里普遍存在（它们有足够大的池可以分区），而在单机自建部署里很少见（一张卡的显存不够放两个模型）。

未解问题

1. **前缀缓存的命中率无法在部署前预测**。命中率取决于租户之间的共享结构（系统提示、少样本模板、多轮对话），而 LRU 与 radix 树的驱逐都没有竞争比保证。多租户下还有一个没有答案的问题：缓存该按租户隔离还是全局复用？隔离损失命中率，复用则带来侧信道与计费归属的麻烦。
2. **PD 分离的增益 G 与到达率 λ 互相耦合**。分离改变 goodput，也就改变最优副本数与放置；扩容或重平衡又需要迁移 KV，而迁移本身要占带宽。DistServe 用离线模拟器枚举配置，Mooncake 用中心调度器在线决策，但两者都缺少一个能证明稳定性的闭环模型——目前更像是「仿真 + 启发式」。
3. **抢占的边际成本是调度器的内生变量**。recompute 之所以便宜，是因为 decode 批里有空闲算力；而「有多少空闲算力」又由调度器决定，于是成本估计与调度决策形成循环依赖。现有系统常用固定或按块大小切换的启发式绕开这个循环，但没有人给出联合优化 batch 与抢占的完整解。
4. **chunk 大小的最优值随负载漂移**。Sarathi-Serve 用离线 profiling 或 Vidur 选 token budget，负载结构一变就要重选。在线自适应地调整 chunk（同时保持 TTFT 与 ITL 双 SLO）目前只有零星工程实践，缺少公开的、可复现的方法与基准。

第 8 章　vLLM 与 SGLang：固定版本的源码比较

可复查的比较基线

本章只读分析两个发布快照，不声称它们是修订日的最新版本：vLLM v0.11.0，提交 `b8b302cde434df8c9289a2b465406b47ebab1c2d`；SGLang v0.5.3.post3，提交 `85ebeecf0694bdb94ace25319775d4b060baf655`。访问日期为 2026-10-22。项目的 `evidence/engine-sources/manifest.json` 保存选定源文件的 URL、字节数、SHA-256 和上游许可证 [70, 71]。下文路径均相对相应仓库；代码框是**删节教学伪代码**，不是可直接替换的完整实现。本机没有 NVIDIA GPU，未运行两种引擎的性能比较。

本章结论

(1) 两者都将请求进度、KV 所有权和执行结果组织成状态机。比较架构首先要找状态的权威位置，再看状态怎样跨进程或跨迭代同步；目录划分本身不能证明延迟优势。

(2) vLLM V1 用已计算 token 数与目标 token 数的差额统一表达 prefill、decode 和草稿验证，但仍有不同队列、编码器预算与 KV 约束。“统一进度”不等于“只有一条无分支路径”。

(3) vLLM 的块哈希与 SGLang 的基数树都支持前缀共享、引用保护和驱逐。树保留层级，块表便于索引；实际命中率、CPU 开销和 goodput 必须在相同工作负载下测量。

8.1　先读一次请求的状态，再读类名

第 7 章讨论的是算法问题，本章把问题映射到代码。贯穿算例的一次 4096/512 请求，进入引擎时首先成为请求状态：输入 token、采样配置、进度、优先级和缓存归属。调度器决定本步允许推进多少 token；KV 管理器保证这些 token 的位置存在；执行器把决定送往模型 worker；模型运行器准备张量并调用 kernel；结果再推进或终止状态机。图 8.1 显示这一闭环。

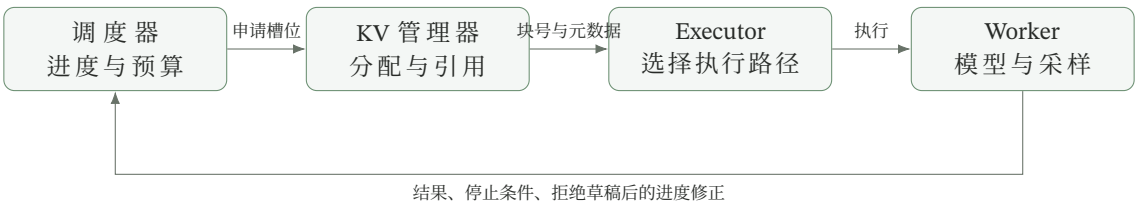


图 8.1　一次迭代的逻辑闭环。箭头表示依赖，不表示所有调用都跨进程或严格串行；具体实现允许异步队列。

表 8.1　固定版本中可直接定位的阅读入口。方法名比漂移的行号更适合后续复查。

问题	文件与符号
vLLM 主循环	`vllm/v1/engine/core.py`：`EngineCore.step`、`step_with_batch_queue`
调度与状态	`vllm/v1/core/sched/scheduler.py`：`Scheduler.schedule`、`_update_after_schedule`、`update_from_output`
块与哈希	`vllm/v1/core/kv_cache_manager.py`：`KVCacheManager`；`vllm/v1/core/block_pool.py`：`BlockPool`
SGLang 主循环	`python/sglang/srt/managers/scheduler.py`：`Scheduler.event_loop_normal`、`event_loop_overlap`
SGLang 缓存	`python/sglang/srt/mem_cache/radix_cache.py`：`RadixCache`、`TreeNode`、`RadixKey`
SGLang 策略	`python/sglang/srt/managers/schedule_policy.py`：`SchedulePolicy`、`PrefillAdder`

表 8.1 的 vLLM 入口将调度和模型执行分开。`EngineCore.step()` 先检查是否仍有请求，然后调用调

度、执行和结果更新。具体 worker 拓扑取决于 executor、TP/PP/DP 和启动参数；单卡执行不必然产生多进程 RPC。API server、EngineCore 与 worker 的分工是理解 V1 的起点 [72]，但“调度器不碰模型张量”不表示 CPU、进程通信或模型运行器没有开销。

SGLang 的 scheduler 进程将批次构造与 TP worker 联系起来，其前端语言和后端运行时是不同接口层。早期论文的程序级前缀复用动机 [73]，不能直接替代本章固定提交的运行时行为。两者都支持不止一种注意力模型和执行后端，本文主要跟踪**普通稠密自回归、完整注意力的路径**。多模态、混合状态模型、推测树和外部 KV 连接器会引入额外状态，不应从本章简化伪代码推断全部功能。

8.2 vLLM：统一 token 进度，并保留真实约束

`Scheduler.schedule()` 的开头说明调度器没有独立的 decoding phase 与 prefill phase。其目标进度 `num_tokens_with_spec` 包括输入、已生成输出和草稿 token。对请求 r，先计算

$$\Delta_r = n_{\text{target},r} + n_{\text{placeholder},r} - n_{\text{computed},r}, \qquad 0 \le q_r \le \Delta_r, \qquad \sum_r q_r \le T_{\text{budget}}. \tag{8.1}$$

这里 q_r 是本步计划计算的 token 数；还须受长 prefill 阈值、最大模型长度、可用 KV、编码器预算及特定功能条件约束。不能只把式 (8.1) 当成完整的可行域。基线版本先遍历 running，再在本步没有抢占时尝试 waiting；有多个循环与条件分支。

vLLM V1 调度骨架（删节教学伪代码）

```
for req in running:
    n = target(req) + placeholders(req) - computed(req)
    n = clip_by_token_budget_length_and_encoder(n)
    while not allocate_kv_slots(req, n):
        victim = select_victim(running, policy)
        free_kv(victim)
        victim.computed = 0
        move_to_waiting(victim)
        if victim is req:
            break
    record_if_scheduled(req, n)
if not preempted_this_step:
    admit_waiting_with_prefix_lookup()
output = build_scheduler_output()
advance_scheduled_progress(output)
return output
```

本步进度何时推进，是一处容易读错的细节。固定版本在 `schedule()` 尾部构造好输出后，调用 `_update_after_schedule()` 将 `num_computed_tokens` 增加本步调度数，并非等 GPU 结果回来后才增加。这样下一轮可以继续为尚未回报的请求准备工作。若投机验证拒绝部分草稿，`update_from_output()` 再减去拒绝数。计划进度与已提交结果的区别，是异步调度状态一致性的核心，不能把计数器名字直接当成物理完成事件。

抢占按请求执行。在该版本的这条路径中，空间不足时选择一个 running 请求，释放其块、将进度归零并放回 waiting。它不是“抢一块只重算一块”，而是以请求为调度单位恢复。缓存或外部 KV 命中可能减少重新进入时的计算，所以“进度归零”也不等于所有 token 必然完整重算。V0 的 CPU swap 机制不能直接套到 V1；连接器和未来版本同样需要另查。

统一进度允许混批，不能保证不干扰。chunked prefill 是本步推进数被截断，prefix cache 是起点前移，投机验证是目标差额增加；它们共享抽象但仍有各自代码。长 prefill token 预算过大仍会延长同批 decode 的一步时间，必须通过第 7 章的 TTFT/ITL 约束调节。Sarathi-Serve 的分块与连续批处理思想提供了相关背景 [58]，不等于任意参数设置均能同时提高两项指标。

8.3 分页 KV：哈希、引用与驱逐是三件事

`KVCacheManager` 管理请求到块的归属；`BlockPool` 管理物理块及其引用；`vllm/v1/core/kv_cache_utils.py` 中的 `hash_block_tokens` 生成前缀标识。对第 j 个完整块，可概括为

$$h_j = H(h_{j-1}, \text{tokens}_j, \text{extra}_j). \tag{8.2}$$

父哈希使同样的块内 token 在不同历史之后具有不同标识。额外键还涉及适配器、多模态输入和缓存隔离域；相同 token 字符串不足以证明 KV 兼容。要跨实例复用，还必须固定模型权重、tokenizer、位置编码、精度、布局与安全命名空间。

`CacheConfig` 在本基线的 `prefix_caching_hash_algo` 默认值为 `sha256`。但根哈希 `NONE_HASH` 由 `init_none_hash` 初始化：未设置 `PYTHONHASHSEED` 时会取随机字节，设置后才由种子导出。因此**选择 SHA-256 本身不能保证跨进程哈希一致**；可复现序列化、根种子和额外键也必须相容。密码学哈希降低碰撞风险，但不替代租户授权或错误配置检查。

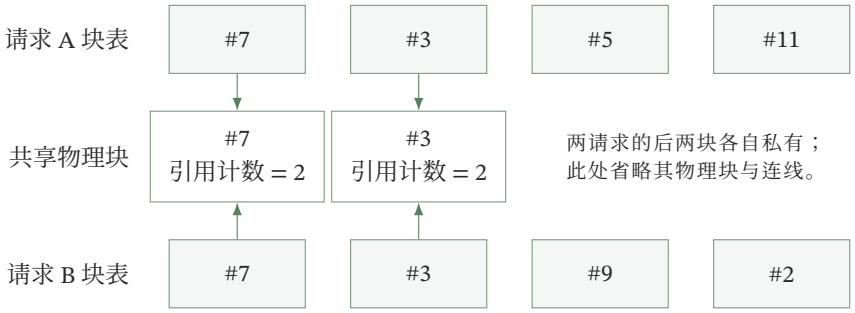


图 8.2 块表用逻辑顺序索引物理块。图只示意共享前两块的引用，不要求物理地址连续；块号不是内容哈希。

图 8.2 中，两请求共享前两块，后续块私有。`BlockPool.touch()` 对命中块增加引用，引用为零的空闲块重新被使用时从空闲队列移除。`free_blocks()` 减引用，只把降为零的块放回队列。**释放请求所有权并不总是立即擦除缓存内容**；下次重新分配物理块时才可能使其旧哈希失效。块表只追加是常见路径的优化，但混合注意力、抢占与新版本不能不加条件地概括成永不更新。

固定版本的 `BlockHashToBlockMap` 允许同一哈希对应多个块，因此“内容相同”不必意味着立即把所有物理副本合并。去重行为要同时考察查找、正在使用的块表与回收次序。按块队列回收不等于完全不理解前缀：哈希携带历史，回收顺序也会影响共享前缀存活。无法仅凭“树”或“哈希表”推导谁的稳态命中率一定更高。

设缓存粒度为 b_p，可复用长度为 L_c。最末不完整块未命中的余数 $L_c - b_p\lfloor L_c/b_p \rfloor$ 至多为 $b_p - 1$。只有余数在 $0, \dots, b_p - 1$ 上均匀时，平均损失才是

$$\mathbb{E}[w] = (b_p - 1)/2. \tag{8.3}$$

这个量是**前缀复用损失**；活动请求最后一页未填满的**分配碎片**是另一种账本。若为取得 logits 还需至少重算一个 token，则 $L_c \le L_{\text{prompt}} - 1$。以 $b_p = 16$、prompt=1000 为教学例，最多命中 992 token，仍有 8 token 处理；prompt=20 时最多命中 16。小前缀损失比例更大，但页大小还影响元数据与 kernel，不能据此说长前缀永远不受影响。`block_size` 的可选值和平台默认值应从启动日志确认，而不是假设所有设备都等于 16。

8.4 SGLang：基数树保留哪些结构

在固定版本中，`RadixKey` 保存 token 序列和 `extra_key`；`TreeNode` 保存 children、parent、key、KV 索引 value、访问时间与 `lock_ref`。`get_child_key` 在页大小为 1 时取首 token，否则取第一页的 token 元组，再把额外命名空间并入字典键。这保证不同隔离域即使 token 相同也不会自然落在同一分支。

查找与分裂。`match_prefix()` 先按页边界截取可匹配长度，再由 `_match_prefix_helper()` 遍历。该基线的 `_key_match_page_size1` 是逐 token 比较；分页版本按页切片比较。不能套用其它提交中的倍增搜

索描述。当命中结束在一条边中间，`_split_node()` 把边拆为公共段与剩余段，共享其对应 KV 索引，而不复制一次完整模型计算。

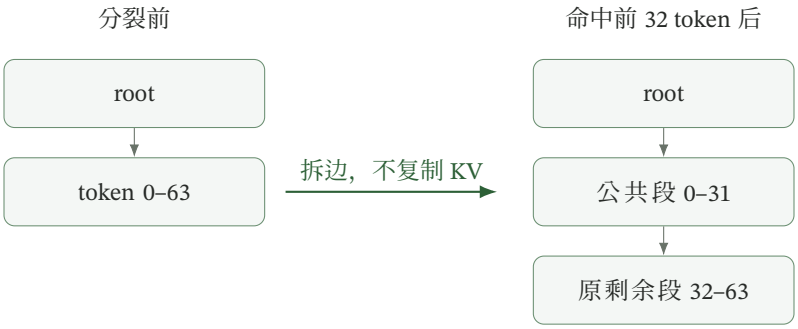


图 8.3 基数树的合法分裂。公共前缀先成为独立节点，原后缀成为其子节点；不能在同一父节点下画两条首 token 相同的边。

图 8.3 中的结构改变发生在查找或插入期间。`lock_ref` 的作用是**使用中的 KV 不被驱逐**：`inc_lock_ref` 沿祖先路径增加计数，首次引用时把相应长度从可驱逐量移到保护量；`dec_lock_ref` 做逆操作。它不是 Python 线程的互斥锁，不能用来证明并发修改安全。读取代码时应另外检查调度器所属进程和实际并发访问方式。

插入与驱逐。`insert()` 复用已有公共段并增加新边。`evict()` 收集可候选叶节点，按策略优先级建堆，跳过正被引用的节点，释放叶的 KV 后再把成为新叶的父节点放回候选集合。此基线的 `RadixCache` 构造器接受 LRU 或 LFU，不把后续 main 的更多策略写成本版能力。一个有 m 个候选叶的驱逐过程还包含建堆与堆操作；“树深次查找”也没有计入边内 token 比较。两种结构的复杂度必须带上 token 长度、块数、分支数和实际驱逐规模。

SGLang 前缀匹配（删节教学伪代码）

```
node = root
remaining = align_to_page(request_key)
while remaining and child_key(remaining) in node.children:
    child = node.children[child_key(remaining)]
    common = compare_segment(child.key, remaining)
    if common < len(child.key):
        node = split_node(child, common)
        append_kv_indices(node)
        break
    append_kv_indices(child)
    node = child
    remaining = remaining[common:]
return matched_indices, node
```

树的优势是保留路径与分支信息，能够表达共享前缀、引用祖先和叶回收。它的代价是比较、节点管理及结构变更；缓存被关闭、几乎无共享、分支很碎或后端页很大时，这一结构未必提高有效产出。页大小为 1 的 token 级匹配也不能外推到所有注意力后端。

8.5 重叠调度：依赖消除比 stream 数量更关键

SGLang 的 `event_loop_overlap()` 先收请求、构造本批并运行它，将结果放入队列后处理上一批的结果。`python/sglang/srt/managers/overlap_utils.py` 中的 `FutureMap` 用未来索引暂代尚未产出的 token，接口包括 `alloc_future_indices`、`store_to_map` 与 `resolve_future`。索引缓冲区和延迟结果让下一轮 CPU 工作不必立即读取尚未完成的采样值；实际同步仍必须沿调用链检查，不能由“异步”二字推断没有等待。

vLLM 在同一快照中既有普通 `step()`，也有 `step_with_batch_queue()` 和 `AsyncScheduler`。批队列路径使用 `execute_model(non_block=True)`，队列容量与运行配置相关；本章不把深度固定成 2。调度进

度先行、worker 持有的输入状态和返回结果之间，需要明确处理依赖与拒绝草稿后的修正。普通同步入口的存在不表示整个框架只能同步执行。

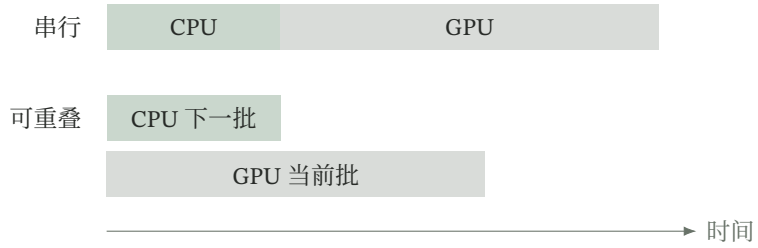


图 8.4 只有依赖允许时，CPU 下一批准备才能覆盖 GPU 当前批。图是教学时间线，不是两种引擎的本机 trace。

设两个阶段的稳态时间为 T_c 与 T_g，假设它们确实可流水重叠、资源互不争用且没有额外开销，则单步时间下界与相对串行的理想加速比为

$$T_{\text{step}} \geq \max(T_c, T_g), \qquad S_{\text{ideal}} = \frac{T_c + T_g}{\max(T_c, T_g)} \leq 2. \tag{8.4}$$

式 (8.4) 是**两阶段模型**的上界，首次填充和最后排空还会削弱有限请求的收益。进程通信、KV 访问、采样、输出序列化和资源争用会增加新开销；存在更多阶段时不能把 2 当成全系统通用上限。同进程或多进程都可以构造流水，拓扑影响开销和一致性，却不直接决定谁的 goodput 更高。

8.6 后端与扩展点：在正确层修改

该版 vLLM 的注意力抽象在 `vllm/attention/backends/abstract.py`，选择入口在 `vllm/attention/selector.py`；基类包括实现类、元数据类、构建器和 KV 形状接口。SGLang 的注册入口在 `python/sglang/srt/layers/attention/attention_registry.py`，模型运行入口在 `python/sglang/srt/model_executor/model_runner.py`。FlashInfer 等库提供的是注意力执行与元数据能力 [74]，不替代请求调度或准入控制。

改调度策略，首先查 token 预算、队列与资源准入；改 prefix 逻辑，查键语义、引用保护和驱逐；改 kernel，查数据类型、head 维度、页布局与工作空间；改采样，还须检查 worker、grammar 等上游约束和输出状态。一个扩展即使局部单测正确，也可能破坏异步结果归属、取消请求后的块回收，或 TP rank 间一致性。不要把“只增加一个类”当成完整的适配工作量。

8.7 把三类负载变成可证伪的比较

表 8.2 将原稿的架构排名改为实验问题。比较两框架时固定权重 revision、tokenizer、精度、硬件拓扑、采样和请求轨迹，并同时记录质量、错误率、TTFT、逐 token ITL 与 goodput。如果框架默认配置不同，先报告各自默认，再做预算、页大小或后端等消融，才能归因。

客户端指标的构造也必须读源码。固定版本的 vLLM `vllm/benchmarks/lib/endpoint_request_func.py` 中，completion 客户端把文本 SSE 事件间隔写入 `itl`；SGLang 的 `python/sglang/bench_serving.py` 在原生接口一次返回多个新 token 时，把该 chunk 间隔除以新 token 数，再重复写入数组。后者是均摊估计，两者都未独立观察 chunk 内每个 token 的到达时刻。因此即使打开 `--save-detailed` 或 `--output-details`，也不能直接把数组最大值当作本书严格的逐 token 最大 ITL。配套基准方案保存这些明细，并要求统一 token 时间与质量埋点后再判定 goodput。

练习与排查入口

给贯穿请求加一个租户 B，其输入 token 与租户 A 完全相同，但使用不同 LoRA。哪些键必须区分？提示：仅把租户标签加在路由层不足以防止缓存误共享，要沿 `extra_key` 或额外哈希构造一直检查到 KV 缓存。再把页从 16 改为 64，计算 1000 token prompt 在保留一个 token 重算时的最大完整页命中数；答案分别是 992 与 960，不能仅比较两次“请求命中”的布尔值。

表 8.2 三类负载的验证设计；所有优劣判断均待实际同条件测量。

负载	要固定和改变的量	应观察的证据
共享长前缀	相同 token 序列、租户命名空间；改变分支与缓存预算，分开冷/热缓存	复用 token 数、分配碎片、驱逐/重算量、TTFT；不只报“命中请求率”
高 QPS 短请求	相同开环到达轨迹；逐项切换 overlap、graph 与批预算	CPU 调度/输入准备时间、GPU 时间、队列、最大 ITL 与错误；不能从进程数推排名
超大 MoE	相同路由与 EP/TP 布局、网络、后端与有效精度	专家负载分布、集合通信、prefill 干扰、质量与 SLO；避免将 kernel 差异归因于缓存树

遇到 TTFT 突升，先分开排队、前缀匹配和 prefill；遇到 ITL 尖刺，查看同批 prefill、集合通信和结果队列；遇到块泄漏，跟踪请求取消、引用减计数与回收；遇到正确率变化，优先核对权重/适配器/精度、缓存键与采样，不把它直接归咎于调度策略。故障域和重试语义见第 15 章，统一基准和合成反例见第 19 章。

未解问题

1. 如何以重算成本、剩余工作和租户 SLO 选择抢占对象？固定版本的启发式不是不可能改进，但长度未知、恢复成本随缓存变化，且必须避免饥饿；需要完整 trace 与失败样本验证。
2. 如何统一跨框架缓存指标？“命中请求”“复用 token”“复用字节”和“省掉的计算”有不同分母。一个共享前缀在不同页大小、注意力布局与多级缓存中，命中率高也未必省时间。
3. 异步调度增加多少可观测与维护成本？理想重叠只描述时间下界，取消、错误、草稿拒绝和多 rank 故障要求额外状态；吞吐改善必须与一致性测试及运行复杂度一起评估。

Part II

AI 基础设施技术栈

第 9 章　AI 基础设施全景与成本模型

本章导读

前八章从算子与引擎解释一次请求如何执行。本章开始把视角移到资源：这些计算占用多少机器，机器在什么边界上计费，满足延迟要求的输出又有多少。阅读本章需要区分功率（kW）、能量（kWh）与吞吐（token/s）；三者的量纲不能互换。

9.1　沿一次请求划分责任

AI 基础设施把模型、加速器、网络与存储组织成可以持续交付的计算服务。模型结构影响基础设施约束，基础设施又影响模型的可部署形态，两者之间不存在完全封闭的边界。本书的五层请求视角依次是：L1 请求与调度、L2 引擎与显存、L3 算子与 kernel、L4 硬件与互联、L5 集群与成本。它们用于追踪同一次请求的因果关系，并不要求组织结构或产品恰好分成五层。

表 9.1　五层请求视角的观测入口。每一行需要一组指标，不能用单个数替代其他层的健康状况。

视角	主要问题	配套指标
L1 请求与调度	请求何时送到哪个副本	排队时间、拒绝率、公平性、缓存局部性
L2 引擎与显存	批次能否持续推进	TTFT、ITL 分布、KV 占用、抢占与命中收益
L3 算子与 kernel	时间花在计算还是访存	kernel 时间、有效带宽、数值误差
L4 硬件与互联	分片是否装得下、通信是否及时	可用 HBM、集合通信延迟、功率
L5 集群与成本	故障后可用容量及合格产出	就绪/恢复时间、goodput、单位成本

产业分析另用七层供应商栈：芯片、内存与互联、编译与 kernel、编排与调度、推理引擎、推理云与 token 服务、端侧与边缘运行时。它回答“谁提供什么能力”，并非第二条请求执行顺序。例如一家推理云供应商可能横跨 L1 路由、L2 引擎和 L5 容量采购，一家编译器团队则主要影响 L3 但也改变 L2 的形状限制。第 16 章沿用这七层做公司图谱。

例如，主例的输入 4096 token 先进入队列，prefill 生成 1.25 GiB 的 BF16 KV，随后逐步生成 512 token。引擎可以报告很高的总输出吞吐，但如果部分请求的最大 ITL 超过 50 ms，它们就不属于本书该 SLO 下的合格产出。硬件繁忙程度、引擎吞吐与服务 goodput 分别回答不同问题。训练和推理也不宜简单对立：大批量 prefill 与训练都可能受计算限制，小批量 decode 常受访存、发射或通信限制；训练还包含反向、优化器与 checkpoint，在线推理则承担排队、流式输出和中断后的用户体验约束。

9.2　先划清成本与产出的边界

设规划期为 H 小时，使用 N_g 张 GPU。自建场景可以把资本支出按使用寿命分摊，再加服务器、网络、机位、维护、人力与设施能源；租赁场景则从合同计费项出发。同一笔支出不能同时放进租金、折旧和配套三个科目。一个便于核对的小时模型是

$$C_h = N_g r_g + C_{h,\text{other}} + P_{\text{IT}}\,\pi\,e, \tag{9.1}$$

其中 r_g 的单位是美元/(GPU h)，$C_{h,\text{other}}$ 是其他未包含的小时费用，P_{IT} 为计量边界内的平均 IT 功率（kW），π 为 PUE，电价 e 为美元/kWh。因此最后一项单位为美元/h。若租金已包含全部电费，最后一项不再另加。若采用年折旧，只需把资本支出除以寿命年数和每年计费小时，并明确残值、融资与闲置如何处理。

定义 G_{out} 为观察窗口内成功且满足指定质量与 *SLO* 的请求所产生的输出 *token* 数除以窗口秒数。窗口

覆盖计费的闲置、重试和故障期间，单位是 token/s。则每百万合格输出 token 的成本为

$$c_{\text{out}} = \frac{C_h}{3600\, G_{\text{out}}}\, 10^6. \tag{9.2}$$

这是输出成本口径；prefill 的机器时间已进入分子，不能因此理解为输入免费。比较供应商价格时还要单列输入、缓存输入、输出、批处理折扣、地区、税费与有效日期。

! 常见误解

若 G_{out} 已从完整观察窗口统计，就不能再乘一次利用率或成功率。只有从满载容量 R 外推产出时，才可引入经过定义的折减因子，例如 $G_{\text{out}} = uR$；此时 u 需要包含哪些闲置或 SLO 损失必须写清。GPU 的忙碌百分比通常不是这个 u，也不能直接换算为可售 token。

▶ 主例的小时预算

本例为教学账本：4 张 80 GB GPU，每张内部资源分摊价 3 美元/(GPU h)，**假设该分摊价不含设施电费**；每卡平均 GPU 功率 0.35 kW，PUE=1.2，电价 0.10 美元/kWh。为便于与全书脚本核对，这一简化账本仅纳入 GPU 功率，CPU、内存、网卡和其他费用在此取零，不能据此作为完整服务器报价。

$$P_{\text{GPU}} = 4 \times 0.35 = 1.4 \text{ kW}, \qquad C_{h,\text{energy}} = 1.4 \times 1.2 \times 0.10 = 0.168 \text{ 美元/h}.$$

合计 $C_h = 12 + 0.168 = 12.168$ 美元/h。再**假设**该副本在输入 4096、输出 512 和本书 SLO 下的有效输出 goodput 为 480 token/s（并非本次 GPU 实测），则

$$c_{\text{out}} = \frac{12.168}{480 \times 3600} 10^6 \approx 7.04 \text{ 美元/百万合格输出 token}.$$

若 3 美元租金本已含电费，就使用 12 美元/h，得到约 6.94 美元/百万输出 token。512 token/请求时，480 token/s 对应合格完成率约 0.9375 请求/s；这不是系统可接受到达率的保证。

这笔账最容易复用的部分是变量关系，而非数值。保持小时成本不变，goodput 从 480 降到 240 token/s，单位成本升为约 14.08 美元；升到 720 则约 4.69 美元。若 GPU 分摊价从 3 改为 2 美元/(GPU h)，其余相同，成本降为约 4.73 美元。这些都是条件敏感性，不代表更高吞吐可以在相同质量、相同 SLO 下无代价取得。利用率、内核效率、模型压缩、采购、缓存命中和容量预留均可成为杠杆；哪个最重要由分项账本决定。

9.3 供电：PUE 的计量边界

PUE（Power Usage Effectiveness）定义为同一时间范围内设施总能量与 IT 设备能量之比 [75]：

$$\pi = \frac{E_{\text{facility}}}{E_{\text{IT}}}. \tag{9.3}$$

UPS、配电与制冷等损耗是否落入分子，由实际测量点和设施边界决定；它们不是一律被排除的项目。因此不能先拿已包含这些损耗的 PUE 乘 IT 用电，再除以同一供电链效率。若手上只有 GPU 芯片功率，必须先补服务器的 CPU、DRAM、网卡、风扇和电源损耗，才能形成与 PUE 分母一致的 IT 功率。TDP 是设计/配置上限，也不是这次负载的平均功率。

分项建模仍然有用。例如某一级转换效率为 η、输出负载为 P 时，该级输入为 P/η、损耗为 $P(1/\eta - 1)$；多个串联级的效率相乘。采用这种方法时，应从头至尾明确每一级的边界，最后只给尚未计入的设施开销加系数。真实预算还要区分平均 kWh 电量费用、峰值 kW 需量费用，以及机柜可用配电容量。电费占 TCO 的比例可能很小，供电容量却仍可能限制扩容，两者并不矛盾。

9.4 散热：热平衡能说明什么

稳态近似下，IT 消耗的电功率最终大多转化为需要排出的热。空气或冷却液的显热平衡为

$$\dot{Q} = \dot{m} c_p \Delta T = \rho \dot{V} c_p \Delta T, \tag{9.4}$$

$\dot{Q}$ 单位 W，$\dot{m}$ 为 kg/s，c_p 为 J/(kg K)，$\dot{V}$ 为 m^3/s。取空气密度 1.15 kg/m^3、比热 1005 J/(kg K)、流量 1.18 m^3/s、温升 12 K，得到约 16.4 kW。这是这些假设下的搬热能力，不是风冷机柜统一的物理上限。实际还受回流、压降、风机曲线、热点、海拔、进风条件和设施余量制约。

液冷利用液体较大的体积热容，把热量从芯片附近搬走；冷板覆盖率、CDU、泵与最终散热系统决定余热和辅助用电。是否采用液冷，需要同时检查机柜功率密度、供回液条件、改造成本、维护与泄漏处置能力，不能只凭一个 PUE 值决定。闭式液体回路也不等于设施零用水：若二次侧采用蒸发式冷却，设施仍消耗水；WUE 需要注明水量边界、位置和时间。

• 能耗与容量分别验算

主例的 0.35 kW/GPU 是平均教学假设，可用于小时能源估算；机柜配电、断路器和冷却容量必须按服务器额定值、同时率和设计余量另行核验。降低 PUE 可以降低设施能耗，但无法补足不足的 GPU HBM 或请求层容量。

选址还需把到用户的 RTT、数据驻留要求、网络可用性、电力供应、当地电价与碳强度一起考虑。可延迟批处理更容易做时间或地域迁移，在线请求则受 TTFT 和故障恢复约束。碳核算也要区分位置法/市场法、时间分辨率和设施用电边界，不能把某个地区的年均值直接当作一次请求的小时碳排。

9.5 把成本模型接到容量与定价

主例的 TTFT 预算为 2 s，最大 ITL 为 50 ms，端到端为 30 s。512 token 输出共有 511 个首 token 之后的间隔；若各间隔都取 50 ms 且 TTFT 取 2 s，末 token 到达不晚于 27.55 s，给协议尾部、其他计时误差和收尾留下约 2.45 s。这只检查三个阈值的相容性，并不证明系统能够达到它们。容量规划必须先测队列与执行分布，再统计满足全部阈值的请求；本书第 15 章进一步处理故障与重试。

定价还要覆盖研发、销售、空置风险和利润，不能把式 (9.2) 直接当成应收价。前缀缓存可减少 prefill 工作，却引入存储、索引与数据传输；更严格的隔离和容灾也可能增加分子、同时提高成功率。评价它们应比较同一负载、同一质量和 SLO 下的总成本与 goodput。

练习与提示

1. 若主例每台服务器另有 0.4 kW 非 GPU IT 功率，重算电费和单位成本。提示：先加到 IT 功率，再乘 PUE，勿再加入同一电源损耗。
2. 若 480 token/s 只来自忙碌的半小时，另外半小时完全空闲，能否仍用于式 (9.2)？提示：把产出和费用的观察窗口对齐。
3. 在固定到达负载上增大 batch 后，总吞吐上升但最大 ITL 违约增加，为什么成本未必下降？提示：分别计算总输出吞吐与合格输出 goodput。

来源与验证边界

PUE 采用 The Green Grid 的度量口径；硬件规格在第 10 章列出。本章成本、功率、风量、吞吐与 SLO 均为全书教学假设或解析复算，不是报价或本次服务器实测。来源快照与访问日期保存于 `evidence/infra-official/`，核对日期为 2026-10-22（北京时间）。

第 10 章　算力、存储带宽与硬件选型

本章导读

规格表给出设备的资源上限，部署需要回答的却是：特定模型、精度、上下文、并发和 SLO 下，哪个资源先耗尽？本章先建立可比较的规格口径，再用主例检查容量、带宽、算力与拓扑。本章所有用峰值计算的时间都是必要下界，不是本次设备实测。

10.1　读规格之前先固定口径

第一步是写全 SKU 与形态。同为 H100，SXM 和 NVL 的显存、功率及互联配置不同；整机的聚合指标也不能直接当作一张 GPU 可以独占的指标。第二步是固定精度与稀疏条件。NVIDIA H100 页面给出的 BF16 Tensor Core 峰值 1979 TFLOPS 带有“with sparsity”脚注；对应稠密峰值为 989.5 TFLOPS，FP8 稠密为 1979 TFLOPS[3]。2:4 结构化稀疏需要满足指定权重模式和支持的 kernel；MoE 只激活部分专家，不自动满足 2:4 条件。

第三步是区分单位和方向。GB 按 10^9 B、GiB 按 2^{30} B；1 TFLOP/s 是 10^{12} 次浮点运算/s，乘加在本书按 2 FLOP 计。网口 400 Gb/s 等于 50 GB/s。NVLink 标称聚合互联带宽、交换机全域总带宽、HBM 本地存储带宽是不同资源，不能互换。最后检查功率上限、时钟、ECC 和设备实际可用内存，它们都会影响可达性能与部署余量。

表 10.1　具体产品规格与可复算的机器平衡点（官方页面核对于 2026-10-22）。算力均为稠密 Tensor Core 峰值；其他量为厂商标称，不是应用实测。

项目	H100 SXM	H200 SXM
HBM 容量	80 GB	141 GB
HBM 带宽	3.35 TB/s	4.8 TB/s
BF16 稠密峰值	989.5 TFLOPS	989.5 TFLOPS
FP8 稠密峰值	1979 TFLOPS	1979 TFLOPS
NVLink 标称聚合	900 GB/s	900 GB/s
可配置最大 TDP	最高 700 W	最高 700 W
BF16 平衡点	295.4 FLOP/B	206.1 FLOP/B
FP8 平衡点	590.7 FLOP/B	412.3 FLOP/B

来源：NVIDIA H100/H200 产品规格 [3, 4]。页面中的 BF16/FP8 数值按稀疏脚注除以 2。GB 为十进制，TDP 是上限而非平均负载功率。

表 10.1 有意只放同口径的两个 SXM 产品。Blackwell 仍可用于学习读表：DGX B200 页面列出 8 个 GPU 合计 1440 GB 显存、64 TB/s HBM 带宽、最大系统功率约 14.3 kW [76]；这些是整机聚合口径。HGX 页同时列出 HGX B200 与 B300，FP4 还区分稀疏/稠密，不能把 B300 的一列或整机 PFLOPS 填进单 B200 表格 [77]。GB200 NVL72 则是另一种系统形态，不能与 DGX B200 的电源和内存配置混用。本书不把页面中的“最高倍数”性能广告作为跨模型结论。

SM 数和 L2 容量仍然有用。SM 数、时钟和指令吞吐共同影响并行度，L2 影响 tile、权重块和 KV 块的短时复用；但跨架构仅比较 SM 个数，或只凭 L2 装不下整模型就判定 L2 无用，都不成立。应结合实际访问流、命中率、占用率与 kernel 时间判断，也不设一个对所有负载通用的命中率阈值。

10.2　机器平衡点与工作负载强度必须配对

对同一精度、同一计算路径，roofline 上界为

$$P_{\text{attain}} \le \min(P_{\text{peak}}, B_{\text{HBM}} I), \qquad I^* = \frac{P_{\text{peak}}}{B_{\text{HBM}}}. \tag{10.1}$$

I 是所讨论 kernel 在同一层存储边界上的 FLOP/实际搬运字节。H100 SXM 的 BF16 平衡点约 295.4 FLOP/B；它不是整个服务的“最佳 batch”，也没有包含通信、发射、排队和内存分配开销。

以理想稠密线性层为例，batch 为 b、参数量为 P 时，计算约 $2Pb$，BF16 权重字节为 $2P$；若暂时忽略激活和 KV 访存，$I \approx b$ FLOP/B。因此 batch=1 和 batch=128 不是同一个 decode 工作点。真实 attention 还要读取随上下文增长的 KV，矩阵形状、GQA 复用、kernel 布局和额外中间张量又会改变有效强度。不能把单请求 decode 的 $I \approx 1$（BF16）或 2（FP8）推广为所有推理的固定值。

• 降低精度时要同时改两边

FP8/FP4 可能提高峰值算力，也可能减少权重字节，但 KV 精度不一定随权重精度改变；量化、反量化与 scale 读写还会加入工作。只把 I^* 提高而保持负载 I 不变，会错判低精度的收益。应重新计算字节和 FLOP，再以质量与端到端 SLO 验证。

10.3 主例：从显存预算到必要下界

本例使用 70B dense GQA 教学模型：$L = 80$、$d = 8192$、$h_q = 64$、$h_{kv} = 8$、$d_h = 128$。BF16 权重近似 140 GB，KV 为

$$k = 2Lh_{kv}d_h \times 2 = 327680 \text{ B/token} = 320 \text{ KiB/token}.$$

输入 4096、输出 512；按最大序列长度 4608 保守预留时，一条请求 KV 约 1.510 GB，即 1.40625 GiB。这里使用 70B 整数量级，不冒充某个 70.6B 模型权重文件的精确大小。

设一个副本用 TP=4、4 张 80 GB 卡，理想情况下 KV 头可均匀分成每 rank 两个，权重也均匀切分。按前言主例，将总显存的 90%纳入预算，再为每卡预留 8 GB 运行时、通信与 workspace。预算为 288 GB，扣除权重和 32 GB 运行空间后，留给 KV 为 116 GB，得到

$$B_{\text{cap}} \le \left\lfloor \frac{4 \times 80 \times 0.9 - 140 - 4 \times 8}{327680 \times 4608/10^9} \right\rfloor = 76. \tag{10.2}$$

这是理想均匀切分的容量上界，实际还受页对齐、复制、非均匀层、图内存和实现约束影响。若遗漏 32 GB 运行空间，会高估为 98 路，不能作为部署承诺。若改用 TP=2，总预算 144 GB，除权重只剩 4 GB，连 16 GB 运行空间也无法保留；只扣权重、不计运行空间的账面值为 2 条，不能据此判断部署可行。只装得下权重与可提供目标并发相差很大。

▶ 四卡主例的 decode 带宽下界

取 24 条活跃序列、用最大长度 4608 保守估计，假设一个 decode step 对每条序列生成一个 token，权重仅从 HBM 读一次，KV 也各读一次，忽略其他读写和通信：

$$D_{\text{step}} \approx 140 + 24 \times 1.50995 = 176.24 \text{ GB}.$$

若 4 张 H100 SXM 的 HBM 带宽可理想相加，下界为 $176.24/(4 \times 3350) \approx 13.15$ ms。若**另假设**只取得 70%聚合带宽，带宽项约 18.8 ms。它低于 50 ms 的 ITL 阈值，但尚未计计算、集合通信、调度和与 prefill 的干扰，所以不能据此宣布 SLO 达标。24 条序列若实测平均每 50 ms 各前进一步，才对应约 480 输出 token/s；这是一个假设工作点，而非从“最大 ITL 不超过 50 ms”推导出的必然吞吐。

prefill 还要查算力。输入 4096 的线性层近似为 $2Pn$，理想因果 attention 两次矩阵乘合计约 $2Ldn^2$，忽略 softmax 与其他小项，得到

$$F_{\text{pf}} \approx 2(70 \times 10^9)4096 + 2(80)(8192)4096^2 \approx 5.954 \times 10^{14} \text{ FLOP}. \tag{10.3}$$

四卡 BF16 峰值约 3.958 PFLOP/s，纯计算下界约 150 ms；若假设有效算力为峰值 30%，约 501 ms。这里把因果上三角运算按理想跳过计数；若 kernel 仍计算部分被 mask 位置，物理 FLOP 会不同。2 s 的 TTFT 预算还包含排队、tokenization、前缀查询、通信与首 token 采样，不能把这个 501 ms 当作 TTFT 测量。

对更大负载，可先列容量、带宽、计算三个必要条件：

$$N_g \geq \max\left(\frac{M_{\text{resident}}}{M_{\text{card,usable}}}, \frac{D_{\text{per second}}}{B_{\text{card,eff}}}, \frac{F_{\text{per second}}}{P_{\text{card,eff}}}\right).$$

但各分子应按实际副本布局计算，权重不能在多个副本间只记一份；从聚合下界上取整后，还要满足 TP/PP/EP 对齐与故障余量。Little 定律 $\bar{B} = \lambda \bar{T}$ 使用的是稳定系统的实际平均驻留时间，不能把 SLO 上限直接当成平均时间。因此放松 ITL 阈值并不强迫系统增加并发或显存。

10.4 HBM、封装与数据搬运能耗

HBM 通过宽接口、堆叠 DRAM 与先进封装提供高带宽。粗略地，若启用的总接口位宽为 w bit、每引脚数据率为 r bit/s，则 $B \approx wr/8$ B/s；实际还须说明有效接口、频率与协议效率。容量、总线宽度、堆叠高度和速率有关联，但不由一条“每叠固定容量”的规则决定所有代际产品。同一架构也可保留内存而屏蔽部分计算单元，不能从封装粒度推出“算力减半必砍带宽”。

搬运的能量可用一个有条件的模型理解：若实际活动数据率为 B_{active} B/s，每传一 bit 的增量能量为 e_{bit} J/bit，则

$$P_{\text{data}} \approx 8 B_{\text{active}} e_{\text{bit}}. \tag{10.4}$$

这是动态活动功率估计，另有刷新、泄漏、控制器等功率；不同芯片、工艺、距离与读写方向的 pJ/bit 不能直接拼成统一表格。用标称带宽乘一个假设能量也不能证明 HBM 占整卡 TDP 的真实百分比。实际能效应同时采集工作负载功率与合格产出，报告 J/合格 token，并说明是否包含 CPU、网卡和设施开销。

10.5 scale-up 域扩大了什么

NVLink/NVSwitch 域扩大会让更多 GPU 通过高速互联组成一个并行通信域；GB200 NVL72 官方页面描述 72 个 GPU、130 TB/s 聚合 GPU 通信带宽 [78]。它不是所有 GPU 共享一个无代价的 HBM 池，也不强制每个副本必须用 72 张卡。软件仍需要权重与 KV 分片、集合通信、流同步和资源放置。

对主例，四卡已满足理想容量预算，增加 TP 的主要收益要由更低 ITL 或更高 goodput 证明。TP 变大时每 rank 工作变小，但集合通信与同步会增加；$h_{kv} = 8$ 还意味着 TP 超过 8 时，某些实现要复制 KV 头或改用其他切分。拓扑更大不保证该模型线性加速。PP 跨域通常传激活，并非每层把全部 140 GB 权重搬过网络；EP 传路由后的 token 激活，MoE 也不在每步读取全部未激活专家权重。这些区别决定不能用“模型字节/互联带宽”直接推出端到端 TPOT。

• 容量、性能与故障域分开选择

先问每副本能否装下，再问目标并行度能否满足 SLO，最后问是否值得付出通信、调度碎片、供电和共同故障风险。超节点可以被切成多个副本；放置策略应使一次节点、交换机或供电故障的影响在可接受范围，而非只追求单次峰值。

10.6 其他加速器与软件栈的同一张验收表

TPU、Trainium、静态数据流处理器、晶圆级器件和专用 Transformer ASIC 代表不同的存储、互联和编译取舍。它们不能一概归结为“小模型才快”“结构一变就失效”或“通用性差所以必然便宜”。片上 SRAM 容量小可能需要多芯片放置权重，但这不意味着每步把全部权重通过网络搬一遍；若权重驻留在各芯片上，链路可能主要传激活。确定的芯片调度也不等于端到端 p99 与 p50 相同，网络、排队与服务软件仍会产生尾延迟。

对 CUDA 与非 CUDA 平台均可采用下列验收流程，而非只查是否存在后端目录：

1. 固定模型、dtype、KV 格式、长度分布、并发与质量基线；核对授权与可取得的具体硬件。
2. 对 attention、量化 GEMM、MoE 路由和归约分别验证数值与边界形状，记录 fallback。
3. 以真实消息大小测集合通信的延迟、有效带宽、拓扑和多作业干扰；不设跨架构统一达峰门槛。

4. 检查编译、冷启动、图捕获、监控、错误恢复与升级回退，不能把“模型可运行”当作“生产可用”。
5. 在相同 SLO 下比较 goodput、功率、完整费用与运维负担；厂商结果另列条件，不替代本次压测。

练习与提示

1. 主例把 KV 改成 1 B/元素而权重仍为 BF16，式 (10.2) 会怎样变化？提示：只将 KV 分母减半，另检查 scale 元数据和质量。
2. 某部署的总输出吞吐不变，实际平均 ITL 从 25 ms 变为 40 ms，Little 定律中的平均在飞量如何变化？这能否证明放松 SLO 导致容量上升？提示：区别观测到的服务时间与阈值。
3. 一个 kernel 加速 2 倍，但占端到端时间的比例只有 10%，忽略其他变化时总加速比是多少？提示：使用第 14 章的 Amdahl 分解，而不是把峰值倍数直接乘服务吞吐。

规格证据与局限

官方规格页面的原始 HTML、提取文本、最终 URL 和 SHA256 保存在 `evidence/infra-official/`，核对日期 2026-10-22（北京时间）。表内峰值和本章下界均未在本次环境上做 GPU 复现；本章明确移除无法核实的售价、IO 能量表、跨厂商通用阈值和广告性能外推。

第 11 章　内存、存储与 KV 缓存基础设施

本章导读

一次请求在 prefill 后留下 KV，decode 每一步读取它，结束后又可能把可复用前缀留给后续请求。本章从硬件暂存层次转向这种有语义、有所有权、有生命周期的数据，分别讨论容量、拉取与重算、兼容性和冷启动。读者应已了解第 7 章的分页与前缀缓存。

11.1　硬件层级与软件缓存不是同一个层级

HBM 是 GPU 的主要容量，L2 保存较近访问的 cache line，SMEM（共享内存）是 kernel 显式管理的片上暂存区。它们共同影响数据搬运，但 L2 和 SMEM 不能当作调度器可长期保留的前缀池。Hopper 调优指南列出 H100 的 50 MB L2 和每 SM 最高 228 KB 共享内存；后者与统一 L1/SMEM 配置有关，单 block 及每 SM 限制也不同 [79]。把每个 SM 的容量相加，并不会得到任意 kernel 都可访问的一块全片共享缓冲。

表 11.1　两种不同的层级。上半部由硬件/kernel 管理短时复用，下半部由系统管理请求状态；实际数据路径不一定逐层经过。

层次	管理单位与用途	必须核对的限制
L2、L1、寄存器	cache line、线程值、短时复用	访问模式、命中、寄存器溢出和并发占用
SMEM	tile、归约中间量、流水缓冲	每 SM/每 block 容量、bank 冲突、同步
GPU HBM	活跃 KV、权重、图与 workspace	实际可用容量、带宽、碎片和分片布局
主机 DRAM 缓存	暂不活跃的 KV 块、会话前缀	NUMA、锁页内存、PCIe/C2C 路径和并发搬运
本地 NVMe	冷前缀、权重分片	小块 IOPS、读放大、队列深度和写入寿命
远端内存/存储	跨实例复用、共享前缀与权重	网络、索引、完整性、租户与容量配额

FlashAttention 的分块展示了 SMEM 的作用。若 $d_h = 128$、每元素 2 B，一个 128×128 的 K tile 占 32 KiB，V tile 也为 32 KiB；仅 K/V 双缓冲就需 128 KiB。Q、softmax 统计量、同步和流水还要占用寄存器或 SMEM。因此 tile 变大虽然增加复用，也可能降低驻留 block 数或溢出；最佳 tile 取决于 kernel 的具体布局，并非“SMEM 越大越快”。L2 则可能截获多个 query head 对同一 KV 块的短时重复读取，收益应由访问流与 profile 确认。

主例 BF16 KV 为 320 KiB/token；4096 输入的 KV 为 1.25 GiB，4608 最大序列为 1.40625 GiB。这已经远大于单卡 L2，但并不意味着 L2 完全无效：kernel 只需在一个很小的时间窗里复用部分 tile。跨轮次、跨请求、跨实例的复用则依赖软件索引，而不是等待硬件 cache 自动完成。

11.2　KV 的热度、粒度与生命周期

先区分两种用途。**活跃 KV 卸载**要在后续 decode 计算前及时取回，会直接占用每步 ITL 预算；**前缀缓存**把已完成或暂不活跃的状态留到下一次请求，主要节省 prefill 并影响 TTFT。后一种命中划算，不代表前一种逐步卸载同样划算。

分页令分配、共享和传输以块为单位。若选 16 token/块，主例一块 KV 总计 5 MiB；4096 输入有 256 块。末块内部碎片小于 16 token，但还要计块表、引用计数、哈希与未被驱逐的缓存状态。不同引擎、attention 后端与硬件可使用不同块大小，不能把 16 写成所有版本的必然默认。细块有利于部分命中和写时复制，粗块通常减少元数据与启动开销。

系统至少需要可观察的状态转换：不存在、写入中、可读、读取中、可驱逐、已失效。消费者不能读取未完成的块，驱逐不能释放仍被 kernel 或 DMA 引用的内存；异步写回还需决定失败时丢弃缓存重算，还是保留本地副本重试。权重和 KV 不必采用相同的持久化策略：权重是可验证的模型资产，KV 通常可

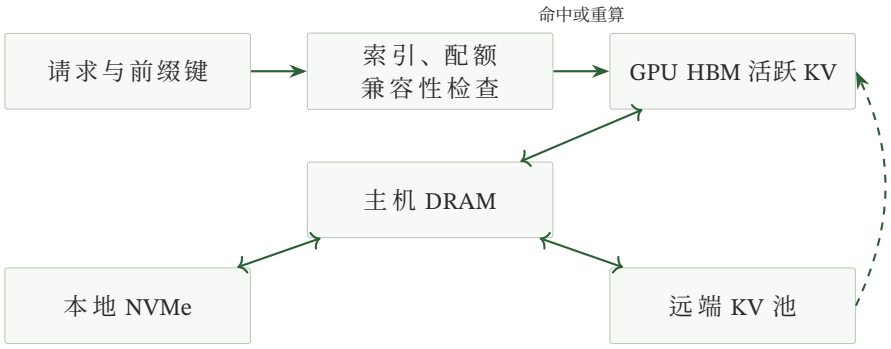


图 11.1 分层 KV 的逻辑关系。虚线示例为后端支持时的直达 GPU 路径；远端不必先经本地 NVMe。索引命中只有在兼容性检查、传输与状态发布完成后才成为可供计算读取的命中。

重算，但重算必须有时间和容量预算。

11.3 完整前缀：拉取还是重算

令参数量为 P，前缀长度为 n，每 token KV 为 k B，有效 prefill 算力为 F FLOP/s。用 $\beta = 2Ld$ 近似理想因果 attention 两次矩阵乘的二次系数，则

$$T_{\text{rec}}(n) \approx \frac{2Pn + \beta n^2}{F}, \qquad T_{\text{fetch}}(n) = t_0 + \frac{kn}{B_{\text{eff}}} + t_{\text{extra}}. \tag{11.1}$$

t_0 包含索引和启动，t_{extra} 包含尚未计入的排队、校验、布局转换与同步；B_{eff} 是整条关键路径实得带宽，可受 NIC、PCIe、NUMA 或存储中最慢处限制。若并行流水线重叠了部分开销，需画依赖图测关键路径，不能机械相加所有阶段。

若暂把额外时间合并为 t_f，完整前缀命中比重算快的条件为

$$B_{\text{eff}} > B^{*}_{\text{whole}}(n) = \frac{kn}{T_{\text{rec}}(n) - t_f}, \qquad T_{\text{rec}}(n) > t_f. \tag{11.2}$$

忽略固定开销时，这个整段平均门槛为 $kF/(2P + \beta n)$。若比较再多缓存一个 token 的边际收益，重算的导数则是 $(2P + 2\beta n)/F$，相应门槛为 $kF/(2P + 2\beta n)$。二者不能混用，固定开销对短前缀尤其重要。

▶ 主例 4096 输入的拉取与重算

取第 10 章的四卡 BF16 有效算力假设 $F = 4 \times 989.5 \times 10^{12} \times 0.30 = 1.1874 \times 10^{15}$ FLOP/s，$\beta = 2 \times 80 \times 8192 = 1310720$。解析重算时间约 0.501 s，KV 大小为 1.342 GB。再假设固定开销 2 ms、额外布局/排队开销暂取零：

路径假设	实得带宽	拉取时间
25 Gb/s 线速的 80%	2.5 GB/s	0.539 s
100 Gb/s 线速的 80%	10 GB/s	0.136 s
400 Gb/s 线速的 80%	40 GB/s	0.0356 s

完整前缀门槛约 $1.342/(0.501 - 0.002) = 2.69$ GB/s，即 21.6 Gb/s。这不是通用门槛：更快的本地 prefill、更忙的共享网络、更小的前缀或更大的固定开销都会改变决策。即使 0.136 s 快于重算，也仍要检查剩余 TTFT 预算与发送端负载。

独立敏感性例：将前缀改为 32768 token（不是主例默认输入），KV 为 10 GiB；在同一四卡有效算力模型下，重算约 5.05 s，100 Gb/s 假设路径拉取约 1.076 s。若换为八卡并假设有效算力翻倍，重算约 2.53 s，KV 传输字节却不变。所以“缓存是否值得”与本地可用于重算的资源紧密相关，不能只按介质名称排序。此外，只有前缀被复用的部分省计算；新后缀仍需计算且可读取已有 KV。

若一次写入成本 T_w、一次读取成本 T_r、替代重算时间 T_c，复用 h 次的时间净收益为

$$h(T_c - T_r) - T_w, \qquad h > \frac{T_w}{T_c - T_r} \quad (T_c > T_r). \tag{11.3}$$

这是时间/资源的简化模型，并未计储存容量机会成本；若缓存挤走其他更有价值的块，还应比较单位存储字节节省的 GPU 工作。命中次数不是命中率，token 命中率也不同于请求命中率；监控应同时报告命中字节、节省 prefill token、搬运延迟与最终 TTFT 收益。

11.4 兼容性、正确性与租户边界

KV 是模型、输入与位置相关状态，内容寻址并不能自动保证语义兼容。共享键至少需要考虑模型权重版本、LoRA/adapter、token 序列、位置与 RoPE 配置、KV dtype 及 scale、attention 变体、层/头切分与物理布局。不同格式可设计显式转换，但不能未经检查直接读取。多租户场景还应考虑隔离 namespace 或 salt，避免未经授权的前缀复用与命中侧信道。

本书固定的 vLLM v0.11.0 块缓存实现可从 `vllm/v1/core/kv_cache_utils.py` 与 `vllm/v1/core/block_pool.py` 追踪哈希、块与引用；SGLang v0.5.3.post3 可从 `python/sglang/srt/mem_cache/radix_cache.py` 追踪前缀树。这些只读快照与完整 commit 见第 8 章和 `evidence/engine-sources/`。这里不把较新分层后端的阈值、布局或 API 倒推为这两个固定版本的行为。

表 11.2 几种 KV 系统分别把哪类责任显式化。功能与后端随版本变化，表中不比较跨负载的最高加速倍数。

系统	主要抽象	部署时还要核对
Mooncake	集群共享 KV 资源与请求调度，连接 prefill/decode 及存储层 [69]	放置、可用带宽、元数据与故障恢复
SGLang/HiCache	从 radix 前缀索引延伸到主机/后端存储 [73, 80]	采用的 release、写回策略、预取取消与兼容格式
vLLM	引擎内块缓存；外部分层或传输经特定 connector 集成	固定版本支持的 connector、布局与生命周期
LMCache	引擎外可复用 KV 存储与搬运接口 [81]	引擎适配版本、后端持久性、索引与租户配额

缓存内容往往可以重算，因此缓存服务故障可退化为 miss；但若大批请求同时退化，prefill 容量可能瞬间被压满。需要限制重算并发、保留降级余量，并验证断网、部分写入、超时取消、旧模型版本和块驱逐竞态。把所有失败都“重试一下”会将局部存储故障放大成全服务排队。

11.5 权重加载：冷启动的字节预算

主例一个四卡副本共有 140 GB BF16 权重；理想预分片每 rank 读取 35 GB。若同一节点只有一条共享 10 GB/s 实得存储路径，读取全副本权重至少 14 s，不能因为有四个 rank 就再除以 4。若每 rank 独立读完整文件，读取放大会更大。加载还包含反序列化、dtype/布局变换、H2D、通信初始化、图捕获和预热，这些阶段哪些能重叠，必须从启动时间线确认。

扩到 R 个副本，存储出口至少提供 $140R$ GB 的独立读取量；本地权重缓存或 P2P 分发可降低源端重复流量，但 P2P 仍占用节点出口与网络。树广播的启动、分块与流水线决定完成时间，不能用“树深 × 整文件传输”当作所有实现的精确耗时。本地 NVMe 缓存可加快复启，已驻留内存或已加载的预热进程则是更强的预热状态，费用也不同。

权重文件需要校验 hash、模型版本和许可，预分片格式还要绑定并行配置。缩小权重精度能够减少载入字节，但若目标运行时要在加载时量化或重排，冷启动未必同比缩短。这一段的扩容滞后将在第 13 章接回服务容量。

11.6 与 checkpoint 共用基础设施时

本书聚焦推理，checkpoint 在此作为共享存储干扰源。70B 模型若保存 BF16 参数 2 B、FP32 主参数 4 B、Adam 一二阶矩各 4 B，则约 14 B/参数、980 GB；这份账不含梯度，也未含调度器、随机数状态、分片

元数据或冗余副本。不同训练策略可不保存主参数或采用不同优化器，不能把 14 B/参数当作固定格式。

大块 checkpoint 写入可能与小块 KV 读取竞争磁盘、网络和元数据服务，影响程度取决于分片数、队列、服务端调度与带宽余量。独立 bucket 只是逻辑命名空间，不保证独立磁盘或网络。可选手段包括物理资源隔离、带宽/IOPS 配额、先写本地再异步上传、错峰以及优先级队列；应实际测量并发 checkpoint 时 KV 命中延迟和 TTFT，而非只看平均磁盘吞吐。

练习与提示

1. 对主例，若一次远端读取另需 80 ms 排队，100 Gb/s 路径仍优于解析重算吗？若只剩 150 ms TTFT 预算又如何？提示：收益和 SLO 可行性是两个条件。
2. 两个请求 token 序列相同，一个使用不同 LoRA。是否可以共享 KV？提示：比较 KV 的生成函数，而不是只比较 token 哈希。
3. 一块前缀写入 40 ms、读取 20 ms、重算 50 ms，最少复用几次才能在该时间模型回本？提示：严格区分达到持平与取得正收益，并说明容量机会成本尚未计入。

证据范围

内存层级规格以 Hopper 官方调优指南核对，解析缓存算例未用 GPU 实测。固定引擎源码只证明对应 release 中的数据结构；Mooncake、LMCache 等论文用于说明架构抽象，其论文负载上的最高性能不外推为本书主例。官方资料核对日期为 2026-10-22（北京时间）。

第 12 章　网络、拓扑与集合通信

本章导读

第 6 章用并行策略切分计算，本章检查这些分片之间的数据能否及时到达。网络时间由消息大小、同步次数、拓扑、共享路径与拥塞共同决定。训练也有小消息，prefill/decode 分离也有大消息；不能只凭“训练网”或“推理网”的名称选型。

12.1　先画通信图，再选介质

一个请求可能包含四种不同流量：TP 层间的激活归约，PP 阶段边界的激活发送，MoE 的 token 分发与回收，以及 prefill/decode 或缓存系统之间的 KV 传输。训练还加入梯度同步和 checkpoint。它们可以经过同一根链路，关键路径却不同。例如 TP 小消息在每层重复，单次启动延迟会累积；大块 KV 搬运则可能受 NIC、PCIe、存储或目标 HBM 入口限制。两种情况都要验带宽，也都可能被尾延迟影响。

表 12.1　通信原语与请求语义。算法字节是理想模型，验收仍须在实际拓扑与并发下测量。

原语/流量	常见用途	验收时关注
AllReduce	TP 激活归约、DP 梯度同步	消息大小扫描、同步开销、rank 最大时长
AllGather/ReduceScatter	分片张量聚合与归约	分片方向、buffer、与计算重叠
Send/Recv	PP 激活、KV 片段传输	双端可达、背压、生命周期与超时
All-to-all	MoE dispatch/combine	专家热点、发送/接收最大值、扇入与分层
批量存储/网络 IO	前缀缓存、权重分发	共享出口、窗口、IOPS 与 TTFT 余量

NVLink/NVSwitch 通常用于高速 GPU 域内互联，PCIe 连接 GPU、CPU 与 NIC，也可以承载 GPU P2P 和集合通信；能否直达取决于主板、交换芯片、IOMMU/ACS 与软件路径。InfiniBand 和 RoCE 提供 RDMA 能力，跨机通信的实际性能仍取决于 NIC、交换机、布线、拥塞控制和运维配置。IB 的链路级信用机制并不消除端到端拥塞，RoCE 也不由一个“开 PFC”的开关保证服务质量。不能从协议名推出千卡成本分界或“买来就能用”。

标称 400 Gb/s 端口单向线速为 50 GB/s，尚未扣编码、包头和协议开销。GPU 到 NIC 若共享 PCIe 上行、NUMA 远端内存或多设备链路，实得数据率可能更低。同样，NVLink 端点双向聚合、某次单向传输、交换机全域聚合、HBM 本地读写是不同指标；第 10 章的规格不能被随意相除来证明应用快几倍。

12.2　ring、递归交换与树分别假设什么

NCCL 的集合通信语义说明 AllReduce、AllGather 与 ReduceScatter 的输入/输出关系，实际算法会按版本、拓扑、协议和消息大小选择 [82]。下面的 α–β 模型只用于解释趋势：p 个 rank，每 rank 输入 M B，一次串行阶段启动延迟为 α，有效单向带宽为 B B/s。

ring AllReduce 由 reduce-scatter 和 all-gather 组成，每阶段 $p-1$ 轮、每轮发送 M/p：

$$V_{\text{ring}} = 2\frac{p-1}{p}M. \tag{12.1}$$

$$T_{\text{ring}} \approx 2(p-1)\alpha + 2\frac{p-1}{p}\frac{M}{B}. \tag{12.2}$$

V 表示每 rank 发送量；若报告发送加接收，口径又多一倍。带宽利用率、分块流水和多 channel 会改变实测，不能将这个式子逐项当成硬件计数器。

当 p 为 2 的幂且分块条件满足，recursive halving 的 reduce-scatter 与 recursive doubling 的 all-gather 合计 $2\log_2 p$ 轮，每 rank 总发送量仍为 $2(p-1)M/p$。每轮消息大小不同，理想时间为

$$T_{\text{rec-exchange}} \approx 2\log_2 p\,\alpha + 2\frac{p-1}{p}\frac{M}{B}. \tag{12.3}$$

它不能简单称为“tree”。朴素树形 reduce 再 broadcast，每层传完整 M，若不流水则约 $2\log_2 p(\alpha + M/B)$；双树、分块树和硬件归约进一步改变带宽项。在理想同带宽模型里递归交换似乎同时有低轮数与低字节，但实际伙伴映射、链路冲突、并行 channel 和 buffer 访问会改变有效 B。不应随意指定一个 tree 效率系数，再把解出的消息阈值说成 NCCL 的通用选择逻辑。

▶ 主例的一次 TP 通信预算

以传统每层两次激活 AllReduce 的 TP=4 实现为教学例，24 条 decode 序列、隐藏维 8192、BF16 激活，则每次输入张量 $M = 24 \times 8192 \times 2 = 393216$ B。ring 每 rank 发送量为 $1.5M = 589824$ B，每层两次、80 层，合计约 94.37 MB/rank/step。假设某条合成通信路径的有效 $B = 50$ GB/s、$\alpha = 3\,\mu$s，一次 ring 时间约

$$6 \times 3\,\mu\text{s} + 589824/(50 \times 10^9)\ \text{s} \approx 29.8\,\mu\text{s}.$$

160 次约 4.77 ms。若启动延迟翻倍，新增约 2.88 ms；仅把带宽翻倍只节约约 0.94 ms。这个例子说明小消息同步次数的重要性，**不是本次 NCCL 实测，也不是所有 TP 实现的精确次数**。融合归约、sequence parallel、通信重叠和 attention 实现都可能改变通信图。

12.3 拓扑提供容量，放置决定用到了哪里

Clos/fat-tree 通过多级交换提供多路径和较大的二分带宽；是否无阻塞需检查上/下行端口、速率、路由与调度，不能只看拓扑名字就保证任意同时流量都有线速。rail-optimized 把不同服务器的同号 NIC 接入同一轨，适合把某些通信限制在轨内，但通信算法也可能先机内聚合，再跨机交换。其收益取决于 rank 放置和实际通信图。

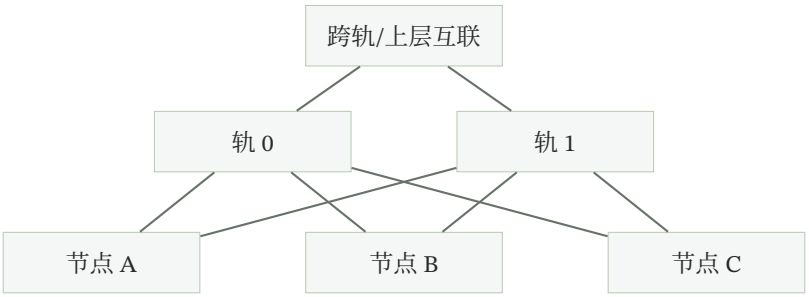


图 12.1 双轨拓扑的逻辑示意。每节点的不同 NIC 通往不同轨，跨轨流量使用上层路径。图未宣称端口等速或无收敛，验收还需给出每条链路和节点内 GPU–NIC 映射。

例如 4 节点、每节点 8 个 rank、8 轨，若均匀 all-to-all 向其余 31 个 rank 发送等量数据，与源 rank 同轨的其他节点 rank 仅 3 个，因此同轨比例为 3/31、跨轨为 28/31。若把本地节点内流量另行走 NVLink，分母和比例又改变；原稿的 7/8 只适用于包含源轨均匀选择等额目标的另一种近似。专家路由通常也不均匀，不能直接套这个比例。

收敛比例如 2:1 表示聚合下行带宽是上行的 2 倍，并不保证某个集合通信延迟恰好翻倍，也不能把“跨轨比例 × 收敛比”线性相加当拥塞模型。热点路径、队列、路径极化和同步决定关键时间。rail 与 EP 并非必然不兼容：分层 dispatch、专家放置、节点内聚合或多路径都可改变跨轨字节；需要实测取舍。

12.4 MoE all-to-all：看最忙的接收端

每 rank 处理 s 个 token，每 token 选 k_e 个专家、隐藏维 d。若每次远端 dispatch 发送完整激活、每次 combine 返回完整输出，数据字节分别为 b_d 与 b_c，平均远端路由比例为 q，则源端期望发送量近似

$$V_{\text{send}} \approx sk_e d(b_d + b_c)q. \tag{12.4}$$

这还未计路由索引、scale、对齐与 padding；同节点合并重复 token 的实现可减少网络字节。均匀分布只给出均值，某个热门专家的 rank 接收量可能显著更高。通信时间至少受下列资源之一限制：

$$T \gtrsim \max\left(\max_i \frac{V_{\text{send},i}}{B_{\text{send},i}}, \max_j \frac{V_{\text{recv},j}}{B_{\text{recv},j}}, \max_{\mathcal{C}} \frac{V_{\mathcal{C}}}{B_{\mathcal{C}}}\right), \tag{12.5}$$

$\mathcal{C}$ 表示一组切分网络的共享链路；启动、路由、同步和队列还会增加时间。

独立于主例的 MoE 教学例：$s = 128$、$k_e = 8$、$d = 7168$、dispatch 为 1 B、combine 为 2 B，若先按 $q = 1$ 上界记源数据，为 22.02 MB/rank/层；50 GB/s 的理想字节下界为 0.440 ms。它不等于端到端 all-to-all 延迟，也不能与一个假设 GEMM 耗时直接相加后宣布通信完全无法隐藏。分块 pipeline 可重叠 dispatch、专家计算与 combine，但有数据依赖的首尾部分仍留在关键路径。验收应记录最大 rank 接收量、专家负载分布和实际 overlap，而非仅源端平均字节。

12.5 KV 传输：带宽、窗口与剩余预算

主例 4096 输入的 KV 为 1.342 GB。第 11 章四卡 prefill 有效算力假设给出约 0.501 s 的本地重算；拉取是否更快应包含排队和固定开销。对长距离流量，除了端口带宽，还要有足够在途字节。简化单 TCP 流上界为

$$B_{\text{eff}} \le \min(B_{\text{bottleneck}}, W/\text{RTT}), \tag{12.6}$$

W 取拥塞窗口、接收窗口与应用可供数据中有效的限制。30 ms RTT 下，100 Gb/s 线速的带宽时延积为 375 MB；若人为限制一条流只有 1 MB 窗口，吞吐上界约 33.3 MB/s，主例 KV 至少传 40.3 s。现代 TCP 通常自动调窗，QUIC 还有流级/连接级流控，这个窗口例不代表其默认实测行为。

表 12.2 主例 KV 的数据时间下界，额外假设传输固定开销 30 ms。仅比较教学路径，不含真实排队、格式转换或 GPU 重算占用。

路径	带宽假设	时间
10 Gb/s、80%实得	1 GB/s	1.372 s
100 Gb/s、80%实得	10 GB/s	0.164 s
100 Gb/s、窗口仅 1 MB、RTT30 ms	不超过 0.0333 GB/s	至少 40.3 s
本地 prefill 重算	四卡 BF16 有效峰值 30%	约 0.501 s

GPU 有空闲会降低重算的资源机会成本，但不会让其时延归零；远端路径即使比重算更快，也可能占掉不可接受的 TTFT 预算或引入故障依赖。因此策略应比较预测完成时间、局部负载、传输费用与成功率，允许在超时后回落重算，同时取消无用传输并限制重复工作。压缩可减少字节，却加入编码、解码与质量约束；CacheGen 给出一种专门编码 KV 的研究路径 [83]，不能省去上述验收。

12.6 尾延迟：同步会放大什么

incast 是多发送者同时汇入有限接收路径，ring 单邻居模式可减少这种扇入，但 ring 也可能受到背景流量、共享链路和慢 rank 影响。all-to-all 并不要求 31 个发送者都持续以全线速灌同一端点；协议分轮、分块、限流与拥塞控制会改变瞬时扇入。队列被填满所需时间可做最坏情形估计，却不能直接证明实际 p99 一定到毫秒。

设每 rank 完成时间独立同分布、CDF 为 $F(t)$，则 p 个 rank 的同步完成时间为最大值，

$$\Pr(T_{\max} \le t) = F(t)^p. \tag{12.7}$$

若每 rank 在某阈值内完成概率为 0.99，64 个独立 rank 全部完成概率仅约 $0.99^{64} = 0.526$。真实网络的 rank 常相关，不能直接用独立模型预测，但无独立性也可用 union bound：若每 rank 超阈概率至多 ϵ，全组超阈概率至多 $p\epsilon$。只有独立指数分布时才有期望最大值等于均值乘调和数，不能将它再与任意排队模型的“倍数”相乘声称得到服务尾延迟。

排查从端到端追踪下钻：先看 ITL 尖刺是否与 collective 重合，再比较各 rank 的到达/离开时间，检查 NIC 重传、ECN/PFC、链路错误、GPU–NIC 路径和并发 IO。记录时钟同步方式与采样粒度，避免用一秒平均网口利用率解释几十微秒的同步事件。

练习与提示

1. TP 主例若每次消息缩小到原来的 1/4，ring 的带宽项与启动项分别怎样变化？提示：消息变小不等于层数和同步轮数减少。
2. 均匀 MoE 负载突然有一个专家接收量翻倍，式 (12.4) 的平均发送量是否足以判断时间？提示：使用式 (12.5) 中的最大值。
3. 需要 128 个独立 rank 整体至少 99%在阈值内，每 rank 需达到多高的完成概率？提示：解 $p_0^{128} \geq 0.99$，并说明现实中相关性需要另测。

本章证据边界

NCCL 官方文档用于核对集合通信语义；硬件标称带宽沿用第 10 章来源。所有通信时间和窗口例为明确参数下的解析或合成教学结果，没有在本次环境执行 GPU/RDMA 压测。拓扑的验收方案应固定 NCCL、驱动、NIC 固件、消息分布、rank 放置与背景负载，同时报告 median、p95、p99 和最大 rank 时间，而非只报告一个带宽峰值。

第 13 章　调度、编排与多租户

本章导读

引擎调度的是 token 与 KV，资源编排调度的是进程、设备与副本。两种控制回路相互配合：主例请求需在 2 s 内看到首 token，但新增四卡副本可能尚在读取 140 GB 权重。本章解释资源声明能保证什么、共享隔离的边界，以及弹性如何接上真实就绪时间。

13.1　Kubernetes 的资源账本与实际设备

传统 device plugin 向 kubelet 注册设备资源名，经 `ListAndWatch` 报告设备与健康状态，在分配时用 `Allocate` 提供设备节点、环境或 CDI 信息。extended resource 使用整数且不可超卖；默认独占设备模型本身不表达“这张卡的 30%带宽”[84]。厂商插件可以把 MIG 或时间片等逻辑资源再次按整数上报，因此 `nvidia.com/gpu: 1` 的具体含义必须连同插件策略查看。

• 一个 Pod 不能跨两台节点拼资源

Pod 被放到一台节点，其设备请求由该节点满足；申请 4 张 GPU 不会自动拼成两台各 2 张卡。若主例 TP=4 位于同一四卡或八卡节点，调度还需挑选正确设备子集与 CPU/NIC 亲和性；若跨节点并行，应由多个 Pod 或 worker 组成，另行保证通信、拓扑和成组就绪。

GPU 数目不等于可用显存。默认独占策略下，引擎仍须在初始化时测可用内存，留出 KV、图与 workspace；共享策略下，还需知道显存额度、其他进程与驱动保留。拓扑则跨越节点放置、kubelet 设备选择、NUMA 和 GPU–NIC 路径。节点标签、node affinity、拓扑管理器、专用调度扩展与厂商驱动可以补充信息，不能把“默认只计整数”推广成整个 Kubernetes 永远没有拓扑能力。

DRA（Dynamic Resource Allocation）进一步用 ResourceClaim、ResourceSlice 与 DeviceClass 表达设备属性与选择，官方在线文档在本次核对时标注该核心特性自 v1.35 稳定，并说明当前调度器不支持 DRA 资源的抢占 [85]。这些是 *2026-10-22* 所读文档的版本状态，并不意味着所有 DRA 驱动或共享子特性都已具备相同成熟度，也不意味着拓扑需求只能用 DRA 解决。部署必须固定 Kubernetes 版本、驱动、feature gate 及调度扩展，验证所需功能的交集。

13.2　准入、放置与成组就绪

准入决定“现在能否开始占资源”，放置决定“落在哪台节点、哪组设备”。Kueue 文档把自身定位为作业队列与准入管理，底层 Pod 到节点的放置仍由调度器完成 [86]；Volcano 等扩展则提供 gang、队列与放置策略。两类工具的职责可以互补，但不能因此假定任意版本、任意控制器叠加都安全。

多 worker 模型需要成组资源。若一个 TP/PP 副本只启动部分 rank，它们可能占住 GPU 等待其余 rank，形成资源锁定或长时间空转。gang/预约式准入可减少这种半成品状态，而健康检查仍须验证全部 rank 和通信组就绪。推理同样有这个需求；“一个副本能服务”不表示“一个副本内部不需要多个 worker”。

对主例，一个四卡副本的就绪条件至少包括：模型版本一致、TP rank 齐全、KV 池可用、通信组初始化完成、必要图/算子预热通过、端点可达。若采用 prefill/decode 分离，两个池不必把整个集群绑成单个 gang，但最小可服务集合和配对能力必须明确；只有 prefill 没有 decode 的资源不应被宣传为可用服务容量。

13.3　MIG、MPS 与时间片的边界

共享应拆成地址空间、容量、带宽、计算、错误和管理权限六个问题，不能只按一个“强/弱”排序。

表 13.1 三层控制回路分工。延迟尺度由实现与负载决定，表内不设固定毫秒或分钟门槛。

平面	需要的信息	可以控制的动作
集群资源	节点、设备、拓扑、配额、故障域	准入、Pod 放置、增减副本、排空与驱逐
请求路由	队列、缓存位置、预计完成时间、租户	拒绝/排队、选择副本、PD 配对
引擎执行	token 预算、KV 块、批次、剩余长度	chunk、连续批处理、请求优先级和抢占

表 13.2 NVIDIA 共享机制的主要边界，具体能力依赖 GPU 与驱动版本。三者都不替代服务级准入与租户安全策略。

机制	提供的边界	需要额外验证
MIG	按支持 profile 划分计算与内存路径，GPU instance 内具相应资源与部分错误隔离	profile 是否容纳权重/KV，GPU 级重置、驱动和主机故障影响
MPS	多进程并发提交，版本相关的计算/内存限制，Volta 及以后各 client 有独立地址空间	约束配置、共享带宽干扰、致命错误传播、版本与用户模式
时间片	将设备暴露为多份逻辑访问机会	不保证每份等量吞吐，无 MIG 式显存/故障隔离，申请多份不保证成比例容量

MIG 的 GPU instance 包含一定的 SM 与内存资源，memory slice 具有独立的 L2、控制器等路径；同一 GPU instance 还可有 compute instance，二者不能混为同等级内存隔离 [87, 88]。这适合模型能放入固定 profile 且需要较稳定资源边界的服务。其代价是 profile 粒度、重配置和显存池被切分；大模型或长上下文未必适用。不能据此断言“推理很少用 MIG”或“任何错误都不外溢”。

MPS 也不宜概括成“所有 client 共享一个地址空间、没有配额”：官方架构文档说明 Volta 以后的独立 GPU 地址空间与有限错误遏制，当前 MPS 文档还描述版本相关的 SM/内存分区能力 [89, 90]。这些限制不等于对所有共享硬件给出严格性能保证。时间片文档明确指出多个逻辑副本不会自动换来成比例 GPU 份额 [91]。

选择前应运行干扰矩阵：以主例在线服务为受害者，分别加入 compute-heavy、bandwidth-heavy、显存增长、故障与突发 IO 邻居，测质量、TTFT/最大 ITL、OOM、恢复时间和 goodput。没有通用的“SM 超过 60%就不能共享”阈值；某个平均利用率低，也可能已被关键路径的短峰值限制。互不信任租户还需主机、容器、驱动与网络安全边界，MIG 不是完整的沙箱。

13.4 弹性容量取决于何时真正就绪

定义 T_r 为从决定扩容到新副本可承接目标负载的时间，包含排队获得节点、镜像与权重、通信、编译/图捕获、预热和注册；检测滞后可另记 T_d。若需求尖峰是持续 T_p 的矩形额外负载，新增容量在 $T_d + T_r$ 后立即全部可用，扩容能覆盖的这段尖峰时间比例是

$$\eta_{\text{el}} = \frac{\max(0, T_p - T_d - T_r)}{T_p}. \tag{13.1}$$

这是特定形状下的教学指标。真实需求斜坡、多个新增副本逐步就绪与排队积压应使用时间积分或离散事件回放，不能把一个比例当成通用弹性质量分数。

▶ 主例的权重加载下界与预热状态

四卡副本权重合计 140 GB，假设读取路径实得 10 GB/s，独立字节下界为 14 s；若只有 1 GB/s 则为 140 s。是否有四个 rank 不改变共享出口的总字节。假设（非实测）检测 10 s、就绪 90 s、矩形尖峰 60 s，则该尖峰覆盖比例为 0；尖峰 600 s 则为 $1 - 100/600 = 83.3\%$。这里的 90 s 是情景参数，**不是软件或物理地板**。

预热应分级：只缓存容器镜像；缓存本地权重；权重已在主机内存；GPU 进程已加载且预热完成。它们占用不同资源、缩短不同阶段，不能统称为“预热池 75 秒”。

对短于检测加就绪时间的不可排队负载，需要常驻余量、预测性扩容、可借用的就绪副本、限流或可

接受的降级；不能承诺事后扩容挽救已经违反 TTFT 的请求。缩容也要有顺序：先停止新流量，排空在途请求和 KV 搬运，再注销端点并释放设备；超过排空期限的长请求需要明确取消或迁移策略，不能只把 Pod 设为不健康然后等待客户端超时。

扩容信号应联合排队时间、等待请求、KV 压力、输入长度、prefill/decode 比例和 SLO 违约。单独使用 GPU 忙碌百分比可能漏掉访存或队列瓶颈，单独使用 KV 占用也可能把可驱逐前缀误当紧急压力。应把新增副本的预计到达时间带入控制回路，并设置迟滞与冷却，防止反复扩缩。

13.5 租户公平与抢占代价

配额限制最大占用而非保证预留，优先级影响调度次序与抢占资格，准入控制限制进入系统的工作量，性能隔离则限制邻居干扰。namespace、RBAC 和 NetworkPolicy 不能替代 HBM 与 GPU 带宽隔离。此外，请求可以每秒占很少 GPU 却长时间保留大量 KV，因此租户额度宜同时考虑请求数、输入/输出 token、KV 驻留量和服务等级，而非只限 QPS。

Pod 级优先级不决定引擎内请求顺序。需要将服务等级从入口传到请求队列，并决定是否允许低优先级请求被暂挂、丢弃 KV 重算或迁移。抢占成本包括已计算前缀、重新排队、网络搬运、客户端截止期和重试放大；不能笼统说推理抢占必须在请求粒度，或所有训练都可低成本 checkpoint。同样，请求不是都只有毫秒，长输出和推理任务可持续很久。

• 缓存局部性也可能损害公平

只按前缀命中选副本，可能把热门租户集中到某几个 worker，让无共享前缀的请求长期等待。路由评分需同时约束预计完成时间、队列、租户额度与缓存收益；缓存 miss 应有可服务路径，不能让“有命中”成为进入系统的必要条件。

13.6 混部：从填谷收益到服务损失

混部可让可延迟批作业使用在线服务暂不需要的资源，但竞争可经 SM、HBM、NIC、存储、CPU 和功率限制传递。应先尝试跨节点分时借用，再评估同卡共享；物理独占隔离往往强于同卡分片，但代价是碎片和闲置，不能固定用一个全序代表所有故障与性能维度。

▶ 一个排队敏感性的合成示例

M/M/1 假设泊松到达、独立指数服务时间、单服务台且先来先服务。设 $\mu = 50/\mathrm{s}$、$\lambda = 30/\mathrm{s}$，$\rho = 0.6$，等待时间（不含服务本身）满足

$$\Pr(W_q > t) = \rho e^{-(\mu-\lambda)t}.$$

$t = 0.2\,\mathrm{s}$ 时约 1.10%。若背景作业使有效服务率降到 35/s、到达率不变，则 $\rho' = 30/35$，尾概率约 31.5%。这是**模型计算**，不是本书请求的实测违约率。真实引擎有批处理、变长 KV、相关到达、优先级和重尾服务，未必满足 M/M/1。若服务确定而到达泊松，对应 M/D/1；不能随意换用 D/M/1 或声称某替代模型必然给出较小尾部。

混部决策至少比较：填谷带来的有效产出，在线 SLO/质量损失，批作业被打断的工作量，checkpoint 与恢复 IO，以及安全和故障外溢成本。checkpoint 间隔应由写入成本与故障/驱逐过程共同决定，没有对所有训练都合适的“五分钟”。灰度时使用与独占基线相同的请求分布，给出停止混部和回退的可执行条件，才有办法把收益接回第 9 章的账本。

练习与提示

1. 主例四卡副本被错误标为 ready，但一个 rank 仍在 warmup，入口应该如何检测？提示：区分进程存活、通信组就绪与目标负载可服务。
2. 资源配额为 4 张 GPU 的租户是否随时拿得到 4 张 GPU？若需要保证，需补哪类机制？提示：上限、预留和准入优先级不同。

3. 一个 300 s 尖峰，检测滞后 20 s，两个副本分别在 40 s 与 100 s 后就绪，怎样估计有效扩容量？提示：分副本对时间积分，别只取平均启动时间。

版本与证据

Kubernetes device plugin/DRA、Kueue 和 NVIDIA MIG/MPS/时间片官方文档的页面快照、URL 与 SHA256 已保存，核对日期为 2026-10-22（北京时间）。版本相关能力以该快照为界，本章没有在现有机器部署 Kubernetes、运行 GPU 共享或验证隔离故障；给出的干扰矩阵为可执行实验方案。

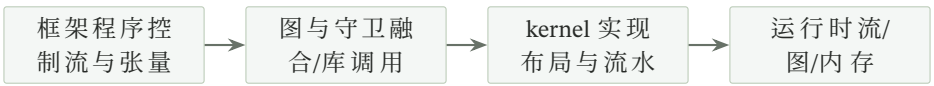
第 14 章　编译器、kernel 与运行时

本章导读

算子公式正确，只说明计算目标正确；图编译、kernel 映射与运行时还要决定实际执行顺序、内存布局与同步。本章解释三者如何配合，也解释为什么单 kernel 提速不能直接变成服务吞吐倍数。前置知识是 roofline、attention IO 及第 8 章的引擎执行路径。

14.1　三层优化分别改哪一笔账

图编译在多个算子间融合、消除冗余和规划内存；kernel 实现决定 tile、线程、流水与存储访问；运行时组织流、图、分配与同步。三者并非完全分离：图编译会调用库 kernel，算子布局限制图融合，运行时图捕获又依赖分配和同步方式。它们都可能报错、静默退化或产生数值错误，不能按层预设“越底层越不易发现”。



贯穿四个阶段的验证：语义/精度、实际生成路径、时间线、显存峰值、冷启动与端到端 SLO

图 14.1　从模型程序到执行的责任链。箭头表示依赖，实际编译可能输出厂商库调用、Triton 或其他后端，运行时并非只在最后才介入。

表 14.1　优化目标与可观察证据。没有任何一列可以替代数值正确性和端到端验证。

层	期望减少什么	验证入口
图编译	框架解释、重复中间张量、冗余算子	graph break/recompile 日志、生成代码、kernel 数量与访存
kernel	单算子计算、搬运、同步浪费	参考输出、tile/profile、有效带宽与算术强度
运行时	CPU 发射、GPU 等待、重复分配、错误同步	CPU/GPU 时间线、graph 命中、显存峰值、错误与长尾

14.2　图编译：守卫、动态形状与图断裂

以 PyTorch 为例，Dynamo 捕获可表示的程序片段并建立 guard，Inductor 优化图后可生成 Triton/C++ 或选择外部库调用。数据相关控制流、不支持的算子、Python 副作用等可能导致 graph break；这意味着分段执行，不必然意味着 *GPU* 同步，也不必然让所有融合收益归零。形状 guard 失败可能触发重编译，符号形状和动态编译则可覆盖一个形状族，具体支持范围随版本变化。

推理中 batch 和 prefill chunk 变化频繁，若每个形状都生成独立产物，编译延迟、缓存体积与 warmup 会成为扩容的一部分。处理方法包括定义支持的动态范围、对少数热点 shape 特化、为长尾提供经过测试的 fallback，并记录重编译次数与首用时间。不能仅凭 warmup 后的平均吞吐宣布部署可用；主例 2 s TTFT 预算也会被首请求编译侵占。

TVM 与 MLIR 提供编译中间表示和可组合变换的研究/工程路径 [92, 93]，它们与手写 kernel、厂商库并非互斥。“新增后端”既要表达语义，也要处理数据布局、低精度、同步与目标指令，不能把多架构适配缩成“只新增一个文件”。

编译产物也并非只能留在本机。PyTorch 官方文档描述本地、远端缓存，以及 `torch.compiler.save_cache_artifacts()` 和 `torch.compiler.load_cache_artifacts()` 等可移

植缓存接口，同时说明需验证 PyTorch/Triton 版本与 GPU 兼容性 [94]。因此更准确的问题是：当前版本、目标 GPU 和编译选项是否可复用、如何校验与回退，而非断言“编译产物无法跨机器共享”。二进制缓存应按可信产物管理，避免加载来源不明的代码。

14.3 kernel 语言：控制权和验证负担

Triton 以程序实例和 tile 表达数据并行，编译器负责一部分映射与流水；CUTLASS/CuTe 把布局、拷贝与矩阵乘组织成可组合组件；TileLang 等 DSL 探索显式 tile 与调度注解的结合 [95, 96, 97]。语言能力随版本演进，不能把某一种流水或 warp specialization 永久列为某 DSL 做不到的事情。

选择从 profile 出发：先定位占总时长的算子，确认是否已有适用的库实现，再评估手写 kernel 能否减少真实瓶颈。一个 roofline 斜坡上的算子未必需要更多 Tensor Core 指令；若访存不合并、寄存器溢出或 occupancy 过低，应先解决数据路径。当数据重用提高时，过大 tile 也可能用更多 SMEM/寄存器换来较少驻留，性能不随 tile 单调增长。

验证分两步。算子级用独立参考实现比较有限值、极端长度、全 mask/部分 mask、空张量、尾块、GQA 映射、NaN/Inf 处理及不同 dtype 容差；模型级再测 logit 误差、任务质量与长上下文退化。“元素误差足够小”不自动保证自回归输出相同，“输出文本不同”也不能单独证明算法错误，需固定随机种子与采样方式、检查数学上期望保持的性质。

14.4 CUDA Graph：减少发射而保留动态数据

CUDA Graph 把依赖关系及一组操作实例化后重复提交，可降低 CPU 发射与调度开销。图节点参数、图更新、条件节点等都有各自限制；实际引擎常采用固定缓冲、shape 桶和运行时元数据来控制复杂度 [98]。因此动态 shape 是设计约束，而非一句“完全无解”。

典型 replay 路径把新 token 与块表写入稳定地址的输入缓冲，再选择兼容的图；attention kernel 从元数据读取实际 KV 长度。是否兼容还取决于 grid、kernel、共享内存、指针、workspace、LoRA、attention 后端和执行模式，不能仅用“决定 grid 的字段”声称给出所有实现的最小充分捕获键。若后端不支持某模式，应显式记录 fallback 比例及触发形状。

▶ 主例的一笔发射开销账

假设四卡 BF16 主例的一个 decode step 有 800 个串行 kernel 边界，每处未被其他工作掩盖的平均空泡为 1.5 μs，则总空泡 1.2 ms。若 GPU 实际有用工作另需 20 ms，理想消除空泡可由 21.2 ms 降到 20 ms，加速约 $21.2/20 = 1.06$ 倍，而非让每个 kernel 本身快 6%。800、1.5 μs 和 20 ms 均为教学假设，真实空泡需从 CPU/GPU 时间线测量，异步提交会使“调用次数 $\times$CPU 调用时间”高估 GPU 关键路径。

桶化会使实际 batch b 上取整到捕获尺寸 $c(b)$。若算子对 dummy token 确实做完整工作，可用 $(c-b)/b$ 估额外 token 槽位，但不同 kernel 的 mask、形状和带宽成本并不同比例。例如自选桶 $\{1, 2, 4, 8, 16, 32, 64\}$ 时，$b = 24$ 上取 32，多出 8 个槽位，即 33.3%；这不是任何固定引擎的默认配置，也不是 33.3%端到端减速。

令实际 batch 分布为 $p(b)$、一套捕获集合为 $\mathcal{C}$、观察窗口有 K 步，更合理的比较是

$$\overline{T}(\mathcal{C}) = \sum_{b} p(b) T_{\text{step}}(b, \mathcal{C}) + \frac{T_{\text{capture}}(\mathcal{C})}{K}, \tag{14.1}$$

并同时施加图显存、KV 容量、冷启动和 SLO 约束。时间、美元或能量可以形成统一目标，但“padding 算力百分比”和“显存占用百分比”不能直接相加求一个通用最优图数。捕获图还可能共享内存池，每新增一图的显存并非固定 40 MB。应对照真实形状分布测桶粒度、fallback、峰值显存和 warmup 时间。

本书固定 vLLM v0.11.0 及 SGLang v0.5.3.post3 的入口见第 8 章；模型 runner 与 attention 后端协商执行模式，配置和捕获集合属于版本相关机制。本章移除旧稿从未固定源码推得的“51 张图”“Blackwell 上限 1024”和目录计数，避免把另一版本实现写成固定 release 事实。

14.5 约束解码：语法正确性也是运行时工作

结构化输出可用文法状态约束下一个合法 token，把非法 token 的 logit 屏蔽。正则可使用有限状态机，而递归文法或复杂 JSON 约束可能需要栈、语法解析和状态压缩，并非所有 schema 都能完整表示成有限状态机。tokenizer 的一个 token 可能跨越多个字符与语法边界，因此“合法字符”到“合法 token”还要做增量匹配。

词表 $|V| = 128000$ 时，紧凑 bitset 占 16000 B；若展开为 32 位浮点 mask 则占 512000 B。前者与主例 4096 上下文的 1.342 GB KV 相比仅约 0.0012%，不是 *5%*。但不能按字节占比推断时间可忽略：CPU 语法推进、掩码构造、H2D、小 kernel 发射和采样同步可能位于关键路径。相同 batch 内不同文法状态仍可以使用逐行 mask 并批量采样，不必因状态不同而逐请求发一个采样 kernel。

表 14.2 约束解码应分别验证的三层性质。语法满足不等于内容事实正确。

层次	所保证的性质	失效或退化入口
文法与 tokenizer	允许的 token 延伸不破坏所支持文法	UTF-8、跨字符 token、递归/枚举规模、EOS
数值与采样	非法 token 被正确屏蔽，合法集合按定义采样	全 mask、NaN、空合法集、温度与 top-k 次序
服务与语义	预算内产出可解析结构，内容满足应用验证	超长输出、拒绝/截断、字段事实和业务约束

jump-forward 尝试直接追加文法唯一确定的延伸，随后批量补算必要 KV，减少串行前向步数。这需要验证 tokenization 边界、文法状态与已计算前缀长度；确定字符比例不等于可跳 token 比例，更不直接给出硬件无关的加速倍数。仍应测输出质量、结构合规率、长度分布、补算代价和有/无约束的 goodput。JSON 冒号后并非所有 schema 都强制同一个空格和引号，具体例子应由该文法确实确定的标点序列构造。

14.6 分页与虚拟内存：分配策略影响 kernel 接口

PagedAttention 把 KV 逻辑块映射到物理块，并使 attention kernel 按块表读取，减少最大长度预留和外部碎片 [2]。它同时包含内存管理抽象与适配的计算访问方式，不能说成“仅仅是分配器”。块表间接寻址、分块传输、共享引用和跨后端布局是代价，但不同 paged kernel 可以采用不同优化，不必各自 fork 一个连续 kernel。

vAttention 探索另一种取舍：预留连续虚拟地址，物理页按需映射，使部分现成连续 attention kernel 继续可用 [65]。CUDA VMM 提供地址预留、物理分配、映射与访问权限等接口，分配粒度可查询且与设备/分配属性相关 [99]，不能用固定“MB 级”替代所有情况。实际还要评估映射开销、地址空间、驱动要求、共享页和前缀写时复制粒度，以及跨芯片后端可用性。

vAttention 论文 v3 摘要报告相对所比较 PagedAttention 版 FlashAttention/FlashInfer 的最高 1.23 倍服务吞吐，这是该论文配置的报告，本次未复现；它不证明 VMM 在所有负载或设备上占优。相应地，PagedAttention 被广泛采用也不证明另一方法“已经输掉生态竞争”，更不能据此预测十年标准。可复用的判断是：布局与共享语义、kernel 覆盖、移植成本和真实 SLO 收益必须一起衡量。

14.7 从单 kernel 结果回到服务收益

FlashAttention-3 论文 v2 报告 H100 上 FP16 最高 740 TFLOP/s、相对所比实现 1.5–2 倍，以及 FP8 接近 1.2 PFLOP/s[11]。这是 attention kernel 在论文所测形状与精度下的结果，不是 70B 服务端到端倍数；prefill 与 decode 也不共享同一 attention 形状和瓶颈。

若优化前某算子占端到端串行关键时间比例 f，单算子加速 s 倍，其他工作不变且暂忽略重叠，则 Amdahl 关系为

$$S_{\text{end}} = \frac{1}{(1-f) + f/s}. \tag{14.2}$$

$f = 0.2$、$s = 2$ 时端到端仅约 1.11 倍；$f = 0.8$ 时才约 1.67 倍。有通信重叠、批次改变或队列时需重测关键路径；服务在饱和附近甚至会出现非线性排队收益，也可能因图显存减少 KV 容量而降低 goodput。两种方向都不能由单 kernel TFLOPS 预先保证。

跨架构维护可以拆成四类工作：语义与边界、布局/同步、调优空间、精度与模型回归。共享上层语义通常减少重复劳动，但是否重写及团队规模由库覆盖、芯片差异与验收范围决定。不能从“架构数相乘”推出团队必翻倍、四种架构后必放弃自研，或支持更多芯片必亏损。更值得长期保存的是性能归因、数值回归、可复现 autotune 记录与可信编译缓存，它们帮助新架构上线，也让旧架构回退更可控。

练习与提示

1. 主例一个热点 kernel 占串行时间 30%，提速 1.8 倍，理想端到端加速多少？提示：先算剩余 70%，再检查是否原本与通信重叠。
2. 对实际 batch 序列 $[8, 24, 24, 48]$，使用 $\{8, 16, 32, 64\}$ 桶，额外槽位总数是多少？平均每批浪费百分比与总额外槽位/总真实 token 是否一样？提示：区分均值的分母。
3. 一个 graph replay 没有加速，如何区分 CPU 已经提前发射、graph 未命中和 padding 抵消收益？提示：需要时间线、模式计数与实际工作量三类证据。

来源与实验范围

CUDA Graph/VMM、Hopper 与 PyTorch 缓存官方资料，以及 FA3 v2、vAttention v3 摘要快照保存在 `evidence/infra-official/`，访问日期为 2026-10-22（北京时间）。本章没有在 GPU 上复现这些论文性能；发射、桶化和 Amdahl 数字是解析/合成教学计算，正式 GPU 方案须记录硬件、driver/CUDA、框架/引擎 commit、形状、精度、预热和计时同步方式。

第 15 章　可靠性与容量规划

本章结论

(1) 独立、恒定失效率假设下，必须全体在线的 N 卡故障域，其失效间隔为单卡假设 MTBF 的 $1/N$。这个模型解释规模效应；公开训练作业中断记录不能直接当作推理卡的物理失效率。

(2) 推理容错同时涉及副本、请求状态与流式输出。摘除失效副本只恢复容量；已经发出的 token、工具副作用和丢失的 KV，需要独立的重试与降级契约。

(3) MFU 衡量必需运算相对峰值的效率，时间利用率衡量有请求执行的时间覆盖，goodput 衡量满足指定质量与 SLO 的产出。完整时间窗内已经测得的 goodput 可直接作成本分母，不能再乘一次利用率。

15.1　故障率算术：为什么故障是常态

先把「故障」从形容词变成数字。设单卡平均无故障时间（mean time between failures，MTBF）为 T，集群有 N 张卡，单卡失效时间服从指数分布。指数分布的关键性质是无记忆：任意时刻随后一小时内失效的概率为 $1-e^{-1/T}$，在 T 远大于一小时时近似 $1/T$，与已运行多久无关。N 张卡各自以 $1/T$ 的速率独立失效，叠加后仍是指数分布，速率相加：

$$\lambda_{\text{cl}} = \frac{N}{T}, \qquad \text{MTBF}_{\text{cl}} = \frac{T}{N}, \qquad \mathbb{E}\left[\text{故障数/天}\right] = \frac{24N}{T}. \tag{15.1}$$

它依赖三条假设：失效率恒定（落在浴盆曲线的平坦段）、各卡独立、故障域是单卡。三条都能被现实推翻：同批次硅片的缺陷是相关的，一个机架交换机的固件缺陷会一次带走几十张卡，供电与制冷故障以机柜为单位。所以式 (15.1) 给的是长期均值，而真实事件是成簇的——可能安静两周，然后一个下午连续三次；只看均值会低估最坏的那天。

▶ 从公开中断报告到教学失效率：不要偷换分母

Meta 的 Llama 3 报告在一个 54 天观察窗口内记录了 466 次作业中断，其中非计划中断 419 次，并描述训练最多使用 16,384 张 H100【论文】[5]。若额外假设整个窗口始终满配，GPU 小时为 $16{,}384\times54\times24 = 21{,}233{,}664$ GPU·h，除以 419 得约 50,677 GPU·h/次中断【自算】。这是假设暴露量对应的作业中断率，混有网络、软件等原因；最大卡数也不等于全窗口实际在线卡数，故不能据此声称单卡硬件 MTBF 为这个数。

本章另取 $T = 45{,}000$ h 作为恒定独立失效的教学输入，不是设备规格或本次实测。式 (15.1) 于是给出：1000 卡的间隔为 45 h、每天 0.533 次；10000 卡的间隔为 4.5 h、每天 5.333 次。图 15.1 对 T 扫描，目的是检查模型敏感性。不要用这些数字推断某生产集群的事故率；部署时须用本地故障分类、设备暴露量与相关故障记录校准。

以下把整个同步作业当作一个故障域，任一失效都令全域停机，并假设修复过程可用一个平均值描述。设平均修复时间（MTTR）为 R，该模型的可用性近似为

$$A = \frac{T/N}{T/N + R} = \frac{T}{T + NR}. \tag{15.2}$$

代入 $N = 1{,}000$、$T = 4.5\times10^4$ h、$R = 10$ min $= 0.167$ h，$NR = 167$ h，$A \approx 99.63\%$，每天约 5.3 分钟不可用；10,000 卡时 $NR = 1{,}667$ h，$A \approx 96.4\%$，每天 51 分钟。可用性缺口在 $NR \ll T$ 时近似为 NR/T，并非所有规模下线性；多副本推理服务更不要求每张卡都在线。该式只描述全域停机的时间损失，不能直接推出带冗余服务的可用性。

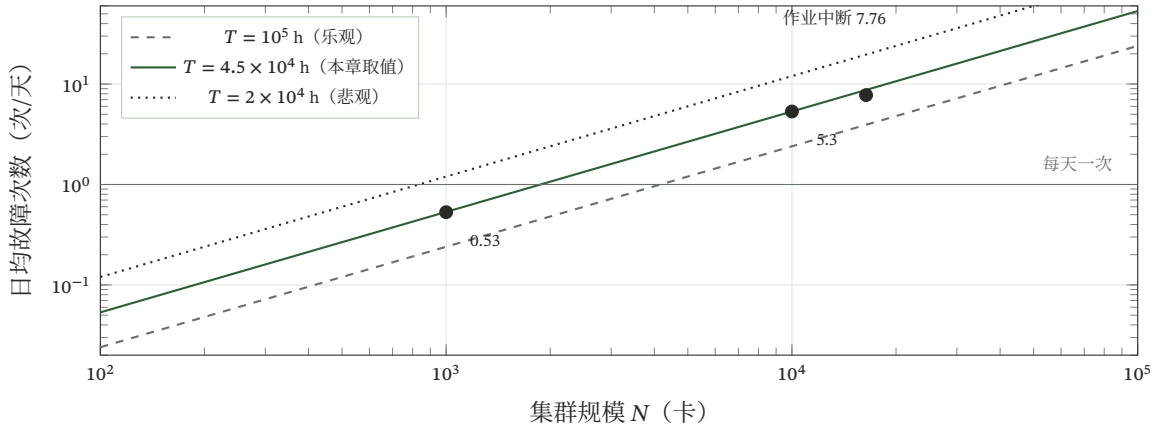


图 15.1 日均故障次数随集群规模线性增长，对数坐标下是一条斜率为 1 的直线。三条线对应假设单卡 MTBF 的高、本章取值、低三档，黑点是 1000 卡、10000 卡的推算值与 Llama 3 的报告作业中断值。独立恒定率模型中规模扩大十倍，事件频率扩大十倍。黑点中的作业中断与单卡失效口径不同，只作尺度参照，不能用其贴合程度验证独立性。

图 15.1 画的是事件频率而非修复时间。改变设备质量、故障域粒度与隔离策略，都能改变模型参数；缩短 R 只是其中一条路径。选择冗余时，应同时考虑同机、同机柜、同交换机与同发布批次的相关失效。

checkpoint 频率不是拍脑袋定的。设写入停顿 δ、间隔 τ、集群 MTBF $M = T/N$。单位时间损失由两部分组成：写入占 δ/τ，故障后回滚丢掉平均 $\tau/2$ 的工作量即 $\tau/(2M)$。总损失率

$$L(\tau) = \frac{\delta}{\tau} + \frac{\tau}{2M}, \qquad \frac{\mathrm{d}L}{\mathrm{d}\tau} = -\frac{\delta}{\tau^2} + \frac{1}{2M} = 0 \implies \tau^\star = \sqrt{2\delta M}. \tag{15.3}$$

这是 Young 在 1974 年给出的一阶近似 [100]。代入 $\delta = 30$ s、$M = 45$ h $= 1.62 \times 10^5$ s，得 $\tau^\star \approx 3{,}118$ s ≈ 52 min，开销 $\delta/\tau^\star \approx 0.96\%$。第 11 章提到业界习惯 30–60 分钟存一次；式 (15.3) 说明这一区间与特定参数假设一致，并不证明其普遍最优。它还要求故障稀少、停顿远小于间隔，且忽略恢复时间、异步写入与相关失效。若写入慢到 5 分钟，$\tau^\star$ 变成 164 min ——此时该修的是写入速度，而不是继续加密频率。

• 训练买时间，推理买副本

训练：故障域是整个作业（同步数据并行下任一张卡出错，全体停摆），恢复动作是回滚到最近的 checkpoint 并重启，代价是 $\tau/2$ 的训练时间加恢复期空转（万卡停 20 分钟即 3,333 GPU·h）。指标是有效训练时间比（ETTR）：Meta 高于 90%[5]，其他报告也讨论恢复效率 [101]，但不同作业的分母与中断分类须单独对齐。**推理**：故障域是一个 TP 组或一个副本，恢复动作是健康检查失败后摘除流量并重路由，代价是在飞请求丢失、KV 重算与摘除期间的容量缺口。它没有「回滚」——已经流给用户的 token 收不回来，只能靠副本冗余与请求级重试预算。

推理集群的故障率没有同等质量的公开统计，本书未找到可靠公开信息。按式 (15.1) 折算，一个 8 卡 TP 组的 MTBF 是 $T/8 = 5{,}625$ h（约 234 天）；100 个这样的组中任一组发生失效的间隔约 $T/800 = 56.25$ h。但这只描述硬件；推理侧还可能发生升级、权重加载失败、显存碎片导致的 OOM，它们不在上述独立硬件模型里，其占比须靠本地事故记录测定。

15.2 静默数据错误：不报错的错误

前面的讨论都默认失效是「响的」：进程退出、CUDA 报错、健康检查失败。还有一类失效不响——计算完成、没有异常、结果是错的，即静默数据错误（silent data corruption，SDC）。三层保护强度不同：存储有 CRC 与端到端校验，链路有 FEC 与重传，计算数据通路的保护范围依硬件而变，软件不能假设一次矩阵乘的所有中间结果都有端到端校验。Google 的观测是每数千台机器里就有几个「反复出错但多数时候正确」的核（mercurial core），朴素检测需要 2 倍额外工作、自动纠正需要 3 倍（三模冗余）【论文】[102]。Meta 在 18 个月里从数十万台机器中筛出数百颗有 SDC 的 CPU，指出这已不是「百万分之一」的罕见事件【论文】[103]。LLM-PRISM 研究永久 GPU 故障对训练的影响 [104]；这类故障注入揭示机制，不能当

作生产推理的发生率统计。

SDC 在推理中的风险不容易靠单一指标发现。以下比较说明检测面的差异，不给出训练与推理风险的普遍排序。

(1) **训练有统计哨兵，但也没有完美 oracle**。loss、梯度范数与激活统计有助于检测异常，却可能漏掉局部错误。推理的输出就是产品，错掉的 token 与模型的随机性仅从自然语言表面通常难以区分，需借助任务验证器或额外证据。

(2) **错误影响依赖位置与任务**。有些梯度错误可能被后续优化吸收，也可能造成发散；一次错误 token 可能正好是函数调用的参数、金额、剂量或条款，下游把错误放大成不可逆后果。

(3) **训练可回滚，推理不可回滚**。训练退回上一个 checkpoint 即可，推理已经发出的 token 无法撤回，需按产品契约中断、纠正或重新生成。

(4) **重放条件苛刻**。批组成、采样温度、KV 状态可能随运行变化，若未记录这些状态，就难以离线重放。固定环境、输入、种子与批组成可提高可复现性，但不能保证复现瞬时硬件故障。

Meta 的报告在 54 天、419 次非计划中断中把 6 次归为 SDC，约占 1.4%【论文】[5]。这是该作业在该窗口下的已识别中断分类，不是全部静默错误的检出率；设备实际暴露量和未检出事件都不完整，不能按卡数直接外推出另一推理集群多久遇到一次。

检测手段的成本与覆盖面需要共同评估。HBM ECC 针对存储位错误，不能据此保证整个计算数据通路；实际保护范围取决于芯片设计。独立冗余计算的两次结果比较可检测不一致，无法靠两份结果判定谁正确，且共同模式错误可能两次都相同；三模冗余也依赖错误独立与多数正确的假设，算力开销约 3 倍[102]；基于校验和的算法级容错（ABFT）借助 GEMM、归约等算子的特定代数关系检测错误，不能不经设计就覆盖 softmax、采样或完整服务；canary 请求（固定输入、已知输出的探针流量）与统计哨兵（logit 熵、重复率、输出长度漂移）边际成本较低但仍需预算，但只能抓住已覆盖路径上的大幅错误。现实的组合是「ECC + 便宜的哨兵+ 出问题时的双跑定位」；是否全量冗余应由任务风险、独立性与检测覆盖决定；可形式化验证的输出也可能优先使用更廉价的验证器，不能仅凭行业名称决定。

15.3 MFU、goodput 与利用率：三个分母

三个指标常被混着说，分子、分母与观察窗都要明确。**MFU**（model FLOPs utilization）的分母是理论峰值算力：

$$\text{MFU} = \frac{\text{观测吞吐（token/s）} \times C_{\text{model}}}{P_{\text{peak}} \times N_{\text{gpu}}}, \tag{15.4}$$

其中 C_{model} 是每 token 必需的浮点运算数，训练约 $6N_{\text{param}}$、推理前向约 $2N_{\text{param}}$。PaLM 的论文把口径定义清楚，并同时给出两个数：540B 模型在 6,144 张 TPU v4 上 MFU 46.2%、硬件 FLOPs 利用率（HFU）57.8%【论文】，差额来自重计算 [105]。同表里 GPT-3 是 21.3%、Gopher 32.5%、Megatron-Turing NLG 30.2%——这些数值提供相关工作的背景；不同模型、硬件与算子计数仍须单独对齐，不能把不同论文的 MFU 简单当成系统排名。$2N$ 和 $6N$ 也只近似参数主导运算，长上下文注意力等额外 FLOPs 需另计。

goodput 可按达标请求/s、达标输出 token/s 或每 GPU 归一化计量，必须写单位。DistServe 考察指定 SLO 达成率下的最大可持续请求率 [67]；本书另用观察窗内达标请求数定义

$$G_{\text{req/GPU}} = \frac{\#\{r : r\text{成功、质量合格且满足预定全部 SLO}\}}{N_{\text{gpu}} t}. \tag{15.5}$$

两种口径有关但不同：一个是在达成率约束下找容量，另一个直接计量达标产出。不得只把 TTFT 达标、但 ITL 违约的请求也计入完整 SLO goodput。

利用率（本书记号 u）的分母是墙钟时间，分子是「卡上真的有请求在算」的时间：

$$u = \frac{1}{N_{\text{gpu}}} \sum_{i=1}^{N_{\text{gpu}}} \frac{\text{卡 } i \text{ 有请求在算的墙钟时间}}{\text{墙钟时间}}. \tag{15.6}$$

三个指标的定义、分母与陷阱对照见表 15.1。

表 15.1 三个指标的定义与陷阱。最后一列是「这个数最容易被谁引用」，也是混淆的来源：三个数出自三套不同的计量目的，却被放在同一张幻灯片上比较。

指标	分母	分子与陷阱	解读
MFU	峰值算力 × 卡数	分子为观测吞吐乘每 token 必需 FLOP，即 FLOP/s；不含重计算；小 batch decode 常受带宽限制，但长 prefill 或大 batch 可能受算力限制	依模型与系统而定
goodput	观察窗时长；按每 GPU 归一化时再乘卡数	只数成功、质量与全部 SLO 达标的请求或输出 token；报数须带单位和长度权重	容量与成本须统一口径
利用率 u	墙钟时间	分子仅计请求执行的时间；与「GPU 占用率」（SM 有 kernel 在跑）不是一回事，后者会把自旋等待算成忙	由观察窗实测

计数时避免重复折损。定义观察窗长 t，客户端实际收到的原始输出数 K_{raw}（含随后失败的流已送出的 token），达标请求的输出数 K_{good}，则

$$Q_{\text{raw}} = \frac{K_{\text{raw}}}{t}, \qquad Q_{\text{good}} = \frac{K_{\text{good}}}{t} = Q_{\text{raw}} \frac{K_{\text{good}}}{K_{\text{raw}}}. \tag{15.7}$$

当 $K_{\text{raw}} > 0$ 时，末项比例是输出 token 加权的达标率，而非请求达标比例。仅当所有提交请求的实际输出等长时，才可把两者互换；失败前只输出部分 token 时尤其不能这样换算。成功完成请求的输出吞吐可另列，避免与原始输出混淆。全书主例的 600 原始输出 token/s、80% 达标若满足这个等长条件，才得到 480 达标输出 token/s【教学假设】。若已直接测到完整时间窗的 480，就不再乘时间利用率、MFU 或一次 SLO 比例。

! 常见误解

MFU 的吞吐若已经在同一完整观察窗测得，空转、批大小与调度停顿的影响已经在其中；把它再乘 u 会重复扣减。只有使用满载时吞吐 q_{busy} 并独立测得时间覆盖率时，才可在适用条件下写 $Q \approx uq_{\text{busy}}$。`nvidia-smi` 的“有 kernel 运行的采样时间比例”也不是有效产出率。成本应使用与计费边界一致的达标产出，必要时同时报告请求/s、输出 token/s、每 GPU 归一化结果。

15.4 SLO 与错误预算

先固定测量边界。客户端在 a 发出请求，收到第 j 个输出 token 的时间为 t_j，在 t_{done} 收到响应完成信号，输出共 m 个 token，则

$$\text{TTFT} = t_1 - a, \qquad \text{ITL}_j = t_j - t_{j-1}\ (j \geq 2),$$
$$\text{TPOT} = \frac{t_m - t_1}{m-1}\ (m > 1), \qquad T_{\text{e2e}} = t_{\text{done}} - a.$$

因此对 $m > 1$ 有

$$T_{\text{e2e}} = \text{TTFT} + (m-1)\text{TPOT} + (t_{\text{done}} - t_m).$$

仅在忽略最后 token 之后的协议收尾时，前两项之和才近似等于端到端延迟。TPOT 是单请求的平均后续 token 间隔，不能替代逐 token ITL 尾部。单 token 响应不定义 TPOT，应单列。服务端 step 时长、网络 SSE 片段间隔与客户端 token 时间戳也不能混用：若一次网络片段携带多个 token，须说明 token 计时的不可辨识性。

主请求输入 4096、输出 512，约定 TTFT 不超过 2 s、每请求最大 ITL 不超过 50 ms、端到端不超过 30 s【教学 SLO】。若前两条都满足，则最后 token 到达延迟 $t_m - a$ 的上界为 $2 + 511 \times 0.05 = 27.55$ s；响应收尾还需在剩余预算内完成，才满足端到端 SLO。正式 SLO 还要给出观察窗、对象集合、成功/质量标准和达成比例。

分位数不是阈值本身。P99 TTFT 是某个请求集合内 TTFT 分布的第 99 百分位；“99%请求满足 2 s”是阈值达成率，两者相关但需要同一集合与窗口。日均 300 万请求时，约 3 万次可能超过 P99，P99.9 尾部约

3000 次；这不说明每个尾部请求都违反产品 SLO，也不说明 P99.9 适用于所有业务。请求、token 和会话三种加权会给不同分位数，跨副本的分位数不能简单取平均。

在独立同分布假设下，一次会话发 30 个请求，若每请求超阈值概率 1%，会话至少一次超阈值概率为 $1-0.99^{30}\approx 26.0\%$【自算】；若事件受同一长 prefill、网络拥塞或发布影响，独立性不成立，需直接统计会话 SLO。

尾延迟的结构性可以用排队论看清。把每个副本近似为 M/M/1（泊松到达、指数服务时间、单服务台），ρ 为利用率、$\mathbb{E}[S]$ 为平均服务时间，则

$$\mathbb{E}[W_q]=\frac{\rho}{1-\rho}\,\mathbb{E}[S],\qquad \Pr\{W_q>t\}=\rho\,e^{-(1-\rho)\,t/\mathbb{E}[S]}. \tag{15.8}$$

平均排队随 ρ 的对应关系是：0.5 → 1.0、0.7 → 2.3、0.8 → 4.0、0.9 → 9.0、0.95 → 19 倍服务时间。取 $\rho=0.73$、令 $\Pr\{W_q>t\}=0.01$，解得 $t_{99}=\mathbb{E}[S]\ln(100\rho)/(1-\rho)=\mathbb{E}[S]\times 4.29/0.27\approx 15.9\,\mathbb{E}[S]$，而均值只有 $2.70\,\mathbb{E}[S]$ —— *P99* 排队是平均排队的 *5.9* 倍【自算】。这不是实现缺陷，是 $1/(1-\rho)$ 这个因子的必然结果。

假设要写清楚：M/M/1 要求泊松到达，而 LLM 流量有日周期与突发；服务时间也不是指数的，动态批处理还会改变并相关化服务时间。到达与服务离散性影响均值和尾部，偏差方向依赖联合分布。所以式 (15.8) 应作为解释负载率敏感性的教学模型；它不是实际系统容量的严格下界或上界，不能作为 SLA 承诺的依据。更一般的形式是 M/G/1 的 Pollaczek–Khinchine 公式 $\mathbb{E}[W_q]=\rho\mathbb{E}[S](1+C_s^2)/[2(1-\rho)]$，$C_s^2$ 是服务时间变异系数的平方 [106]；指数服务对应 $C_s^2=1$，退化为式 (15.8)。

▶ 尾延迟的测量陷阱：直方图分辨率与样本量

设自定义 TTFT 直方图相邻边界为 0.1 s、0.25 s【教学例】，落在这一区间的尾分位数只能按桶内假设插值，无法由桶计数恢复真实 0.11 s 与 0.24 s 的差别。SLO 阈值应设置桶边界；精细归因需请求级事件记录。桶配置随引擎版本变化，不能把某次源码中的桶硬编码成长期事实。

10^4 请求中，P99.9 之上的期望样本只有 10 个；相对计数波动约 $1/\sqrt{10}=32\%$，并不足以称作稳定估计。35 QPS 收集这些请求约 4.8 分钟，3 QPS 约 55.6 分钟；更低置信区间需要更多样本或适当统计模型。报告窗口、样本数、桶边界、超时与失败的处理方式，并保留短窗口告警与长窗口考核，两者目的不同。

错误预算把「发布速度」变成可计算的资源。设 SLO 为 99.9%（30 天滚动窗口），预算即允许失败的请求数，燃烧率是实际错误率相对预算的倍数：

$$B_{30}=(1-\mathrm{SLO})\times Q_{30},\qquad \mathrm{burn}=\frac{\text{实际错误率}}{1-\mathrm{SLO}}. \tag{15.9}$$

日请求 300 万的系统中，$B_{30}=0.001\times 9\times 10^7=9\times 10^4$ 次。若一次发布把错误率推到 2%，燃烧率 $=0.02/0.001=20$，预算 1.5 天烧完。政策因此可写成无歧义规则：燃烧率超阈值的窗口冻结功能性发布，只允许修复类变更；预算充裕时允许更激进的实验。这套机制出自 Google 的 SRE 实践 [107]，它对 AI 服务尤其重要：模型、引擎、prompt 三层同时在变，没有预算约束时「发布快」与「稳」只能靠部门间谈判。

15.5 profiling 工具链：四个视角

性能问题只有一个症状——慢，但成因分布在四个层次：时间去哪了（时间线）、这个 kernel 为什么慢（单 kernel）、哪段 Python 代码在等（算子级）、GPU 在等谁（系统层）。四层对应四件工具，用错层次会得到正确的错误答案。

四类工具的分工见表 15.2：时间线看整体形状，单 kernel 看达峰比，算子级看归因，内核态看「GPU 在等谁」。四者不能互相替代——用错层次会把「调度空洞」误诊成「kernel 慢」，然后花一周去优化一个本来就不慢的 kernel。

用法自顶向下：先 Nsight Systems 找空洞；空洞在 kernel 之间就查 eBPF 的 CPU 与 IO；在 kernel 内部

表 15.2 四件 profiling 工具的分工。第二列的粒度决定了它们能回答的问题：时间线工具看不见 kernel 内部的停顿，单 kernel 工具看不见 kernel 之间的空隙，这是必须两件一起用的根本原因。

工具	观测粒度	回答什么问题 / 代价	何时用
Nsight Systems	进程级时间线：kernel、memcpy、NCCL、CUDA API	时间花在哪个阶段；计算与通信是否重叠；有没有空洞。开销低，可跑完整 step，支持多进程多卡 [108]	**第一步**：先看 10–20 个 step 的时间线，找最大的空洞
Nsight Compute	单个 kernel 内部：roofline、访存事务、占用率	这个 kernel 离 roofline 多远、卡在带宽还是延迟。用 kernel replay 逐遍采集，每遍前可清空缓存以保证可复现 [109]；开销高，不适合整段跑	时间线里占比超过 10% 的那个 kernel
torch profiler	算子级：aten op、显存分配、Python 调用栈	哪个 op、哪一层、哪次 all-reduce 慢；Python 侧开销占多少；显存峰值在哪。带调用栈时开销明显上升 [110]	已知慢在模型内部，需要把时间归因到模块与代码行
eBPF	内核态：syscall、TCP 重传、off-CPU 阻塞、页缓存、块设备	GPU 在等谁；数据加载卡在存储还是网络；进程是否被调度出去。开销低，可长期挂在生产机上 [111]	时间线有大段 GPU 空闲，且 CPU 侧看不出原因

就用 Nsight Compute 打点；看不出热点则用 torch profiler 归因到算子。eBPF 可补充操作系统层的“GPU 为什么在等”证据，这步也最常被跳过：8 卡节点的数据加载线程被 Python GIL 卡住时，时间线上是纯粹的空洞，GPU 侧指标却都正常。

15.6 可观测性：从指标到一次请求的全链路

前面几节讲的是「怎么知道系统好不好」，这一节讲「怎么知道它为什么不好」。两者的工具完全不同：前者是聚合指标（利用率、goodput、P99），后者是单请求追踪。生产系统两者都要，但仅有前者仍不足以诊断个体请求。

15.6.1 三层可观测性，回答三个不同的问题

层	数据形态	能回答的问题与不能回答的问题	存储成本
Metrics	时间序列，聚合到分钟/秒	能：趋势、告警、容量规划。不能：单个请求为什么慢	依序列基数、标签、保留期与压缩而变
Traces	单请求的跨服务调用树，带耗时与属性	能：一次请求的时间花在哪一段。不能：这一段为什么慢	依跨度数、属性与采样比例而变
Logs	离散事件文本	能：具体的错误、参数、异常路径。不能：统计意义上的分布	依事件量、字段、采样与索引而变

推理服务的特殊性在于：它的一次请求在时间上极长。一次 4k 上下文的生成可能持续 30 秒，期间发生数百次 GPU step。传统 trace 的「span」粒度太粗（一个 30 秒的 span 什么也说明不了），需要把请求与 step 级执行事件关联。可按批次记录 step 事件或对异常请求采样 span，记录 batch 组成、token 数与耗时；不必对每个请求的每一步永久存一份重复 span。

这也带来一个工程约束：trace 数据量随请求数 × 步数增长。一个 1000 QPS、平均 200 步的服务，每秒产生 20 万个 span。因此必须采样，而采样策略的选择直接决定能否抓到长尾问题。

! 常见误解

「只用 1% 均匀采样就够」——可能漏掉低频故障，也无法保证单次异常有完整上下文。P99 的问题只占 1% 的请求，按 1% 采样后，你要看的那个形态只剩万分之一的样本，且这些样本的 *batch* 组成、上下文长度、并发状态都是随机组合，无法归因。
一种实用组合是常规随机样本加尾部采样：先定义一组「值得记录」的条件（TTFT 超阈值、ITL 抖动超阈值、请求被抢占、KV 命中率异常低、输出长度超过分布尾部），对命中的请求提升保留率，并设置缓冲、速率上限、隐私脱敏和采样开销预算。仅保留异常样本会产生选择偏差，不能拿它估计全体

SLO 达成率。这样样本量可控，且每一条都自带「为什么被记下」的标签。

15.6.2 必须埋的六个点

不是所有指标都值得埋。以「定位一次慢请求」为目标反推，最小集合是六个：

(1) **到达与排队**——请求进入调度器的时间、被调度的时间。差值就是排队延迟，它是 SLO 违约最常见的来源，却最容易被算进「推理时间」。
(2) **本步批组成**——本步的 batch 大小、各请求分到的 token 数、是否有 prefill 混入。第 7 章讲过 ITL 尖刺来自 prefill 混批，但没有批组成记录就无法证实这一点。
(3) **KV 状态**——本步的 KV 占用率、是否发生抢占、是否命中前缀缓存。抢占是最难从外部观察的现象：请求没有失败，只是变慢了。
(4) **每步的 GPU 忙闲**——不是利用率平均值，而是每步的有效计算时间除以步时长。平均值会掩盖「一半的步很忙、一半的步空转」。
(5) **输出长度与结束原因**——是自然结束、达到长度上限、还是被超时中断。三者的延迟分布完全不同，混在一起看会得出错误结论。
(6) **版本与配置的指纹**——模型的量化方式、max_num_seqs、块大小、是否开了投机解码。没有这组指纹，跨时间的对比毫无意义：配置变了，指标自然变。

最后一条在实践中比看起来重要。一次「升级后变慢」的排查，最常见的根因不是代码退化，而是默认配置变了——例如 max_num_seqs 被调小、块大小被改、或者某个优化在特定形状下自动关闭。把配置指纹写进 trace 属性，能让这类问题在几分钟内定位，而不是几天。

15.6.3 尾延迟的归因：把 P99 拆成三段

一个可操作的做法是把延迟按来源而不是按阶段拆开。对每个被记录的慢请求，计算三项占比：

$$1 = \underbrace{\frac{T_{\text{queue}}}{T_{\text{total}}}}_{\text{排队占比}} + \underbrace{\frac{T_{\text{bubble}}}{T_{\text{total}}}}_{\text{空洞占比}} + \underbrace{\frac{T_{\text{compute}}}{T_{\text{total}}}}_{\text{真算占比}}, \qquad T_{\text{total}} = T_{\text{queue}} + T_{\text{bubble}} + T_{\text{compute}}. \tag{15.10}$$

其中 T_{bubble} 是「轮到你了但 GPU 没在为你算」的时间（等通信、等 kernel 启动、等别的请求的 prefill 结束）。这是按请求关键路径划分、互斥且穷尽时的记账恒等式。通信与计算可能重叠，不能把各 GPU 事件时长直接相加；一个慢样本也可能由多类原因共同贡献。

这个方法的价值在于它把「慢」变成了可执行的动作：调度占比高就该改调度策略（第 7 章），空洞占比高就该查通信与 kernel 启动（第 14 章），真算占比高才说明是硬件或算法问题。没有这一步拆分，所有优化都是在猜。

▶ 合成教学归因示例（未运行生产实验）

某服务 P99 TTFT 为 3.2 秒，而 P50 只有 0.6 秒。拆开三个 P99 样本：

样本	排队	空洞	真算	归因
A	2.6 s	0.1 s	0.5 s	排队：该副本过载，应扩副本或改路由
B	0.3 s	2.4 s	0.5 s	空洞：等一个长 prefill 的 chunk 结束
C	0.4 s	0.2 s	2.6 s	真算：该请求 prompt 特别长（合理）

三个样本的「慢」来自三件完全不同的事：A 是容量不足，B 是混批的调度问题，C 是负载本身的形状（不是问题）。若不拆分，看到的只是「P99 是 3.2 秒」，而最可能的反应是「加机器」——这对 B 和 C 都无效，对 A 有效但成本最高。

这个例子说明了一件事：**尾延迟的治理不是「把 P99 压下去」，而是「把 P99 的样本分类，只治该治的**

那一类」。而分类的前提是埋点足够细到能算出这三项——这就是为什么可观测性不是运维的附加项，而是性能工程的前置条件。

15.7 容量规划：从请求到卡数

容量规划至少需要长度联合分布、到达过程、缓存可复用 token 比例、质量与延迟 SLO、版本和故障域。输入均值只能给资源下界；P99 长度不能简单替换均值来得到 P99 服务时延，因为长度、到达突发与批组成相关。图 15.2 给出迭代顺序。



图 15.2 容量规划从解析下界走向服务级验证。每次改变模型、并行度、缓存和调度策略，都可能改变主约束，需在相同负载与 SLO 下重新比较。

Little 定律对稳定且守恒的系统给出平均在途数 $\bar{B} = \lambda \mathbb{E}[T]$，不要求泊松到达。若只数 decode 中的请求，平均输出长度为 $\bar{m}$，实际平均后续 token 间隔近似为 $\bar{\tau}$，忽略长度与速率相关性才可近似写

$$\bar{B}_{\text{decode}} \simeq \lambda\, \mathbb{E}[(m-1)\text{TPOT}] \simeq \lambda(\bar{m}-1)\bar{\tau}. \tag{15.11}$$

这里必须代入实际驻留时间，不能代入允许的 SLO 上限后断言“放宽 SLO 必然增加显存”。放宽约束扩大可选策略集合，最优成本原则上不会因此变差；只有主动采用更慢的策略并维持相同到达率时，在途数才可能增长。

▶ 把全书主请求接到容量下界

主例为 70B 稠密 GQA、80 层、8 个 KV 头、头维 128，BF16 权重 140 GB、KV 每 token 327,680 B。4096 输入+512 输出在保守保留 4608 个位置时占 1.50994944 GB = 1.40625 GiB KV/请求【自算】。四张 80 GB 卡若仅按 90%物理容量供权重、KV、工作区使用，共 288 GB；留权重 140 GB 后 148 GB。再按配置为每卡预留 8 GB 工作区，KV 预算为 116 GB，容量上限为 $\lfloor 116/1.50994944 \rfloor = 76$ 个请求。它只验证静态容量；块对齐、瞬时激活、CUDA Graph、投机分支和实际分片布局仍可能要求更多余量，不能把 76 当作 SLO 并发保证。

设这一四卡副本在指定负载与 *SLO* 下的达标输出吞吐为 480 token/s【教学输入，非本次 GPU 测量】，等长 512 输出且全部任务成功时，对应 480/512 = 0.9375 达标请求/s。若目标为 8 达标请求/s，则仅从产出估算至少 $\lceil 8/0.9375 \rceil = 9$ 个四卡副本。要在任意一个独立副本失效后仍有 9 个，至少配置 10 个；若采用故障后最多 80%经验证可持续容量的策略，则应先计算 $\lceil 8/(0.9375 \times 0.8) \rceil = 11$ 个承载副本，再增加一个故障冗余，共 12 个、48 张卡【自算】。这里 80%是额外运维预留，不是对已含 SLO 筛选的 480 再扣一次成功率。

这些数没有证明 SLO。480 本身须由开放环到达压测得到，副本间负载均衡、缓存局部性和故障后突发也须验证。容量不足可能表现为排队、抢占或拒绝；服务有界拒绝优于让无限队列把所有请求拖过 SLO。

15.8 故障场景与请求恢复契约

把请求生命周期映射到组件，才能把“可靠”变成可排查的动作。表 15.3 给出最小故障矩阵；它是建议验证方案，不是本次已执行的生产故障注入。

表 15.3 主请求的故障场景、入口与安全恢复条件。所有重试都受截止时间和总尝试预算约束。

场景	证据与排查入口	恢复/降级与验证条件
prefill 前副本退出	路由健康、队列、引擎进程和请求 ID	在未输出前重路由；使用幂等键避免重复计费；统计重试额外负载
输出一半时 TP 组失效	token 序号、完成原因、GPU 错误与网络时间线	旧流不能无缝当新流继续；声明中断，按协议重放或重新生成；工具副作用不可自动重做
KV 缓存驱逐或远端不可达	命中 token 数、块分配/抢占、传输超时	有预算时重算；限制并发回源，防止缓存雪崩；租户间保持缓存隔离
集合通信卡住	rank 健康、NCCL 错误/超时、链路与交换机计数	以 TP/EP 完整故障域摘除，避免只重启一个 rank 造成等待；独立副本接流量
OOM 或扩容冷启动	显存分项、长度、图捕获、权重加载就绪时间	准入限制、缩短允许最大长度或暂停低优先级流量；就绪后再接请求
发布/机柜相关失效	版本指纹、故障分布、供电/网络事件	分批发布与跨域副本；保留旧版回滚容量；验证域级失效而非仅单进程退出

重试不是免费容错。若每次尝试独立失败概率为 p、最多两次尝试，期望尝试数为 $1+p$；故障后失败相关或副本已满时，这个模型失效，重试会放大过载。采用全链路 deadline、指数退避与抖动、全局重试预算、请求/工具幂等以及并发上限；只有在另一故障域有剩余容量且 deadline 允许时，重试才有机会改善 goodput。流式结果的去重、计费与恢复语义应写进 API 契约。

• 练习与提示：冗余买的是什么

把上述 8 请求/s 改为 10 请求/s，并假设一个机柜含两个四卡副本。先按故障后 80%容量算所需承载副本，再加“同一故障域同时失去两个”的冗余。提示：$\lceil 10/(0.9375\times 0.8)\rceil = 14$，至少 16 副本；还须让副本分布到不同机柜。这个算术仍不覆盖交换机或发布带来的相关失效，不能直接宣称四个九可用性。

未解问题

1. **硬件独立失效模型没有覆盖软件与发布故障**。公开的大规模训练报告同时记录了硬件、软件、依赖与通信问题 [5, 101]；具体归因比例受作业和分类规则影响，不能直接迁移为本书推理集群的输入。一次发布可能在共同触发条件下引入相关失效，因此需要单独验证发布、回滚与故障域预算，不能默认所有故障服从齐次泊松过程。
2. **自然发生的 SDC 检出率难估**。现场记录只包含已被发现的事件，未检出事件的分母通常不可直接观察。可用已知注入位置和故障类型的实验测量特定检测器的覆盖，也可对确定性子算子、带标签任务和冗余结果做对照；这些覆盖率仍不等于真实硬件事件分布下的全系统检出率。投入决策可结合注入结果、失效后果和不确定性范围估计期望损失，不必假定所有 LLM 输出都有唯一答案。
3. **P99.9 在低 QPS 下难以精确测量**。只跑 3 QPS 的副本攒够 10^4 个样本要约 56 分钟，但尾部期望只有 10 个样本，其间副本可能已换过一次；跨副本聚合又会把「某个坏副本」的信号平均掉。是否存在可用的分层估计方法（按副本、按请求类型分别估计再合并），必须说明估计方法、样本量与置信区间。

Part III

产业、科学与展望

第 16 章　公司与开源生态：七层栈上的十六个观察点

本章结论

(1) 公司定位应从产品边界、计费单位和承担的风险出发。同一家公司可以同时卖卡时、模型 API 和客户环境内的软件；一个“推理云”标签不能替代成本账。

(2) 开源引擎与商业服务不是一一对应关系。开源降低采用成本，维护者可以提供适配、稳定性和支持服务；客户仍能选择自建或其他服务商，价值捕获要用实际交付和续约验证。

(3) 官网产品、融资公告与财务报表能支持的结论不同。本章给出截至 2026 年 10 月 22 日的产品快照；没有同口径的一手财务材料，就不推断公司毛利，更不把一家公司的局部业务推成全行业结论。

16.1　七层商业视图与五层执行模型

前文用一次请求的执行路径组织五层系统；这里沿用第 9 章的七层栈，观察谁提供组件、谁交付服务、谁承担闲置产能。两种视图的目的不同，不能把层数当作互相竞争的分类。一次请求从路由到调度、执行、KV 与集群，通常跨越多家供应商，也可能全部由一家垂直集成。

表 16.1 只列产品与生态的代表。它不表示市场份额、投资排名或某公司仅属于一层。开源项目不是公司；把 Triton、MLX 或 llama.cpp 列进生态图，也不意味着它们有相同的收入模式。

表 16.1　七层商业视图。实际采购还要确定版本、部署边界、计费条款和支持责任。

层	代表与产品	可核验的问题
L1 加速器	NVIDIA、AMD 及其他 GPU/专用芯片	买到的 SKU、内存、精度吞吐与软件支持矩阵是什么
L2 内存与互联	HBM、NVLink/NVSwitch、以太网与 IB 生态	拓扑、可用带宽、故障域和尾部拥塞由谁负责
L3 编译与 kernel	Triton、CUTLASS、厂商库与编译器	哪些算子原生支持，哪些需回退；数值与调试契约是什么
L4 算力与编排	无问芯穹、PPIO、Modal 等平台	用户买资源还是买达标服务；空闲、冷启动与跨域代价谁承担
L5 引擎与系统软件	vLLM、SGLang、TensorRT-LLM 及商业支持	模型/硬件/负载组合下的可复现性能和升级责任
L6 托管推理	Inferact、Fireworks、Together、Baseten、DeepInfra、硅基流动等	API 或专用部署的质量、SLO、账单、保留数据与容量承诺
L7 本地与边缘	llama.cpp、MLX 与设备运行时	权重格式、内存预算、离线能力与功耗/延迟限制

判断一项服务先问三个问题：按什么单位收费；它提高了哪种资源的可用产出；当上游把类似能力加入默认产品时，它还提供什么。第三问是竞争情景，不是“必然被吞并”的预言。芯片厂、云平台与独立软件既会竞争也会合作，合同和迁移成本比公司标签更能说明关系。

16.2　两个锚点：异构基础设施与引擎平台

无问芯穹与 Inferact 提供两种值得比较的产品边界：前者的公开材料强调异构基础设施，后者强调从 vLLM 出发的推理平台。表 16.2 区分官网自述与本书的分析，避免把营销定位误读成独立性能结论。

融资公告证明公司在某时点作出过披露，不证明收入、利润或投资回报。累计融资额除以一轮估值也不是“融资效率”：它混合了轮次、时间、股权稀释与不同币种。

表 16.2 两个观察锚点；产品信息来自官网，访问日期 2026 年 10 月 22 日。

维度	无问芯穹	Inferact
可核验入口	官方网站的算力/平台与技术说明 [112]	创立公告与 Platform/Cloud 产品页 [113, 114]
主要产品边界	异构算力与基础设施服务 ;具体芯片/模型适配须查版本矩阵	Platform 运行于客户算力，Cloud 运行于其提供的算力
技术核验问题	适配后哪些算子回退、跨芯片数值与故障如何处理	相同 vLLM 基线上额外优化、稳定性与支持的贡献如何归因
经济核验问题	标准产品与定制交付收入如何区分	软件、算力、支持的收费边界与利用率风险如何分配
融资披露	本次未取得可复核的一手完整融资/估值表, 不列累计数字	官方 2026 年 1 月公告称融资 1.5 亿美元、估值 8 亿美元；为公司自述
本书分析的风险	适配矩阵变动、客户定制与交付成本	开源可替代性、持续维护与客户价值验证

本章取证范围

2026 年 10 月 22 日重新读取的官网、产品/定价页与仓库页面，以及 URL、重定向、抓取日期和 SHA-256，保存在 `evidence/company-sources/`。动态页面只是访问时快照；官网自述支持“公开提供/宣称什么”，不支持“实际达到什么”。
本章未取得可核验的硅基流动等公司一手招股书分部报表；未获一手财务资料的主体不列估值、毛利、收入或亏损，也不据此作行业推断。媒体估值须有可引用来源；未列数字不表示相关主体从未披露。

16.3 L4：异构算力为什么可能形成服务

异构适配至少有模型、芯片、精度、长度形状和框架版本五个维度。最简单的组合计数是

$$A = M \times N, \tag{16.1}$$

其中 M 为模型族数，N 为硬件/软件目标数。它是潜在适配组合数，不是市场规模或收入公式；共享后端可能减少工作，特殊算子、版本更新与长跑稳定性又会增加工作。

表 16.3 把一个“支持异构”的主张拆成可以验证的项目。这比把官网支持的芯片数乘模型数更有操作性。

表 16.3 异构服务的技术证据清单。论文或官网宣称需落到相应版本和实验条件。

主张	应读取的证据	失效边界
可运行	模型/芯片/精度支持矩阵、回退路径、质量集	“能加载”不等于有可接受速度或正确数值
可通信	集合通信库、网络拓扑、消息大小分布与超时处理	某拓扑的混训结果不直接适用于低延迟 decode
可调度	全链路排队、缓存局部性、跨域传输和迁移记录	远端闲卡不一定能在 deadline 内接请求
可经营	长时间 goodput、故障切换、计费和运维成本	提高忙碌时间可能以 SLO 与隔离为代价

▶ 聚合的节约上限，不等于服务商毛利

假设两个中心资源成本相同，满载达标吞吐也相同，时间覆盖率从 30%提高到 50%，其他条件不变【教学假设】。固定成本主导时，单位成本比为 $c_{50}/c_{30} = 0.30/0.50 = 0.6$，降幅 40%。这只是客户原单位成本的可分配节约上限，并非聚合方收入或毛利。还要扣跨域网络、软件、支持、重试与额外容量，并确保提高的是达标产出而非纯忙碌比例。若高利用率使有效吞吐下降，40%就不成立。

技术论文讨论异构训练系统 [115]，可帮助理解通信与放置；其训练实验不能替代异构推理验证。判断商业可持续性，应观察标准配置的复用比例、适配周期、故障修复时间和续约，而不是推断“每个客户必然需要线性增加人力”。工程服务和软件收入可以共存，比例需要报表或合同证明。

16.4 L5：开源引擎与商业交付的边界

第 8 章固定版本比较 vLLM 和 SGLang。本章换一个问题：用户在免费可部署的代码之外，为什么还可能付费？通常是为了已验证的模型/硬件支持、稳定性、性能调优、监控、升级和支持响应；这些价值必须以明确交付物体现，而不能从维护者身份直接推导。

Inferact 的创立公告称其由 vLLM 创建者和核心维护者创办，并披露前述融资【公司公告】[113]。产品页明确区分 Platform 与 Cloud，也列出驻场推理工程师、模型支持、专用优化、伸缩与计量等交付【公司自述】[114]。这些材料支持产品形态的描述，不支持其收费金额、客户收益或收入占比；网页未提供的字段，本书不填入猜测。

开源治理、代码许可证和公司控制权还要分开。代码允许使用、修改和分发，不表示商标、托管服务或支持义务一并开放；仓库贡献者也不等于某一家公司。许可证与治理应回到固定版本仓库核查。与芯片厂的合作公告即使存在，也不能证明付费金额、排他性或“芯片厂停止自建”。

表 16.4 引擎相关交付的三种边界。毛利取决于实际成本，不能由产品名称直接判断。

形态	客户买到什么	提供方仍承担什么
开源软件/支持	可部署代码与约定支持响应	维护、适配、回归和支持；不必承担客户卡的闲置
客户算力上的平台	配置、编排、观测、优化和升级	平台正确性与交付；算力与数据边界依合同划分
托管 API/专用部署	指定模型能力与容量/质量/SLO	资源、利用率、缓存、故障恢复与计量；专用保留实例可转移部分风险

表 16.4 也说明“免费引擎”和“收费服务”可以互补。商业护城河的待验证项是：同样的代码公开后，服务商能否持续节约客户总成本，或减少客户自身不愿承担的工程风险。人才、生态和客户关系都是可能原因，不能直接折成固定估值倍数。

16.5 L6：公开报价能证明什么

报价页能核查产品名称、计量单位和当时列价，不能核查内部成本。本次取证的 Fireworks、Together、Baseten、Modal 与 DeepInfra 页面显示多种 API 或专用资源模式 [116, 117, 118, 119, 120]。同一平台可同时提供按 token 和按资源收费，故不能把它永久归为“算力批发商”或“token 工厂”。

表 16.5 把公司财务主张降到其证据真正支持的层级。

材料	可以支持	不能单独支持
API 定价页	某日、某模型和计费档的列价	实际成交价、客户折扣、单位成本或毛利
公司融资公告	公司披露的轮次与金额	经营质量、估值合理性、技术领先幅度
独立基准	特定版本/硬件/流量下的测量	所有模型与全部租户的成本优势
经核验的分部报表	该主体、期间与会计口径的收入/成本	竞争对手成本、全行业盈利性、单请求贡献

▶ 通用毛利算例：成本下降与涨价的比例不同

令单位售价为 p、完整同口径单位成本为 c，毛利率 $g = 1 - c/p$。某教学服务从 $g = -19\%$ 改善到 30%：若售价不变，成本须从 $1.19p$ 降到 $0.70p$，降幅 $1 - 0.70/1.19 = 41.18\%$；若成本不变，售价须变为原价的 $1.19/0.70 = 1.70$ 倍，即涨 70%，不是涨 41%【自算】。

调度提高利用率、降低资源单价、减少无效工作、调整质量/模型与改变定价，都可能改变毛利；谁贡献了多少，要分别测量。这个算例不是任何一家公司的财务预测。

16.6 十六个观察点：公司、产品与生态分别记账

以下卡片的“证据”只指公开身份和产品边界；没有进行 GPU 性能复现。产品页与仓库访问日期统一为 2026 年 10 月 22 日。

16.6.1 基础设施与平台

1. 无问芯穹（Infinigence AI）。公司；观察位置 L4。官网强调通用 AI 基础设施与异构算力服务 [112]。核查问题是支持矩阵、客户数据边界、调度域与交付责任；不从官网支持数量推市场份额。

2. HPC-AI Tech / Colossal-AI 生态。公司与开源项目分开观察；位置 L4/L5。ColossalAI 仓库提供大模型训练/推理相关代码 [121]，仓库能力不等于每一项商业支持承诺。核查问题是采用的版本、实际产品组件和维护范围。

3. PPIO。公司；位置 L4/L6。公开官网提供云与 AI 基础设施相关服务 [122]。需针对具体 SKU、区域、配额、持久化与网络计费检查合同。

4. Modal。公司；位置 L4/L6。定价页按 GPU、CPU、内存等资源计量，并说明按秒收费 [119]。冷启动、容器生命周期、空闲规则与存储费用共同决定账单，不能只比较 GPU 秒价。

16.6.2 引擎平台与托管推理

5. Inferact。公司；位置 L5/L6。Platform/Cloud 边界与创立公告如表 16.2；用户需比较其相对固定开源基线的增量收益、支持义务与退出成本。

6. 硅基流动（SiliconFlow）。公司；位置 L5/L6。官方文档列模型 API、预留实例、企业级 MaaS 等产品 [123]。这些产品覆盖不同风险边界。

7. Fireworks AI。公司；位置 L6。官方页区分 serverless 模型计价与按 GPU 秒计的按需部署 [116]。比较时需明确模型、共享或专用容量、批处理档及最低资源要求。

8. Together AI。公司；位置 L4/L6。官网列 serverless、批处理、预置吞吐与专用推理等模式 [117]。核查重点是产品档位与容量保证，不能从公司名称推断单一收费模式。

9. Baseten。公司；位置 L6。公开页提供专用部署、模型 API 与不同企业部署方案，资源有 GPU 和实例计量 [118]。SLO、冷启动与客户云内部部署条件应分别确认。

10. DeepInfra。公司；位置 L6。公开页按模型给输入/输出 token 价格，并列其他模型服务类型 [120]。量化、模型修订、缓存、输出限制与延迟条件须纳入同一评测，最低列价不能单独证明最低任务成本。

16.6.3 开源引擎与本地生态

11. vLLM。开源项目；位置 L5。调度、分页 KV、后端与扩展点见第 8 章的固定版本。任何商业公司对它的贡献不等于独占控制；使用者需自己确定升级和支持责任 [124]。

12. SGLang。开源项目；位置 L5。固定版本的 RadixCache、调度与执行路径见第 8 章。前端编程接口与服务端性能应分别评估，不把“有 DSL”当某类请求必然更快的证据 [125]。

13. TensorRT-LLM。NVIDIA 主导的开源软件项目；位置 L3/L5。仓库支持矩阵、构建与运行时是核验入口 [126]。它与商业支持、硬件绑定程度、其他引擎的后端复用关系应分开讨论。

14. llama.cpp / ggml 生态。开源项目；位置 L7。关注 GGUF 权重、后端和跨平台执行，公开代码为能力证据 [127]。社区量化格式可用于本地实验，不能直接外推多租户集群 goodput。

15. Apple MLX。开源数组框架；位置 L3/L7。面向 Apple silicon，统一内存与惰性执行等设计以官方仓库为准 [128]。它不是一项等价于 GPU 云 API 的收费服务，模型 serving 还需相应上层组件。

16. Kubernetes 与 GPU 编排生态。开源编排生态；位置 L4。设备插件、队列、拓扑与配额见第 13 章。它分配资源而不自动提供 token 级 SLO，推理路由和 KV 局部性仍需引擎/服务层处理。

16.7 三条可检验的竞争假设

(1) **异构增加可复用适配资产的价值**。检验：同样模型/精度/SLO 下，新增硬件适配时间、回退比例和长跑故障率是否改善；反例是硬件收敛或客户定制成本始终超过复用节约。

(2) **开源标准扩大采用，也压低通用功能的议价空间**。检验：企业愿为支持、可靠性或特定组合优化支付多少；反例是专有模型/硬件形成不可迁移的架构优势。引擎最终收敛到几个不是本书可证明的数字。

(3) **单位经济学随需求密度和 SLO 分化**。检验：同模型/质量/流量下的达标任务成本、现金贡献和续约，而非仅列价；反例是容量稀缺导致保留实例长期溢价，或业务负载使批处理/缓存几乎没有复用空间。

• 练习与提示：比较一个具体采购方案

从本章任选一个按资源和一个按 token 计费的产品，用同一份 4096 输入、512 输出的主请求流量，列出报价所缺的字段。提示：除单位价格，还须问区域/币种/税费、最低预留、缓存折扣、失败与重试计费、质量、TTFT/ITL 分位数、数据边界以及未使用容量的责任。字段不齐时，结论应是不可直接比较。

未解问题

1. 开源服务商如何证明长期增量价值？公开单点基准不能替代跨版本的稳定性、支持响应和迁移成本，缺少统一的纵向评测。
2. 按结果计费会把多少模型质量风险转给基础设施层？结果可验证性、任务长度和失败归因不统一，目前无法由 token 单价直接换算。
3. 中立性如何被验证？股权关系、治理、数据隔离、兼容性承诺和客户退出权分别影响中立性，不能由“第三方”三个字推定。

第 17 章　单位经济学：从资源账到达标任务成本

本章结论

(1) 成本的分子与分母必须覆盖同一对象、时间窗和服务边界。主例四卡资源分摊价加独立设施电费为 12.168 美元/小时，除以 480 达标输出 token/s，得 7.042 美元/百万输出 token【教学自算】；这个数包含处理配套输入的资源，不能再加一遍输入成本。

(2) 按卡时、按 token、按平台授权分别转移不同风险。出租方仍承担未售出的卡时，API 方承担共享容量和 SLO，软件方仍有维护、交付与支持成本。计费单位本身不能证明毛利高低。

(3) 增量贡献、经营现金与含折旧全成本回答不同问题。填闲置容量和新增容量应分开决策，质量、尾延迟、机会成本与恢复冗余都要进入同一账本，报价不能直接反推出竞争者利润。

17.1　三个单位与两套边界

资源的单位是 GPU 小时、kWh 和网络/存储字节；产品的单位可以是输入 token、输出 token、请求或成功任务。它们之间没有跨模型不变的换算常数。一个 token 对应的 FLOPs、KV 读取和延迟随模型、长度、精度、缓存和批组成变化；任务是否完成还受输出质量影响。

第 9 章给出 TCO 加法，第 15 章固定 goodput 与 SLO 口径。本章先把主请求接进成本，再讨论计价与经营。主例是 70B 稠密 GQA、4096 输入和 512 输出、四张 80 GB 卡；SLO 为 TTFT 不超过 2 s、每请求最大 ITL 不超过 50 ms、端到端不超过 30 s。480 达标输出 token/s 为教学吞吐输入，不代表本次在 GPU 上测得。所有价格以美元计，都是内部建模假设，未含税、折扣、地区网络出流和服务商利润。

租赁与自建不可叠加

本章主例使用每 GPU 小时 3 美元的资源分摊价，不含设施电费；独立加入平均 0.35 kW/GPU、PUE 1.2、电价 0.10 美元/kWh。若使用含电云租价，应把独立电费项置零；若资源分摊价已经含折旧、机位和运维，也不能再把自建年成本加上。本章后面的自建例是替代方案的教学账，不是主例的附加费用。时间截至 2026 年 10 月 22 日；没有把这些假设当成任何区域的市场报价。

17.2　贯穿请求：每小时、每输出 token、每请求

表 17.1　四卡主例的小时资源账。平均功耗是输入假设，不是 TDP 或实测能耗。

项目	计算与边界	美元/小时
资源分摊	4×3，含分摊项范围由内部账定义	12.000
设施电力	$4 \times 0.35 \times 1.2 \times 0.10$	0.168
合计	资源不含电时相加；含电租价时删独立电费	12.168

令 Q_g 为整个观察窗内的达标输出 token/s，小时成本 C_h 的单位为美元/小时，则

$$c_{\text{out},g} = \frac{C_h}{3600 Q_g}\, 10^6, \qquad c_{\text{request},g} = \frac{C_h}{3600 \lambda_g}. \tag{17.1}$$

式 (17.1) 的第一项单位为美元/百万达标输出 token，第二项为美元/达标请求。主例 $Q_g = 480$ 且输出等长 512，故 $\lambda_g = 480/512 = 0.9375$ 请求/s，得到

$$c_{\text{out},g} = \frac{12.168 \times 10^6}{3600 \times 480} = 7.0417, \qquad c_{\text{request},g} = 0.0036053 \text{ 美元}.$$

每小时达标输出 1,728,000 token、达标请求 3375 次。每个请求还有 4096 输入，若为展示而把达标请求的输入与输出 token 简单相加，分母变为 $480 \times (4096 + 512)/512 = 4320$ token/s，成本为 $7.0417/9 = 0.7824$ 美元/百万混合 token【自算】。它只是同一总成本换分母，没有说明输入和输出各自的边际成本。

! 常见误解

若 480 已经覆盖空转、失败与 SLO 违约，就不能再乘 u 或 80%成功率。反过来，若 600 是原始输出 token/s、80%是请求达标率，只有输出等长时才有 $Q_g = 600 \times 0.8 = 480$。长输出更容易超时的服务，须直接累计达标请求的输出数。未达标请求消耗的资源仍留在分子，不能因从分母剔除它们就同时删掉成本。

完整服务成本还应包含 CPU、内存、KV 存储、网络、平台支持和重试。主例表 17.1 未分项测量这些成本，所以应称给定边界的资源成本，不能声称客户最终账单或企业全成本已被算尽。主例脚本的参数和结果见本书实验记录。

17.3 输入、输出与缓存：先算账单，再谈资源

账单的通用表达式为

$$R = \frac{p_{\text{in}} K_{\text{in,miss}} + p_{\text{cache}} K_{\text{in,hit}} + p_{\text{out}} K_{\text{out}}}{10^6} + R_{\text{other}}, \tag{17.2}$$

其中每个 p 按美元/百万对应 token 计；缓存命中与未命中输入分开，R_{other} 包括保留实例、最低消费、存储等费用。模型 tokenizer、隐藏推理 token、失败重试和免费额度的计量规则也须明确。

作为账单算术而非成本推断，本次读取 DeepInfra 定价页时，Llama-3.3-70B-Instruct-Turbo 列价为输入 0.10、输出 0.32 美元/百万 token【官网快照，2026 年 10 月 22 日】[120]。若 4096/512 请求全部按这两档收费，单次列价账单为 $(4096 \times 0.10 + 512 \times 0.32)/10^6 = 0.00057344$ 美元；按输入输出总数平均为 0.12444 美元/百万混合 token【自算】。这不包含未披露折扣、服务质量与额外费用，也不说明该服务商的成本或毛利。不能拿四卡 BF16 教学成本去推断使用其他精度、硬件、并发和缓存策略的服务商亏损多少。

prefill 与 decode 单位成本可能不同，但没有固定的“输出必贵四到六倍”定律。只在同模型、相近每 token 必需 FLOPs、相同资源价格且忽略其他约束时，可以用有效算力利用率作粗略解释：

$$\frac{c_{\text{in}}}{c_{\text{out}}} \approx \frac{\text{MFU}_{\text{out}}}{\text{MFU}_{\text{in}}}. \tag{17.3}$$

注意力长度项、KV 带宽、排队、量化、共享 prefill/decode 资源与价格策略都可能破坏此近似。公开报价反映成本、供需、质量与竞争策略，不能当成 MFU 的直接测量。缓存复用省去部分 prefill，但仍需保存/检索 KV、读取上下文、生成后续输出，并承担一致性与隔离责任 [69]。

成本归因还要防止双算：若总小时成本除以输出吞吐，已经包含处理相配输入的资源，就不能把同一输入再按某比例加到总成本上。若确实需要输入与输出分摊，应先按独立 P/D 池、GPU 时间或明确的作业成本分配方法拆分分子，并披露这一会计选择。

17.4 三种计费，风险转移到哪

出租方的单位毛利并非与利用率无关。若自有卡全天成本 c_h、挂牌价 p_h、被付费预订比例 s，其收入为 $24sp_h$、成本约 $24c_h$；当固定成本主导时，毛利为 $1 - c_h/(sp_h)$。客户把已租到的卡跑多满，通常不改变这笔租约收入；提供方能卖掉多少卡时，却决定其利润。保留实例与按需实例对这项风险的转移程度也不同。

同理，授权软件的复制成本可能低，持续支持与客户定制未必低；人力增长是否线性、企业版是否比公有云更赚钱，都需实际收入/成本归集。按卡时买入再提供 API 的模式也不是算术上必然失败：需求聚合、量化、批处理、缓存、支持和商业折扣都可能创造价值。可持续性的检验是同一配置下的达标任务收入是否覆盖相应成本与风险，而不是两个不同模型的网页列价之间有没有表面价差。

表 17.2 三种计费方式。风险归属取决于合同，而不是公司名称。

维度	按资源时长	按 token/API	软件授权/客户侧平台
客户购买	可用资源或保留容量	调用结果与约定服务水平	使用权、维护、支持和交付
客户风险	租得资源能否高效利用	提示/输出长度与任务成功率	自有算力利用率和运维分工
提供方风险	售出率、故障、采购与闲置	共享容量、长度分布、缓存、质量与 SLO	维护、适配、回归、交付与支持
常见附加项	机型/区域/最低租期/网络存储	输入输出/缓存/批处理/保留实例档	年费/项目/席位/用量/支持等级
验证重点	相同资源规格与计费时长	同一真实流量的账单及达标任务数	升级成本、支持响应与退出可行性

17.5 利用率、SLO 与敏感性

若只有满载时达标吞吐 $q_{\text{busy},g}$，并假设活跃期间效率稳定、空闲期间无产出，才可写 $Q_g = uq_{\text{busy},g}$。在固定成本主导时，$c \propto 1/u$。放宽 SLO 扩大可选配置集合，但是否能提高 u 或吞吐仍要测量；长上下文可能已受 KV 容量或带宽限制。图 17.1 将这个条件模型画出来。

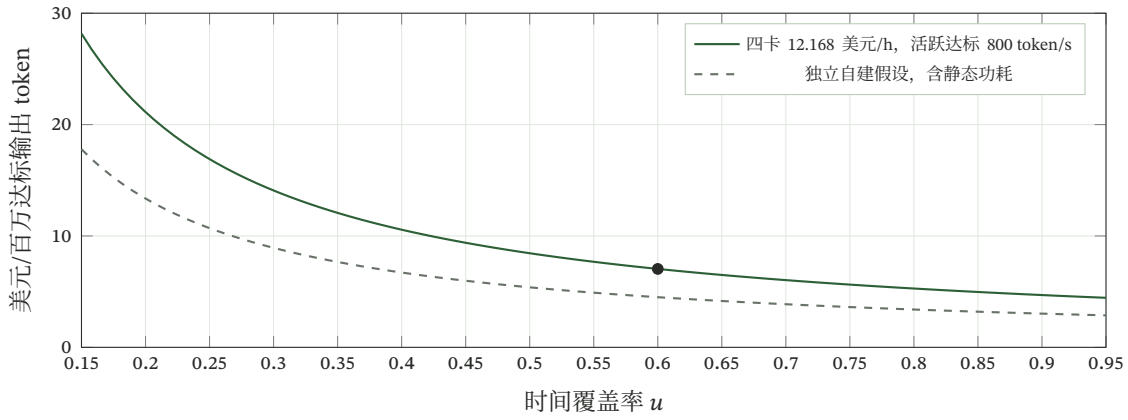


图 17.1 教学敏感性曲线。四卡曲线仅假设活跃达标 800 token/s 且小时成本固定，$u = 0.6$ 时等于主例 480；这与“原始 600、80%通过”是两套不同分解，不能混乘。自建曲线的独立参数见下一节。没有纳入不同模型的公司报价来推断盈亏。

主例可以直接做局部敏感性：保持总小时成本 12.168 美元，Q_g 从 480 减到 240，成本从 7.042 变 14.083 美元/百万输出；增至 960 则降为 3.521。保持 $Q_g = 480$，资源单价从 3 改为 1.5 或 6 美元/GPU 小时，成本分别为 3.569 和 13.986。PUE 从 1.2 改为 1.5，仅电力项从 0.168 变 0.210 美元/h，总单位成本变 7.066；这只说明本例电力占比低，不证明所有部署对能耗不敏感。低资源价格、端侧电池、功率限额或高电价下结论可能改变。

SLO 的作用不宜用一个普遍的 $u_{\max}$ 公式承诺。M/M/1 教学模型在负载率 ρ、均值服务时间 $\mathbb{E}[S]$ 下有

$$\Pr(W_q > T) = \rho \exp\left[-\frac{(1-\rho)T}{\mathbb{E}[S]}\right] \le \varepsilon. \tag{17.4}$$

对固定 T、ε，数值求解可得到这个模型的负载率上限。ρ 不是 GPU 时间利用率 u，continuous batching 也不是 M/M/1；输入长度、prefill 服务时间和 TTFT 边界都需另计。故配置选择更准确地写成

$$x^* = \arg\min_{x \in \mathcal{X}} c(x) \quad \text{满足} \quad \Pr\{r\text{质量与全部 SLO 达标} \mid x, \mathcal{W}\} \ge a, \tag{17.5}$$

其中 x 包括并行度、批大小、chunk、缓存、精度和副本数，$\mathcal{W}$ 为负载分布，a 为目标达成率。这个优化问题需要实测或经过校准的模型，并非解析式给出生产上界。

图 17.2 提示应在同样质量与 SLO 下比较成本，而不是先追最大 token/s 再补一个延迟数字。需要足够长的开放环到达压测、排队与拒绝计数，并在故障/扩容窗口重复。分层产品可以给不同任务不同期限，但是否节约成本由负载复用和资源曲线决定。

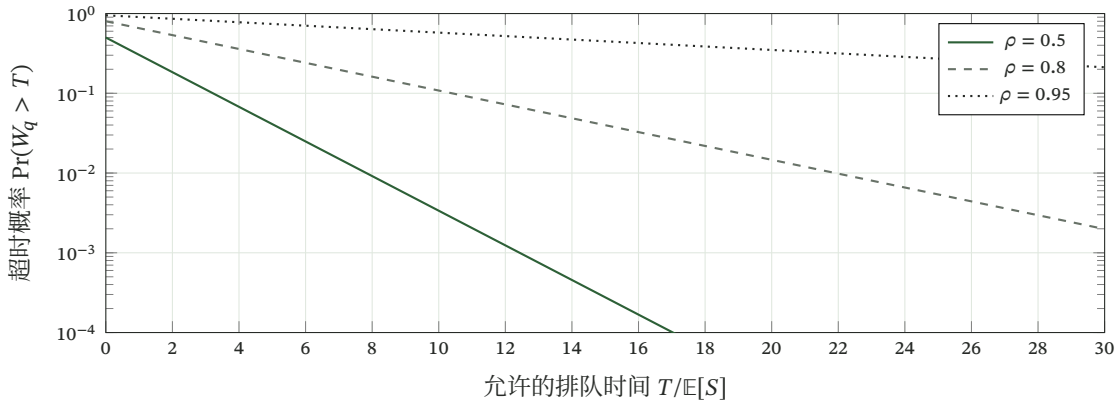


图 17.2 式 (17.4) 的 M/M/1 条件曲线，仅解释为什么更高负载率可能放大排队尾部。它不包含 LLM 批处理、KV 抢占和网络，不作为实际 TTFT 容量保证。

17.6 自建账本与三种成本

以下是独立的自建教学账：采购每卡 30,000 美元，残值零、3 年直线折旧；机位 2500、网络 1500、人力运维 2600 美元/年，故不含电的固定年成本 $F = 16{,}600$ 美元。假设每卡空闲功耗 $w_0 = 0.15$ kW、活跃功耗 $w_1 = 0.60$ kW，PUE 1.2、电价 0.10 美元/kWh，活跃时达标吞吐按每卡分摊 200 输出 token/s。均为假设，不是 *70B* 在单卡上独立部署，也不是市场报价或测量。

平均功耗模型为 $\bar{w}(u) = w_0 + (w_1 - w_0)u$，年电费 $E(u) = 8760\pi e\bar{w}(u)$；年达标输出为 $8760 \times 3600 \times 200u$。因此

$$c_{\text{full}}(u) = \frac{[F + E(u)]10^6}{8760 \times 3600 \times 200u} \simeq \frac{2.65691}{u} + 0.075. \tag{17.6}$$

静态功耗也被摊到 token 上，不能一面按全年 TDP 付电费，一面又说电费只随产出线性增加。$u = 0.6$ 时电费 441.504 美元/年，全成本 17,041.504 美元/年，单位全成本约 4.503 美元/百万达标输出 token【自算】。

把 10,000 美元折旧移除，经营现金支出为 7041.504 美元/年，对同一产出的平均现金成本约 1.861 美元/百万输出。若设备、机位与人员已付费且不挤占更有价值任务，额外活跃相对空闲的电量成本近似 $\pi e(w_1 - w_0)10^6/(3600 \times 200) = 0.075$ 美元/百万输出。这个增量值还没含新增网络、存储、磨损、支持或在线 SLO 损失，故不是通用的“物理价格底线”。

三者用途不同：增量贡献决定是否接一笔可容纳的任务；平均现金成本检验一段经营期能否支付现金费用；全成本检验资产更新与长期可持续性。平均现金亏损不意味着每一个新增 token 都产生负贡献，折旧也不等于可忽略的长期成本。若要扩产，应比较新增资产的现金流、预计利用率、残值与融资成本，不能简单把现有低边际电费外推。

17.7 批处理：把期限作为调度自由度

离线打分、评测、合成数据和 embedding 常允许较长完成期限。折价可以交换等待和重排的权利，但折价幅度、期限和容量限制是具体产品条款，不存在通用“24 小时必然五折”定律。

若一批任务 T 个计费单位、每单位价格 p_b，增量现金支出 ΔC、挤掉在线任务的机会成本 C_{opp}，接单贡献为

$$\Delta\Pi = p_b T - \Delta C - C_{\text{opp}}.$$

填闲置时段与新增容量的 ΔC 不同；是否提高了平均毛利又是另一问题。单位要统一：若 p_b 按百万 token 报价，T 须按百万 token 计，或显式除以 10^6。不能把每 token 和每百万 token 混在同一公式。

时间松弛允许按长度和共享前缀重排、提高批量、推迟到低价/低碳时段，并按平均完工量调投机参数；但 batch 仍受显存、带宽、计算、截止时间、公平性和失败恢复约束，不是装满 KV 就一定最快。离线请求也可能共享很长系统前缀，缓存收益不能预设较低；P/D 分离是否值得仍取决于资源互补和 KV 传输，而非“没有 TTFT 就无收益”。在线服务在公平与 deadline 约束内也可重排，二者是自由度大小的差别。

17.8 地区、合规与完整任务成本

跨地区比较至少固定币种汇率日期、税费、GPU/CPU 规格、网络出流、保留周期、账单粒度、SLO 与数据边界。电价只是资源账的一项；采购/融资、需求密度、容灾和数据治理可能更大，也可能在低硬件成本场景下由能源主导。没有相同模型与服务质量，不能把不同厂商的列价差解释为地理成本差。

生成式 AI 服务和数据出境相关规则具有适用范围、例外与持续更新 [129, 130]。本书只把数据驻留、访问控制、审计、评估与迁移约束列入架构和成本输入，不声称所有业务必须本地建集群，也不提供未经核实的评估报价或固定审批时长。具体方案须按实际服务对象、数据类别和现行规则判断。

• 练习与提示：从便宜 token 到便宜成功任务

假设方案 A 单请求账单 0.001 美元、任务成功率 70%，B 为 0.0013 美元、成功率 95%，一次失败允许独立重试一次且重试同价。比较成功任务成本，并讨论独立性不成立时怎么办。提示：两次内成功概率为 $1-(1-p)^2$，期望尝试数 $2-p$，成本/成功任务为 单价$(2-p)/[1-(1-p)^2] =$ 单价$/p$；A 约 0.001429、B 约 0.001368 美元。还要计超时、人工检查与副作用，不能只凭 token 单价选择。

未解问题

1. 折旧、残值、融资和需求量如何联合建模？三年直线折旧是会计输入，硬件经济寿命取决于转售、能效和后续可用负载；没有通用年限能替代敏感性分析。
2. 需求价格弹性、质量和 SLO 如何共同影响利润？降价可能带来更多需求，也可能使队列越过稳定区；只估价格弹性不足以确定最优价。
3. 缓存、多租户与共享 P/D 池的成本如何分摊？同一份 KV 可以服务多次请求，计费 token 与资源消耗不同。公平、可审计的分摊规则仍需要结合合同与服务测量。

第 18 章　前沿科学：当推理成为科研仪器

本章结论

(1) 科学负载不是单一 serving 类型：序列编码、自回归设计、图力场、扩散结构预测和气候场外推的计算/访存形态不同。先识别算子与依赖，再决定哪些 LLM 优化适用。

(2) 可复现性要区分逐位一致、容差内一致与统计等价。固定 seed 只约束随机源，固定 batch 也不能跨库、驱动和硬件承诺逐位相同；标准应由科学问题及验证方法决定。

(3) 工具等待可以降低独占会话保留下的资源使用率；跨会话复用可能填补空闲，仍受 KV 和工具容量约束。加 GPU 能够增加并发任务数，不能缩短同一任务不可并行的工具依赖链。

18.1　科学负载不是「更难的对话」

科学工作流会改变对话服务常用的优化目标，但并非所有科学请求都只能 batch=1。候选分子筛选、集合天气预报与批量序列编码通常有大量可并行样本；依次做实验的主动学习或 agent 工作流才可能缺少独立并发。对话输出也可能是需要严格验证的代码或工具参数，科学输出也常允许统计误差。

真正要列出的是输入/输出结构、跨请求独立性、可接受数值差异、期限与错误代价。输出可能是序列、坐标、能量、力或网格场，是否采用 token 解码、KV cache 与 grammar 约束，取决于模型结构。以下表格给出两类典型形态，不是领域的普遍分类。

表 18.1 把差异按可观测的量拆开，每一行都对应一个系统后果。

表 18.1　科学负载与消费级对话负载的逐项对照。右列是可直接观测的系统后果，其中 Roofline 上界只适用于正文的稠密序列模型，padding 比例也仅是线性工作量的近似。

维度	消费级对话	科学负载	系统后果
批组成	到达流量允许连续批处理	筛选可大批；闭环实验可能低并发	先按独立样本与长度分桶，测量 padding 或 ragged 开销
上下文	文本 token 与多模态输入	序列、图、坐标或网格场	KV、图边、激活和中间态按结构记账
算术强度	decode 强度随 batch 与 KV 变化	小图可能启动受限，长序列可能算力受限	理论上界随算子/硬件变化
输出形态	自然语言 token	FASTA/SDF/JSON/网格场	仅 token 生成可能用 grammar；结构还需物理/任务验证
延迟目标	TTFT 与 TPOT	作业完成时间、截止时间	按截止时间与任务质量验收
可复现性	允许重采样	逐位、容差或统计等价，依任务定义	验证批处理引入的差异，不预设必须禁用
优化杠杆	拼批、量化、投机解码	形状分桶、图 kernel、域分解、算子融合	量化和投机收益需按适用对象判断

batch=1 的算术强度可以算出上界。设参数量 N、序列长度 L、FP16 权重。一次前向的计算量是 $2NL$ FLOP，必须读取的权重是 $2N$ 字节。权重远大于 L2，每个样本都要重读一遍，于是单样本的固定成本与 MFU 上界分别是

$$t_w = \frac{2N}{B_{\text{mem}}}, \qquad \text{MFU} \le \min\left(1, \frac{2NL/P}{2N/B_{\text{mem}}}\right) = \min\left(1, \frac{L}{I^*}\right), \tag{18.1}$$

其中 $I^* = P/B_{\text{mem}} \approx 295$ FLOP/byte 就是第 2 章的机器平衡点。取 $N = 15 \times 10^9$（ESM-2 的最大档【论文】[131]），$t_w = 30\,\text{GB}/3.35\,\text{TB/s} \approx 8.96$ ms【自算】：每个样本，无论多小，都要先付这 *9* 毫秒。50 个残基的短肽 $L = 50$，MFU 上界只有 17%；对另一假设的相同规模自回归模型，单请求单步 $L = 1$ 时上界

0.34%。这个比值是指定 BF16 稠密线性层模型下的 MFU 上界，不是 GPU 时间利用率；还忽略 attention FLOPs、激活流量、缓存与 kernel 开销。ESM-2 序列编码不是自回归 decode，不能把它的序列长直接当 decode batch。

长序列并不缺算术强度：在上述模型中 400 个位置已接近或越过机器平衡点。但新的浪费随之出现 —— padding。长度从 50 个残基的肽跨到 4000 个残基的复合物，若实现采用 padding 到批内最长位置，线性层有效工作量比例为 $\sum_i L_i/(BL_{\max})$，全注意力部分则近似 $\sum_i L_i^2/(BL_{\max}^2)$；不能对两者共用一个无条件百分比。长度分桶、packing 或 ragged 实现可以减少浪费，也会引入索引与调度成本。

输出侧还有一笔：科学服务要 JSON schema、SDF 价键、FASTA 字母表，工程上通常实现为 grammar-constrained decoding，每步只在合法 token 集合上重新归一化。JSON 尾部空格通常符合语法，不能把文本逐位差异等同于解析失败；FASTA 字母表或分子价键约束也只能保证部分合法性，不能保证生物活性或物理正确。

18.2 确定性：比长度更硬的分水岭

非确定性从哪里来。矩阵乘 $C = AB$ 的 K 维要被切成 tile 累加，切法由 (M, N, K) 与库的启发式共同决定。batch 组成一变，M 变，库可能换一套 tile，甚至启用 split-K 把 K 维拆给多个线程块再用原子加合并；浮点加法不满足结合律，归约顺序一变，结果的最后几位就变。cuBLAS 文档把边界写得很清楚：同一 toolkit 版本、同一架构、同样 SM 数的 GPU 上逐位相同，但跨 toolkit 版本不保证；而且一旦多 CUDA 流并发，这个保证不再成立，因为库会为并发流挑选不同的内部实现【官方文档】[132]。

误差不能靠固定“10^{-6}”概括。对标准浮点顺序求和，若无溢出且采用通常舍入模型，有绝对前向误差界

$$|\hat{s} - s| \le \gamma_{K-1} \sum_{i=1}^{K} |x_i|, \qquad \gamma_n = \frac{nu_{\text{round}}}{1 - nu_{\text{round}}}, \quad nu_{\text{round}} < 1. \tag{18.2}$$

这里 u_{round} 是单位舍入误差，FP32 约 2^{-24}；严重抵消时 $|s|$ 很小，相对误差可远大于 Ku_{round}。该界不是 Tensor Core 混合精度、并行树归约或 softmax 输出的完整误差界，$\sqrt{K}u_{\text{round}}$ 也只能在额外随机误差假设下作量级直觉。

微小 logit 差异只有在候选间隔足够小时才可能改变 argmax；是否影响科学结论要进一步用任务容差、统计重复、守恒量与物理验证判断。混沌动力学的逐点轨迹会分离，不意味着其统计量或科学结论必然不可复现。训练、推理、工具及分析脚本应分别记录版本和误差来源。

可以组合使用以下四类方法。

（a）**固定 batch 与固定形状**。所有请求补到同一个 B 与同一个 L，kernel 选择更容易固定；仍须锁定硬件、库版本、算法与并发流等，才能检验逐位一致。代价是等待与空转：低到达率下 GPU 为凑批闲置，补齐到 $L_{\max}$ 又浪费计算。对一次一个样本的科学负载，等待凑批与 padding 可能超过收益；可以用更小固定形状或明确的容差标准折中。

（b）**逐请求单独跑**。不做批处理，每个请求独占一次前向。减少跨请求批组成变化，但不保证所有并发与库算法都确定。权重读取主导、理想共享成立时，相对批量 B 的 decode 吞吐可能约为 $1/B$；具体 prefill/decode 损失必须测量，不能给无条件倍数。

（c）**接受非确定性**。允许实现选择更多优化，但吞吐是否最高仍需测量。只要研究预先定义并验证统计等价或容差标准，非逐位结果也可以支持科学结论。筛选阶段（几百万候选里挑几百个）的判据是阈值而非精确值，可以接受；报告阶段按问题需要做确定性复算或统计重复，并公开所采用的标准。把这两个阶段分开，是科学 serving 最实用的一条工程约定。

（d）**batch-invariant kernel**。把关键算子的归约划分固定为不依赖批组成，是减少跨 batch 数值差异的一条路线 [133]。支持范围、通信和 attention 后端限制依版本而变，不能把某套最新开关移植到旧 release 并假设同样有效。本书固定比较的 vLLM v0.11.0 文档说明离线复现设置，同时明确相同硬件与 vLLM 版本的边界，在线服务不保证可复现 [134]。这与“所有模型已经有 batch 不变模式”是不同的主张。

! 常见误解

「设了随机种子就能复现」是最常见的误解。种子只固定采样用的伪随机数序列；这里的不确定性来自浮点归约顺序，发生在采样之前。同一个 seed、同一个 prompt，只要 batch 组成不同，logits 就可能差 10^{-6}，argmax 就可能翻。固定 batch 与 kernel 也不能替代随机源控制：使用随机采样时，种子与随机数调用路径仍必须记录。判断一条流水线是否可复现，应共同检查随机源、batch、kernel、库/硬件版本与输出等价标准，`manual_seed` 不是充分条件。

18.3 三类 foundation model 的成本结构

生物：成本依序列模型结构和生成方式而变。ESM-2 最大 15B；ESMFold 用单序列替代 MSA，把 AlphaFold2 的数据库检索从关键路径上拿掉【论文】[131, 135]；AlphaFold3 用扩散模块替换结构模块，代价是每样本要跑多次去噪【论文】[136]。成本形态由长度决定。Evo 2 把上下文推到 10^6 token、单核苷酸分辨率【论文】[137]。作为假设全注意力自回归模型的对照，若 32 层、GQA 8 头、头维 128、BF16 KV，则每 token $2 \times 32 \times 8 \times 128 \times 2 = 131{,}072$ B，百万 token 要 131.072 GB KV【自算】。它不是 Evo 2 的实际内存测量：Evo 2 采用混合结构，卷积/状态空间层和注意力层需要分别记账，双向编码模型也不一定保留自回归 KV。把假设的全注意力账本套在任何“40B、百万上下文”模型上，会夸大或误置成本。单细胞一侧，scGPT 把同一套注意力成本搬到基因维度，把基因标记及表达信息编码为输入，实际长度取决于预处理、选定基因集和模型配置【论文】[138]。

材料：成本在消息传递的迭代与 kernel 启动上。GNoME 研究图网络辅助的大规模晶体结构筛选【论文】[139]；MACE 与 NequIP 用等变消息传递做力场【论文】[140, 141]；M3GNet 覆盖周期表【论文】[142]。一个 100 原子的晶胞、5 Å 截断、每原子约 40 个邻居，边数约 4000；hidden=256、4 层消息传递，FLOPs 约 $4 \times 4000 \times 256^2 \times 2 \approx 2.1$ GFLOP【自算】。按 300 TFLOPS 有效算力只要 7 μs，但一次前向要启动多个 kernel；若再假设 40 次不可重叠 kernel 启动、每次 3 微秒，则仅启动就需 120 微秒，明显超过 7 微秒理想计算时间【教学算例，非实测】。实际等变算子的张量积、邻居构建和访存都比这个线性混合近似复杂，不能由 2.1 GFLOP 直接预报 MACE 或 NequIP 延迟。这类小图配置可能受启动与不规则访存限制，较大图或更复杂等变计算则可能转为计算受限。拼批可能摊薄启动，但不同原子数需考虑图打包、索引或 padding；图实现不一定补到最大图；投机解码则没有对象，因为输出不是 token 流。

气候：显存容量、网格/图操作与计算共同约束。GraphCast 在 0.25° 网格上做 10 天预报，论文在单张 TPU v4 上报告一分钟内完成【论文】[143]；Aurora 在百万小时级数据上预训练，可微调到空气质量、海浪与台风路径【论文】[144]。0.25° 网格是 $721 \times 1440 \approx 1.04 \times 10^6$ 个格点，70 个变量、FP32 一个场就是 290 MB；hidden=512 的节点特征一层就是 2.1 GB，若再假设某模型有 3600 万参数，FP32 权重约 144 MB【自算】；这是用于比较激活与权重尺度的假设账，不是 GraphCast 或 Aurora 完整实现的逐项显存测量。激活是权重的十几倍，于是「多放一个集合成员」直接等价于「多一份激活」：每个成员的工作集按 4.5 GB 计（状态 290 MB 加两层节点特征），50 个成员就是 225 GB，可分到多张卡独立执行，或顺序执行来换时间，不必把单个成员强行跨卡切分。若实测确认权重占比很小，单独量化权重的容量收益也小——权重本来就不是瓶颈；投机解码也没有对象——输出是一整片场。拼批在这里退化为数据并行：50 个成员就是 50 倍激活，先撞显存墙。

三类例子说明，不能只从参数量判断瓶颈，必须测量工作集、算子形状、激活与执行开销。不以自回归 token 为输出的模型，不能直接套用 token 投机；权重占比低时，单独量化权重的收益也受限。激活量化、图打包或减少迭代仍可能有效，需另行验证。表 18.2 把三类的成本结构并排，最后一列是各自的失效点。

18.4 agentic science：工具调用占空比

科学发现正在从「一次前向」变成「多轮假设—验证循环」：模型生成假设，调用模拟器或数据库，读回结果，再改假设。对 infra 而言，这次变化的关键不在模型更强，而在一次请求内部变成串行。第 k 轮依赖第 $k-1$ 轮的工具返回值，同一会话的依赖轮次不能并行计算，但不同会话的就绪步仍可交错或拼批；而工具调用是外部阻塞的，等的是一个进程、一个排队系统或一次实验，不是显存。如果资源被该会话独

表 18.2 三类科学 foundation model 的成本结构。「失效的传统优化」一列说明的是机制，而不是性能倍数：每一类都至少有一项消费级主力手段在这里失去作用对象。

类别与代表	单次请求的形态	主导成本项	失效的传统优化
生物：ESM-2、ESMFold、AlphaFold3、Evo 2、scGPT	一条序列、一个基因组窗口、一个细胞	按编码、混合架构、生成与扩散分别记账	非生成模型不适用 token 投机；生成模型须检验采样/复现契约
材料：GNoME、MACE、NequIP、M3GNet	一个晶胞、一帧构型	邻居与等变图算子；kernel 启动/访存/计算	非自回归，无 token 可投机；补图浪费
气候：GraphCast、Aurora	一个初始场、一个集合成员	10^6 格点激活；显存容量	权重量化的收益受占比限制；非 token 输出不直接投机

占保留且没有其他就绪工作，GPU 会在这段时间空闲；可复用服务不必如此。

这也是 test-time compute 在科学场景里的形态：不是多生成几千个 token，而是多跑几轮模拟——前者能被 batching 摊薄，后者每轮都要等外部工具返回。

定义工具调用占空比

$$\eta_{\text{tool}} = \frac{t_{\text{tool}}}{t_{\text{tool}} + t_{\text{gpu}}}, \qquad U_{\text{eff}} = (1 - \eta_{\text{tool}})\, U_{\text{batch}},$$
$$N(\eta_{\text{tool}}) = \frac{N_0}{1 - \eta_{\text{tool}}} \quad \text{(每份资源仅保留一个会话)}. \tag{18.3}$$

η_{tool} 是一个纯观测量：在一段墙钟里，工具阻塞时间占多少。它把「agent 慢」翻译成容量问题——只有每个会话独占保留一份 GPU 资源、工具等待期间也不释放时，维持任务吞吐所需保留资源才按该比例膨胀。单纯 batch=1 不意味着禁止在会话之间复用。

▶ 工具占 50% 时需要多少额外 GPU

设目标吞吐 $R = 60$ 任务/小时，每任务 GPU 时间 $t_g = 600$ s、工具时间 $t_t = 600$ s，故 $\eta_{\text{tool}} = 0.5$，单任务墙钟 1200 s。

基线（$\eta_{\text{tool}} = 0$，单会话独占）：单卡 3600/600 = 6 任务/小时，$R = 60$ 需 10 卡。

$\eta_{\text{tool}} = 0.5$，**单会话独占保留：**单卡墙钟翻倍，3600/1200 = 3 任务/小时，需 20 卡。由式 (18.3)，$N = N_0/(1 - \eta_{\text{tool}}) = 10/0.5 = 20$，即**多 10 卡、+100%**。新增保留容量让不同任务并发执行；每任务 GPU 工作量仍为 600 秒，等待工具的 600 秒也没有被缩短。

若允许会话之间复用：单卡常驻 C 个会话即可把空闲填满，条件是 $C(1-\eta_{\text{tool}}) \geq 1$，即 $\eta_{\text{tool}} = 0.5$ 时每卡只需 2 个会话，10 卡仍然够，额外 GPU 为 0。代价转移到了别处：常驻会话翻倍（每卡 2 份 KV），单任务墙钟仍是 1200 s。若 SLO 要求 900 秒内完成，单任务 600+600 秒的依赖链已不满足：回到 20 张卡也无效。须减少 GPU 关键路径、把工具时间压到 300 秒以内，或修改任务算法与 SLO。

还有一个更硬的上限：若工具端本身只有 40 任务/小时的容量，则 $R = 60$ 不可达，此时加多少卡都无效——瓶颈已经完全离开 GPU。

这个算例区分了计算需求、保留需求和端到端时延。工具占空比 0.5 时，10 张卡的 GPU 计算能力原则上足够，但独占预留策略要 20 张；在理想错峰下允许每卡两会话即可填空。会话到达随机、GPU 阶段重叠、KV 转移和工具容量都会使理想结果变差。主请求每个完整 KV 约 1.51 GB，能常驻多少会话应按实测显存账判断，不能凭“科学负载”统一断言只能 1–2 会话。图 18.1 画出有限会话容量下的理想保留资源模型。

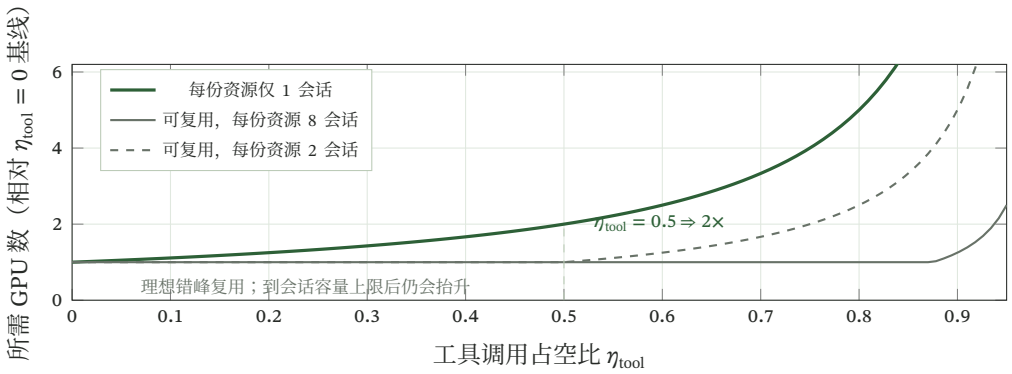


图 18.1 工具调用占空比与所需 GPU 数的关系。独占一个会话时保留资源曲线沿 $1/(1-\eta_{\text{tool}})$ 发散，$\eta_{\text{tool}}=0.5$ 就要两倍卡；能在会话间复用时，空闲可被其他就绪会话吸收，曲线是平的，直到 $C_{\max}(1-\eta_{\text{tool}})<1$ 才开始抬升。两条会话复用曲线为 $\max\{1, 1/[C_{\max}(1-\eta_{\text{tool}})]\}$，假设理想错峰且忽略切换。它不保证延迟 SLO，也不把工具时间当成可由 GPU 加速的部分。

18.5 可复现性的工程实践

把上面的分析落成清单，每一项对应一个可观测的量。

1. **锁权重与镜像**。记录权重文件的 SHA256、模型 revision、容器 digest，以及 CUDA/cuBLAS/NCCL 版本。cuBLAS 的逐位保证以 toolkit 版本相同为前提【官方文档】[132]。
2. **锁 GPU 型号与数量**。同代不同 SKU 的 SM 数与频率不同，cuBLAS 的保证要求同架构、同 SM 数；报告里写「H100」不够，要写到 SXM/NVL/PCIe。
3. **记录并控制 batch 与调度**。对逐位目标，固定批组成、形状、并发与算法，并做实际双跑验证；若版本提供 batch 不变算子，记录支持范围。容差或统计目标可接受更多动态策略，但应量化差异。
4. **关掉会改变数值的加速项**。TF32、split-K、多流 workspace 都会改最后几位；对应地把 matmul 精度设为 ieee、设 `CUBLAS_WORKSPACE_CONFIG`。
5. **记录实际启动的 kernel**。同一次前向应看到同一组 kernel 名与同样的 `num_splits`；换 backend 或驱动，这组名字就会变。
6. **在输出层定义等价，而不是在 bit 层**。结构用坐标容差或 RMSD，离散输出记录 argmax 与 margin；按任务与精度定义 margin 阈值，对低间隔样本作稳定性检查，单独统计，而不是当成错误。
7. **做双跑与跨机校验**。若合同是逐位一致，同机重复应逐位相同；容差与统计合同则按各自检验，跨机型跑一次并记录差异分布，后者决定能否迁移。

报告时还要写明属于哪一档：逐位可复现、容差内可复现，还是统计等价。三档对应三种实验效力，混用会让「复现失败」变成无法归因的争论。

未解问题

1. **确定性 kernel 的性能代价仍需跨组合验证**。某些确定性实现固定归约或限制算法选择，但限制项随版本和后端不同，不能列成所有实现共有的开关；公开材料也不足以替代本地测量。形状空间很大，常见标称覆盖有限，而一部分科学负载落在瘦长 GEMM 与小 batch 等边角形状。
2. **工具与 GPU 调度状态仍需协同**。模拟器是外部进程，GPU 调度器看不见它的队列长度与预计返回时间，因此无法做全局的占空比优化。若两侧没有状态接口，GPU 侧可能只看到会话暂停或变慢；已有编排可以传递部分状态，但准确预测工具返回仍依负载。要解决它，需要一个跨 GPU 与工具的统一排队模型，而两侧的调度目标并不一致。
3. **科学输出的等价标准依任务而变**。结构预测可用 RMSD，但序列设计、晶体稳定性、气候场不能共用一个数值阈值；需要领域内预先定义的容差、统计与物理检验，才能自动验收。

第 19 章　开放问题、前沿方向与学习路线

本章结论

(1) 本章列出的十四条开放问题可以归为四组：物理与算法上限、机制与数值、系统与调度、经济与计价。判断一条问题适合什么验证路径，可以先看「为什么今天没解决」这一栏——理由涉及带宽或功率，可能需要算法、硬件和工作负载共同变化；涉及模型、证明或语义，需要明确问题范围；涉及经济性，则还依赖价格、需求和风险。分类帮助选择验证方法，不承诺一篇论文即可解决。

(2) 学习路线必须让论文、源码、实验三层指向同一个问题。只读论文会得到错误的性能直觉，只读源码会得到无法迁移的局部知识，只做实验则不知道自己究竟在验证什么。三层的毕业标准分别是：能自己算出一张 roofline 与 KV 账本、能判断一个设计的瓶颈迁移、能读懂当月的标题在解决什么。

(3) 全书框架可压缩为一句话：推理加速可以减少计算、数据移动或等待，也可以增加复用与并发；代价须在精度、尾延迟、通信和复杂度中完整记账。新技术出现时不必追新闻，只需问它动的是哪一段。

19.1　十四条开放问题

一条问题值得写进这一节，必须满足两个条件之一：它改变了某一层的损益表结构，或者它让今天的工程实践缺少可判定的对错。按这个标准，纯粹的性能打磨（例如把某个 kernel 再挤快一点）不入选，而「没有理论」「没有语义」「没有确定性」入选，因为它们决定了优化能否被复现与迁移。

表 19.1　十四条开放问题及其「为什么今天没解决」。第三列只给方向，不给承诺；凡涉及数字的判断均在正文或脚注给出可复算的依据。

编号与问题	为什么今天没解决	可能的突破口
P1 稀疏注意力在 prefill 也受算力限制时能否不掉点	稀疏块可减少计算与 KV 读取，但索引、路由、gather 和不规则负载也有成本；部分方法先压缩或用廉价索引选择，不一定扫描全量 KV。长 prefill 下计算与访存的主约束还会改变 [16, 145]。	把块路由与前一层矩阵乘融合，或从预训练阶段就稀疏化。
P2 MQA/GQA 的容量与训练稳定性边界	共享 KV 改变表达容量与优化路径；一些 Transformer 训练研究讨论 logit 增长与数值稳定，但不能仅据此断言 MQA 必然不稳定。架构、初始化、归一化和训练数据交织，使消融昂贵 [146, 25]。	小规模代理实验加谱分析，把不稳定钉到具体的初始化与归一化组合上。
P3 PD 分离的 KV 能否走光互联或 CXL 池化	KV 是「大块 + 低延迟」的混合负载：光互联省距离与功耗，但要付光电转换的固定延迟；CXL 给的是 load/store 语义，而 KV 传输要的是大块 DMA 与一致性。是否回本取决于容量复用、传输、硬件成本与隔离，不只由 P/D 配比决定。	比较页访问与消息传输的真实成本；对可安全共享的前缀研究多播与远端复用。
P4 投机解码的接受率与验证成本预测	接受率依模型对、数据、采样策略和上下文而变，并非随长度必然坍塌；验证也有额外计算、KV 写入与调度代价。缺少可跨负载迁移的接受率/成本联合预测模型。	draft 训练数据按长度分层采样；在线估计 α 并自适应调 γ。
P5 量化目标与投机校正的质量边界	精确拒绝采样允许 draft 与 target 分布不同；若实际概率和残差采样正确，保证相对已部署 *target* 的分布，而不是原始高精度模型。目标量化误差、有限精度实现和接受率变化需分别测量。	核验概率计算/采样校正；分别报告目标量化质量与投机引入的偏差，避免用容差相等替代精确校正。
P6 MoE 的 all-to-all 能否被拓扑彻底消除	路由是数据相关的，拓扑只能降低单次通信成本，消不掉「专家不在本地」这一事实；通信路径与跳数依专家放置和 fabric 拓扑而定，不能仅由 EP 度数推出 [44]。	让路由决策知道专家放置（路由与 EPLB 联合优化）；层次化专家先机架内再跨机架。
P7 可迁移的 goodput 调度保证	goodput 依赖到达过程、长度分布、SLO 与 batch 组合，是带约束的随机优化；排队论也能研究尾部，但动态批处理、长度相关与缓存状态使简单模型难以校准。	在流体极限下求最优控制；或先给仿真下界，再证启发式的近似比。

编号与问题	为什么今天没解决	可能的突破口
P8 SLO 感知的自动配置	配置空间（并行度 × batch × chunk × 量化 × 副本数）组合爆炸，每个点都要在真实硬件上测；模拟器精度不足以替代实测；测量成本与环境变化限制了搜索覆盖。	性能模型加少量在线 probe 的贝叶斯优化，把搜索降级为校准。
P9 异构芯片的算子语义统一	语义不止算子签名，还包括舍入模式、NaN 传播、atomic 顺序与 stream 语义，各家定义不同；编译器能吃掉算子内优化，吃不掉库实现质量与调试工具。	定义可移植语义子集，其余显式降级；把数值差异写成契约而非隐含假设。
P10 推理的确定性与 batch 不变性	浮点加法不满足结合律，split-K 的归约顺序与 attention 的 tile 划分随 batch 变化，同一请求在不同 batch 下 logits 可能不同 [147]；固定归约可减少差异，但性能代价取决于算子和形状。	batch 无关 kernel（固定 tile 与 split），把确定性做成可售的 SLO 维度。
P11 长上下文的经济性上限	全注意力的 $O(L^2)$ FLOPs 与 KV 的 $O(L)$ 容量共同增加成本；实际可用上下文受任务、检索位置与模型能力影响，不能只凭标称窗口判断收益 [33, 148]，为极少数超长请求付全量容量成本并不划算。	分级上下文与检索式记忆；把「有效上下文」做成可测、可定价的指标。
P12 test-time compute 的反向要求	同一请求依赖链不可并行，但不同会话的就绪步可 batch，P/D 分离也可能继续提高资源复用。需要同时优化单任务时延、跨任务并发和质量 [149]。	把思考做成可中断会话、中间状态持久化，让多请求的思考步交错执行。
P13 功率墙：单柜功率上限成为硬约束	GB200 NVL72 单柜 72 颗 GPU [78]；若仅作每 GPU 1.2 kW 的设计功率假设，GPU 侧为 86.4 kW【自算，非整柜实测功耗】，会对传统低功率机柜提出供电与散热改造要求。接入周期取决于当地电网、站点与审批条件，不能套用统一年数。	液冷与高压直流；把功率作为调度的一等资源；地理调度到有电的地方。
P14 推理的碳成本是否计入定价	碳强度按小时与地区波动，token 定价却是静态的；PUE 与 WUE 的核算口径不统一；客户与地区的披露、采购和监管条件不同，尚缺可跨产品直接比较的逐任务核算口径。	碳感知调度，把 batch 挪到低碳时段与地区；把碳强度做成 API 的一个参数。

表 19.1 里最容易被误读的是 P1 与 P11：它们看起来是算法问题，但收益能否兑现取决于计费口径。即使按 token 计价，也可通过上下文档位、缓存与输入输出价格部分反映成本；这些价格仍不能替代对真实工作负载的核算，客户买的是「能不能用」，而供应方付的是「容量按峰值备」。这也是为什么第 9 章的成本模型必须先于算法讨论——成本与计价边界会影响配置取舍。

19.2 依赖关系：哪些问题在等别的问题

十四条不是并列的。物理层的问题决定可行域：如果单柜功率封顶，单柜内“多堆卡换吞吐”受限；扩机柜还受网络、电力与成本约束，硬件规划会约束 P1 与 P11 的候选方案。机制层的问题决定能否走到可行域的边界：P2、P5、P9、P10 影响可行方案与数值保证。系统层把前两者的结论翻译成可部署的形态，经济层决定值不值得走到边界。

图 19.1 是研究规划的提示，不是严格依赖图。物理、算法、调度与经济性往往相互反馈：改变稀疏算法可能减少带宽需求，改变服务期限也可能避开峰值容量。每个研究问题都应明确可控变量、基线、失效条件与测量边界。

19.3 三个深水区

19.3.1 接受率能不能被预测

投机解码 [60] 的加速比有一个闭式表达，但它的输入——逐 token 接受过程——依模型和负载而变。这一族方法的演进恰好可以用「草稿从哪来」串起来：从独立的 draft 模型 [60]，到在特征层做草拟的 EAGLE [61, 62]，再到不另设草稿模型、直接在目标模型上加解码头并行出多个候选的 Medusa[64]。三条路线的训练成本、验证结构与接受规则不同；下面的链式简化公式不能直接覆盖所有树形候选或实现。



图 19.1 十四条开放问题的分层依赖。箭头表示约束自上而下传导：功率与带宽封顶决定机制层还剩多少设计空间，机制层的缺失决定系统层能否被调优，经济层决定值不值得走到边界。反向的影响同样存在——价格变化会重新打开机制层的问题，所以这一排序描述的是当下，而不是永久。

▶ 投机解码的盈亏平衡接受率

设 draft 长度 γ，每个 draft token 的成本是 target 单步的 c 倍，各位置在前缀已接受条件下的接受概率同为 α，且验证耗时恰等于一个 target 单步、忽略额外通信与调度。一次验证步的期望产出为

$$E = \sum_{i=0}^{\gamma} \alpha^i = \frac{1-\alpha^{\gamma+1}}{1-\alpha},$$

其中第一项是必得的一个 token，第 i 个 draft token 需要前 i 个都被接受，概率为 α^i。成本为 $1+\gamma c$，于是

$$S(\alpha,\gamma,c) = \frac{1-\alpha^{\gamma+1}}{(1-\alpha)(1+\gamma c)}.$$

取 $\gamma = 5$、$c = 0.1$，令 $S = 1$：

$$1-\alpha^6 = 1.5(1-\alpha) \quad\Longrightarrow\quad \alpha^* \approx 0.33426.$$

含义：接受率跌到约三分之一，投机解码就不再赚钱。这是给定成本假设下的代数阈值，不是实测接受率。若验证成本为 v 倍 target 步，分母应改 $v+\gamma c$；$\alpha = 1$ 时用极限 $E = \gamma + 1$。判断 128k 下的收益需要测接受过程与验证/草稿成本，而非预设接受率随长度下降。

这条问题的难点在测量而非算法：要拟合 $\alpha(L)$，需要同一模型在多个长度上的接受率曲线，而每一条曲线都要一次完整的服务压测。公开实验的模型、长度和采样设置覆盖有限；应保存按位置和长度分组的接受率、验证耗时及质量，而不只报最终加速比。

19.3.2 确定性的代价是多少

浮点加法不满足结合律，而 split-K 的归约长度、attention 的 tile 划分都随 batch 变化，所以同一请求在 batch=1 与 batch=32 下可能得到不同的 logits [147]。这可能是合法浮点算法选择的结果，不必然是程序缺陷；固定归约与执行条件可降低这种差异，也应测量其性能影响。

工程界已讨论 batch 不变的 kernel 路线 [133]，但是否支持某模型、精度、GPU 与 attention 后端，应回到具体 release 验证。固定 batch 与归约只是条件的一部分；跨硬件、跨库版本的逐位复现不由 seed 单

独保证。对科学负载，先决定逐位、容差或统计等价哪一档足够（第 18 章），再测性能代价；聊天、代码与工具调用也可能有复现需求。

19.3.3 串行思考与跨请求并发

▶ 单任务依赖链与跨任务吞吐

一次任务需 k 轮串行生成，每轮 t 个 token，忽略 prefill/工具等待时，单流 decode 速率 r 给出下界 $T = kt/r$。该任务的后续轮次不能提前并行，但其他任务可以在同一副本交错或形成连续 batch。只有额外要求这个副本独占一个会话、并且权重读取主导、batch 为 B 时吞吐近似线性增加，单会话相对多会话吞吐才约为 $1/B$【条件模型】。这不是 test-time compute 在所有服务中的必付税。工具等待、长 KV 与 deadline 会减少可复用并发，也可能让 P/D 资源拆分更有价值，需实际测量。

推理时计算可以改善某些任务质量 [149]，比较单位应同时包含成功任务、每任务 token 和端到端时延。更多 token 若提高成功率，也可能减少外部重试；若只增加无效思考，则增大成本与尾部。用相同质量目标评估，比给所有思考请求统一乘 $1/B$ 更合理。

19.4 论文阅读路线：按问题分三层

不要按时间顺序读，按问题顺序读。表 19.2 给出三层选读；每层有一个明确的自检问题，可以据此回查前置概念。选篇的标准不是「引用量高」，而是**每篇都对应本书某一章的推导**——读完它，那一章的公式应该能自己重推一遍。

表 19.2 从入门到前沿的阅读路线。第二列是「读完能回答的问题」，也是检验有没有读懂的唯一标准；凡答不上来，说明还没读到实验条件那一节。

文献	读完能回答的问题
第一层 · 必读（毕业标准：能独立算出一张 roofline 与一份 KV 账本）	
Orca [57]	静态 batch 为什么浪费算力；迭代级调度把它拿回来多少。
PagedAttention [2]	KV 浪费的来源；分页与前缀共享为什么正交（第 7 章）。
FlashAttention [1]	为什么 attention 不必物化 L^2 中间矩阵；IO 复杂度与 FLOPs 的区别（第 3 章）。
FlashAttention-2 [10]	同样的 FLOPs 为何还能再快一截；work partitioning 动的是哪一段。
Fast Transformer Decoding [6]	MQA 用表达力换什么；GQA [7] 如何在两者之间连续插值。
DeepSeek-V2 [9]	MLA 如何把每 token 的 KV 压一个量级；解耦 RoPE 为什么不能被吸收。
Sarathi-Serve [58]	chunked prefill 如何改造 TTFT 与 ITL 的权衡；stall 从哪里来。
Scaling Transformer Inference [150]	batch 上限由什么决定；KV 量化省的是容量还是带宽。
第二层 · 进阶（毕业标准：能判断一个设计的瓶颈会迁移到哪一层）	
DistServe [67]	PD 分离的收益边界；什么负载下拆开是负收益（第 7 章）。
Mooncake [69]	KV 池化在集群尺度怎么做；中心调度器的状态从哪来（第 11 章）。
DeepSeek-V3 [8]	671B 的 MoE 如何组织专家与并行布局；EP 布局与通信如何重叠（第 5 章）。
GPTQ [29]	4 bit 权重为什么还能保精度；二阶近似的假设何时失效（第 4 章）。
FlashAttention-3 [11]	Hopper 上 attention 的新瓶颈是什么；异步与 FP8 各自动哪一段。
DeepEP [44]	all-to-all 的工程上限在哪；dispatch 与 combine 怎么压延迟（第 12 章）。
第三层 · 前沿（毕业标准：能读懂当月标题在解决什么问题）	
DeepSeek-V3.2 [17]	稀疏注意力落到生产模型上的精度代价与工程代价。
EAGLE-3 [63]	特征草拟如何训练并评估接受率；训练时的 scaling 是否可迁移。
NVFP4 [28]	4 bit 的数值边界在哪；块缩放为什么是 FP4 可用的前提。
Test-Time Compute [149]	「多想一会儿」何时优于「换个更大的模型」，以及它的单位是什么。

! 常见误解

读论文最常见的方式错误是「按时间顺序通读」，结果是先读了一堆 2022 年的调度文章，却还不知道 roofline 是什么，于是无法判断那些文章到底在优化什么。正确的顺序是**先建立物理约束（第 2 章），再读机制**——有了 I^* 和 $kL\ell$ 这两个量，一篇论文的关键实验条件（模型、硬件、batch、长度、精度、框架）会在摘要里自己跳出来。

读法上还有一条经验：第一遍只读摘要、图 1 与实验条件表，判断结论是否适用于你的场景；第二遍读方法，把它的假设写成一句话；第三遍去源码里找对应实现，或至少复现一个数字。绝大多数「加速比争议」来自实验条件不同——模型、硬件、batch、长度、精度、框架版本，六项里缺任何一项，那个数字就不可比。

19.5 源码阅读路线：固定版本与符号入口

源码路线与第 8 章同一基线：vLLM v0.11.0（commit `b8b302c`），SGLang v0.5.3.post3（commit `85ebeec`），访问 2026 年 10 月 22 日。完整 40 位 commit、选定文件、URL 与校验和在 `evidence/engine-sources/manifest.json`。这些是可复查的比较基线，不宣称截至修订日最新；路径或机制变化时应新建对照，不把旧行号贴到最新分支。

1. `vllm/v1/core/sched/scheduler.py` 中的 `Scheduler.schedule()`：本步哪些请求获分配多少 token。把 prefill/decode 当成请求工作量，而非两个永久队列的直觉，需与实际分支逻辑一起读。
2. `vllm/v1/core/kv_cache_manager.py` 中的 `get_computed_blocks()`、`allocate_slots()`：命中块何时被找到，分配失败如何反馈调度。再查 `single_type_kv_cache_manager.py`，避免把支持多种 KV 结构的接口当成一种实现。
3. `vllm/v1/core/kv_cache_utils.py` 的 `hash_block_tokens()` 与 `block_pool.py` 的 `BlockPool`：哈希键如何串联前缀、空闲/复用块怎样管理。模型、LoRA、多模态或租户相关额外键也会影响可安全共享范围。
4. `vllm/v1/engine/core.py` 的 `EngineCore.step()`，再到 `vllm/v1/worker/gpu_model_runner.py`：读调度输出如何进入执行、何时返回结果，不把 Python 调用次数当 GPU 步时长。
5. 该 vLLM 版本的 attention 接口位于 `vllm/attention/selector.py` 与 `vllm/attention/backends/abstract.py`。原稿的 `vllm/v1/attention/backends/registry.py` 不属于这一版本；选择器、缓存布局与 kernel 支持矩阵共同决定后端。
6. SGLang 从 `python/sglang/srt/mem_cache/radix_cache.py` 的 `RadixCache.match_prefix()`、`insert()`、`evict()` 出发，再读 `managers/schedule_policy.py`：共享前缀树、锁引用与可驱逐节点如何影响选择。
7. 对照 SGLang 的 `managers/scheduler.py` 中 `event_loop_normal()`、`event_loop_overlap()` 与 `managers/overlap_utils.py`，最后读 `model_executor/model_runner.py`。重叠调度省的是可重叠的 CPU/GPU 等待，并非逻辑依赖自动消失。

先读调度输出与 KV 数据结构，再带主请求追踪：4096 输入前缀若命中一个完整块，节约在哪，最后非完整块如何处理，剩余 512 输出的 KV 何时释放。以函数名和固定 commit 定位比引用漂移行号更稳妥；选定源码仅供只读分析，不在本任务中部署生产服务。

19.6 一台 Mac 上能做的三个实验

以下是可继续执行的实验方案，不表示本次已经运行 *llama.cpp* 或 *GPU* 引擎。本次实际运行的资源/成本、独立数值参考和合成请求 trace 检查，另在本章实验记录中给出命令、配置与结果。Mac 结果只用于验证模型或机制。

方案一：用 llama.cpp 检查权重读取与量化边界。固定 commit、模型权重 SHA-256、tokenizer、

F16/Q8/Q4 量化格式、Metal 后端、线程、GPU 层数、电源模式与热状态。在相同 prompt 与输出长度下，运行该版本 `llama-bench --help` 确认选项，再做多次预热和重复；如采用 `llama-bench -m MODEL.gguf -p 512 -n 128 -r 5`，将完整输出和实际命令保存。比较 decode 速率 r_d 与权重文件大小 S 的关系，Sr_d 只是假设每步读全权重的表观带宽，不包含 KV 与激活，可能受页缓存和分层 offload 影响。

待检验假设是某些小 batch、短上下文、权重读取主导配置下，量化可能减小带宽时间；不承诺达到本机带宽的固定百分比，也不承诺 prefill 各量化档相近。反量化、算子实现与质量差异都需报告；不能由文件变小直接推出等比例加速。

方案二：分页内部碎片的标准库仿真。固定随机种子和明确的请求长度分布，比较按最大长度预留、按实际长度连续分配、按 b 个 token 分页的分配量。分页内部浪费为 $\sum_i [b\lceil L_i/b\rceil - L_i]$，对 $b = 16$ 和 128 分别统计；$L = 0$、恰整块、整块加 1 与很短请求为边界用例。平均浪费约 $b/2$ 只在长度余数接近均匀时成立，对整块长度可为零，短请求相对浪费也可能很高。它不仿真外部碎片、前缀共享、分配器锁或 kernel 性能，不能用仿真百分比直接声称重现 PagedAttention 的服务加速 [2]。

方案三：成本和达标率敏感性。使用本书主例脚本，分别扫描资源单价、有效输出吞吐、平均功耗、PUE 和长度；每次只改变一个参数并保持分母语义不变。区分满载吞吐乘时间覆盖率与完整观察窗的 goodput，避免再乘一次利用率。用独立手算检查一组值、零产出不能计算单位成本、含电租价不得再加电费等边界。成本对利用率的 $1/u$ 关系依赖固定成本与活跃效率不变，模型规模也不必使任务成本线性变化。

! 常见误解

Mac 上跑出的数字不能外推到数据中心。统一内存带宽与 HBM 差一个数量级，macOS 的 mmap 与页缓存还会污染测量。这三个实验的合法用途是验证比例关系与机制，不是给出绝对性能；把 Mac 的 token/s 写进容量规划，是本章最容易被犯的错误。

19.7 本次已经运行的最小验证

本节记录 2026 年 10 月 22 日在 macOS arm64、Python 3.9.6 上实际执行的检查。程序只用 Python 标准库，配置、代码、命令与 JSON 结果随书提供。执行环境没有 NVIDIA GPU，也没有安装本节所比较的两个推理引擎；因此这些结果不支持任何 GPU 吞吐排名。它们分别验证资源账本、数值递推与指标计算。

在 `manuscript/` 目录运行：

实际执行入口；结果写入 `experiments/results/`

```
/usr/bin/python3 experiments/resource_cost.py
/usr/bin/python3 experiments/kernel_math_check.py
/usr/bin/python3 experiments/check_trace.py
```

验证一：把同一请求算到成本分母。配置来自 `experiments/case_config.json`，与前言的 70B GQA 合同一致。全模型每 token 的 KV 为

$$k = 2 \times 80 \times 8 \times 128 \times 2 = 327680 \text{ byte} = 320 \text{ KiB}.$$

权重占 $70\times10^9\times2 = 140$ GB。四卡聚合可分配空间减去权重和运行空间后，KV 预算为 $4\times80\times0.9-140-4\times8 = 116$ GB。预留 4608 个位置的单请求需要 1.40625 GiB，故纯容量上限是 $\lfloor 116\times10^9/(4608\times327680)\rfloor = 76$。程序用独立整数参考核对主数值，并检查第 76 个可以容纳、第 77 个超预算。这一边界没有证明 76 路请求满足 TTFT 或 ITL；TP 分片不支持时程序直接拒绝计算。

PD 传输使用 prefill 的 4096 个 token，负载为 1.25 GiB。聚合 100 Gbit/s 瓶颈乘 0.7 有效载荷比例后，字节下界为 0.15339 s；协议、排队和布局转换会增加时间。资源费用 12 USD/h，加上 $4\times0.35\times1.2\times0.10 = 0.168$ USD/h 电费，得到 12.168 USD/h。在假设合格输出 goodput 为 480 token/s 时，

$$c_{\text{out}} = \frac{12.168}{3600 \times 480} \times 10^6 = 7.041667 \text{ USD/百万合格输出 token}.$$

此数未计人力、网络、存储、税费和冗余容量。若资源单价已经含电，必须删除额外电费。将同一组达标

请求的输入也纳入分母，会变成 0.782407 USD/百万输入输出 token；这只是口径变换，不能作为系统变快的证据。

表 19.3 成本敏感性【脚本自算】。每格为 USD/百万合格输出 token；电费保持 0.168 USD/h。goodput 与单价在此独立变化，仅用于解释模型。

每卡资源分摊价	240 token/s	480 token/s	720 token/s
1.5 USD/(GPU·h)	7.139	3.569	2.380
3.0 USD/(GPU·h)	14.083	7.042	4.694
4.5 USD/(GPU·h)	21.028	10.514	7.009

表 19.3 说明固定小时成本下分母的作用：goodput 减半会使单位成本翻倍，但不能由此假设硬件扩容后 goodput 必然线性增加。脚本还扫描输入 2048、4096、8192、32768 token（输出均 512），BF16 KV 的纯容量并发上限分别为 138、76、40、10。将每个 KV 元素从 2 byte 改为 1 byte、暂不计量化元数据时，对应上限为 276、153、81、21。这些是一项变量变化的容量账，不保证该量化格式有可用 kernel，更未验证质量或 SLO。

验证二：在线 softmax 的独立数值参考。`kernel_math_check.py` 以在线最大值与归一化和递推实现分块 softmax，对照 70 位 Decimal 直接指数求和参考，固定种子 20261022。71 组检查包含单元素、最后一个不完整块、极端 logits，以及不同因果可见前缀。最大绝对误差为 4.44×10^{-16}，通过预设的 10^{-12} 阈值。这支持第 3 章递推在这些输入上的数值正确性，不等于对所有浮点输入的证明，也不验证 FlashAttention GPU kernel 的内存流量或速度。

验证三：一个平均值如何掩盖一次卡顿。`synthetic-trace.jsonl` 是五条人工构造的请求，分别覆盖正常请求、TTFT 超限、ITL 尖峰、质量失败和响应失败；观察窗为 10 s。全部已接收输出为 20 token，其中响应成功的输出为 16 token；只有正常请求的 4 token 同时满足成功、质量和逐请求 SLO。因此原始输出速率为 2 token/s，成功输出速率为 1.6 token/s，合格输出 goodput 为 0.4 token/s。卡顿样例的平均 TPOT 为 40 ms，最大 ITL 却为 80 ms，超过本书的 50 ms 门槛。仅凭平均 TPOT 会错误接纳它。

检查器还验证单输出 token 的 TPOT/ITL 无定义，拒绝把 chunk 到达时刻冒充逐 token 时刻，并检查时间戳与观测窗口边界。合成轨迹只用于检验计数和筛选，不能代表真实负载的尾部。原始结果、断言与重算入口均在 `experiments/README.md` 中；`results/run-manifest.json` 保存实际命令和源文件校验和，修改配置后应重跑。

19.8 在 GPU 上接续验证的合同

`experiments/run_benchmark.sh` 提供两个固定版本客户端的烟雾检查，只向已经运行的服务发请求，不启动服务。它保存包版本、硬件信息、命令和官方 JSON；模型 revision、tokenizer、服务端精度、TP 布局和启动参数仍需实验者记录。vLLM v0.11.0 与 SGLang v0.5.3.post3 的参数依据本书固定提交核对，不能将独立的随机负载当成公平的跨引擎对比。

正式比较应重放同一份 token 与到达轨迹，分别测试定长和混合长度、冷/热前缀、开环到达与闭环并发，再逐项消融 prefix cache、chunk、投机和 PD 分离。每组预热后至少测量 10 分钟、使用至少三个种子，并等待所有请求终止或另报删失样本。完整矩阵、字段和限制写在实验 README。同时保留错误、质量评分、逐请求 TTFT/E2E、最大 ITL 和各类分位数，才能计算本书的联合 goodput。随机 token 数据只能压测性能，不能证明任务质量；只有 chunk 时间戳时应报告 chunk gap。没有原始 token 时刻和逐请求质量信息时，官方汇总吞吐不能反推出本书的严格 goodput。

• 读者练习：把“更快”写成可否证的结论

任选一个功能，在同一模型与负载上写出它预期减少的计算、数据移动或等待；列出一个可能抵消收益的条件，并指定需要保存的计数器或时间戳。先用资源脚本判断容量可行，再用同一轨迹比较质量与 SLO。如果只得到 kernel 速度或单次成功案例，结论就应停留在相应证据范围。

19.9 把全书接回一次请求

回到全书开头的那句话。Part I 动的是流量本身——KV、权重、激活；Part II 动的是流量的搬运路径——存储、网络、调度、编译；Part III 动的是谁为这些流量付钱。三者其实是同一件事的三个切面，这也是为什么第 2 章的一个公式可以贯穿到第 16 章的公司判断。

任何一项新技术出现时，先问它减少了多少计算、数据移动或等待，提高了哪种复用与并发，再问质量、尾延迟、隔离、可靠性和成本代价记在哪里。同一请求与同一 SLO 把这些问题连成可验证的账本，避免把 kernel 峰值直接推广成服务收益。

未解问题

1. **开放问题能否在事前分类？**本章只能给事后判据：看它是否已有可复算的损益公式。但「物理约束」与「暂时的工程限制」在当下往往无法区分——共享 KV 的表达容量、确定性的代价，都可能在某天被证明只是初始化或 kernel 的问题，那时表 19.1 的分组就要重写。缺的是一个能事前判断「这条约束会不会被绕开」的判据。
2. **路线图如何不过期？**源码路径绑定本书明确的 release/commit，行号会漂移，最佳实践也会反转：两年前 PD 分离还是少数派方案，今天已是默认选项之一。一份不会过期的路线图不存在；能做的只是把锚点从「结论」换成「问题」，并让每个答案都附带它的失效条件。
3. **本书框架的适用边界在哪？**如果模型架构把 KV 消掉（线性注意力、SSM 混合），第 3、7、11 章的前提会同时失效；如果推理成本降到接近于零，调度与存储层的成本结构可能改变，但不会仅因算术推理便宜就自动消失。本书给出的是一套在「注意力 + KV cache」范式内成立的定位方法。范式变了，问题清单要重写；但「降低哪段流量、提高哪段复用」这个问法，大概还会继续好用——这正是一本技术书能给出的最好的东西：不是结论，而是一组还会继续好用的问题。

附录 A 术语表

本书出现的术语按主题分组。每条给出中文名、英文名（主要讨论章节），以及一句可直接用于判断的定义。术语的中文译名不统一时，以英文名为准。

A.1 推理的物理与系统

预填充 Prefill（第 1 章）处理输入 prompt 的阶段。此时一次处理多个输入位置；较长输入的稠密矩阵乘常受算力限制，短输入或特殊算子仍需测量。

解码 Decode（第 1 章）逐 token 生成的阶段。单请求时线性层近似向量-矩阵乘；小 batch 常受带宽或启动开销限制，不能外推到任意并发。

算术强度 Arithmetic Intensity（第 2 章）每字节显存流量对应的浮点运算数，单位 FLOP/byte。

机器平衡点 Machine Balance（第 2 章）给定精度的峰值算力除以显存带宽，单位 FLOP/byte；是理想模型中两条上界的交点。

roofline 模型（第 2 章）以算术强度为横轴、可达性能为纵轴的性能上界模型。

TTFT, Time To First Token（第 1、7 章）客户端发出请求到首个有效输出 token 到达的时间；包含接入、排队、暴露的 prefill 与首输出路径。

TPOT, Time Per Output Token（第 1、7 章）同一请求首末输出 token 到达时间差除以输出数减一；只在输出数大于一时定义。

ITL, Inter-Token Latency（第 1、7 章）同一请求相邻输出 token 的逐次到达间隔；TPOT 是这些间隔的请求内平均，最大 ITL 与各类分位数另行统计。

E2E, End-to-End Latency（第 1 章）客户端发出请求到响应完成的时间；末 token 到达后若仍有结束协议尾部，也计入 E2E。

goodput（第 7、15 章）固定观察窗内成功、质量合格且满足全部预定 SLO 的完成请求数或其输出 token 数，除以窗口时长；报数须注明 request/s 或 output token/s，以及是否按 GPU 归一化。

SLO, Service Level Objective（第 7 章）服务等级目标，如「P99 TTFT 小于 1 秒」。

MFU, Model FLOPs Utilization（第 15 章）有效模型 FLOP 速率与所用精度硬件峰值之比；不等于设备忙碌率，也不自动衡量 SLO。

PUE, Power Usage Effectiveness（第 9 章）同一边界与期间内设施总能耗与 IT 能耗之比，包括制冷、配电等非 IT 损耗。

A.2 KV cache 与内存管理

KV cache（第 2 章）缓存已处理 token 的 Key 与 Value，避免 decode 时重算全部历史。

PagedAttention（第 7 章）把 KV cache 切成固定大小的块、用映射表管理，像操作系统分页一样减少连续分配的外部碎片，并为共享提供映射基础；仍有页内碎片与元数据成本。

块大小 Block Size（第 8 章）KV cache 分配的最小粒度，以 token 计。太小则映射开销大，太大则内部碎片多。

prefix caching（第 7 章）对模型、适配器、位置、token 前缀与隔离域相容的请求复用已计算的 KV；分页映射与缓存命中策略是不同机制。

RadixAttention（第 8 章）SGLang 用基数树组织前缀缓存，天然表达前缀的层级共享关系。

锁引用 lock_ref（第 8 章）SGLang 中防止正在使用的节点被驱逐的引用计数。

抢占 Preemption（第 7 章）暂停部分请求以让出资源；恢复可用重算或换出等路径，须保留已输出状态，具体能力依引擎版本。

重算 Recompute（第 7 章）释放部分或全部 KV，恢复时重新计算；线性层成本近似随长度增长，完整注

意力还含二次项。

换出 Swap（第 7 章）把 KV 暂存到其他存储层，恢复时搬回；成本包含双程字节、有效链路带宽、固定开销与排队。

PagedAttention 的碎片 Fragmentation（第 7 章）页内未用位置构成内部碎片；映射非连续物理块可减少连续分配的外部碎片，但元数据、共享引用与块池容量仍有成本。

chunked prefill（第 7 章）把长 prompt 的 prefill 切成多块，与 decode 混合调度，以平滑 ITL。

continuous batching（第 7 章）以迭代为单位调整活动批次；结束者退出，等待者在预算、显存与策略允许时加入。

PD 分离 Prefill-Decode Disaggregation（第 7 章）把 prefill 与 decode 放在不同实例或资源池，允许分别选择并行与配置；代价包括 KV 交接、两侧排队与故障处理，跨节点时另受网络约束。

A.3 注意力结构

MHA, Multi-Head Attention（第 3 章）每个注意力头有独立的 Q/K/V，KV 量与头数成正比。

MQA, Multi-Query Attention（第 3 章）所有查询头共享一组 K/V；相同头维度下 KV 量是 MHA 的 $1/n_h$，质量影响需训练和任务评测。

GQA, Grouped-Query Attention（第 3 章）查询头分组、组内共享 K/V。KV 组数为 $n_g = n_{\text{kv}}$，共享比 $g = n_h/n_g$；同头维与精度下缓存为 MHA 的 $1/g$。

MLA, Multi-head Latent Attention（第 3 章）DeepSeek 提出的低秩压缩注意力；所述实现缓存共享潜向量及位置相关分量，而非每头完整 K/V，布局依模型配置。

FlashAttention（第 3 章）通过分块与 online softmax 避免将完整注意力分数矩阵写回 HBM；IO 界还依赖头维度与片上容量。数学上精确，浮点结果未必逐位相同。

online softmax（第 3 章）分块增量维护最大值、归一化因子及输出累积量，使 softmax 注意力无须物化完整分数矩阵。

分块 Tiling（第 3 章）把大矩阵切成能放进 SRAM 的小块，减少对 HBM 的访问。

sliding window attention（第 3 章）每个 token 只关注最近 w 个位置；固定窗口下，一条长度 N 的序列注意力计算量为 $O(Nw)$，历史可见集随之改变。

attention sink（第 3 章）保留部分起始位置与近期窗口，可缓解特定模型的流式退化；不恢复全部被删历史信息。

稀疏注意力 Sparse Attention（第 3 章）只计算部分 token 对之间的注意力。代表：块稀疏（MoBA）、原生可训练稀疏（NSA）。

线性注意力 Linear Attention（第 3 章）利用核映射与状态累计使序列计算近似线性；模型形式、状态容量与任务质量不同于完整 softmax 注意力。

SSM, State Space Model（第 3 章）用状态递推代替注意力，代表 Mamba 系列。

混合架构 Hybrid Architecture（第 3 章）在模型里按比例混合注意力层与 SSM 层，兼顾长上下文效率与检索能力。

KV 头数 n_{kv}（第 2 章）决定每 token KV 字节数的关键参数。

A.4 量化与稀疏

W4A16（第 4 章）权重 4 bit、激活 16 bit；格式、缩放元数据和实际矩阵乘精度须另注明。

W8A8（第 4 章）权重与激活各 8 bit；可为整数或浮点，具体格式与 kernel 支持必须另行说明。

FP8（第 4 章）8 位浮点；E4M3 比 E5M2 有更多尾数位，E5M2 有更大指数范围，误差仍依数据与缩放而定。

MXFP4（第 4 章）微缩放格式，每 32 个元素共享一个指数缩放因子。

NVFP4（第 4 章）使用更细块的缩放结构及额外尺度；相对其它格式的质量与性能需同条件测量。

分块缩放 Block Scaling（第 4 章）一组元素共享一个缩放因子，是 4 bit 量化保住精度的关键。

GPTQ（第 4 章）逐列量化并用二阶信息补偿误差。

AWQ（第 4 章）按激活幅值识别显著通道，对其缩放保护。

SmoothQuant（第 4 章）把激活的量化难度部分迁移到权重上，使两者都更易量化。
QServe（第 4 章）W4A8KV4 的算法与系统协同设计。
2:4 结构化稀疏（第 5 章）每 4 个权重保留 2 个，在格式、精度与 kernel 支持匹配时可使用稀疏张量核；有效端到端收益还受元数据和其他算子限制。
非结构化稀疏（第 5 章）任意位置剪枝，理论压缩率高但访存不友好。
MoE, Mixture of Experts（第 5 章）每 token 路由到部分专家；总参数、活跃参数、共享专家与跨卡通信须分别记账，不能只由总参数量估算一次前向。
专家并行 EP, Expert Parallelism（第 5 章）把不同专家放在不同设备上，需要 all-to-all 通信。
EPLB, Expert Parallel Load Balancing（第 5 章）动态调整专家放置以均衡各卡负载。
冗余专家 Redundant Experts（第 5 章）为热点专家复制副本，缓解负载不均。
grouped GEMM（第 5 章）把多个专家的矩阵乘合并成一次 kernel 调用，减少启动开销。

A.5 并行与通信

TP, Tensor Parallelism（第 6 章）层内切分张量；经典稠密 Transformer 方案常含两次归约，融合与其它布局可改变原语。
PP, Pipeline Parallelism（第 6 章）按层切分，级间传激活，存在流水线气泡。
DP, Data Parallelism（第 6 章）复制模型、切分数据。推理时无需梯度同步。
CP / SP, Context / Sequence Parallelism（第 6 章）CP 按上下文切分注意力，可用 ring 等通信；SP 常指对特定激活算子的序列分片，二者不同。
all-reduce（第 12 章）把各 rank 的同形张量按指定运算归约，并让每个 rank 获得相同结果；是 TP 常用原语。
all-to-all（第 12 章）每设备向所有设备发送不同数据，EP 的核心原语。
ring all-reduce（第 12 章）环形 reduce-scatter 加 all-gather；p 个 rank、每 rank 输入 M byte 时，理想每 rank 发送量为 $2(p-1)M/p$，不是发送加接收总量。
NVLink / NVSwitch（第 12 章）GPU 高速互联及交换结构，构成 scale-up 域；覆盖范围可跨板、跨机柜，依代际与系统设计。
InfiniBand（第 12 章）支持 RDMA 的互联架构；成本与运维难度取决于拓扑、规模、设备和团队经验。
RoCEv2（第 12 章）在可路由以太网上承载 RDMA；拥塞、丢包与隔离策略影响性能和尾延迟。
UEC, Ultra Ethernet（第 12 章）Ultra Ethernet Consortium 制定的以太网协议与规范体系；功能可用性还取决于规范版本、设备及软件实现。
CXL（第 12 章）面向内存与加速器的一致性互联协议。
fat-tree / rail-optimized（第 12 章）两种典型集群拓扑，后者让同号网卡走同一交换机。
超节点 Supernode（第 10 章）把较多加速器置于同一高速 scale-up 域；域规模与拓扑依系统而定，实际 TP 扩展仍受通信、容量与 SLO 约束。

A.6 引擎与运行时

vLLM V1 架构（第 8 章）engine core 负责调度、executor 负责分发、worker 与 model runner 负责执行。
block manager（第 8 章）管理 KV 块的分配、释放与哈希索引。
链式哈希 Chained Hash（第 8 章）块标识包含父哈希、块内 token 与额外键；在键语义相容与无碰撞假设下识别共享前缀。
attention backend（第 8 章）可插拔的注意力实现（FlashAttention、FlashInfer、Triton 等）。
overlap scheduling（第 8 章）把 CPU 侧调度与 GPU 侧计算重叠，隐藏调度开销。
CUDA Graph（第 14 章）捕获执行依赖后重复提交，减少逐 kernel 的主机启动开销；仍有图启动、捕获与内存成本。
图捕获 Graph Capture（第 14 章）录制 CUDA Graph 的过程；动态 shape 常需桶化、填充或重捕获，并受地址与控制流约束。

Triton（第 14 章）面向 GPU 的类 Python kernel 编程语言。
CUTLASS / CuTe（第 14 章）NVIDIA 的模板化高性能 kernel 库。
torch.compile（第 14 章）PyTorch 的图编译前端，把 Python 代码降到可优化的图。
投机解码 Speculative Decoding（第 7 章）用小模型草拟若干 token、大模型一次验证，把多次串行 decode 变成一次并行验证。
EAGLE（第 7 章）在特征层做草拟的投机解码系列方法。
接受率 Acceptance Rate（第 7 章）草拟 token 被目标验证接受的比例；加速比还取决于验证、草拟、批大小、上下文和回收成本。
盈亏平衡接受率 Break-even Acceptance（第 7 章）使 $E = v + \gamma c$ 的条件接受率；只有假设目标验证成本 $v = 1$、草稿长度 $\gamma = 5$、单草稿成本 $c = 0.1$ 时约为 0.334，成本均相对普通目标一步归一化。
树形验证 Tree Verification（第 7 章）把草稿组织成树、用树形 mask 验证多分支；候选位置的计算、临时 KV 与被拒分支的回收均需预算，具体布局依实现。
约束解码 Constrained Decoding（第 14 章）把文法编译成状态机，每步对不允许的 token 打 $-\infty$ 掩码，在约束正确实现且请求成功完成时限制语法；不保证内容正确，截断与工具协议仍需处理。
jump-forward（第 14 章）文法在某状态下只允许唯一 token 序列时，直接追加这些 token、跳过前向。用总算力换更少的串行步数。
encoder cache（第 7 章）缓存视觉编码器的输出，供同一图像在多请求中复用。复用对象是编码结果而非 LLM KV，因此省去的是相应编码器前向；LLM 前缀复用需要另一套缓存相容条件。
视觉 token Vision Token（第 7 章）图像经 patch 切分与编码后进入 LLM 的 token。成本可分布在编码、投影、LLM prefill 与后续 KV；贯穿算例每个接入 LLM 的位置预留 320 KiB KV。
embedding 推理（第 2 章）通常通过一次编码前向产生向量，无自回归 KV 长期保留；短序列、小 batch 仍可能受带宽、启动或服务开销限制。
模型级联 Cascade（第 7 章）先让小模型回答，判据不通过再升级到大模型；忽略额外验证与状态成本时，相对大模型的平均成本为 $r + 1 - p$，其中 r 是成本比、p 是判据通过率；升级路径的额外延迟需另测。
模型路由 Routing（第 7 章）在生成前选择模型，避免每个请求必经串行升级；分类错误可能使质量不足或成本增加，须在独立负载校准。
通过率 Pass Rate（第 7 章）级联判据接受小模型结果的请求占比，不等于真实正确率；由负载、模型、判据与阈值共同决定，须在独立标注集同时校准漏判与误判。
延迟三分拆（第 15 章）把单请求时间拆成排队、空洞、真算三项。三类对应三种完全不同的治理动作。

A.7 基础设施与运维

TCO, Total Cost of Ownership（第 9 章）折旧、电力、机位制冷、网络、人力运维之和。
利用率 Utilization（第 9 章）须指明是卡时忙碌率、设备资源占用率还是有效产出比例；成本还依赖吞吐、SLO、质量与单位资源费用。
MIG, Multi-Instance GPU（第 13 章）把一张 GPU 切成多个隔离实例，硬件级分区。
MPS, Multi-Process Service（第 13 章）多进程共享同一 GPU，并发但无严格隔离。
gang scheduling（第 13 章）一组 Pod 必须同时被调度，否则全部等待。
device plugin（第 13 章）K8s 暴露 GPU 资源的机制。
冷启动 Cold Start（第 13 章）扩容后到能服务请求之间的准备时间，主要是模型加载。
MTBF, Mean Time Between Failures（第 15 章）可修复对象相邻失效间的平均运行时间；本书单卡教学参数 T 在独立、恒定失效率假设下给出全域首失效间隔 T/N，不能由训练作业中断数直接推成设备规格。
MTTR, Mean Time To Repair（第 15 章）指定故障域从失效到恢复的平均修复时间；与 MTBF 必须使用相同对象和时间单位，不能等同于请求重试时延。
SDC, Silent Data Corruption（第 15 章）不报错但结果错误的数据损坏。
训练 goodput（第 15 章）训练文献中的有效训练进展或有效算力指标，须按原文定义解释；不能与本书服务侧达标请求或 token 产出率直接互换。
容量规划 Capacity Planning（第 15 章）从请求分布反推所需卡数。

冷板式液冷 Cold Plate（第 9 章）冷板贴在 CPU/GPU 上带走主要热量，内存与电源仍靠风冷。可支持较高热密度，但上限由冷板、流量、设施供回水、配电及残余风冷共同约束。
浸没式液冷 Immersion Cooling（第 9 章）将设备置于介电液中带走热量；材料兼容、液体管理、维护与热排放链路均需重新验证。
WUE, Water Usage Effectiveness（第 9 章）在明确边界内每 kWh IT 电量消耗的水（升）；闭式设备侧回路不等于设施或电力生命周期零耗水。
批处理 API Batch API（第 17 章）以更宽完成时限换取不同价格的推理接口；折扣、时限和可用模型以具体条款与日期为准。本质是把负载的时间弹性折成折扣。

A.8 产业与商业

异构算力聚合（第 16 章）在统一资源接口下管理多种硬件，并分别处理算子、通信与性能差异；不是某一地区独有。
中立层 Neutral Layer（第 16 章）对提供适配、调度或平台服务这一产品边界的分析称呼；公司也可能经营芯片、模型或云业务，是否中立须按合同与控制权判断。
NeoCloud（第 17 章）新一代 GPU 云，自建或长租算力，按卡时或 token 出售。
单位经济学 Unit Economics（第 17 章）单个计费单位的收入与成本结构。
上游吞并测试 Upstream Absorption（第 16 章）判断某层价值是否会被芯片厂商或云厂商的垂直整合吃掉的分析框架。
利用率杠杆 Utilization Leverage（第 16 章）毛利率对 GPU 利用率的敏感度。
按卡时计费 / 按 token 计费 / license（第 17 章）三种计费形态，对应完全不同的毛利结构。

附录 B 符号表

表 B.1 汇总物理量，表 B.2 汇总加速与调度模型，表 B.3 汇总成本、可靠性和请求指标。符号以公式旁的局部定义为准；同形符号可能因章节承担不同角色，不能只凭字母进行跨章代入。

B.1 物理层：带宽、算力与强度

表 B.1 roofline 与字节账本的主要符号。峰值、有效值和聚合范围必须使用同一口径。

符号	含义	单位	章节
P	指定精度的加速器峰值算力 ;稠密与稀疏口径另标。第 7 章的 chunk 模型另以 P 表示 prompt 长度	FLOP/s	第 2 章
P_{eff}	同一计算范围的有效计算速率 ;实测值与教学假设分别标注	FLOP/s	第 2 章
B_{mem}	HBM 理论带宽；单卡或聚合范围由公式声明	byte/s	第 2 章
B_{eff}	指定存储或传输路径的有效带宽；同一 HBM 口径时不超过 B_{mem}	byte/s	第 2、12 章
F, Q	同一次计算的 FLOP 数与指定存储层级的流量；张量尺寸不自动等于实际 HBM 流量	FLOP；byte	第 2 章
I	算术强度 F/Q	FLOP/byte	第 2 章
I^*	机器平衡点 P/B_{mem}；H100 SXM 稠密 BF16 约 295，FP8 约 591	FLOP/byte	第 2 章
N	模型参数量 ;写作 N_{param} 时含义相同。并行与可靠性公式中的裸 N 可另指设备数	个参数	第 2 章
W	标量权重总字节数 Nb_w；矩阵 $\mathbf{W}$、通信窗口 W 是其它局部用法	byte	第 2 章
L	模型层数；L_{MoE} 专指路由 MoE 层数	层	第 2、5 章
b_w, b_a, b_{kv}	权重、激活、KV 每元素的载荷字节数 ;BF16 为 2，纯 4 bit 载荷为 0.5，缩放与索引另计	byte/元素	第 2、4 章
B	解码 batch 的请求数；第 12 章的通信模型另以 B 表示有效单向带宽	请求	第 2 章
B^*	单个 $d\times d'$ 线性层中权重与激活流量相等的 batch：$b_w dd'/[b_a(d+d')]$	请求	第 2 章
B_{wc}	忽略 KV 和激活的权重/计算交叉 batch，$I^* b_w/2$；BF16 时数值等于 I^*	请求	第 2 章
B_{kv}	KV 与权重流量相等的 batch，$W/(kL\ell)$	请求	第 2 章
B_{comp}	忽略注意力 FLOP 的代理模型中，计算与总读取相等的 batch：$W/(2N/I^* - kL\ell)$，仅在分母为正时有意义	请求	第 2 章
ℓ	已处理并可见的上下文长度，包括输入与已经生成的位置	token	第 2 章

符号	含义	单位	章节
ℓ^*	代理模型的 KV/线性计算交叉长度 $2NB_{\text{mem}}/(kLP)$；主例套用 H100 峰值时约 1448，并非服务或注意力性能阈值	token	第 2 章
n	prefill 输入长度；请求时间公式也用 o 或 m 表示输出长度，须按局部定义读取	token	第 2、7 章
c_{kv}	每 token 每层缓存的元素数；显式 K/V 为 $2n_{\text{kv}}d_h$，所述 MLA 为 $d_c + d_h^R$	元素	第 3 章
k	每 token 每层 KV 字节数 $c_{\text{kv}}b_{\text{kv}}$	byte/token/层	第 2、3 章
kL	每 token 全模型 KV 字节数；主例 BF16 为 327680 B，即 320 KiB	byte/token	第 2 章
M_{KV}	KV 容量；实际分配还可能包含页对齐、复制、临时分支与元数据	byte	第 2 章
n_h, n_{kv}	查询头数与 KV 头数；GQA 的 KV 组数 $n_g = n_{\text{kv}}$	个	第 3 章
d_h, d_{head}	每头维度的两种写法；MLA 的非位置与位置维度在正文另分下标	维	第 3 章
g	GQA 共享比 n_h/n_g，即每个 KV 组共享的查询头数；同条件缓存比为 $1/g$	—	第 3 章
Φ	功率；是否包含主机、网络或制冷须声明，不能把 TDP 直接当实测功率	W	第 2 章

B.2 加速、调度与通信

表 B.2 加速与调度模型的局部符号。同形的 α、c、γ 在不同小节使用不同单位。

符号	含义与适用条件	单位	章节
S	相对同质量基线的加速比：旧时间/新时间	—	第 4、7 章
ρ_{KV}	同一步 KV/权重字节比 $BkL\ell/W$。只改权重且同为带宽受限、效率相同时，纯流量比为 $(1+\rho_{\text{KV}})/(b'_w/b_w+\rho_{\text{KV}})$	—	第 4 章
η_0, η_q	基线与量化路径的有效 HBM 带宽/理论带宽；须按相同硬件和形状测量	—	第 4 章
Δt	量化新增解包、通信或启动等时间，须与公式其他时间项使用相同单位	s 或 ms	第 4 章
α（量化）	SmoothQuant 中激活/权重量化难度的迁移参数	—	第 4 章
s	非结构化稀疏的保留比例；$s = 0.1$ 表示保留 10%，不表示压缩后总字节只剩 10%	—	第 5 章
b_v, b_i	稀疏值与索引每元素字节数；行指针等其它元数据可另加	byte	第 5 章
$Q_{\text{act,eff}}$	激活 gather 在缓存与事务粒度作用下的实际有效流量；不将 sector 浪费系数乘到全部压缩权重上	byte	第 5 章

符号	含义与适用条件	单位	章节
E, K	第 5 章 all-to-all 模型中的 EP 组设备数与每 token 选择的路由专家数；E 不表示专家总数	个	第 5 章
k_e, q	第 12 章每 token 的专家选择数与平均远端路由比例	个；—	第 12 章
κ, κ_c	串行相加教学模型中的步时长/纯 decode 时长，分别为 $1+P\alpha/t_d$ 与 $1+c\alpha/t_d$	—	第 7 章
P, c（分块）	chunked prefill 的 prompt 长度与块大小；此处 P 不表示算力	token	第 7 章
α（分块）	局部线性 prefill 时间系数；重算模型也用它表示线性项系数，不代表任意长度的完整 attention 成本	s/token	第 7 章
t_d	固定 decode 负载下的纯 decode 一步时间	s	第 7 章
α（投机）	每个位置在此前候选均已接受条件下的接受概率；恒定 α 是链式教学近似	—	第 7 章
γ, c, v（投机）	草稿长度、单个草稿成本和一次目标验证成本；后两者相对普通目标一步 t_0 归一化	token；—；—	第 7 章
E（投机）	一轮验证期望输出数 $\sum_{i=0}^{\gamma}\alpha^i$；净加速比为 $E/(v+\gamma c)$	token	第 7 章
r_{KV}	投机验证中历史 KV 的有效加载倍率；由 tile、缓存和 attention 实现决定	—	第 7 章
β, γ（重算）	$a_w+\alpha n+\beta n^2$ 的二次时间系数，以及换出加换回的每 token 时间；此处 γ 不是草稿长度	s/token^2；s/token	第 7 章
r	LoRA 小节为低秩维数；级联小节为小模型/大模型单位成本比 c_1/c_2	维；—	第 6、7 章
p（级联）	判据接受小模型结果的请求比例，不等于真实正确率	—	第 7 章
τ	第 6 章的逐跳延迟；可靠性章另为 checkpoint 间隔	s	第 6、15 章
α（通信）	第 12 章集合通信模型中每个串行阶段的启动延迟，不是 prefill 系数	s	第 12 章
ρ_{link}	KV 到达字节率/有效链路字节率；大于 1 时，给定负载不可持续，仅小于 1 仍不保证 SLO	—	第 7 章
ρ	单服务台排队负载率 $\lambda\mathbb{E}[S]$；M/M/1 中等于 λ/μ，稳态要求小于 1	—	第 7、13 章
λ, μ	一致请求单位下的到达率与服务率；不是 decode 迭代频率	1/s	第 13 章
η_{el}	矩形尖峰教学例中可被新增容量覆盖的时间比例 $\max(0,T_p-T_d-T_r)/T_p$	—	第 13 章
η_{tool}	会话工具等待时间占工具等待与 GPU 执行总墙钟时间的比例；理想会话复用可填补空闲	—	第 18 章
u_{round}	FP32 舍入到最近时的单位舍入误差 2^{-24}；不要与时间利用率混用	—	第 18 章

B.3 成本、可靠性与请求指标

表 B.3 成本、容量与服务指标。成本中的货币以各算例声明为准，贯穿主例使用美元。

符号	含义与分母	单位	章节
N_g, N_{gpu}	成本或归一化范围内 GPU 数；可靠性公式也用裸 N 表示故障域设备数	张	第 9、15 章
r_g	单卡小时资源费用；含不含电费、主机和网络须声明	USD/(GPU·h)	第 9 章
C_h	同一服务边界的小时总成本，$N_g r_g + C_{h,\text{other}} + P_{\text{IT}}\pi e$；已含电的费用不重复加电费	USD/h	第 9 章
P_{IT}	同一边界的平均 IT 功率；主例仅把 GPU 功率纳入这一项，其它部件未测	kW	第 9 章
π, e	PUE 与电价；PUE 是同边界、同期间的设施总能量/IT 能量	—；USD/kWh	第 9 章
G_{out}, Q_g	完整观察窗内达标完成请求的输出 token 总数/窗口时长；包含计费的空闲、失败与故障期	token/s	第 9、17 章
λ_g	同一观察窗内达标完成请求率；变长输出时不能仅用平均 token 率随意换算	request/s	第 17 章
c_{out}, $c_{\text{out},g}$	每百万达标输出 token 成本 $10^6 C_h/(3600 G_{\text{out}})$；输入处理成本已在分子	USD/百万输出 token	第 9、17 章
$c_{\text{request},g}$	每个达标请求成本 $C_h/(3600\lambda_g)$	USD/request	第 17 章
p, c, g（毛利）	同一计费单位的售价、完整成本和毛利率 $g = 1 - c/p$；此处 g 不是 GQA 共享比	货币/单位；—	第 16 章
u（时间）	平均每卡有请求执行的墙钟时间/观察窗；与 MFU、排队负载率和达标产出率不同	—	第 15 章
MFU	模型必需 FLOP 速率/同精度设备聚合峰值；近似 $2N$ 的前向 FLOP 不能覆盖任意长度的 attention	—	第 15 章
$G_{\text{req/GPU}}$	成功、质量合格且满足全部预定 SLO 的请求数/（GPU 数 × 窗口秒数）	request/(s GPU)	第 15 章
T, M（可靠性）	单卡教学 MTBF 与全域首失效间隔 $M = T/N$；要求设备独立、恒定失效率且任一失效触发该域事件	h	第 15 章
λ_{cl}	上述假设下故障域失效率 N/T；不自动等于生产服务中断率	1/h	第 15 章
R, A（可靠性）	平均修复时间与全域停机模型的可用性 $A = M/(M+R)$；不是冗余服务的通用可用性公式	h；—	第 15 章
δ, τ	checkpoint 写入停顿与间隔；Young 近似 $\tau^\star = \sqrt{2\delta M}$ 中三项须统一时间单位	s	第 15 章
$\bar{B}$	平均在途请求数；稳定且守恒时 Little 定律为 $\bar{B} = \lambda \mathbb{E}[T]$，$T$ 此处是实际驻留时间	请求	第 10、15 章
TTFT	客户端首有效输出到达时间 $t_1 - t_0$	s	第 1、7 章

符号	含义与分母	单位	章节
ITL_i	相邻输出 token 到达间隔 $t_i - t_{i-1}$，$i \geq 2$；最大值与各类分位数分别统计	s	第 1、7 章
TPOT	单请求平均间隔 $(t_o - t_1)/(o-1)$，输出数 $o > 1$；单输出时无定义	s	第 1、7 章
E2E	响应完成时刻减请求发出时刻 $t_{\text{done}} - t_0$；仅忽略结束协议尾部时约为 $t_o - t_0$	s	第 1 章

! 先核对角色、单位与观察窗

B 在 roofline 公式中是 batch，在通信公式中可以是带宽；P 在物理模型中是算力，在 chunked prefill 小节中是 prompt 长度。g 在 GQA 中是共享比，在商业算例中是毛利率；r 则可能是 LoRA 秩或级联成本比。特别是 α，在量化与投机模型中无量纲，在局部 prefill 模型中为 s/token，在通信模型中为 s。符号表保留这些局部用法，跨公式代入前必须核对定义与单位。
已从完整计费观察窗测得的 goodput 已含闲置、重试与不达标损失，不能再乘一次利用率或成功率。只有从满载吞吐外推时，才能另引入经过定义和验证的折减因子。

B.4 单位与量级换算

本书以 GB = 10^9 B、TB = 10^{12} B 表示十进制容量，以 KiB = 2^{10} B、MiB = 2^{20} B、GiB = 2^{30} B 表示二进制容量。二进制与十进制同名前缀的倍率分别为 1.024、1.048576 和 1.073741824；误差百分比取决于选哪一个作分母，计算时先还原为 byte，再按表头输出。速率与算力使用十进制前缀：TB/s = 10^{12} B/s，TFLOPS = 10^{12} FLOP/s。带宽还须注明单向、双向或聚合；Gb/s 中的 b 是 bit，GB/s 中的 B 是 byte，8 bit=1 byte。

量	本书明确口径后的写法	容易混淆的写法
显存容量	80 GB ≈ 74.51 GiB	80 GiB
BF16 权重	140 GB ≈ 130.39 GiB	140 GiB
主例 KV	320 KiB/token × 8192 = 2.5 GiB	2.5 GB
HBM 带宽	3.35 TB/s = 3350 GB/s	3.35 TiB/s
端口线速	400 Gb/s = 50 GB/s，尚未扣协议开销	400 GB/s
算力	989 TFLOPS 与 989 TFLOP/s 是同一单位	把 FLOP/s 当成 FLOP

参考文献

[1] Tri Dao, Daniel Y. Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. *FlashAttention: Fast and Memory-Efficient Exact Attention with IO-Awareness*. 2022. arXiv: 2205.14135v2 [cs.LG]. URL: https://arxiv.org/abs/2205.14135v2 (visited on 10/22/2026).

[2] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, et al. *Efficient Memory Management for Large Language Model Serving with PagedAttention*. 2023. arXiv: 2309.06180v1 [cs.LG]. URL: https://arxiv.org/abs/2309.06180v1 (visited on 10/22/2026).

[3] NVIDIA. *NVIDIA H100 Tensor Core GPU: Product Specifications*. URL: https://www.nvidia.com/en-us/data-center/h100/ (visited on 10/22/2026).

[4] NVIDIA. *NVIDIA H200 Tensor Core GPU*. URL: https://www.nvidia.com/en-us/data-center/h200/ (visited on 10/22/2026).

[5] Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, et al. *The Llama 3 Herd of Models*. 2024. arXiv: 2407.21783v3. URL: https://arxiv.org/abs/2407.21783v3 (visited on 10/22/2026).

[6] Noam Shazeer. *Fast Transformer Decoding: One Write-Head is All You Need*. 2019. arXiv: 1911.02150v1 [cs.NE]. URL: https://arxiv.org/abs/1911.02150v1 (visited on 10/22/2026).

[7] Joshua Ainslie, James Lee-Thorp, Michiel de Jong, Yury Zemlyanskiy, Federico Lebrón, and Sumit Sanghai. *GQA: Training Generalized Multi-Query Transformer Models from Multi-Head Checkpoints*. 2023. arXiv: 2305.13245v3 [cs.CL]. URL: https://arxiv.org/abs/2305.13245v3 (visited on 10/22/2026).

[8] DeepSeek-AI, Aixin Liu, Bei Feng, et al. *DeepSeek-V3 Technical Report*. 2024. arXiv: 2412.19437v2 [cs.CL]. URL: https://arxiv.org/abs/2412.19437v2 (visited on 10/22/2026).

[9] DeepSeek-AI, Aixin Liu, Bei Feng, et al. *DeepSeek-V2: A Strong, Economical, and Efficient Mixture-of-Experts Language Model*. 2024. arXiv: 2405.04434v5 [cs.CL]. URL: https://arxiv.org/abs/2405.04434v5 (visited on 10/22/2026).

[10] Tri Dao. *FlashAttention-2: Faster Attention with Better Parallelism and Work Partitioning*. 2023. arXiv: 2307.08691v1 [cs.LG]. URL: https://arxiv.org/abs/2307.08691v1 (visited on 10/22/2026).

[11] Jay Shah, Ganesh Bikshandi, Ying Zhang, Vijay Thakkar, Pradeep Ramani, and Tri Dao. *FlashAttention-3: Fast and Accurate Attention with Asynchrony and Low-precision*. 2024. arXiv: 2407.08608v2 [cs.LG]. URL: https://arxiv.org/abs/2407.08608v2 (visited on 10/22/2026).

[12] Iz Beltagy, Matthew E. Peters, and Arman Cohan. "Longformer: The Long-Document Transformer". In: *arXiv preprint* (2020). arXiv: 2004.05150v2 [cs.CL]. URL: https://arxiv.org/abs/2004.05150v2 (visited on 10/22/2026).

[13] Albert Q. Jiang, Alexandre Sablayrolles, Arthur Mensch, et al. "Mistral 7B". In: *arXiv preprint* (2023). arXiv: 2310.06825v1 [cs.CL]. URL: https://arxiv.org/abs/2310.06825v1 (visited on 10/22/2026).

[14] Guangxuan Xiao, Yuandong Tian, Beidi Chen, Song Han, and Mike Lewis. *Efficient Streaming Language Models with Attention Sinks*. 2023. arXiv: 2309.17453v4. URL: https://arxiv.org/abs/2309.17453v4 (visited on 10/22/2026).

[15] Enzhe Lu, Zhejun Jiang, Jingyuan Liu, et al. *MoBA: Mixture of Block Attention for Long-Context LLMs*. 2025. arXiv: 2502.13189v1 [cs.LG]. URL: https://arxiv.org/abs/2502.13189v1 (visited on 10/22/2026).

[16] Jingyang Yuan, Huazuo Gao, Damai Dai, et al. *Native Sparse Attention: Hardware-Aligned and Natively Trainable Sparse Attention*. 2025. arXiv: 2502.11089v2. URL: https://arxiv.org/abs/2502.11089v2 (visited on 10/22/2026).

[17] DeepSeek-AI, Aixin Liu, Aoxue Mei, et al. *DeepSeek-V3.2: Pushing the Frontier of Open Large Language Models*. 2025. arXiv: 2512.02556v1 [cs.CL]. URL: https://arxiv.org/abs/2512.02556v1 (visited on 10/22/2026).

[18] Albert Gu and Tri Dao. *Mamba: Linear-Time Sequence Modeling with Selective State Spaces*. 2023. arXiv: 2312.00752v2 [cs.LG]. URL: https://arxiv.org/abs/2312.00752v2 (visited on 10/22/2026).

[19] Matanel Oren, Michael Hassid, Nir Yarden, Yossi Adi, and Roy Schwartz. *Transformers are Multi-State RNNs*. 2024. arXiv: 2401.06104v2 [cs.CL]. URL: https://arxiv.org/abs/2401.06104v2 (visited on 10/22/2026).

[20] Tri Dao and Albert Gu. *Transformers are SSMs: Generalized Models and Efficient Algorithms Through Structured State Space Duality*. 2024. arXiv: 2405.21060v1. URL: https://arxiv.org/abs/2405.21060v1 (visited on 10/22/2026).

[21] Opher Lieber, Barak Lenz, Hofit Bata, et al. "Jamba: A Hybrid Transformer-Mamba Language Model". In: *arXiv preprint* (2024). arXiv: 2403.19887v2 [cs.CL]. URL: https://arxiv.org/abs/2403.19887v2 (visited on 10/22/2026).

[22] Simran Arora, Sabri Eyuboglu, Aman Timalsina, et al. "Zoology: Measuring and Improving Recall in Efficient Language Models". In: *arXiv preprint* (2023). arXiv: 2312.04927v1 [cs.CL]. URL: https://arxiv.org/abs/2312.04927v1 (visited on 10/22/2026).

[23] Yujun Lin, Haotian Tang, Shang Yang, et al. *QServe: W4A8KV4 Quantization and System Co-design for Efficient LLM Serving*. 2024. arXiv: 2405.04532v3. URL: https://arxiv.org/abs/2405.04532v3 (visited on 10/22/2026).

[24] Paulius Micikevicius, Dusan Stosic, Neil Burgess, et al. *FP8 Formats for Deep Learning*. 2022. arXiv: 2209.05433v2. URL: https://arxiv.org/abs/2209.05433v2 (visited on 10/22/2026).

[25] Mingjie Sun, Xinlei Chen, J. Zico Kolter, and Zhuang Liu. *Massive Activations in Large Language Models*. 2024. arXiv: 2402.17762v2. URL: https://arxiv.org/abs/2402.17762v2 (visited on 10/22/2026).

[26] Tim Dettmers, Mike Lewis, Younes Belkada, and Luke Zettlemoyer. *LLM.int8(): 8-bit Matrix Multiplication for Transformers at Scale*. 2022. arXiv: 2208.07339v2. URL: https://arxiv.org/abs/2208.07339v2 (visited on 10/22/2026).

[27] Bita Darvish Rouhani, Ritchie Zhao, Ankit More, et al. *Microscaling Data Formats for Deep Learning*. 2023. arXiv: 2310.10537v3. URL: https://arxiv.org/abs/2310.10537v3 (visited on 10/22/2026).

[28] NVIDIA. *Introducing NVFP4 for Efficient and Accurate Low-Precision Inference*. 官方文档, 访问于 2026 年 10 月. June 2025. URL: https://developer.nvidia.com/blog/introducing-nvfp4-for-efficient-and-accurate-low-precision-inference/ (visited on 10/22/2026).

[29] Elias Frantar, Saleh Ashkboos, Torsten Hoefler, and Dan Alistarh. *GPTQ: Accurate Post-Training Quantization for Generative Pre-trained Transformers*. 2022. arXiv: 2210.17323v2. URL: https://arxiv.org/abs/2210.17323v2 (visited on 10/22/2026).

[30] Ji Lin, Jiaming Tang, Haotian Tang, et al. *AWQ: Activation-aware Weight Quantization for LLM Compression and Acceleration*. 2023. arXiv: 2306.00978v6. URL: https://arxiv.org/abs/2306.00978v6 (visited on 10/22/2026).

[31] Guangxuan Xiao, Ji Lin, Mickael Seznec, Hao Wu, Julien Demouth, and Song Han. *SmoothQuant: Accurate and Efficient Post-Training Quantization for Large Language Models*. 2022. arXiv: 2211.10438v7. URL: https://arxiv.org/abs/2211.10438v7 (visited on 10/22/2026).

[32] Saleh Ashkboos, Amirkeivan Mohtashami, Maximilian L. Croci, et al. *QuaRot: Outlier-Free 4-Bit Inference in Rotated LLMs*. 2024. arXiv: 2404.00456v2. URL: https://arxiv.org/abs/2404.00456v2 (visited on 10/22/2026).

[33] Cheng-Ping Hsieh, Simeng Sun, Samuel Kriman, et al. *RULER: What's the Real Context Size of Your Long-Context Language Models?* 2024. arXiv: 2404.06654v3. URL: https://arxiv.org/abs/2404.06654v3 (visited on 10/22/2026).

[34] Zirui Liu, Jiayi Yuan, Hongye Jin, et al. *KIVI: A Tuning-Free Asymmetric 2bit Quantization for KV Cache*. 2024. arXiv: 2402.02750v2. URL: https://arxiv.org/abs/2402.02750v2 (visited on 10/22/2026).

[35] Asit Mishra, Jorge Albericio Latorre, Jeff Pool, et al. *Accelerating Sparse Deep Neural Networks*. 2021. arXiv: 2104.08378v1. URL: https://arxiv.org/abs/2104.08378v1 (visited on 10/22/2026).

[36] NVIDIA. *NVIDIA A100 Tensor Core GPU Architecture*. 2020. URL: https://images.nvidia.com/aem-dam/en-zz/Solutions/data-center/nvidia-ampere-architecture-whitepaper.pdf (visited on 10/22/2026).

[37] Song Han, Jeff Pool, John Tran, and William J. Dally. *Learning both Weights and Connections for Efficient Neural Networks*. 2015. arXiv: 1506.02626v3. URL: https://arxiv.org/abs/1506.02626v3 (visited on 10/22/2026).

[38] Mingjie Sun, Zhuang Liu, Anna Bair, and J. Zico Kolter. *A Simple and Effective Pruning Approach for Large Language Models*. 2023. arXiv: 2306.11695v3. URL: https://arxiv.org/abs/2306.11695v3 (visited on 10/22/2026).

[39] Elias Frantar and Dan Alistarh. *SparseGPT: Massive Language Models Can Be Accurately Pruned in One-Shot*. 2023. arXiv: 2301.00774v3. URL: https://arxiv.org/abs/2301.00774v3 (visited on 10/22/2026).

[40] Noam Shazeer, Azalia Mirhoseini, Krzysztof Maziarz, et al. *Outrageously Large Neural Networks: The Sparsely-Gated Mixture-of-Experts Layer*. 2017. arXiv: 1701.06538v1. URL: https://arxiv.org/abs/1701.06538v1 (visited on 10/22/2026).

[41] Damai Dai, Chengqi Deng, Chenggang Zhao, et al. *DeepSeekMoE: Towards Ultimate Expert Specialization in Mixture-of-Experts Language Models*. 2024. arXiv: 2401.06066v1. URL: https://arxiv.org/abs/2401.06066v1 (visited on 10/22/2026).

[42] Samyam Rajbhandari, Conglong Li, Zhewei Yao, et al. *DeepSpeed-MoE: Advancing Mixture-of-Experts Inference and Training to Power Next-Generation AI Scale*. 2022. arXiv: 2201.05596v2. URL: https://arxiv.org/abs/2201.05596v2 (visited on 10/22/2026).

[43] Reza Yazdani Aminabadi, Samyam Rajbhandari, Minjia Zhang, et al. *DeepSpeed Inference: Enabling Efficient Inference of Transformer Models at Unprecedented Scale*. 2022. arXiv: 2207.00032v1. URL: https://arxiv.org/abs/2207.00032v1 (visited on 10/22/2026).

[44] DeepSeek-AI. *DeepEP: an efficient expert-parallel communication library*. URL: https://github.com/deepseek-ai/DeepEP (visited on 10/22/2026).

[45] DeepSeek-AI. *EPLB: Expert Parallelism Load Balancer*. URL: https://github.com/deepseek-ai/EPLB (visited on 10/22/2026).

[46] Trevor Gale, Deepak Narayanan, Cliff Young, and Matei Zaharia. *MegaBlocks: Efficient Sparse Training with Mixture-of-Experts*. 2022. arXiv: 2211.15841v1. URL: https://arxiv.org/abs/2211.15841v1 (visited on 10/22/2026).

[47] Changho Hwang, Wei Cui, Yifan Xiong, et al. *Tutel: Adaptive Mixture-of-Experts at Scale*. 2022. arXiv: 2206.03382v2. URL: https://arxiv.org/abs/2206.03382v2 (visited on 10/22/2026).

[48] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. *Megatron-LM: Training Multi-Billion Parameter Language Models Using Model Parallelism*. 2019. arXiv: 1909.08053v4. URL: https://arxiv.org/abs/1909.08053v4 (visited on 10/22/2026).

[49] Yanping Huang, Youlong Cheng, Ankur Bapna, et al. *GPipe: Efficient Training of Giant Neural Networks using Pipeline Parallelism*. 2018. arXiv: 1811.06965v5 [cs.DC]. URL: https://arxiv.org/abs/1811.06965v5 (visited on 10/22/2026).

[50] Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. *ZeRO: Memory Optimizations Toward Training Trillion Parameter Models*. 2019. arXiv: 1910.02054v3 [cs.LG]. URL: https://arxiv.org/abs/1910.02054v3 (visited on 10/22/2026).

[51] William Fedus, Barret Zoph, and Noam Shazeer. *Switch Transformers: Scaling to Trillion Parameter Models with Simple and Efficient Sparsity*. 2021. arXiv: 2101.03961v3. URL: https://arxiv.org/abs/2101.03961v3 (visited on 10/22/2026).

[52] Dmitry Lepikhin, HyoukJoong Lee, Yuanzhong Xu, et al. *GShard: Scaling Giant Models with Conditional Computation and Automatic Sharding*. 2020. arXiv: 2006.16668v1. URL: https://arxiv.org/abs/2006.16668v1 (visited on 10/22/2026).

[53] Hao Liu, Matei Zaharia, and Pieter Abbeel. *Ring Attention with Blockwise Transformers for Near-Infinite Context*. 2023. arXiv: 2310.01889v4. URL: https://arxiv.org/abs/2310.01889v4 (visited on 10/22/2026).

[54] Deepak Narayanan, Mohammad Shoeybi, Jared Casper, et al. *Efficient Large-Scale Language Model Training on GPU Clusters Using Megatron-LM*. 2021. arXiv: 2104.04473v5 [cs.DC]. URL: https://arxiv.org/abs/2104.04473v5 (visited on 10/22/2026).

[55] NVIDIA. *NVLink and NVSwitch*. URL: https://www.nvidia.com/en-us/data-center/nvlink/ (visited on 10/22/2026).

[56] vLLM Project. *vLLM Documentation: Optimization — Parallelism Strategies*. URL: https://docs.vllm.ai/en/latest/configuration/optimization.html (visited on 10/22/2026).

[57] Gyeong-In Yu, Joo Seong Jeong, Geon-Woo Kim, Soojeong Kim, and Byung-Gon Chun. "Orca: A Distributed Serving System for Transformer-Based Generative Models". In: *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. USENIX Association, 2022, pp. 521–538. URL: https://www.usenix.org/conference/osdi22/presentation/yu (visited on 10/22/2026).

[58] Amey Agrawal, Nitin Kedia, Ashish Panwar, et al. *Taming Throughput-Latency Tradeoff in LLM Inference with Sarathi-Serve*. 2024. arXiv: 2403.02310v3 [cs.LG]. URL: https://arxiv.org/abs/2403.02310v3 (visited on 10/22/2026).

[59] Amey Agrawal, Ashish Panwar, Jayashree Mohan, Nipun Kwatra, Bhargav S. Gulavani, and Ramachandran Ramjee. *SARATHI: Efficient LLM Inference by Piggybacking Decodes with Chunked Prefills*. 2023. arXiv: 2308.16369v1 [cs.LG]. URL: https://arxiv.org/abs/2308.16369v1 (visited on 10/22/2026).

[60] Yaniv Leviathan, Matan Kalman, and Yossi Matias. *Fast Inference from Transformers via Speculative Decoding*. 2022. arXiv: 2211.17192v2. URL: https://arxiv.org/abs/2211.17192v2 (visited on 10/22/2026).

[61] Yuhui Li, Fangyun Wei, Chao Zhang, and Hongyang Zhang. *EAGLE: Speculative Sampling Requires Rethinking Feature Uncertainty*. 2024. arXiv: 2401.15077v3. URL: https://arxiv.org/abs/2401.15077v3 (visited on 10/22/2026).

[62] Yuhui Li, Fangyun Wei, Chao Zhang, and Hongyang Zhang. *EAGLE-2: Faster Inference of Language Models with Dynamic Draft Trees*. 2024. arXiv: 2406.16858v2. URL: https://arxiv.org/abs/2406.16858v2 (visited on 10/22/2026).

[63] Yuhui Li, Fangyun Wei, Chao Zhang, and Hongyang Zhang. *EAGLE-3: Scaling up Inference Acceleration of Large Language Models via Training-Time Test*. 2025. arXiv: 2503.01840v3. URL: https://arxiv.org/abs/2503.01840v3 (visited on 10/22/2026).

[64] Tianle Cai, Yuhong Li, Zhengyang Geng, et al. *Medusa: Simple LLM Inference Acceleration Framework with Multiple Decoding Heads*. 2024. arXiv: 2401.10774v3. URL: https://arxiv.org/abs/2401.10774v3 (visited on 10/22/2026).

[65] Ramya Prabhu, Ajay Nayak, Jayashree Mohan, Ramachandran Ramjee, and Ashish Panwar. *vAttention: Dynamic Memory Management for Serving LLMs without PagedAttention*. 2024. arXiv: 2405.04437v3. URL: https://arxiv.org/abs/2405.04437v3 (visited on 10/22/2026).

[66] Zhenyu Zhang, Ying Sheng, Tianyi Zhou, et al. *H_2O: Heavy-Hitter Oracle for Efficient Generative Inference of Large Language Models*. 2023. arXiv: 2306.14048v3. URL: https://arxiv.org/abs/2306.14048v3 (visited on 10/22/2026).

[67] Yinmin Zhong, Shengyu Liu, Junda Chen, et al. *DistServe: Disaggregating Prefill and Decoding for Goodput-optimized Large Language Model Serving*. 2024. arXiv: 2401.09670v3. URL: https://arxiv.org/abs/2401.09670v3 (visited on 10/22/2026).

[68] Pratyush Patel, Esha Choukse, Chaojie Zhang, et al. *Splitwise: Efficient generative LLM inference using phase splitting*. 2023. arXiv: 2311.18677v2. URL: https://arxiv.org/abs/2311.18677v2 (visited on 10/22/2026).

[69] Ruoyu Qin, Zheming Li, Weiran He, et al. *Mooncake: A KVCache-centric Disaggregated Architecture for LLM Serving*. 2024. arXiv: 2407.00079v5. URL: https://arxiv.org/abs/2407.00079v5 (visited on 10/22/2026).

[70] vLLM Contributors. *vLLM v0.11.0 source snapshot*. Release v0.11.0; immutable commit in URL; local SHA-256 source manifest provided. 2025. URL: https://github.com/vllm-project/vllm/tree/b8b302cde434df8c9289a2b465406b47ebab1c2d (visited on 10/22/2026).

[71] SGLang Contributors. *SGLang v0.5.3.post3 source snapshot*. Release v0.5.3.post3; immutable commit in URL; local SHA-256 source manifest provided. 2025. URL: https://github.com/sgl-project/sglang/tree/85ebeecf0694bdb94ace25319775d4b060baf655 (visited on 10/22/2026).

[72] vLLM Team. *vLLM V1: A Major Upgrade to vLLM's Core Architecture*. Published 2025-01-27. 2025. URL: https://vllm.ai/blog/2025-01-27-v1-alpha-release (visited on 10/22/2026).

[73] Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, et al. *SGLang: Efficient Execution of Structured Language Model Programs*. 2023. arXiv: 2312.07104v2. URL: https://arxiv.org/abs/2312.07104v2 (visited on 10/22/2026).

[74] Zihao Ye, Lequn Chen, Ruihang Lai, et al. *FlashInfer: Efficient and Customizable Attention Engine for LLM Inference Serving*. 2025. arXiv: 2501.01005v2 [cs.LG]. URL: https://arxiv.org/abs/2501.01005v2 (visited on 10/22/2026).

[75] The Green Grid. *PUE: A Comprehensive Examination of the Metric*. Definition and measurement boundary for Power Usage Effectiveness; page accessed 2026-10-22. 2014. URL: https://www.thegreengrid.org/en/resources/library-and-tools/20-PUE%3A-A-Comprehensive-Examination-of-the-Metric (visited on 10/22/2026).

[76] NVIDIA. *DGX B200: The Foundation for Your AI Factory*. URL: https://www.nvidia.com/en-us/data-center/dgx-b200/ (visited on 10/22/2026).

[77] NVIDIA. *NVIDIA Blackwell GPU (B200): Product Specifications*. URL: https://www.nvidia.com/en-us/data-center/b200/ (visited on 10/22/2026).

[78] NVIDIA. *GB200 NVL72: A Single GPU Scale-Up Domain of 72 Blackwell GPUs*. URL: https://www.nvidia.com/en-us/data-center/gb200-nvl72/ (visited on 10/22/2026).

[79] NVIDIA. *Hopper Tuning Guide*. URL: https://docs.nvidia.com/cuda/hopper-tuning-guide/ (visited on 10/22/2026).

[80] SGLang Team. *HiCache System Design and Optimization*. URL: https://docs.sglang.ai/advanced_features/hicache_design.html (visited on 10/22/2026).

[81] Yuhan Liu, Yihua Cheng, Jiayi Yao, et al. *LMCache: An Efficient KV Cache Layer for Enterprise-Scale LLM Inference*. 2025. arXiv: 2510.09665v2. URL: https://arxiv.org/abs/2510.09665v2 (visited on 10/22/2026).

[82] NVIDIA. *NCCL: Collective Operations*. URL: https://docs.nvidia.com/deeplearning/nccl/user-guide/docs/usage/collectives.html (visited on 10/22/2026).

[83] Yuhan Liu, Hanchen Li, Yihua Cheng, et al. "CacheGen: KV Cache Compression and Streaming for Fast Large Language Model Serving". In: *Proceedings of the ACM SIGCOMM 2024 Conference*. 2023. arXiv: 2310.07240v6 [cs.DC]. URL: https://arxiv.org/abs/2310.07240v6 (visited on 10/22/2026).

[84] The Kubernetes Authors. *Device Plugins*. URL: https://kubernetes.io/docs/concepts/extend-kubernetes/compute-storage-net/device-plugins/ (visited on 10/22/2026).

[85] The Kubernetes Authors. *Dynamic Resource Allocation*. URL: https://kubernetes.io/docs/concepts/scheduling-eviction/dynamic-resource-allocation/ (visited on 10/22/2026).

[86] The Kubernetes Authors. *Kueue: Overview*. URL: https://kueue.sigs.k8s.io/docs/overview/ (visited on 10/22/2026).

[87] NVIDIA. *Multi-Instance GPU (MIG) User Guide*. URL: https://docs.nvidia.com/datacenter/tesla/mig-user-guide/ (visited on 10/22/2026).

[88] NVIDIA. *Multi-Instance GPU User Guide: Concepts*. URL: https://docs.nvidia.com/datacenter/tesla/mig-user-guide/concepts.html (visited on 10/22/2026).

[89] NVIDIA. *Multi-Process Service (MPS)*. URL: https://docs.nvidia.com/deploy/mps/ (visited on 10/22/2026).

[90] NVIDIA. *CUDA Multi-Process Service: Architecture*. URL: https://docs.nvidia.com/deploy/mps/architecture.html (visited on 10/22/2026).

[91] NVIDIA. *Time-Slicing GPUs in Kubernetes*. URL: https://docs.nvidia.com/datacenter/cloud-native/gpu-operator/latest/gpu-sharing.html (visited on 10/22/2026).

[92] Tianqi Chen, Thierry Moreau, Ziheng Jiang, et al. "TVM: An Automated End-to-End Optimizing Compiler for Deep Learning". In: *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 2018. arXiv: 1802.04799v3. URL: https://arxiv.org/abs/1802.04799v3 (visited on 10/22/2026).

[93] Chris Lattner, Mehdi Amini, Uday Bondhugula, et al. *MLIR: A Compiler Infrastructure for the End of Moore's Law*. 2020. arXiv: 2002.11054v2. URL: https://arxiv.org/abs/2002.11054v2 (visited on 10/22/2026).

[94] PyTorch Contributors. *torch.compile End-to-End Caching and Compiler Cache Artifacts*. URL: https://docs.pytorch.org/tutorials/recipes/torch_compile_caching_tutorial.html (visited on 10/22/2026).

[95] Philippe Tillet, H. T. Kung, and David Cox. "Triton: An Intermediate Language and Compiler for Tiled Neural Network Computations". In: *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages (MAPL)*. 2019, pp. 10–19. DOI: 10.1145/3315508.3329973. URL: https://doi.org/10.1145/3315508.3329973 (visited on 10/22/2026).

[96] NVIDIA. *CUTLASS: CUDA Templates for Linear Algebra Subroutines (CuTe Layout Algebra)*. URL: https://github.com/NVIDIA/cutlass (visited on 10/22/2026).

[97] Lei Wang, Yu Cheng, Yining Shi, et al. *TileLang: A Composable Tiled Programming Model for AI Systems*. 2025. arXiv: 2504.17577v2. URL: https://arxiv.org/abs/2504.17577v2 (visited on 10/22/2026).

[98] NVIDIA. *CUDA C++ Programming Guide: CUDA Graphs*. URL: https://docs.nvidia.com/cuda/cuda-programming-guide/04-special-topics/cuda-graphs.html (visited on 10/22/2026).

[99] NVIDIA. *CUDA C++ Programming Guide: Virtual Memory Management*. URL: https://docs.nvidia.com/cuda/cuda-programming-guide/04-special-topics/virtual-memory-management.html (visited on 10/22/2026).

[100] John W. Young. "A First Order Approximation to the Optimum Checkpoint Interval". In: *Communications of the ACM* 17.9 (1974), pp. 530–531. DOI: 10.1145/361147.361115. URL: https://doi.org/10.1145/361147.361115 (visited on 10/22/2026).

[101] Borui Wan, Gaohong Liu, Zuquan Song, et al. *Robust LLM Training Infrastructure at ByteDance*. 2025. arXiv: 2509.16293v4. URL: https://arxiv.org/abs/2509.16293v4 (visited on 10/22/2026).

[102] Peter H. Hochschild, Paul Turner, Jeffrey C. Mogul, et al. "Cores that Don't Count". In: *Proceedings of the Workshop on Hot Topics in Operating Systems (HotOS '21)*. 2021, pp. 9–16. DOI: 10.1145/3458336.3465297. URL: https://doi.org/10.1145/3458336.3465297 (visited on 10/22/2026).

[103] Harish Dattatraya Dixit, Sneha Pendharkar, Matt Beadon, et al. *Silent Data Corruptions at Scale*. 2021. arXiv: 2102.11245v1. URL: https://arxiv.org/abs/2102.11245v1 (visited on 10/22/2026).

[104] Abhishek Tyagi, Saurabh Hukerikar, Nirmal Saxena, et al. *LLM-PRISM: Characterizing Silent Data Corruption from Permanent GPU Faults in LLM Training*. 2026. arXiv: 2604.10390v2. URL: https://arxiv.org/abs/2604.10390v2 (visited on 10/22/2026).

[105] Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, et al. *PaLM: Scaling Language Modeling with Pathways*. 2022. arXiv: 2204.02311v5. URL: https://arxiv.org/abs/2204.02311v5 (visited on 10/22/2026).

[106] Mor Harchol-Balter. *Performance Modeling and Design of Computer Systems: Queueing Theory in Action*. Cambridge University Press, 2013. DOI: 10.1017/CBO9781139226424. (Visited on 10/22/2026).

[107] Betsy Beyer, Chris Jones, Jennifer Petoff, and Niall Richard Murphy. *Site Reliability Engineering: How Google Runs Production Systems*. O'Reilly Media, 2016. URL: https://sre.google/sre-book/table-of-contents/.

[108] NVIDIA. *NVIDIA Nsight Systems User Guide*. URL: https://docs.nvidia.com/nsight-systems/UserGuide/index.html (visited on 10/22/2026).

[109] NVIDIA. *NVIDIA Nsight Compute User Interface and Metrics Reference*. URL: https://docs.nvidia.com/nsight-compute/NsightCompute/index.html (visited on 10/22/2026).

[110] PyTorch Team. *torch.profiler: PyTorch Profiler API*. URL: https://docs.pytorch.org/docs/stable/profiler.html (visited on 10/22/2026).

[111] Brendan Gregg. *BPF Performance Tools: Linux System and Application Observability*. Addison-Wesley, 2019. URL: https://www.brendangregg.com/bpf-performance-tools-book.html (visited on 10/22/2026).

[112] 无问芯穹. 无问芯穹：通用人工智能基础设施服务商. 动态官网产品快照；不作为性能或财务审计证据. URL: https://www.infinigence-ai.com/ (visited on 10/22/2026).

[113] Inferact Team. *Introducing Inferact: Building the Future of AI Inference*. Jan. 2026. URL: https://inferact.ai/news/introducing-inferact-building-the-future-of-ai-inference (visited on 10/22/2026).

[114] Inferact. *Inferact Platform / Cloud: Frontier Inference You Control*. URL: https://inferact.ai/ (visited on 10/22/2026).

[115] Si Xu, Zixiao Huang, Yan Zeng, et al. *HETHUB: A Distributed Training System with Heterogeneous Cluster for Large-Scale Models*. 2024. arXiv: 2405.16256v2. URL: https://arxiv.org/abs/2405.16256v2 (visited on 10/22/2026).

[116] Fireworks AI. *Fireworks Pricing*. URL: https://fireworks.ai/pricing (visited on 10/22/2026).

[117] Together AI. *Pricing*. URL: https://www.together.ai/pricing (visited on 10/22/2026).

[118] Baseten. *Cloud Pricing*. URL: https://www.baseten.co/pricing/ (visited on 10/22/2026).

[119] Modal. *Pricing: Pay per Second for GPUs and CPUs*. URL: https://modal.com/pricing (visited on 10/22/2026).

[120] DeepInfra. *Simple Pricing, Deep Infrastructure*. URL: https://deepinfra.com/pricing (visited on 10/22/2026).

[121] HPC-AI Tech and ColossalAI Contributors. *ColossalAI: Making Large AI Models Cheaper, Faster and More Accessible*. 项目身份与产品边界快照，不用于本书固定引擎源码比较. URL: https://github.com/hpcaitech/ColossalAI (visited on 10/22/2026).

[122] PPIO. *PPIO：Agentic Cloud* 产品与服务. URL: https://ppio.com/ (visited on 10/22/2026).

[123] 硅基流动. *SiliconFlow* 平台简介. URL: https://docs.siliconflow.cn/docs/userguide/introduction (visited on 10/22/2026).

[124] vLLM Contributors. *vLLM: A High-Throughput and Memory-Efficient Inference and Serving Engine for LLMs*. 仅用于生态身份；源码机制以本书固定 v0.11.0 为准. URL: https://github.com/vllm-project/vllm (visited on 10/22/2026).

[125] SGLang Contributors. *SGLang: A High-Performance Serving Framework for Large Language Models and Multimodal Models*. 仅用于生态身份；源码机制以本书固定 v0.5.3.post3 为准. URL: https://github.com/sgl-project/sglang (visited on 10/22/2026).

[126] NVIDIA and TensorRT-LLM Contributors. *TensorRT LLM*. URL: https://github.com/NVIDIA/TensorRT-LLM (visited on 10/22/2026).

[127] llama.cpp Contributors. *llama.cpp: LLM Inference in C/C++*. URL: https://github.com/ggml-org/llama.cpp (visited on 10/22/2026).

[128] MLX Contributors. *MLX: An Array Framework for Apple Silicon*. URL: https://github.com/ml-explore/mlx (visited on 10/22/2026).

[129] 国家互联网信息办公室, 国家发展和改革委员会, 教育部, et al. 生成式人工智能服务管理暂行办法. 2023 年 7 月 13 日公布，2023 年 8 月 15 日施行. July 2023. URL: https://www.gov.cn/zhengce/zhengceku/202307/content_6891752.htm (visited on 10/22/2026).

[130] 国家互联网信息办公室. 数据出境安全评估办法. 国家互联网信息办公室令第 11 号，2022 年 9 月 1 日施行. July 2022. URL: https://www.cac.gov.cn/2022-07/07/c_1658811536396503.htm (visited on 10/22/2026).

[131] Zeming Lin, Halil Akin, Roshan Rao, et al. “Evolutionary-scale prediction of atomic-level protein structure with a language model”. In: *Science* 379.6637 (2023). ESM-2 up to 15B parameters; ESMFold replaces MSA with a single sequence, pp. 1123–1130. DOI: 10.1126/science.ade2574. URL: https://doi.org/10.1126/science.ade2574 (visited on 10/22/2026).

[132] NVIDIA. *cuBLAS Documentation: Results Reproducibility*. URL: https://docs.nvidia.com/cuda/cublas/index.html (visited on 10/22/2026).

[133] Thinking Machines Lab. *Defeating Nondeterminism in LLM Inference*. Blog post on batch-variant kernels; accessed 2026-10-22. 2025. URL: https://thinkingmachines.ai/blog/defeating-nondeterminism-in-llm-inference/ (visited on 10/22/2026).

[134] vLLM Contributors. *Reproducibility: vLLM v0.11.0 Documentation*. 固定版本文档，区分离线设置、在线服务与相同硬件/版本边界. URL: https://docs.vllm.ai/en/v0.11.0/usage/reproducibility.html (visited on 10/22/2026).

[135] John Jumper, Richard Evans, Alexander Pritzel, et al. “Highly accurate protein structure prediction with AlphaFold”. In: *Nature* 596 (2021), pp. 583–589. DOI: 10.1038/s41586-021-03819-2. URL: https://doi.org/10.1038/s41586-021-03819-2 (visited on 10/22/2026).

[136] Josh Abramson, Jonas Adler, Jack Dunger, et al. “Accurate structure prediction of biomolecular interactions with AlphaFold 3”. In: *Nature* 630 (2024), pp. 493–500. DOI: 10.1038/s41586-024-07487-w. URL: https://doi.org/10.1038/s41586-024-07487-w (visited on 10/22/2026).

[137] Garyk Brixi, Matthew G. Durrant, Jerome Ku, et al. “Genome modeling and design across all domains of life with Evo 2”. In: *bioRxiv* (2025). Preprint; 7B and 40B parameters, 1 million token context. DOI: 10.1101/2025.02.18.638918. (Visited on 10/22/2026).

[138] Haotian Cui, Chloe Wang, Hassaan Maan, et al. “scGPT: toward building a foundation model for single-cell multi-omics using generative AI”. In: *Nature Methods* 21 (2024), pp. 1470–1480. DOI: 10.1038/s41592-024-02201-0. URL: https://doi.org/10.1038/s41592-024-02201-0 (visited on 10/22/2026).

[139] Amil Merchant, Simon Batzner, Samuel S. Schoenholz, Muratahan Aykol, Gowoon Cheon, and Ekin Dogus Cubuk. “Scaling deep learning for materials discovery”. In: *Nature* 624 (2023). GNoME; 2.2 million crystal structures screened with graph networks, pp. 80–85. DOI: 10.1038/s41586-023-06735-9. URL: https://doi.org/10.1038/s41586-023-06735-9 (visited on 10/22/2026).

[140] Ilyes Batatia, Dávid Péter Kovács, Gregor N. C. Simm, Christoph Ortner, and Gábor Csányi. “MACE: Higher Order Equivariant Message Passing Neural Networks for Fast and Accurate Force Fields”. In: *Advances in Neural Information Processing Systems (NeurIPS)*. Vol. 35. 2022. arXiv: 2206.07697. URL: https://arxiv.org/abs/2206.07697.

[141] Simon Batzner, Albert Musaelian, Lixin Sun, et al. “E(3)-equivariant graph neural networks for data-efficient and accurate interatomic potentials”. In: *Nature Communications* 13 (2022), p. 2453. DOI: 10.1038/s41467-022-29939-5. URL: https://doi.org/10.1038/s41467-022-29939-5 (visited on 10/22/2026).

[142] Chi Chen and Shyue Ping Ong. "A universal graph deep learning interatomic potential for the periodic table". In: *Nature Computational Science* 2 (2022), pp. 718–728. DOI: 10.1038/s43588-022-00349-3. URL: https://doi.org/10.1038/s43588-022-00349-3 (visited on 10/22/2026).

[143] Remi Lam, Alvaro Sanchez-Gonzalez, Matthew Willson, et al. "Learning skillful medium-range global weather forecasting". In: *Science* 382.6677 (2023), pp. 1416–1421. DOI: 10.1126/science.adi2336. URL: https://doi.org/10.1126/science.adi2336 (visited on 10/22/2026).

[144] Cristian Bodnar, Wessel P. Bruinsma, Ana Lucic, et al. *A Foundation Model for the Earth System*. 2024. arXiv: 2405.13063v3 [cs.LG]. URL: https://arxiv.org/abs/2405.13063v3 (visited on 10/22/2026).

[145] Jiaming Tang, Yilong Zhao, Kan Zhu, Guangxuan Xiao, Baris Kasikci, and Song Han. *Quest: Query-Aware Sparsity for Efficient Long-Context LLM Inference*. 2024. arXiv: 2406.10774v2 [cs.LG]. URL: https://arxiv.org/abs/2406.10774v2 (visited on 10/22/2026).

[146] Mitchell Wortsman, Peter J. Liu, Lechao Xiao, et al. *Small-scale proxies for large-scale Transformer training instabilities*. 2023. arXiv: 2309.14322v2 [cs.LG]. URL: https://arxiv.org/abs/2309.14322v2 (visited on 10/22/2026).

[147] Jiayi Yuan, Hao Li, Xinheng Ding, et al. *Understanding and Mitigating Numerical Sources of Nondeterminism in LLM Inference*. 2025. arXiv: 2506.09501v2 [cs.LG]. URL: https://arxiv.org/abs/2506.09501v2 (visited on 10/22/2026).

[148] Nelson F. Liu, Kevin Lin, John Hewitt, et al. *Lost in the Middle: How Language Models Use Long Contexts*. 2023. arXiv: 2307.03172v3 [cs.CL]. URL: https://arxiv.org/abs/2307.03172v3 (visited on 10/22/2026).

[149] Charlie Snell, Jaehoon Lee, Kelvin Xu, and Aviral Kumar. *Scaling LLM Test-Time Compute Optimally can be More Effective than Scaling Model Parameters*. 2024. arXiv: 2408.03314v1 [cs.LG]. URL: https://arxiv.org/abs/2408.03314v1 (visited on 10/22/2026).

[150] Reiner Pope, Sholto Douglas, Aakanksha Chowdhery, et al. *Efficiently Scaling Transformer Inference*. 2022. arXiv: 2211.05102v1 [cs.LG]. URL: https://arxiv.org/abs/2211.05102v1 (visited on 10/22/2026).