

DeepSeek-V4 深度技术解读笔记

基于《DeepSeek-V4: Towards Highly Efficient Million-Token Context Intelligence》preview 技术报告。本笔记假设读者已熟悉 V3/V3.2 架构，重点解读 V4 的新机制（mHC、CSA、HCA、Muon、OPD、Anticipatory Routing 等）。

0. TL;DR 速览

模型规模

模型	总参	激活参	层数	隐层 d	注意力头 n_h	MoE 专家 (shared / routed / 激活)	MTP 深度
V4-Pro	1.6T	49B	61	7168	128	1 / 384 / 6	1
V4-Flash	284B	13B	43	4096	64	1 / 256 / 6	1

三大架构创新一句话

- Hybrid Attention (CSA + HCA)**：CSA 沿序列维度做 $1/m$ 压缩后再跑 DeepSeek Sparse Attention；HCA 做更激进的 $1/m'$ ($m' \gg m$) 密集压缩；两者层间交错排布。
- Manifold-Constrained Hyper-Connections (mHC)**：把 HC 的残差变换矩阵 B 约束在 Birkhoff 多面体（双随机矩阵流形）上，保证 $\|B\|_2 \leq 1$ ，从而在深层堆叠下稳定信号传播。
- Muon 优化器**（替代 AdamW 为主力优化器）：基于 Newton-Schulz 迭代的正交化更新，收敛更快更稳。

1M 上下文效率（相对 V3.2）

- V4-Pro：单 token FLOPs **27%**、KV cache **10%**
- V4-Flash：单 token FLOPs **10%**、KV cache **7%**

核心基准结论

- Knowledge**：V4-Pro-Max 在 SimpleQA-Verified **57.9** 领先所有开源模型约 20 个百分点，Chinese-SimpleQA **84.4**；距 Gemini-3.1-Pro 仍有差距但已大幅缩小。
- Reasoning**：Codeforces Rating **3206**（领先 GPT-5.4 的 3168）、HMMT 2026 Feb 95.2、Apex Shortlist 90.2、IMOAnswerBench 89.8；Lean 形式化 Putnam-2025 **120/120** 满分。
- Agent**：与 K2.6、GLM-5.1 对标，Terminal Bench 2.0 67.9、SWE-Verified 80.6。
- Long-Context**：MRCCR 8-needle 在 1M 仍保 0.59 MMR；CorpusQA 1M 在真实场景超过 Gemini-3.1-Pro。
- Pro vs Flash**：Flash 在知识上因参数小而掉队，但 reasoning 配大 thinking budget 与 Pro 可比；agent 任务高难度时仍落后 Pro。

1. 架构（Architecture）

V4 保留 Transformer 与 MTP 模块，其上引入三项关键升级（mHC / Hybrid Attention / Muon），MoE 框架沿用 DeepSeekMoE，MTP 配置与 V3 完全一致。

1.1 继承自 V3 的设计及其微调

项	V3 / V3.2	V4
MoE 亲和度激活	Sigmoid(\cdot)	$\sqrt{\text{Softplus}(\cdot)}$
前几层 FFN	Dense	前 3 个 MoE 层改用 Hash routing
路由节点数约束	有	取消
负载均衡	auxiliary-loss-free	auxiliary-loss-free + 序列级 balance loss（权重 0.0001）
Bias 更新速度	—	0.001
MTP	同 V3	同 V3（深度 1）

问题 → 方案 → 效果：Sigmoid 在极稀疏场景下梯度饱和； $\sqrt{\text{Softplus}}$ 具有更平滑、非饱和的形状。Hash routing 在浅层稳定 token-expert 映射，避免早期路由抖动向后累积。取消路由节点数限制是为了给更细粒度的 EP 调度留空间（配合 §2.1 的 wave 调度）。

1.2 Manifold-Constrained Hyper-Connections (mHC)

1.2.1 标准 HC 回顾

残差流宽度从 \mathbb{R}^d 扩展为 $\mathbb{R}^{n_{\text{hc}} \times d}$ 。记 $X_l = [x_{l,1}; \dots; x_{l,n_{\text{hc}}}]^T \in \mathbb{R}^{n_{\text{hc}} \times d}$ ，标准 HC 引入三个线性映射 $A_l \in \mathbb{R}^{1 \times n_{\text{hc}}}$ ， $B_l \in \mathbb{R}^{n_{\text{hc}} \times n_{\text{hc}}}$ ， $C_l \in \mathbb{R}^{n_{\text{hc}} \times 1}$ ：

$$X_{l+1} = B_l X_l + C_l \mathcal{F}_l(A_l X_l) \quad (1)$$

其中 $A_l X_l \in \mathbb{R}^d$ 才是实际送入层 \mathcal{F}_l （如 MoE）的输入。 $n_{\text{hc}} \ll d$ ，因此开销极低。****问题****：堆叠多层后 HC 出现严重数值不稳定，阻碍规模化。

1.2.2 Birkhoff 流形约束

核心创新是把 B_l 约束到双随机矩阵构成的 Birkhoff 多面体 \mathcal{M} ：

$$B_l \in \mathcal{M} := \{M \in \mathbb{R}^{n \times n} \mid M \mathbf{1}_n = \mathbf{1}_n, \mathbf{1}_n^T M = \mathbf{1}_n^T, M \geq 0\} \quad (2)$$

由于双随机矩阵的谱范数 $\|B_l\|_2 \leq 1$ ，残差变换是****非扩张（non-expansive）****的，正向/反向传播都更稳定；而 \mathcal{M} 对矩阵乘法封闭，保证深堆叠下稳定性不衰减。 A_l, C_l 通过 Sigmoid 约束到非负有

界范围，避免信号抵消。

1.2.3 动态参数化

参数分解为「动态（输入相关）+ 静态（输入无关）」两部分。令 $\hat{X}_l = \text{RMSNorm}(\text{vec}(X_l)) \in \mathbb{R}^{1 \times n_{\text{hc}} d}$:

$$\tilde{A}_l = \alpha_l^{\text{pre}} \cdot (\hat{X}_l W_l^{\text{pre}}) + S_l^{\text{pre}} \quad (3)$$

$$\tilde{B}_l = \alpha_l^{\text{res}} \cdot \text{Mat}(\hat{X}_l W_l^{\text{res}}) + S_l^{\text{res}} \quad (4)$$

$$\tilde{C}_l = \alpha_l^{\text{post}} \cdot (\hat{X}_l W_l^{\text{post}})^T + S_l^{\text{post}} \quad (5)$$

其中 $W_l^{\text{pre}}, W_l^{\text{post}} \in \mathbb{R}^{n_{\text{hc}} d \times n_{\text{hc}}}$, $W_l^{\text{res}} \in \mathbb{R}^{n_{\text{hc}} d \times n_{\text{hc}}^2}$; S_* 为静态 bias; α_* 为可学习门控, **初始化为小值**。

1.2.4 施加约束

对 A, C 施 Sigmoid (C 额外 $\times 2$) :


$$A_l = \sigma(\tilde{A}_l) \quad (6)$$

$$C_l = 2\sigma(\tilde{C}_l) \quad (7)$$

对 \tilde{B}_l : 先 $M^{(0)} = \exp(\tilde{B}_l)$ 保正, 再用 **Sinkhorn-Knopp 迭代** 投影到 Birkhoff 流形 :

$$M^{(t)} = \mathcal{T}_r \left(\mathcal{T}_c \left(M^{(t-1)} \right) \right) \quad (8)$$

其中 $\mathcal{T}_r, \mathcal{T}_c$ 分别为行/列归一化。实际取 $t_{\text{max}} = 20$ 次迭代; $n_{\text{hc}} = 4$ 。

 论文未深入展开：为何 Birkhoff 流形既足够保证稳定性，又不过度限制表达能力；以及 Sinkhorn 20 步是否可被更少迭代加 Padé 近似替代。详细推导在 Xie et al., 2026 的专门论文中。

关键点

1. $\|B\|_2 \leq 1$ 是 mHC 稳定性的数学根源；Sigmoid 化的 A, C 补上 "非负有界" 的信号通道。
2. 动态 + 静态参数化：动态部分让残差变换对输入敏感，静态 bias + 小初始化 α 让训练初期接近 identity 残差。
3. $n_{\text{hc}} = 4$ 在扩展表达能力与额外开销间取得平衡；Sinkhorn 20 步是实用折衷。

1.3 混合注意力：CSA + HCA

CSA 与 HCA 层间交错排布，前几层特殊：V4-Flash 前 2 层用纯 SWA；V4-Pro 前 2 层用 HCA；之后均为 CSA/HCA 交错。

1.3.1 Compressed Sparse Attention (CSA)

第一步：带 overlap 的 token 级压缩。 给定隐状态 $H \in \mathbb{R}^{n \times d}$ ，生成两套 KV 与权重：

$$C^a = H \cdot W^{aKV}, \quad C^b = H \cdot W^{bKV} \quad (9)$$

$$Z^a = H \cdot W^{aZ}, \quad Z^b = H \cdot W^{bZ} \quad (10)$$

每 m 个 KV 压缩为 1 个，使用可学习位置偏置 $B^a, B^b \in \mathbb{R}^{m \times c}$ ：

$$\left[S_{mi:m(i+1)-1}^a; S_{m(i-1):mi-1}^b \right] = \text{Softmax}_{\text{row}} \left(\left[Z_{mi:m(i+1)-1}^a + B^a; Z_{m(i-1):mi-1}^b + B^b \right] \right) \quad (11)$$

$$C_i^{\text{Comp}} = \sum_{j=mi}^{m(i+1)-1} S_j^a \odot C_j^a + \sum_{j=m(i-1)}^{mi-1} S_j^b \odot C_j^b \quad (12)$$

当 $i = 0$ 时 $Z_{m(i-1):mi-1}^b$ 用 $-\infty$ padding, $S_{m(i-1):mi-1}^b$ 用 0 padding。

Overlap 机制： 每个 C_i^{Comp} 由 $2m$ 个 KV 聚合，但 C^b 与 C^a 在相邻压缩块间的索引**交叠**，因此有效压缩率仍为 $1/m$ （并非 $1/(2m)$ ），同时相邻压缩块共享上下文，缓解压缩边界的信息断裂。

第二步：Lightning Indexer（DSA 风格稀疏选择）。 先按相同方式压缩得到 $K^{\text{IComp}} \in \mathbb{R}^{(n/m) \times c^I}$ 。查询侧走低秩通路：

$$c_t^Q = h_t \cdot W^{DQ} \quad (13)$$

$$\left[q_{t,1}^I; \dots; q_{t,n_h^I}^I \right] = q_t^I = c_t^Q \cdot W^{IUQ} \quad (14)$$

其中 $c_t^Q \in \mathbb{R}^{d_c}$ 是**查询压缩潜向量**（ d_c 即 query 压缩维度）， W^{IUQ} 把它升到 $c^I n_h^I$ 。索引得分：

$$\left[w_{t,1}^I; \dots; w_{t,n_h^I}^I \right] = w_t^I = h_t \cdot W^w \quad (15)$$

$$I_{t,s} = \sum_{h=1}^{n_h^I} w_{t,h}^I \cdot \text{ReLU} \left(q_{t,h}^I \cdot K_s^{\text{IComp}} \right) \quad (16)$$

$$\mathcal{C}_t^{\text{SprsComp}} = \{C_s^{\text{Comp}} \mid I_{t,s} \in \text{Top-k}(I_{t,:})\} \quad (17)$$

ReLU 打分 + 加权求和设计继承自 V3.2 DSA。

第三步：Shared KV MQA。 查询复用同一个 c_t^Q ：

$$[q_{t,1}; \dots; q_{t,n_h}] = q_t = c_t^Q \cdot W^{UQ} \quad (18)$$

$$o_{t,i} = \text{CoreAttn} \left(\text{query} = q_{t,i}, \text{key} = C_t^{\text{SprsComp}}, \text{value} = C_t^{\text{SprsComp}} \right) \quad (19)$$

注意 Key 与 Value 是同一向量（MQA 的极端化：不仅头间共享， $K = V$ ）。

第四步：Grouped Output Projection。 直接把 $o_t \in \mathbb{R}^{cn_h}$ 投回 \mathbb{R}^d 开销过高。先分 g 组，每组 $o_{t,i}^G \in \mathbb{R}^{cn_h/g}$ 投到 d_g 维 ($d_g < cn_h/g$)，再把 $[o_{t,1}^{G'}; \dots; o_{t,g}^{G'}] \in \mathbb{R}^{d_g g}$ 整体投到 \mathbb{R}^d 得 \hat{o}_t 。

1.3.2 Heavily Compressed Attention (HCA)

与 CSA 的核心差异：

- 压缩率 $m' \gg m$ ；
- 无 overlap：只用单套 W^{KV}, W^Z ，没有 a/b 两套；
- 压缩后直接 dense attention（不做 top-k 稀疏）。

$$C = H \cdot W^{KV}, \quad Z = H \cdot W^Z \quad (20, 21)$$

$$S_{m'i:m'(i+1)-1} = \text{Softmax}_{\text{row}} (Z_{m'i:m'(i+1)-1} + B) \quad (22)$$

$$C_i^{\text{Comp}} = \sum_{j=m'i}^{m'(i+1)-1} S_j \odot C_j \quad (23)$$

查询与投影流程与 CSA 相同：

$$c_t^Q = h_t \cdot W^{DQ} \quad (24)$$

$$[q_{t,1}; \dots; q_{t,n_h}] = q_t = c_t^Q \cdot W^{UQ} \quad (25)$$

$$o_{t,i} = \text{CoreAttn} (\text{query} = q_{t,i}, \text{key} = C^{\text{Comp}}, \text{value} = C^{\text{Comp}}) \quad (26)$$

为什么 HCA 不需要 sparse？因为 $m' = 128$ 已经足够大，压缩后的序列长度本身就很短，dense 运算可以接受。

1.3.3 其他关键细节（CSA/HCA 共用）

- **Q/KV RMSNorm**：在核注意力前，对每个 query head 和（唯一的）压缩 KV head 单独做 RMSNorm，抑制 attention logits 爆炸。
- **Partial RoPE**：对 query、KV entry 的最后 64 维施加 RoPE。由于 KV 同时充当 Key 与 Value，核输出 $\{o_{t,i}\}$ 会携带 KV 的绝对位置 embedding，于是对 $o_{t,i}$ 再施加位置 $-i$ 的 RoPE（最后 64 维），使最终输出只保留 "query 与 KV 之间的相对距离"。这是一个很巧妙的补偿步。
- **Sliding Window 辅助分支**：由于压缩/稀疏设计使 query 无法访问自己所在压缩块内的细粒度信息，且最近 token 对 query 往往更相关，CSA/HCA 都并联一条 SWA 分支，传递最近 n_{win} 个未压缩的 KV。V4-Flash 与 V4-Pro 均取 $n_{\text{win}} = 128$ 。
- **Attention Sink**：引入可学习 sink logits $\{z'_h\}$ ，在 softmax 分母额外加 $\exp(z'_h)$ ：

$$s_{h,i,j} = \frac{\exp(z_{h,i,j})}{\sum_k \exp(z_{h,i,k}) + \exp(z'_h)} \quad (27)$$

允许某个 head 的总注意力权重不等于 1（甚至接近 0），避免强制分配无意义注意力。

1.3.4 效率讨论

混合存储与低精度：

- **KV cache 混合精度**：RoPE 的 64 维用 BF16，其他维度用 FP8 → 相比纯 BF16 减半 KV cache 体积。
- **Lightning Indexer 的注意力计算用 FP4**，加速长序列评分。
- **Attention top-k** 比 V3.2 更小（Flash=512，Pro=1024），中短序列效率也提升。
- 相对 **BF16 GQA8 (head_dim=128)** 基线，V4 系列 1M 上下文下 KV cache 降至约 2%。
- 相对 V3.2（本身已是高效基线）：V4-Pro 单 token FLOPs 27%、KV 10%；V4-Flash 更是降到 10%/7%（FLOPs 按等效 FP8 口径）。

🔑 关键点

1. CSA 的 " $1/m$ 压缩 + top-k 稀疏" 两级削减是 1M 上下文的计算支柱；HCA 的 " $1/m'$ 重压缩 + dense" 是 KV cache 压缩的主要来源。
2. Overlap 机制让相邻压缩块共享 $2m$ 个 token 上下文，是缓解「压缩信息截断」的关键。
3. 混合精度 KV（BF16 RoPE + FP8 其他 + FP4 indexer）再压一层效率。
4. Partial RoPE 对输出施加位置 $-i$ ，把"绝对位置"改造为"相对距离"，是非常优雅的补偿。
5. SWA 辅助分支 + Attention Sink 覆盖压缩/稀疏的两大盲点：局部细粒度依赖与"无可分配注意力"场景。

1.4 Muon 优化器

Algorithm 1 完整流程：

```
Require: 学习率  $\eta$ , 动量  $\mu$ , 权重衰减  $\lambda$ , 更新重缩放因子  $\gamma$ 
每步  $t$ :
  对每个逻辑独立权重  $W \in \mathbb{R}^{n \times m}$ :
     $G_t = \nabla L_t(W_{t-1})$  # 计算梯度
     $M_t = \mu \cdot M_{t-1} + G_t$  # 累积动量
     $O'_t = \text{HybridNewtonSchulz}(\mu \cdot M_t + G_t)$  # Nesterov + Newton-Schulz
     $O_t = O'_t \cdot \sqrt{\max(n, m)} \cdot \gamma$  # RMS 重缩放
     $W_t = W_{t-1} \cdot (1 - \eta \cdot \lambda) - \eta \cdot O_t$  # 权重衰减 + 更新
```

AdamW 保留的模块：embedding、prediction head、mHC 的静态 bias S_* 与门控因子 α_* 、所有 RMSNorm 的 weight。其余（mHC 的动态 $W^{\text{pre/res/post}}$ 、CSA/HCA 的所有投影、MoE 专家权重等）一律走 Muon。

Nesterov + weight decay + RMS 重缩放：Nesterov 即 $O'_t = \text{NS}(\mu M_t + G_t)$ ；RMS 重缩放到 0.18（见 §3.2.2）是为了复用 AdamW 的学习率超参，避免重调。

Hybrid Newton-Schulz（共 10 步）：对输入先 $M_0 = M / \|M\|_F$ 归一化，然后迭代

$$M_k = aM_{k-1} + b(M_{k-1}M_{k-1}^T)M_{k-1} + c(M_{k-1}M_{k-1}^T)^2M_{k-1} \quad (28)$$

- 前 8 步： $(a, b, c) = (3.4445, -4.7750, 2.0315)$ ，激进地把奇异值推向 1。
- 后 2 步： $(a, b, c) = (2, -1.5, 0.5)$ ，温和地把奇异值精确稳定在 1。

两段策略权衡「快速收敛」与「末端数值精度」。

不用 QK-Clip 的原因：V4 在 query / KV entry 上已有 RMSNorm (§1.3.3)，天然抑制 logits 爆炸；QK-Clip 在 Moonshot Muon 方案中是必须的，但 V4 架构下冗余。

🔑 关键点

1. Muon 并非替代所有优化器——embedding / head / bias / gate / RMSNorm 仍然 AdamW，这是**梯度几何结构**的差异使然（Muon 依赖「完整梯度矩阵」做正交化）。
2. RMS 重缩放到 0.18 让 Muon 可以直接复用 AdamW 的 LR，迁移成本大幅降低。
3. 两段 Newton-Schulz 系数是 Muon 论文调优经验的延续；BF16 下 NS 仍稳定（见 §2.5.1 的分桶实现）。

2. 基础设施（General Infrastructures）

2.1 细粒度 EP 通信-计算重叠

MoE 层按操作拆成 4 个主要阶段：**Dispatch（通信）→ Linear-1（计算）→ Activation (SwiGLU + FP8 Cast) → Linear-2（计算）→ Combine（通信）**。

论文的关键洞察：单层 MoE 中**通信时间 < 计算时间**，因此只要融合好流水，通信可以被计算完全 hide，计算才是最终瓶颈。这意味着系统可以容忍更低的**互联带宽**而不损失端到端性能。

三种方案对比：

方案	重叠粒度	理论加速
Naive	无重叠，串行 Dispatch → L1 → L2 → Combine	1.0×
Comet (Zhang et al. 2025b)	Dispatch ↔ L1、L2 ↔ Combine 分别重叠	1.42×
V4（MegaMoE）	专家切成 wave，每 wave 内 Dispatch / L1 / L2 / Combine 全流水	1.92×

Wave 调度：把 expert 切成若干 wave，一旦 wave 内所有 expert 通信就绪，就立即开启下一 wave 的计算而不管其他。稳态下当前 wave 的计算、下一 wave 的 token 传输、已完成 expert 的结果回传**并发进行**，在 expert 间形成细粒度流水。这对 **RL rollout / 高并发 agent 服务**（长尾小 batch）尤其有利。

实测加速（相对强基线非融合实现）：

- 通用推理工作负载：1.50 ~ 1.73×
- 延迟敏感场景（RL rollout、高速 agent 服务）：最高 1.96×

开源：CUDA 实现的 mega-kernel **MegaMoE**（DeepGEMM 组件）。

对硬件厂商的 4 条建议：

1. **C/B 比（Computation-Communication Ratio）**：设峰值算力 C 、互联带宽 B 、计算量 V_{comp} 、通信量 V_{comm} ，通信完全隐藏条件 $C/B \leq V_{\text{comp}}/V_{\text{comm}}$ 。V4-Pro 每 token-expert 对需 $6hd$ FLOPs（SwiGLU 的 gate/up/down 三矩阵）但只需 $3h$ bytes（FP8 Dispatch + BF16 Combine）通信，简化为

$$\frac{C}{B} \leq 2d = \mathbf{6144 \text{ FLOPs/Byte}}$$

即 1 GB/s 带宽可 hide 6.1 TFLOP/s 算力。超过此平衡点后，加带宽收益递减。呼吁未来硬件**瞄准这种平衡点**而非无脑堆带宽。

2. **功耗预算**：极端 kernel fusion 让 compute/memory/network 同时满载，功耗成为关键瓶颈；未来硬件要为全并发负载预留功率 headroom。

3. **通信原语**：采用 **pull-based**（每 GPU 主动远程读），避免细粒度 push 的高 notification latency。若未来硬件提供更低 cross-GPU signaling latency，push 会更自然。
4. **激活函数**：建议用无 **exp/division** 的低开销 **element-wise** 激活替代 SwiGLU，减轻 post-GEMM 压力；同时去掉 gate projection，相同参数预算下 intermediate 维度 d 更大，进一步放松带宽压力。

2.2 TileLang：灵活高效的 Kernel 开发

V4 精巧的架构如果直接写 Torch ATen 会产生数百个细粒度算子。V4 用 **TileLang** (Wang et al., 2026) 作为 DSL 写融合 kernel，在开发效率与运行效率间取得平衡。

2.2.1 Host Codegen

问题：加速器越来越强，CPU-side 编排开销（shape 校验、参数 marshaling 等）成为小 kernel 的利用率瓶颈；Python host 代码往往引入几十至几百微秒固定开销。

方案：在 IR 层面联合生成 device kernel 和 lightweight host launcher，把数据类型、rank/shape 约束、stride/layout 假设等元数据从 frontend 解析嵌入；launcher 下沉到基于 **TVM-FFI** 的 host 源码，利用其紧凑调用约定与零拷贝 tensor 互操作。

效果：CPU-side 校验开销从数十至数百 μs 降到 $< 1 \mu s$ /调用。

2.2.2 Z3 SMT 求解器集成

问题：TileLang kernel 充斥复杂 tensor index 算术，编译 pass（layout inference、memory hazard 检测、bound analysis）需要强形式化整数分析能力。

方案：集成 **Z3** SMT 求解器，把 TileLang 整数表达式翻译到 **QF_NIA**（quantifier-free non-linear integer arithmetic）。基于 ILP 求解器可无缝处理标准线性整数表达式，而其非线性能力可处理变量形状下的 vectorization 等高级挑战。

效果：在合理资源限制下，Z3 把编译耗时控制在几秒内，显著提升 vectorization、barrier insertion、code simplification 等 pass 的优化能力。

2.2.3 数值精度与 bitwise 复现

生产场景下数值正确性和复现性与吞吐同等重要：

- **默认关闭 fast-math**，精度近似仅通过显式 opt-in 前端算子提供 (`T.__exp`, `T.__log`, `T.__sin`)。
- 需要严格 IEEE-754 时提供带显式 rounding 模式的 intrinsic：`T.ieee_fsqrt`, `T.ieee_fdiv`, `T.ieee_add`。
- 与 NVCC 对齐代数化简和 lowering 规则以避免 bit-level 差异；`T.annotate_layout` 允许锁定 layout 相关 lowering 决策，使累加顺序与手写 CUDA 基线 **bit-identical**。

2.3 Batch-Invariant 与确定性 kernel 库

设计目标：pre-train / post-train / inference 三段 pipeline 实现 bitwise 一致，便于 debug、稳定性分析、post-training 行为一致性。

2.3.1 Batch Invariance

Batch invariance 要求同一 token 输出与其在 batch 中位置无关。主要挑战：

- **Attention**：放弃 split-KV（会破坏 batch invariance），但这会引入严重的 wave-quantization 问题。方案是 **dual-kernel 策略**：
 - **Kernel 1**：单 SM 处理整条序列，保证满载 wave 高吞吐；
 - **Kernel 2**：为尾部 partial wave 服务，用多 SM 处理单序列降低 latency，并利用 **thread-block cluster distributed shared memory** 高速跨 SM 交换数据；
 - 两 kernel 精心对齐累加路径保证 bitwise 一致。

最终 batch-invariant decoding 的开销"可忽略"。

- **Matrix Multiplication**：放弃 cuBLAS（无法保证 batch invariance），全面替换为 **DeepGEMM**。极小 batch size 传统上用 **split-k** 提升性能，但 split-k 不保 batch invariance。V4 大多数场景放弃 split-k，通过一套其他优化补齐性能缺口。

2.3.2 Determinism

非确定性通常来自原子加法的累加顺序不定，主要在反向传播：

- **Attention Backward**：稀疏 attention 反传传统上用 `atomicAdd` 累加 KV 梯度（浮点加法非结合 → 非确定）。V4 为每个 SM 分配独立累加 buffer，最后做一次全局确定性求和。
- **MoE Backward**：多 SM 跨 rank 并发写同一 buffer 时协商写位置会引入非确定。V4 在单 rank 内做 token-order 预处理，跨 rank 做 buffer 隔离，保证 expert parallelism 的发送结果和 MoE backward 累加顺序都确定。
- **mHC 中的 GEMM**：mHC 存在一个 output 维度仅 24 的 GEMM，极小 batch 下不得不用 split-k。V4 的做法是每个 split 输出到独立位置，再用一个 kernel 做确定性 reduction，兼顾性能与确定性。

2.4 FP4 (MXFP4) 量化感知训练 (QAT)

应用范围：

1. **MoE expert 权重**（GPU 显存占用大头）；
2. **CSA indexer 的 QK path**：QK 激活缓存、加载、相乘全部 FP4，加速长序列的 attention score 计算；
3. **Index scores $I_{:,}$** 从 FP32 量化到 BF16，top-k 选择器获得 **2× 加速**，同时保持 **99.7% 的 KV entry 召回率**。

FP4 → FP8 无损反量化：对 MoE 权重，optimizer 保存 FP32 master weights，量化到 FP4，再反量化回 FP8 做计算。之所以无损：FP8 (E4M3) 比 FP4 (E2M1) 多 2 个指数位，动态范围更大。只要每个 FP8 量化块 (128×128 tile) 内所有 FP4 sub-block (1×32 tile) 的 scale 因子比值不超过某阈值，fine-grained scale 信息就可完全吸收到 FP8 扩展动态范围中。实验验证当前权重满足此条件，因

此整个 QAT pipeline 可以零改动复用既有 FP8 训练框架。反向按 STE (Straight-Through Estimator) 直通 FP32 master, 也无需再量化转置权重。

推理/rollout 阶段：直接使用**真实 FP4 量化权重** (非模拟量化), 保证采样行为与线上部署一致, 并实现真实的 kernel 显存加载加速与内存节省。

2.5 训练框架

2.5.1 Muon 的高效实现

挑战：Muon 需要**完整梯度矩阵**做正交化, 与 ZeRO (按元素切分参数) 天然冲突。

Dense 参数的 ZeRO bucket 分配：

- **限制 ZeRO 并行度上限**, 用 **knapsack 算法**把参数矩阵分配给各 rank, 使每 rank 负载大致均衡。
- 每 rank 的 bucket **padding 到跨 rank 最大 bucket size** 以支持高效 reduce-scatter ; 每 rank 最多管 5 个矩阵时, padding overhead < 10% 内存。
- 当 data parallelism 规模超过 ZeRO 上限时, **在多余 DP 组上冗余计算 Muon 更新**, 用算力换总 bucket 显存。

MoE 参数的独立优化：每个 expert 独立处理。先把所有 expert 所有层的 SwiGLU down projection 扁平化拼接, 再拼 up projection 与 gate ; pad 后均匀切到所有 rank, **不切任何逻辑独立矩阵**。由于 expert 数量大, padding overhead 可忽略。

批执行 Newton-Schulz：每 rank 上同形状连续参数**自动 merge**, 批量执行 NS 迭代提升硬件利用率。实验发现 NS 在 **BF16 矩阵乘下仍稳定**, 于是进一步把跨 DP rank 的 **MoE 梯度量化到 BF16** (stochastic rounding), 通信量减半。

BF16 梯度聚合的数值稳健性：传统 tree/ring reduce-scatter 在低精度下会累积误差, V4 改用 **two-phase approach**：

1. **All-to-all 交换 local gradient** ;
2. 每 rank 在 **FP32** 做本地求和。

2.5.2 mHC 的实现优化

mHC 增加了激活显存和 pipeline 阶段间通信量。V4 用三策略抑制开销：

1. **训练/推理专用融合 kernel** ;
2. **选择性重计算**：重计算大多数层间 hidden state 和所有 normalized layer input, 但**避免重计算计算密集操作** ;
3. **调整 DualPipe 1F1B 重叠方案**以容纳增加的 pipeline 通信, 同时 mHC 内某些操作并发执行。

最终效果：mHC wall-time overhead 仅 **6.7% of overlapped 1F1B stage**。

2.5.3 长上下文 Context Parallelism

传统 CP 按序列维度切分，每 rank 管连续 s tokens。这对 CSA/HCA 的压缩机制有两个挑战：

- 样本是多序列 pack 而成，每序列独立压缩（尾部不足 m 的 token 丢弃），故各 rank 的**压缩 KV 长度不一且通常 $< s/m$** ；
- 压缩需要 m 连续 KV，**可能跨越相邻 CP rank 边界**。

两阶段通信：

1. **Stage 1**：每 rank i 向 rank $i + 1$ 发送其**最后 m 个未压缩 KV**；rank $i + 1$ 把接收到的部分与本地 s 个未压缩 KV 一起压缩，产生**固定长度 $s/m + 1$ 的压缩 entries**（含 padding）。
2. **Stage 2**：跨所有 CP rank 做 **all-gather**，再用 **fused select-and-pad** 算子重组为总长度 $cp_size \cdot s/m$ 的完整压缩 KV 集合，padding 放尾部。

对 HCA 和 CSA 的 indexer，每个 query token 可见的压缩 KV 范围可规则预计算；对 CSA 的 sparse attention，top-k 选择器**显式指定**可见压缩 KV 索引。

2.5.4 扩展自动微分实现 tensor 级 activation checkpointing

问题：传统 activation checkpointing 粒度是整个 module，trade-off 不细；手写整层 forward/backward 可以精细控制但失去 autograd 便利。

方案：实现 **tensor 级 activation checkpointing with autograd support**。开发者只需写 forward，在需要的 tensor 上加注解，框架用 **TorchFX** 追踪完整计算图，对每个注解的 tensor **反向遍历识别最小重计算子图**，并在相应梯度计算前**插入该子图**。

无额外开销：重计算通过直接释放注解 tensor 的 GPU 内存并重用重计算 tensor 的 storage pointer 实现，无 memory copy。由于 graph tracing 是具体执行的，storage pointer 可追踪，能**自动去重**共享 storage 的 tensor（例如 reshape 的 input/output）。

2.6 推理框架

主要差异在 KV cache 管理（详见下）。

2.6.1 KV Cache 结构与管理

Hybrid attention 引入多种异构 KV entry：

- Lightning indexer 的额外维度（embedding size 与主 attention 不同）；
- CSA / HCA 分别 $1/m$ 、 $1/m'$ 压缩后的长度不同；
- SWA 层有独立的 cache 大小和 hit / eviction 策略；
- 压缩分支中**尾部不足 m 的 token** 需缓存原始 hidden state 待未来凑齐 m 个后压缩。

违反 PagedAttention 假设的两个关键障碍：

1. 缓存策略多样（如 SWA 的滑窗淘汰）；
2. 高性能 attention kernel 的对齐约束。

解决方案（见图 6 布局）：

KV Cache 分两部分：

- **Classical KV Cache**：存 CSA 的 indexer KV 与主 KV、HCA 的 KV。每 cache block 覆盖 $\text{lcm}(m, m')$ 个原始 token，产生 $k_1 = \text{lcm}(m, m')/m$ 个 CSA 压缩 token 和 $k_2 = \text{lcm}(m, m')/m'$ 个 HCA 压缩 token。每 request 分配多个 block。
- **State Cache**（应对策略多样性挑战）：把 SWA 与压缩分支的未压缩尾 token 视为状态空间模型的状态，由当前位置唯一决定。预分配固定大小的 state cache pool，动态分配给各序列。block 内 SWA 段存最近 n_{win} 个 KV，CSA/HCA 段存尚未压缩的尾状态。

稀疏 attention kernel 协同设计（应对对齐挑战）：传统高性能 kernel 假设每 block 固定 B tokens；V4 的高性能 sparse attention kernel 允许每层 block 中 token 数量可变而不损性能。block 原始 token 数可以是 $\text{lcm}(m, m')$ 的任意倍数，padding block 以对齐 cache line 还能进一步提升性能。

2.6.2 磁盘 KV Cache 存储

对 shared-prefix 请求，on-disk KV cache 可消除重复 prefill。

CSA / HCA 压缩 KV：全部存盘。命中 prefix 时读取对应压缩 KV 直到最后完整压缩块；尾部不足一个压缩块的部分仍需重算（因为未压缩 KV 不存盘）。

SWA KV 处理（未压缩 + 每层都有 → 量约为压缩 KV 的 8 倍）。三种策略按存储/计算 trade-off 选择：

策略	存 储	计算 冗余	说明
Full SWA Caching	最大	零冗余	全存。读时只取最后 n_{win} 个即可。SSD 下访问模式失衡，对写密集型不友好。
Periodic Checkpointing	中	中	每 p 个 token 存一次（存最近 n_{win} 个 SWA KV）。命中时加载最近 checkpoint，重算剩余尾 token。 p 可调。
Zero SWA Caching	最小	最大	完全不存。利用 SWA 层 t 的 KV 只依赖前一层最近 n_{win} 个 SWA KV 这一性质， L 层模型重算最后 $n_{\text{win}} \cdot L$ 个 token 即可恢复最后 n_{win} 个 SWA KV。

🔑 关键点

1. **EP 通信 ≤ 计算 → 可完全 hide**：V4 把理论加速从 Comet 的 $1.42\times$ 推进到 $1.92\times$ ，实测 RL rollout 达 $1.96\times$ 。
2. **TileLang 三大优势**：host codegen ($< 1\mu s$ 调用开销)、Z3 辅助非线性整数分析、bitwise 复现。
3. **确定性设计面面俱到**：Attention/MoE/mHC 的反传都有针对性无原子加方案。
4. **FP4 无损反量化 FP8** 依赖 scale 比约束，实验证明满足 → 零改动复用现有 FP8 pipeline。

5. **Muon + ZeRO 的融合**：dense knapsack 分桶 + MoE 扁平化切分 + BF16 梯度同步 + two-phase all-to-all + FP32 local sum。
 6. **异构 KV cache** 把 SWA/尾 token 当状态空间模型，与压缩块 KV 分治管理；磁盘层提供三种 SWA 存储策略覆盖不同部署场景。
-

3. 预训练

3.1 数据构建

- **总量**：Flash 32T tokens, Pro 33T tokens。
- **web 数据**：过滤批量自动生成 / 模板化内容，降低 model collapse 风险。
- **数学与代码**：仍是训练核心，**mid-training 阶段引入 agentic 数据**强化 coding 能力。
- **多语料**：构建更大多语种语料，提升 long-tail 跨文化知识。
- **长文档**：V4 特别强调**科学论文、技术报告**等长文档 curation。
- **Tokenizer**：沿用 V3，**词表仍 128K**，新增少量用于上下文构造的 special token；保留 token-splitting 与 FIM 策略。
- **预处理差异**：从不同源 pack 文档以最小化样本截断（受 Ding et al., 2024 启发）；与 V3 不同之处是 **pre-train 阶段使用 sample-level attention masking**。

3.2 模型与训练设置

3.2.1 DeepSeek-V4-Flash 超参表

项	值
Transformer 层数	43
隐层 d	4096
前 2 层	纯 SWA
之后层	CSA / HCA 交错
CSA 压缩率 m	4
CSA attention top-k	512
Indexer 头数 n_h^I	64
Indexer 头维 c^I	128
HCA 压缩率 m'	128
查询头数 n_h	64
头维 c	512
Query 压缩维 d_c	1024
输出投影组数 g	8
每组输出维 d_g	1024
SWA 窗口 n_{win}	128
MoE 层	所有 block（前 3 层 Hash routing）
Shared expert	1
Routed expert	256
Expert 中间维	2048
激活 expert	6
MTP 深度	1
mHC 扩展 n_{hc}	4
Sinkhorn 迭代 t_{max}	20
总参 / 激活	284B / 13B

3.2.2 DeepSeek-V4-Pro 超参表

项	值
Transformer 层数	61
隐层 d	7168
前 2 层	HCA
之后层	CSA / HCA 交错
CSA 压缩率 m	4
CSA attention top-k	1024
Indexer 头数 n_h^I	64
Indexer 头维 c^I	128
HCA 压缩率 m'	128
查询头数 n_h	128
头维 c	512
Query 压缩维 d_c	1536
输出投影组数 g	16
每组输出维 d_g	1024
SWA 窗口 n_{win}	128
MoE 层	所有 block（前 3 层 Hash routing）
Shared expert	1
Routed expert	384
Expert 中间维	3072
激活 expert	6
MTP 深度	1
mHC 扩展 n_{hc}	4
Sinkhorn 迭代 t_{max}	20
总参 / 激活	1.6T / 49B

3.2.3 优化器与学习率调度

Muon 超参（两模型共用）：

- Momentum $\mu = 0.95$
- Weight decay $\lambda = 0.1$
- Update RMS rescale 到 **0.18**（复用 AdamW LR）

AdamW 超参（embedding / head / RMSNorm）：

- $\beta_1 = 0.9, \beta_2 = 0.95, \varepsilon = 10^{-20}$, weight decay 0.1

Batch size 调度：从小批逐步增大并保持峰值。

- Flash 峰值 batch：**75.5M tokens**
- Pro 峰值 batch：**94.4M tokens**

学习率调度：

	Peak LR	End LR	Warmup
V4-Flash	2.7×10^{-4}	2.7×10^{-5}	前 2000 步 线性 warmup
V4-Pro	2.0×10^{-4}	2.0×10^{-5}	前 2000 步 线性 warmup

训练尾部 cosine 衰减到 end LR。

序列长度扩展：**4K → 16K → 64K → 1M**（四段渐进）。

Dense → Sparse Attention 调度：

- 先用 **dense attention warmup 第 1T tokens**（Pro 的 dense 阶段比 Flash 更长）；
- 在序列长度到 **64K** 时引入 sparse attention；
- 引入预设一个**短阶段对 Lightning Indexer 进行 warmup**；
- 随后大部分训练都走 sparse。

负载均衡：bias 更新速度 **0.001**， balance loss 权重 **0.0001**（防止序列内极度不均衡）。

MTP loss 权重：大部分训练为 **0.3**，开始 LR decay 时降到 **0.1**。

3.3 缓解训练不稳定


V4 发现 loss spike 的经验规律：**spike 与 MoE 层的 outlier 强相关**，routing 机制会加剧 outlier。单纯 rollback 无法根本解决。V4 从两个方向入手：**打破 routing 引发的恶性循环** 和 **直接抑制异常值**。

3.3.1 Anticipatory Routing（异步路由）

核心思路：在步 t 用当前参数 θ_t 做特征计算，但路由索引用历史参数 $\theta_{t-\Delta t}$ 计算并应用。即把 backbone 与 router 的更新解耦。

工程实现：为避免两次加载模型参数，在步 $t - \Delta t$ 提前预取步 t 的数据，"anticipatorily" 计算并缓存 t 要用的路由索引。精心编排 pipeline 与 EP 通信重叠，把 Anticipatory Routing 额外 wall-clock 开销压到约 20%。

动态启用：引入自动检测机制，loss spike 发生时触发短 rollback 并切换到 Anticipatory Routing；运行一段时间后回到标准训练。最终额外训练开销可忽略。


 论文未深入展开：为何解耦路由能打破「routing \rightarrow outlier \rightarrow spike」反馈回路的机理； Δt 的最优取值；是否会引入某种路由滞后。论文明确承认 "underlying principles remain insufficiently understood"。

3.3.2 SwiGLU Clamping

做法：

- 线性分量 clamp 到 $[-10, 10]$ ；
- gate 分量上界 cap 到 10（无下界约束）。

Flash 与 Pro 全程启用。实测有效消除 outlier，不损性能。

 论文未深入展开：为何 $[-10, 10]$ 这一具体区间；gate 不设下界的理由；该机制与 RMSNorm 抑制 logits 爆炸的分工。

3.4 基座评测 (Table 1)

评测维度：World Knowledge (12 个 benchmark)、Language & Reasoning (5 个)、Code & Math (6 个)、Long Context (LongBench-V2)。

三模型对比：

Benchmark	V3.2-Base (37B/671B)	V4-Flash-Base (13B/284B)	V4-Pro-Base (49B/1.6T)
MMLU-Pro	65.5	68.3	73.5
MMLU-Redux	87.5	89.4	90.8
AGIEval	80.1	82.6	83.1
C-Eval	90.4	92.1	93.1
MultiLoKo	38.7	42.2	51.1
Simple-QA verified	28.3	30.1	55.2
FACTS Parametric	27.1	33.9	62.6
HumanEval	62.8	69.5	76.8
MATH	60.5	57.4	64.5
LongBench-V2	40.2	44.7	51.5

核心结论：

- 1. **V4-Flash-Base 以更小参数超越 V3.2-Base**：激活参 13B vs 37B，总参 284B vs 671B，但绝大多数 benchmark 更优，尤其在 world knowledge 和长上下文场景。证明架构 + 数据 + 训练优化能弥补甚至超越参数优势。
- 2. **V4-Pro-Base 几乎全面领先**：在 SimpleQA Verified (55.2 vs V3.2 的 28.3)、FACTS Parametric (62.6 vs 27.1) 等知识型 benchmark 上提升巨大；MATH、HumanEval 等推理代码也有显著增幅。
- 3. **BigCodeBench 是唯一明显退步的项**（V3.2 63.9 → Flash 56.8 / Pro 59.2），论文未解释。
- 4. LongBench-V2 上 V4-Pro-Base 51.5 相对 V3.2 的 40.2 是一次明显飞跃。

🔑 关键点

- 1. 序列长度**四段式渐进** + **dense → sparse 两段式**是 V4 预训练的关键调度。
- 2. Muon RMS=0.18 让 AdamW LR 可直接复用，避免为新优化器重调超参。
- 3. Anticipatory Routing 与 SwiGLU Clamping 是 trillion-MoE 稳定训练的实用配方，但机理待进一步研究。
- 4. V4-Flash-Base 以更小参数击败 V3.2-Base 是 V4 架构有效性的最强证据。

4. 后训练

4.1 Pipeline : Specialist Training + On-Policy Distillation

后训练整体沿用 V3.2 pipeline, 但把原来的 mixed RL 阶段整体替换为 On-Policy Distillation (OPD)。流程：

1. **Specialist Training**：为每个领域（math、code、agent、instruction following 等）独立训练一个 expert：
- **SFT on domain-specific high-quality data** 建立基础；

• **GRPO RL** 用 domain reward model 进一步优化；
2. **OPD**：用 multi-teacher OPD 把多个 expert 的能力合并到一个统一 student（详见 §4.2）。

4.1.1 Reasoning Effort 三模式

模式	特征	典型场景	Response 格式
Non-think	基于习惯/简单规则的快速直觉响应	日常琐事、应急响应、低风险决策	<div></think> summary</div>
Think High	有意识的逻辑分析，更慢但更准	复杂问题求解、规划、中风险决策	<div><think> thinking tokens </think>summary</div>
Think Max	推理能力推到极限	探索模型推理边界	1. 系统提示开头注入特殊指令；2. <div><think>thinking tokens </think> summary</div>

每个模式对应不同的 length penalty 和 context window, RL 训练时各自优化。

Think Max 注入的系统提示：

Reasoning Effort: Absolute maximum with no shortcuts permitted.

You MUST be very thorough in your thinking and comprehensively decompose the problem to resolve the root cause, rigorously stress-testing your logic against all potential paths, edge cases, and adversarial scenarios.

Explicitly write out your entire deliberation process, documenting every intermediate step, considered alternative, and rejected hypothesis to ensure absolutely no assumption is left unchecked.

4.1.2 Generative Reward Model (GRM)

对于难以规则化验证的任务, V4 放弃传统 scalar reward model, 改用 rubric-guided GRM：

- **actor 本身即 GRM**：同一模型同时担任生成与评判；

- 对 GRM 直接施加 RL 优化，与 generative 能力联合提升；
- 需要的人工标注极少，模型用自身推理能力评估 policy trajectory。

优势：评分过程包含推理，打分更鲁棒；避免传统 reward hacking。

4.1.3 新 Tool-Call Schema

引入特殊 token `<|DSML|>`，采用 XML 格式的工具调用：

```
## Tools
You have access to a set of tools to help answer the user's question.
You can invoke tools by writing a "<|DSML|tool_calls>" block like the following:

<|DSML|tool_calls>
  <|DSML|invoke name="$TOOL_NAME">
    <|DSML|parameter name="$PARAMETER_NAME" string="true|false">
      $PARAMETER_VALUE
    </|DSML|parameter>
    ...
  </|DSML|invoke>
  ...
</|DSML|tool_calls>
```

String 参数置 `string="true"`；其他类型（number/bool/array/object）按 JSON 格式传且置 `string="false"`。

若启用 thinking mode（由 `<think>` 触发），必须在调用工具或回答前把推理写在 `<think>...</think>` 里。

效果：XML 格式显著降低转义失败与 tool-call 错误，提供更鲁棒的模型-工具交互接口。

4.1.4 Interleaved Thinking 策略

V3.2 的策略：跨 tool-result 轮次保留推理，但用户新消息到来时丢弃所有推理。在复杂 agent 工作流程中每次用户发言都会“冲掉”积累的推理状态。

V4 利用 1M 上下文扩展，细化为两种场景：

- **Tool-Calling 场景**（图 7a）：全程保留所有推理内容，包括跨越用户消息边界。保持长时 agent 任务的累积思维链。
- **General Conversational 场景**（图 7b）：保留 V3.2 原策略，新用户消息到来即丢弃旧推理（在无工具场景下持久推理收益有限）。

注：Terminus 等通过用户消息模拟工具交互的 agent framework 不会触发 tool-calling 路径，V4 继续推荐为这类架构使用 non-think 模型。

4.1.5 Quick Instruction

动机：chatbot 场景中生成回答前常需执行若干**辅助任务**（判断是否触发 web 搜索、意图识别等）。传统做法另起小模型 → **无法复用 KV cache 需冗余 prefill**。

方案：在输入序列后追加一组 special token，每个对应一个辅助任务，**直接复用已计算的 KV cache**。某些任务可**并行执行**。

效果：显著降低用户侧 TTFT，消除维护额外小模型的工程负担。

支持的 Quick Instruction tokens：

Special Token	用途	格式
< action >	判断用户 prompt 是否需要 web 搜索还是可直接回答	...< User >{prompt}< Assistant ><think>< action >
< title >	首次助手响应后生成简洁对话标题	...< Assistant >{response}< end_of_sentence >< title >
< query >	为 prompt 生成搜索 query	...< User >{prompt}< query >
< authority >	分类 prompt 对来源权威性的需求	...< User >{prompt}< authority >
< domain >	识别 prompt 的领域	...< User >{prompt}< domain >
< extracted_url >/(< read_url >)	判断 prompt 中各 URL 是否需 fetch 读取	...< User >{prompt}< extracted_url >{url}< read_url >

4.2 On-Policy Distillation

目标：把多个 domain expert 的能力合并到统一 student。给定 N 个专家 $\{\pi_{E_1}, \dots, \pi_{E_N}\}$ ：

$$\mathcal{L}_{\text{OPD}}(\theta) = \sum_{i=1}^N w_i \cdot D_{\text{KL}}(\pi_{\theta} \parallel \pi_{E_i}) \tag{29}$$

- w_i 为 expert 权重（按相对重要性）；
- 反向 KL $D_{\text{KL}}(\pi_{\theta} \parallel \pi_{E_i})$ 的计算**依赖从 student 采样 trajectory**（保持 on-policy）；
- 反向 KL 的 mode-seeking 性质确保 student 针对当前任务**选择性地**学习最相关的专家（math 任务跟 math expert，code 任务跟 code expert）；
- 10+ 个 teacher**，覆盖多个领域。

为何用 full-vocabulary logit distillation 而非 per-token KL 估计：

过往工作常把 full-vocab KL 简化为 per-token KL 估计，然后复用 RL 框架，把

$$\text{sg} \left(\log \frac{\pi_{E_i}(y_t \mid x, y_{<t})}{\pi_{\theta}(y_t \mid x, y_{<t})} \right)$$

当作 per-token advantage。虽省资源，但**梯度估计方差高、易训练不稳**。

V4 采用 **full-vocabulary logit distillation**，保留完整 logit 分布计算反向 KL，梯度估计更稳、更忠实于教师知识。工程上这需要大量优化（见 §4.3.2）才可行。

4.3 RL / OPD 基础设施

4.3.1 FP4 量化集成

Rollout 和所有 inference-only forward（含 teacher 与 reference model）**直接用 native FP4 权重**，降低 memory traffic 与采样延迟。Training step 通过**无损 FP4→FP8 反量化**（§2.4）无缝复用既有 FP8 mixed-precision 框架，反向 pipeline 零改动。

4.3.2 Full-Vocab OPD 的 Teacher 调度

支持**数量无上限的 teacher**，每个可能达万亿参数。关键技术：

- **集中式分布式存储 + on-demand 加载**：所有 teacher 权重 offload 到集中存储，forward 时按需加载；**ZeRO-like 参数分片**缓解 I/O 与 DRAM 压力。
- **只缓存 last-layer teacher hidden state**，不 materialize 完整 logits：词表 $|V| > 100k$ 时 materialize logits 不可行（即使 spool 到磁盘）。训练时从缓存取出 hidden state，经对应 prediction head **on-the-fly 重构 full logits**，重计算开销可忽略，**完全避免 logits 显存爆炸**。
- **Teacher 按 index 排序样本分发**：数据 dispatching 时按 teacher index 排序 training sample，保证每个 mini-batch 内**每个 teacher head 仅加载一次**，任意时刻 device 只驻留一个 teacher head。
- **异步加载/卸载**：参数和 hidden state 加载/卸载在后台异步进行，不阻塞 critical path。
- **专用 TileLang KL kernel**：精确计算 teacher-student logits 的 KL 散度，加速计算同时**限制动态显存分配**。

4.3.3 可抢占、容错的 Rollout 服务

背景：集群用 cluster-wide 抢占式调度器，任何任务随时可能被抢占；大规模 GPU 集群硬件故障频发。

Token 粒度 Write-Ahead Log (WAL)：每生成一个 token 立即 append 到该 request 的 WAL。抢占时暂停推理引擎，保存未完成 request 的 KV cache。恢复时用持久化 WAL + 保存的 KV cache 继续 decode；若硬件故障可用 WAL 里的 token 重跑 prefill 重建 KV cache。

关键正确性考量：**不能从头重生成未完成 request**——这会引入**长度偏差**（短响应更可能在中断中幸存，重跑后模型产出短序列的倾向会系统性增强）。即使 inference stack 是 batch-invariant + deterministic 可用一致 seed 重跑，仍有 decode 重复开销，**不如 token 级 WAL**。

4.3.4 百万 token 上下文的 RL Framework

针对 1M 序列：

- **Rollout 阶段**：用 §4.3.3 的 WAL 服务。
- **Inference/Training 阶段**：rollout 数据分解为**轻量元数据**与**重量 per-token 字段**。数据 dispatching 只加载元数据做全局 shuffle 和 packing 布局计算；per-token 字段通过**共享内存 dataloader**加载以消除 node 内冗余，**mini-batch 粒度用完即释放**，大幅缓解 CPU 和 GPU 显存压力。
- **On-device mini-batch 数量动态决定**：根据工作负载在 compute throughput 与 I/O overlap 间高效 trade-off。

4.3.5 DSec：Agent 沙箱基础设施

DeepSeek Elastic Compute (DSec) 是生产级沙箱平台，由三个 Rust 组件组成：

- **Apiserver**：API gateway；
- **Edge**：per-host agent；
- **Watcher**：cluster monitor。

通过自定义 RPC 协议互联，基于 **3FS** 分布式文件系统**水平扩展**。单集群支持**数十万并发沙箱实例**。

4 条设计动机：

1. Agentic 负载高度异构，从轻量 function call 到完整软件工程 pipeline；
2. 环境镜像多且大，需快速加载与迭代；
3. 高密度部署要求高 CPU/内存利用率；
4. Sandbox 生命周期需与 GPU 训练调度协调（抢占 + checkpoint 恢复）。

四种执行底层（统一 Python SDK **libdsec**）：

底层	特点	实现
Function Call	无状态调用，消除冷启动	预热 container 池
Container	Docker 兼容	EROFS on-demand 加载高效镜像组装
microVM	VM 级隔离，安全敏感、高密度	Firecracker
fullVM	支持任意 guest OS	QEMU

所有四者共用 API：命令执行、文件传输、TTY 访问；切换只需参数变更。

分层镜像存储：

- **Container** : base image 和 filesystem commit 作为 3FS-backed 只读 EROFS 层直接 mount 到 overlay lowerdir ; 文件元数据本地可用, data block 按需从 3FS fetch ;
- **microVM** : 用 **overlaybd** 格式, 只读 base layer 在 3FS 跨实例共享, 写入走本地 COW 层 ; snapshot 可 chain, 支持版本化与**毫秒级恢复**。

高密度并发下的两类瓶颈优化 :

1. 缓解虚拟化环境中重复 page cache 占用, 应用 memory reclamation 实现安全 overcommit ;
2. 缓解 container runtime 的 spinlock 竞争, 降低 per-sandbox CPU 开销。

轨迹日志与抢占安全恢复 : 每个 sandbox 维护全局有序轨迹日志, 持久记录每条命令及其结果。三大用途 :

1. **Client fast-forwarding** : 训练抢占时 sandbox 资源保留 ; 恢复时**重放已完成命令的缓存结果**, 加速恢复的同时避免非幂等操作重复执行 ;
2. **Fine-grained provenance** : 每次状态变更来源可追溯 ;
3. **Deterministic replay** : 历史会话可从 trajectory 完整重现。

4.4 标准评测 (Table 6、7)

4.4.1 评测配置

- **Reasoning / Knowledge** : temperature=1.0, context window 8K / 128K / 384K 对应 Non-think / High / Max。
- **Math** : 用特定 prompt 模板 ; Pro-Max 用 "证明 + 答案" 加强模板引出更深推理。
- **Formal Math** : Lean v4.28.0-rc1 agentic setting, 访问 Lean compiler 与语义 tactic search, 最多 500 tool calls ; 另一 compute-intensive pipeline 先用 self-verification 筛选自然语言候选解, 再由 formal agent 给出 Lean 证明。**严格 verifier Comparator 同时接受两种 setting 才判对。**
- **Agent (code)** : 内部评测框架, 最小工具集 (bash + file-edit), max 500 步, 最大上下文 512K。
- **Agent (search)** : 内部 harness + websearch + Python tool, max 500 步, 512K 上下文。BrowseComp 用 V3.2 的 discard-all context management。
- **1M 长上下文** : MRCR + CorpusQA。统一 Claude Opus 4.6 和 Gemini 3.1 Pro 的评测配置 ; GPT-5.4 因 API 响应不稳未评。

4.4.2 核心分数 (V4-Pro-Max vs 主要对手)

Benchmark	Opus-4.6- Max	GPT-5.4- xHigh	Gemini-3.1- Pro-High	K2.6- Thinking	GLM- 5.1	V4-Pro- Max
MMLU-Pro	89.1	87.5	91.0	87.1	86.0	87.5
SimpleQA-Verified	46.2	45.3	75.6	36.9	38.1	57.9
Chinese- SimpleQA	76.4	76.8	85.9	75.9	75.0	84.4
GPQA Diamond	91.3	93.0	94.3	90.5	86.2	90.1
HLE	40.0	39.8	44.4	36.4	34.7	37.7
LiveCodeBench	88.8	—	91.7	89.6	—	93.5
Codeforces Rating	—	3168	3052	—	—	3206
HMMT 2026 Feb	96.2	97.7	94.7	92.7	89.4	95.2
IMOAnswerBench	75.3	91.4	81.0	86.0	83.8	89.8
Apex	34.5	54.1	60.9	24.0	11.5	38.3
Apex Shortlist	85.9	78.1	89.1	75.5	72.4	90.2
LongMRCR 1M	92.9	—	76.3	—	—	83.5
CorpusQA 1M	71.7	—	53.8	—	—	62.0
Terminal Bench 2.0	65.4	75.1	68.5	66.7	63.5	67.9
SWE Verified	80.8	—	80.6	80.2	—	80.6
SWE Pro	57.3	57.7	54.2	58.6	58.4	55.4
SWE Multilingual	77.5	—	—	76.7	73.3	76.2
BrowseComp	83.7	82.7	85.9	83.2	79.3	83.4
HLE w/ tools	53.1	52.0	51.6	54.0	50.4	48.2
GDPval-AA (Elo)	1619	1674	1314	1482	1535	1554
MCPAtlas Public	73.8	67.2	69.2	66.6	71.8	73.6
Toolathlon	47.2	54.6	48.8	50.0	40.7	51.8

4.4.3 核心结论

Knowledge :

- V4-Pro-Max 在 SimpleQA-Verified 领先所有开源模型约 **20 个百分点**；
- Chinese-SimpleQA **84.4** 显著超越所有开源对手；
- 但仍落后 Gemini-3.1-Pro (75.6 / 85.9)，在 MMLU-Pro / GPQA / HLE 上仅略微领先 K2.6 与 GLM。
- V4-Flash 知识评测明显落后 V4-Pro (参数规模所致)。

Reasoning :

- V4-Pro-Max 在开源模型中**全面领先**，多项指标匹敌闭源 SOTA。
- Coding 竞赛尤强：Codeforces Rating **3206** 为首次开源模型追平闭源 (胜过 GPT-5.4 的 3168)，在人类候选者中 rank **23rd**。
- V4-Flash-Max 在 code / math reasoning 上超越此前最佳开源模型 K2.6-Thinking。
- Apex Shortlist **90.2** 领先一众对手。
- **Formal Math** (图 8) :
 - Practical Regime (Putnam-200 Pass@8, minimal tools, bounded sampling) : V4-Flash-Max **81.00**, 远超 Seed-1.5-Prover / Gemini-3-Pro 的 26.50 与 Seed-2.0-Pro 的 35.50 ；
 - Frontier Regime (Putnam-2025, 混合 formal-informal + 大 compute) : V4 **120/120** proof-perfect, 匹敌 Axiom、超越 Aristotle 100/120、Seed-1.5-Prover 110/120。

Agent : 与 K2.6、GLM-5.1 对标，总体略逊于闭源 frontier。V4-Pro 在 MCPAtlas 与 Toolathlon (覆盖广泛工具/MCP 服务) 表现良好，说明**模型非仅在内部框架上 overfit**。V4-Flash 在 agent 任务上明显弱于 V4-Pro (尤其 Terminal Bench 2.0)。Terminal-Bench 2.0 Verified 子集上 V4-Pro 约 **72.0**。

1M 长上下文 (图 9, MRCCR 8-needle) :

Input Tokens	V4-Pro-Max	V4-Flash-Max
8K	0.90	0.91
16K	0.85	0.84
32K	0.94	0.87
64K	0.90	0.85
128K	0.92	0.87
256K	0.82	0.76
512K	0.66	0.60
1024K	0.59	0.49

- 在 128K 内几乎无退化；
- 超过 128K 后出现可见下降，但 1M 下 V4-Pro-Max 仍保 0.59 MMR，**超过 Gemini-3.1-Pro**（但落后 Claude Opus 4.6）；
- 在更接近真实场景的 **CorpusQA 1M** 上，V4-Pro **62.0** 超过 Gemini-3.1-Pro 的 53.8。

Reasoning Effort（Table 7）：Max 模式（更长 context、更弱 length penalty）在**最难**任务上显著优于 High 模式。Figure 10 显示：

- HLE 上 V4-Pro 的 token efficiency 高于 V3.2；
- Terminal Bench 2.0 上 V4 系列相对 V3.2 性能大幅跃升。

Flash vs Pro 的 Non-Think / High / Max 三模式消融：

	Non-Think	High	Max
V4-Flash — MMLU-Pro	83.0	86.4	86.2
V4-Flash — HLE	8.1	29.4	34.8
V4-Flash — HMMT	40.8	91.9	94.8
V4-Flash — Terminal Bench	49.1	56.6	56.9
V4-Pro — MMLU-Pro	82.9	87.1	87.5
V4-Pro — HLE	7.7	34.5	37.7
V4-Pro — HMMT	31.7	94.0	95.2
V4-Pro — Terminal Bench	59.1	63.3	67.9

Non-think 模式在难任务（HMMT、HLE）下表现极差，凸显推理链条的必要性。

4.5 真实世界任务

4.5.1 中文写作

功能性写作（V4-Pro vs Gemini-3.1-Pro, Table 12 汇总）：

- 总胜率 **62.7% vs 34.1%**（tie 3.25%）；
- 各大类均占优（Business 65.16%、Media 57.96%、Everyday 69.49%、Oral 58.62%、Official 54.78%、Academic 63.43%）；
- 差距根源：Gemini 偶尔让其固有风格偏好压过用户显式要求。

创意写作（Table 13）：

- 指令遵循胜率 **60.03%**（vs Gemini-3.1-Pro 39.44%）；
- 写作质量胜率 **77.48%**（vs 22.35%）——显著提升；
- 覆盖 fiction / fan fiction / prose / poetry / lyrics 等 18 个子类。

高难度/多轮创意写作（Table 14, V4-Pro vs Claude Opus 4.5）：

- 复杂指令遵循：DS 46.9% vs Opus 53.1%；
- 多轮写作：DS 45.6% vs Opus 51.7%；
- 总体：DS 45.9% vs Opus 52.0% —— Opus 4.5 在高难度/多轮场景仍占优。

4.5.2 Search

RAG vs Agentic Search（Table 9）：

难度	类别	#	Agent Win	RAG Win	Tie	Agent%	RAG%	Tie%
Easy	客观问答	196	110	43	43	56.1	21.9	21.9
Easy	主观问答	321	198	56	67	61.7	17.4	20.9
Hard	客观问答	168	102	33	33	60.7	19.6	19.6
Hard	主观问答	184	126	27	31	68.5	14.7	16.8
Total		869	536	159	174	61.7	18.3	20.0

Agentic Search 全面优于 RAG，且难度越高优势越大。

成本对比（Table 10）：

模式	Tool Calls	Prefill tokens	Output tokens
V4 Agentic Search	16.2	13649	1526
V4 RAG	—	10453	1308

Agentic 仅略贵，大多数 tool call 并行。

V4-Pro vs V3.2 RAG Q&A（Table 11）：

- 客观问答：V4 27.3% / V3.2 8.7% / Tie 64.0%；
- 主观问答：V4 28.5% / V3.2 11.1% / Tie 60.4%；
- 最大提升在 **Single-value Search** 和 **Planning & Strategy**；
- 在 Comparison、Recommendation 等需平衡多视角推理的场景 V3.2 仍相对有竞争力。

4.5.3 白领任务

30 个中文高级专业任务，覆盖 13 个行业（金融、教育、法律、科技等），包含信息分析、文档生成、文档编辑，在内部 agent harness（含 bash + web search）中评估。

V4-Pro-Max vs Opus-4.6-Max：

维度	Win Rate	V4-Pro-Max	Opus-4.6-Max
Analysis	V4 55% / Tie 8% / Opus 37%	98.32	96.68
Generation	V4 52% / Tie 10% / Opus 38%	—	—
Editing	V4 47% / Tie 18% / Opus 35%	—	—
Overall 不败率	63%	—	—
Task Completion	—	98.32	96.68
Instruction Following	—	87.76	88.88
Content Quality	—	83.32	78.00
Formatting Aesthetics	—	76.68	72.68
Overall Score	—	86.52	84.06

- 定性：
- V4 优势：**主动推断隐含意图**（补充 insights + 自验证步骤）、**长文连贯生成**（vs Opus 倾向过简 bullet list）、严格遵循中文专业文档规范（如标准化分级编号）；
 - V4 弱项：偶尔忽略具体格式约束（Instruction Following 略逊 Opus）、摘要长文能力较弱、PPT 视觉设计仍有改进空间。

4.5.4 Code Agent

从 **50+ 内部工程师** 收集 ~200 个挑战任务，覆盖 feature development / bug fixing / refactoring / diagnostics，跨 PyTorch / CUDA / Rust / C++ 多栈；严格筛选后保留 **30 任务**。

R&D Coding Benchmark（Table 8）：

模型	Pass Rate
Haiku 4.5	13%
Sonnet 4.5	47%
V4-Pro-Max	67%
Opus 4.5	70%
Opus 4.5 Thinking	73%
Opus 4.6 Thinking	80%

开发者调研 ($N = 85$, 均已用 V4-Pro 做 agentic coding) : "V4-Pro 是否可作为默认主力 coding 模型" :

- 52% yes、39% 倾向 yes、<9% no ;
- 反馈亮点 : 大多数任务满意 ;
- 痛点 : 偶尔低级错误、模糊 prompt 误解、偶尔 over-thinking。

🔑 关键点

1. Specialist Training + OPD 替代 mixed RL 是 V4 后训练的方法论核心 : 先分专精, 再用 full-vocab reverse KL 在 student 自采 trajectory 上融合, 避免了 mixed RL 的相互干扰。
 2. 三种 Reasoning Effort 模式给了推理预算 vs 响应延迟的显式 knob, Think Max 通过系统提示进一步释放深度推理。
 3. GRM = actor 本身, 让"生成能力与评判能力"联合优化, 小样本标注即可。
 4. 新 `[DSML]` + XML tool-call schema 降低转义/调用错误 ; Quick Instruction 用 KV cache 复用省下 TTFT。
 5. Interleaved Thinking 区分 tool vs 对话场景 : tool 场景 1M 上下文全程保留推理, 显著增益 agent 任务。
 6. OPD 基础设施关键 : centralized teacher hidden state 缓存 + on-the-fly logits 重构 + per-teacher-index 排序加载 + 专用 TileLang KL kernel。
 7. Token 级 WAL 是应对抢占/故障时避免长度偏差的正确方案, 远优于从头重生成。
 8. DSec 四种执行底层 (Function Call / Container / microVM / fullVM) 统一 SDK, 覆盖 agent 任务从轻到重的完整光谱。
-

5. 结论、局限与未来方向

5.1 V4 的贡献总结

- Hybrid Attention (CSA + HCA) 在 1M 上下文下相对 V3.2 单 token FLOPs 降至 27% / KV cache 降至 10% (Flash 进一步到 10% / 7%), native 支持百万 token。
- V4-Pro-Max 在开源模型中全面刷新 SOTA : 知识领先、推理逼近 frontier 闭源、agent 能力竞争力强。
- V4-Flash-Max 以高性价比达到前代开源最佳水平的推理性能。
- 为 test-time scaling、长 horizon agentic 任务、online learning 等下一代范式打下必要基础。

5.2 已知局限

1. 架构复杂度 : 为快速落地保留了不少"已验证有效"的小部件, 架构不够精简优雅。未来将做更系统、更原理导向的研究, 把架构蒸馏到最本质设计, 不牺牲性能前提下简化。

- 2. **Anticipatory Routing 与 SwiGLU Clamping 的机理未充分理解**：两者实证有效，但底层原理尚不清楚。未来将在训练稳定性的基础问题上发力，强化内部 metric 监控，追求**更原理性、更可预测的稳定大规模训练**。
- 3. **知识类评测仍落后 Gemini-3.1-Pro**（如 SimpleQA-Verified 57.9 vs 75.6）；高难度中文创意写作与多轮场景仍被 Claude Opus 4.5 略占上风。
- 4. **Agent 能力**在 SWE Pro、HLE w/ tools 等任务上仍被闭源 frontier 模型领先。
- 5. **Flash 与 Pro 在知识任务差距显著**，小规模模型在知识保留上仍受参数限制。

5.3 未来方向

- 1. **更稀疏的 embedding 模块**（Cheng et al., 2026, Conditional Memory via Scalable Lookup），在 MoE + 稀疏注意力之外开辟新的稀疏化维度。
- 2. **多模态能力集成**。
- 3. **低延迟架构与系统技术**，让长上下文部署与交互更响应式。
- 4. **长时 / 多轮 agentic 任务**持续迭代。
- 5. **更好的数据 curation 与合成策略**，提升智能、鲁棒性、在更广场景下的实用性。
- 6. **向 online learning 演进**：百万 token 上下文为此奠定了基础。

附录：核心超参与数字速查

项	V4-Flash	V4-Pro
总参 / 激活	284B / 13B	1.6T / 49B
层数 / d	43 / 4096	61 / 7168
$n_h / c / d_c$	64 / 512 / 1024	128 / 512 / 1536
g / d_g	8 / 1024	16 / 1024
$m / m' / \text{attn top-k}$	4 / 128 / 512	4 / 128 / 1024
n_h^I / c^I	64 / 128	64 / 128
n_{win}	128	128
Shared / Routed / Activated experts	1 / 256 / 6	1 / 384 / 6
Expert 中间维	2048	3072
$n_{\text{hc}} / \text{Sinkhorn } t_{\text{max}}$	4 / 20	4 / 20
MTP 深度	1	1

项	V4-Flash	V4-Pro
训练 tokens	32T	33T
Peak / End LR	$2.7 \times 10^{-4} / 2.7 \times 10^{-5}$	$2.0 \times 10^{-4} / 2.0 \times 10^{-5}$
Peak batch	75.5M	94.4M
Warmup 步	2000	2000
Muon momentum / WD / RMS	0.95 / 0.1 / 0.18	0.95 / 0.1 / 0.18
AdamW $\beta_1 / \beta_2 / \varepsilon$ / WD	0.9 / 0.95 / 10^{-20} / 0.1	0.9 / 0.95 / 10^{-20} / 0.1
Bias 更新速度	0.001	0.001
Balance loss 权重	0.0001	0.0001
MTP loss 权重	0.3 \rightarrow 0.1	0.3 \rightarrow 0.1
Dense warmup tokens	1T (Flash) ; Pro 更长	—
Sparse 引入长度	64K	64K
序列长度扩展	4K \rightarrow 16K \rightarrow 64K \rightarrow 1M	4K \rightarrow 16K \rightarrow 64K \rightarrow 1M

SwiGLU Clamping : linear $\in [-10, 10]$, gate 上界 10。 **Anticipatory Routing** : overhead ~20%, 异常触发自动启用。 **Newton-Schulz 10 步** : 前 8 步 (3.4445, -4.7750, 2.0315), 后 2 步 (2, -1.5, 0.5)。 **MegaMoE 理论加速** : $1.92\times$ (V4-Flash 配置下) ; 实测 $1.50 \sim 1.73\times$ 通用 / $1.96\times$ RL rollout。 **FP4/FP8 C/B 比** : V4-Pro 每 GB/s 带宽可 hide 6.1 TFLOP/s ($2d = 6144$ FLOPs/Byte)。 **mHC overhead** : 6.7% of overlapped 1F1B。 **CPU 校验开销** (TileLang Host Codegen) : $< 1\mu s$ /调用。 **Index scores FP32 \rightarrow BF16** : top-k 选择器 $2\times$ 加速, 99.7% recall。