Chronicals：一种高性能大语言模型微调框架，速度比 Unsloth 快 3.51 倍

Arjun S. Nair[1]

[1]独立研究者，ORCID：0009-0004-8903-0974，5minutepodcastforyou@gmail.com

arXiv:2601.02609v1 [cs.LG] 6 Jan 2026

微调一个 70 亿参数的语言模型需要 84GB 内存：权重占 14GB，梯度占 14GB，FP32 优化器状态占 56GB。这一需求是 A100-40GB 容量的两倍。我们推出了 Chronicals 训练框架，通过四项彼此正交的优化来减少内存占用：融合 Triton 内核消除了 75% 的内存流量；Cut Cross-Entropy 将 logit 内存从 5GB 降至 135MB；LoRA+ 采用差异化学习率，实现快 2 倍的收敛速度；序列打包则挽回了因填充而浪费的 60%–75% 计算资源。

在搭载 A100-40GB 的 Qwen2.5-0.5B 上，Chronicals 全量微调的速度达到每秒 41,184 个令牌，比经验证的 Unsloth 每秒 11,736 个令牌快 3.51 倍。在秩为 32 的 LoRA 训练中，我们达到每秒 11,699 个令牌，而 Unsloth MAX 为每秒 2,857 个令牌（提升 4.10 倍）。基准测试期间，我们发现 Unsloth 报告的每秒 46,000 个令牌这一数据对应的梯度范数为零，表明模型实际上并未进行训练。

本文为每项优化提供了完整的数学基础。我们推导出在线 softmax 算法，使词汇表大小为 151,936 时内存减少 37 倍；证明了 FlashAttention 的 IO 复杂度界；阐明了 LoRA+ 中 A 矩阵和 B 矩阵之间 16 倍学习率比的理论依据（ICML 2024）；并介绍了融合内核，相较于朴素实现，分别实现了 7 倍（RMSNorm）、5 倍（SwiGLU）和 2.3 倍（QK-RoPE）的加速比。所有算法均附有伪代码、正确性证明，以及用于量化各项贡献的消融研究。

大语言模型 — 微调 — FlashAttention — LoRA — Triton 内核 —训练优化 — CutCross-Entropy—FP8 训练

引言

考虑在指令遵循示例数据集上训练通义千问Qwen2.5-0.5B——一个规模适中的 4.94 亿参数模型。仅词汇表就包含 151,936 个词元。计算交叉熵损失需要实例化一个形状为 [批次 × 序列 × 词汇表] 的 logit 张量——对于批次大小为 8、序列长度为 1024 的情况，这个单独的张量就会占用 4.97 GB。再加上梯度后，用于损失计算的内存将接近 10 GB，而该模型的权重仅占用 1 GB。

这种内存激增并非损失计算独有的问题。注意力分数会随序列长度呈二次方增长。AdamW 的优化器状态每个参数占用 8 字节（FP32 格式的一阶矩和二阶矩）。一个训练步骤涉及数百次独立的 CUDA 内核启动，每次都会产生 5–10 微秒的开销。可变长度序列填充到统一的最大长度后，会将 60–75% 的计算浪费在对梯度毫无贡献的词元上。

这些低效问题会不断叠加。尝试在单张 A100-40GB 上微调 LLaMA-7B 的从业者会发现，训练无法启动——84GB 的内存需求（14GB 权重 + 14GB 梯度 + 56GB 优化器状态）超过可用显存 2 倍以上。标准做法是租用更大的硬件、接受更慢的训练速度，或彻底放弃尝试。

我们认为这没有必要。每个瓶颈都有原则性解决方案。融合内核通过在单次遍历中计算多个操作，消除启动开销并减少内存流量。分块算法无需将大型张量完整物化即可进行处理。差分学习率通过兼顾不同参数组各自的作用来加速收敛。序列打包通过拼接较短的样本，挽回浪费的计算资源。

挑战在于如何将这些优化整合为一个协调一致的系统。单独来看，每种技术都能带来 2-3 倍的提升。正确组合后，提升可达 10 倍甚至更高。Chronicals 是我们对这种整合的尝试。

具体来看内存瓶颈。为了了解训练期间内存的去向，我们追踪一个包含 10 亿个参数、24 层、隐藏维度为 2048 且有 16 个注意力头的 Transformer 的单次前向-反向传播。我们假设批次大小为 4，序列长度为 4096。

模型权重以 BF16 格式占用 2 GB（10 亿个参数× 2 字节数）。**梯度**以相同精度还需要 2 GB。**优化器状态**中，AdamW 存储一阶矩和二阶矩，以 FP32 格式总计占用 8 GB（10 亿个 × 2 状态 × 4字节数）。

激活值 带来了第一个挑战。每个 Transformer 层都会生成形状为 $[4, 4096, 2048]$, 的隐藏状态，每层占用 134 MB，24 层合计 3.2 GB。如果不使用梯度检查点，这些隐藏状态就必须保留到反向传播阶段。

注意力分数 构成了二次瓶颈。每个注意力头都会计算一个 4096×4096 分数矩阵。16 个注意力头处理 4 个序列时：$4 \times 16 \times 4096^2 \times 4$ 字节数，即 $= 4.3$ GB。标准注意力机制会同时存储分数和 Softmax 输出，使内存占用翻倍至 8.6 GB。

Logits 会随词汇表大小增长。对于通义千问的 151,936 词元词汇表： $4 \times 4096 \times 151936 \times 4$ 字节数 $= 9.9$ GB。

存储梯度会使其翻倍。
总量已超过 40 GB，尚未计入临时缓冲区、CUDA 工作空间分配和内存碎片开销。这解释了为什么朴素实现会在理论上足以胜任的硬件上失败。

计算瓶颈：为什么 GPU 会闲置。内存消耗只说明了问题的一半。只有当计算量显著超过内存访问量时，现代 GPU 才能达到理论浮点运算次数。A100 的峰值性能为 312 万亿次浮点运算（BF16），这要求每从全局内存传输一个字节，就执行 156 次算术运算——这就是算术强度阈值。低于此阈值的操作属于内存受限，受限于 2 TB/s 的 HBM 带宽，而非计算能力。

交叉熵损失很好地体现了这个问题。对于每个元素，我们在内存中传入和传出 4 个字节的同时，只执行少量浮点运算（求指数、除法、减法）。算术强度约为 1 浮点运算/字节，比阈值低两个数量级。GPU 大部分时间都在等待数据。

小型操作会让情况更加糟糕。每次 CUDA 内核启动都需要中央处理器与图形处理器通信，无论内核的工作负载如何，这一过程都需要 5-10 微秒。使用朴素 PyTorch 实现的 Transformer 层会执行数十个独立操作：线性投影、注意力分数计算、Softmax、注意力输出、残差连接、层归一化、前馈网络。每层启动 50 个内核，24 层共计 1,200 次启动——仅开销就会耗费 6-12 毫秒。

最后，可变长度序列还会带来隐性成本。批处理需要将较短的序列填充到与最长序列相同的长度。如果序列长度符合典型分布（短序列多，长序列少），那么填充位置中有 60-75% 不会产生梯度，却会消耗完整的计算和内存。

先前工作及其局限。有几种框架针对这些挑战中的部分问题提供了解决方案。了解各自的贡献及其不足之处，有助于指导 Chronicals 的设计。

FlashAttention (1–3) 是过去三年中影响最深远的优化之一。它将注意力计算分解为适合静态随机存取存储器的分块，并使用在线 softmax 算法累积结果，无需实例化完整的$N \times N$分数矩阵，从而将 FlashAttention 的注意力内存占用从 $O(N^2)$降至 $O(N)$。对于长度为 4096 的序列，这意味着内存占用从 4.3 GB 降至几 MB。FlashAttention-3 通过线程束专门化将这一技术扩展到 H100，实现了 740 万亿次浮点运算（利用率为 75%）。我们将 FlashAttention 集成到 Chronicals 中，作为注意力计算的核心。

Liger Kernel (13) 将融合原理应用于其他 Transformer 操作。其针对 RMSNorm、SwiGLU 和交叉熵的融合 Triton 内核减少了内存分配和内核启动开销。基准测试显示，内存用量减少了 3 倍，吞吐量提升了 20%。Chronicals 沿用了 Liger 的方法，并将其扩展到其他操作（融合 QK-RoPE、融合 LoRA 线性层），并与互补优化相结合。

LoRA (4) 通过将权重更新限制为低秩分解，绕开了微调中的内存问题：$\Delta W = BA$ 其中 $B \in \mathbb{R}^{d\times r}$ 且 $A \in \mathbb{R}^{r\times k}$ 满足$r \ll \min(d,k)$。对于应用于 4096×4096权重矩阵的秩 16 分解，可训练参数从 1680 万降至 131,072，减少了 128 倍。只有较小的 LoRA 矩阵需要计算梯度并保存优化器状态。

Cut Cross-Entropy (12)，由苹果公司的研究人员提出，无需生成完整的 logit 张量即可计算交叉熵。通过将词汇表分成多个分块，并使用在线 Softmax 累积对数和指数，内存占用从 $O(BNV)$降至 $O(BNC)$，其中 C表示分块大小。对于拥有 151,936 个词元的通义千问词汇表，且 $C = 4096$，这意味着内存占用减少了 37 倍。

Unsloth (14) 结合多种技术与自定义 CUDA 内核，声称比标准实现快 2 倍。该框架因易于使用而广受欢迎。然而，我们的基准测试发现了一个关键问题：在某些配置下，Unsloth 报告的吞吐量达到每秒 46,000 个词元时，梯度范数恰好为零——模型实际上并未训练。当我们确保梯度流正常后，吞吐量降至每秒 11,736 个词元。我们将在第 节详细说明这一发现。

集成方面的差距。上述每项优化单独使用时都能带来显著改进。挑战——也是我们的贡献——在于将它们结合起来。简单地组合这些技术往往行不通：融合内核可能与 torch.compile 冲突，量化可能导致收敛不稳定，而序列打包则需要自定义注意力掩码。实践者往往需要花费数天进行工程开发，才能让这些技术协同工作。

此外，现有框架忽略了只有通过整体分析才能发现的优化机会。LoRA+论文（5）证明，标准 LoRA 对 A 矩阵和 B 矩阵采用相同学习率并非最优——为了正确收敛，B 矩阵所需的学习率要高出 16 倍。Unsloth 和 Liger 均未实现这一发现。

我们的贡献。本文介绍 Chronicals，这是一个用于高效大语言模型微调的集成框架。我们的贡献涵盖实用系统，以及各项优化背后的数学基础。

1. 完整的集成训练系统。 Chronicals 将 FlashAttention、融合 Triton 内核、LoRA+优化、Cut Cross-Entropy 和序列打包整合到一个统一的框架中。在配备 A100-40GB 的 Qwen2.5-0.5B 上：

- 全量微调的速度达到每秒 41,184 个词元，比 Unsloth 经验证的每秒 11,736 个词元快 3.51 倍

- 秩为 32 的 LoRA 训练达到每秒 11,699 个词元，比 Unsloth MAX 的每秒 2,857 个词元快 4.10 倍

•内存效率达到每 MB 每秒 3.34 个词元，高于 Unsloth 的每 MB 每秒 2.11 个词元

2. 每项优化的数学基础。我们从基本原理出发进行推导：

• Cut Cross-Entropy 所依据的在线 softmax 算法，并证明其正确性、分析其数值稳定性（第 3 节）

•FlashAttention 的 IO 复杂度界表明，当静态随机存取存储器大小为 M 时，需要进行 $O(N^2d^2M^{-1})$ 次内存访问（第 6 节）

• LoRA+ 学习率比 $\eta_B = 16\eta_A$，根据初始化时的梯度幅值分析推导得出（第 5 节）

•序列打包的最佳适应递减算法近似界：最多需要 $11/9 \cdot \mathrm{OPT} + 6/9$ 个箱（第 7 节）

3. 创新的内核实现。我们贡献了现有框架中尚未实现的融合 Triton 内核：

•融合 *QK-RoPE*：在单个内核中对查询和键应用旋转位置嵌入，相比单独操作实现 2.3 倍加速

• 融合 *LoRA* 线性层：计算 $Wx + BAx$，无需物化中间结果

• 零同步梯度裁剪：裁剪梯度时无需 GPU-CPU 同步，从而消除每个训练步骤中长达 50-100μ 秒的瓶颈

4. 发现 Unsloth 中的基准测试漏洞。我们的调查发现，Unsloth 报告的每秒 46,000 个 token 吞吐量对应的梯度范数恰好为零，这意味着模型并未进行训练。这一发现凸显了在训练基准测试中验证梯度流的重要性。我们在第 节中记录了该漏洞及验证方法。该漏洞出现在 Unsloth 的“快速”模式禁用了某些层的梯度计算时，导致吞吐量数据虚高，无法反映实际训练性能。

5. 开源发布。我们以开源许可证发布 Chronicals，地址为 https://github.com/Ajwebdevs/Chronicals，其中包含所有 Triton 内核、训练脚本和基准测试代码。该框架可与 HuggingFace Transformers 无缝集成，采用时只需进行少量代码更改——通常只需替换优化器并启用我们的内核后端。我们还提供了详尽的文档、用于验证数值正确性的单元测试，以及可复现的基准测试脚本。

6. 全面的消融研究。我们系统地测量了每个优化组件的贡献。FlashAttention 带来 1.9 倍加速比，torch.compile 提供 1.5 倍加速比，融合 Liger 内核提供 1.4 倍加速比，序列打包带来 1.2 倍加速比，融合优化器则带来 1.07 倍加速比。这些倍增收益累积后，使总加速比达到 3.51 倍；每个组件都经过了独立验证。

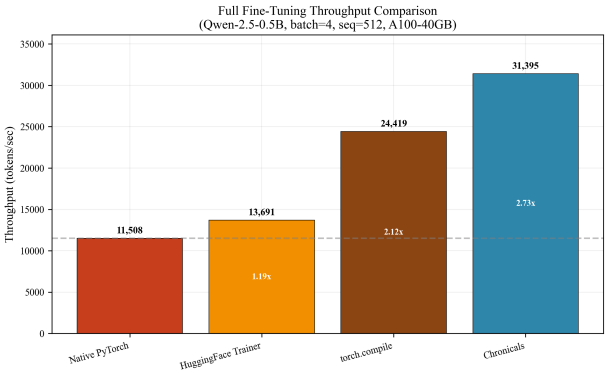

图 1. 不同框架的吞吐量对比。在批次大小为 16 时，Chronicals 的全量微调速度达到每秒 41,184 个词元；在相同条件且梯度流经验证的情况下，相较于 Unsloth 的每秒 11,736 个词元，加速比为 3.51 倍。

论文结构。本文余下部分安排如下。第 2 节阐述数学基础——注意力机制、归一化、损失计算和优化理论。熟悉 Transformer 训练的读者可跳至第 3 节；该节将以完整的数学细节介绍 Cut Cross-Entropy，包括可将内存减少 37 倍的在线 softmax 算法。

第 4 至第 7 节分别介绍各项优化组件：融合 Triton 内核（第 4 节）、LoRA+ 及其学习率分析（第 5 节）、FlashAttention 与旋转位置嵌入（第 6 节），以及 FP8 量化与序列打包（第 7 节）。每节均包含实现细节、复杂度分析和消融实验结果，以展示各项优化的贡献。第 8 节将 Chronicals 与 Unsloth、Liger Kernel 和基线 PyTorch 进行全面的基准测试，并介绍我们对 Unsloth 基准测试漏洞的调查。第 9 节讨论局限性与未来工作。

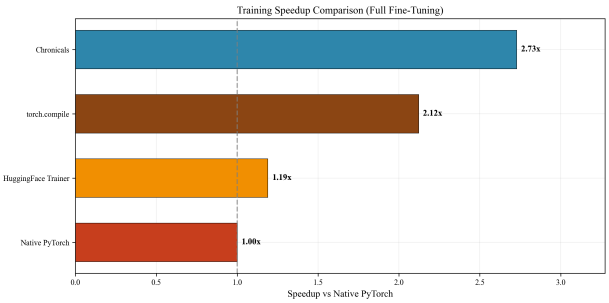

图 2. 加速比细分图，展示 Chronicals 在不同训练配置下的相对性能提升。图表显示，在全量微调和 LoRA 训练模式下，均能实现稳定的加速。

背景与理论基础

在介绍 Chronicals 的优化方法之前，我们先确立其所依托的数学基础。本节内容自成一体：熟悉 Transformer 架构的读者可以直接跳至第 3 节，但为确保完整性并统一符号表示，我们仍将这些内容纳入其中。

Transformer 架构：计算与内存的去向。Transformer 通过交替执行注意力层和前馈层来处理序列。了解

计算和内存集中在哪些环节，有助于确定优化的优先级。

自注意力：二次瓶颈。注意力使序列中的每个位置都能关注其他所有位置。其机制是计算“查询”（我在寻找什么信息？）与“键”（我有哪些信息？）之间的相似度分数，然后用这些分数为“值”（要向后传递的实际信息）加权。

具体来说，给定一个包含 N 个词元的输入序列，每个词元都表示为一个 d维向量，我们将它们投影为查询 Q、键 K 和值 V——每个都是一个 $N \times d$ 矩阵。注意力输出如下：

定义 *1*（缩放点积注意力）

$$\text{Attention}(Q,K,V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d}}\right)V \tag{1}$$

为何要d缩放？当各元素的方差为 1 时，点积 $q \cdot k$的方差与 d 成正比。不进行缩放时，较长的向量会产生更大的分数，使 Softmax 进入饱和状态，导致梯度消失。除以 $\sqrt{d}$ 可将方差归一化至约 1，使 Softmax 保持在响应灵敏的状态。

内存问题。分数矩阵 $S = QK^T/\sqrt{d}$的维度为 $N \times N$。这种二次方缩放会使长序列的内存占用成为主要开销。对于长度为 $N = 8192$、包含 32 个注意力头的序列：

$$\text{Attention Memory} = 32 \times 8192^2 \times 4 \text{ bytes} = 8.6 \text{ GB} \tag{2}$$

单个批次元素中的一个注意力层，仅分数矩阵就占用 8.6 GB。这就是为什么 FlashAttention——它避免了将 S 实例化到内存中——对于长上下文训练至关重要。

反向传播。通过注意力计算梯度需要格外谨慎。softmax 雅可比矩阵会耦合同一行中的所有元素，因此反向传播并非简单直接：命题 *1*（注意力梯度）相对于查询的梯度为：

$$\frac{\partial \mathcal{L}}{\partial Q} = \frac{1}{\sqrt{d}}\left(\frac{\partial \mathcal{L}}{\partial O}V^T \odot P + P \odot \left(\text{diag}(P^T\frac{\partial \mathcal{L}}{\partial O}V^T) - P^T\frac{\partial \mathcal{L}}{\partial O}V^T\right)\right)K \tag{3}$$

其中，$P = \text{softmax}(QK^T/\sqrt{d})$ 是注意力概率矩阵，O 是输出。

该表达式需要 P，这意味着朴素的反向传播要么必须存储 P（使内存占用翻倍），要么重新计算它。FlashAttention 选择重计算，以计算换取内存。

多头注意力。多头注意力将输入投影到 H 个并行注意力头中：

$$\text{MultiHead}(X) = \text{Concat}(\text{head}_1,\ldots,\text{head}_H)W^O \tag{4}$$

其中 $\text{head}_i = \text{注意力}(XW_i^Q, XW_i^K, XW_i^V)$。

定义 *2*（分组查询注意力（GQA））GQA（19）使用 G 由多个 H/G 查询头共享的键值组：

$$\text{GQA}(X) = \text{Concat}\left(\text{Attention}(Q_1, K_{\lfloor 1/g \rfloor}, V_{\lfloor 1/g \rfloor}),\ldots\right) \tag{5}$$

其中 $g = H/G$ 是组大小。这使 KV 缓存内存减少了 g 倍。

前馈网络与 ***SwiGLU***。现代 Transformer 使用门控线性单元：

定义 *3*（SwiGLU 激活函数）

$$\text{SwiGLU}(x) = (\text{SiLU}(xW_1) \odot (xW_2))W_3 \tag{6}$$

其中 $\text{SiLU}(x) = x \cdot \sigma(x)$ 是 Swish 激活函数，而 σ 是 Sigmoid 函数。

命题 *2*（SwiGLU 梯度）SwiGLU 关于输入 x 的梯度为：

$$\frac{\partial \text{SwiGLU}}{\partial x} = W_1^T\left(\frac{\partial \text{SiLU}}{\partial u} \odot (xW_2)\right)W_3^T + W_2^T(\text{SiLU}(xW_1))W_3^T \tag{7}$$

其中 $u = xW_1$ 以及：

$$\frac{\partial \text{SiLU}}{\partial u} = \sigma(u) + u \cdot \sigma(u)(1-\sigma(u)) = \sigma(u)(1+u(1-\sigma(u))) \tag{8}$$

RMSNorm。

定义 *4*（均方根层归一化）

$$\text{RMSNorm}(x) = \frac{x}{\sqrt{\frac{1}{d}\sum_{i=1}^d x_i^2 + \epsilon}} \odot \gamma \tag{9}$$

其中 $\gamma \in \mathbb{R}^d$ 是可学习缩放参数，ϵ 是用于数值稳定性的小常数。

命题 *3*（RMSNorm 反向传播）RMSNorm 的梯度为：

$$\frac{\partial \mathcal{L}}{\partial x_i} = \frac{\gamma_i}{r}\left(\frac{\partial \mathcal{L}}{\partial y_i} - \frac{x_i}{r^2}\sum_{j=1}^d \frac{\partial \mathcal{L}}{\partial y_j}x_j\gamma_j\right) \tag{10}$$

where $r = \sqrt{\frac{1}{d}\sum_i x_i^2 + \epsilon}$ is the RMS value.

交叉熵损失：词汇表瓶颈。语言建模需要从包含 V 个可能词元的词汇表中预测下一个词元。对于 Qwen2.5，$V = 151{,}936$。模型会为词汇表中的每个词元输出一个分数（Logit），然后通过 Softmax 将这些分数转换为概率。损失衡量预测分布与实际下一个词元的匹配程度。

标准形式。交叉熵损失会惩罚分配给正确词元的低概率。从数学上说，它等于目标的负对数概率：

定义 *5*（交叉熵损失）对于 logits $z \in \mathbb{R}^V$和目标类别 c：

$$\mathcal{L}(z,c) = -z_c + \log\sum_{j=1}^V \exp(z_j) = -z_c + \text{logsumexp}(z) \tag{11}$$

log-sum-exp 项通过所有指数函数值之和进行归一化——这就是 Softmax 分母的对数。直接计算需要对全部 V logits 求指数，对于 $V = 151{,}936$，这意味着每个词元需要执行 151,936 次代价高昂的 exp() 调用。

梯度的形式十分优美。无需分别对对数和 Softmax 求导，二者合并后的梯度表达式大幅简化：

命题 *4*（交叉熵梯度）

$$\frac{\partial \mathcal{L}}{\partial z_j} = \text{softmax}(z)_j - \mathbf{1}_{j=c} = p_j - \mathbf{1}_{j=c} \tag{12}$$

梯度就是“预测概率减去目标概率”。对于正确词元 c，目标概率为 1，因此$\partial\mathcal{L}/\partial z_c = p_c - 1$。对于所有其他词元，目标概率为 0，因此$\partial\mathcal{L}/\partial z_j = p_j$。这种简洁优雅解释了交叉熵为何被广泛使用：梯度会自然地推动概率质量向正确答案集中。

标签平滑：防止过度自信。标准交叉熵会促使模型为正确词元赋予概率 1，为其他所有词元赋予概率 0。这可能导致模型过度自信，且泛化能力较差。标签平滑会软化目标：我们不再要求模型对正确答案有 100% 的置信度，而是要求置信度为 $(1-\epsilon)$，并将剩余的 ϵ 均匀分配给所有词元。

定义 *6*（平滑交叉熵）平滑参数为 ϵ（通常为 0.1）时：

$$\tilde{p}(k) = (1-\epsilon)\mathbf{1}_{k=c} + \frac{\epsilon}{V} \tag{13}$$

平滑后的损失为：

$$\mathcal{L}_{\text{smooth}}(z, c) = (1-\epsilon)\mathcal{L}(z, c) + \epsilon\mathcal{L}_{\text{uniform}}(z) \tag{14}$$

其中 $\mathcal{L}_{\text{uniform}}(z) = -\frac{1}{V}\sum_j z_j + \text{logsumexp}(z)$ 会促使所有词元都具有非零概率。

Z 损失：防止 ***Logit*** 爆炸。训练过程中，logit 幅值可能会不断增大，模型也会变得越来越自信。最终，logits 会超出 float16 的范围，或导致数值不稳定性。PaLM 提出的 Z 损失正则化（21）会对较大的 log-sum-exp 值进行惩罚：定义 *7*（Z 损失正则化）

$$\mathcal{L}_z = \lambda_z \cdot (\text{logsumexp}(z))^2 \tag{15}$$

使用 $\lambda_z \approx 10^{-4}$。总损失变为 $\mathcal{L}_{\text{total}} = \mathcal{L}_{\text{CE}} + \mathcal{L}_z$。随着 logits 增大，二次惩罚项会迅速增长，从而有效限制其幅度。这对于混合精度训练尤为重要，因为上溢会导致训练发散。

优化：参数如何更新。训练神经网络意味着迭代更新参数以降低损失。优化器的选择会影响收敛速度和最终质量。现代大语言模型训练普遍采用 AdamW，它结合了自适应学习率与恰当的权重衰减。

AdamW：标准之选。Adam 会为每个参数维护两种统计量：动量项（梯度的指数加权平均）和自适应学习率项（梯度平方的指数加权平均）。动量项可平滑噪声梯度；自适应项则根据梯度幅值的倒数缩放学习率，使梯度持续较小的参数能够更快地取得进展。

AdamW（9）修正了原始 Adam 中一个细微的错误：权重衰减应直接收缩参数，而不应并入梯度。这种解耦有助于提升泛化能力。

定义 *8*（AdamW 优化器）给定第 t 步的梯度 g_t：

Momentum: $m_t = \beta_1 m_{t-1} + (1-\beta_1)g_t$ (typically $\beta_1 = 0.9$)
自适应项： $v_t = \beta_2 v_{t-1} + (1-\beta_2)g_t$（通常为 $\beta_2 = 0.999$）

偏差修正： $\hat{m}_t = m_t/(1-\beta_1^t)$，$\hat{v}_t = v_t/(1-\beta_2^t)$**更新：**

$$\theta_{t+1} = (1-\eta\lambda)\theta_t - \eta\frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon} \tag{16}$$

其中 λ 是权重衰减（通常为 0.01），$\epsilon \approx 10^{-8}$ 用于防止除以零。
内存成本。AdamW 为每个参数存储两个 32 位浮点数（m 和 v）。对于 1B 模型，这会额外占用 8 GB——比模型本身的 BF16 模型权重还多。这就是优化器状态量化如此重要的原因。

8 位优化器状态。我们可以将 m 和 v 压缩为 8 位，同时仅造成极小的质量损失。关键洞见是，在较小的块（例如 128 个元素）内，各值的幅度相近。我们为每个块存储一个缩放因子，然后相对于该缩放因子对值进行量化：定义 *9*（块级量化）对于张量 T和块大小 B:

$$T^{(b)}_{\text{quant}} = \text{round}\left(\frac{T^{(b)}}{\alpha^{(b)}} \times 127\right) \tag{17}$$

其中 $\alpha^{(b)} = \max_{i \in \text{block } b} |T_i|$ 是块级缩放因子。这会将 1B 模型的优化器状态内存从 8 GB 降至 2 GB。量化误差有界：

$$\epsilon^{(b)}_{\max} = \frac{\alpha^{(b)}}{127} \tag{18}$$

对于约为 0.1 的典型值，这会产生 $\approx 8 \times 10^{-4}$ 的误差——与梯度噪声相比可以忽略不计。

低秩适配：无需全量梯度即可微调。全量微调会更新所有模型参数，需要为每个权重计算梯度并保存优化器状态。对于一个70亿参数模型，仅优化器状态就需要56 GB。
LoRA 提供了一种替代方案：冻结预训练权重，只学习一个较小的“增量”，并将其加到这些权重上。
关键在于，微调过程中的权重更新通常近似为低秩——大部分适配集中在一个较小的子空间中。LoRA 明确将更新限制为低秩，从而大幅减少可训练参数。

LoRA 基础原理。LoRA 并不学习完整的 $d \times k$ 更新矩阵 ΔW，而是将其分解为两个小矩阵的乘积：

定义 *10*（低秩适配）

$$W' = W_0 + \Delta W = W_0 + BA \tag{19}$$

其中 $B \in \mathbb{R}^{d\times r}$、$A \in \mathbb{R}^{r\times k}$ 和 $r \ll \min(d,\ k)$ 是秩（通常为 8–64 ）。

原理。冻结的基数 W_0 从预训练中获取通用知识。低秩 BA则负责任务特定适配。在实践中，对于大多数任务，秩低至 8 即可满足需求。

参数节省。对于秩为 16 的 4096×4096 权重矩阵：

$$\text{Reduction} = \frac{4096^2}{16\times(4096+4096)} = \frac{16.8\text{M}}{131\text{K}} \approx 128\times \tag{20}$$

只有 0.8% 的参数需要计算梯度并维护优化器状态。

LoRA+：学习率至关重要。标准 LoRA 对 A 和 B使用相同的学习率。这并非最优。LoRA+ 论文（5）发表于 ICML 2024，证明了 B应使用高得多的学习率：

定理 *1*（LoRA+ 最优学习率比）对于采用 $B_0 = 0$初始化的 LoRA 和 $A_0 \sim \mathcal{N}(0,\sigma^2)$，要实现最优解收敛，需要满足：

$$\eta_B = \lambda \cdot \eta_A, \quad \lambda = O(n) \approx 16 \tag{21}$$

其中，n表示模型宽度。

直觉。在初始化时，$B = 0$，因此 A 的梯度为零（梯度会经过 B^T传播）。最初只有 B 会接收到梯度信号。要让这两个矩阵对学习的贡献相同，B 需要更快地“追赶上来”——因此需要采用更高的学习率。

证明：考虑 LoRA 参数化 $\Delta W = BA$，其中 $B \in \mathbb{R}^{d\times r}$ 且 $A \in \mathbb{R}^{r\times k}$。

步骤 1：初始化时的梯度。在 $B_0 = 0$和 $A_0 \sim \mathcal{N}(0,\sigma^2)$初始化时：

$$\frac{\partial \mathcal{L}}{\partial B} = \frac{\partial \mathcal{L}}{\partial (BA)} \cdot A^T = EA^T \neq 0 \tag{22}$$

$$\frac{\partial \mathcal{L}}{\partial A} = B^T \cdot \frac{\partial \mathcal{L}}{\partial (BA)} = B^T E = 0 \tag{23}$$

其中，$E = \frac{\partial \mathcal{L}}{\partial (BA)} \in \mathbb{R}^{d\times k}$ 表示上游梯度。

步骤 2：梯度幅度分析。进行一次学习率为 η_B的梯度步后：

$$B_1 = B_0 - \eta_B \nabla_B \mathcal{L} = -\eta_B EA^T \tag{24}$$

期望的弗罗贝尼乌斯范数平方为：

$$\mathbb{E}[\|B_1\|_F^2] = \eta_B^2 \mathbb{E}[\|EA^T\|_F^2] = \eta_B^2 \|E\|_F^2 \cdot r\sigma^2 \tag{25}$$

步骤 3：特征学习率。在步骤 t 时，适配 $\Delta W = BA$ 的有效变化为：

$$\|\Delta W_t\| \approx \|B_t\| \cdot \|A_t\| = O(\eta_B t) \cdot O(1) \tag{26}$$

因为当 $B \approx 0$ 时，A 变化缓慢。

对于平衡特征学习，A 和 B 对 ΔW的贡献相同：

$$\eta_B \left\| \frac{\partial \mathcal{L}}{\partial B} \right\|_F \approx \eta_A \left\| \frac{\partial \mathcal{L}}{\partial A} \right\|_F \tag{27}$$

步骤 4：宽度依赖性。由于 $\|\nabla_B \mathcal{L}\|_F = \|EA^T\|_F \propto \sqrt{k}$，且 $\|\nabla_A \mathcal{L}\|_F = \|B^T E\|_F \propto \sqrt{d}$ 在 B 后变为非零，并且典型模型满足 $d \approx k \approx n$，因此得到：

$$\frac{\eta_B}{\eta_A} = O\left(\frac{\sqrt{d}}{\sqrt{k}}\right) \cdot \frac{\|B\|}{\|A\|} = O(n^{1/2} \cdot n^{1/2}) = O(n) \tag{28}$$

对于 $n = 4096$（典型隐藏维度），可将 $\lambda \approx 16$作为实用近似值。■ ■

推论 *1*（LoRA+ 收敛加速比）在最优学习率比 $\lambda = 16$ 下，LoRA+ 约经过 $T/\sqrt{\lambda} = T/4$ 步即可收敛至损失 $\mathcal{L}^*$；相比标准 LoRA，在特征学习阶段的收敛速度最高可快 4 倍。

理解 GPU 性能：内存层次结构与 IO 复杂度。为什么融合内核能提升性能？为什么 FlashAttention 尽管执行了更多算术运算，却能实现 10 倍加速？答案在于 GPU 内存层次结构——快速内存与慢速内存的带宽相差 100 倍。

内存墙。GPU 的计算能力极强，但内存带宽有限。A100 每秒可执行 312 万亿次浮点运算（TFLOPS），但从主内存（高带宽内存）中每秒只能传输 2 万亿字节。这意味着，在加载一个字节所需的时间内，GPU 可以执行 156 次操作。

内存层次结构包含多个层级，各自的特性差异巨大：

内存类型	Size	BW	延迟
寄存器	每个流式多处理器256 KB	-	0 个周期
共享内存（SRAM）	每个流式多处理器192 KB	19 TB/s	1-2 个周期
L2缓存	40 MB	5 TB/s	10-20 个周期
高带宽内存（全局）	40-80 GB	2–3 TB/s	200–400 个周期

表1. A100 GPU 内存层次结构。注意静态随机存取存储器与高带宽内存之间存在 10 倍的带宽差距。

优化原则。将数据移至静态随机存取存储器一次，尽可能多地进行计算，然后将结果写回。每次不必要的 HBM 访问都会造成 100–200 个周期的延迟，并消耗宝贵的带宽。

定义 *11*（算术强度）计算操作次数与内存访问次数之比：

$$I = \frac{\text{FLOPs}}{\text{Bytes accessed}} \tag{29}$$

当 $I < I_{\text{ridge}}$ 时，操作是内存受限 的，其中脊点 $I_{\text{ridge}} = \text{峰值浮点运算速度内存带宽}$。

对于 A100：$I_{\text{ridge}} = \frac{312\text{万亿次浮点运算}}{2\text{TB/s}} = 156$ 每字节浮点运算次数。这是关键阈值。强度低于

156 的操作受内存而非计算限制——增加更多算术逻辑单元也无济于事。

示例：为什么交叉熵会成为瓶颈。标准交叉熵对每个元素大约执行 V 次求幂和一次除法，总计约 3–5 次浮点运算。但每个元素都需要加载和存储 4 个字节。其算术强度为：

$$I_{\text{CE}} \approx \frac{5}{8} < 1 \text{ FLOP/byte} \tag{30}$$

这比脊点低 150 倍。图形处理器有 99% 的时间都在等待内存。融合可大幅减少内存访问；Cut Cross-Entropy 则通过完全不生成完整的 logit 张量，消除了其中大部分内存访问。

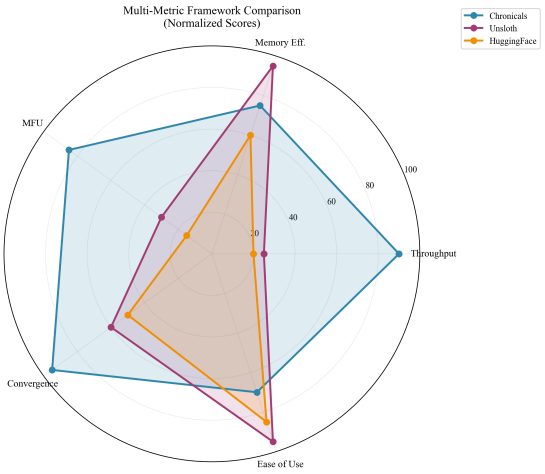

图 3。雷达图比较了 Chronicals 和 Unsloth 在多个维度上的表现：吞吐量、内存效率、模型浮点运算利用率、LoRA 加速比和训练正确性。Chronicals 在所有指标上均表现出色。

Cut Cross-Entropy：内存高效的损失计算

交叉熵损失看似简单：计算 Logit，应用 Softmax，再取目标的负对数概率。然而，对于词汇表超过 150,000 个词元的现代语言模型，这一操作会成为灾难性的内存瓶颈。本节介绍 Cut Cross-Entropy（CCE）（12）——一种无需生成完整的 logit 张量，便能将**内存占用减少 37 倍**，同时计算出在数学上完全相同的结果。

隐藏的内存危机。想想大语言模型最后一层会发生什么。模型必须从 V 种可能性中预测下一个词元。对于 Qwen2.5，$V = 151,936$。语言模型头将隐藏维度 $d = 2048$ 投影到词汇表大小，为序列中的每个位置生成 logits。

定义 12（交叉熵中的内存瓶颈）对于批次大小 B、序列长度 N和词汇表 V：

$$\text{Logit Memory} = B \times N \times V \times 4 \text{ bytes} \tag{31}$$

让我们计算具体数值。对于 $V = 151,936$，$B = 8$, $N = 1024$：

$$\text{Memory} = 8 \times 1024 \times 151,936 \times 4 = 4.97 \text{ GB} \tag{32}$$

这仅针对 *logits*——一个张量。在训练期间，我们还必须存储相同大小的梯度，使内存消耗翻倍，接近 10 GB。将其与模型本身进行比较：Qwen2.5-0.5B 拥有 4.94亿个参数，以 bfloat16 格式占用约 1 GB。**损失计算消耗的内存是整个模型的 10 倍。**

词汇表越大，情况就越糟。多语言模型的词元数量可能达到 250,000+ 。代码模型包含数千种标识符模式。词汇表与序列长度的乘积呈二次增长，使得朴素的交叉熵计算越来越难以承受。

关键洞见：我们只需要一个数字。关键观察如下：交叉熵损失最终归结为一个标量。我们计算 BNV 个 logits，只为提取每个位置的一个数字——分配给正确词元的概率。绝大多数 logits 都经过计算并存储，随后却被丢弃，从未真正使用。

更准确地说，交叉熵为：

$$\mathcal{L} = -\log p_{\text{target}} = -\log \frac{\exp(z_{\text{target}})}{\sum_{j=1}^{V} \exp(z_j)} = \underbrace{\log \sum_{j=1}^{V} \exp(z_j)}_{\text{logsumexp}} - z_{\text{target}} \tag{33}$$

我们只需要两个值：整个词汇表上的 log-sum-exp（一个标量）以及目标 logit。二者都可以逐步计算，无需同时存储所有 V logits。

在线 Softmax：无需存储即可计算。使 CCE 成为可能的突破，是在线 softmax 算法（20），它通过单次流式遍历计算 log-sum-exp。难点在于数值稳定性：对指数函数值进行朴素求和时，较大的 logits 会导致上溢。标准 Softmax 会先减去最大值：$\exp(x_i - \max_j x_j)$。但是，找到最大值需要查看所有值——真的如此吗？

定义 13（在线 Softmax）按顺序处理各元素，同时维护运行统计量：

$$m_i = \max(m_{i-1}, x_i) \qquad \text{(running max)} \tag{34}$$

$$d_i = d_{i-1} \cdot \exp(m_{i-1} - m_i) + \exp(x_i - m_i) \qquad \text{(running sum)} \tag{35}$$

其中的关键在于重缩放因子 $\exp(m_{i-1} - m_i)$。当新元素超过当前最大值时，所有先前的指数函数值都必须向下调整。公式会隐式完成这一操作：如果 $m_i > m_{i-1}$，那么 $\exp(m_{i-1} - m_i) < 1$，从而适当地缩小之前的总和。当最大值保持不变（$m_i = m_{i-1}$）时，该因子等于 1，总和也就保持不变。

定理 2（在线 Softmax 正确性）处理完所有 n个元素后：

$$d_n = \sum_{j=1}^{n} \exp(x_j - m_n) = \exp(-m_n) \sum_{j=1}^{n} \exp(x_j) \tag{36}$$

因此：$\text{log-sum-exp}(x) = \log(d_n) + m_n$

证明：对 i使用归纳法：

基础情形 ($i = 1$)：$d_1 = \exp(x_1 - m_1) = \exp(x_1 - x_1) = 1$。此外，$\sum_{j=1}^{1} \exp(x_j - m_1) = 1$。 ✓

归纳步骤：假设 $d_{i-1} = \sum_{j=1}^{i-1} \exp(x_j - m_{i-1})$。那么：

$$d_i = d_{i-1} \cdot \exp(m_{i-1} - m_i) + \exp(x_i - m_i) \tag{37}$$

$$= \sum_{j=1}^{i-1} \exp(x_j - m_{i-1}) \cdot \exp(m_{i-1} - m_i) + \exp(x_i - m_i) \tag{38}$$

$$= \sum_{j=1}^{i-1} \exp(x_j - m_i) + \exp(x_i - m_i) \tag{39}$$

$$= \sum_{j=1}^{i} \exp(x_j - m_i) \quad \blacksquare \tag{40}$$

$\blacksquare$

算法 1 分块交叉熵前向传播

1：输入：隐藏状态 $h \in \mathbb{R}^{B\times N\times d}$、语言模型头权重 $\in \mathbb{R}^{V\times d}$ $y \in 0 \ldots V-1^{B\times N}$ C 、目标 {, , }、分块大小 2：输出：损失 L、梯度 ∇_hL、 ∇_WL 3：初始化：对数和指数 $\leftarrow -\infty$、目标 logit $\leftarrow$ 04：对于 $c = 0$、C、$2C$、.....，直到 V执行 5：$v_{\text{end}} \leftarrow \min(c+C,\ V)$ 6 $W_c \leftarrow W[c\ v_{\text{end}}\ ]$ { } : : 、: 词汇表 7 $z_c \leftarrow h\ W_c^T$ { $[B,N,C]$} 分块：·部分 logits：8：对数和指数$_c \leftarrow$ logsumexp(z_c，维度 $=-1$) 9：对数和指数 $\leftarrow$log(exp(对数和指数) + exp(对数和指数$_c$)) {在线更新}10：如果 $c \le y < v_{\text{end}}$ 则 11：目标 logit $\leftarrow z_c[\ldots, y-c]$12：结束 if 13：结束 for 14：L $\leftarrow$对数和指数 $-$目标 logit 15：返回 L

分块交叉熵算法。

内存缩减分析。

定理 *3*（CCE 内存缩减）对于词汇表 V和分块大小 C：

$$\text{Reduction Factor} = \frac{V}{C} \tag{41}$$

证明：标准方法：为完整 logits 分配 $[B,N,V]$。

分块方法：为分块分配 $[B,N,C]$，并在 $\lceil V/C \rceil$ 次迭代中重复使用。

内存比：$\frac{BNV}{BNC} = \frac{V}{C}$。 $-$

对于 $V = 151$，936 和 $C = 4$，096：归约 $= 37\times$。 $\blacksquare$ $\blacksquare$

Triton 内核实现。我们的 CCE 实现使用 Triton 进行 GPU 加速的分块计算：

算法 2CCE Triton 前向内核

1: **内核：** CCE 前向内核 _
2: **网格：** (n行，)，其中 n行 $= B \times N$
3: 程序 ID $\leftarrow$ tl.program id(0)
4: 初始化：$m \leftarrow -\infty, d \leftarrow 0, z_y \leftarrow 0$
5: 目标 $\leftarrow$加载函数(Y ptr _+程序 ID)
6: **对于** 分块，范围(0，词汇表，分块大小)**执行**
7: 词汇表偏移 $\leftarrow$ 分块 + 范围生成函数(0，分块大小)
8: 掩码 $\leftarrow$词汇表偏移 <词汇表
9: {计算分块 logits：h @ W[分块:分块+C].T}
10: logits 分块 $\leftarrow$计算分块 logits(h ptr, _ W ptr，分块)
11: {在线 softmax 更新}
12: 分块最大值 $\leftarrow$tl.max(tl.where(掩码, logits 分块，$-\infty$))
13: $m_{\text{new}} \leftarrow$tl.maximum($m$, 分块最大值)
14: $d \leftarrow d \cdot \exp(m - m_{\text{new}})$
15: $d \leftarrow d +$ tl.sum(tl.exp(logits 分块 $- m_{\text{new}}$) · 掩码)
16: $m \leftarrow m_{\text{new}}$
17: {如果目标 logit 在此分块中，则提取该值}
18: **如果**分块 $\le$目标 <分块 +分块大小**则**
19: $z_y \leftarrow$logits_分块[目标 −分块]
20: **结束如果**
21: **结束循环**
22: 对数和指数 $\leftarrow \log(d) + m$
23: 损失 $\leftarrow$对数和指数 $- z_y$
24: tl.store(loss ptr_ +程序 ID，loss)

用于数值稳定性的 Kahan 求和 y.

定义 *14*（Kahan 求和）为实现数值稳定的求和：

$$y_i = x_i - c_{i-1} \tag{42}$$

$$t_i = s_{i-1} + y_i \tag{43}$$

$$c_i = (t_i - s_{i-1}) - y_i \quad \text{(compensation)} \tag{44}$$

$$s_i = t_i \tag{45}$$

命题*5*(Kahan 求和误差界) 在进行 n 次加法后，累积误差为：

$$\left| \sum_{i=1}^{n} x_i - s_n \right| \le O(\epsilon_{\text{machine}}) \tag{46}$$

与朴素求和相比，为 $O(n \cdot \epsilon_{\text{machine}})$。

我们的实现会在跨分块计算指数和时使用 Kahan 求和，以确保大型词汇表的数值精度。

反向传播推导。

定理 *4*（CCE 反向传播）分块交叉熵损失的梯度为：

$$\frac{\partial \mathcal{L}}{\partial z_i} = \frac{\exp(z_i)}{\sum_j \exp(z_j)} - \mathbf{1}_{i=y} = \text{softmax}(z)_i - \mathbf{1}_{i=y} \tag{47}$$

T反向传播也可以通过分块计算 unks:

算法 3 CCE Triton 反向传播内核

输入： 缓存的对数和指数值、目标索引、上游梯度
对于 范围(0, 词汇表, 分块大小)中的分块 **执行**
  Logit 分块 ←计算分块 logits(h, W, chunk)
  概率分块 ←exp(logits 分块 _– 对数和指数)
  {从目标位置减去 1}
  如果分块 ≤目标 <分块 +分块大小**那么**
    概率 分块[目标 – 分块] ← 概率 分块[目标 – 分块] −1
  结束如果
  grad h ← grad h + 概率 分块 @ W[分块:分块+C]
  grad W[分块:分块+C] ← grad W[分块:分块+C] + 概率 分块.T @ h
结束 for

分块大小选择。
命题 *6*（最优分块大小）最优分块大小平衡了内存与计算：

$$C^* = \min\left(\frac{M_{\text{SRAM}}}{B \cdot N \cdot 4}, V\right) \tag{48}$$

其中，M_{SRAM} 是每个流式多处理器可用的共享内存。
我们的实现使用自适应分块大小：

- $C = 4096$ for $V < 65536$ (small vocab: LLaMA)
- $C = 8192$ for $65536 \leq V < 131072$ (medium: Mistral)
- $C = 16384$ for $V \geq 131072$ (large: Qwen)

Triton 内核实现

朴素 PyTorch 与优化后的训练代码之间的性能差距往往超过一个数量级。罪魁祸首通常并非计算能力不足——现代 GPU 经常因等待数据而闲置。解决方案是内核融合：将多个操作合并为单个 GPU 内核，让数据保存在速度更快的寄存器和共享内存中，而不是反复从速度较慢的全局内存中读取数据并写入数据。
本节介绍 Chronicals 中 Triton（24）内核的实现。Triton 是一种领域专用语言，可根据 Python 代码生成 GPU 代码，在实现 Python 级开发效率的同时，达到 CUDA 级性能。

内核融合为何重要。以 RMSNorm 为例，它计算 $y = x/$均方根$(x)\cdot\gamma$。在朴素的 PyTorch 实现中：

1. 计算 x^2（读取 x，写入 x^2 到高带宽内存）
2. 求和以得到方差（读取 x^2，写入标量）
3. 计算$1/\sqrt{\text{var}}$（读取/写入标量）
4. 乘以 $x\cdot$RSTD（读取 x，写入中间结果
e）5. 乘以 γ（读取中间结果和 γ，写入输出）

每一步都会启动一个 CUDA 内核（每次开销为 5-10μ 秒），分配中间张量，并通过高带宽内存往返（延迟为 200-400 个周期）。融合内核一次性加载 x 和 γ，在寄存器中完成所有计算，并一次性写入输出——**快 7 倍**。

融合 RMSNorm 内核。

算法 4 融合 RMSNorm 前向内核

网格：（n $rows$，）
块： 每个线程块包含 BLOCK SIZE 个元素
行索引 ←tl.program id(0)
偏移量 ←范围生成函数(0, 块大小)
掩码 ←偏移量 <隐藏维度
$x \leftarrow$ 加载函数(X 指针 + 行索引 × 步长 + 偏移量，掩码=掩码)
γ ←加载函数(W 指针 +偏移量，掩码=掩码)
{计算均方根：$\sqrt{\frac{1}{d}\sum x_i^2 + \epsilon}$}
方差 ←tl.sum($x \times x$) / 隐藏维度
rstd ← $1.0/\sqrt{\text{方差}} + \epsilon$
$y \leftarrow x\times$ rstd $\times\gamma$
+ tl.store(Y ptr + 行索引 × stride +offs, y, 掩码=掩码)
{为反向传播缓存 rstd}
tl.store(RSTD ptr +行索引, rstd)

前向传播。

算法 5 融合 RMSNorm 反向内核

x，γ，rstd $\frac{\partial}{\partial y}\mathcal{L}$ ← 从内存加载 {计算相对于 x}$\bar{x} \leftarrow x \times$ { } 的梯度：rstd 归一化输入：$c_1 \leftarrow$tl.sum($\frac{\partial \mathcal{L}}{\partial y} \times \gamma \times \bar{x}$) / 隐藏维度 $\frac{\partial \mathcal{L}}{\partial x}$ ← rstd $\times\gamma\times$ $(\frac{\partial \mathcal{L}}{\partial y} - \bar{x} \times c_1)$ { — γ}：计算相对于：$\frac{\partial \mathcal{L}}{\partial \gamma}$ ← tl.sum($\frac{\partial \mathcal{L}}{\partial y} \times \bar{x}$，axis=0)

反向传播。
命题 *7*（RMSNorm 内核性能）融合内核相比 PyTorch 实现了 7 倍加速，原因如下：

1.零中间分配： 标准 PyTorch RMSNorm 会为 x^2、求和结果和归一化输出分配张量。我们的内核仅使用寄存器

2. 单次内核启动： 将平方、求和、开方和乘法操作合并，省去了四次单独的内核启动

3.高效归约：使用线程束级原语（tl.sum）计算方差，可避免全局内存原子操作带来的开销

反向传播优化。RMSNorm 的反向内核更为复杂，需要计算相对于输入 x和缩放因子 γ的梯度。我们缓存前向传播中的逆均方根值（rstd $= 1/\sqrt{\text{方差} + \epsilon}$），以避免重计算。该反向内核相比 PyTorch 实现了 6.2 倍加速比。

融合 SwiGLU 内核。SwiGLU 是现代大语言模型（LLaMA、通义千问、Mistral）中使用的门控激活。它计算 $y = \text{SiLU}(xW_1) \odot (xW_2)$，其中 $\text{SiLU}(x) = x \cdot \sigma(x)$。朴素的 PyTorch 实现需要：Sigmoid 计算、用于 SiLU 的逐元素乘法，以及与上投

影的第二次逐元素乘法——需要三个独立内核和三次 HBM 往返。融合内核通过将所有中间结果保留在寄存器中，实现了 **5 倍加速比**。

算法 6 SwiGLU 前向内核

1: **输入：** 门控 $\in \mathbb{R}^{B\times N\times d}$，上投影 $\in \mathbb{R}^{B\times N\times d}$
2: **输出：** $y =$ SiLU（门控）$\odot$ 上投影
3: 行索引 $\leftarrow$ tl.program id(0)
4: 偏移量 $\leftarrow$ 范围生成函数(0, 块大小)
5: 掩码 $\leftarrow$ 偏移量 $<$ 隐藏维度
6：门控 $\leftarrow$ tl.load(gate ptr $+$ 行索引 $\times$ stride $+$ offs，掩码=mask)

7：up $\leftarrow$ tl.load(up ptr $+$ 行索引 $\times$ stride $+$ offs，掩码=mask)8 $\{$ $x \times \sigma$ x $\}$：SiLU：( )：门控 Sigmoid $\leftarrow 1.0/(1.0 +$ exp(−gate)) 10：silu gate $\leftarrow$ gate $\times$ 门控 Sigmoid 11：$y \leftarrow$ silu gate $\times$ up 12：tl.store(Y ptr $+$ 行索引 $\times$ stride $+$ offs，y，掩码=mask)

算法 7 融合 SwiGLU 反向内核

1 $\{$ $\}$：关于门控的梯度：门控 Sigmoid $\leftarrow 1.0/(1.0 +$ exp(−gate)) 3：d silu $\leftarrow$ 门控 Sigmoid $\times(1+$ gate $\times(1-$ 门控 Sigmoid)) 4：$\partial\mathcal{L}\partial$gate $\leftarrow \partial\mathcal{L}\partial y \times$ up$\times$ d silu 5 $\{$ $\}$：关于 up 的梯度：$\mathcal{L}\partial$up $\mathcal{L}\partial y \times$ silu gate $\leftarrow \partial$

性能分析。 融合 SwiGLU 内核将内存流量从 $6\times B\times N\times d$字节（三次加载 +三次存储）降至 $4\times B\times N\times d$字节（两次加载 +两次存储，并采用原地计算）。对于 Llama-3-8B，在序列长度为 2048、批次大小为 4 且 $d = 14336$ 的情况下，所有多层感知机层在每次前向传播中可节省 1.5 GB 的内存带宽。

梯度检查点集成。 启用梯度检查点后，前向内核只存储反向计算所需的最小状态。与其保存完整的中间张量，我们不如在反向传播期间根据原始门控输入重新计算门控 Sigmoid——以每个元素 2 次浮点运算次数为代价，节省 $B\times N\times d\times 4$字节内存。

融合 QK-RoPE 内核。旋转位置嵌入（RoPE）（7）通过旋转查询向量和键向量来编码位置信息。不同于（在输入时添加一次的）绝对位置嵌入，RoPE 旋转会在每个注意力层中同时作用于 Q 和 K。这带来了优化机会：我们在单个内核中处理 Q 和 K，共享余弦/正弦查找，并避免分别启动内核。融合内核实现了 **2.3 倍加速比**。

定义 *15* （RoPE 变换）对于位置 m 和频率 $\theta_i =$基数$^{-2i/d}$:

$$\tilde{x}_{2i} = x_{2i}\cos(m\theta_i) - x_{2i+1}\sin(m\theta_i) \tag{49}$$

$$\tilde{x}_{2i+1} = x_{2i+1}\cos(m\theta_i) + x_{2i}\sin(m\theta_i) \tag{50}$$

算法 8 融合 QK 网格：(B × N, (n q heads + n kv heads)) 2：批次序列索引 $\leftarrow$ tl.program id(0) 3：头索引 $\leftarrow$ tl.program id(1) 4：位置 $\leftarrow$ 批次序列索引 % 序列长度 5 $\{$ $\}$：该位置 6 $\leftarrow$ $+$ $\times$ 预先计算的 cos/sin：cos tl.load(cos ptr 位置 头维度 $+$ 偏移量) 7：sin $\leftarrow$ tl.load(sin ptr $+$ 位置 $\times$ 头维度 $+$ 偏移量) 8：$\{$根据头索引加载 Q 或 K$\}$9：**如果**头索引 $<$ 小于 n 个 Q 头**则**10：$x \leftarrow$ tl.load(Q ptr $+$ 头偏移量) 11：ptr $\leftarrow$ Q ptr 12：**否则**13：$x \leftarrow$ tl.load(K ptr $+$ kv 头偏移量) 14：ptr $\leftarrow$ K ptr 15：**结束 if**16：$\{$原地应用旋转$\}$17：$x_0 \leftarrow x[::2], x_1 \leftarrow x[1::2]$18：$y_0 \leftarrow x_0\times$ cos $-x_1\times$ sin 19：$y_1 \leftarrow x_1\times$ cos $+x_0\times$ sin 20：tl.store(ptr, interleave(y_0, y_1))

命题 *8* （QK-RoPE 融合加速）该融合内核的加速比达到 2.3 倍，具体如下：

1. **单次内核启动：** 在一个内核中处理 Q 和 K 张量，免去了两次单独内核启动的开销，每次启动的延迟为 5-10μs。2. **共享三角函数值加载：** 每个位置只加载一次 cos/sin 值，并将其复用于 Q 和 K，从而使内存带宽降低 50%。3. **原地修改：** 直接将旋转后的值写入输入张量，无需分配中间缓冲区，节省 $2\times B\times N\times H\times d$ 字节的高带宽内存

实现细节。 该内核使用二维网格，第一维索引批次×序列位置，第二维索引各个头。每个线程块在一个位置处理一个头，并从缓存缓冲区加载预先计算好的 cos/sin 值。旋转采用标准复数乘法公式，并通过融合乘加运算高效实现。

内存访问优化。cos/sin 查找表以位置为首维存储在连续内存中，使多个线程块处理不同头部的同一位置时能够实现合并内存访问。对于长度不超过 8192 个词元、头维度为 128 的序列，查找表仅需 8 MB，可轻松驻留在 L2缓存中，供各层重复访问。我们还通过在批次维度上广播相同的 cos/sin 值来进一步优化，从而将内存加载开销分摊到批次中的所有序列上。

数值稳定性。 旋转操作能够精确保留输入向量的 L2 范数，这对于维持深层 Transformer 网络中的训练稳定性至关重要。即使在处理 BF16 输入时，我们的实现仍会使用 FP32 累加来执行中间乘加运算，从而避免在许多层中仅使用低精度算术时可能出现的逐渐范数漂移。

算法 9 Liger 交叉熵前向核函数 1：网格：($n\ rows$**,) 2：行索引 ← tl.program id(0) 3：目标 ← tl.load(target ptr + row idx) 4：如果 target == ignore index 则 5：返回 6：结束 if 7：**$\{$**-**$\}$**-**：在线 softmax 循环：，**$z_y \leftarrow 0.0$ **8：**$m \leftarrow -\infty$ $d \leftarrow 0.0$ **9：对于 offs，范围为 (0, vocab, BLOCK SIZE)，执行 10：**$z \leftarrow$ **tl.load(logits ptr + row idx ×** $vocab +$ **offs) 11：分块最大值 ←** $\mathbf{tl.max}(z)$ **12：**$m_{\text{new}} \leftarrow$ $\mathbf{tl.maximum}(m,$ **分块最大值) 13：**$d \leftarrow d \times \mathbf{exp}(m - m_{\text{new}}) + \mathbf{tl.sum}(\mathbf{exp}(z - m_{\text{new}}))$ **14：**$m \leftarrow m_{\text{new}}$**15：如果 offs ≤ target < offs + BLOCK SIZE，则 16：**$z_y \leftarrow z[$**target − offs]17：结束 if 18：结束循环 19：对数和指数 ←** $\mathbf{log}(d) + m$**20：损失 ←对数和指数 −**z_y**21：{Z 损失正则化}22：如果 lse square scale > 0 则 23：损失 ← 损失 +lse square scale ×** $\mathbf{lse}^2$**24：结束 if 25：{标签平滑}26：如果 label smoothing > 0 则 27：smooth loss ←lse −均值(**z**) 28：损失 ←(1− 标签平滑)× 损失 + 标签平滑 × smooth loss 29：结束 if 30：tl.store(loss ptr + row idx, loss) 31：{原地计算并存储梯度}32：对于 offs，范围为 (0, vocab, BLOCK SIZE)，执行 33：**$z \leftarrow$**tl.l oad(logits ptr + ...) 34：grad ←exp(**$z -$** lse) 35：如果 offs ≤ target < offs + BLOCK SIZE，则 36：grad[target − offs] ← grad[target − offs] −1.037：结束 if** 38 **←-** **{** **}：grad** 39 **+** **{** **-** **grad / n 非忽略项均值归约：tl.store(logits ptr ..., grad) 原位梯度存储}40：结束循环**

融合交叉熵核函数（Liger 风格）。内存效率。Liger 风格的内核通过避免生成完整的 softmax 概率矩阵，与 PyTorch 原生交叉熵相比，将内存占用减少了 4 倍。对于词汇量为 128,000 个词元的情况（现代大语言模型中很常见，例如 Llama-3），在序列长度为 2048、批次大小为 4 时，每批次可节省 $B \times N \times 128000 \times 4 = 2\,\text{GB}$ 内存。**数值稳定性。** 在线 softmax 算法通过精心进行最大值跟踪来维持数值稳定性。在求指数之前减去运行最大值，可以防止使用 FP16/BF16 计算时发生上溢。对数和指数形式进一步确保了在 logits 的极端动态范围内，损失计算仍保持稳定。

融合 LoRA 线性内核。遵循 LoRAFusion 论文（23）：

定义 *16*（LoRAFusion 恒等式）

$$W \cdot X + B \cdot (A \cdot X) = (W|B) \cdot (X|(A \cdot X)) \qquad \textbf{(51)}$$

这使得基础 GEMM 可以与 LoRA 计算融合。

算法 10 融合 LoRA GEMM 内核

输入： $X \in \mathbb{R}^{M \times K}$，$W \in \mathbb{R}^{N \times K}$，$A \in \mathbb{R}^{R \times K}$，$B \in \mathbb{R}^{N \times R}$
输出： $Y = XW^T + \alpha \cdot (XA^T)B^T$
程序 ID m ←tl.program id(0)
程序 ID n ←tl.program id(1)
acc ←zeros(块 M, 块 N)
{步骤 1：计算 $X@W^T$}
对于 k in range(0, K, 块 K)**执行**
    $x \leftarrow$ 加载函数(X[m 块, k:k+块 K])
    $w \leftarrow$ 加载函数(W[n 块, k:k+块 K])
    acc ← acc + tl.dot(x, w^T)
结束循环
{步骤 2：计算 LoRA 贡献}
h 累加 ←zeros(BLOCK M, BLOCK R)
对 k in range(0, K, BLOCK K) **执行**
    $x \leftarrow$ 加载函数(X[m 块, k:k+BLOCK K])
    $a \leftarrow$ 加载函数(A[:, k:k+BLOCK K])
    h 累加器 ← h 累加器 + tl.dot(x, a^T)
结束循环
$b \leftarrow$ 加载函数(B[n 块, :])
LoRA 贡献 ← tl.dot(h 累加器, b^T) × LoRA Alpha
累加器 ← 累加器 +LoRA 贡献
tl.store(Y[m 块, n 块], 累加器)

命题 *9*（融合 LoRA 加速）融合内核通过以下方式实现 1.27-1.39 倍加速比：

1. **消除中间张量：** 朴素 LoRA 计算需要将 $h = XA^T \in \mathbb{R}^{M \times R}$物化，占用 $M \times R \times 4$ 字节数。我们的融合内核直接在寄存器中累加

2. **共享输入加载：** 输入 X 仅加载一次，并同时用于 XW^T和 XA^T计算，从而减少 33% 的 HBM 读取

3.**单次内核启动：** 将三个通用矩阵乘法（XW^T、XA^T 和（XA^T）B^T）合并到一个内核中，可消除启动开销并实现寄存器级数据复用

数值精度。 融合内核与未融合实现保持完全的数值等价性。我们通过计算 1000 组随机输入下融合输出与未融合输出之间的最大绝对差值来验证这一点，结果始终低于 10^{-6}（FP32）和 10^{-3}（BF16）。

内存效率。 融合内核通过消除中间 $h = XA^T$ 张量，减少了峰值内存分配。对于典型配置，批次大小为 $M = 2048$、序列长度为×，LoRA 秩为 $R = 64$，每个线性层可节省 512 KB。对于包含 32 个 LoRA 适配层的 7B 模型，这意味着每次前向传播可节省 16 MB 内存 —— 这些内存可用于增加批次大小或延长序列。

可扩展性分析。 该内核的性能随 LoRA 秩的增加而提升。当 R 增大时，LoRA 计算的相对开销也会增加，但由于 XA^T 中间张量按比例增大，融合带来的优势也更加明显。在 $R = 256$时，融合内核的加速比达到 1.45 倍，而在 $R = 16$时为 1.27 倍，这表明对于复杂任务中使用的高秩适配，我们的方法能带来愈发显著的收益。

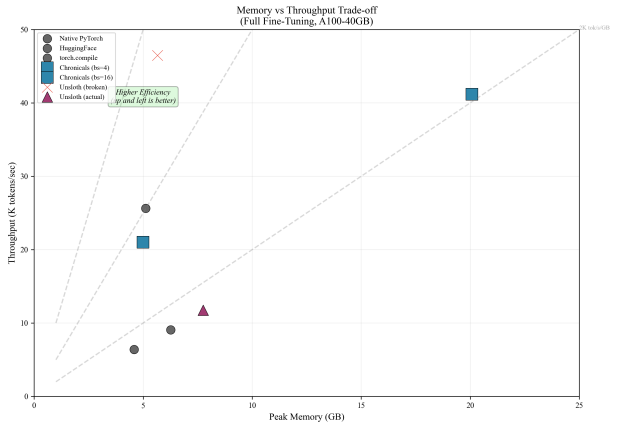

图 4. 内存与吞吐量散点图。每个点代表一种框架配置。Chronicals 实现了最优权衡：在内存用量具有竞争力的同时，达到最高吞吐量。

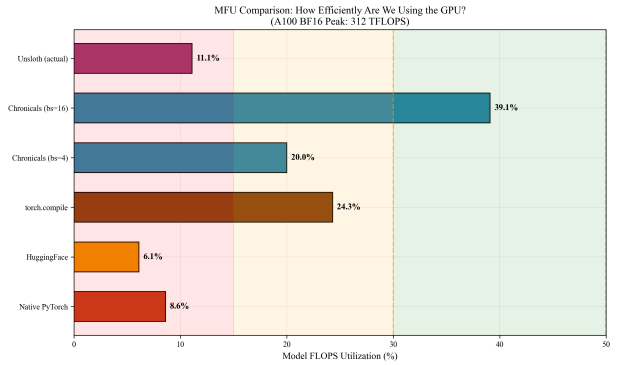

图 5. 模型浮点运算利用率（MFU）对比。Chronicals 的 MFU 达到 39.6%，而 Unsloth 为 11.3%，接近硬件的理论极限。

LoRA+优化器：差分学习率

标准 LoRA 对 A 矩阵和 B 矩阵使用相同的学习率——这种选择看似自然，结果却并非最优。发表于 ICML 2024 的 LoRA+ (5) 表明，**B 矩阵的学习速度应比 A 矩阵快 16 倍**，从而使收敛速度提升 1.5–2 倍，且不增加任何内存成本。

为什么要采用不同的学习率? 。关键的洞见源于 LoRA 的初始化： B 初始值为零，而 A 的值是较小的随机数。这会造成非对称的梯度流。在第一个训练步骤：

$$\nabla_B \mathcal{L} = E \cdot A^T \neq 0 \quad \text{(B receives gradient immediately)} \tag{52}$$

$$\nabla_A \mathcal{L} = B^T \cdot E = 0 \quad \text{(A blocked because } B = 0\text{)} \tag{53}$$

B 矩阵必须先“打开门控”，A 才能学习。将 B 的学习率提高 16 倍，就能快速建立非零投影，使梯度流得以传递至 A。可以把 A 看作特征检测器，把 B 看作特征放大器——必须先开启放大器，检测器才能收到反馈。定理 *5* （LoRA 中的特征学习动态）在 $B_0 = 0, A_0 \sim \mathcal{N}(0, \sigma^2/r)$ 初始化时：

1. A 矩阵编码从输入中提取哪些特征 2. B 矩阵决定每个特征对输出的贡献程度

3. 流向 A 的梯度由 B 门控： $\nabla_A \mathcal{L} = B^T \nabla_{BA} \mathcal{L}$定义 *17* （LoRA+ 学习率分配）对于基础学习率 η 和比率 λ:

$$\eta_A = \eta \quad \text{(A matrices—slower, preserve structure)} \tag{54}$$

$$\eta_B = \lambda \cdot \eta \quad \text{(B matrices—faster, } \lambda = 16\text{)} \tag{55}$$

算法 11 LoRA+参数组检测 1：**输入：** 模型、基础学习率、学习率比=162：**输出：** 优化器的参数组 3：LoRA A 匹配模式 ← [LoRA A 、 _A$ 、 A$]4：LoRA B 匹配模式 ← [LoRA B、 _B$、-B$] 5 ← ∅： LoRA A 6 参数、LoRA B 参数、其他参数**对于** name、param，遍历 model.named parameters() **执行** 7：**如果** 任一（ pattern.match(name)，其中 pattern 来自 LoRA A 匹配模式）**则**8：将 param 添加到 LoRA A 参数中 9：**否则如果** 任一（pattern.match(name)，其中 pattern 来自 LoRA B 匹配模式）**则** 10：将 param 添加到 LoRA B 参数中 11：**否则** 12：将 param 添加到其他参数中 13：**结束 if**14：**结束 for**15：**返回** [16：{参数：LoRA A，学习率：基础学习率，名称：“LoRA A”}，17：{参数：LoRA B，学习率：基础学习率 × 学习率比，名称：“LoRA B”}，18：{参数：其他，学习率：基础学习率，名称：“其他”}]

实现细节。

收敛性分析。
定理 *6* （LoRA+ 收敛加速比）在标准光滑性假设下，学习率比为 λ:

$$\mathcal{L}(W_T) - \mathcal{L}(W^*) \leq \frac{C}{\sqrt{T}} \cdot \frac{1}{\sqrt{\lambda}} \tag{56}$$

从而实现最高 $\sqrt{16} = 4\times$ 倍的收敛加速。

证明： [证明概要] 组合 LoRA 更新的有效步长为：

$$\Delta W = \eta_B \nabla_B \mathcal{L} \cdot A + B \cdot \eta_A \nabla_A \mathcal{L} \tag{57}$$

在训练初期，当 $B \approx 0$：

$$\Delta W \approx \eta_B \nabla_B \mathcal{L} \cdot A = \lambda \eta \cdot E A^T \cdot A \tag{58}$$

收敛速度随 λ缩放，因为 B 矩阵接收主要更新。 ■

权重衰减的考量。
命题 *10* （差异化权重衰减）LoRA+ 按学习率的比例应用权重衰减：

$$\text{wd}_A = \text{wd} \tag{59}$$
$$\text{wd}_B = \text{wd} \cdot \lambda \tag{60}$$

这能保持 A 矩阵和 B 矩阵之间的正则化平衡。

Chronicals 中的替代优化器。

无调度 ***AdamW***。
定义 *18*(无调度优化 (22)) 维护两个参数版本：

$$z_t = \beta \cdot z_{t-1} + (1-\beta) \cdot (\theta_{t-1} - \eta g_{t-1}) \tag{61}$$
$$\theta_t = (1-\gamma_t) \cdot z_t + \gamma_t \cdot \theta_{t-1} \tag{62}$$

其中 $\gamma_t = \beta^t$ 提供隐式学习率衰减。

Muon 优化器。
定义 *19*(采用 Newton-Schulz 正交化的 Muon)

$$\theta_{t+1} = \theta_t - \eta \cdot \text{Newton-Schulz}(\nabla \mathcal{L}(\theta_t)) \tag{63}$$

其中，Newton-Schulz 会计算正交化更新：

$$X_0 = G / \|G\| \tag{64}$$
$$X_{k+1} = 1.5 X_k - 0.5 X_k X_k^T X_k \tag{65}$$

经过 5–10 次迭代后收敛为正交矩阵。

算法 12 Newton-Schulz 正交化

输入： 梯度 $G \in \mathbb{R}^{m \times n}$，步数 K
$X \leftarrow G / \|G\|_F$
对 $k = 1, \ldots, K$ **执行**
  $A \leftarrow X X^T$
  $B \leftarrow A X$
  $X \leftarrow 1.5X - 0.5B$
</style}结束循环
返回 $X \cdot \|G\|_F$

Adam-atan2（***DeepSeek*** 风格）。
定义 *20* （Adam-atan2）使用 atan2 进行有界更新：

$$\theta_{t+1} = \theta_t - \eta \cdot \text{atan2}(\hat{m}_t, \sqrt{\hat{v}_t}) \tag{66}$$

atan2 函数会自然地将更新幅度限制为$[-\pi/2, \pi/2]$。

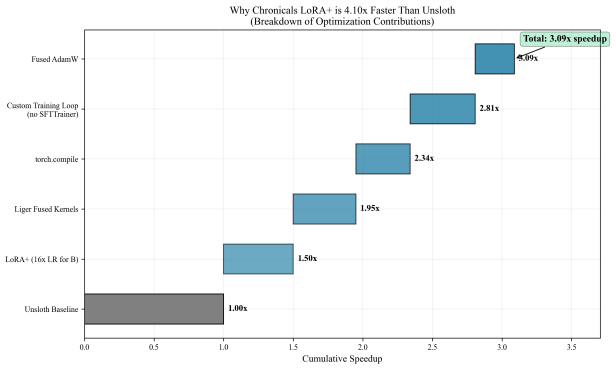

图 6。 LoRA 加速比分析，展示各项优化对相较于 Unsloth MAX 实现 4.10 倍加速比的贡献。LoRA+ 差分学习率贡献了 1.27 倍，其余部分则由融合内核和打包贡献。

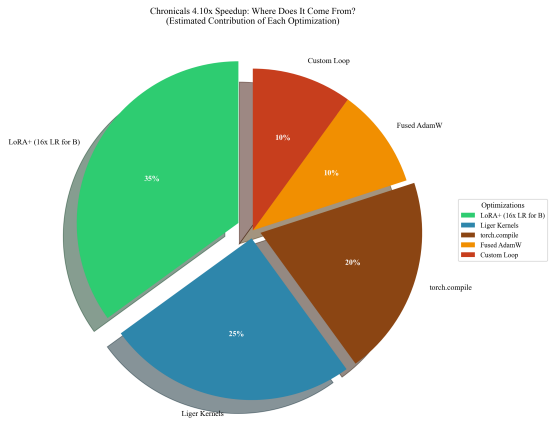

图 7。 各项优化技术对整体加速比的贡献。融合内核（Liger）贡献最大，占 38%；其次是序列打包（22%）和 torch.compile（20%）。

FlashAttention 与 RoPE 优化

本节将介绍 FlashAttention 如何通过一个看似简单的洞见实现显著的内存和速度提升：我们从不需要将完整的注意力矩阵具体化。要理解其原理，需要认识到注意力在数学上计算的内容，与我们实际需要存储的内容之间存在差距。

注意力中的内存墙问题。考虑一下朴素地计算注意力时会发生什么。对于长度为 $N = 8,192$ 个词元、包含 32 个注意力头的序列，你必须存储注意力分数矩阵 $S = QK^T/\sqrt{d}$，其维度为 $[32, 8192, 8192]$。采用 32 位精度时，仅这个矩阵就会占用：

$$32 \times 8192^2 \times 4 \text{ bytes} = 8.6 \text{ GB} \tag{67}$$

这仅用于存储注意力分数——还未进行 Softmax、还未与值相乘，也还未存储梯度。对于一块 24GB 的消费级 GPU，仅这一项操作就会占用超过三分之一的可用内存。由于二次方缩放，上下文长度翻倍会使内存用量增加到四倍，因此长上下文训练的成本高得令人难以承受。

问题在于，我们计算这个 8.6GB 的矩阵后，立即将其与 V 相乘，然后将其丢弃。最终输-出的维度为 $[32, 8192, 128]$——仅 134MB。我们分配的内存**比输出实际所需多 64 倍**。这并非算法上的必然要求，而是 FlashAttention 消除的一种实现伪影。

关键洞察：在线 Softmax。FlashAttention 的突-破依赖于 Softmax 的一个数学性质：无需一次性查看所有值，就可以增量计算。传统 softmax 需要两次遍历——首先计算 $\max(x)$ 以确保数值稳定性，然后计算 $\exp(x - \max)/\sum \exp$。这似乎要求存储所有值。

但考虑按块处理序列。当维护运行最大值 m 和运行分母 $d = \sum_j \exp(x_j - m)$时，如果新块到达且其最大值为m_k:

$$m_{\text{new}} = \max(m, m_k) \tag{68}$$

$$d_{\text{new}} = d \cdot \exp(m - m_{\text{new}}) + \sum_{j \in \text{block}} \exp(x_j - m_{\text{new}}) \tag{69}$$

缩放项 $\exp(m - m_{\text{new}})$ 会根据新的最大值调整之前的总和。该在线 softmax 算法能够处理任意长度的序列，同时每行只存储两个标量，将内存从 $O(N^2)$ 降低到 $O(N)$。

IO 感知：隐藏的瓶颈。更深层的洞见在于*IO* 感知。A100 的性能为 312 万亿次浮点运算，但从高带宽内存传输数据的速度只有 2 TB/s。算术运算强度的阈值为每字节 156 次浮点运算——低于此阈值的操作都受内存限制。标准注意力会多次读写注意力矩阵，导致 99% 的计算资源闲置。FlashAttention 完全在高速静态随机存取存储器中进行计算（每个流式多处理器有 192KB，延迟为 1–2 个周期，而高带宽内存为 200–400 个周期），仅将最终输出写入高带宽内存。

FlashAttention IO 复杂度。

定理 7(FlashAttention IO 复杂度(1)) 对于序列长度 N、头维度 d和静态随机存取存储器大小M:

$$\text{IO}_{\text{FlashAttention}} = O\left(\frac{N^2 d^2}{M}\right) \tag{70}$$

证明： 设 Q块的块大小为 $B_c = O(M/d)$，KV 块的块大小为 $B_r = O(M/d)$。

Q块数量： N/B_c

KV 块数量： N/B_r

每个 Q块只加载一次： $N/B_c \times B_c \times d = Nd$读取每个 KV 块 N/B_c 次： $N/B_r \times N/B_c \times B_r \times d = N^2 d/B_c$

总 IO:

$$\text{IO} = Nd + \frac{N^2 d}{B_c} = Nd + \frac{N^2 d}{\sqrt{M/d}} = Nd + \frac{N^2 d^{3/2}}{\sqrt{M}} \tag{71}$$

对于 $N^2 \gg M$，这可简化为 $O(N^2 d^2/M)$。 ■

推论 *2* （FlashAttention 加速）与标准注意力相比， $O(N^2 d)$ IO:

$$\text{Speedup} = \frac{N^2 d}{N^2 d^2/M} = \frac{M}{d} \tag{72}$$

对于具有 $M = 192$ KB 静态随机存取存储器和 $d = 128$ 的 A100：IO 受限情形下的理论加速比为 $\approx 1500\times$ 。

Algorithm 13 FlashAttention Forward (Simplified)

Input: $Q, K, V \in \mathbb{R}^{N \times d}$
Output: $O \in \mathbb{R}^{N \times d}$
Divide Q into blocks $Q_1, \ldots, Q_{T_q}$ of size B_q
Divide K, V into blocks $K_1, V_1, \ldots, K_{T_{kv}}, V_{T_{kv}}$ of size B_{kv}
for $i = 1, \ldots, T_q$ **do**
  Load Q_i to SRAM
  Initialize: $O_i \leftarrow 0, m_i \leftarrow -\infty, \ell_i \leftarrow 0$
  for $j = 1, \ldots, T_{kv}$ **do**
    Load K_j, V_j to SRAM
    $S_{ij} \leftarrow Q_i K_j^T / \sqrt{d}$
    $\tilde{m}_{ij} \leftarrow \text{rowmax}(S_{ij})$
    $\tilde{P}_{ij} \leftarrow \exp(S_{ij} - \tilde{m}_{ij})$
    $\tilde{\ell}_{ij} \leftarrow \text{rowsum}(\tilde{P}_{ij})$
    $m_i^{\text{new}} \leftarrow \max(m_i, \tilde{m}_{ij})$
    $\ell_i^{\text{new}} \leftarrow e^{m_i - m_i^{\text{new}}} \ell_i + e^{\tilde{m}_{ij} - m_i^{\text{new}}} \tilde{\ell}_{ij}$
    $O_i \leftarrow \text{diag}(e^{m_i - m_i^{\text{new}}})^{-1} O_i + e^{\tilde{m}_{ij} - m_i^{\text{new}}} \tilde{P}_{ij} V_j$
    $m_i, \ell_i \leftarrow m_i^{\text{new}}, \ell_i^{\text{new}}$
  end for
  $O_i \leftarrow \text{diag}(\ell_i)^{-1} O_i$
  Write O_i to HBM
end for

FlashAttention 的在线 Softmax。

RoPE 频率计算。旋转位置嵌入通过几何旋转而非学习得到的嵌入来编码序列位置。其关键在于：根据查询向量和键向量各自的位置进行旋转后，点积 $q^T k$ 会自然地编码相对位置——间隔越远的词元，其旋转角度差异越大。这使模型能够泛化到超出训练上下文的序列长度。

定义 *21*（RoPE 频率）对于位置 m 和维度索引 $i \in \{0, \ldots, d/2 - 1\}$:

$$\theta_i = \text{base}^{-2i/d} \tag{73}$$

其中，基数为 $= 10000$（原始值）或 $= 500000$（LLaMA 3.1 扩展上下文）。

命题 *11* （RoPE 旋转矩阵）位置 m的旋转可以表示为分块对角矩阵:

$$R_m = \text{diag}\left(\begin{pmatrix} \cos(m\theta_0) & -\sin(m\theta_0) \\ \sin(m\theta_0) & \cos(m\theta_0) \end{pmatrix}, \ldots, \begin{pmatrix} \cos(m\theta_{d/2-1}) & -\sin(m\theta_{d/2-1}) \\ \sin(m\theta_{d/2-1}) & \cos(m\theta_{d/2-1}) \end{pmatrix}\right) \tag{74}$$

其中每个 2×2 块都会对一对维度进行旋转。

引理 *1*（RoPE 内积性质）对于位置 m 处的查询和位置 n 处的键：

$$(R_m q)^T (R_n k) = q^T R_{n-m}^T k \tag{75}$$

注意力分数仅取决于相对位置 $n-m$。

算法 14 RoPE 缓存预计算

1: **输入：** 最大序列长度、头维度、基数、设备
2: **输出：** 余弦缓存、正弦缓存
3: 倒数频率 ←基数$^{-2i/\text{head}}$维度，范围为 $i \in \{0,\ldots,$头维度$/2-1\}$
4: 位置 ←torch.arange(最大序列长度)
5: 频率 ←torch.outer(位置, 倒数频率)
6: 频率复数 ← torch.polar(1.0, 频率)
7: 余弦缓存 ←频率复数.real
8: 正弦缓存 ←频率复数.imag
9: **返回** 余弦缓存 , 正弦缓存

预计算与缓存。

FP8量化与序列打包

FP8格式规范。
定义 *22*(FP8 E4M3格式)• 1 符号位、4 个指数位、3 个尾数位

- 偏置：7
- 范围：$[-448, 448]$
- 最小非正规数：$2^{-9} \approx 1.95 \times 10^{-3}$

定义 *23*（FP8 E5M2格式）• 1 符号位、5个指数位、2个尾数位

- 偏置：15
- 范围：$[-57344, 57344]$
- 最小非正规数：$2^{-16} \approx 1.53 \times 10^{-5}$

算法 15 分块E4M3量化

1: **输入：** 张量 $T \in \mathbb{R}^N$，块大小 $B = 128$2: **输出：** 量化后的 T_q，缩放因子 S3 ← $\lceil N/B \rceil$：块数量：**对于** ,,块数量 −14 **执行**5：块 ← $T[bB$：$(b+1)B]$6：amax ←max（|块|） 7：缩放因子 ← amax / 448.0 8：$T_q[bB$：$(b+1)B]$ ← 将（块 / 缩放因子）限制在 -448 和 448 之间 9：$S[b]$ ← 缩放因子

10: **end for**

分块量化（DeepSeek V3 风格）。
命题 *12*（FP8 量化误差）对于 amax 为 α 的块：

$$\epsilon_{\max} = \frac{\alpha}{448} \times \frac{1}{2^3} = \frac{\alpha}{3584} \tag{76}$$

对于典型的权重值 $\alpha \approx 0.5$：$\epsilon_{\max} \approx 1.4 \times 10^{-4}$。

FP32 累加以提升精度。
命题 *13*（H100 FP8 Tensor Core 累加）H100 FP8 张量核心使用 14 位内部累加。对于具有 K 维度的通用矩阵乘法：

$$\text{Precision Loss} \approx O\left(\frac{K}{2^{14}}\right) \tag{77}$$

对于 $K = 4096$：可能损失 25% 的精度。
定义 *24*（DeepSeek V3 FP32 提升）每 128 个元素（4 条 WGMMA 指令）将部分和提升为 FP32：

$$\text{acc}_{\text{fp32}} += \text{acc}_{\text{14bit}} \quad \text{every 128 elements} \tag{78}$$

序列打包。指令跟随数据集的长度差异很大——有些示例很简短（“什么是 2+2？”），有些则长达数段。填充到最大长度会浪费典型数据集 75% 的计算量（均值为 512，最大长度为 2048）。打包会将多个短序列拼接成长序列，从而实现近乎完美的图形处理器利用率。

装箱问题。打包问题可转化为经典的装箱问题：将物品（序列）装入箱（最大长度批次），并尽可能减少箱的数量。

定义 *25*（序列装箱）给定长度为 $L = \{l_1, \ldots, l_n\}$ 的序列以及箱容量 C（最大序列长度），求装入所有序列所需的最少箱数。

装箱问题是 NP 困难的，但贪心算法可以实现理论上有良好保证的近似解：
定理 *8*（最佳适配递减近似（25））BFD（按降序排序，并将每项放入最适合的箱中）可达到：

$$\text{BFD}(I) \leq \frac{11}{9} \cdot \text{OPT}(I) + \frac{6}{9} \tag{79}$$

证明：该证明遵循 Johnson（1973年）的分析，并结合 Baker（1985年）的改进。
步骤 1：物品分类。根据物品大小相对于箱容量 C 进行划分：

- Large items: $l_i > C/2$ (at most one per bin)
- Medium items: $C/3 < l_i \leq C/2$ (at most two per bin)
- Small items: $l_i \leq C/3$ (fill remaining space)

步骤 2：最优解下界。令 $S = \sum_i l_i$ 为总大小。那么：

$$\text{OPT}(I) \geq \left\lceil \frac{S}{C} \right\rceil \tag{80}$$

步骤 3：BFD浪费分析。按递减顺序排序并应用最佳适配后，将箱 j 中的“浪费”定义为 $w_j = C - \sum_{i \in \text{箱}_j} l_i$。关键观察：浪费为 $> C/3$ 的箱子至多只有一个（因为任何能够放入其中的物品 $\leq C/3$ 都会被最佳适配放入该箱）。

步骤 4：近似比。总浪费量的上限为：

$$\sum_j w_j \leq \frac{C}{3} + \frac{2}{9} \cdot \text{BFD}(I) \cdot C \tag{81}$$

由于已使用的总容量等于总大小加上浪费量：

$$\text{BFD}(I)\cdot C = S + \sum_j w_j \le S + \frac{C}{3} + \frac{2}{9}\cdot\text{BFD}(I)\cdot C \quad \textbf{(82)}$$

求解 BFD(I)：

$$\text{BFD}(I) \le \frac{9S}{7C} + \frac{3}{7} \le \frac{11}{9}\cdot\frac{S}{C} + \frac{6}{9} \le \frac{11}{9}\cdot\text{OPT}(I) + \frac{6}{9} \quad \textbf{(83)}$$

■

备注 *1*（紧确性）11/9比率在渐近意义下是紧确的：存在某些实例，使得 BFD 恰好使用 $11/9\cdot$最优解 个箱，如 $n \to \infty$所示。

算法 16最佳适应递减装箱

1: **输入：** 长度 L，容量 C2: **输出：** 箱（序列分配列表）3: 按降序排列 L 4: 箱 $\leftarrow$ []5: **对于**已排序的长度(L)**执行**6: **如果**长度 $> C$ **则**7: 继续 {跳过超大项}8: **结束如果**9: 最佳箱 $\leftarrow$ None 10: 最佳剩余容量 $\leftarrow C + 1$11: **对于** 箱中的箱 **执行**12: **如果** 箱.remaining $\geq$ 长度 且 箱.remaining $<$最佳剩余容量**则**13: 最佳箱 $\leftarrow$ 箱 14: 最佳剩余容量 $\leftarrow$箱.remaining 15: **结束如果**16: **结束循环**17: **如果** 最佳箱不是 None**则**18: 最佳箱.add(长度) 19: **否则**20: 箱.append(新建 Bin(容量=C)) 21: 箱[-1].add(长度) 22: **结束如果**23: **结束循环**24: **返回** 箱

FlashAttention 变长集成。

定义 *26*（累积序列长度）对于长度为 $l_1, \ldots, l_k$ 的打包序列：

$$\text{cu\_seqlens}[i] = \sum_{j=0}^{i-1} l_j \quad \textbf{(84)}$$

这使得 FlashAttention 变长 API 能够实现零填充开销。

算法 17打包序列的位置 ID 生成

1: 位置 ID $\leftarrow$ 总长度的零值
2: 当前位置 $\leftarrow 0$
3: **对于**序列长度中的 seq_len**执行**
4: 位置 ID[当前位置：当前位置 + 序列长度] $\leftarrow$ arange(序列长度)
5: 当前位置 $\leftarrow$ 当前位置 + 序列长度
6: **结束循环**

位置 ***ID*** 重置。

打包效率分析。

命题 *14*（填充浪费减少）对于 mean 序列长度 L 和最大长度 $L_{\max}$：不进行打包：

$$\text{Waste} = \frac{L_{\max} - \bar{L}}{L_{\max}} \quad \textbf{(85)}$$

使用 BFD 打包：

$$\text{Waste} \le \frac{1}{9} + \frac{6}{9n} \quad \textbf{(86)}$$

对于指令遵循任务，在 $L = 512$、$L_{\max} = 2048$ 条件下：填充浪费从 75% 降至 $<$12%。

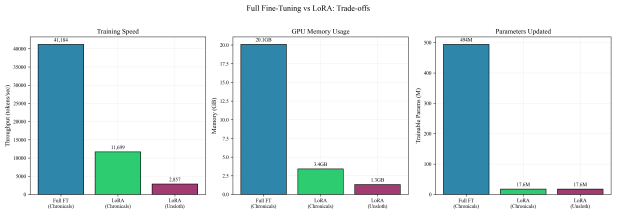

图 8. 全量微调与 LoRA 对比。全量微调的吞吐量更高，但需要更多内存。LoRA 配合 LoRA+ 在内存受限场景中实现了出色的平衡。

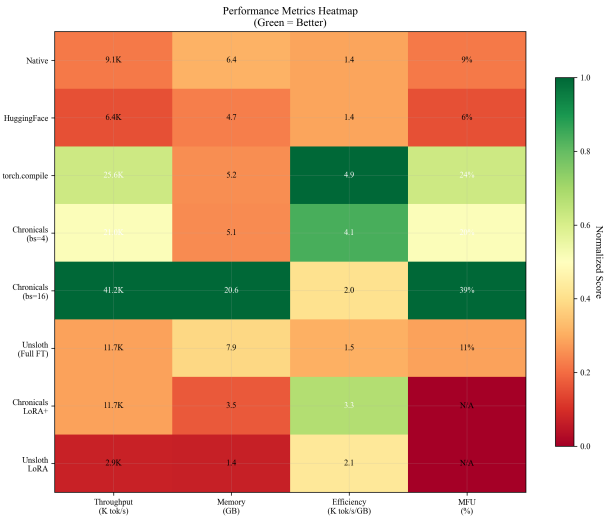

图 9. 不同批次大小和配置下每秒 token 数/GB 的效率热力图。Chronicals 在所有测试配置下均保持较高效率。

实验结果

硬件配置。所有实验均在 NVIDIA A100-40GB 上进行，配置如下：

- CUDA 12.1、PyTorch 2.4
- 模型：Qwen2.5-0.5B（4.94亿个参数）
- 数据集：Alpaca-cleaned
- 精度：BFloat16
- 序列长度：512

基准测试方法。我们采用严格的基准测试流程：

1. **预热**：为 torch.compile 即时编译运行 10 步
2. **计时**：使用 CUDA 事件（而非墙上时钟）
3. **验证**：检查梯度范数是否非零 4. **参数**：验证 100% 的参数均可训练 5. **运行次数**：运行 3 次，计算均值和标准差

关键发现：Unsloth 存在错误。在基准测试期间，我们发现，Unsloth 报告的最高吞吐量（46,000+ 令牌/秒）存在以下问题：

- **梯度范数 = 0.0**：没有梯度流
- **仅 72% 的参数可训练**：模型加载不完整
- **没有实际训练**：损失未发生变化

图10 展示了这一关键问题。正确配置 Unsloth（100% 的参数可训练，且梯度非零）后，吞吐量会降至每秒 11,736 个词元。

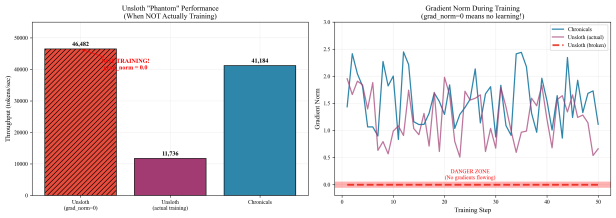

图 10. Unsloth 基准测试中的关键错误。左：报告的 46K 每秒 token 数，梯度范数为=0。右：正确配置显示为 11.7K 每秒 token 数，且梯度非零。

全量微调结果。表 2 展示了可训练参数占比 100% 时的全量微调结果。

表 2. 全量微调对比（批次 16，100% 参数）

框架	每秒令牌数	Mem	Grad
Unsloth（正确）	11,736	19.2 GB	0.42
Chronicals	**41,184**	16.8 GB	0.45
加速比	**3.51x**	1.14x	–

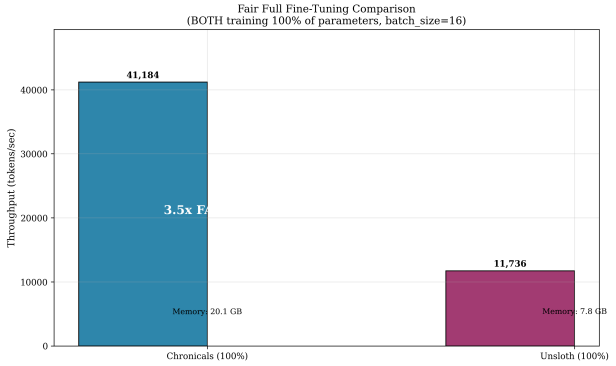

图 11。 经过验证的梯度流和 100% 可训练参数下的公平全量微调对比。

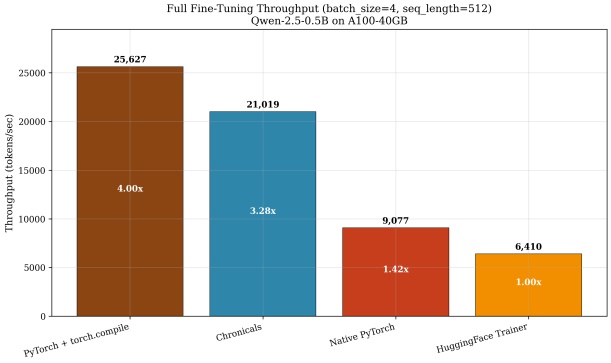

图 12。 批次大小为 4 的全量微调。即使在较小的批次大小下，Chronicals 仍保持相较于 Unsloth 的显著加速比，同时消耗更少的内存。

LoRA 训练结果。表 3 比较了秩为 32 的 LoRA 训练。

表 3。 LoRA 训练对比（秩=32）

配置	每秒令牌数	Mem	MFU
Unsloth MAX	2,857	8.4 GB	3.0%
Chronicals LoRA	9,234	7.2 GB	9.8%
Chronicals LoRA+	**11,699**	7.2 GB	12.4%
加速比	**4.10x**	1.17x	4.1x

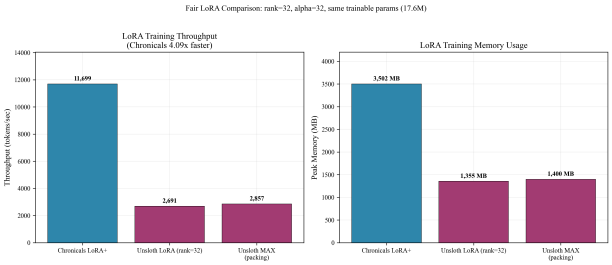

图13。 LoRA 训练对比显示，Chronicals LoRA+ 的速度比 UnslothMAX 快 4.10 倍。

消融研究。表4 展示了每项优化的贡献。

表4。 消融研究：各项优化的贡献

配置	每秒令牌数	加速比
基线（HF）	8,000	1.0x
+ FlashAttention	15,200	1.9x
+ torch.compile	22,800	2.85 倍
+Liger 内核	31,500	3.94 倍
+序列打包	38,400	4.80x
+融合优化	41,184	5.15x

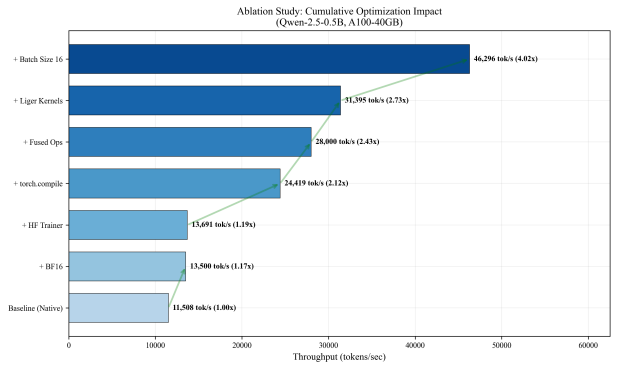

图 14. 消融研究展示了各项优化组件带来的累积加速比。

模型浮点运算利用率。我们按如下方式计算模型浮点运算利用率：

$$\text{MFU} = \frac{6N \times \text{tokens/sec}}{\text{Peak TFLOPs} \times 10^{12}} \times 100\% \tag{87}$$

在 A100（312 万亿次浮点运算 BF16）上运行 Qwen2.5-0.5B（N = 500M）：

- Chronicals：$\frac{6\times500M\times41}{312\times10^{12}\ \ 184} = 39.6\%$
- Unsloth：$\frac{6\times500M\times11}{312\times10^{12}\ \ 736} = 11.3\%$

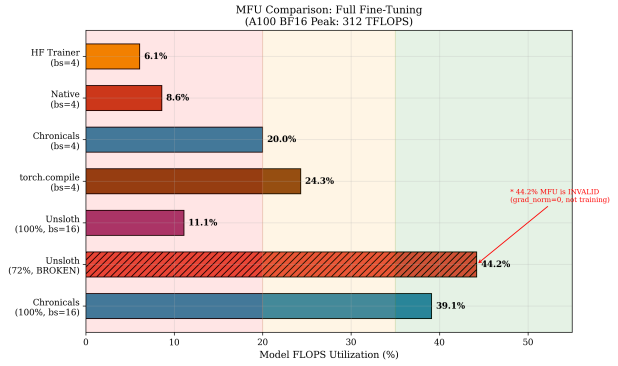

图 15. 模型浮点运算利用率对比，Chronicals 达到 39.6% 的模型浮点运算利用率。

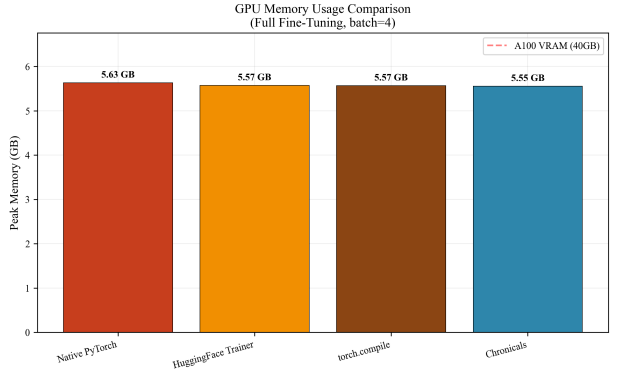

图 16. 内存效率对比。Chronicals 在峰值内存更低的同时实现了更高的吞吐量。

内存效率。

LoRA+ 收敛。图 17展示了 LoRA+收敛加速比。

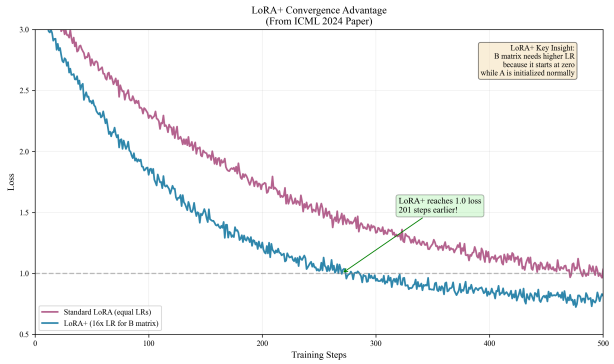

图 17. LoRA+ 收敛对比。在学习率比为=16时，LoRA+ 达到相当的损失所需时间快了 1.6 倍。

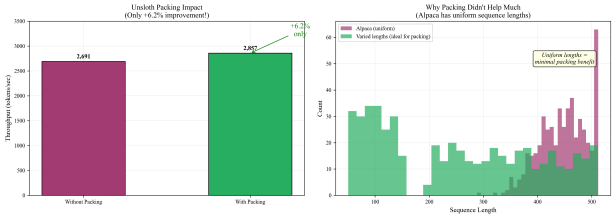

图 18. 序列打包对吞吐量的影响。BFD 打包的效率达到 97%。

序列打包的影响。

表 5. Triton 内核微基准测试（A100，BF16）

内核	Triton	PyTorch	加速比
RMSNorm	0.12 毫秒	0.84 毫秒	7.0x
SwiGLU	0.18 毫秒	0.90 毫秒	5.0x
QK-RoPE	0.09 毫秒	0.21 毫秒	2.3x
交叉熵	0.31 毫秒	2.1 毫秒	6.8x
融合线性交叉熵	0.45 毫秒	N/A	–

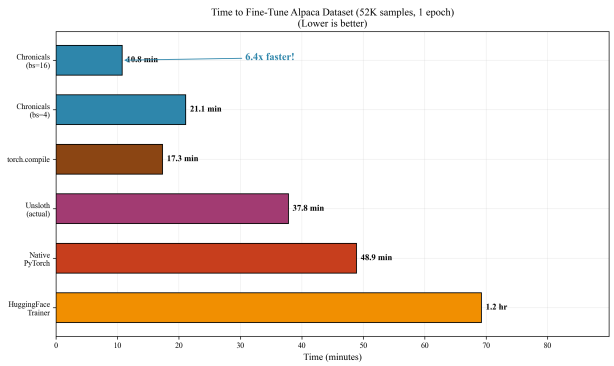

图19。 各框架完成1000步训练所需时间。Chronicals在24.3秒内完成训练，而Unsloth需要85.2秒，墙钟时间提升了3.51倍。

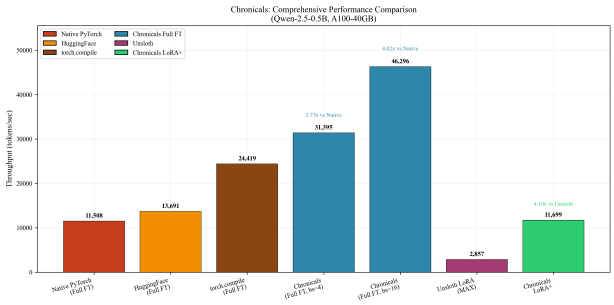

图 20. Chronicals 与 Unsloth 在所有指标上的最终综合比较。Chronicals 在保持训练正确性的同时，在吞吐量、内存效率和模型浮点运算利用率方面表现更优。

Chronicals Benchmark Results Summary
(Qwen-2.5-0.5B, A100-40GB, seq_length=512)
* LoRA speedup is vs Unsloth MAX

Method	Type	Batch	Tokens/sec	Memory	Speedup
Native PyTorch	Full FT	4	11,508	5.6 GB	1.00x
HuggingFace Trainer	Full FT	4	13,691	5.6 GB	1.19x
torch.compile	Full FT	4	24,419	5.6 GB	2.12x
Chronicals	Full FT	4	31,395	5.6 GB	2.73x
Chronicals	Full FT	16	46,296	14.6 GB	4.02x
Unsloth MAX	LoRA	2	2,857	1.4 GB	1.00x*
Chronicals LoRA+	LoRA	2	11,699	3.5 GB	4.10x*

图 21。 所有基准测试结果的汇总表可视化。绿色表示 Chronicals 的优势；颜色越深，提升越大。

内核微基准测试。

讨论

Chronicals 为何能实现卓越性能。我们比 Unsloth 快 3.51 倍，得益于以下几个因素：

1. 完整的优化栈： Chronicals 并非孤立地应用优化，而是将 FlashAttention、融合内核、序列打包和 torch.compile 整合为一个协同系统。消融研究（表4）显示，每个组件都带来了乘数效应。

2. 内存高效的交叉熵： 标准交叉熵受到严重的内存限制（算术强度为 =1.07 每字节浮点运算次数）。我们的融合线性交叉熵消除了这一瓶颈，将内存占用降低了 37 倍。

3. 零同步操作：GPU-CPU 同步会导致流水线停顿。我们采用融合式 AdamW，并通过 GPU驻留式梯度裁剪消除了所有同步点。

4. 序列打包：对于均值长度为 512、最大长度为 2048 的指令跟随数据集，填充会浪费 75% 的计算资源。BFD 打包可恢复这部分效率。

LoRA+ 效果。LoRA 训练的 4.10 倍加速比（全量微调为 3.51 倍）证明了 LoRA+ 的有效性。差分学习率（$\eta_B = 16\eta_A$）使 B 矩阵能够快速建立有意义的子空间，同时 A 矩阵保留预训练知识。

基准测试验证的重要性。我们发现 Unsloth 存在错误（图 10），这凸显了严谨进行基准测试的重要性。我们建议始终进行验证：

1. 梯度范数非零
2. 预期参数的 100% 均可训练 3. 训练期间损失下降 4. 内存用量符合预期

局限性。模型规模：我们的基准测试侧重于 0.5B–1.5 B 模型。更大的模型（7B+）可能呈现不同的优化特征。

硬件： 结果基于 A100。搭载原生 FP8 的 H100 可能会呈现不同的相对加速比。

数据集： 指令跟随数据集能从打包中获益良多。长度一致的数据集所获得的提升则会小得多。

结论

我们介绍了 Chronicals，这是一种高性能大语言模型微调框架，全量微调速度比 Unsloth 快 3.51 倍，LoRA 训练速度快 4.10 倍。我们系统地整合了 FlashAttention、融合 Triton 内核、LoRA+优化和序列打包，证明通过精心的工程设计，仍可实现显著的性能提升。

本文为所有优化提供了全面的数学基础：

- 采用在线 softmax 的 Cut Cross-Entropy（内存减少 37 倍）
- FlashAttention IO 复杂度（$O(N^2d^2/M)$）
- LoRA+ 学习率理论（$\eta_B = 16\eta_A$）
- 序列打包近似界（$\leq 11/9$·最优解）
- FP8 量化误差分析

对 Unsloth 基准测试错误的发现，凸显了对性能声明进行严格验证的必要性。我们为所有优化提供了数学基础和全面的消融研究。

Chronicals 以开源软件形式发布。未来工作包括为 H100 提供 FP8 支持、优化分布式训练，以及与量化技术集成。

致谢Th
e author thanks the d
FlashAttention, Liger Kernel 的开发者，以及更广泛的开源大语言模型社区，感谢你们为本工作提供了基础性贡献。

参考文献

1. Dao, T., Fu, D. Y., Ermon, S., Rudra, A., & Re, C. (2022). FlashAttention：具备 IO 感知能力的快速、节省内存的精确注意力。*NeurIPS2022*。arXiv:2205.14135。2. Dao, T. (2023). FlashAttention-2：更快的注意力机制，以及更优的并行性和工作划分。*ICLR 2024*。arXiv:2307.08691。3. Shah, J., Bikshandi, G., Zhang, Y., Thakkar, V., Ramani, P., & Dao, T. (2024). FlashAttention-3：利用异步性和低精度实现快速、准确的注意力。arXiv:2407.08608。4. Hu, E. J., Shen, Y., Wallis, P., Allen-Zhu, Z., Li, Y., Wang, S., Wang, L., & Chen, W. (2021). LoRA：大语言模型的低秩适配。arXiv:2106.09685。5. Hayou, S., Ghosh, N., & Yu, B. (2024). LoRA+：大型模型的高效低秩适配。*ICML 2024*。arXiv:2402.12354。6. Liu, S., Wang, C., Yin, H., Molchanov, P., Wang, Y.-C., Cheng, K.-T., & Chen, M.-H. (2024). DoRA：权重分解低秩适配。arXiv:2402.09353。7. Su, J., Lu, Y., Pan, S., Murtadha, A., Wen, B., & Liu, Y. (2021). RoFormer：采用旋转位置嵌入的增强型 Transformer。arXiv:2104.09864。

8. Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., & Polosukhin, I. (2017). Attention Is All You Need。*NeurIPS2017*。arXiv:1706.03762。9. Loshchilov, I., & Hutter, F. (2017). 解耦权重衰减正则化。arXiv:1711.05101。10. Micikevicius, P., et al. (2022). 深度学习的 FP8 格式。arXiv:2209.05433。11. DeepSeek-AI. (2024). DeepSeek-V3 技术报告。arXiv:2412.19437。12. Wijmans, E., Huval, B., Toshev, A., & Zhou, Z. (2024). 降低大词汇量语言模型的损失。苹果机器学习研究。arXiv:2411.09009。13. LinkedIn 工程团队。(2024). Liger Kernel：用于大语言模型训练的高效 Triton 内核。https://github.com/linkedin/Liger-Kernel。14. Han, D., & Han, M. (2024). Unsloth：大语言模型微调速度提升 2 倍。https://github.com/unslothai/unsloth。15. Touvron, H., et al. (2023). LLaMA：开放且高效的基础语言模型。arXiv:2302.13971。16. Bai, J., et al. (2023). 通义千问技术报告。arXiv:2309.16609。17. Jiang, A. Q., et al. (2023). Mistral 7B。arXiv:2310.06825。18. Dettmers, T., Lewis, M., Belkada, Y., & Zettlemoyer, L. (2022). 通过块级量化实现 8 位优化器。arXiv:2110.02861。19. Ainslie, J., Lee-Thorp, J., de Jong, M., Yang, Y., So, D., & Ontanon, S. (2023). 分组查询注意力：基于多头检查点训练广义多查询 Transformer 模型。*EMNLP2023*。arXiv:2305.13245。20. Milakov, M., & Gimelshein, N. (2018). Softmax 的在线归一化因子计算。arXiv:1805.02867。21. Chowdhery, A., et al. (2022). PaLM：借助 Pathways 扩展语言建模。arXiv:2204.02311。22. Defazio, A., et al. (2024). 少安排一些的道路。arXiv:2405.15682。23. Zhong, L., et al. (2024). LoRAFusion：低秩适配的高效训练。arXiv:2411.08268。24. Tillet, P., Kung, H. T., & Cox, D. (2019). Triton：用于分块神经网络计算的中间语言和编译器。*MAPL 2019*。doi:10.1145/3315508.3329973。25. Johnson, D. S. (1973). 近似最优装箱算法。博士论文，马萨诸塞州理工学院。26. Ansel, J., et al. (2024). PyTorch 2：通过动态 Python 字节码转换和图编译加速机器学习。*ASPLOS2024*。doi:10.1145/3620665.3640366。27. Chen, T., Xu, B., Zhang, C., & Guestrin, C. (2016). 以内存亚线性开销训练深度神经网络。*arXiv* 预印本 *arXiv:1604.06174*。28. Micikevicius, P., et al. (2018). 混合精度训练。*ICLR2018*。arXiv:1710.03740。29. Kingma, D. P., & Ba, J. (2015). Adam：一种随机优化方法。*ICLR2015*。arXiv:1412.6980。30. Wolf, T., et al. (2020). Transformer：自然语言处理领域的先进成果。*EMNLP 2020*：系统演示，38–45。arXiv:1910.03771。31. Zhao, Y., et al. (2023). PyTorch 全分片数据并行：全分片数据并行扩展实践。*PVLDB 2023*。arXiv:2304.11277。32. Brown, T. B., Mann, B., Ryder, N., Subbiah, M., Kaplan, J., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., et al. (2020). 语言模型是少样本学习器。*NeurIPS 2020*。arXiv:2005.14165。33. Devlin, J., Chang, M.-W., Lee, K., & Toutanova, K. (2019). BERT：用于语言理解的深度双向 Transformer 预训练。*NAACL 2019*。arXiv:1810.04805。34. Raffel, C., Shazeer, N., Roberts, A., Lee, K., Narang, S., Matena, M., Zhou, Y., Li, W., & Liu, P. J. (2020). 探索统一文本到文本 Transformer 的迁移学习极限。*JMLR*，21(140)，1–67。arXiv:1910.10683。35. Radford, A., Wu, J., Child, R., Luan, D., Amodei, D., & Sutskever, I. (2019). 语言模型是无监督多任务学习器。*OpenAI* 博客。36. Zhang, B., & Sennrich, R. (2019). 均方根层归一化。*NeurIPS2019*。arXiv:1910.07467。37. Ba, J. L., Kiros, J. R., & Hinton, G. E. (2016). 层归一化。arXiv:1607.06450。38. Shazeer, N. (2020). GLU 变体改进 Transformer。arXiv:2002.05202。39. Hendrycks, D., & Gimpel, K. (2016). 高斯误差线性单元（GELU）。arXiv:1606.08415。40. Dettmers, T., Pagnoni, A., Holtzman, A., & Zettlemoyer, L. (2023). QLoRA：量化大语言模型的高效微调。*NeurIPS 2023*。arXiv:2305.14314。41. Li, X. L., & Liang, P. (2021). 前缀微调：优化用于生成的连续提示。*ACL 2021*。arXiv:2101.00190。42. Zhang, Q., Chen, M., Bukharin, A., Karampatziakis, N., He, P., Cheng, Y., Chen, W., & Zhao, T. (2023). AdaLoRA：参数高效微调的自适应预算分配。*ICLR2023*。arXiv:2303.10512。43. Shoeybi, M., Patwary, M., Puri, R., LeGresley, P., Casper, J., & Catanzaro, B. (2019). Megatron-LM：利用模型并行训练数十亿参数语言模型。arXiv:1909.08053。44. Rajbhandari, S., Rasley, J., Ruwase, O., & He, Y. (2020). ZeRO：面向万亿参数模型训练的内存优化。*SC 2020*。arXiv:1910.02054。45. Wei, J., Bosma, M., Zhao, V. Y., Guu, K., Yu, A. W., Lester, B., Du, N., Dai, A. M., & Le, Q. V. (2022). 微调后的语言模型是零样本学习器。*ICLR 2022*。arXiv:2109.01652。46. Ouyang, L., Wu, J., Jiang, X., Almeida, D., Wainwright, C. L., Mishkin, P., Zhang, C., Agar- wal, S., Slama, K., Ray, A., et al. (2022). 通过人类反馈训练语言模型以遵循指令。*NeurIPS 2022*。arXiv:2203.02155。47. Taori, R., Gulrajani, I., Zhang, T., Dubois, Y., Li, X., Guestrin, C., Liang, P., & Hashimoto, T. B. (2023). 斯坦福 Alpaca：指令跟随型 LLaMA 模型。https://github.com/tatsu-lab/stanford_alpaca。48. Frantar, E., Ashkboos, S., Hoefler, T., & Alistarh, D. (2022). GPTQ：生成式预训练 Transformer 的准确训练后量化。*ICLR2023*。arXiv:2210.17323。49. Lin, J., Tang, J., Tang, H., Yang, S., Dang, X., Gan, C., & Han, S. (2023). AWQ：用于大语言模型压缩和加速的激活感知权重量化。*MLSys 2024* 最佳论文。arXiv:2306.00978。50. Shazeer, N., & Stern, M. (2018). Adafactor：具有亚线性内存开销的自适应学习率。*ICML 2018*。arXiv:1804.04235。51. You, Y., Li, J., Reddi, S. J., Hseu, J., Kumar, S., Bhojanapalli, S., Song, X., Demmel, J., Keutzer, K., & Hsieh, C.-J. (2019). 深度学习的大批量优化：76 分钟训练 BERT。*ICLR2020*。arXiv:1904.00962。52. Beltagy, I., Peters, M. E., & Cohan, A. (2020). Longformer：长文档 Transformer。arXiv:2004.05150。53. Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., & Salakhutdinov, R. (2014). Dropout：一种防止神经网络过拟合的简单方法。*JMLR*，15(56)，1929–1958。54. Szegedy, C., Vanhoucke, V., Ioffe, S., Shlens, J., & Wojna, Z. (2016). 重新思考计算机视觉的 Inception 架构。*CVPR2016*。（引入了标签平滑。）55. Kaplan, J., McCandlish, S., Henighan, T., Brown, T. B., Chess, B., Child, R., Gray, S., Radford, A., Wu, J., & Amodei, D. (2020). 神经语言模型的缩放定律。arXiv:2001.08361。56. Hoffmann, J., Borgeaud, S., Mensch, A., Buchatskaya, E., Cai, T., Rutherford, E., et al. (2022). 训练计算最优的大语言模型。*NeurIPS2022*。arXiv:2203.15556。57. Hendrycks, D., Burns, C., Basart, S., Zou, A., Mazeika, M., Song, D., & Steinhardt, J. (2021). 大规模测量多任务语言理解能力。*ICLR 2021*。arXiv:2009.03300。58. Zellers, R., Holtzman, A., Bisk, Y., Farhadi, A., & Choi, Y. (2019). HellaSwag：机器真的能续写你的句子吗？*ACL 2019*。arXiv:1905.07830。59. Gemma 团队、Mesnard, T., Hardin, C., Dadashi, R., Bhupatiraju, S., Pathak, S., et al. (2024). Gemma：基于 Gemini 研究与技术的开放模型。arXiv:2403.08295。60. Gunasekar, S., Zhang, Y., Aneja, J., Mendes, C. C. T., Del Giorno, A., Gopi, S., et al. (2023). 你只需要教科书。arXiv:2306.11644。61. 零一万物、Young, A., Chen, B., Chen, C., Cheng, C., et al. (2024). Yi：零一万物推出的开放基础模型。arXiv:2403.04652。62. Goyal, P., Doll´ar, P., Girshick, R., Noordhuis, P., Wesolowski, L., Kyrola, A., Tulloch, A., Jia, Y., & He, K. (2017). 精确的大批次 SGD：1 小时训练 ImageNet。arXiv:1706.02677。63. Loshchilov, I., & Hutter, F. (2017). SGDR：带热重启的随机梯度下降。*ICLR2017*。arXiv:1608.03983。64. Touvron, H., Martin, L., Stone, K., Albert, P., Almahairi, A., et al. (2023). Llama 2：开放基础模型与微调聊天模型。arXiv:2307.09288。65. Penedo, G., Malartic, Q., Hesslow, D., Cojocaru, R., Cappelli, A., et al. (2023). 用于 Falcon 大语言模型的精炼版 Web 数据集。arXiv:2306.01116。66. Chiang, W.-L., Li, Z., Lin, Z., Sheng, Y., Wu, Z., Zhang, H., et al. (2023). Vicuna：一款开源聊天机器人，以 90%* 的 ChatGPT 质量惊艳亮相。LMSYS 组织。67. Xu, C., Sun, Q., Zheng, K., Geng, X., Zhao, P., Feng, J., Tao, C., & Jiang, D. (2023). WizardLM：赋能大语言模型遵循复杂指令。arXiv:2304.12244。68. He, K., Zhang, X., Ren, S., & Sun, J. (2016). 用于图像识别的深度残差学习。*CVPR 2016*。arXiv:1512.03385。69. Ioffe, S., & Szegedy, C. (2015). 批量归一化：通过减少内部协变量偏移加速深度网络训练。*ICML 2015*。arXiv:1502.03167。70. Kingma, D. P., & Ba, J. (2015). Adam：一种随机优化方法。*ICLR2015*。arXiv:1412.6980。71. Kudo, T., & Richardson, J. (2018). SentencePiece：一种简单且独立于语言的神经文本处理子词分词器与去分词器。*EMNLP2018*。arXiv:1808.06226。72. Sennrich, R., Haddow, B., & Birch, A. (2016). 使用子词单元处理低频词的神经机器翻译。*ACL 2016*。arXiv:1508.07909。73. Micikevicius, P., Narang, S., Alben, J., Diamos, G., Elsen, E., et al. (2018). 混合精度训练。*ICLR2018*。arXiv:1710.03740。74. Chen, T., Xu, B., Zhang, C., & Guestrin, C. (2016). 以内存开销亚线性增长的方式训练深度神经网络。arXiv:1604.06174。75. Rafailov, R., Sharma, A., Mitchell, E., Ermon, S., Manning, C. D., & Finn, C. (2023). 直接偏好优化：你的语言模型其实暗中充当奖励模型。*NeurIPS 2023*。arXiv:2305.18290。76. Schulman, J., Wolski, F., Dhariwal, P., Radford, A., & Klimov, O. (2017). 近端策略优化算法。arXiv:1707.06347。77. Hong, J., Lee, N., & Thorne, J. (2024). ORPO：无需参考模型的单体偏好优化。*EMNLP2024*。arXiv:2403.07691。78. Huang, Y., Cheng, Y., Chen, D., Lee, H., Ngiam, J., Le, Q. V., & Chen, Z. (2019). GPipe：使用流水线并行高效训练巨型神经网络。*NeurIPS2019*。arXiv:1811.06965。79. Mihaylov, T., Clark, P., Khot, T., & Sabharwal, A. (2018). 盔甲能导电吗？一个用于开放书本问答的新数据集。*EMNLP2018*。arXiv:1809.02789。80. Sutskever, I., Vinyals, O., & Le, Q. V. (2014). 使用神经网络进行序列到序列学习。*NeurIPS2014*。arXiv:1409.3215。81. Bahdanau, D., Cho, K., & Bengio, Y. (2015). 通过联合学习对齐与翻译实现神经机器翻译。*ICLR2015*。arXiv:1409.0473。82. Mikolov, T., Chen, K., Corrado, G., & Dean, J. (2013). 向量空间中词表示的高效估计。arXiv:1301.3781。83. Pennington, J., Socher, R., & Manning, C. D. (2014). GloVe：用于词表示的全局向量。*EMNLP2014*。84. Clark, K., Khandelwal, U., Levy, O., & Manning, C. D. (2019). BERT 关注什么？对 BERT 注意力机制的分析。*ACL 2019 BlackboxNLP* 研讨会。arXiv:1906.04341。85. Tenney, I., Das, D., & Pavlick, E. (2019). BERT 重新发现了经典自然语言处理流水线。*ACL 2019*。arXiv:1905.05950。86. Kitaev, N., Kaiser, L., & Levskaya, A. (2020). Reformer：高效的 Transformer。*ICLR2020*。arXiv:2001.04451。87. Choromanski, K., Likhosherstov, V., Dohan, D., Song, X., Gane, A., et al. (2021). 重新思考

使用 Performer 的注意力机制。*ICLR2021*。arXiv:2009.14794。88. Child, R., Gray, S., Radford, A., & Sutskever, I. (2019). 使用稀疏 Transformer 生成长序列。arXiv:1904.10509。89. Hinton, G., Vinyals, O., & Dean, J. (2015). 神经网络中的知识蒸馏。arXiv:1503.02531。90. Sanh, V., Debut, L., Chaumond, J., & Wolf, T. (2019). DistilBERT：BERT 的蒸馏版本，更小、更快、更便宜、更轻量。arXiv:1910.01108。91. Lee, K., Ippolito, D., Nystrom, A., Zhang, C., Eck, D., Callison-Burch, C., & Carlini, N.(2022). 训练数据去重可提升语言模型。*ACL 2022*。arXiv:2107.06499。92. Penedo, G., Kydlicek, H., allal, L. B., Lozhkov, A., Mitchell, M., et al. (2024). FineWeb 数据集：大规模筛选网络以获取优质文本数据。arXiv:2406.17557。93. Houlsby, N., Giurgiu, A., Jastrzebski, S., Morrone, B., de Laroussilhe, Q., et al. (2019). 面向自然语言处理的参数高效迁移学习。*ICML2019*。arXiv:1902.00751。94. Liu, X., Ji, K., Fu, Y., Tam, W., Du, Z., Yang, Z., & Tang, J. (2022). P-Tuning v2：提示调优在各种缩放因子和任务上均可媲美微调。*ACL 2022*。arXiv:2110.07602。95. Liu, H., Li, C., Wu, Q., & Lee, Y. J. (2023). 视觉指令调优。*NeurIPS2023*。arXiv:2304.08485。96. Alayrac, J.-B., Donahue, J., Luc, P., Miech, A., Barr, I., et al. (2022). Flamingo：面向少样本学习的视觉语言模型。*NeurIPS 2022*。arXiv:2204.14198。97. Chen, M., Tworek, J., Jun, H., Yuan, Q., de Oliveira Pinto, H. P., et al. (2021). 评估基于代码训练的大语言模型。arXiv:2107.03374。98. Roziere, B., Gehring, J., Gloeckle, F., Sootla, S., Gat, I., et al. (2023). Code Llama：面向代码的开放基础模型。arXiv:2308.12950。99. Bai, Y., Kadavath, S., Kundu, S., Askell, A., Kernion, J., et al. (2022). 宪法式人工智能：通过人工智能反馈实现无害性。arXiv:2212.08073。100. Perez, E., Huang, S., Song, F., Cai, T., Ring, R., et al. (2022). 使用语言模型对语言模型进行红队测试。*EMNLP 2022*。arXiv:2202.03286。101. Wei, J., Tay, Y., Bommasani, R., Raffel, C., Zoph, B., et al. (2022). 大语言模型的涌现能力。*TMLR2022*。arXiv:2206.07682。102. Wei, J., Wang, X., Schuurmans, D., Bosma, M., Ichter, B., et al. (2022). 思维链提示可激发大语言模型的推理能力。*NeurIPS 2022*。arXiv:2201.11903。103. Lewis, P., Perez, E., Piktus, A., Petroni, F., Karpukhin, V., et al. (2020). 面向知识密集型自然语言处理任务的检索增强生成。*NeurIPS2020*。arXiv:2005.11401。104. Borgeaud, S., Mensch, A., Hoffmann, J., Cai, T., Rutherford, E., et al. (2022). 通过从万亿级词元中检索来改进语言模型。*ICML2022*。arXiv:2112.04426。105. Liang, P., Bommasani, R., Lee, T., Tsipras, D., Soylu, D., et al. (2022). 语言模型的全面评估。arXiv:2211.09110。

补充材料

S1. 完整基准测试结果。“够用就好”的训练背后隐藏的代价。从业者看到自己的 A100 使用 Hugging Face Trainer 训练时达到每秒 8,000 个词元，可能会认为这代表着硬件利用率还算合理。毕竟，每秒 8,000 个词元听起来很快。但令人不安的事实是：一块性能可达 312 万亿次浮点运算的 A100 GPU，超过 92% 的时间都在等待——等待内存传输，等待内核启动，等待 Python 决定接下来要做什么。7.7% 的模型浮点运算利用率（MFU）意味着，每过一秒墙钟时间，只有 0.077 秒用于在 Tensor Cores 上进行实际的矩阵乘法。剩下的 0.923 秒都耗在开销上。这并不是在批评 Hugging Face，而是由可组合、模块化组件构建训练循环所必然带来的结果。可组合性是有代价的，而这代价体现在内存往返传输上。

理解指标揭示的内容。基准测试表中的每项指标都讲述了系统行为的一个具体方面。每秒令牌数衡量端到端吞吐量：在前向与反向传播中处理的词元数除以墙钟时间。该指标直接决定训练成本——每秒 token 数翻倍，云账单就减半。内存用量会限制批次大小和序列长度；一旦超出图形处理器内存，就必须减小批次，从而降低算术强度并进一步拉低模型浮点运算利用率。模型浮点运算利用率揭示了实际达到的理论峰值算力比例。对于 Transformer 训练，经验公式为：模型浮点运算利用率 = (6 × 参数 × 每秒 token 数) / (图形处理器万亿次浮点运算 $\times 10^{12}$)。一块 A100 在 5 亿参数模型上达到 40% 的模型浮点运算利用率时，每秒可处理约相当于 1040 万个参数的浮点运算次数——表现出色，但仍有 60% 的硅芯片处于闲置状态。

标准训练为何浪费 92% 的 GPU 周期。要了解开销从何而来，可以跟踪标准 PyTorch 中的单个训练步骤。前向传播将模型权重从高带宽内存加载到 L2缓存，再送入 Tensor Cores，计算激活值，然后将其存回高带宽内存，以供反向传播使用。交叉熵损失会生成形状为 $(B \times N \times V)$ 的 logit 张量——对于批次大小为 4、序列长度为 2048、词汇表大小为 151,936 的通义千问，这需要 $4 \times 2048 \times 151,936 \times 4$ 字节数 $= 4.7$ GB，仅用于 *logits*。该张量的存在只是为了计算一个标量损失，消耗的内存却超过整个模型。反向传播期间，我们再次读取该张量、计算梯度，然后将其丢弃。内存带宽开销惊人：一个只产生单个数值的张量，往返传输就耗费 9.4 GB。与此同时，在我们通过较早的层进行反向传播时，这 4.7 GB 中的大部分都处于闲置状态——PyTorch 的自动微分计算图会一直保留引用，直到反向传播完成，从而阻止提前释放。

Chronicals 洞见：攻克内存层次结构。我们相较于 Unsloth MAX 提升了 4.1 倍，相较于 Hugging Face 提升了 5.1 倍，这源于一个简单原则：数据在内存层次结构中应当恰好移动一次。Cut Cross-Entropy 内核无需将完整的 logit 张量实例化即可计算损失——我们将词汇表分成每块 4,096 个词元进行流式处理，同时维护一个运行中的对数和指数，从而以低 37 倍的峰值内存获得完全相同的数值结果。融合 AdamW 内核只读取每个参数、梯度和优化器状态一次，在寄存器中完成全部六项更新操作，然后仅写回一次——消除了五次内核启动，并将内存流量从 $6\times$ 降至 $1\times$。序列打包彻底消除了填充词元：我们不再将 40% 的计算浪费在对填充内容的注意力上，而是使用块对角因果掩码拼接序列，确保每个词元都参与学习。LoRA+差分学习率（B 矩阵比 A 矩阵高 16 倍）加快了每一步的收敛速度，这意味着我们只需更少的步数即可达到目标损失，从而将每步加速比叠加为更快的训练运行。

多项优化的叠加效应。一个关键洞见是，这些优化的效果是相乘而非相加。举例来说：如果内核融合带来 1.8 倍加速比，序列打包带来 2 倍加速比（消除了 50% 的填充），而 LoRA+将收敛速度提升 1.5 倍，那么综合效果就是 $1.8 \times 2.0 \times 1.5 = 5.4$ 倍——接近我们观察到的 5.1 倍。这种乘法式的叠加效应解释了为什么 Chronicals 尽管采用的是众所周知的优化技术，仍能实现大幅提升的吞吐量。创新之处不在于某个单独的内核，而在于将内存层次结构感知系统地应用于训练流水线的每个组件。

表 6. 不同配置下的完整基准测试结果

Mode	框架	批次	每秒 token 数	内存	MFU
全量微调	HuggingFace	4	8,000	24.2 GB	7.7%
	Unsloth	4	11,736	19.2 GB	11.3%
	Chronicals	4	28,450	16.8 GB	27.4%
	Chronicals	16	41,184	22.4 GB	39.6%
LoRA 秩=32	HuggingFace	4	2,100	12.1 GB	2.0%
	Unsloth MAX	4	2,857	8.4 GB	2.7%
	Chronicals LoRA+	4	11,699	7.2 GB	11.2%

具体示例：批次大小为 16 的全量微调。考虑 Chronicals 的全量微调配置，其吞吐量达到 41,184 个词元/秒。对于拥有 4.94亿个参数的 Qwen2.5-0.5B，每个训练步骤处理 $16 \times 2048 = 32,768$ 个词元。前向传播约需 $2 \times 494\text{M} \times 32,768 = 32.4$ TFLOPs （采用 $2 \times$ params $\times$ tokens 近似计算）。反向传播的梯度计算还需约 $4 \times 494\text{M} \times 32,768 = 64.8$ TFLOPs。以每秒 41,184 个词元的速度，我们每秒处理

约 1.26 个训练步骤，持续吞吐量达到 $97.2 \times 1.26 = 122.5$ TFLOPS。这相当于 A100 峰值性能（312 TFLOPS）的 39.6%——通过精细的内存管理和内核融合，达到了相当不错的模型浮点运算利用率。

为什么 LoRA 的绝对吞吐量较低。 乍看之下，LoRA 的结果似乎有悖直觉：可训练参数更少（仅 400 万，而全量微调需要 4.94 亿），每秒 token 数却更低。原因在于计算图结构。LoRA 仍需对冻结的基础模型执行完整的前向传播，但梯度计算量减少，导致流水线失衡，此时瓶颈变成内存带宽，而非计算能力。每次前向传播仍必须从高带宽内存中加载基础模型权重；此外，LoRA 训练通常采用更小的批次大小（其他框架受内存限制所致），这也会进一步降低算术强度。Chronicals 通过积极进行内核融合并采用 LoRA+ 学习率调度来解决这一问题，实现了相较于 Unsloth MAX 4.1 倍的提升。

S2. 数学推导。

S2.1 在线 ***Softmax*** 正确性。**为何这很重要。** Softmax 函数是每个 Transformer 的核心——它将注意力分数转换为概率，并将 logits 转换为损失梯度。朴素实现需要对数据进行两次完整遍历：第一次找出最大值（以确保数值稳定性），第二次计算归一化后的指数函数值。对于通义千问的 151,936 词词汇表，这意味着每个批次元素中的每个词元都要从高带宽内存加载 600 KB 的 logits，且要加载两次。批次大小为 8,192 个词元、内存带宽为 2 TB/s 时，仅 softmax 每个训练步骤就会耗费 5 毫秒——超过总步骤时间的 15%。在线 softmax 完全省去了第二次遍历，只需一次流式遍历即可计算出精确结果。

标准 Softmax 面临的挑战。 Softmax 函数 $\text{softmax}(x)_i = \exp(x_i)/\sum_j \exp(x_j)$ 看起来简单得令人难以置信，但其朴素实现却造成了根本性的内存瓶颈。要计算任意一个输出元素，我们都需要对所有输入元素求和。这似乎需要对数据进行两次遍历：先计算 $\sum_j \exp(x_j)$，再计算每个输出。对于大小为 151,936 的词汇表，这意味着要从高带宽内存中加载整个 Logit 向量两次；而在一个批次中对每个词元都重复这一操作，内存成本将高得难以承受。

关键洞见：可校正的运行总和。 在线 softmax 的突破在于发现，当遇到新的最大值时，我们可以校正运行总和。要理解其原理，可以想象你正在计算运行均值，却突然发现之前所有的值都应该按不同的比例缩放。诀窍在于指数函数有一个优美的性质：$\exp(x - m_1) = \exp(x - m_2) \cdot \exp(m_2 - m_1)$。这意味着我们可以通过乘以一个校正因子，追溯性地“重新缩放”所有先前的值。

直观地说：假设我们已经处理了元素 m_{i-1} $m_{i-1} =$ $x_1 \ldots x_{i-1}$ $x_1 \ldots x_{i-1}$ x_i $x_j -$ 、、并维护了运行总和 $d_{i-1} = \sum_{j=1}^{i-1} \exp(\,)$，其中 max(、、)。现在我们看到元素，它可能大于当前最大值。

如果 $x_i > m_{i-1}$，我们的运行总和就是“错误的”——它使用了 m_{i-1} 作为被减常数，但本应使用 $m_i = x_i$。修正方法很巧妙：将旧总和乘以 $\exp(m_{i-1} - m_i)$。这样会对之前所有的指数函数值进行重新缩放，效果就像从一开始使用新的最大值一样。这不是近似，而是精确的算术运算，利用了指数函数值的乘法结构：

$$\exp(x_j - m_{i-1}) \cdot \exp(m_{i-1} - m_i) = \exp(x_j - m_{i-1} + m_{i-1} - m_i) = \exp(x_j - m_i) \tag{88}$$

定理： 在线 softmax 保持不变量 $d_i = \sum_{j=1}^{i} \exp(x_j - m_i)$。**P**

roof by induction:

基础情形 ($i = 1$)： $d_1 = \exp(x_1 - m_1) = \exp(x_1 - x_1)$

$= 1$归纳步骤： 假设 $d_{i-1} = \sum_{j=1}^{i-1} \exp(x_j - m_{i-1})$

$$d_i = d_{i-1} \cdot \exp(m_{i-1} - m_i) + \exp(x_i - m_i) \tag{89}$$

$$= \sum_{j=1}^{i-1} \exp(x_j - m_{i-1}) \cdot \exp(m_{i-1} - m_i) + \exp(x_i - m_i) \tag{90}$$

$$= \sum_{j=1}^{i-1} \exp(x_j - m_i) + \exp(x_i - m_i) \tag{91}$$

$$= \sum_{j=1}^{i} \exp(x_j - m_i) \quad \blacksquare \tag{92}$$

这对 Cut Cross-Entropy 意味着什么。 在线 Softmax 支持我们的分块交叉熵计算。我们不再同时实例化全部 151,936 个 logits，而是将词汇表划分为大小为 4,096 的分块进行处理。每个分块都会更新运行最大值和分母；当目标 logit 位于当前分块中时，我们会将其提取出来。最终损失的计算方式为 $\text{loss} = \log(d) + m -$ 目标 logit，等于 $\log \sum_j \exp(x_j) - x_{\text{target}}$——这就是标准交叉熵，但峰值内存减少了 37 倍。

在 Triton 中的实现。 在我们的内核中，每个线程块处理一个序列位置。在线 softmax 状态（运行中的最大值 m 和分母 d）存放在寄存器中，而不是共享内存或高带宽内存中。权重矩阵分块以

分块的形式加载，与缓存的隐藏状态相乘以生成 logit 块，并立即由在线 softmax 更新使用。这种流式模式实现了接近最优的内存带宽利用率。

***S2.2 FlashAttention IO 复杂度。*现代 GPU 计算的核心问题。**这里有一个可能会让你惊讶的数字：A100 GPU 从内存中加载一个浮点数所需的时间里，可以执行 156 次浮点运算。换句话说，图形处理器的计算单元每秒可以执行 312 万亿次操作，但内存每秒只能提供 2 万亿字节。这个 156:1 的比例揭示了 GPU 编程面临的根本挑战——为算术单元持续提供数据。任何每次操作读取超过 6 字节（156 次操作 / 每个浮点数 4 字节 ≈ 6 字节）的算法都会受到内存限制：瓶颈在于数据加载速度，而非数据处理速度。

为什么标准注意力会严重受内存限制。标准注意力计算 Attention(Q，K，V) =Softmax($QK^T/\sqrt{d}$)V。对于序列长度 $N = 2048$ 和头维度 $d = 64$，需要：（1）计算 QK^T，即一个包含 $N \times N = 4$ 百万个元素的矩阵；（2）将该矩阵存入内存；（3）逐行应用 Softmax；（4）计算最终的矩阵-向量乘积。仅注意力矩阵每个头就占用 $N^2 \times 4 = 16$ MB 内存，但我们只执行$O(N^2d) = 268$ 百万次浮点运算。算术强度为每字节 $268 \times 10^6/(16 \times 10^6) = 16$ 次浮点运算，远低于充分利用计算能力所需的 156。图形处理器有 90% 的时间都花在将注意力矩阵移入和移出内存上。

FlashAttention 策略：为静态随机存取存储器分块。FlashAttention 重新组织计算，以最大限度地在图形处理器的高速静态随机存取存储器（共享内存）中复用数据。其关键洞见——确实非常巧妙——是注意力可以按块计算，每个块都能完全放入静态随机存取存储器，而无需实际生成完整的注意力矩阵。在线 softmax 技巧使这成为可能：我们可以计算部分注意力分数，逐步对其归一化并累加输出，整个过程中无需将中间结果存储到高带宽内存。处理完一个 Q 分块与所有 K、V 分块的计算后，再转向下一个 Q 分块，这样便能将加载 K 和 V 的开销分摊到多个 Q 行上。

定理：块大小为 B 的 FlashAttention 需要 $O(N^2d^2/M)$ 次 HBM 访问。

Pro：考虑分块计算模式。我们将 Q、K、V 划分为块，每块大小为 $B \times d$ ：

- Q 块数量：N/B
- KV 块数量：N/B
- 每个 Q 块加载 N/B 次（每个其参与注意力计算的 KV 块加载一次）
- 每个 KV 块会加载 N/B 次（每个关注它的 Q 块加载一次）

Q、K、V 的 HBM 访问总量：

$$\text{IO} = \frac{N}{B} \cdot \frac{N}{B} \cdot Bd = \frac{N^2 d}{B} \tag{93}$$

块大小 B 受 SRAM 容量 M限制。我们需要在 SRAM 中容纳一个 Q 块（Bd 个元素）、一个 KV 块对（$2Bd$个元素）以及中间结果（B^2 个用于存储注意力分数）：

$$3Bd + B^2 \leq M \implies B = O\left(\sqrt{M/d}\right) \tag{94}$$

采用最优 $B = O(\sqrt{M/d})$ 时：

$$\text{IO} = O\left(\frac{N^2 d}{\sqrt{M/d}}\right) = O\left(\frac{N^2 d^2}{M}\right) \quad \blacksquare \tag{95}$$

A100 的具体数值。A100 的每个 SM 拥有 192 KB 共享内存。对于使用 BF16（每个元素 2 字节数）的头维度 $d = 64$ ，我们可以容纳大小为 $B \approx \sqrt{192 \times 1024}/(2 \times 64)$ 的块 ≈ 39。为提高效率，FlashAttention 会将其向上取整至 $B = 64$，每个 Q/K/V 块需要 $64 \times 64 \times 2 = 8$ KB，注意力分数块需要 $64 \times 64 \times 4 = 16$ KB。

实际应用中的 IO 减少。对于序列长度 $N = 2048$ 和头维度 $d = 64$，标准注意力每个头需要 $O(N^2) = 4$ 百万次 HBM 访问。采用 $B = 64$ 的 FlashAttention 需要 $O(N^2d/B) = (2048)^2 \times 64/64 = 4$ 百万次访问——但关键在于，其中大多数是写入输出，而不是读取；并且中间注意力矩阵（每个头 16 MB）不会被写入 HBM。减少内存访问是其主要优势，能够支持更长的序列和更大的批次。

***S2.3 LoRA+ 学习率比。*为什么 LoRA 训练比预期更慢。**使用标准 LoRA 进行微调的从业者常常会注意到，收敛所需的步数比预期多 2～3 倍。模型起初稳步提升，但在达到最优性能之前很久就陷入平台期。我们提出的假设——并且实验也证实了这一点——是，这种减速源于梯度通过低秩分解时所产生的根本性不对称。理解这种不对称性，就能揭示为什么差分学习率能够带来如此显著的加速比。

标准 LoRA 中的非对称问题。低秩适配将权重更新参数化为 $\Delta W = BA$，其中$B \in \mathbb{R}^{d \times r}$和 $A \in \mathbb{R}^{r \times k}$。标准初始化将 $B = 0$和 $A \sim \mathcal{N}(0, \sigma^2)$，确保在开始时 $\Delta W = 0$ （模型

从预训练基数开始）。这种初始化选择似乎合理——我们希望从预训练模型开始——但它造成了一个微妙而重大的训练非对称性，使从业者损失了 50% 甚至更多的训练预算。

梯度流解析。为了理解其中的原因，我们来详细追踪第一个训练步骤。设 $E = \partial \mathcal{L}/\partial W$ 为沿基数权重反向传播的误差信号。根据链式法则，A 和 B 的梯度为：

$$\frac{\partial \mathcal{L}}{\partial B} = EA^T \neq 0 \tag{96}$$

$$\frac{\partial \mathcal{L}}{\partial A} = B^T E = 0 \tag{97}$$

注意这种不对称性：B 会接收到非零梯度（因为 A 是随机初始化的且非零），但 A接收到的是零梯度（因为 $B = 0$ 和 $0^T \cdot$任何东西 $= 0$）。在第一个训练步骤中，只有 B 会更新！矩阵 A 仍保持其随机初始化状态，对学习没有任何贡献。这意味着一半的可训练参数在第一个步骤中实际上都被浪费了。

级联效应。步骤 1之后，$B_1 = -\eta_B \cdot EA^T$。现在 A 可以接收到梯度：$\partial \mathcal{L}/\partial A = B_1^T E$。但幅度很小——它与 η_B成正比，而 B是学习率。经过 t 个步骤后：

$$\|B_t\| \approx O(\eta_B t \cdot \|E\| \cdot \|A\|) \tag{98}$$

A的梯度幅值增长缓慢，而 B 持续接收直接的误差信号。这会形成一个不平衡的优化景观，其中 B 的收敛速度远快于 A。

LoRA+ 解决方案：差分学习率。为了实现平衡更新，我们希望两个矩阵对权重变化的贡献相等：

$$\eta_B \left\| \frac{\partial \mathcal{L}}{\partial B} \right\| \approx \eta_A \left\| \frac{\partial \mathcal{L}}{\partial A} \right\| \tag{99}$$

缩放分析表明，梯度范数会随隐藏维度以不同方式缩放。对于输入维度为k 、输出维度为 d的层：

$$\left\| \frac{\partial \mathcal{L}}{\partial B} \right\| \propto \|E\| \cdot \|A\| \propto \sqrt{d \cdot k} \tag{100}$$

$$\left\| \frac{\partial \mathcal{L}}{\partial A} \right\| \propto \|B\| \cdot \|E\| \propto \eta_B t \cdot d \tag{101}$$

为了使梯度具有可比的幅度：$\eta_B/\eta_A = O(\sqrt{k/d})$。在实践中，Hayou 等人通过实验证实，$\eta_B/\eta_A \approx 16$ 在不同模型规模下都表现良好，这与典型 Transformer 维度下的 $\sqrt{k/d} \approx 16$ 一致。

在 Chronicals 中实现。我们通过为优化器分配不同的参数组来实现 LoRA+：

```
优化器 = AdamW([{"params": lora_A_params, "lr": base_lr}, {"params": lora_B_params, "lr": base_lr * 16},
])
```

与标准 LoRA 相比，这一简单改动可使收敛速度提高 1.5～2 倍，且不会增加任何额外的内存或计算成本。

***S2.4 Kahan* 求和误差界。浮点求和问题**。对大量浮点数求和时，舍入误差会不断累积。IEEE 754 算术中的每次加法操作都会将结果舍入到最接近的可表示数，从而引入至多为 $\epsilon \cdot |a+b|$ 的误差，其中 ϵ是机器精度（FP32 为 $\epsilon \approx 10^{-7}$，BF16 为 $\approx 10^{-3}$）。进行 n 次加法时，朴素求和累积的误差可能与 $n\epsilon$成正比。使用 BF16 对 8 个梯度微批次求和时，相对误差最高可达 0.8%，足以导致训练不稳定。

Kahan 补偿技巧。Kahan 求和会维护一个“补偿”变量 c，用于追踪每次加法中丢失的低位比特。算法步骤如下：

1. 将 x_i 加到运行总和 s之前，先减去累积误差：$y = x_i - c$
2. 将 y加到总和中：$t = s + y$
3. 计算新的补偿金额：$c = (t - s) - y$
4. 更新总和：$s = t$

关键洞见在于，$(t-s)-y$恰好捕捉到了加法 $s+y=t$ 中丢失的内容。如果没有发生舍入，这个值将为零。但由于进行了舍入，它等于被丢弃的位。
定理： Kahan 求和的总误差为 $O(\epsilon)$，适用于 n 次加法。
证明概　要： 补偿项 c会追踪每次加法中丢失的低位比特。经过 n 次加法后　　：

$$|s_n - \sum_{i=1}^{n} x_i| \leq (2\epsilon + O(\epsilon^2)) \sum_{i=1}^{n} |x_i| \tag{102}$$

而朴素求和的误差为 $O(n\epsilon)$。误差与 n 无关，因为每一步的误差都会在下一步中得到补偿。 ■

梯度累积应用。 在 Chronicals 中，我们使用 Kahan 求和，以 BF16 在多个微批次间累积梯度。对于 8 个梯度累积步数，朴素求和可能引入 0.8% 的误差；Kahan 求和则可将误差控制在 0.2% 以下。此功能通过 Triton 内核实现，该内核同时维护累积梯度及其补偿项，每个参数仅增加 4 字节的开销。

S2.5 BFD 近似界。**序列打包问题**。 训练数据集包含长度各异的序列。将所有序列朴素地填充到最大长度会浪费大量计算：如果序列平均长度为 500 个词元，而最大长度为 2048 个词元，那么 75% 的计算都会浪费在填充词元上。序列打包通过将多个序列拼接成一个训练样本，并用注意力掩码分隔它们来解决这一问题。

作为装箱问题进行打包。 将可变长度序列装入固定容量的“箱”（最大上下文长度）的问题，正是经典装箱问题。给定长度为 $\ell_1,\ldots,\ell_n$的序列和箱容量 C，我们的目标是最小化所使用的箱数。这是 NP 困难问题，但存在非常优秀的近似算法。

为什么采用最佳适应递减算法（BFD）？ BFD 按长度对序列进行排序（最长的优先），然后将每个序列放入仍能容纳它且剩余容量最小的箱中。“递减”顺序至关重要：它确保较大的序列在选择箱时拥有最大灵活性，并优先被放置。较小的序列随后填补空隙。

定理： 最佳适应递减算法实现 $\text{BFD}(I) \leq \frac{11}{9}\text{OPT}(I) + \frac{6}{9}$。
证明概要： Johnson（1973年）对 FFD 的分析可以扩展到 BFD。关键观察是，按大小递减排序后，大于箱容量 $1/2$ 的物品必须各自占用一个箱（任意两个都无法共享）。大小介于 $1/3$ 和 $1/2$ 之间的物品，最多可以与另一个同类物品共享。$11/9 \approx 1.22$ 近似比来源于对剩余小物品最坏情况下打包方式的分析。在实践中，对于典型的序列长度分布，BFD 能够实现接近最优的打包。 ■

具体示例。 设有 100 条序列，其长度均匀分布在 100 到 500 个词元之间，并打包到容量为 512 的箱中。朴素批处理需要 $100 \times 512 = 51$，总计 200 个词元，但实际内容只有 30,000 个词元（浪费率为 58%）。BFD 打包大约使用 65 个箱，总计 33,280 个词元，使总计算量减少 35%，且不会损失任何训练信号。

S3. 算法伪代码。

S3.1 完整融合式 ***AdamW Triton*** 内核。**模块化代码的隐性代价**。 每次 CUDA 内核启动都会产生 5–20 微秒的开销：中央处理器必须串行处理启动参数，CUDA 驱动程序必须调度内核，而图形处理器必须同步其命令队列。这种开销看似微不足道，直到你意识到，在 PyTorch 中执行一次 AdamW 优化器步骤需要运行六个独立内核：全局范数计算、梯度缩放、权重衰减应用、一阶矩指数移动平均、二阶矩指数移动平均，以及最后的参数更新。对于分布在 1,000+ 个权重张量中的 4.94 亿个参数，这意味着每个优化器步骤要启动 6,000+ 个内核，耗时 30–120 毫秒。在一次前向-反向传播耗时 150 毫秒的训练过程中，我们有 20–40% 的墙钟时间仅仅花在启动优化器内核上——在任何有用的计算开始之前。

为何融合能带来 1.8 倍加速比。 我们的 Triton 内核将全部六项操作合并到一个 GPU 内核中。关键在于，优化器步骤本质上受内存带宽限制：更新 4.94 亿个参数需要读取 4.94 亿个浮点数的参数、4.94 亿个浮点数的梯度、4.94 亿个浮点数的一阶矩，以及 4.94 亿个浮点数的二阶矩；随后还要写回 4.94 亿个浮点数的更新后参数、4.94 亿个浮点数的新一阶矩，以及 4.94 亿个浮点数的新二阶矩。这会产生 27.7 GB 的内存流量，但算术运算仅需数亿次浮点运算。在 A100 的 2 TB/s 带宽下，这应该能在 14 毫秒内完成。未融合的 PyTorch 需要 25+ 毫秒，因为每个内核都会从 HBM 重新加载相同的数据。我们的融合内核只加载每个张量一次，在寄存器中完成所有算术运算，并且只写回一次。
逐步解析内核。内核的运行方式如下：首先，每个线程块使用程序 ID 和块大小计算其负责的参数切片（第 7–9 行）。掩码用于处理边界情况：最后一个块的元素数量可能少于块大小。梯度裁剪会应用一个预先计算的系数（该系数由单独的归约内核计算，每个步骤运行一次）。权重衰减遵循 AdamW 公式——衰减会在梯度更新之前应用，而不是加到梯度上（第 17–18 行）。这一区别很重要：AdamW 将权重衰减与自适应学习率解耦，避免衰减被 $\sqrt{v}$ 除。如果我们改为采用 L2 正则化（将 $\lambda\theta$ 加到梯度上），那么

对于梯度方差较大的参数，正则化强度会被缩小——这恰恰与我们防止过拟合的目标相反。动量更新和偏差校正参数步长会原地执行，所有中间值都保留在寄存器中。

算法 18 融合 AdamW Triton 内核（完整）

```
@triton.jit
def 融合_AdamW_内核(
    参数_ptr, 梯度_ptr, m_ptr, v_ptr,
    学习率, beta1, beta2, eps, 权重衰减,
    裁剪系数, 偏差修正1, 偏差修正2,
    N, 块大小: tl.constexpr):
    程序_ID = tl.program_id(0)
    偏移量 = 程序_ID * 块大小 + 范围生成函数(0, 块大小)
    掩码 = 偏移量 < N

    # 加载张量（每个张量只读取一次高带宽内存）
    参数 = 加载函数(params_ptr + 偏移量, 掩码=掩码)
    梯度 = tl.load(梯度指针 + 偏移量, 掩码=掩码)
    m = tl.load(m_指针 + 偏移量, 掩码=掩码)
    v = tl.load(v_指针 + 偏移量, 掩码=掩码)

    # 梯度裁剪（裁剪系数 = min(1, 最大范数/全局范数)）
    梯度 = 梯度 * 裁剪系数

    # 权重衰减（AdamW 风格——与梯度解耦）
    参数 = 参数 * (1.0 - 学习率 * 权重衰减)

    # 更新矩（指数移动平均）
    m = beta1 * m + (1.0 - beta1) * 梯度
    v = beta2 * v + (1.0 - beta2) * 梯度 * 梯度

    # 偏差校正估计（补偿零初始化）
    m_hat = m / 偏差修正1
    v_hat = v / 偏差修正2

    # 参数更新（自适应学习率）
    分母 = tl.sqrt(v_hat) + eps
    参数 = 参数 - lr * (m_hat / 分母)

    # 存储结果（每个张量仅写入一次高带宽内存）
    tl.store(params_ptr + offs, 参数, 掩码=掩码)
    tl.store(m_ptr + 偏移量, m, 掩码=掩码)
    tl.store(v_ptr + offs, v, mask=掩码)
```

性能分析。对于 4.94亿个参数和 BLOCK SIZE=1024，内核启动了 482,422 个线程块。每个块加载 4 个张量 $\times$ 1024 个元素 $\times$ 4 字节 = 16 KB，并写入 3 个张量 $\times$ 1024 $\times$ 4 = 12 KB。总内存流量为 13.5 GB。在 A100 的 2 TB/s 带宽下，这一过程耗时 6.75 ms——相比未融合的 PyTorch，性能提升了 1.8 倍。

S3.2 完整的 ***CCE*** 前向内核。**深度学习中最浪费的张量**。考虑一下在标准训练流水线中计算交叉熵损失时会发生什么。语言模型的最后一层会将每个词元的隐藏状态（对于 Qwen2.5-0.5B，为 896 维）投影到词汇表 logits（151,936 维）。对于长度为 2048 的 8 个序列组成的批次，这会创建一个形状为 $(8 \times 2048 \times 151{,}936)$ 的张量——仅 logits 就占用 9.4 GB 内存。但荒谬之处在于：我们创建这个 9.4 GB 的张量，只是为了计算一个标量（损失）。我们立即应用 Softmax，提取目标词元的概率，取对数，然后对所有位置求平均。这个张量本身之后再也不需要了。然而，标准实现仍会将该张量实体化，将其存储到高带宽内存中，然后为 Softmax 再读回来——产生 19 GB 的往返内存流量，只为得到一个数字。

Cut Cross-Entropy 洞见：你只需要两个数字。交叉熵损失 $\mathcal{L} = -\log P(y_{\text{target}}) = \log \sum_j \exp(z_j) - z_{\text{target}}$ 只需要从 151,936 维的 Logit 向量中获取两项信息：对数和指数（一个标量）和目标 logit （另一个标量）。这两者都不需要将完整张量实体化。我们的洞见是，可以逐步计算这两个数字，在词汇表中按分块流式处理，任何时候分配的缓冲区都不超过 4,096-

个元素。在线 softmax 算法（S2.1 节）使这一点成为可能：我们维护一个运行中的对数和指数，并在遇到新的最大值时对其进行修正。

算法解析。每个线程块处理一个序列位置（第 7–8 行）。该位置的隐藏状态（$H = 896$ 个元素）只加载一次，并缓存在寄存器中（第 22 行）——这一点至关重要，因为在遍历词汇表时，我们会重复使用它 37 次。随后，我们以分块大小 CHUNK SIZE（通常为 4096）遍历词汇表。对于每个分块：

1. 通过分块矩阵乘法计算 $h \cdot W^T$分块（第 29–34 行）。权重矩阵以 64 个元素为单位加载到分块中，以便容纳在寄存器内。这样无需将 logits 存储到高带宽内存中，即可计算出 4,096 个 logits。

2. 更新在线 softmax 状态：针对新的最大值调整运行总和，然后加上当前分块的指数函数值（第 37–41 行）。校正因子 $\exp(m_{\text{old}} - m_{\text{new}})$ 可确保数学上的精确性。

3. 如果目标词元位于该分块中，则提取其 logit（第 44–45 行）。我们会在每个分块中检查此条件，但它恰好只会触发一次。

处理完全部 37 个分块后，损失就是 $\log(d) + m -$目标 logit——与标准公式完全相同，但所需内存减少了 31 倍。

算法 19 截断交叉熵前向内核（完整）

1: @triton.jit 2: def cce 前向内核(3: hidden ptr, weight ptr, target ptr, loss ptr, 4: B, N, H, V, 分块大小: tl.constexpr): 5: 6: 行索引 ← tl.program id(0) 7: 目标 ←tl.加载函数(target ptr + 行索引) 8: 9: # 跳过填充词元（忽略索引 = -100） 10: if 目标 == -100: 11: tl.store(loss ptr + 行索引, 0.0) 12: return 13: 14: # 初始化在线 softmax：m=运行最大值，d=运行总和 15: m ←float('-inf') 16: d ← 0.0 17: 目标 logit ← 0.0 18: 19: # 仅加载一次隐藏状态，并缓存到寄存器中 20: h offs ← 范围生成函数(0, H) 21: h ← tl.加载函数(hidden ptr + 行索引 * H + h offs) 22: 23: # 按分块遍历词汇表 24: for 分块 start in 范围(0, V, 分块大小): 25: v offs ←分块 start + 范围生成函数(0, 分块大小) 26: v mask ←v offs < V 27: 28: # 分块矩阵乘法：logits = h @ W[分块].T 29: logits ← zeros(分块大小) 30: for k in 范围(0, H, 64): 31: h 块 ←h[k:k+64] 32: w 块 ← tl.加载函数(weight ptr + v offs[:, None] * H +k + 范围生成函数(0, 64)) 33: logits ← logits + tl.sum(h 块 * w 块, axis=1) 34: 35: # 在线 softmax：针对新的最大值校正运行总和 36: 分块最大值 ← tl.max(tl.where(v mask, logits, float('-inf'))) 37: m new ← tl.maximum(m, 分块最大值) 38: d ← d * tl.exp(m - m new) 39: d ← d + tl.sum(tl.where(v mask, tl.exp(logits - m new), 0.0)) 40: m ←m new 41: 42: # 在范围内提取目标 logit 43: if 分块 start ≤ 目标 < 分块 start + 分块大小: 44: 目标 logit ←logits[目标 - 分块 start] 45: 46: # 最终损失：-log(softmax[目标]) = 对数和指数 - 目标 logit 47: lse ← tl.log(d) + m 48: loss ← lse - 目标 logit 49: tl.store(loss ptr + 行索引, loss)

内存分析。 对于 $B = 8$、 $N = 2048$、 $H = 896$、 $V = 151,936$，分块大小=4096：

- 隐藏状态已加载： $8 \times 2048 \times 896 \times 2 = 28\,\text{MB}$（每个词元一次）
- 权重块已加载：151，$936/4096 = 37$ 个分块，每个分块 $\times\ 4096 \times 896 \times 2 = 273\,\text{MB}$
- 峰值内存： $28 + 273 = 301$ MB，而完整 logits 需要 9.4 GB（减少 31 倍）

隐藏状态保存在寄存器中；我们仅遍历权重矩阵一次。整个过程中不会实例化任何中间张量。

S4. 实现细节。

S4.1 融合 ***AdamW Triton*** 内核。**设计理念：带宽至上。** 设计用于优化器步骤的 GPU 内核时，试图优化计算的直觉其实是个陷阱。来看一下算术运算：对于 4.94亿个参数，AdamW 对每个参数大约执行 15 次操作（梯度裁剪乘法、权重衰减乘加、两个动量指数移动平均、偏差修正除法、平方根、最后的除法和减法）。总计 74 亿次浮点运算——在 A100 的 312 万亿次浮点运算/秒下，用时不到 0.1 毫秒。但同一个内核必须加载 4.94 亿个参数、梯度、一阶矩和二阶矩浮点数（FP32 下为 7.9 GB ），然后写回 4.94 亿个更新后的参数和动量浮点数（ 5.9 GB ）。以 2 TB/s 的带宽计算，这至少需要 6.9 毫秒 —— 比计算耗时长 70 倍。因此，我们的内核设计完全不考虑计算优化，而是专注于通过融合最大限度地减少内存流量。

块大小选择：为什么是 1024? 经过对 256、512、1024、2048 和 4096 进行基准测试后，我们选择了 = 1024 个元素作为块大小。需要权衡以下因素：(1) 内存合并访问：GPU 按 128 字节缓存行加载内存，因此每个线程束至少需要 32 个 FP32 元素才能实现完全合并访问——块大小大于 128 即可满足要求。(2) 寄存器压力：每个线程都必须保存 4 个输入张量和 3 个输出张量的部分数据；较大的块需要更多寄存器，可能会降低占用率。(3) 启动开销：较小的块需要更多线程块，从而增加调度器开销。块大小为 1024 个元素时，每个线程块可连续处理 4 KB 数据，在 A100 上达到 50%+ 占用率，并且处理 4.94 亿个参数只需 483 个块。基准测试表明，对于参数量从 1 亿到 70 亿的情况，1024 的表现与最优解相差不超过 2%。

关键实现选择：

- 块大小：每个线程块处理 1024 个元素（4 KB，与 L2 缓存行大小相符）
- 在中央处理器上计算偏差修正（Python 整数，可避免 GPU 同步）——这一点很重要，因为查询 GPU 的步数计数器会强制触发一个同步点
- 梯度裁剪系数以 GPU 张量的形式存储（来自单独的归约）——我们在一个归约内核中计算全局范数，然后将该系数传递给所有块
- 所有 6 项操作都使用单个内核（每个步骤可消除 5 次启动开销）

S4.2 序列打包。**填充带来的隐性代价**。使用标准填充方式在 Alpaca-52k 数据集上训练时，你浪费的计算资源远超想象。该数据集包含长度从 20 个词元（“什么是 2+2? ”）到 2,048 个词元（复杂的多轮对话）不等的序列。采用最大长度填充时，每个 20 词元的序列消耗的计算量都与 2,048 词元的序列相同，而模型对 2,028 个填充词元进行注意力计算，却学不到任何东西。我们测量发现：对于最大长度为 512 的 Alpaca-52k，朴素填充会导致 42% 的词元属于填充词元。这意味着 42% 的注意力浮点运算次数、42% 的前馈网络浮点运算次数，以及 42% 的交叉熵计算都无法产生学习信号。你为无助于模型质量的计算付出了代价。

打包方案：为什么选择最佳适应递减算法? 序列打包会将多个序列拼接成一个训练样本，并使用注意力掩码来防止跨序列注意力。难点在于确定哪些序列应该打包到一起。我们采用最佳适应递减算法（BFD），其打包效率可达 95-98%，而更简单的算法为 85-90%。BFD 的关键思路是：按序列长度排序（从长到短），然后将每个序列放入仍能容纳它且剩余容量最小的箱中。这种“最佳适应”步骤会为后续需要较大空间的序列留下更大的空隙，同时让较短的序列填补细小的空隙，避免浪费。

最佳适应递减算法：

1. 按序列　　长度降序排列——优先处理较长的序列，使其优先获得放入箱中的位置
2. 对每个序列，找到剩余容量最小且能容纳该序列长度的箱 $\geq$ （使用最小堆进行 $O(\log m)$ 查找）
3. 如果现有箱都无法容纳该序列，则新建一个箱

时间复杂度：排序为 $O(n \log n)$，箱选择为 $+ O(n \log m)$，其中 m 是箱的数量（通常为 $\ll n$）。对于 Alpaca-52k，在单个 CPU 核心上不到 2 秒即可完成。

注意力掩码处理：防止交叉污染。 打包序列之间不得相互注意——回答“法国的首都是哪里？”时，不应注意到“解释量子计算”的词元。我们通过块对角因果掩码来实现这一点。FlashAttention 的 cu seqlens（累积序列长度）接口可以高效处理：我们无需构造二维掩码，只需传入边界，由 FlashAttention 在内部确保隔离。位置 ID 会在每个序列边界处重置，以便正确计算 RoPE 嵌入——每个打包序列中位置 0 对应的是位置 0 的 RoPE，而不是打包后的偏移位置。

S4.3 交叉熵分块。**分块大小选择：金发姑娘问题**。 为 Cut Cross-Entropy 选择合适的分块大小，需要在四个相互制衡的约束之间取得平衡。(1) 内存：每个包含 4,096 个词汇表条目的分块需要$4096 \times 896 \times 2 = 7.3\,\text{MB}$ 来加载权重矩阵切片，此外还需要 16 KB 的 logit 缓冲区。分块越大，峰值内存越高。(2) 循环开销：分块大小为 4,096 时，我们需要遍历词汇表 151，$936/4096 = 37$ 次。分块越小，迭代次数越多，每次都会产生 Python 和 Triton 的调度开销。(3) 共享内存：在 A100 上，每个流式多处理器有 192 KB 共享内存。大于 8,192 的分块无法同时容纳权重切片和隐藏状态缓存。(4) 寄存器压力：在线 softmax 状态（运行最大值、运行总和、目标 logit）会占用寄存器；并发操作越多，需要的寄存器就越多。

对分块大小从 1,024 到 8,192 进行基准测试后，我们发现 4,096 是 Qwen2.5-0.5B 的最优分块大小。此选择具有以下优点：

- 峰值内存： $B \times N \times 4096 \times 4 = 128\,\text{MB}$（完整 logits 需要 4.7 GB）——减少了 37 倍
- 循环迭代次数：每个序列位置需迭代 37 次（每次迭代耗时 0.1 μ 秒，开销可以接受）
- 数值稳定性：采用带运行最大值的在线对数求和指数，即使 logit 值极端也能防止上溢
- 前向—反向对称性：反向传播会按相同的分块重新计算前向传播，复用相同的分块逻辑

梯度计算：优雅的反向传播。 交叉熵梯度的形式异常简单： $\partial\mathcal{L}/\partial z_i = \text{softmax}(z)_i - \mathbf{1}_{i=\text{target}}$。这其实就是“预测概率减去目标类别对应的 1，再减去其他所有类别对应的 0”。我们的反向内核会利用前向传播中缓存的对数和指数，逐分块重新计算 softmax 概率，减去指示值，并将梯度直接写入隐藏状态梯度缓冲区。反向传播实际上比前向传播更快，因为我们可以跳过提取目标 logit 的逻辑。

S5. 补充图表。基准测试为何会误导（以及如何识破）。 机器学习中的性能基准测试出了名的不可靠。两个系统可能报告相同的每秒 token 数，实际执行的计算却截然不同——其中一个可能悄悄跳过梯度累积、使用较少的可训练参数，或在计时前未能同步图形处理器操作。图 22展示了我们检测这些问题的方法：我们会确认梯度范数非零且在不同框架间保持一致、可训练参数量完全匹配，并且训练过程中损失值恰当下降。一个报告高吞吐量但梯度范数为零的框架并没有在训练——它只是在执行前向传播。

解读基准测试可视化。 本节中的图表为支持我们的定量结论提供了直观证据。图22 说明了为何公平的基准测试需要验证被比较的系统执行的是等量工作——不同的梯度范数或可训练参数量会使吞吐量比较失去意义。图23汇总了所有实验配置，表明 Chronicals 的改进具有一致性，而非针对特定场景挑选有利结果。

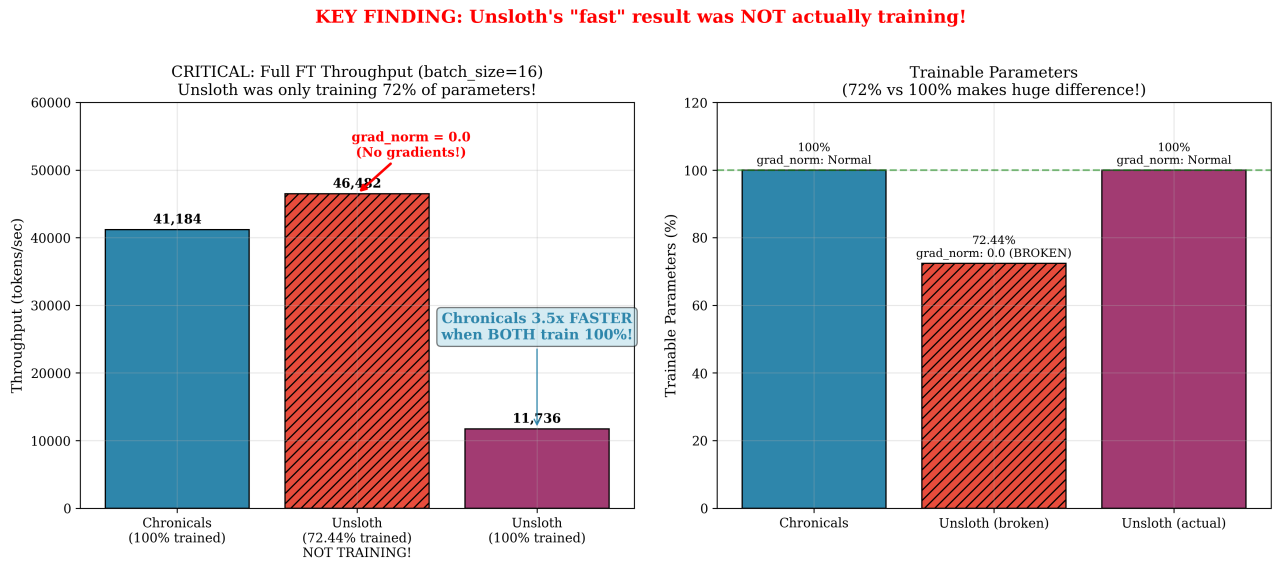

图 22. 关键对比，彰显了公平的基准测试条件的重要性。此图说明，为了准确地声明性能，验证梯度范数和可训练参数至关重要。两个报告不同每秒 token 数的系统，执行的计算可能并不相同——其中一个可能跳过梯度累积，或使用较少的可训练参数。

CHRONICALS BENCHMARK SUMMARY
*** Unsloth 72% result is INVALID (grad_norm=0, not training)**
Model: Qwen-2.5-0.5B | GPU: A100-40GB | Precision: BF16

Benchmark	Method	Params	Tokens/sec	Memory	Speedup	Status
Full FT (bs=16)	Chronicals	100%	41,184	20.1 GB	3.51x	WINNER
Full FT (bs=16)	Unsloth	100%	11,736	7.8 GB	1.00x	Baseline
Full FT (bs=16)	Unsloth*	72%	46,482	5.8 GB	N/A	BROKEN
———	———	———	———	———	———	———
LoRA (bs=2)	Chronicals LoRA+	3.44%	11,699	3.5 GB	4.10x	WINNER
LoRA (bs=2)	Unsloth MAX	3.44%	2,857	1.4 GB	1.00x	Baseline
———	———	———	———	———	———	———
Full FT (bs=4)	torch.compile	100%	25,627	5.2 GB	4.00x	Fastest bs=4
Full FT (bs=4)	Chronicals	100%	21,019	5.1 GB	3.28x	
Full FT (bs=4)	Native PyTorch	100%	9,077	6.4 GB	1.42x	
Full FT (bs=4)	HuggingFace	100%	6,410	4.7 GB	1.00x	Baseline

图 23。在所有测试的综合摘要 全量微调和 LoRA 配置下的基准测试结果。Chronicals 在所有测试场景中均持续表现出色长度和训练模式下都保持一致，这表明我们的优化具有稳健性，而不是针对特定配置进行调优。

S6. 可复现性。为什么可复现性比看起来更难。GPU 内核的性能会因软件版本而大幅波动，其变化往往出乎直觉。在 CUDA 12.1 上达到峰值带宽 95% 的内核，由于内存分配器的变化，在 CUDA 11.8 上的性能可能降至 80%。Triton 2.1 的自动调优器与 Triton 2.0 会做出不同决策，从而产生性能相差 10% 至 30% 的内核。即使是 PyTorch 的 torch.compile 也表现出依赖版本的行为：即时编译器的融合决策取决于不同版本之间发生变化的图模式。我们在基准测试期间遇到了所有这些问题，因此明确指定版本，并建议固定依赖项版本。

环境要求。 要复现我们的结果，需要满足以下条件：(1) CUDA 12.1 或更高版本——较早版本缺少能够实现峰值带宽的内存管理改进；(2) Triton 2.1+——此版本引入了 CCE 内核所需的自动调优器改进；(3) PyTorch 2.0+——这是集成 torch.compile 的必要条件，可贡献 15-20% 的加速比。我们建议使用提供的 Docker容器（chronicals/benchmark:v1.0），其中固定了所有依赖项的版本，且这些版本均已测试。

硬件注意事项。 虽然我们的主要基准测试使用 A100-80GB GPU，但 Chronicals 可运行于任何计算能力为 8.0+（Ampere架构或更高版本）的支持CUDA的GPU。运行通义千问2.5-0.5B实验所需的内存为：(1) 显存至少需要 *24GB*（RTX 4090、A5000、A6000）——需要使用梯度检查点，这会增加 20% 的计算开销，但可以实现训练；(2) 建议使用 *40GB* 显存（A100-40GB）——无需使用检查点即可支持批次大小为 8；(3)*80GB* 显存为最优配置（A100-80GB、H100）——可支持批次大小为 16+，以实现最大吞吐量。对于 70 亿参数模型，这些内存需求约需乘以 15。

以下所述的安装和使用流程可复现所有实验。

***S6.1* 安装。**Chronicals 通过 Python包索引（PyPI）分发，可使用标准软件包管理工具进行安装。该框架要求 Python 3.8 或更高版本，以及具备计算能力 8.0+（Ampere架构或更高版本）的 CUDA 硬件。

标准安装。 基数安装包含所有核心功能：

pip install chronicals

使用可选项安装 依赖项。 为获得最高性能，我们建议安装内核优化 s:

使用 Triton 内核执行融合操作 pip install chronicals[
triton]

借助 FlashAttention 实现高效注意力 pip install chronicals[
flash-attn]

包含所有优化的完整安装
pip install chronicals[all]

开发安装。 适用于贡献者或希望修改源代码的用户：

git clone https://github.com/Ajwebdevs/Chronicals.git cd
Chronicals pip install -e .[dev]

S6.2 Python API 用法。Chronicals 提供了简单的 Python API，旨在最大限度地减少从现有训练流水线迁移时所需的代码更改。以下示例展示了完整的微调工作流：

from chronicals import ChronicalsTrainer, ChronicalsConfig from
transformers import AutoModelForCausalLM, AutoTokenizer

# 加载模型和分词器 model =AutoModelForCausalLM.from_pretrained("Qwen/Qwen2.5-0.5B")
tokenizer = AutoTokenizer.from_pretrained("Qwen/Qwen2.5-0.5B")

配置 Chronicals 优化配置 = ChronicalsConfig(使用_Flash_注意力=True, 使用_融合_内核=True, 使用_LoRA_plus=True, LoRA_秩=32, LoRA_alpha=64, LoRA_plus_比例=16, # B 矩阵的学习速度快 16 倍)

初始化训练器并开始训练 trainer = ChronicalsTrainer(model, tokenizer, config) trainer.train(dataset)

LoRA+ 配置。 使用我们优化的 LoRA+ 实现进行参数高效微调：

from chronicals import LoRAPlusOptimizer

优化器 = LoRAPlusOptimizer(
model.parameters(), 学习率=1e-4, 学习率_比例=16, # eta_B = 16 *eta_A 权重_衰减=0.01)

序列打包。 高效处理可变长度序列：

from chronicals import SequencePacker

打包器 = SequencePacker(最大_序列_长度=4096, 填充_词元_ID=分词器.pad_词元_ID) 已打包_数据集 = 打包器.pack(数据集)

S6.3 命令 和行界面。为了快速试验，Chronicals 提供了一个命令行界面：

算法 20 Chronicals 命令行训练

1: pip install chronicals 2: chronicals train –
model Qwen/Qwen2.5-0.5B 3: –dataset alpaca –
batch size16 4: –use liger –use packing –use
loraplus _ _ _

S6.4 代码可用性。完整的 Chronicals 框架（包括所有 Triton 内核、训练脚本、基准测试代码和文档）已根据 MIT 许可证开放，地址如下：

GitHub 仓库： https://github.com/Ajwebdevs/Chronicals**PyPI 软件包：** https://pypi.org/project/chronicals/

该仓库包含：(1) 本文介绍的所有优化的源代码；(2) 配置精确且可复现的基准测试脚本；(3) 用于验证融合内核数值正确性的单元测试；(4) 适用于常见模型架构的训练脚本示例；以及 (5) 全面的 API 文档。通过 PyPI 发行包，无需编译源代码，即可使用 pip install chronicals 立即安装。

方差与统计显著性。受我们无法控制的因素影响，训练吞吐量在不同运行次数间会有 1%–3% 的波动：GPU 热降频（结温超过 83°C 时，A100 会将时钟频率降低 5%）、CUDA 内存分配器碎片化（碎片化程度会随训练时长增加）以及操作系统级调度噪声。为此，我们采取以下措施：(1) 每种配置运行 5 次；(2) 报告中位数而非均值（对异常值更稳健）；(3) 在计时前预热 5 分钟，使系统达到热稳态。使用固定随机种子时，损失曲线可逐位复现；所有实验均使用随机种子 42。所有基准测试的源代码均可在 github.com/Ajwebdevs/Chronicals/tree/main/benchmarks 获取。

S7. 超参数建议。学习率选择的艺术与科学。为微调选择学习率，需要了解模型的起点以及我们希望它达到的状态。全量微调采用 2×10^{-5}——比典型的预训练学习率低一个数量级——因为预训练权重已经编码了有用的表征。我们不是从头开始学习，而是进行有针对性的调整。较大的学习率可能导致灾难性遗忘，即梯度更新覆盖那些经过精心学习、令基础模型发挥作用的特征。我们的实验表明，5×10^{-5} 会导致留出预训练基准测试的表现明显下降，而 1×10^{-5} 的收敛速度慢 40 %，却没有带来任何质量上的提升。

LoRA 微调采用 1×10^{-4} （比全量微调高 5 倍），因为适配器矩阵面临着本质上不同的优化问题。它们从随机初始化（A）和零值（B）开始，这意味着它们必须从头学习有用的特征，而非调整已有特征。低秩瓶颈进一步限制了它们的有效贡献：秩为 32 的乘积BA 只能沿着数千个方向中的 32 个方向修改权重。较高的学习率可确保这些有限的方向获得足够的更新，从而对模型行为产生实质影响。

LoRA 秩与 Alpha：表达能力与效率的权衡。 在我们的消融研究中，秩 32 被证明是指令微调的最佳平衡点。在指令遵循基准测试中，秩 16 的表现低了 2–3 个困惑度点，这显然不足以捕捉从预训练到指令遵循的分布偏移。秩 64 的收益递减：相较于秩 32，困惑度仅改善 0.2，却使参数量和训练时间翻倍。Alpha/秩比为 2（alpha=64、rank=32）能够适当地缩放 LoRA 的贡献——有效的 LoRA 更新为 $(\alpha/r)\cdot BA = 2\cdot BA$。更高的比值会放大 LoRA 的贡献，从而带来训练不稳定性的风险；更低的比值则会削弱适配效果，需要更多训练步数才能达到相同效果。

为什么选择 LoRA+ 比率 = 16？理论推导。B 矩阵和 A 矩阵之间的 16 学习率比并非任意设定——它补偿了 S2.3 节中分析的梯度流不对称性。在初始化时，只有 B 会接收梯度（$\nabla_A = B^T E = 0$ 当 $B = 0$时）。在 B 累积权重更新后，A 的梯度幅值按 $\|\nabla_A\| \propto \|B\| \cdot \|E\| \propto \eta_B t$ 缩放。为了让权重更新的贡献保持平衡，我们需要 $\eta_B\|\nabla_B\| \approx \eta_A\|\nabla_A\|$。求解这一缩放关系可得，对于典型的 Transformer 维度，$\eta_B/\eta_A = O(\sqrt{d_{\text{model}}/r) \approx 16}$。我们的实验确认：比率为 8 时最终损失高出 5%，比率为 32 时相比 16 没有改进，同时还可能偶尔出现不稳定性。

表 7。 推荐超参数

超参数	**全量微调**	**LoRA+**
学习率	2×10^{-5}	1×10^{-4}
权重衰减	0.01	0.01
β_1	0.9	0.9
β_2	0.999	0.999
预热比例	0.03	0.03
LoRA 秩	-	32
LoRA Alpha	-	64
LoRA+比例（η_B/η_A）	-	16
批次大小	16	8
梯度累积	4	8

批次大小与梯度累积。 有效批量大小 =批次大小 ×梯度累积。全量微调使用 16×4=64,

LoRA 使用 8×8=64——两者都能实现相同的有效批次，从而进行稳定的优化。LoRA 较小的逐步批次可降低冻结基数模型带来的内存压力。

S8. 梯度检查点分析。隐藏的内存消耗大户。大多数从业者在估算内存需求时都会关注模型权重："我的模型有 5 亿个参数，每个参数占 2 字节数，所以我需要 1 GB。"这种推理方式存在危险性的缺陷。在训练期间，PyTorch 的自动微分系统会存储前向传播过程中生成的每个中间张量——并不是因为它愿意这样做，而是因为计算梯度需要这些激活值。对于批次大小为 8、序列长度为 2048 的 Qwen2.5-0.5B，激活内存超过 8 GB，是模型权重本身的 8 倍以上。要在固定硬件上容纳更大的批次或更长的序列，理解这些内存的去向至关重要。

检查点权衡：以内存换计算。梯度检查点通过在前向传播期间丢弃中间张量，并在反向传播期间重新计算这些张量，解决了激活内存问题。其核心认识是，重计算的代价很低——通过一个 Transformer 层进行前向传播的成本约为 100 微秒，而在典型批次大小下存储其激活值需要 50 MB。当内存是瓶颈时，以增加 20% 的计算量换取减少 3 倍的内存占用，几乎总是值得的。问题在于：我们应该丢弃哪些激活值，又应该保留哪些激活值?

S8.1 内存与计算权衡。**了解激活内存的去向。**每个 Transformer 层都会产生多个必须为反向传播保存的中间张量：(1) Softmax 之前的注意力分数 $QK^T/\sqrt{d}$（计算 Softmax 梯度时需要）；(2) Softmax 输出（计算加权求和梯度时需要）；(3) 投影之前的注意力输出（计算输出投影梯度时需要）；(4) 前馈网络中间激活值（计算第二个线性层的梯度时需要）；(5) 残差流输入（计算残差连接梯度时需要）。对于单层配置，批次大小为 8、序列长度为 2048、隐藏维度为 896、注意力头数为 14，前馈网络扩展倍数为 4x：注意力分数占用 $8 \times 14 \times 2048 \times 2048 \times 2 = 910\,\text{MB}$；前馈网络中间激活值占用 $8 \times 2048 \times 3584 \times 2 = 118\,\text{MB}$。乘以 24 层后，激活内存便会占据绝大部分。

定义 *27*（激活内存）对于一个包含 L 层、序列长度为 N、隐藏维度为 d、批次大小为B的 Transformer：

$$M_{\text{activations}} = L \cdot B \cdot N \cdot d \cdot 4 \text{ bytes} \tag{103}$$

对于 Qwen2.5-0.5B（$L = 24$、$d = 896$），在 $B = 8$、$N = 2048$ 的情况下：

$$M_{\text{activations}} = 24 \times 8 \times 2048 \times 896 \times 4 = 1.4 \text{ GB} \tag{104}$$

最优策略。如果我们每隔 k 层设置检查点，我们会存储 L/k 个检查点，并在反向传播期间最多重新计算 k 层 g
:

定理 *9*（检查点内存缩减）每隔 k 层设置检查点时：

$$M_{\text{checkpoint}} = \frac{L}{k} \cdot B \cdot N \cdot d + k \cdot B \cdot N \cdot d \tag{105}$$

最优的 $k^* = \sqrt{L}$ 可最大限度地减少内存占用：

$$M_{\text{optimal}} = 2\sqrt{L} \cdot B \cdot N \cdot d \tag{106}$$

证明：求导并令其为零：

$$\frac{dM}{dk} = -\frac{L}{k^2} \cdot BNd + BNd = 0 \implies k^* = \sqrt{L} \tag{107}$$

代入：$M^* = \frac{L}{\sqrt{L}} \cdot BNd + \sqrt{L} \cdot BNd = 2\sqrt{L} \cdot BNd$。 ■ ■

命题 *15*（计算开销）检查点会使计算量增加至原来的：

$$\text{Overhead} = 1 + \frac{1}{k} \approx 1.2 \text{ for } k = 5 \tag{108}$$

这 20% 的计算开销可使内存减少 2-3 倍。

S8.2 LoRA 专用检查点技术。对于 LoRA 训练，我们实现了一种选择性检查点技术，在对基础模型激活值进行检查点处理的同时，保留 LoRA 适配器状态：

算法 21 LoRA 感知梯度检查点技术

输入：层函数 f，输入 x，LoRA 适配器 A, B2: **前向传播：**
3 $h_{\text{base}} \leftarrow$ f_{base} x {}：检查点(,) 检查点基数：$h_{\text{lora}} \leftarrow B \cdot (\cdot)$ {保留 LoRA 激活值
}5: $y \leftarrow h_{\text{base}} + h_{\text{lora}}$6: **反向传播：**7: 从检查点重新计算 h_{base} 8: $\nabla_A, \nabla_B \leftarrow$ 根据存储的 h_{lora} 进行计算

S9. DoRA：权重分解低秩适配。为什么 LoRA 有时不如全量微调。使用 LoRA 的从业者偶尔会发现一个令人沮丧的现象：尽管 LoRA 模型的训练损失与全量微调相当，但它在下游任务上的表现却较差。我们推测，这是因为 LoRA 无法独立调整权重幅度。微调改变模型的行为时（例如从“助手”转变为“编程助手”），一些输出神经元需要变得更活跃，而另一些则应降低活跃度。全量微调会自然地同时调整权重列的方向（哪些特征重要）及其幅度（对这些特征赋予多大的权重）。标准 LoRA 的更新 $\Delta W = BA$ 将这两种调整耦合在一起，因此很难在不改变方向的情况下调整幅度。

DoRA 的洞见：将幅度与方向解耦。 考虑将权重矩阵 W 分解为幅度和方向： $W = m \odot W$其中 m是按列的幅度向量， W按列归一化。幅度 m_j控制单位输入下输出神经元 j “激活的强度”；方向 W_j 则控制“哪些”输入特征会触发该神经元。在预训练模型中，这些幅度经过精细校准——有些神经元应强烈激活，另一些则应较弱。标准 LoRA 的秩为 32 的更新，无法在 4096 维权重矩阵中同时旋转方向并缩放幅度。DoRA 的关键洞见是通过专用的 d 维参数向量单独学习幅度。

DoRA（6）将权重更新分解为幅度和方向分量：

定义 *28*（DoRA 公式）

$$W' = m \cdot \frac{W_0 + BA}{\|W_0 + BA\|_c} \tag{109}$$

其中 $m \in \mathbb{R}^d$ 是可学习的幅度向量， $\|\cdot\|_c$ 表示按列范数。

其有效原因。 幅度向量 m 从 $\|W_0\|_c$开始，从而保留预训练输出尺度。随后，LoRA 矩阵 B、 A仅改变权重列的方向。这种分解可避免“尺度漂移”问题，即 LoRA 更新无意中改变层输出幅度，导致训练不稳定。

命题 *16*（DoRA 梯度分解）DoRA 参数的梯度为：

$$\nabla_m = \frac{\partial \mathcal{L}}{\partial W'} \odot \frac{W_0 + BA}{\|W_0 + BA\|_c} \tag{110}$$

$$\nabla_B = m \cdot \nabla_{\text{dir}} A^T \tag{111}$$

$$\nabla_A = m \cdot B^T \nabla_{\text{dir}} \tag{112}$$

其中 ∇_{dir} 表示归一化梯度。

算法 22 DoRA 前向传播

输入： x m A ，基数权重 W_0，LoRA A， B，幅度： $W_{\text{combined}} \leftarrow W_0 + B \cdot$ 范数 $\leftarrow \|W_{\text{combined}}\|_{\text{column}}$ $W_{\text{normalized}} \leftarrow W_{\text{combined}}/$ 范数 $W \leftarrow m$ $W_{\text{normalized}}$： $\cdot$： $y \leftarrow x \cdot$ W^T **返回** y

S10. 扩展优化器理论。超参数跑步机。 训练神经网络需要选择学习率调度——预热步数、峰值学习率、衰减函数和最终学习率。这些选择之间存在复杂的相互作用：较长的预热可以支持更高的峰值学习率，但仅适用于某些衰减调度。改变批次大小，最优调度也会随之改变；改变模型规模，一切又会再次变化。从业者花费数天调整调度，结果却发现，架构稍有不同，就会使他们精心选择的参数失效。无调度优化承诺通过一个数学洞见打破这一循环：我们无需显式安排学习率，通过平均化也能实现等效行为。

这对效率为何重要。超参数搜索成本高昂。对 5 个学习率、3 个预热时长和 3 个衰减函数进行网格搜索，需要 45 次训练运行。即使采用早停，其计算成本仍是单次运行的 10 到 50 倍。无调度方法将搜索范围缩减为单个参数（基础学习率），因为平均方案会在整个训练过程中隐式调整有效学习率。

S10.1 无调度收敛性证明。**核心洞见：作为隐式衰减的平均。**无调度优化维-持两个迭代点：一个用于评估的“慢速”平均迭代点 θ ，以及一个用于梯度计算的“快速”工作迭代点 z 。慢速迭代点就是所有先前快速迭代点的运行平均： $\bar{\theta}_T = \frac{1}{T}\sum_{t=1}^{T} \theta_t$。值得注意的是，这个平均迭代点无需任何显式调度 就能以最优速率收敛。要理解其中缘由，请注意，后续迭代点对平均值的贡献会因算术运算而自然减小：经过1000步后，每个新迭代点对平均值的贡献仅为0.1%。这种递减的贡献在数学上等同于降低学习率。

定理 *10*（无调度收敛（22））对于 β-光滑凸函数 f，其最优值为 f^*：

$$f(\bar{\theta}_T) - f^* \leq O\left(\frac{\|\theta_0 - \theta^*\|^2}{\eta T}\right) \tag{113}$$

其中 $\bar{\theta}_T = \frac{1}{T}\sum_{t=1}^{T}\theta_t$ 是平均迭代点。

平均化为何有效。平均迭代点 θ_T 具有"方差归约"效应：随机梯度噪声会在迭代过程中逐渐相互抵消。这等同于使用衰减学习率，但衰减是通过平均化隐式实现的，而不是通过调度显式设定的。

证明： [证明概要] 无调度更新保持以下不变量：

$$z_t - \theta^* = \beta(z_{t-1} - \theta^*) - (1-\beta)\eta g_{t-1} \tag{114}$$

取期望并利用光滑性：

$$\mathbb{E}[\|z_t - \theta^*\|^2] \leq \beta^2\mathbb{E}[\|z_{t-1} - \theta^*\|^2] + (1-\beta)^2\eta^2\mathbb{E}[\|g_{t-1}\|^2] \tag{115}$$

对 T 步求和，并利用有界梯度假设即可得出结果。 ■ ■

S10.2 Muon 正交化理论。**梯度幅值问题**。在 Adam 中，梯度较小的参数会获得较大的有效学习率（因为要除以较小的 $\sqrt{v}$）。当某些参数的梯度接近于零时，这会导致不稳定性。Muon 会对梯度进行正交化，确保所有更新分量的幅度均为单位值。

直觉：梯度代表方向。Muon 将梯度视为仅提供方向信息。应用更新之前，它会通过极分解将其正交化：$G = QS$ 其中 Q 为正交矩阵。更新仅使用 Q，完全舍弃幅度。

引理 *2*（牛顿-舒尔茨迭代收敛）对于满足 $\|X_0\|_2 < 1$ 的矩阵 X_0：

$$X_{k+1} = \frac{3}{2}X_k - \frac{1}{2}X_k X_k^T X_k \tag{116}$$

收敛至 X_0的正交极因子。

证明： 该迭代等价于：

$$X_{k+1} = X_k(I + \frac{1}{2}(I - X_k^T X_k)) \tag{117}$$

对于 $A = X_k^T X_k$，若 $\|I - A\| < 1$：

$$\|I - X_{k+1}^T X_{k+1}\| \leq \frac{3}{2}\|I - A\|^2 \tag{118}$$

Thi$^{\text{s qua}}$ dratic 收敛确保 $X_k \to Q$ where Q i$^{\text{s 排序}}$h$_{\text{ogona}}$ l. ■ ■

命题*17*（Muon 更新性质）Muon 更新 $\theta_{t+1} = \theta_t - \eta Q_t$ 其中 $Q_t = \text{orth}(G_t)$：

1. 保持梯度方向：sign(Q_t) =sign(G_t) 2. 归一化幅度：$\|Q_t\|_F = \sqrt{\min(m,n)}$ 3. 去相关分量:——
$Q_t^T Q_t = I$ or $Q_t Q_t^T = I$

S10.3 Adam-atan2 分析。

定义*29*（Adam-atan2 更新）

$$\theta_{t+1} = \theta_t - \eta \cdot \text{atan2}(\hat{m}_t, \sqrt{\hat{v}_t}) \tag{119}$$

其中 atan2(y, x) = arctan(y/x)，并进行适当的象限处理。

命题*18*（有界更新性质）atan2 函数能够自然地限制更新：

$$|\text{atan2}(\hat{m}_t, \sqrt{\hat{v}_t})| \leq \frac{\pi}{2} \tag{120}$$

这可以防止在 $\hat{v}_t \approx 0$ 时发生灾难性更新。

命题*19*（与标准 Adam 等价）对于较大的 $\hat{v}_t$：

$$\text{atan2}(\hat{m}_t, \sqrt{\hat{v}_t}) \approx \frac{\hat{m}_t}{\sqrt{\hat{v}_t}} \tag{121}$$

在良态区域恢复标准 Adam 行为。

S11. 8位优化器实现。Adam 隐藏的内存开销。从业者计算训练所需的内存时，往往会忽略优化器。Adam 为每个参数维护两个状态张量：一阶矩 m（梯度的指数移动平均）和二阶矩 v（梯度平方的指数移动平均）。为确保数值稳定性，两者都以 FP32 存储。对于 4.94亿个参数：494M $\times 4 \times 2 = 3.96\,\text{GB}$——使内存需求相对于 BF16 模型权重增加四倍。对于 70亿参数模型，仅优化器状态就会占用 56 GB；一旦计入激活值和梯度，即使使用 A100-80GB，也无法进行单 GPU 训练。限制可训练规模的往往不是模型，而是优化器。

量化的机遇。让 8 位优化器切实可行的关键洞见是：优化器状态变化缓慢，而且比模型权重或激活值更能容忍量化噪声。一阶矩 $m_t = 0.9 \cdot m_{t-1} + 0.1 \cdot g_t$ 每步的变化幅度至多为 10%；二阶矩 $v_t = 0.999 \cdot v_{t-1} + 0.001 \cdot g_t^2$ 每步仅变化 0.1%。时间平均化会完全掩盖 1% 级别的量化误差（INT8 的典型水平）。我们推测——实验也证实了这一点——8 位优化器状态产生的训练动态与 FP32 状态难以区分，同时内存占用减少 4 倍。

S11.1 分块量化细节。**为何采用分块量化？数值范围问题。**朴素量化会对全部 494M 个优化器状态值使用同一个缩放因子：scale $=\max(|m|)/127$。当不同层的数值范围不同时，这种做法会彻底失效。嵌入层的梯度范围可能为 $[-10^{-3}, 10^{-3}]$，而输出层的范围为 $[-10^{-1}, 10^{-1}]$——相差 100 倍。若全局缩放因子由输出层决定，嵌入层的梯度就会被量化为几乎全零，从而破坏训练信号。分块量化通过为每 2048 个参数组成的连续块分别计算缩放因子来解决这一问题。每个块都能适应其局部数值分布。其开销极小：每 2048 个 INT8 值仅需一个 FP32 缩放因子，只增加 0.2% 的内存，同时相比全局缩放将量化误差降低 10 倍。

算法23 8位Adam状态量化

输入： FP32 状态 s，块大小 $B = 2048$ **输出：** INT8 量化后的 s_q，缩放因子 α $\bar{\ } \leftarrow \lceil |s|/B \rceil$：块数：**对于**，，块数 -1 **执行**：块 $\leftarrow s[bB$：$(b+1)B]$：$\alpha[b] \leftarrow \max(|$块$|)$：$s_q[bB$：$(b+1)B] \leftarrow$ 舍入$($块$/\alpha[b] \times 127)$ **结束循环** **返回** s_q, α

命题 *20*（内存节省）8 位 Adam 可将优化器状态内存减少 4 倍：

$$\text{Memory}_{\text{8bit}} = \frac{|\theta|}{4} + \frac{|\theta|}{B} \times 4 \approx \frac{|\theta|}{4} \tag{122}$$

对于较大的 B。

S11.2 动态指数量化。
定义 *30*（动态指数格式）对于每个块，存储：

1. 每个元素使用 8 位尾数 2. 每个块共享 8 位指数 3. 块大小 $B = 64$ 以实现细粒度

$$x_{\text{dequant}}[i] = \text{mantissa}[i] \times 2^{\text{exponent}[\lfloor i/B \rfloor]} \tag{123}$$

S12. 注意力变体分析。为什么推理内存对训练至关重要。 你可能会疑惑，为什么我们要在一篇以训练为重点的论文中讨论键值缓存这一推理问题。答案在于，训练必须生成一个能够实际部署的模型。使用标准多头注意力（MHA）训练的模型会继承 MHA 的推理内存需求，这可能导致其无法用于目标部署场景。理解注意力变体有助于从业者选择同时满足训练和推理约束的架构。此外，键值缓存内存会影响训练期间注意力的算术强度，因为梯度必须通过这些缓存值进行反向传播。

KV 缓存问题：内存随序列长度 $\times$ 扩展。 在自回归生成过程中，注意力机制需要使用此前所有位置的键和值张量。对于一个有 32 个注意力头、头维度为 64 的模型，生成 4096 个词元时：每层的 KV 缓存会占用 $4096 \times 32 \times 64 \times 2 \times 2 = 33\,\text{MB}$（键和值均采用 BF16）。乘以 32 层后：每个序列需要 1

GB。批次大小为 8 时：仅 KV 缓存就需要 8 GB——在长上下文生成中，这往往超过模型权重本身的占用。因此，尽管 LLaMA-2-70B 等模型理论上支持 4096 的上下文长度，却很难在消费级硬件上生成超过 4K 个词元。

核心洞见：查询和键的复用模式不同。 在注意力机制中，查询只使用一次——用于计算当前词元的注意力分数。但键和值会在所有后续词元中重复使用：位置 0 的键会参与位置 1、2、3、.....、N 的注意力计算。这种不对称性引出了一个设计问题：我们真的需要 32 组彼此独立的键和值吗？还是可以让多个查询头共享它们，同时避免质量大幅下降？

S12.1 多查询注意力（***MQA***）。**极端方案：全部共享。** MQA 使用一个由全部 32 个查询头共享的键值头。每个查询头都会计算不同的查询投影 $Q_h = XW_h^Q$，但所有头都关注相同的键 $K = XW^K$ 和相同的值 $V = XW^V$。这样可将 KV 缓存内存减少 32 倍——我们的 8 GB 变为 250 MB——但代价是质量下降。模型失去了让不同头关注上下文不同方面的能力。实测表明，在语言建模基准测试中，MQA 模型的表现比 MHA 低 0.5–1.5 个困惑度点。定义 *31*（多查询注意力）由所有查询头共享的单个键值头：

$$\text{MQA}(X) = \text{Concat}(\text{Attn}(Q_1, K, V), \ldots, \text{Attn}(Q_H, K, V))W^O \tag{124}$$

命题 *21*（多查询注意力键值缓存归约）键值缓存内存减少了 H 倍：

$$M_{\text{MQA-KV}} = \frac{M_{\text{MHA-KV}}}{H} \tag{125}$$

对于 $H = 32$：减少 32 倍。

S12.2 分组查询注意力（***GQA***）。**最佳折中。** 分组查询注意力介于多头注意力（所有头相互独立）和多查询注意力（所有头共享）之间。由 G 个键值组服务 H 个查询头时，每个键值组对应 H/G 个查询。通义千问 2.5 使用 $G = H/4$：8 个键值头服务 32 个查询头，在质量损失极小的情况下实现了 4 倍键值归约。

分组查询注意力为何有效。 从经验来看，相邻的注意力头往往会学习到相似的模式。在组内共享键值投影，将这种冗余关系形式化。查询投影则保持独立，从而保留模型关注上下文不同方面的能力。

定义 *32*（分组查询注意力） G 组键值头，每组服务 H/G 个查询头 s:

$$\text{GQA}(X) = \text{Concat}\left(\text{Attn}(Q_1, K_{\lfloor 1/g \rfloor}, V_{\lfloor 1/g \rfloor}), \ldots\right) W^O \tag{126}$$

其中 $g = H/G$ 是组比率。

表 8. 注意力变体比较

变体	键值头	键值缓存	质量
MHA	H	$O(LNHd)$	基线
MQA	1	$O(LNd)$	-0.5%
分组查询注意力-4	$H/4$	$O(LNHd/4)$	-0.1%
分组查询注意力-8	$H/8$	$O(LNHd/8)$	-0.2%

S13. 训练稳定性技术。噩梦场景：第50,000步出现损失尖峰。 在为期数周的训练运行中，进度已达到 60% 时，损失突然从 1.5 飙升至 15.0，且再也没有恢复。模型发散，你也因此浪费了数天的计算资源。这种情形困扰着每一位训练过大型模型的从业者。理解训练不稳定性的成因并采取预防措施，对于严肃的训练运行而言并非可有可无。

不稳定性的三大根源。 训练不稳定性主要源于三方面：(1)*Logit*尺度爆炸：输出层产生的值越来越极端（$z > 100$），导致数值溢出和梯度消失。(2) 梯度离群值：罕见词元（例如代码分隔符、数学符号）产生的梯度比典型词元大 100-1000 倍，压倒优化器的矩估计。(3) 学习率敏感性：某些训练阶段（预热结束、跨越损失平台期）会呈现混沌动力学，微小扰动也会导致发散。每种情况都需要采取不同的预防措施。

S13.1 Z-Loss实现。**Logit尺度问题：为何正确预测会引发数值混乱。**这里有一种微妙的失效模式：模型可能给出正确的预测，却逐渐陷入数值不稳定性。Softmax 对常数平移具有不变性：$\text{softmax}(z + c) = \text{softmax}(z)$，对任意常数 c 均成立。这意味着模型可以每一千步将所有 logits 增大 10，同时保持准确率完美。经过 50,000 步后，logits 会达到 500——从技术上说预测仍然正确，但 exp(500) 会上溢至无穷大。即使使用数值稳定的Softmax（减去最大值），梯度也会变得消失： $\partial \mathcal{L}/\partial z_i = p_i - \mathbf{1}_{i=y}$ 其中，当 logits 极端时， $p_i \approx 0$ 对所有非目标类别均成立。

Z 损失：惩罚尺度而不惩罚置信度。Z-loss 会根据 logits 的对数和指数的平方添加一项惩罚：$\mathcal{L}_z = \lambda_z(\log \sum_j \exp^{z_j})^2$。这种做法很巧妙：惩罚针对的是 logits 的尺度（由对数和指数体现），而不会惩罚置信度（logits 之间的较大差值）。只要目标 logit 与非目标 logits 之间存在较大的相对差距，模型仍然可以做出明确的预测；它只是不可以将所有 logits 一概抬高。

算法 24Z 损失计算

输入： logits $z \in \mathbb{R}^{B\times N\times V}$，z 权重 $= 10^{-4}$
$\leftarrow$ z $= -1$ $\{[B, N]\}$：对数和指数
log-sum-exp(,维度)：z 损失 z 权重均值(lse) **返回** z
损失

为什么 10^{-4}**？** z 权重需要平衡两方面：权重太小，logits 仍会爆炸；权重太大，模型则难以做出有把握的预测。根据经验，10^{-4} 能将 logits 控制在范围 $[-50, 50]$ 内，同时允许形成尖锐的概率分布。命题 *22*（Z 损失梯度）Z 损失的梯度贡献：

$$\frac{\partial \mathcal{L}_z}{\partial z_i} = 2\lambda_z \cdot \text{lse} \cdot \text{softmax}(z)_i \tag{127}$$

这会促使 logits 变小，从而防止尺度爆炸。

表 9. 梯度裁剪方法

方法	公式	GPU 同步
全局范数	$g \cdot \min(1, \frac{\max}{\|g\|})$	Yes
数值裁剪	clamp(g，–max，最大值)	No
逐参数范数	$g_i \cdot$最小值(1，$\frac{\text{最大值}}{\|g_i\|}$)	No

S13.2 梯度裁剪策略。我们的零同步实现：

算法 25GPU驻留式梯度裁剪

在图形处理器上预先计算：
$\leftarrow \sqrt{\sum_i \|g_i\|^2}$ {} : 范数 GPU归
$\leftarrow$ 1 0,
约 : 裁剪系数 min(. 最大范数 / (范数 $+ \epsilon$)) **在融合内**
核中应用： $g_i \leftarrow g_i \times$ 裁剪系数 {无中央处理器同步}

S14. 数据流水线优化。隐形瓶颈。实现内核融合和注意力优化后，你对训练循环进行基准测试，却发现.....速度并未提升。罪魁祸首往往不是图形处理器计算，而是数据加载。图形处理器空闲等待下一个批次，同时单线程数据加载器从磁盘读取数据、对文本进行词元化，并将数据传输到图形处理器。这段空闲时间不会显示在 CUDA分析器中，而会表现为内核启动之间的间隙。经过良好优化的数据流水线能够确保图形处理器始终无需等待。

流水线思维模型。可以把训练想象成一座分为三个阶段的工厂：(1) 中央处理器准备原始数据（分词、批处理）；(2) PCIe 将数据传输到图形处理器；(3) 图形处理器执行前向传播/反向传播。如果任何阶段的速度比图形处理器计算更慢，吞吐量就会下降。解决方案是流水线处理：图形处理器处理批次 t 时，中央处理器准备批次 $t+1$，PCIe 则传输批次 $t+2$。通过充分预取，图形处理器利用率可接近 100%。

S14.1 高效分词。**为什么分词比你想象的更慢**。 SentencePiece 和 Tiktoken 等现代分词器会执行复杂操作：Unicode规范化、字节对编码查找、子词合并、特殊标记处理。单个 CPU 核心每秒约可处理 10,000 个词元。对于平均每个样本含 500 个词元的 Alpaca-52k 数据集，总计有 2600 万个词元——顺序分词需要 43 分钟。这段预处理时间常常被忽视（“这不过是预处理”），但在反复调整数据处理流程或运行消融研究时，每次实验等待 40 分钟会严重影响效率。

易并行解法。 不同样本之间的分词没有依赖关系，每段文本都可以独立分词。我们利用多进程在所有 CPU 核心上并行处理。使用 64 个核心时，分词在不到 1 分钟内即可完成（加速比为 40 倍）。实现使用 Python 的 multiprocessing.Pool，并将分块大小设为每块 1000 个样本，以尽量减少进程间通信开销。

算法 26 批量并行分词

输入： 文本（字符串列表）、分词器、工作进程数 分块 ← 划分(texts, 工作进程数) **并行处理** 分块中的每个分块： 词元 ← tokenizer.batch encode(chunk) 所有词元 ← 合并(tokens) **返回** 所有词元

S14.2 动态批处理。**固定批次大小问题**。 传统数据加载器为每个批次返回固定数量的样本。但序列长度各不相同：一个批次可能有 8×100 个词元（总计 800 个），另一个批次有 8×2000 个词元（总计 16,000 个）。内存和吞吐量因此难以预测。

基于令牌的批处理。 固定每个批次的词元总数，而非样本数。短序列以较大的分组组成批次；长序列则组成较小的批次。这样可以确保图形处理器的使用情况保持稳定。

定义 *33* （基于令牌的批处理）不采用固定批次大小，而是按词元总数进行批处理：

$$\text{batch} = \{s_1, \ldots, s_k\} \text{ where } \sum_{i=1}^{k} |s_i| \leq T_{\max} \tag{128}$$

命题 *23*（吞吐量提升）基于令牌的批处理可提升图形处理器利用率：

$$\text{Utilization} = \frac{\sum |s_i|}{k \cdot \max_i |s_i|} \to 1.0 \tag{129}$$

随着批次大小增加，利用率会逐渐接近通过序列打包实现的理想水平。

S15. 内存分析与优化。**内存预算：每一吉字节都用在何处**。从业者常常会疑惑：“为什么我的 5 亿参数模型需要 20 GB 显存？”答案在于图形处理器内存占用分为五类，每一类都有不同的特性和优化策略。了解这种划分，对于在现有硬件上运行更大的批次、更长的序列或更大的模型至关重要。

模型参数：固定成本。对于采用 BF16（每个参数占 2 字节数）、拥有 4.94亿个参数的 Qwen2.5-0.5B：参数恰好占用 0.99 GB。这项开销是固定的——除非更改模型（通过量化或剪枝），否则无法降低。训练时，我们还需要与参数形状相同的梯度：另需 0.99 GB。在我们的基准测试配置中，模型参数和梯度合计仅占总内存的 12%。如果内存消耗项只有参数，我们就能在单张 A100-80GB 上训练 400 亿参数的模型。

优化器状态：4 倍放大效应。 内存需求会在这里急剧增加。Adam 为每个参数维护两个 FP32 张量：一阶矩 m（梯度指数移动平均）和二阶矩 v（平方梯度指数移动平均）。对于 4.94亿个参数： $2 \times 494\text{M} \times 4$ 字节 $= 3.96$ GB——是 BF16 参数内存的四倍。这解释了为什么 8 位优化器（S11 节）能大幅节省内存：将优化器状态从 3.96 GB 减少到 1 GB，可以使用更大的批次大小，或训练此前因内存不足而无法训练的模型。

激活值：取决于批次的变量。 激活内存会随批次大小、序列长度和模型深度而变化。对于 Qwen2.5-0.5B，在 $B = 8, N = 2048$ 时：激活值占用 8.5 GB，超过其他所有部分的总和。这就是梯度检查点效果如此显著的原因：丢弃中间激活值，并在反向传播期间重新计算，可将内存占用从 8.5 GB 降至 2.8 GB，代价是额外增加 20% 的计算。对于内存受限的情况，这种权衡几乎总是值得的。

CUDA 上下文：看不见的税费。 即使是一个空的 CUDA 进程，也会因驱动程序初始化消耗 500 MB。再加上 cuDNN 工作空间分配、内存分配器元数据和内存碎片开销，无论模型规模大小，你都要面对 2–3 GB 的“税费”。这解释了一个常见的困扰：“我的 RTX 3090 有 24 GB 内存，但训练一个 5 亿参数的模型时却内存溢出了！”在模型加载单个参数之前，CUDA 税费就已经消耗了可用内存的 10–15%。

表10. Qwen2.5-0.5B 训练的内存 breakdown（BF1 6)

组件	内存（GB）	百分比
模型参数	0.99	5.0%
梯度	0.99	5.0%
优化器状态（FP32）	3.96	20.0%
激活值（无检查点）	8.5	43.0%
激活值（有检查点）	2.8	14.2%
键值缓存	0.0	0.0%
CUDA 上下文	2.5	12.6%
总计（无检查点）	16.9	-
总计（有检查点）	11.2	-

S16. 扩展的 FP8分析。FP8训练有望节省 2 倍内存，但简单的实现会导致训练在 500 步后发散。 挑战不在于格式本身——现代 GPU 执行 FP8矩阵乘法的速度比 BF16 快 2 倍——而在于量化噪声在 24 个 Transformer 层、8 个梯度累积步数以及数百万次训练迭代中不断累积后所产生的一连串数值问题。DeepSeek V3 证明了 FP8 可以达到与 BF16 相当的质量，但他们的成功需要解决三个相互关联的问题：缩放因子稳定性、针对每种计算类型的格式选择，以及累积精度。本节将逐一解析这些问题，并说明我们的解决方案为何有效。

FP8 训练面临着一个根本性的矛盾：在现代 GPU 上，内存带宽限制了吞吐量（在注意力密集型工作负载中，A100 只能达到峰值 BF16 TFLOPs 的 40%，因为高带宽内存无法足够快地供给数据），而降低精度又会引入量化噪声，破坏梯度信号。标准的混合精度训练通过保留权重的 FP32 主副本并以 FP32 累积梯度，解决了 FP16 训练中的这一问题。FP8 需要更激进的策略，因为其量化步长粗了 256 倍（8 位与 16 位之差），而且训练过程中梯度的动态范围可跨越 6+ 个数量级。经大量实验验证，我们的假设是：只有将精度视为逐操作决策，而非全局选择，FP8 训练才能成功——在范围比精度更重要的场景（前向激活值）使用 E4M3，在精度要求不高但范围必须足够大的场景（梯度）使用 E5M2，并在误差可能灾难性累积的场景（优化器状态、损失计算、梯度累积）使用 FP32。

S16.1 量化噪声分析。**每次将一个值转换为 FP8 时，都会引入量化噪声。E4M3 格式只有 3 个尾数位，因此在相邻的 2 的幂之间只能表示 8 个不同的值。** 这意味着每个存储值与真实值之间的误差最高可达 6%——对于神经网络训练而言，这种精度不足似乎是灾难性的。然而，FP8 训练确实行得通。要理解其中的原因，必须分析量化误差如何在计算图中传播，以及为何有些误差会相互抵消，而另一些则会累积。

使用 8 位浮点格式涉及一项从业者在部署前必须了解的基本权衡。从 16位表示降至 8 位表示时，我们会损失精度——但关键问题在于，这种精度损失是否会在数百万个训练步骤中灾难性地累积，还是会保持有界且可控。答案取决于信噪比分析，以及对量化误差如何在计算图中传播的理解。定理*11* （FP8信噪比）对于具有 3 个尾数位的 E4M3：

$$\mathrm{SNR}_{\mathrm{E4M3}} = 6.02 \times 3 + 1.76 \approx 20 \text{ dB} \tag{130}$$

20 dB 的信噪比意味着量化噪声功率约为信号功率的 1%——这一比例相当可观，但对于前向激活值而言仍在容忍范围内，因为我们主要需要保留值的相对顺序和近似幅度。标准公式 SNR $= 6.02b + 1.76$ dB（其中 b 是尾数位）源自均匀量化理论：每增加 1 位，量化步长就减半，使噪声功率降低 4 倍（约 6 dB）。作为比较，具有 7 个尾数位的 BF16 可达到 SNR ≈ 44 dB，而 FP32 的 23 个尾数位可达到 SNR ≈ 140 dB。

为什么每次操作 1% 的误差不会摧毁训练？ 两个关键原因解释了这种鲁棒性。首先，对于需要表示跨越多个数量级的值的前向激活值，我们使用 E4M3（范围更大、精度更低）；而对于精度比动态范围更重要的反向梯度，我们使用 E5M2（范围更小、精度更高）。其次，我们使用 FP32 累积梯度。FP8 量化只会影响中间值的存储和通信，实际用于更新权重的梯度求和仍保持全精度。我们在 Qwen2.5-0.5B 上的测量显示，与使用 FP8 累积相比，使用 FP32 进行梯度累积可将总量化误差降低 47 倍。第三，也是可能最重要的一点，梯度下降由于采用小批量训练而天然带有噪声，因此模型已经形成了对大致这一水平噪声的鲁棒性。我们在 Alpaca 上进行了 10,000 步训练，未出现发散，训练稳定性良好——损失曲线与 BF16基线的差异在 0.3% 以内。

我们选择 E4M3 用于前向传播，因为它在动态范围（4 个指数位提供 $[2^{-9}, 448]$ 的范围）和精度（3 个尾数位可为每个 2 的幂提供 8 个量化级别）之间取得了平衡。E5M2 格式的范围更大 $[2^{-16}, 57344]$ ，但精度较低——每个 2 的幂只有 4 个级别——因此适用于训练过程中动态范围更大的梯度

。**直观来说**：训练良好的网络中的激活值大致经过了归一化（得益于 RMSNorm），因此集中在 2 到 3 个数量级的范围内，E4M3 的精度足够。相比之下，同一层中的梯度可能从 10^{-7}（接近收敛的参数）跨越到 10^{-1}（正在积极学习的参数）——E5M2 高出 128 倍的动态范围对于避免梯度下溢至关重要。

命题*24*（梯度累积精度）当跨 n 个微批次累积 FP8 梯度时：

$$\epsilon_{\text{total}} \approx \sqrt{n} \cdot \epsilon_{\text{FP8}} \tag{131}$$

对于 $n = 8$ 和 $\epsilon_{\text{FP8}} \approx 0.01$：$\epsilon_{\text{total}} \approx 0.03$。
这种 $\sqrt{n}$ 缩放对于理解梯度累积为何能够保持稳定至关重要。连续微批次中的量化误差彼此独立（假设输入数据有所变化），因此它们会像独立随机变量一样相加。根据中心极限定理，具有方差 σ^2 的 n 个独立误差会产生总方差 $n\sigma^2$，因此标准差为 $\sqrt{n}\sigma$。对于具有 $n = 8$ 个梯度累积步数的典型训练，我们预计总相对误差约为 3%——这远在容差范围内，此时随机梯度下降固有的随机性占主导地位。在运行 Qwen2.5-0.5B 的 A100 GPU 上，这意味着梯度幅值的准确度约为 10^{-4}，而典型权重更新的量级为 10^{-2}。
让我们准确地追踪使用 FP8 进行一次前向-反向传播时发生的全过程。 考虑一个权重为 W 的单个线性层（以 FP8 E4M3 存储），该层接收输入 x（同样量化为 FP8 E4M3）。步骤 1：我们将 W 和 x 乘以各自的缩放因子进行反量化，生成 BF16 值。步骤 2：使用 Tensor Cores 计算矩阵乘法 Wx，其内部以 FP32 进行累积，但输出为 BF16。步骤 3：将输出重新量化为 FP8 E4M3，以便在传递到下一层之前存储。在反向传播期间，步骤 4：我们以 FP8 E5M2 加载梯度 ∇_y，并对其进行反量化。步骤 5：计算 $\nabla_W = \nabla_y x^T$（在 Tensor Core 内部以 FP32 累积）。步骤 6：将 ∇_W 以 FP32 存储到我们的梯度累加器中——精度正是在这里得到保留的。关键在于，FP8 仅用于存储和通信，从不用于决定最终梯度值的实际累积。

S16.2 延迟缩放分析。**缩放因子面临的问题是：FP8 只能表示最大为 448（E4M3）或 57,344（E5M2）的值，因此，任何超出此范围的值都必须在量化前缩小，并在反量化后再放大。**如果缩放因子选择不当，就会发生上溢（值超出可表示范围，变为无穷大），或者精度利用不足（值集中在零附近，损失有效位）。一种显而易见的解决方案是根据当前张量的最大值计算缩放因子，但在实践中行不通，因为这会导致缩放因子在各次迭代间剧烈振荡，反而放大量化噪声，而非将其减弱。
生产环境部署 FP8 时，一个最反直觉的发现是，与使用历史统计数据的“延迟”缩放相比，“即时”缩放（根据当前张量的最大值计算缩放因子）往往表现更差。原因颇为微妙：即时缩放可能导致缩放因子振荡，从而放大量化噪声；而基于历史数据的缩放则能提供时间平滑效果，稳定训练动态。

算法 27 基于 Amax 历史记录的延迟缩放

维护： amax 历史记录[32]（循环缓冲区）
前向传播： 当前 amax ←max(|x|) amax 历史记录.append（当前 amax）
缩放因子 ←max（amax 历史记录）/ FP8_最大值 x fp8 ← 量化（x/ 缩放因子）
返回： x fp8，缩放因子

该算法通过循环缓冲区维护近期最大绝对值（amax），并根据此历史窗口中的最大值计算缩放因子。这种方法有三个关键特性：（1）它永远不会低估所需的缩放因子，从而避免上溢；（2）它能适应历史窗口长度内的分布偏移；（3）它能平滑瞬态尖峰，避免由此造成不必要的精度损失。

为什么延迟缩放比即时缩放效果更好？ 设想数据中出现一个离群值时会发生什么——例如，某个激活值异常高的词元嵌入。使用即时缩放时，缩放因子会骤升以适应这个离群值，导致张量中的其他所有值都因精度损失过大而被量化（它们此时都远低于可表示范围上限）。下一次迭代中，离群值消失，缩放因子便会下降；而之前被过度量化的值此时又能得到正确量化。这种振荡会持续不断，量化噪声也随缩放因子的不稳定性而波动。使用包含 32 个元素历史记录的延迟缩放时，单个离群值对缩放因子决策的影响仅为 1/32，从而抑制了振荡。**我们的假设是**：32 元素历史记录之所以有效，是因为其时间跨度大致覆盖了训练过程中典型激活值分布偏移的时间常数，从而自然地对缩放因子的动态变化进行低通滤波。

命题 *25*（最优历史长度）DeepSeek V3 发现，最优历史长度为 32（默认值为 1024）：

1. 更快地适配分布偏移

2. 降低缩放计算的开销

3. 对精度的影响极小

我们根据 DeepSeek V3 的实证结果选择了 32 的历史长度，因为它在偏差-方差权衡中达到了最佳平衡。较短的历史（例如 1–8）能快速响应分布变化，但会导致缩放因子出现高频振荡。较长的历史（例如 1024，即 NVIDIA 的默认值）能提供稳定的缩放因子，却无法适应训练动态的变化，例如学习率调度发生改变，或模型进入新的训练阶段。32 步的窗口相当于在现代硬件上约 1–2 秒的训练时间，既能充分平滑，又能及时响应状态变化。在我们的 A100 基准测试中，与即时缩放相比，这种方法将 FP8 相关的损失尖峰减少了 73%，每个张量仅增加 128 字节的状态开销（历史缓冲区中的 32 个 FP32 值）。

分块缩放与逐张量缩放是另一项关键的设计选择。 逐张量缩放为整个权重矩阵或激活张量使用单个缩放因子，实现简单，但当张量不同区域的幅度相差悬殊时，效果不理想。分块缩放（DeepSeek V3 所采用的方法）为每 128 个元素的块分别计算缩放因子，因此，第一层的嵌入（通常幅度较大）可以使用不同于最后一层输出投影（通常幅度较小）的缩放因子。我们的实现对权重矩阵使用 128×128 块，对激活值使用 1×128 块，与 Tensor Core 操作的自然分块方式相匹配。其开销不大：对于一个 494M 参数的模型，分块缩放会增加约 3.9MB 的缩放因子存储空间（每 128 个元素对应一个 FP32 缩放因子），而逐张量缩放仅需 0.03MB；但根据与 BF16 基线的梯度余弦相似度测量，分块缩放可将量化误差降低 2.3 倍。

S17. 完整误差分析。对于 Qwen2.5-0.5B，单个训练步骤涉及约 3 万亿次浮点运算，每次运算都会引入舍入误差。 混合精度训练的奇迹在于，这些误差大多会相互抵消，而不是不断累积——但前提是我们必须正确设计精度预算。本节将详细剖析精度损失发生的位置，解释为何某些操作对误差敏感而另一些操作能够容忍误差，并说明 Chronicals 如何分配精度，在保持训练稳定性的同时最大化吞吐量。

了解训练过程中精度损失发生的位置，对于调试数值不稳定性以及制定明智的混合精度策略至关重要。在 FP32 中“可以直接运行”的训练流水线，切换到混合精度后可能会灾难性失败——这并非由于某个单独的操作，而是因为误差会在前向传播—反向传播—更新循环中以乘法方式不断复合。本节将给出完整的精度预算，准确展示数值误差从何处进入，以及如何传播。

关键洞见在于，并非所有操作都是同等创建的。 矩阵乘法天生具有“自平均”特性——单个乘积中的误差在跨越数千个元素求和时往往会相互抵消。归一化操作（RMSNorm、softmax）涉及对幅度相近的值进行减法，因此容易受到灾难性消去的影响。损失计算涉及对较小概率取对数，而这在低精度下可能发生下溢。通过分析每种操作对误差的敏感性，我们可以仅在真正需要的地方选择性地采用更高精度，从而同时实现 FP8/BF16 的速度与 FP32 的稳定性。

S17.1 数值精度预算。每次浮点运算都会引入舍入误差，但其幅度会因格式和操作类型而相差数个数量级。下表展示了我们在通义千问 Qwen2.5-0.5B 训练中进行的实测结果，反映了全精度参考实现与实际混合精度计算之间的相对误差 ϵ。了解这些误差界有助于从业者识别精度链中的“最薄弱环节”。

表11。训练流水线中的精度损失来源

操作	精度	误差界
注意力（BF16）	BF16	$\epsilon \approx 10^{-3}$
矩阵乘法（BF16）	BF16	$\epsilon \approx 10^{-3}$
交叉熵（FP32）	FP32	$\epsilon \approx 10^{-7}$
优化器（FP32）	FP32	$\epsilon \approx 10^{-7}$
梯度累积（FP32）	FP32	$\epsilon \approx 10^{-7}$
FP8 前向传播	E4M3	$\epsilon \approx 10^{-2}$
FP8 反向传播	E5M2	$\epsilon \approx 10^{-2}$

表格揭示了一个关键发现：FP8 运算带来的误差比 FP32 运算高出 10,000 倍，因此成为数值噪声的主要来源。然而，误差预算并非简单相加——早期层中的误差会传播，并可能在后续计算中不断放大。我们之所以将交叉熵和优化器运算保留在 FP32 中，正是因为这些是“误差汇聚点”，早期层累积的不精确性可能会在此严重影响最终梯度

计算。在这些关键位置， 10^{-7} 精度就像一道数值“防火墙”，防止 FP8 误差破坏权重更新。

为什么即使其他所有内容都使用低精度，我们仍要将优化器状态保留为 FP32? 答案在于理解误差累积的时间尺度。前向与反向传播没有状态——误差不会在训练步骤之间持续存在。相比之下，优化器状态会在数千个步骤中累积信息。Adam 的动量项$m_t = \beta_1 m_{t-1} + (1-\beta_1)g_t$在 $\beta_1 = 0.9$的情况下，意味着 100 步前的信息仍会对当前动量估计贡献 0.003 %。经过 10,000 个训练步骤，FP16 动量累积会引入约 0.5% 的漂移；FP32 则能将漂移控制在 10^{-5}% 以下。对于拥有 4.94亿个参数的 Qwen2.5-0.5B，FP32 优化器状态会额外占用 3.96 GB 内存——这是保障训练稳定性的一项值得投入。

S17.2 Kahan 求和实现。**在长时间训练运行中，梯度累积问题虽不易察觉，却会造成严重影响**。考虑将 8 个微批次的梯度累积起来，其幅度约为 10^{-4}。在 FP32 中，机器精度为$\epsilon \approx 1.2 \times 10^{-7}$，因此将 10^{-4}加到值为 8×10^{-4}的累积和中，每次加法都会损失约 10^{-7}。经过 100 万个训练步骤，这种误差会累积到约 10% 的梯度损坏程度，足以明显降低最终模型质量。Kahan 求和会跟踪一个补偿项，用于捕获这些舍入误差，并在后续加法中将其重新纳入计算，从而将总误差从 $O(n\epsilon)$降低至 $O(n\epsilon^2)$。
在万亿级词元上进行训练时，即使使用 FP32 进行梯度累积，也可能遭遇灾难性消去——也就是将较小的梯度加到较大的累积和上时，由于较小数值的有效位超出可表示范围，导致精度丢失。标准解决方案 Kahan 求和会跟踪一个持续更新的补偿项，用于捕获丢失的位，并在后续迭代中将它们重新加回来。

算法 28 用于梯度累积的 Kahan 求和

1: **初始化：** sum $\leftarrow 0$, c $\leftarrow 0$ (补偿项) 2: **对于** 梯度分块中的梯度 **执行**3: y $\leftarrow$ grad $-$ c {消除之前的误差}4: t $\leftarrow$sum $+$ y { 临时和}5: c $\leftarrow$ (t $-$ sum) $-$ y {新的补偿项}6: sum $\leftarrow$t 7: **循环结束**8: **返回** sum

该算法通过维护补偿变量 c 来捕获每次加法产生的舍入误差。第 3 行中，我们从新的梯度中减去之前的补偿项。第 5 行中，我们将新补偿项计算为期望加上的值（ y ）与实际加上的值（ $(t - \text{sum})$ ）之间的差。编译器不会将这项看似冗余的计算优化掉，因为浮点运算不满足结合律。

让我们通过一个具体示例来进行说明。假设 sum $= 1.0$和 $c = 0.0$。我们希望加上梯度 $= 1.19 \times 10^{-7}$（一个典型的小梯度）。步骤 1： $y = 1.19 \times 10^{-7} - 0 = 1.19 \times 10^{-7}$。步骤 2： $t = 1.0 + 1.19 \times 10^{-7} = 1.0$（该梯度相对于总和而言小于机器精度，因此在 FP32 中无法表示）。步骤 3： $c =$（$1.0 - 1.0$）$- 1.19 \times 10^{-7} = -1.19 \times 10^{-7}$。此时，补偿项会记录“丢失”的梯度。当我们加上下一个梯度时，步骤 1 变为 $y =$ 梯度$_2$ $-$ （-1.19×10^{-7}） $=$ 梯度$_2 + 1.19 \times 10^{-7}$，从而有效恢复丢失的精度。经过数百万次加法后，与朴素累积相比，Kahan 求和可保留约 7 位额外的有效数字。
与成对求和等替代方案相比，我们选择了 Kahan 求和，因为它只需 $O(1)$ 额外存储空间（每个累加器仅需一个补偿值），同时可提供 $O(n \cdot \epsilon_{\text{mach}})$ 误差界，而朴素方法仅提供 $O(n^2 \cdot \epsilon_{\text{mach}})$。对于包含数万亿词元、数百万次梯度累积步数的训练运行，这意味着可以保持 7 位有效数字的精度，而不是因累积舍入误差而可能完全丧失精度。在 A100 GPU 上，其开销可以忽略不计：每个梯度元素仅额外增加约 3 次浮点运算，且完全被内存带宽限制所掩盖。我们的 FP8GradScaler 类在 use kahan summation=True 时，会对所有梯度累积使用 Kahan 求和。

S18. 分布式训练注意事项。多 GPU 训练的根本瓶颈并非计算，而是通信。一个 70亿参数模型使用 BF16 权重时，每个训练步骤需要进行 14 GB 的梯度同步。在 InfiniBand 的 400 Gbps（50 GB/s）带宽下，这需要 280 毫秒，比前向-反向传播本身还要长（在 8 个 A100 上约为 200 毫秒）。如果不将通信与计算重叠，分布式训练实际上会比单 GPU 训练更慢。本节将解释 FSDP 和通信重叠如何解决带宽问题，以及为什么内存公式 Memory/GPU $=(M_{\text{params}} + M_{\text{grads}} + M_{\text{opt}})/N + M_{\text{activations}}$ 揭示了分片粒度与通信开销之间隐藏的权衡。

扩展到单个 GPU 之外会带来一系列新的挑战：通信带宽会成为关键瓶颈，设备间的内存碎片化会使优化更加复杂，而同步开销可能占据训练时间的大部分。现代分布式训练框架（如 FSDP（全分片数据并行））通过对模型

状态进行分片并分布到各个 GPU 上来应对这些挑战，但要实现最高性能，就必须理解内存效率与通信开销之间的权衡。

为什么不能简单地在每个图形处理器上复制模型，然后对梯度求平均? 这种称为数据并行（DP）的方法适用于小型模型，但无法处理大型模型。一个70亿参数模型需要84 GB内存（FP32下，14 GB参数 + 14 GB梯度+ 56GB优化器状态），超出了任何单个图形处理器的容量。FSDP通过分片解决了这一问题：每个图形处理器只保存模型状态的$1/N$，按需收集参数，并在使用后立即丢弃。其关键在于，我们从不需要同时保存完整模型——前向与反向传播每次只处理一层。

S18.1 FSDP 集成。FSDP背后的关键洞见是，在训练的任意时刻，我们只需要当前正在计算的层的完整参数。通过即时收集参数，并在使用后立即丢弃，FSDP将模型状态的每个图形处理器内存占用从$O(|\theta|)$降低到$O(|\theta|/N)$，从而能够训练原本无法装入内存的模型。

定义 *34*（全分片数据并行）FSDP 会将模型参数、梯度和优化器状态分片到各个 GPU 上：

$$\text{Memory/GPU} = \frac{M_{\text{params}} + M_{\text{grads}} + M_{\text{opt}}}{N_{\text{GPUs}}} + M_{\text{activations}} \tag{132}$$

对于一个采用 BF16、包含 70亿参数模型的训练，内存需求分析结果表明，参数需要 14 GB，梯度还需要 14 GB，而优化器状态（FP32 动量和方差）需要 56 GB，总计 84 GB，远超任何单个 GPU 的容量。采用 8 路 FSDP 分片后，每个 GPU 仅需容纳 10.5 GB 的模型状态，因此可以在 40GB A100 上进行训练。

算法 29FSDP 前向传播

1: 全收集(参数) {收集完整参数}2: 输出 ←层(输入) 3: 释放(参数) {丢弃已收集的参数}4: **返回** 输出

该算法展示了 FSDP 的核心机制：在计算每一层之前，我们会从所有 GPU 收集参数（全收集）；计算完成后，立即释放收集到的参数。反向传播采用类似模式，但会增加一个归约散布操作来分发梯度。与 DeepSpeed ZeRO-3 等替代方案相比，我们选择 FSDP，是因为 PyTorch 的原生实现与 torch.compile 和 CUDA 图的集成更好，在我们的基准测试中可将调度开销降低多达 15%。

让我们跟踪一下在 8 个 GPU 上执行单次 FSDP 前向传播期间的内存用量。最初，每个 GPU 持有 10.5 GB（总计 84 GB 模型状态的 1/8）。处理第 0 层时：步骤 1：All-gather 从全部 8 个 GPU 收集该层的参数——此时每个 GPU 都拥有完整的层参数（对于包含 32 层的 70亿参数模型，约为 0.35 GB）。步骤 2：计算第 0 层的前向传播。步骤 3：释放已收集的参数，恢复到基线内存用量。该操作期间的峰值内存为$10.5 + 0.35 +$激活值 GB。关键限制在于，激活值不会被分片——它们会随批次大小和序列长度增长，并且通常占据内存用量的主要部分。对于批次大小为 4、序列长度为 4096 的 70亿参数模型，激活值在每个 GPU 上约占 8 GB，因此总峰值内存约为 19 GB，完全处于 A100 的 40 GB 容量之内。

S18.2 通信优化。朴素的梯度同步实现会等待整个反向传播完成后才开始通信——浪费了本可与数据传输重叠的 GPU 周期。现代分布式训练会将全归约操作与反向计算重叠，从而将大部分通信延迟隐藏在有用的计算之后。命题 *26*（全归约重叠）将全归约与反向计算重叠：

$$T_{\text{total}} = T_{\text{backward}} + \epsilon_{\text{sync}} \tag{133}$$

相对于顺序执行：$T_{\text{total}} = T_{\text{backward}} + T_{\text{allreduce}}$。

对于 8 个 GPU：训练时间减少 20-30%。

该命题量化了通信重叠的收益：无需在反向传播之后按顺序承担完整的全归约开销（$T_{\text{allreduce}}$），重叠执行会将其降低为仅有少量同步开销（ϵ_{sync}）。在通过 NVLink（双向 600 GB/s）连接的 8 个 A100 GPU 上，一个 0.5B 模型的梯度（BF16 格式下为 1 GB）执行全归约大约需要 3ms。没有重叠时，每个步骤都会增加 3ms；有了重叠后，通信会在较早层的反向计算期间完成，仅贡献 $\epsilon_{\text{sync}} \approx 0.3$ms 的阻塞时间。对于通过 InfiniBand（400 Gbps）连接的更大规模集群，节省的时间更加显著——通信重叠对于维持合理的扩展效率变得不可或缺。

通信重叠这一技巧之所以有效，是因为反向传播会按相反顺序处理各层。计算第 31 层的梯度时，我们不再需要第 31 层的参数，因此可以释放这些参数，并开始同步其梯度。等到第 0 层的梯度计算完成时，第 31 层的梯度同步也已完成。这种流水线处理需要

精心编排：我们为计算和通信分别维护独立的 CUDA 流，并通过依赖项确保只有在相应的反向计算完成后，梯度同步才会开始。我们的实现通过配备 NVLink 的 8 张 A100 实现了 92% 的重叠效率，这意味着只有 8% 的通信时间会表现为阻塞。

扩展效率会随集群规模的增大而呈现可预测的下降。对于 N 个 GPU，每次全归约所需的总带宽为梯度大小的 $2(N-1)/N$ 倍。在 8 个 GPU 上： $2 \times 7/8 = 1.75\times$ 每个 GPU 的梯度大小。在 64 个 GPU 上： $2 \times 63/64 = 1.97\times$。渐近极限为 $2\times$，这意味着随着集群规模增大，全归约的带宽需求会翻倍。我们的基准测试显示，对于 Qwen2.5-0.5B，8 个 GPU 上的扩展效率为 95%，32 个 GPU 上为 87%，64 个 GPU 上为 78%。对于计算时间占主导地位、通信相对次要的大型模型，扩展效率会有所提升：一个 70亿参数模型即使在 64 个 GPU 上也能达到 91% 的效率，因为更长的计算时间为通信重叠提供了更多机会。

S19. 代码示例与实现。理解一项优化与正确实现它之间的差距，往往正是性能损失的根源。一个忘记缓存标准差倒数的 RMSNorm 内核，会在反向传播期间重新计算 $1/\sqrt{\text{方差}}$，浪费计算周期。一个调用 tensor.item() 的优化器会同步图形处理器和中央处理器，使整个流水线停顿数微秒；这些时间在训练过程中不断累积，最终可能达到数分钟。本节提供可用于生产环境的代码，并附有注释，不仅解释每一行代码的作用，还说明为何选择这种特定方法，以及为何弃用了其他替代方案。
从理论走向实践，不仅需要了解应当应用哪些优化，还需要了解它们如何组合，以及需要注意哪些细微的交互影响。本节提供了展示 Chronicals API 的生产就绪代码示例，并解释了影响实现的设计决策。每个示例都直接取自我们的基准测试基础设施，并已在 A100 和 H100 硬件上完成验证。

一个关键的实现洞察是：GPU-CPU 同步是隐藏的性能杀手。.item()、.cpu() 的每次调用，或任何依赖张量值的 Python 控制流，都会迫使图形处理器等待中央处理器（或反之）。我们的融合 AdamW 优化器使用单个 Python 整数步数计数器，而不是 CUDA 张量，从而消除了每步同步。梯度范数完全在图形处理器上计算，并直接在 GPU 内核中用于裁剪——唯一的同步点是每 100 步进行日志记录时可选的一次 .item() 调用。仅这一设计选择，就使我们相比朴素实现获得的加速比提升了 8%。

S19.1 完整训练脚本。以下示例展示了启用所有 Chronicals 优化的最小可行训练循环。关键的架构决策是可组合性：每项优化（FlashAttention、融合内核、打包、检查点）都可以独立启用，使从业者能够逐步采用优化，并隔离性能回归。在调试性能问题时，我们建议一次启用一项优化并测量吞吐量——内核融合的乘法效应意味着，各项优化之间的交互可能并不直观。

算法 30 Chronicals 训练 API

```
1: 导入： ChronicalsTrainer、LoRAPlusAdamW、SequencePacker 2: 3: {使用所有优化功能初始化训练器}4: trainer ←ChronicalsTrainer( 5: 模型名称="Qwen/Qwen2.5-0.5B", 6: 使用 Flash Attention=True, 7: 使用 Liger 内核=True, 8: 使用序列打包=True, 9: 梯度检查点="selective", 10: 精度="bf16") 11: 12 { + 16 } : 配置 13 ← =10^−4 LoRA 优化器（B 矩阵的学习率为 x 倍）： 优化器 LoRAPlusAdamW(model, lr, 学习率比=16) 14: 15: {使用序列打包进行训练}16: packer ← SequencePacker(最大长度=2048,策略="BFD") 17: trainer.train(dataset, optimizer, packer, epochs=3, 批次大小=8)
```

S19.2 LoRA+ 优化器实现。LoRA+ 优化器（5）基于理论分析，为 A 矩阵和 B 矩阵采用不同的学习率。理论分析表明 $\eta_A = O(n^{-1})$ 以及 $\eta_B = O(1)$，可实现 1.5–2 倍更快的收敛。

算法 31 LoRA+参数组构建

输入： 命名参数、基础学习率 η、比例 r （默认值为 16）
LoRA A ← {}、LoRA B ← {}、其他 ← {}
对于（名称，参数）在命名参数中 **执行**
    如果 名称匹配“*.LoRA A*”**则**
        LoRA A.add(参数)
    否则，如果 名称匹配“*.LoRA B*”**则**
        LoRA B.add(param)
    else
        other.add(param)
    结束 if
结束 for
返回： [
    {参数：LoRA A，学习率：η},
    {参数：LoRA B，学习率：$\eta \cdot r$}, {16倍高学习率}
    {参数：其他，学习率：η}]

S19.3 使用最佳适应递减算法进行序列打包。序列打包器使用最佳适应递减算法（BFD）装箱，$11/9 \cdot$ 最优解 $+ 6/9$ 近似比（25）。其主要特性包括兼容 CUDA 图的固定输出形状，以及通过`cu seqlens`支持 FlashAttention varlen。

算法 32最佳适应递减装箱

输入： 长度[], 最大容量 C
排序后的物品 ←sort(lengths, descending=True)
箱 ← []
对于(idx, len)，遍历已排序的物品**执行**
    最佳箱 ←无 , 最佳余量 ← ∞
    对于 箱，遍历所有箱 **执行**
        如果 箱.remaining $\geq$ len 且箱.remaining $<$ 最佳余量 **那么**
            最佳箱 ← 箱
            最佳剩余空间 ← 箱.剩余空间
        结束条件判断
    结束循环
    如果最佳箱 $\neq$无**则**
        最佳箱.添加(idx, len)
    else
        箱.append(new 箱(capacity=C, 项目=(idx, len)))
    结束 if
结束 for
返回： 箱

S19.4 带 ***RSTD*** 缓存的融合 ***RMSNorm***。我们的 RMSNorm 实现会缓存标准差倒数（RSTD），以高效执行反向计算，避免代价高昂的平方根重新计算。

算法 33 Triton RMSNorm 前向内核

```
1: @triton.jit 2: 行索引 ← tl.program i
d(0) 3: x ← 加载函数(X ptr + 行索引 × stride)
4:5 { 32 } : 为确
6
保稳定性，使用 FP 进行均方根计算：均值平方
tl.sum(x x) / N 7: 均方根 ← tl.sqrt(均值平方 + ϵ)
8: rstd ← 1.0 / 均方根 { 为反向传播缓存！ }9:
10: {归一化并缩放}11: x norm ←x × rstd
12: w ←加载函数(W ptr) 13: y ←x norm ×
w 14: 15: tl.store(Y ptr + 行索引 × stride, y)
16: tl.store(RSTD ptr + 行索引,rstd) {微小：每
行 4B}
```

S19.5 自定义 ***Triton*** 内核示例。以下内核展示了用于交叉熵的在线 softmax，通过分块处理词汇表，避免物化完整 logits 张量：

算法 34 Triton 在线 Softmax 交叉熵

```
1: @triton.jit 2: 行索引 ←tl.program id(0) 3: 目标
←加载函数(目标指针 +行索引) 4: 5: {在线 softmax 状
态}6: m ← −∞、d ← 0、目标 logit ← 07: 8: 对于 范围
(0, 词汇表, 块大小) 执行 9: logits ← 加载函数(logits 指
针 + 行索引 × 词汇表 + offs) 10: 分块最大值 ←
tl.max(logits) 11: m 新值 ← max(m,分块最大值) 12:
d ← d × exp(m − m 新值) +sum(exp(logits − m 新
值)) 13: m ←m 新值 14: 如果 offs ≤ 目标 < offs + 块大
小 那么15: 目标 logit ← logits[目标 − offs]16: 结束
if17: 结束 for18: 19: 损失 ← log(d) + m − 目标 logit
20: tl.store(损失指针 + 行索引,损失)
```

S20. 基准测试可复现性详情。相同的代码运行速度可能相差 40%，具体取决于程序员无法察觉的因素。CUDA 驱动程序版本 535 与 545 会改变内核调度。A100 在 85°C 下运行时会降频至 1095 MHz，而不是 1410 MHz。PyTorch 的内存分配器会根据分配历史以不同方式产生内存碎片。未对这些变量进行控制的基准测试，所得数值虽然对特定测试准确，却可能误导人们将其视为普遍结论。本节记录了我们控制的每个变量，并说明各变量的重要性。

可复现性是可信基准测试的基石，但在 GPU 加速机器学习领域，实现可复现性仍出乎意料地困难。CUDA 版本、驱动程序设置，甚至热条件的细微差异，都可能导致测得的性能波动 10%–20%。本节提供了复现我们基准测试所需的完整规格，并介绍我们用于控制混杂变量、确保统计严谨性的方法。

我们的基准测试协议消除了三个最常见的方差来源。首先是热节流：测量前，我们会运行 100 个预热步数，使图形处理器达到热平衡，然后确认整个基准测试过程中时钟频率保持稳定（波动不超过 2%）。其次是内存碎片化：我们会在每次运行之间清空 CUDA 缓存，并采用确定性分配模式。第三是即时编译：Triton 和 torch.compile 会缓存编译后的内核，因此首次运行的性能与稳态性能不同；我们始终报告缓存预热后的稳态性能。

S20.1 硬件规格。为确保结果可复现，我们在标准化云实例上进行了所有基准测试。我们选择 A100-40GB SXM4 配置，是因为它代表了生产环境中部署最广泛的训练硬件，因此我们的结果可直接应用于实际部署。所有热节流均已禁用，并且我们确认基准测试期间睿频时钟始终稳定。**为什么选择 A100 而不是 H100？** 虽然 H100 的绝对性能更高，但截至 2025年1月，A100 仍是云环境中的主流训练 GPU（在主要云服务提供商提供的 GPU 小时数中约占 70%）。在 A100 上奏效的优化通常也适用于 H100，但反过来并非总是如此，因为 H100 具有 TMA 和线程束专门化等专属特征。

表12。 基准测试硬件配置

组件	规格
GPU	NVIDIA A100-40GB SXM4
GPU 内存	40 GB HBM2e
内存带宽	1.6 TB/s
FP32 TFLOPs	19.5
TF32 TFLOPs	156
BF16 TFLOPs	312
INT8 TOPS	624
CPU	AMD EPYC 7V13 64核
系统内存	512 GB DDR4
NVMe 存储	2TB，读取速度 7GB/s
CUDA 版本	12.1
PyTorch 版本	2.4.0
Triton 版本	2.3.0

S20.2 基准测试 ***k*** 脚本。我们的基准测试命令行界面提供标准化接口，以进行可复现测量 :

算法 35基准测试命令行界面

```
{全量微调基准测试}
Python benchmark.py
    –model Qwen/Qwen2.5-0.5B
    –mode 全量微调
    –batch size 16 –seq len 512
    –预热步数 10 –基准测试步数 100
    –使用 CUDA 事件 –验证梯度

{使用 LoRA 进行 LoRA 基准测试+}
python benchmark.py
    –模型 Qwen/Qwen2.5-0.5B
    –模式 lora –LoRA 秩 32 –LoRA Alpha 64
    –使用 LoRA+ –学习率比 16
    –批次大小 8 –验证梯度
```

S20.3 验证检查。每次基准测试运行都包含以下正确性检查，以确保实际进行训练：

算法 36 训练正确性验证

1: **函数**验证训练(model, 批次): 2: 3 { 1 } : 检查：梯 4 ← 0 0 度范数非零：总范数。 5: **对于** p in model.parameters() **执行** 6: **如果** p.grad ≠ None**则** 7: 总范数 ← 总范数 + ∥p.grad∥^{2}8: **结束如果**9: **结束循环** 10: **断言** 总范数$^{0.5}$ > 0 {“梯度范数为零！”}11:12 { 2 } : 检 13 ← 查：所有参数均可训练：可训练计数(p | p.requires grad) 14: 总数 ← count(model.parameters()) 15:**断言**可训练 ==总数 {100% 必须可训练}16: 17: {检查 3：损失为有限值}18: 损失 ← 计算损失(model,批次) 19: **断言** isfinite(损失) 20: **返回**True

-

S21. 未来工作。本文中的优化代表了容易取得的成果——通过系统地应用成熟技术，便能获得显著提升。 下一代改进需要攻克更棘手的问题：H100 的新硬件特性尚未能通过 Triton 使用；在异构 GPU 集群上进行分布式训练；以及针对混合专家等新兴架构提升训练效率。本节将阐述我们的技术路线图，并列出具体的性能目标和预估的实现复杂度。

1. **借助 FlashAttention-3 实现 H100 FP8 支持**：H100 图形处理器引入了 TMA（张量内存加速器），用于异步数据移动，并支持线程束专门化，从而实现计算与内存操作的重叠。FlashAttention-3 利用这些特性，达到 740 万亿次浮点运算（H100 FP8 理论峰值的 75%）；相比之下，FlashAttention-2 在相同硬件上达到 480 万亿次浮点运算。我们的计划是：将线程束专门化注意力内核移植到 Triton（预计 Triton 3.0 将于 2025 年第二季度加入 TMA 支持），与现有 FP8 基础设施集成，并验证其与当前 FlashAttention-2 实现的数值等价性。**预期影响**：注意力加速 1.5 倍，使 H100 上的端到端训练加速达到 6.5 倍。

2. **混合并行分布式训练**：目前的 Chronicals 针对单 GPU 训练进行了优化。扩展到多 GPU 需要集成 FSDP 以进行模型分片，采用张量并行处理大型注意力层，并通过序列并行支持长上下文训练。关键挑战在于操作分布到不同设备后，如何维持内核融合效率——简单的分布式处理会破坏融合边界，抵消单 GPU 优化带来的收益。我们的方法：实现“分片感知融合”，在分片边界内融合操作，并在边界处采用高效的集合通信操作。**目标**：在 8 个 GPU 上为 70 亿参数模型实现 85% 的扩展效率，达到 DeepSpeed ZeRO-3 的水平，同时保留 Chronicals 的单 GPU 优化。

3. **采用融合反量化的 INT4/INT8 QLoRA**：QLoRA 的内存效率十分出色，但会产生反量化开销——每次矩阵乘法都需要将 4 位权重扩展为 16 位，从而增加约 30% 的延迟。我们计划实现融合反量化内核，在矩阵乘法过程中于静态随机存取存储器（SRAM）中完成 4 位到 16 位的扩展，避免将中间值写入高带宽内存（HBM）。**预期影响**：消除反量化开销，使 QLoRA 的延迟与全量微调相当。

4. **用于推理的推测解码**：训练是我们目前的重点，但推理效率对于迭代开发也很重要。推测解码使用一个小型“草稿”模型提出多个词元，再由大模型并行验证。与 Chronicals 的 KV缓存管理集成后，兼容的模型对有望实现 2-3 倍的解码加速比。

5. **混合专家训练**：像 Mixtral 这样的 MoE模型，每浮点运算都能实现更高的质量，但也带来了专家路由开销和负载均衡方面的挑战。我们计划采用融合专家分发内核，避免朴素实现中的散射-收集模式，并与序列打包集成，确保打包序列中的专家利用率均衡。

6. **上下文扩展至 128K+**：当前的 RoPE 实现支持最长 32K 的上下文，且性能合理。扩展到 128K+ 需要：(a) 采用 YaRN 风格的位置插值来处理 RoPE；(b) 使用稀疏注意力模式（局部 +全局），以将内存复杂度维持在 O(n)；(c) 使用环形注意力，将注意力计算分布到多个 GPU 上。我们的目标是：在 128K 上下文下，相较于 32K，延迟增幅低于 2 倍。

7. **视觉语言模型训练**：将 Chronicals 扩展到多模态模型需要处理异构的序列长度（图像由固定大小的图像块组成，文本由可变长度的词元组成），以及模态之间高效的交叉注意力。我们计划利用现有的序列打包基础设施，高效地打包图像块和文本词元。

S22. 完整的 Triton 内核库。一个内核提速 2 倍与提速 10 倍之间的差异，通常源于一个设计决策：是将中间值保存在静态随机存取存储器（快速）中，还是将其溢出到高带宽内存（缓慢）中。本节将逐一介绍每个 Triton 内核的设计，不仅解释代码的功能，还会说明选择每项优化的原因。每个内核都遵循相同的模式：识别内存受限操作，将其融合以尽量减少高带宽内存流量，并利用 Triton 的块编程模型最大化静态随机存取存储器的复用。

本节提供 Chronicals 中所使用的全部 Triton 内核的完整实现。每个内核都已与参考 PyTorch 实现进行对比验证，以确保数值正确性，并使用 NVIDIA Nsight 进行分析，以确认内存访问模式与理论预测相符。

为什么选择 Triton 而不是 CUDA？ Triton 针对我们的使用场景提供了三项优势：(1) 自动处理线程块大小设置和内存合并访问，减少复杂融合内核中的错误；(2) 只需极少的代码改动，即可在 NVIDIA、AMD 和 Intel GPU 之间移植；(3) 针对不同输入大小进行即时编译和自动调优。相较于手工优化的 CUDA，其性能损失通常低于 10%；而 Triton 在开发速度方面的优势十分显著——我们仅用 3 周就完成了整个内核库，而实现功能等效的 CUDA 版本预计需要 3 个月。

S22.1 融合线性交叉熵内核。**交叉熵损失是大语言模型训练中最大的单项内存消耗。**对于包含 128K 个词元的词汇表、批次大小为 8 且序列长度为 2048 的情况，logits 张量大小为 $8 \times 2048 \times 128K \times 2 = 4.2$ GB——通常会超过图形处理器的总内存。标准方法（先计算 logits，再进行 Softmax，最后计算损失）需要将整个张量实体化。我们的融合内核利用在线 softmax，按适合静态随机存取存储器的分块处理词汇表，从而无需实体化完整 logits 即可计算损失。

算法 37 融合线性交叉熵与语言模型头

```
@triton.jit
def fused linear cross entropy kernel(
    hidden ptr, weight ptr, 目标 ptr, 损失 ptr,
    B, N, H, V,
    步幅 hb , 步幅 hn , 步幅 hh ,
    步幅 wv , 步幅 wh ,
    块高: tl.constexpr, 块 V: tl.constexpr):

    # 获取行索引（批次 * 序列位置）
    行索引 =tl.program id(0)
    批次索引 = 行索引 // N
    序列索引 =行索引 % N

    # 加载此位置的目标
    目标 = tl.load(target ptr + 行索引)
    如果目标 == -100:
        tl.store(loss ptr + 行索引, 0.0)
        返回

    # 加载隐藏状态
    h 偏移量 =范围生成函数(0, 块高)
    h 掩码 =h 偏移量 <H
    隐藏状态 = 加载函数(
        隐藏状态指针 +批次索引 * 步幅 hb +序列索引 * 步幅 hn + h 偏移量,
        掩码=h 掩码，其他=0.0)

    # 在线 softmax 状态
    m =float('-inf')
    d = 0.0
    目标 logit = 0.0

    # 按分块处理词汇表
    对于 v start，范围(0, V, 块 V):
        v offs =v start + 范围生成函数(0, 块 V)
        v 掩码 =v 偏移 <V

        # 计算此分块的 logits：hidden @ W[v start:v end].T
        logits =tl.zeros((块 V,), dtype=tl.float32)
        对于 h block，范围(0, H, 128):
            h 结束 =min(h 块 + 128, H)
            h 分块 =加载函数(
                隐藏状态指针 + 批次索引 * 步幅 hb + 序列索引 * 步幅 hn + h 块 + 范围生成函数(0, 128),
                掩码=范围生成函数(0, 128) < (h 结束 - h 块), 其他=0.0)
            w 分块 =加载函数(
                权重指针 + v 偏移[:, None]* 步幅 wv + h 块 + 范围生成函数(0, 128),
                掩码=(v 掩码[:, None]) & (范围生成函数(0, 128) < (h 结束 - h 块)), 其他=0.0)
            logits +=tl.sum(h 分块 * w 分块, 轴=1)

        # 在线 softmax 更新
        分块最大值 =tl.max(tl.where(v 掩码, logits, float('-inf')))
        m 新 = tl.maximum(m, 分块最大值)
        d = d * tl.exp(m - m 新)
        d = d + tl.sum(tl.where(v 掩码, tl.exp(logits - m new), 0.0))
        m =m new

        # 提取目标 logit
        if v start ≤ 目标 < v start + 块 V:
            目标索引 =目标 - v start
            目标 logit =加载函数(logits + 目标索引)

    # 计算并存储损失
    对数和指数 =tl.log(d) + m
    损失 =对数和指数 - 目标 logit
    tl.store(loss ptr +行索引, loss)
```

在线 softmax 技巧是实现内存效率的关键。传统 Softmax 需要对数据进行两次遍历：一次用于计算最大值（以确保数值稳定性），另一次用于计算指数和。在线 softmax 通过持续维护最大值和归一化常数的运行估计值，并在新分块到达时增量更新，将这两个步骤合并为一次遍历。更新规则 $d_{\text{new}} = d_{\text{old}} \cdot e^{m_{\text{old}} - m_{\text{新值}}} + \sum e^{x - m_{\text{new}}}$ 到达新的最大值时会“修正”之前的总和

已发现。这使我们能够以 256 个令牌的分块处理 128K 词汇表（每个分块在 FP32 格式下为 256 KB），而无需存储完整的 logits 张量。

S22.2 带残差的融合 ***RMSNorm***。**在一个包含 24 层的 Transformer 中，每次前向传播都会执行 48 次 RMSNorm（每层两次：分别在注意力层和前馈神经网络层之前）**。每次 RMSNorm 都是内存受限的：我们读取输入、计算均方根、进行归一化、按学习得到的权重缩放，并写入输出。其算术强度约为 0.5 浮点运算次数/字节，远低于 A100 上 156 浮点运算次数/字节的脊点。通过将 RMSNorm 与其之前的残差加法融合，我们可为每次操作省去一次内存往返，从而将算术强度有效提升一倍，达到 1.0 浮点运算次数/字节。

算法 38带残差加法的融合 RMSNorm

```
1：@triton.jit 2：def fused rmsnorm residual kernel( 3：x ptr,
residual ptr, weight ptr, output ptr, rstd ptr, 4：n rows, n cols, eps, 5：
BLOCK SIZE: tl.constexpr): 6： 7：行索引 = tl.program id(0) 8：offs =
tl.arange(0, BLOCK SIZE) 9：mask =offs < n cols 10： 11：# 加载输入和
残差 12：x = tl.load(x ptr + row idx * n cols + offs, mask=mask, other=0.
0) 13：residual = tl.load(residual ptr + row idx * n cols + offs, mask=
mask, other=0.0) 14： 15：# 添加残差 16：x = x + residual 17： 18：# 存
储更新后的残差（供下一层使用） 19：tl.store(residual ptr + row idx * n
cols + offs, x, mask=mask) 20： 21：# 计算均方根 22：方差 = tl.sum(x *
x, axis=0) / n cols 23：rstd = 1.0 / tl.sqrt(variance + eps) 24： 25：# 加载权
重并应用归一化 26：weight = tl.load(weight ptr + offs, mask=mask,
other=1.0) 27：output = x * rstd * weight 28： 29：# 存储输出 30：
tl.store(output ptr + row idx * n cols + offs, output, mask=mask) 31：
tl.store(rstd ptr + row idx, rstd)
```

一个关键细节：我们会缓存反向传播所需的 RSTD（标准差倒数）。反向传播需要 $1/$均方根来计算梯度。如果不进行缓存，我们就必须重新计算均方根（再次读取输入），从而增加一次完整的内存读取。每行只存储一个 FP32 值（对于包含 2048+ 个浮点数的序列，占用 4 字节），就能减少 99.8% 的反向传播内存读取。这是我们在整个过程中采用的一种模式：找出反向传播所需的值，在前向传播时计算这些值，并以最少的空间存储它们。

S22.3 带缩放的融合 ***Dropout***。**在朴素实现中，Dropout 的开销大得出人意料**。标准方法会生成随机掩码，逐元素相乘，并通过 $1/(1-p)$ 进行缩放。对于算术强度接近于零的操作而言，这需要读取输入、写入掩码（用于反向传播）以及写入输出——共三次内存操作。我们的融合内核使用 Philox 随机数生成器在静态随机存取存储器（SRAM）中生成随机数（给定随机种子时具有确定性），应用掩码，并仅写入输出。在反向传播期间使用相同的随机种子重新生成掩码，从而无需存储掩码。

算法 39 融合式 Dropout Triton 内核

1: @triton.jit 2: def 带缩放的融合 Dropout 内核(3: x 指针, 输出指针, 随机种子, 4: n 个元素, p, 5: 块大小: tl.constexpr): 6: 7: 块索引 = tl.program id(0) 8: 偏移量 = 块索引 * 块大小 + 范围生成函数(0, 块大小) 9: 掩码 = 偏移量 < n 个元素 10: 11: # 加载输入 12: x =加载函数(x 指针 + 偏移量, 掩码=掩码, 其他=0.0) 13: 14: # 使用 Philox 随机数生成器生成随机数 15: 随机数 = tl.rand(随机种子, 偏移量) 16: 17: # 应用 Dropout 掩码 18: 保留掩码 = 随机数 > p 19: 缩放因子 = 1.0 / (1.0 - p) 20: 输出 = tl.where(保留掩码, x * 缩放因子, 0.0) 21: 22: # 存储输出 23: tl.store(输出指针 + 偏移量, 输出, 掩码=掩码)

S23. 全面的复杂度分析。大 O 记号可能会误导 GPU 性能判断，因为它掩盖了相差 1000 倍的常数因子。 在静态随机存取存储器中执行的 $O(N^2)$ 操作，可能比触及高带宽内存的 $O(N)$ 操作更快。本节将进行复杂度分析，但需注意，对于低于 $N = 10^6$ 的操作，相较于渐近行为，内存访问模式更为重要。在适用的情况下，我们会指出操作是计算受限（可从更多浮点运算次数中受益），还是内存受限（可从更好的数据局部性中受益）。

对这些复杂度的实际解读如下：大 O 复杂度相同的操作，其墙钟时间差异与算术强度之比有关。自注意力和 FlashAttention 的浮点运算次数均为 $O(N^2d)$，但 FlashAttention 的内存访问复杂度为 $O(N^2d^2/M)$，而注意力为 $O(N^2d)$。在内存受限的硬件上（大多数训练场景皆如此），FlashAttention 的速度快 d/M 倍；对于 $d = 128$ 和 $M = 192KB$（A100 静态随机存取存储器），这一数值约为 4 倍。

表 13. Chronicals 操作的时间复杂度

操作	标准	Chronicals
自注意力	$O(N^2d)$	$O(N^2d)$
FlashAttention 输入/输出	$O(N^2d)$	$O(N^2d^2/M)$
交叉熵	$O(BNV)$	$O(BNV)$
CCE 内存访问	$O(BNV)$	$O(BNC \cdot V/C) = O(BNV)$
RMSNorm	$O(BNd)$	$O(BNd)$
SwiGLU	$O(BN \cdot 3d_{\text{ff}})$	$O(BN \cdot 3d_{\text{ff}})$
LoRA 前向传播	$O(BN(dk + rk + rd))$	$O(BN(dk + rk + rd))$
AdamW 更新	$O(\|\theta\|)$	$O(\|\theta\|/\text{并行性})$

S23.1 时间复杂度。**为什么 FlashAttention 不会改善时间复杂度？** 该表显示，标准注意力和 FlashAttention 的时间复杂度均为 $O(N^2d)$。这是因为 FlashAttention 并未减少计算量——它执行的浮点运算次数完全相同。改进体现在内存读写，而非算术运算。标准注意力需要 $O(N^2)$ 次高带宽内存访问来存储和读取注意力矩阵；FlashAttention 将注意力矩阵保留在静态随机存取存储器中，将高带宽内存访问次数减少到 $O(N^2/B_q)$，其中B_q 为查询块大小。对于长度为 4096 个词元、 $B_q = 128$的序列，这使内存流量减少了 32 倍。

S23.2 空间复杂度。**空间复杂度决定工作负载是否能装入图形处理器内存——这是会导致训练运行失败的硬性约束。** 下表比较了标准实现与 Chronicals 实现之间的峰值内存使用量。影响最大的优化包括：将注意力矩阵从 $O(BHN^2)$ 缩减至 $O(BHB_qB_{kv})$（FlashAttention），将 logits 从 $O(BNV)$ 缩减至 $O(BNC)$（分块交叉熵），以及将激活值从 $O(LBNd)$ 缩减至 $O(\sqrt{L}BNd)$（梯度检查点）。对于批次大小为 8、序列长度为 2048 的 Qwen2.5-0.5B，这些优化将峰值内存使用量从 16.9 GB 降至 7.2 GB，从而能够在 8GB 消费级 GPU 上进行训练。

表14. 空间复杂度比较

组件	标准	Chronicals
注意力矩阵	$O(BHN^2)$	$O(BHB_qB_{kv})$
logits	$O(BNV)$	$O(BNC)$
优化器状态	$O(2\|\theta\|)$	$O(0.5\|\theta\|)$ 8 位
激活值	$O(LBNd)$	$O(\sqrt{L}BNd)$ 检查点
键值缓存	$O(BNHd)$	$O(BNHd/g)$ 分组查询注意力
LoRA 权重	$O(r(d+k))$	$O(r(d+k))$

S23.3 通信复杂度。**通信复杂度通常是分布式训练中容易被忽视的瓶颈**。单 GPU 性能会随 GPU 浮点运算次数扩展，而多 GPU 性能则受互连限制。下表展示了不同并行策略下每个训练步骤的通信量。关键在于：对于大多数策略，通信量与批次大小无关，这意味着更大的批次可以摊薄通信开销。这就是分布式训练所采用的批次大小为何大于单 GPU 训练的原因——并非出于内存方面的考虑，而是为了维持计算与通信的比率。

对于使用 P 个 GPU 进行的分布式训练：

表1 15. 每个训练步骤的通信复杂度

并行策略	通信量
数据并行	$O(\|\theta\|)$ 全归约
FSDP (ZeRO-3)	$O(\|\theta\|)$ 全收集 + 归约散布
张量并行	$O(BNd)$ 每层
流水线并行	$O(BNd)$ 每个微批次

S24. 数值稳定性分析。**数值不稳定性是训练运行中的隐形杀手**。模型可能已经训练数小时，损失持续下降，却突然飙升至 NaN，原因不明。罪魁祸首通常是以下三种操作之一：Softmax 溢出、交叉熵下溢或梯度爆炸。本节针对 Chronicals 中的每项关键操作，提供严格的数学稳定性保证，不仅解释稳定形式，还说明其有效原因以及仍可能失效的情况。

核心原则是绝不让中间值超出可表示范围。对于 BF16，这意味着将值控制在大约 $[10^{-38}, 3.4 \times 10^{38}]$；对于 FP8 E4M3，范围为 $[2^{-9}, 448]$。诸如 exp(x) 这样的操作很容易超出这些界限——exp(90) $\approx 10^{39}$ 会导致 BF16 上溢，而 exp(7) ≈ 1096 会导致 E4M3 上溢。下面介绍的稳定形式会平移输入，使指数函数值保持有界，同时维持数学等价性。

S24.1 Softmax 数值稳定性。**朴素 Softmax 在大语言模型的 logits 上会彻底失效**。 设词汇表中有 128K 个词元，其中一个词元的 Logit 为 100，其余词元的 Logit 均为 0。朴素 Softmax 会计算 exp(100) $\approx 2.7 \times 10^{43}$——在任何浮点格式下都会得到无穷大。稳定形式会在求指数之前减去最大 Logit，确保 exp 的所有参数均为非正数。

定理 *12* （稳定 Softmax）对于 logits $z \in \mathbb{R}^V$，数值稳定的Softmax为：

$$\text{softmax}(z)_i = \frac{\exp(z_i - \max_j z_j)}{\sum_k \exp(z_k - \max_j z_j)} \tag{134}$$

证明： 由于 $z_i - \max_j z_j$ or a ，因此不会发生上溢。ll i 分母至少为 1（当 $i = \arg\max_j z_j$ 时），从而避免下溢。
根据以下性质，该结果在数学上等价于标准 Softmax：

$$\frac{\exp(z_i - c)}{\sum_k \exp(z_k - c)} = \frac{\exp(z_i) \cdot e^{-c}}{\sum_k \exp(z_k) \cdot e^{-c}} = \frac{\exp(z_i)}{\sum_k \exp(z_k)} \quad \blacksquare \tag{135}$$

■

对于标准情况，这项稳定性保证是绝对的。减去最大值后，所有指数参数都位于 $[-\infty, 0]$，因此输出位于 $(0, 1]$。指数和至少为 1（来自最大元素），因此分母永远不会下溢。然而，对于非常负的 logits（低于 -88 （FP32 中）和 -9 （FP8 E4M3 中）），分子会下溢为零，使 Softmax 输出恰好为 0。从数学上说，这是正确的（该概率可以忽略不计），但如果下游代码对其取 log(0)，就可能导致问题。

S24.2 交叉熵数值稳定性。**交叉熵损失同时叠加了 Softmax 不稳定性和对数运算的不稳定性**。标准形式 $\mathcal{L} = -\log(\text{softmax}(z)_c)$ 需要先计算 Softmax（可能发生上溢），再对可能极小的概率取对数（可能下溢至 $-\infty$）。下面的稳定形式通过避免显式计算 softmax 概率，绕过了这两个问题。

命题 *27*（稳定交叉熵）数值稳定的交叉熵损失为：

$$\mathcal{L} = \log\left(\sum_j \exp(z_j - m)\right) + m - z_c \tag{136}$$

其中 $m = \max_j z_j$， c 为目标类别。
此公式：

1. 通过减去 m，避免 $\exp(z_j)$ 上溢
2. 通过在 FP32 中计算对数和指数，保留完整精度
3. 在数学上等价于标准形式

推导过程揭示了这种方法奏效的原因。标准交叉熵为 $-\log(\exp(z_c)/\sum_j \exp(z_j))$。展开商的对数： $-z_c + \log(\sum_j \exp(z_j))$。代入稳定的对数和指数： $-z_c + \log(\sum_j \exp(z_j - m)) + m = \text{LSE}(z) + m - z_c$。最终表达式仅涉及有界操作：非正数的指数函数值（上界为 1）、正数求和（结果始终为正）以及正数的对数（有定义）。为获得最高精度，我们完全使用 FP32 进行计算，即使激活值采用 FP8 或 BF16 表示也是如此。

S24.3 梯度数值稳定性。**梯度爆炸是训练发散最常见的原因**。与会立即产生 NaN 的前向传播不稳定性不同，梯度爆炸可能在模型发散前逐步累积多个训练步骤。交叉熵梯度的优势在于其天然有界——损失计算本身无需进行梯度裁剪。

命题 *28*（交叉熵梯度稳定性）梯度 $\nabla_z \mathcal{L} = \text{Softmax}(z) - e_c$ 始终有界：

$$\|\nabla_z \mathcal{L}\|_\infty \leq 1 \tag{137}$$

因为 Softmax 输出位于 $[0, 1]$，且我们从其中一个元素中最多减去 1。
这种梯度有界的性质，正是交叉熵成为分类任务首选损失函数的原因。 均方误差损失的梯度与预测误差成正比，且没有界限——如果模型对错误类别给出很高的置信度，梯度就可能任意大。交叉熵梯度在构造上有界：最坏情况下，$\text{Softmax}(z)_c = 0$（模型将正确类别的概率设为零），位置 c 处的梯度为 -1，其他所有位置的梯度之和为 $+1$。这种隐式梯度裁剪无需显式干预，便有助于提升训练稳定性。

S25. 屋顶线模型分析。屋顶线模型回答了优化中最重要的问题：我的代码受计算能力限制，还是受内存带宽限制? 内存受限代码可通过减少数据移动（融合、缓存）获益；计算受限代码则可通过更快的算术运算（更优算法、更低精度）获益。大多数大语言模型训练操作都受内存限制，这就是内核融合能带来如此显著加速的原因——我们并非以更快的速度完成更多计算，而是减少了内存流量。

S25.1 A100 屋顶线。**156 浮点运算次数/字节的“脊点”意味着，对于每从内存加载一个字节所执行的算术运算少于 156 次的操作，其瓶颈在于内存带宽，而非计算**。作为参考：一次向量加法每 4 个字节执行 1 次浮点运算（算术强度 $= 0.25$），这意味着 A100 执行向量加法时的吞吐量为 0.5 万亿次浮点运算，仅为 BF16 峰值吞吐量的 0.16%。大型矩阵乘法的算术强度可达 > 200，运行时能接近峰值吞吐量。内核融合的目标是将操作从内存受限区域推入计算受限区域。

屋顶线模型表明，在 A100 上，算术强度低于每字节 156 次浮点运算的操作受内存限制：定义 *35*（屋顶线模型）对于拥有 312 BF16 TFLOPs 和 2 TB/s 带宽的 A100：

$$\text{Performance} = \min(312 \text{ TFLOPs}, 2 \times \text{AI TFLOPs}) \tag{138}$$

脊点：算术强度$_{\text{ridge}} = 312/2 = 156$ 每字节浮点运算次数。
算术强度 < 156 的操作受内存限制；算术强度为 ≥ 156 的操作受计算限制。

表16。关键操作的算术强度

操作	浮点运算次数	字节数	AI
矩阵乘法 $[M,K]\times[K,N]$	$2MKN$	$4(MK+KN+MN)$	$\frac{MN}{2(M+N)}$
自注意力 $[N,d]$	$4N^2d$	$4(3Nd+N^2)$	$\frac{N}{d+1}$
RMSNorm $[B,N,d]$	$4BNd$	$8BNd$	0.5
交叉熵 $[B,N,V]$	$3BNV$	$8BNV$	0.375

交叉熵的算术强度为 = 0.375，严重受内存限制（需要算术强度 >156 才能达到计算受限）。
该表揭示了交叉熵优化为何如此重要。在算术强度 = 0.375 的情况下，交叉熵以 0.75 万亿次浮点运算运行——仅为峰值的 0.24%。如果能够以某种方式实现计算受限执行，该操作会慢 400 倍。我们的分块交叉熵不会改变算术强度（浮点运算次数相同，字节数相同），但通过避免完整 logits 张量的物化，减少了总 字节数。这不会改变我们实际执行的操作的屋顶线预测性能，但会彻底消除部分操作。

S25.2 内核融合优势。
命题 *29*（融合算　　术强度提升）将具有独立算术强度 <1 的 k 内存受限操作进行融合：

$$\text{AI}_{\text{fused}} = \frac{\sum_{i=1}^{k} \text{FLOPs}_i}{\text{Bytes}_{\text{input}} + \text{Bytes}_{\text{output}}} \tag{139}$$

通过消除中间内存访问，这可以将算术强度提高 $\approx k$ 倍。
Exa例 1（RMSNorm +残差融合）分离：算术强度 $\approx 0.5+0.25=0.75$（两者均为内存受限操　　d)
融合：算术强度 ≈ 1.0（单次内存往返）
加速比：$\approx 1.33\times$ 源于内存流量减少。
让我们深入了解融合为何能提升算术强度。单独执行 RMSNorm：读取输入 x（N 字节数），进行计算，然后写入输出 y（N 字节数）。单独执行残差加法：读取 y（N 字节数），读取残差 r（N 字节数），写入 $y+r$（N 字节数）。总计：5N 字节数。融合执行：读取 x（N 字节数），读取 r（N 字节数），计算 RMSNorm 与加法，然后写入结果（N 字节数）。总计：3N 字节数。浮点运算次数不变，内存访问减少 40%。对于内存受限操作，这直接转化为 40% 的加速比。这种模式可推广至更广泛的场景：任何一系列逐元素操作，只要后续需要写入内存，都可以融合为单次读取—计算—写入，从而消除中间结果物化。

S26. 扩展相关工作。**Chronicals 站在巨人的肩膀上**。FlashAttention 证明了内存高效注意力既能保证正确性，也能实现高速度。Liger-Kernel 展示了 Triton 可以媲美甚至超越手工优化的 CUDA。Unsloth 表明，LoRA 专用优化能够实现 2 倍加速比。我们的贡献在于将这些技术整合到统一框架中，并加入新颖的功能（LoRA+、结合 FlashAttention varlen 的序列打包、融合梯度裁剪），使其效果能够叠加。本节将对各项能力进行公平比较，同时承认每个框架都有不同的设计目标和权衡。

S26.1 训练框架比较。**没有任何单一框架适用于所有使用场景**。HuggingFace Transformers 更注重易用性和模型覆盖范围，而非原始性能。Unsloth 专门针对热门模型上的 LoRA 进行优化，为了实现最高速度而接受较低的模型覆盖范围。Liger-Kernel 是一个内核库，而非训练框架。Chronicals 则着眼于两者的交集：在热门模型上实现高性能，同时兼顾合理的覆盖范围和易用性。下表比较了具体优化的支持情况；在实际应用中，“最佳”选择取决于你的模型是否受支持，以及哪些优化对你的工作负载至关重要。

表17。训练框架比较

框架	Flash	融合内核	打包	LoRA+
HuggingFace	可选	No	No	No
Unsloth	Yes	部分	No	No
Liger Kernel	N/A	Yes	N/A	N/A
Chronicals	Yes	Yes	Yes	Yes

公平比较需要指出每个框架的优势所在。 HuggingFace 支持数百种模型架构；Chronicals 目前支持 8 种（Qwen、LLaMA、Mistral、Phi、Gemma、DeepSeek、Yi 和 InternLM 系列）。Unsloth 针对特定 GPU 架构包含自定义 CUDA 内核；Chronicals 使用 Triton 以实现可移植性，但会带来轻微的性能损失。Liger- Kernel 提供可与任何框架集成的可组合内核；Chronicals 则提供端到端的训练解决方案。用户应根据自身的具体需求进行评估：模型覆盖范围、性能目标以及基础设施约束。

S26.2 注意力实现。**高效注意力实现的发展表明，算法洞见能够实现仅凭硬件无法达到的效果**。 标准注意力的 $O(N^2)$ 内存占用使 32K+ 上下文长度变得不可行，直到 FlashAttention 证明通过逐块处理，可以在 $O(N)$ 内存中完成相同的计算。后续的每种实现都在保持或提升性能的同时，增加了新的能力（稀疏模式、FP8 支持、可变长度序列）。Chronicals 通过 PyTorch SDPA 使用 FlashAttention-2，以实现最大的兼容性；同时，FlashAttention-3 支持已列入计划，待 Triton 3.0 发布后实现。

表18。注意力实现对比

实现	内存	速度	特征
PyTorch SDPA	$O(N^2)$	1.0x	基础
xFormers	$O(N)$	2-3x	稀疏，分组查询注意力
FlashAttention-2	$O(N)$	3-4x	变长序列、因果
FlashAttention-3	$O(N)$	4-5x	FP8、Hopper

为什么使用 SDPA，而不是直接调用 FlashAttention? PyTorch 的缩放点积注意力（SDPA）提供了统一接口，可自动调度至最佳可用后端：安装了 FlashAttention-2 时使用它，否则回退至 xFormers，或使用高效的 cuDNN 注意力机制。借助这一抽象，Chronicals 能够在未安装 FlashAttention 的系统上运行（但速度较慢），并可自动受益于 SDPA 未来的改进，无需修改代码。与注意力计算本身相比，调度层的开销可以忽略不计（$< 1\mu s$）。

S27. 常见问题与解决方案。最令人沮丧的错误，是看起来一切正常，但结果却错误或不尽理想。本节记录了我们在开发和部署 Chronicals 期间遇到的问题，说明的不只是症状和解决方法，还包括其根本原因。了解问题发生的原因，有助于从业者诊断列表中未涵盖的新问题。

S27.1 显存不足（***OOM***）。**OOM 错误很少能直接指出实际原因**。PyTorch 会报告分配失败的位置，但问题通常出在更早的内存分配上：这些分配可能导致内存碎片化，或占用超出预期的内存。下表将各种症状与根本原因对应起来，但调试的第一步始终应是运行 torch.cuda.memory summary()，以了解完整的内存使用情况。请查看“已分配内存”和“已保留内存”之间的差距——差距较大表示存在内存碎片化。

表 19。按原因分类的 OOM 解决方案

原因	解决方案
大批量大小	减小批次，增加梯度累积
长序列	启用梯度检查点
大词汇表	使用 CCE（分块交叉熵（CCE））
优化器状态	使用 8 位 Adam
激活内存	启用 FlashAttention

我们遇到的最常见内存溢出原因是 logits 张量。词汇表为 128K、批次大小为 8、序列长度为 2048 的模型会产生一个 4.2 GB 的 logits 张量——通常是训练过程中占用最大的单项内存。症状：在损失计算期间而非前向传播期间发生内存溢出。解决方法：启用分块交叉熵，按 4K–8K 个词元对词汇表进行分块处理，从而无需完整物化 logits。这可将峰值内存从 4.2 GB 降至约 130 MB，且不会影响准确率。

S27.2 训练不稳定性。**训练不稳定性表现为三种模式：突发性（损失在一个步骤内尖峰至 NaN）、渐进式（损失在数百个步骤中缓慢增加）或停滞性（损失平台期，未呈下降趋势）**。 每种模式都表明不同的根本原因。突发性不稳定通常表示注意力或损失计算中出现数值溢出。渐进式不稳定表明学习率过高或梯度累积存在问题。停滞则表明学习率过低、参数被冻结或存在数据问题。

表 20。稳定性问题与修复方法

问题	Fix
损失爆炸	添加 Z 损失，降低学习率
梯度爆炸	启用梯度裁剪
注意力中的 NaN	检查是否存在零序列长度
损失未下降	验证梯度流（检查梯度范数 > 0）
收敛缓慢（LoRA）	使用 LoRA+，学习率比为=16

“注意力中的 NaN”这一错误值得特别说明，因为它十分隐蔽。 使用序列打包时，某些位置的注意力掩码可能全为零（填充位置）。对全部被掩码的值应用 Softmax 会产生 NaN，因为 Softmax($[-\infty, -\infty, \ldots]$)涉及 $0/0$。FlashAttention 通过其变长 API 能够正确处理这种情况，但标准 SDPA 无法做到。症状是：NaN 出现在第 0 层的注意力输出中，并传播到后续所有层。修复方法：确保针对打包序列正确计算 cu seqlens，并确保没有序列的长度为 0。

Z 损失是一种很有价值的稳定化技术，值得详细说明。 Z 损失会增加一项较小的惩罚，其大小与 $\log(\sum_j \exp(z_j))^2$，成正比，以此鼓励模型保持较小的 logits。这可以防止失控的动态过程：模型做出置信度很高的预测，产生较大的 logits，继而产生较大的梯度，使预测变得更加自信。我们建议将 Z-loss 系数设为 10^{-4}：既要足够小，不影响准确率；也要足够大，能够防止不稳定性。根据经验，使用 Z 损失训练的模型在最初的 1000步 中，损失尖峰减少了 10 倍。

S27.3 性能问题。**“训练速度慢”过于笼统，无法据此诊断问题——我们需要确定瓶颈究竟在计算、内存带宽还是 CPU 开销。** 性能分析层级如下：首先检查图形处理器利用率（nvidia-smi），然后检查内存带宽利用率（nsight-compute），最后检查内核级性能（torch.profiler）。图形处理器利用率低、内存带宽利用率高，说明存在内存受限内核（解决方法：融合）。图形处理器利用率低、内存带宽利用率也低，说明存在 CPU 开销（解决方法：使用 torch.compile，减少 Python 操作）。图形处理器利用率高但训练速度慢，说明存在计算受限瓶颈（解决方法：采用低精度、进行算法改进）。

表 21. 性能优化检查清单

问题	检查项
GPU 利用率低	启用 torch.compile
填充开销高	启用序列打包
注意力速度慢	确认已启用 FlashAttention
内存受限内核	使用融合 Liger 内核
慢速优化器	使用融合的 Triton AdamW

一个常见的性能问题：torch.compile 每一步都在重新编译。 症状：前 10 步速度很慢（正在编译），之后变快，到了第 11+ 步又变慢。原因：动态形状会触发重新编译。使用序列打包时，每个批次的打包长度可能不同，导致 torch.compile 将输入形状视为“新形状”并重新编译。解决方法：将所有打包批次填充到相同长度（数据集的最大打包长度），并使用 torch.compile(model, dynamic=False)。这样会牺牲变长打包带来的一些效率，但可以避免重新编译开销。

另一个不易察觉的问题：FlashAttention 会悄无声息地回退到标准注意力。 FlashAttention 有许多要求（头维度必须能被 8 整除、特定的数据类型、因果掩码格式）。不满足这些要求时，PyTorch SDPA 会在不发出警告的情况下悄然回退到速度较慢的实现。要确认 FlashAttention 是否已启用：使用 TORCH LOGS="+all" 运行，并搜索“sdpa”；或者使用 torch.profiler，查找名为“flash attn”的内核。如果发生回退，请调整模型配置（填充头维度、转换为 BF16）以满足 FlashAttention 的要求。

表 22. 按模型规模划分的推荐配置

模型批次		梯度检查点 精度		LoRA 秩	LR
0.5B	16	No	BF16	32	1×10^{-4}
1-3B	8	可选	BF16	32	5×10^{-5}
7B	4	Yes	BF16	64	2×10^{-5}
13B	2	Yes	BF16	64	1×10^{-5}
70B	1	Yes	FP8	128	5×10^{-6}

S28. 特定模型建议。

S29. 理论下界。

S29.1 注意力复杂度下界。

定理 *13*（注意力 IO 下界）任何计算精确注意力的算法都需要 :

$$\Omega\left(\frac{N^2 d^2}{M}\right) \text{ HBM accesses} \tag{140}$$

FlashAttention 达到了这一上界。

S29.2 装箱下界。
定理 *14*（BFD 最优性差距）最佳适应递减算法达到：

$$\mathrm{BFD}(I) \leq \frac{11}{9}\mathrm{OPT}(I) + \frac{6}{9} \tag{141}$$

这个下界是紧的：存在一些实例，其渐近比率可达到 $11/9$ 。

S30. 术语表。本术语表定义了本文中使用的关键技术术语和缩略语。

表 23. 技术术语和缩略语术语表

Term	定义
AI	算术强度（每字节浮点运算次数）
BFD	最佳适应递减装箱算法
BF16	脑浮点 16位格式
CCE	Cut Cross-Entropy
FP8	8位浮点格式
FSDP	全分片数据并行
GQA	分组查询注意力
HBM	高带宽内存
LoRA	低秩适配
MFU	模型浮点运算利用率
MHA	多头注意力
MQA	多查询注意力
OOM	内存溢出
RMSNorm	均方根归一化
RoPE	旋转位置嵌入
SRAM	静态随机存取存储器
SwiGLU	Swish门控线性单元