广义张量并行：以每秒 10+ 百亿亿次浮点运算的性能驱动万亿参数大语言模型训练

ShiqingFan[†]、Deepak Narayanan[†]、Jimmy Zhang、Jiangfei Duan、Prajwal Singhania、Haoran Zhang、Wanying Wang、Philip Monk、Xuwen Chen、Jared Casper、Jun Yang、Mohammad Shoeybi、Bryan Catanzaro 和 Vijay Anand Korthikanti[*, †]

英伟达

arXiv:2611.32057v1 [cs.DC] 24 Nov 2026

摘要。 使用高达万亿参数的最先进大语言模型（LLM）进行预训练，需要大量内存和计算资源。现有并行化方案往往要在每 GPU 内存与吞吐量之间取舍：数据并行会复制参数；张量并行和全分片数据并行（FSDP）则分别为每个微批次增加激活值和参数的通信开销；流水线并行在流水线填充和排空期间会使 GPU 闲置。我们展示，对于线性层，五种常用并行方式（数据并行、张量并行、序列并行、上下文并行和 FSDP）都是同一框架——广义张量并行（GTP）——的特例。在该框架中，每一层的参数和激活值都经过分区，并且任一方都可以在使用前通过全聚合重新物化。在 GTP 中，参数全收集是矩阵乘法（通用矩阵乘法，GEMM）一个输入上的可选预处理步骤；而 FSDP 实现通常将其作为整个 Transformer 块之前的步骤，这会增加内存占用并延迟通用矩阵乘法。GTP 视角使我们能够按通用矩阵乘法而非按块调度全收集：每次全收集都尽早发起以隐藏通信开销，同时又尽可能延后以限制内存占用。我们的实现已开源，地址为 https://github.com/NVIDIA/Megatron-LM；借助该实现，一个内部的 2T 参数混合专家（MoE）模型在 12,288 个 GB300 GPU 上达到**13.0 模型EFLOP/s**（每秒百亿亿次浮点运算），弱扩展效率为 98%，并可在 512 个 GPU 上运行 100 万词元的序列。在 256–1,024 个 GPU 上，我们的实现在三个混合专家模型上的吞吐量达到**1.05–1.55× Megatron-FSDP 的吞吐量**，在 DeepSeek-V4-Pro 代理模型上的吞吐量则达到 **1.27–1.31× 采用流水线并行的调优三维并行方案的吞吐量**。

1. 引言

现代大语言模型（LLM）的参数量已达数万亿 [3, 5–7, 30, 32]。这些模型的参数、优化器状态和激活值远超单个 GPU 的内存容量。[1] 要在数万亿词元上训练这些模型，就需要较高的吞吐量，以将训练时间控制在合理范围内 [6, 14, 30

]因此，要在数千个 GPU 上高效地进行分布式预训练，需满足两个关键要求。首先，每 GPU 内存中用于存储参数、优化器状态和激活值的占用应大致随 GPU 数量增加而按比例缩减，从而让更大的模型和更长的序列能够装入可用内存。其次，GPU 应保持忙碌：通信应与计算重叠，并且每个 GPU 上的矩阵乘法（通用矩阵乘法）应高效运行。

现有的并行化方案都至少会在以下某一项要求上有所取舍。数据并行（DP）会在每个 GPU 上复制参数，因此随着 GPU 数量增加，参数内存并不会减少；分布式优化器 [57] 可以对优化器状态进行分片。张量并行（TP） [61]会对参数进行分片，并且可以与序列并行（SP） [24] 结合，从而进一步在张量并行区域之间对激活值进行分片。然而，TP 会在每一层、每个微批次中传输激活值。这种通信很难被隐藏，并且只有在 NVLink

*负责此项工作。

†联系作者：Shiqing Fan <shiqingf@nvidia.com>、Deepak Narayanan <dnarayanan@nvidia.com> 和 Vijay Anand Korthikanti <vkorthikanti@nvidia.com>。[1]我们的论述同样适用于其他加速器，例如张量处理单元（TPU）：这类设备在设备组内通过高速互连通信，而组与组之间则通过较慢的网络连接。

© 2026年 英伟达。保留所有权利。

域内才足够快，这限制了 TP 的并行度；较大的 TP 并行度还会缩小通用矩阵乘法，降低算术强度和 GPU 利用率。流水线并行（PP） [15, 34, 35] 通信量较少，但如果采用朴素方式保留多个在途微批次的激活值，其激活值内存占用会接近未分布式计算的水平；此外，在流水线填充和排空期间，GPU 会处于空闲状态，并且要求各阶段的计算量相对均衡。全分片数据并行（FSDP）和 ZeRO-3 [57, 75] 会对参数、梯度和优化器状态进行分片，同时保持较大的通用矩阵乘法规模（前提是批次足够大），但会在前向传播和反向传播中为每个微批次对每个参数执行全聚合，从而增加通信开销。上下文并行（CP） [20, 27, 29, 68] 会将每个序列拆分到多个 GPU 上，以减少激活内存。专家并行（EP） [25, 71] 会将混合专家（MoE）模型中的专家分布到多个 GPU 上；CP 和 EP 都可以与上述方案组合使用。

在本文中，我们展示了，对于线性层，五种常用的并行方式（DP、TP、SP、CP 和 FSDP）都是一个统一框架的特例，我们将其称为广义张量并行（GTP；第 2 节）。在 GTP 中，参数矩阵沿隐藏维度进行分片，与 TP 中一样；输入激活值则沿词元维度进行分片。词元分区可以包含完整序列（DP），也可以包含每个序列的一部分（CP）。线性层独立处理这些分区；在 CP 中，注意力层和状态空间层还会在持有同一序列不同部分的 GPU 之间交换其状态（例如注意力层的键和值）。因此，每个线性层的 GEMM 都有两个分片输入，即参数和激活值，并且任一输入都可以在使用前通过全聚合重新物化：对于 SP，是激活值；对于 FSDP，则是参数。分片的并行度以及每个输入进行重新物化所跨越的 GPU 数量，会在通信与内存之间进行权衡，并且可以针对稠密层权重和专家权重分别选择。当两个输入都完全分片时，每 GPU 内存中的参数、优化器状态和激活值都会按 GPU 数量成比例缩减。

参数重新物化可以减少内存占用，但并非没有代价：每个微批次都会在前向传播和反向传播中对参数执行全聚合，并对其梯度执行归约散播（1.5× 在通信精度相同的情况下，这相当于数据并行每步一次的梯度同步通信量；第2.2节）。不同于张量并行的激活通信，参数全收集并不依赖前置计算，因此可以在使用该参数的层之前发起。何时发起每次全收集因此成为一个会带来不同后果的选择：如果重新物化的时机距离所需参数的计算过近，就会暴露通信时间 [67, 75]，而如果重新物化得过早，则会增加内存占用。现有的 FSDP 实现会以可配置的单元调度重新物化，通常以 Transformer 块为单位 [63] （第3.1 节和第7节）。当单元是一个完整的块时，该块的第一次通用矩阵乘法必须等待该块所有参数全收集完成，并且该块的所有参数会同时驻留在内存中。在 GTP 中，参数全收集则对应于序列并行的激活全聚集：它是通用矩阵乘法某个输入上的预处理步骤。相较于按块进行重新物化，按通用矩阵乘法调度参数重新物化，可以根据计算和内存需求，灵活调整每次全收集的发起时机、每个通用矩阵乘法需要等待的内容，以及哪些参数共享同一个集合通信。对于混合专家层，我们利用这一灵活性，在某个专家权重对应的通用矩阵乘法之前整整一个混合专家块发起其全收集（第3.3节）。

总结而言，我们作出以下贡献：

- **统一框架**。 我们将 DP、TP、SP、CP 和 FSDP 表述为参数和激活分片的特例，并可选地结合输入重计算（第2节）。
- **逐个 GEMM 进行重新物化调度**。我们通过独立的预取链（C1）、消费者专属等待（C2）以及合并的专家全收集（C3；第 3.3 节），按 GEMM 调度参数全收集；在使用 256 个 GPU 的两个基准模型上，我们的完整实现相较于分别移除 C1、C2 和 C3 的变体，速度分别提升了 4.6–13.9%、2.8–18.7% 和 3.5–9.0%（第 5.7 节）。

- **经过大规模评估的开放实现**。我们的 GTP 实现已在 Megatron-Core（MCore）中开源，并支持采用 AdamW 和 Muon 的稠密层与混合专家模型；借助该实现，一个内部 2T 参数的混合专家模型（Internal-2T）在 12,288 个 GB300 GPU 上以 8K 的序列长度达到 13.0 模型EFLOP/s，从 4,096 个 GPU 扩展至 12,288 个 GPU 时的弱扩展效率达到 98%（第 5.3 节）。GTP 还支持在 512 个 GPU 上运行完整的 Internal-2T 模型，序列长度最高可达 1M 词元（第 5.6 节）。在 256–1,024 个 GPU 上的三个混合专家模型中，我们的实现达到了 Megatron-FSDP（MFSDP；第 $55\times$ 的 1.05–$1.55\times$ 倍吞吐量（第5.4 节）。在 DeepSeek-V4-Pro（DSV4-Pro）代理模型上，其吞吐量达到采用流水线并行的调优三维并行方案（第5.5 节）的 1.27–$1.31\times$ 倍。代码可在https://github.com/NVIDIA/Megatron-LM获取。

组织结构。 第 2 节介绍 GTP 框架并分析其通信与内存。第 3 节说明我们如何针对每个 GEMM 调度参数全收集，以隐藏通信并限制内存占用。第 4 节介绍我们在 MCore 和 Transformer Engine 中的实现。第 5 节通过实证将 GTP 与适当的基线进行比较。第 6 节针对 GTP 的配置提供实践指导，包括何时将其与 PP 结合。第 7 节回顾相关工作。

2. 广义张量并行

大语言模型的大部分参数位于线性层中：注意力投影、多层感知机以及混合专家层中的专家。每个线性层都是两个输入之间的 GEMM：一个激活矩阵 X，每一行对应一个词元；以及一个参数矩阵 W。分布式处理这些层归结为决定如何拆分和存储 X 以及 W 并将其分布到各个 GPU 上。在本节中，我们将展示 DP、TP、SP、CP 和 FSDP 分别对 X 和 W作出了不同选择，并且这五种方式都是同一框架——广义张量并行（GTP；第 2.1 节）的特例。随后，我们将分析这些选择的通信（第 2.2 节）和内存（第 2.3 节）开销。PP [15, 34, 35]与此正交：它将模型的层划分为不同阶段，各阶段之间仅通过点对点方式交换激活值及其梯度。GTP 可与 PP 组合使用；第 6.3 节将讨论何时将二者结合。

其他层需要采用略有不同的处理方式。对于较小的权重张量，例如 LayerNorm [2] 或 RMSNorm [72] 权重，可以进行复制，以减少不必要的通信。注意力操作本身没有参数，而且无论是否使用 GTP，其在上下文并行下的通信方式都相同。

2.1. 一个线性层，两种输入

我们以一个多层感知机块为例：先执行一个 $h \to h'$ 通用矩阵乘法，再执行一个 $h' \to h$ 通用矩阵乘法，其中 h 是模型的隐藏维度，h' 是多层感知机的中间维度（例如，$h' = 4h$ 在许多稠密模型中）；其他线性层的行为也类似。

张量并行 [61] 沿隐藏维度将参数划分到 t 个 GPU 上：$h \to h'$ 通用矩阵乘法按输出列切分，$h' \to h$ 通用矩阵乘法按输入行切分（分别对应列并行层和行并行层）。随后，每个 GPU 计算每一层的一部分，并在每个张量并行区域结束时通过全归约合并各 GPU 的部分输出。全归约依赖前一个通用矩阵乘法的输出，因此若要让两者重叠，就必须将通用矩阵乘法拆分为更小的通用矩阵乘法 [65, 66], ，这会进一步降低算术强度。

数据并行和上下文并行 [20, 27, 29] 沿词元维度对激活值进行分区，并将其分布到 c GPU 上（图 1，顶部）。每个 GPU 使用自己的词元进行计算，并在每个优化器步骤中对参数梯度执行一次全归约；当梯度以桶为单位进行规约时，该操作可与反向传播重叠 [26]。包含完整序列的词元分区实现数据并行；拆分序列的分区实现上下文并行，

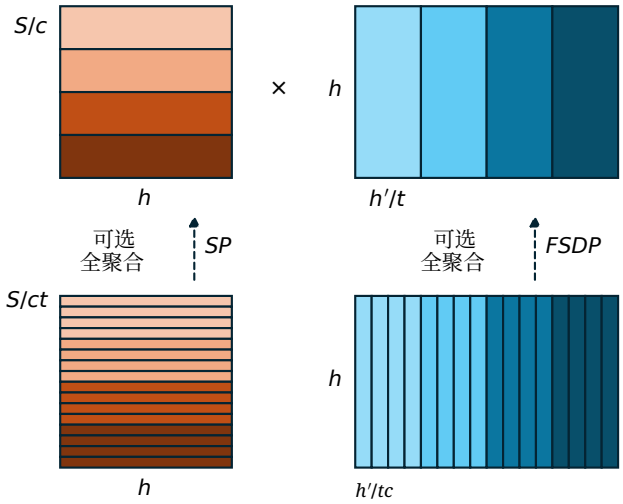


图 1 | 使用 $t = c = 4$ 对通用矩阵乘法进行分片和重新物化。上图：权重分布在 t 个 GPU 上，激活值的行（词元）分布在 c 个 GPU 上；每种阴影表示一个分片。下图：每个输入都可以选择以较小分片的形式存储，并在通用矩阵乘法前进行全聚合；激活值跨 t 组（SP）全聚合，权重跨 c 组（FSDP）全聚合。

这在线性层中与数据并行相同，但在注意力层和状态空间层中，共享同一序列的 GPU 之间需要进行通信。

在 GTP 中，这两者是两个独立的分片轴，因此一层会分布在 $n = t \cdot c$ 个 GPU 上；由于它们作用于不同的输入，因此可以自然地组合 [35]。仅进行分片仍会留下一些复制状态：每个张量并行分区都会在与其共享的 c 个 GPU 上复制，而 t 组中的每个 GPU 都持有相同的边界激活，即每个张量并行区域输入和输出处的激活值（例如，多层感知机块的输入 X）。可以通过进一步分片复制的张量，并在使用前通过全聚合重新物化来消除这两种复制（图1，下图）。跨 t 组重新物化边界激活即为 SP， [24], 它将每个区域的全归约替换为前置的全聚合和后置的归约散射。跨 c 组重新物化权重，即为 FSDP 和 ZeRO-3 [57, 75] 中的参数分片：在前向传播和反向传播的每个微批次中，权重都会在使用前从其分片中全聚合，并在使用后释放；梯度则会通过归约散射返回各分片。与激活全聚合不同，这些参数全聚合不依赖先前的计算，因此可以在使用它们的通用矩阵乘法执行前提前发起。

重新物化不必覆盖整个分片组。参数可以在一个权重重新物化组（简称权重组）内进行分片和重新物化，该组包含 r 个 c GPU：每个 GPU 持续存储每个权重的 $1/(t \cdot r)$，而权重组会在每次使用该权重的通用矩阵乘法之前，全收集其完整的张量并行分区（$1/t$ 权重的）。权重组 c/r 持有相同的副本，并像数据并行一样，每个优化器步骤同步一次梯度。当 $t = 1$时，设置$r = c$ 可得到全分片数据并行， $r = 1$ 可得到数据并行，中间值则对应混合分片数据并行（HSDP） [75]；更大的t 则将这些选择与张量并行结合。每个微批次的全收集因此都限制在权重组内，该组可部署在 NVLink 等高速互连上 [46], ，而每步一次的副本同步则通过较慢的链路进行（第4.1节）。

因此，每个权重都会在 p 个 GPU 之间进行分片，我们将其称为 *GTP 并行度*：

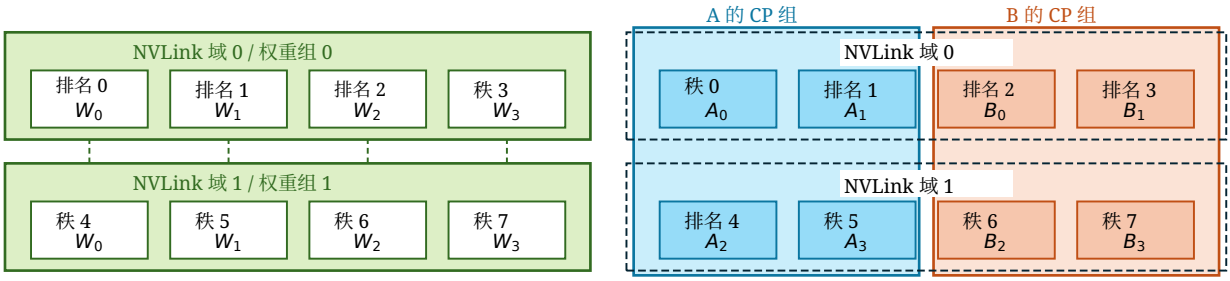
$$p = \underbrace{t}_{\text{TP degree}} \times \underbrace{r}_{\text{re-materialization degree}} . \tag{1}$$

例如，采用 TP2 的 GTP4（$t = 2,\ r = 2$）会将每个权重拆分为两个张量并行分区，并将每个分区存储为两个分片，这些分片会在使用前于权重组内进行全聚合。对于 $r = 1$，GTP 会退化为结合数据并行的数据并行张量并行；对于 $t = 1$，GTP 会退化为全分片数据并行或混合分片数据并行。表 1 将常见方案置于这一空间中。

方案	t	c	重新物化激活值	重新物化参数
数据并行 / 上下文并行	1	n	–	–
TP	n	1	–	–
张量并行+序列并行	n	1	✓	–
全分片数据并行 / ZeRO-3	1	n	–	✓ $(r = c)$
HSDP	1	n	–	✓ $(1 < r < c)$
GTP	t	c	✓	✓ $(r \leq c)$

表 1 | ：作为分片与重新物化选项的并行方案。

混合专家层。 混合专家模型 [6, 9, 32, 60] 无需对每个词元激活所有参数，即可提升模型容量：混合专家块将多层感知机替换为 E 个专家多层感知机和一个路由器，由路由器将每个词元发送给其中的 k 个。专家并行（EP） [25, 71] 将完整专家分片到不同 GPU 上，并通过一对全对全集合通信，将每个词元路由到存有其对应专家的 GPU，再将结果传回。在混合专家块中，EP 的分发与合并全对全通信取代了 TP+SP 的激活全聚集与归约散射，但只交换路由掩码选中的词元。专家 GTP（EGTP）将 GTP 的参数分片与重新物化应用于路由专家权重。EGTP 并行度为 r_e时，权重组中的每个 GPU 存储这些权重的 $1/r_e$，并在每次使用这些权重的通用矩阵乘法之前全聚合完整权重。稠密 GTP 与 EGTP 的并行度分别独立配置。稠密线性层（包括注意力投影）采用上述 TP 、 SP 与 GTP 的组合，而路由器权重等较小参数则保持副本。



(a) 权重视图　　（b）数据视图

图 2 | 八个 GPU 上独立的权重和数据视图（TP=PP=1，微批次大小为 1）。（a）每个 NVLink 域存储 W 的 4 路分片副本；权重全聚合（AG）和梯度归约散播（RS）均在域内进行。虚线连接用于同步同一分片的副本。（b）样本 A 和 B 分别跨越 CP 组 $\{0,1,4,5\}$ 和 $\{2,3,6,7\}$中的四个序列分区；注意力通信跨越不同域。此小型布局仅用于说明：在实际应用中，如有可能，最好将 CP 组保持在同一 NVLink 域内。

长序列。GTP 无需使用 TP，即可将一个长序列拆分到多个 GPU 上。线性层会独立处理每个词元，因此权重组中的 GPU 可以分别处理同一序列的不同部分。一个序列也可以跨越多个权重组（图2）。拆分一个

序列并分布到多个 GPU 上，不会改变存储各权重分片的 GPU（第4.1.2节）。权重全聚合仍局限于各权重组内，而注意力层和状态空间层则通过序列的 CP 组进行通信。

2.2. 通信

t 与 c 的选择需要在参数通信和激活通信之间进行权衡：参数通信会随着 t 的增大而减少，而激活通信则会随着 t的增大而增加。

为量化这种权衡，我们对 MLP 块的通信量进行建模，单位为每个 GPU 上的张量元素数；条件为每个优化器步骤包含一个具有 S 个词元的微批次，并且在参数重新物化时采用完全参数分片（$r = c$）。系数 γ_W 和 γ_X 分别用于缩放参数量和激活量，相对于不进行重新物化的 TP+CP 情况进行计算，其中 $\gamma_W = \gamma_X = 1$。在不进行重新物化时，MLP 块会在 c组上执行一次梯度全归约，并在 t组上执行两次激活全归约，分别用于前向传播和反向传播。参数重新物化会将梯度全归约替换为两次参数全收集（每个传播阶段各一次）和一次梯度归约散播，因此 $\gamma_W = 1.5$。激活重新物化会在每个传播阶段使用 SP 的全收集和归约散播，并额外对权重梯度通用矩阵乘法的输入执行一次全收集，因此 $\gamma_X = 1.25$：

$$\mathrm{comm}(t, c) = \gamma_W \frac{4hh'}{t}\left(1 - \frac{1}{c}\right) + \gamma_X \frac{4Sh}{c}\left(1 - \frac{1}{t}\right). \tag{2}$$

采用 $r < c$ 时，每个微批次的参数全收集和梯度归约散播都在由 r 个 GPU 组成的权重组内进行，而 c/r 副本还会在每个优化器步骤中执行一次梯度全归约。如果集合通信采用不同精度，则必须根据每个元素的字节数进行加权。在梯度累积期间，除非以增加内存开销为代价，在各微批次间保留已物化的参数 [53]，否则每个微批次都会重复进行参数全收集。

对于公式 2中的单微批次模型，设 $s = S/n$ 为每个 GPU 分配到的微批次词元数。对于较大的 t-组和 c-组，参数项近似为 $4\gamma_W hh'/t$，激活项近似为 $4\gamma_X sht$。将这些项分别除以对应的组带宽 B_c 和 B_t，即可在所有集合通信采用相同精度时，得到仅考虑带宽的通信时间模型。对于$\gamma_W = 1.5$、$\gamma_X = 1.25$以及固定的 s，模型所预测的通信时间在以下条件下达到最小值

$$t^* = \sqrt{1.2 \cdot (h'/s) \cdot (B_t/B_c)}. \tag{3}$$

每个 GPU 上的词元数较多时，会将 t^* 推向 1（全分片数据并行）；每个 GPU 上的词元数较少时，则倾向于在 NVLink 域内使用更大的 t 。

在 s固定时，TP 每个 GPU 的激活值量与 $t - 1$ 成正比（公式 2），因为每个 GPU 上的通用矩阵乘法都会消耗其 $s \cdot t$ -组中所有 t 个词元的激活值。相比之下，对于混合专家模型，每个词元最多会被发送到 k 个 GPU（即承载该词元所路由至的专家的 GPU），因此无论 EP 组大小如何，一个 GPU 的 EP 数据量最多是其词元数的 k 倍。因此，EP 组可以远大于 TP 组。

上述通信时间模型忽略了两种影响：通信与计算之间的任何重叠，以及每次集合通信的延迟项（对于训练中典型的消息大小而言，该项很小）。参数全收集最能体现重叠的重要性，因为它们可以在使用这些参数的通用矩阵乘法之前发起（第 2.1节）。接下来，我们将估算可用于隐藏这些通信的计算量。

将全聚合隐藏在计算之后。 每个专家处理更多词元，可增加计算量，而不增加权重全聚合量。考虑 TP1 下的一个专家通用矩阵乘法。均衡路由

到由 EP GPU 组成的专家并行组后，每个专家会获得 $s_e = \text{EP} \cdot s \cdot k/E$ 个词元。其权重包含 hh' 个元素，每个元素占 b 字节，分片到 $r_e > 1$ 个 GPU 上，其中 h' 是该专家的中间维度。在实际达到的 GEMM 吞吐量 F （ FLOP/s ）和全聚合带宽 B_r^e （字节 /s ）下，一次通用矩阵乘法的时间与一次权重全聚合的时间之比为

$$\frac{t_{\text{GEMM}}}{t_{\text{AG}}} = \frac{2s_e hh'/F}{b(1-1/r_e)hh'/B_r^e} = \frac{2s_e B_r^e}{b(1-1/r_e)F}. \tag{4}$$

对于固定的 F 和 B_r^e，权重大小 hh' 会相互抵消，因此比率取决于每个专家的词元数，而非权重大小；在实践中， F 还取决于通用矩阵乘法的形状。对于 TP1 下的稠密层，每个 GPU 处理 s 个词元，并在由 r个 GPU 组成的权重组中全聚合权重；同样的分析也适用。比率越高，隐藏通信的机会就越大；实际重叠情况还取决于集合通信启动开销、网络竞争和调度（第3.3节）。

2.3. 内存

对参数进行分片（TP）可减少每个 GPU 上的参数内存占用，对词元进行分片（DP 或 CP）可减少每个 GPU 上的激活内存占用；重新物化可将这两种缩减扩展至全部n 个 GPU。对于不进行重新物化的 MLP 块，每个 GPU 上的参数存储量为 $2hh'/t$，边界激活值存储量为 Sh/c，以上均按张量元素计数，且不包括中间激活。将参数在 r 个 GPU 上重新物化，可将前者减少至 $2hh'/(tr)$，即 $2hh'/n$，其中 $r = c$；重新物化激活值可将后者减少至 Sh/n。梯度分片遵循参数分片，而分布式优化器则将优化器状态分片至全部 n 个 GPU。峰值内存还包括重新物化的参数和通信缓冲区，其大小取决于全聚合操作提前发起的程度（第 3节）。

3. 按 GEMM 调度参数重新物化

第 2 节将参数重新物化视为 GEMM 某一输入的预处理步骤。基于这一视角，本节介绍如何按 GEMM 调度重新物化：通过消费者接口和策略，使每次全聚合都与使用该数据的 GEMM 协调执行。权重的 消费者 是读取该权重的 GEMM，而其视图 则是这些 GEMM 读取的重新物化副本。每个专家都是一个包含两个线性层 FC1 和 FC2 的多层感知机；采用分组 GEMM 时，每个 GPU 会在一次 GEMM（GFC1）中计算其所有本地专家的 FC1，并在另一次 GEMM（GFC2）中计算 FC2。

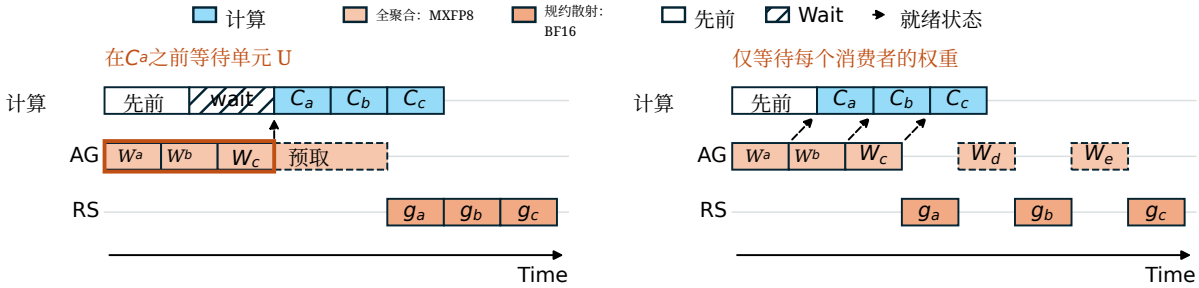
3.1. 调度问题

如第 2.1 节所述，与激活全聚合不同，参数全收集可以在使用它们的 GEMM 执行之前发起。任何此类调度都必须权衡三个相互关联的因素，在混合专家模型中尤为如此。

P1：计算通信重叠。每次全聚合都必须足够早地发起，确保在消费者需要之前完成；消费者也不应等待无关的通信。同一模块中的消费者可能在不同时间需要各自的参数：在融合式 MoE 层中，GFC1 和 GFC2 是不同的通用矩阵乘法，因此如果两者都等待同一组全聚合，GFC1 通用矩阵乘法就会被 GFC2 权重的全聚合延迟。跨 NVLink 域分片的专家参数需要更长的预取窗口，因为它们的全聚合使用速度较慢的域间 InfiniBand [18] 或 RoCE [19] 链路。

P2：内存占用。重新物化会将每个 GPU 的持久化权重内存缩减为一个存储分片（每个权重的 $1/(t \cdot r)$）。不过，无论 r如何，每个视图都包含完整的张量并行分区（权重的 $1/t$）。每个完整梯度也会一直保留在缓冲区中，直到归约散射读取完毕。因此，峰值内存取决于调度中同时存活的视图和梯度缓冲区数量：提前进行更多预取会增加视图，而重叠梯度归约则会让缓冲区保留更长时间。调度应尽可能晚地物化每个参数，并在其最后一个读取者完成后立即释放。

P3：集合通信效率。分别对参数张量执行全聚合，可独立调度每次全聚合，但每次集合通信都会产生启动开销和通信启动延迟。对于较小的张量，这些固定开销可能超过数据传输时间，使全收集受延迟限制。合并会将多个张量的全收集合并为一次集合通信，只需支付一次启动开销，并将启动成本分摊到更大的数据传输中。然而，合并不同消费者所需的参数可能会延迟较早的消费者，并使更多视图保持存活。分组通用矩阵乘法体现了这种权衡：一次分组通用矩阵乘法中的本地专家权重会被一同使用，因此可以共用一次集合通信；而 GFC1 和 GFC2 在不同时间需要各自的权重。



（a）MFSDP：单元就绪　　（b）GTP：每个消费者的就绪状态

图 3 | 反向传播调度（示意图）：(a) MFSDP 按单元等待；(b) GTP 按消费者等待。 C_i、 W_i 和 g_i 分别表示消费者、其权重和其梯度。在两个面板中，AG 和 RS 在相同的 GPU 上共享一个通信器，并按顺序执行。

现有 FSDP 实现中的调度。现有 FSDP 实现通过模块钩子调度重新物化，通常作用于 Transformer 块（第 7 节）。模块运行前，钩子会等待其模块的所有参数全收集完成。采用块级钩子时，一个块中的第一个 GEMM 因此会等待该块中的每个参数全收集（P1；图 3），而临时内存和集合通信分组也会跟随该块（P2–P3）。虽然也可以在更小的模块上设置钩子，但模块钩子无法深入融合算子内部：当专家权重采用分片时，融合专家层的 GFC1 GEMM 也会等待 GFC2 的权重。相反，我们针对每个消费者分别调度重新物化。表 2比较了 PyTorch FSDP2、MFSDP 和 GTP 对参数全收集的调度方式。

3.2. 消费者接口

每个读取分片权重的通用矩阵乘法都通过通用消费者接口获取其视图，而不是通过模块钩子获取（图 4）。前向传播、激活重计算和输入梯度（数据梯度）通用矩阵乘法读取 W。权重梯度通用矩阵乘法只读取该层的输入激活值和输出梯度，因此可以在权重梯度计算之前释放视图。随后，消费者将权重梯度交回，以便将其归约散射到存储分片中。

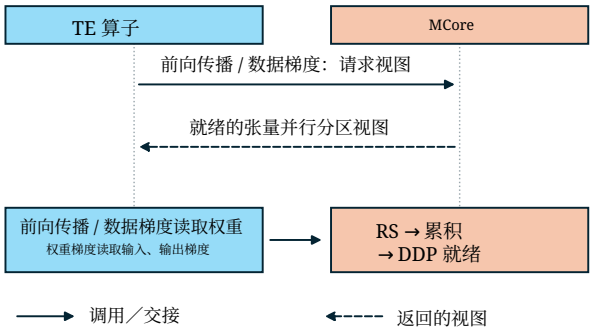
	PyTorch FSDP2 [51, 54]	**MFSDP** [43]	**GTP**
调度粒度	被封装的模块 `fully_shard`（通常为一个 Transformer 块）。	Transformer 块或更细粒度的子模块（可配置）。	单个通用矩阵乘法。
何时问题（C1）	下一个模块，按执行顺序或显式目标。	下一个参数组，按配置的顺序，在预取预算内。	每条链：下一个稠密层通用矩阵乘法；下一个混合专家块的专家。
等待什么 GEMM 等待 (C2)	其模块的所有全收集操作。	已挂钩模块的所有全收集（一个 Transformer 块或更小的子模块）。	仅其自身的权重。
融合式 MoE 层	分片时，GFC1 还会等待 GFC2 的权重。	分片时，GFC1 还会等待 GFC2 的权重。	GFC1 和 GFC2 各自等待自己的权重。

表 2 | 各实现如何调度参数全收集。第3.3 节定义了 C1、C2 和 GTP 的预取链。在两种全分片数据并行实现中，均可配置钩子粒度；第5.4.1节介绍了经过评估的 MFSDP 配置。

```python
# MCore: implements TE's DistributedWeight.
class GTPShardedParam(torch.nn.Parameter):
  def materialize_group_for_forward(self):
    w = self.wait_gather()  # Wait for W
    self.next_w.all_gather_async()  # Prefetch
    return w

# TE: fused grouped expert MLP.
def forward(x, gfc1, gfc2):
  w1 = gfc1.materialize_group_for_forward()
  h = activation(fwd_grouped_gemm(x, w1))
  w2 = gfc2.materialize_group_for_forward()
  return fwd_grouped_gemm(h, w2)
```

（a）消费者 API



（b）TE–MCore 交互

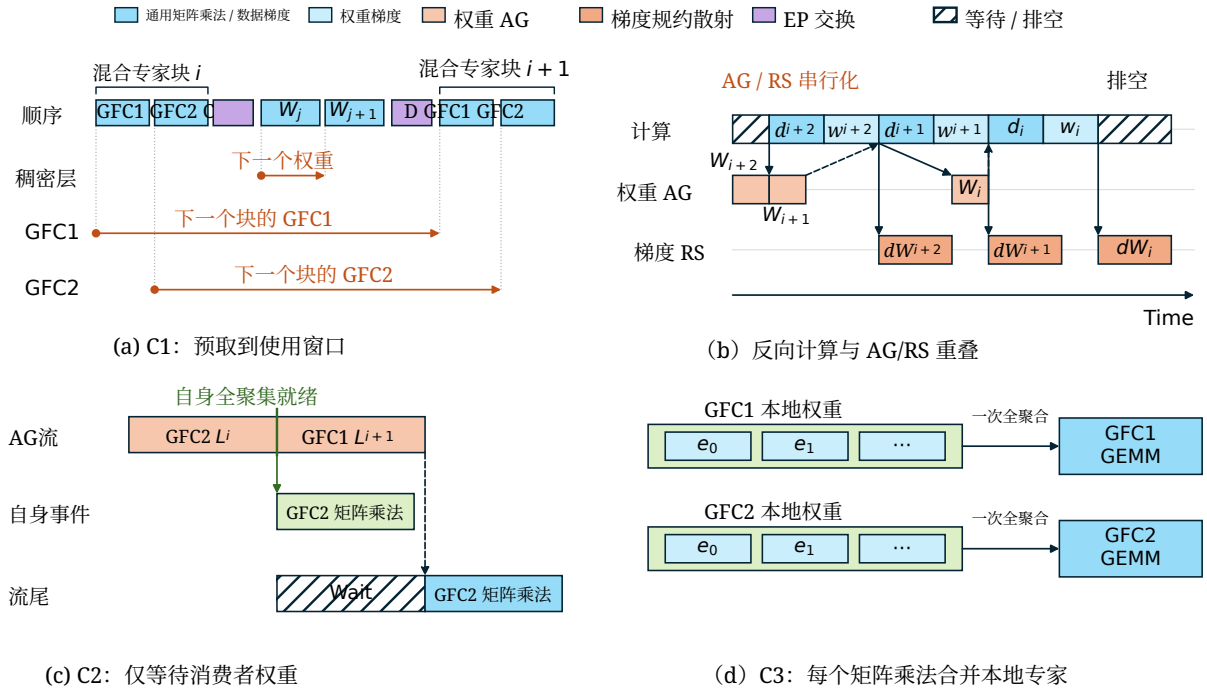
图 4 | Megatron-Core（MCore）与 Transformer Engine（TE）之间的消费者接口。（a）TE 请求 GFC1 和 GFC2 的权重视图；MCore 等待每次全聚合，并预取后续消费者的权重。（b）前向传播和数据梯度 GEMM 使用权重视图；权重梯度使用X 和 dY，并将梯度交给 MCore 进行归约散射和累加。启动与缓冲区管理未在图中展示。

通用矩阵乘法等待其权重的全聚合完成，然后发起后续通用矩阵乘法权重的全聚合，接着开始计算。第 3.3节中的调度策略决定预取目标和集合通信分组。第 4 节介绍 Transformer Engine 和 MCore 如何实现该接口。

3.3. 调度策略

按通用矩阵乘法调度全聚合涉及三个决策：何时发起每次全聚合（C1）、每个通用矩阵乘法需要等待什么（C2），以及哪些参数共享一次集合通信（C3）。这些决策在混合专家层中尤为重要，因为 GFC1 和 GFC2 在一个融合模块内运行，且专家全聚合可能跨越 NVLink 域。

何时发起集合通信（C1）。我们选择前瞻距离，以平衡通信重叠与预取权重占用的内存。预取链指定在通用矩阵乘法使用当前权重时，接下来要



(a) C1：预取到使用窗口

（b）反向计算与 AG/RS 重叠

(c) C2：仅等待消费者权重

（d）C3：每个矩阵乘法合并本地专家

图 5 | GTP 的三种参数调度策略。(a) C1：预取下一个权重稠密层和下一个块专家。(b) 通过串行化全聚合和规约散射实现反向传播重叠。(c) C2：逐消费者等待。(d) C3：逐通用矩阵乘法进行专家合并。D 和 C 分别表示 EP 分发与合并； d 和 w分别表示数据梯度和权重梯度； L_i 和 e_j 分别表示一个层和一个专家权重。

进行全聚合的权重。链成员关系决定前瞻距离；第4.3 节介绍注册和预热。

- **稠密权重：用较短的前瞻隐藏快速的全聚合。** 当稠密权重的权重组位于同一 NVLink 域内时，其全聚合速度较快，因此较短的前瞻即可满足需求，并能限制预取视图所占用的内存。默认情况下，所有稠密权重共享一条链：消耗 W_i 会在前向传播中预取 W_{i+1}，并在反向传播中预取 W_{i-1}（图 5（a、b））。全聚合 W_{i+1} 是否能够被完全隐藏，取决于其持续时间相对于 W_i 上 GEMM 的计算时间。
- **专家权重：用较长的前瞻隐藏较慢的全聚合。** 跨域 EGTP 全聚合需要更长的重叠窗口（第 4.1.3 节），因此 GFC1 和 GFC2 使用独立的默认链。GFC1 和 GFC2 分别预取下一个混合专家块中的对应权重，使其全聚合能够与当前块的其余部分、中间的稠密层以及 EP 令牌交换重叠（图 5（a））。这一更长的窗口提供了更多计算来隐藏较慢的全聚合（P1），但也使预取视图保持活动状态的时间更长（P2）。

在反向传播中，每层先运行数据梯度计算，再运行权重梯度计算，而权重梯度计算不会读取权重。因此，在数据梯度计算之前发起下一次全聚合，就能使其与两次通用矩阵乘法重叠（图 5(b)）。权重梯度计算完成后，其梯度会进行规约散射，这可以与反向传播接下来处理的各层中的通用矩阵乘法重叠。每个 GPU 都会保留规约后的梯度分片，直到执行优化器步骤。

每个通用矩阵乘法等待什么（C2）。我们的实现会在每次全聚合后、排队任何后续预取之前记录一个完成事件，并且每个通用矩阵乘法只等待其自身事件（图5（c））。全分片数据并行也会为每次全聚合记录一个事件 [51]，但其模块钩子会等待配置单元中的每个

全聚合。当 GFC1 和 GFC2 共享一个单元时，GFC1 通用矩阵乘法也会等待 GFC2 的权重。

哪些参数共享一次集合通信（C3）。为一个分组通用矩阵乘法提供输入的本地专家权重会被一起消费，因此会在一次合并集合通信中进行全聚合，并为 GFC1 和 GFC2 分别设置组（图5（d））。这种分组方式提高了集合通信效率（P3）。逐专家全聚合要求为每个本地专家分别执行一次集合通信，而将 GFC1 和 GFC2 组合在一起会耦合那些在不同时间需要其权重的消费者。第 5.7 节评估了这种合并策略带来的吞吐量收益。

4. 实现

本节介绍我们如何在 Megatron- Core（MCore）中实现 GTP，以及第 3节中调度策略的实现方式[42]和 Transformer Engine（TE） [47]。图 6 总结了具体实现：TE 算子消费权重视图，而 MCore 管理分片存储、通信、预取调度和 CUDA 图重放。TE 将第 3.2 节的消费者接口定义为 `DistributedWeight` 协议，MCore 的 `GTPShardedParam` 实现了该协议。TE 算子通过同时接受分布式权重和普通张量的函数来获取权重并返回梯度，因此无论是否使用 GTP，都可以运行相同的算子代码。当 MCore 构造 TE 线性层或分组线性算子时，会传入存储分片的填充后的行数，因此该算子绝不会分配完整权重。

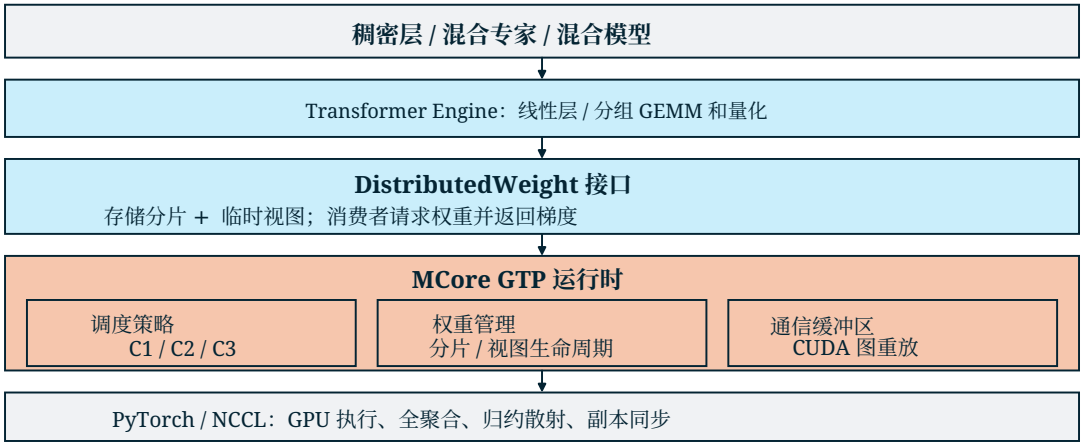


图 6 | GTP 架构。TE 通过 `DistributedWeight`；MCore 管理调度（§3.3）、权重（§4.2）、缓冲区（§4.2.3）和图重放（§4.4）。运行时构建于 PyTorch 之上，并使用 NCCL 进行集合通信。颜色用于区分模型与基础设施（灰色）、计算与接口（蓝色）以及 GTP 运行时（橙色）。

4.1. GPU 组与放置

MCore 将 GPU 组织成多个组，每个组都是一组共同执行集合通信的 GPU。每个 GPU 都属于一个张量并行组（持有每个权重的不同张量并行分区的 GPU）、一个权重组（每个 GPU 分别存储同一张量并行分区的不同分片）、一个副本组（持有相同分片的 GPU）以及一个上下文并行组（共享同一序列的 GPU）。

4.1.1. 权重组

GTP 将每个张量并行分区（$1/t$ 权重的）拆分为 r 个分片（$1/(t\cdot r)$ 的权重），权重组中的每个 GPU 各存储一个分片。每个 GPU 仅存储自己的分片，并在每次使用该分区的通用矩阵乘法之前，通过全收集获取完整的张量并行分区。GTP 沿行（输出维度）拆分每个张量并行分区，即采用 PyTorch 和 Transformer Engine 用于权重的 `[out, in]` 布局。在列并行层中，张量并行和 GTP 因此沿同一维度进行拆分；在行并行层中，张量并行拆分输入维度，而 GTP 拆分输出维度。权重梯度在权重组内进行规约散射。随后，MCore 的 `DistributedDataParallel` 包装器（DDP）会同步每个分片副本之间的梯度，而 MCore 的分布式优化器则在这些副本之间分区存储优化器状态。

并非所有权重都适合分片。较小的权重分片后节省的内存有限，但每次使用它进行通用矩阵乘法前仍需执行一次全收集。此外，它还可能需要大量填充：DSV4-Pro 的 KV 投影只有 512 行，因此在 64 个 GPU 上对其进行分片时，填充量会使其大小变为原来的四倍（第 8 [58]4.2.2 节 和 5.10节）。GTP 可以改为让这类权重保持副本：这样，每个 GPU 都会存储完整的张量并行分区，而且该权重无需全收集或填充。例如，我们在 DSV4-Pro 上的运行会让 KV 投影保持副本。

4.1.2. 在权重组内及权重组之间拆分序列

一个较长的序列可能需要拆分到多于一个权重组所包含的 GPU 数量上。GTP 不会扩大权重组，因为这会使权重全聚集跨越 NVLink 域；而是将序列拆分到多个权重组中。在 $q \times m$ 放置方式下，每个序列会拆分到每个q 权重组中的 m 个 GPU 上，实际 CP 度数为 qm。我们将 m 称为外层上下文并行度。在大小为 r的每个权重组内，剩余因子 r/q 会沿批次维度拆分相互独立的序列。图 2(b) 展示了 2×2 这种情况：每个序列跨越两个权重组中的各两个 GPU。

例如，表6 中的 1M 词元配置使用 $r = q = 64$ 和 $m = 2$。每个序列跨越两个 64-GPU 权重组，每个 NVLink 域各一个，从而实现带 GTP64 的 CP128，并且权重组内不进行批次拆分（$r/q = 1$）。权重全聚合使用 NVLink，而 CP通信跨越不同域。每个 GPU 存储每个分片稠密权重的 $1/64$ ，是采用单个 128-GPU 权重组时所存储 $1/128$ 的两倍。

4.1.3. 跨*NVLink*域的专家权重分片

我们将每个 EP组置于一个 NVLink 域内（一个域的 72 个 GPU 中的 64 个 [46]），因此 EP 的分发与合并通信始终在 NVLink 上进行。在没有 EGTP 的情况下，每个 EP组中处于相同位置的 GPU 都持有相同的专家权重。EGTP 将这些权重分片到不同域中的 r_e 个此类 GPU 上，使每个 GPU 的专家权重存储量大约减少 r_e倍。随后，专家全聚合会跨越速度较慢的域间网络。独立的 GFC1 和 GFC2 链使每次全聚合都能获得一个完整的混合专家块来完成（第3.3节）。

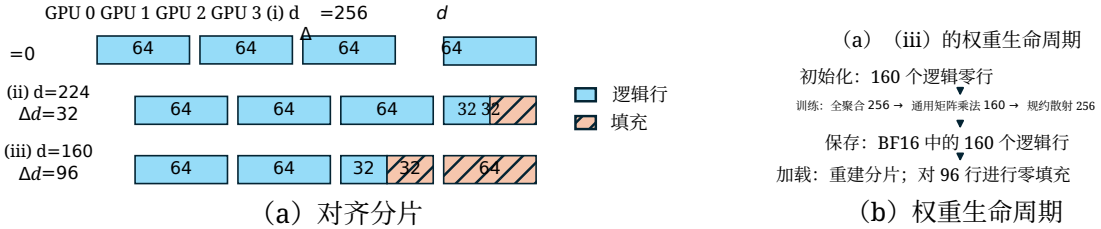
4.2. 权重存储与通信缓冲区

每个 GPU 都会持久存储自己的权重分片，并需要临时缓冲区来存放全聚合的张量并行分区以及归约散播的梯度。这些缓冲区的格式和大小决定了内存使用量和集合通信开销。

4.2.1. 低精度权重

GTP 可使用 TE 针对这些格式提供的训练方案，将权重分片存储为 MXFP8 [58] 或 NVFP4 [37], 。优化器每步仅对更新后的 FP32 主分片量化一次。随后，全收集直接传输量化分片及其缩放因子，因此每个微批次都无需重新量化权重。每个存储分片包含两种量化布局：用于前向传播 GEMM 的按行布局，以及用于 dgrad GEMM 的按列布局。每次全收集仅传输其 GEMM 所读取的布局。因此，低精度存储既能减少持久化权重内存，也能减少全聚合通信流量。但它不会降低梯度通信的精度（BF16 或 FP32，单独配置），也不会降低主权重或优化器状态的精度（默认均为 FP32）。

4.2.2. 对齐填充



（a）对齐分片　　（b）权重生命周期

图 7 | （a）四个存储分片的 MXFP8 对齐（每个分片 64 行）。情况（i–iii）展示了逐渐增加的填充；在（iii）中，GPU 3 完全由填充组成。（b）对应（a）（iii）的权重生命周期，不将填充计入通用矩阵乘法和检查点。

每个 GPU 为每个权重存储其张量并行分区的一个分片。对于一个张量并行分区（$1/t$ 权重的）包含 d 行，并在一个由 r 个 GPU 组成的权重组中进行划分时，每个存储分片（$1/(t\cdot r)$ 权重的）名义上包含 d/r 行。MXFP8 和 NVFP4 要求每个存储分片的行数为对齐 a的倍数；对于 MXFP8，该对齐为 32，对于 NVFP4 为 16，（对于 BF16 为 $a = 1$）。每个 GPU 仅量化其自身的存储分片（第4.2.1节），因此量化块不能跨越两个存储分片。这些对齐方式使分片边界保持在量化块边界上，从而允许 GTP 直接全聚合量化后的权重及其缩放因子，而无需对其进行反量化和重新量化。当 d/r 不是该对齐值的倍数时，GTP 会向张量并行分区追加零行，使每个存储分片都具有~相同的对齐后行数。填充后的张量并行分区有 d 行，其中 Δd 行为填充，每个存储分片有 d_{local} 行：

$$\tilde{d} = ar\left\lceil \frac{d}{ar} \right\rceil, \qquad \Delta d = \tilde{d} - d, \qquad d_{\text{local}} = \frac{\tilde{d}}{r}. \tag{5}$$

~因此，全聚合会重建一个包含 d 行的张量并行分区：先是 d 逻辑行，随后是 Δd 填充行（图7（a））。

通用矩阵乘法仅使用逻辑行。在前向传播、数据梯度或重计算通用矩阵乘法之前，GTP 会从全收集的张量并行分区中裁掉填充部分，并在不进行反量化的情况下调整其缩放因子。权重梯度通用矩阵乘法仅计算逻辑行的梯度，GTP 将这些梯度放入一个~零填充缓冲区，其中包含 d 行，然后将其归约散射回存储分片。检查点也只存储逻辑行（图7（b））。加载检查点时，会根据目标分片配置重新计算填充部分（第4.6节）。因此，填充会增加存储和通信开销，但不会增加计算量。第5.10节量化了这种开销在各基准模型上的情况。

当权重无法被均匀整除时，FSDP2 也会填充其分片 [52], ，但填充量不会达到低精度格式所要求的对齐标准。在 TE 的 FSDP2 集成中，不支持的 MXFP8 切分会回退到高精度 [48]。其中，NVFP4 全收集会拒绝行数不能被 16 整除的分片。

4.2.3. 对称用户缓冲区

如第 2.2 节所述，重新物化以额外通信为代价换取更低的内存占用。降低通信开销的一种方法是对归约散射使用单次算法（先进行全到全交换，再进行本地累加），这样便可以在链路上传输低通信精度的数据，同时采用 FP32 累加（图 8（a））。不过，这需要额外的接收缓冲区（每个对等 GPU 各一个）、内存流量以及流式多处理器（SM）工作。将缓冲区注册到 NCCL，作为对称用户缓冲区（UB），则可启用 NVLink SHARP（NVLS）归约散射，并在交换机内进行 FP32 累加，从而无需这些接收缓冲区和单独的本地归约 [13, 45]（图 8（b））。

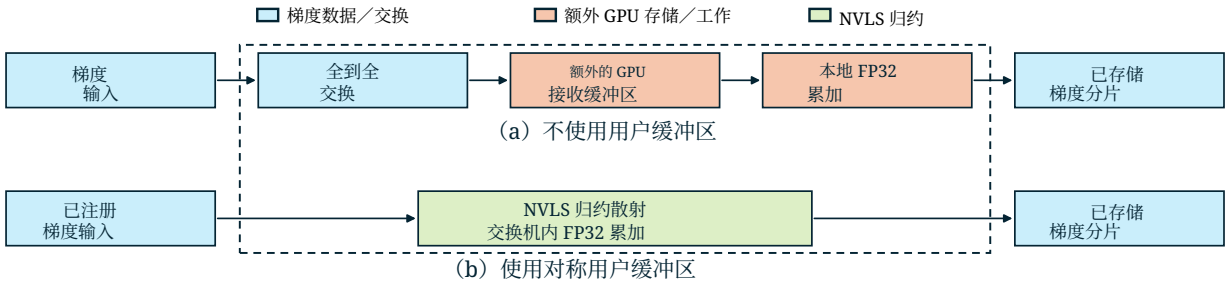
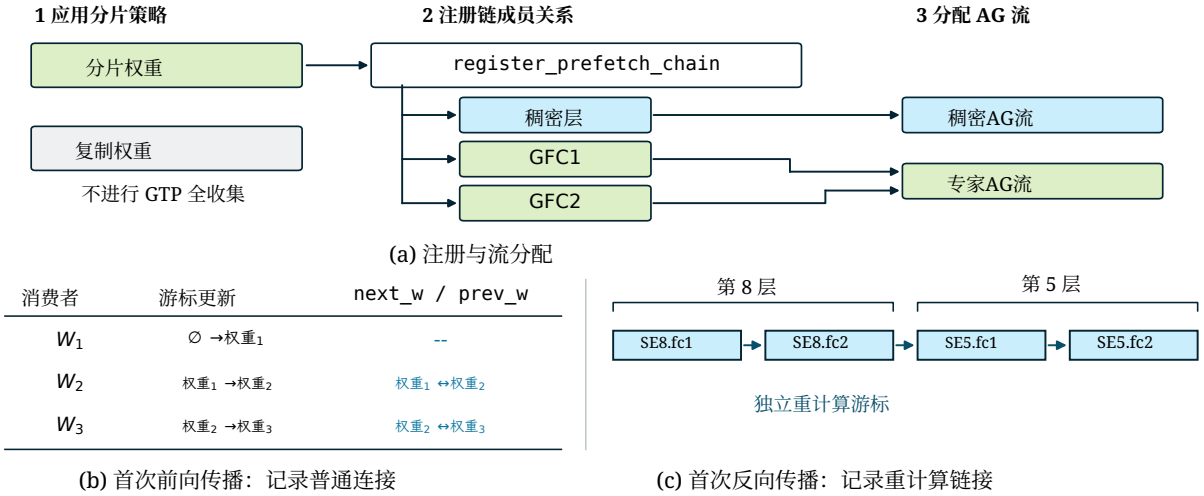


图 8 | 使用和不使用对称用户缓冲区时的梯度归约。两条路径都通信 BF16 梯度，并在 FP32 中进行累加。虚线区域显示了 NVLS 如何通过单个归约散射，替代全到全交换、额外的接收缓冲区和本地 FP32 累加。

GTP 可以只注册梯度归约散射的输入（UB RS），也可以同时注册存储的权重分片和全收集的张量并行分区（UB AG+RS）。注册权重缓冲区后，NCCL 的全收集也可以通过 NVLS 或直接内存访问（DMA）直接访问对等 GPU 的内存 [13]。第5.9节比较了两种模式的吞吐量和内存使用量。三种机制让对称用户缓冲区切实可用：

- **缓冲区注册。** GTP 为每个权重组从一个内存池分配对称用户缓冲区，并为稠密层权重和专家权重分别设置内存池，再将每个内存池注册到该组的 NCCL 中。注册是一种集合操作：组内所有 GPU 会一起注册各自的内存池。GTP 在模型构建期间、首次前向传播之前注册每个内存池。内存池扩容时，新分配的内存会自动注册。
- **缓冲区复用与同步。** 每个归约散射输入都保存来自 wgrad GEMM 的完整张量并行分区梯度，其大小是存储分片的 r 倍。因此，为每个权重都单独配置一个已注册的输入缓冲区，会占用与未分片梯度相同的内存量。GTP 通过一个按元素数量、数据类型和组分类的后进先出（LIFO）空闲列表复用已注册的缓冲区。即使形状不同，只要元素数量相同，权重就可以共用一个缓冲区。释放缓冲区时，会附带一个表示其归约操作已完成的完成事件。下一个分配该缓冲区的流会在覆盖其中的数据之前等待此事件，而无需进行设备级同步。
- **梯度直接写入。** 支持的 Transformer Engine 权重梯度矩阵乘法会将其输出直接写入图 8（b）中注册的梯度输入缓冲区，从而跳过额外的复制。该缓冲区已包含必要的填充行。其他通用矩阵乘法，以及输出数据类型与缓冲区不同的通用矩阵乘法，则会写入临时缓冲区，随后由 GTP 将其复制到已注册的缓冲区中。



(a) 注册与流分配

(b) 首次前向传播：记录普通连接

(c) 首次反向传播：记录重计算链接

图 9 | 预取链设置与预热。(a) 分片权重分别注册到独立的链中；GFC1 和 GFC2 共享一个专家全聚合流。(b) 前向传播消费者记录 `next_w/prev_w`链接并推进游标。(c) 共享专家（SE）重计算会学习独立的顺序，图示为 Nemotron-3 Super 的第 8 层，然后是第 5 层（从零开始的索引）。

4.3. 预取链注册与初始化

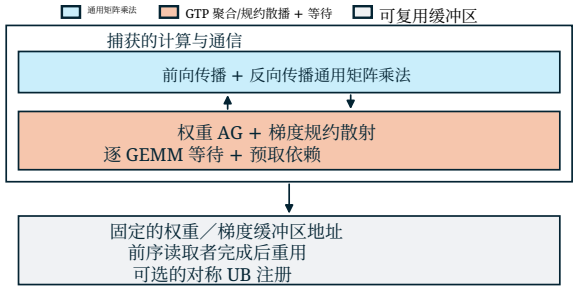
用户通过调用 `register_prefetch_chain` 并提供链名称和权重列表，将权重分配给链。随后，MCore 会将所有剩余的分片权重分配给第 3.3 节所述的默认链；复制权重不属于任何链。同一条链中的所有权重必须属于同一组，而分组线性算子会将一条链与其所有本地专家关联起来。同一组中的链可以共用一条通信流；例如，GFC1 和 GFC2 的全收集共用专家流（图 9（a））。

在首次前向传播期间，每条链都会记录每个权重的后继和前驱（图 9（b））。常规反向传播会沿前驱链接进行；激活重计算则需要单独的顺序。它会在反向传播期间重新运行前向传播 GEMM，因此按反向顺序遍历各层，但每层内的 GEMM 按前向顺序执行，并且只针对需要重计算的层。图 9（c）中，Nemotron-3 Super 先重计算第 8 层、再重计算第 5 层的共享专家，并在每层中先运行 FC1，再运行 FC2。首次反向传播会单独记录这一顺序；重计算会将按行布局全收集到自己的缓冲区中，与 dgrad GEMM 所用的按列布局分开。CUDA 图重放要求每个训练步都按记录的顺序访问每条链中的权重。

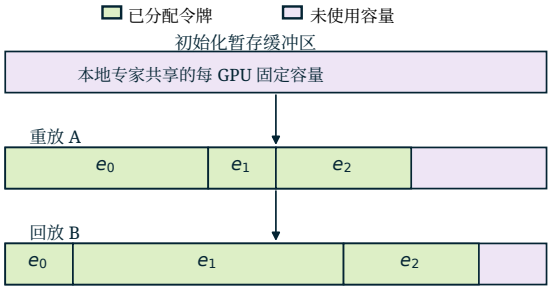
4.4. 全迭代 CUDA 图

GTP 使用全迭代 CUDA 图 [38]来减少内核启动和主机调度开销。要捕获一个训练步的各个微批次中的前向传播和反向传播，需要保留 GTP 的参数调度，并适应不同训练步之间变化的 MoE 令牌数量（图 10）。

在重放过程中保留 GTP 调度。 预热会建立第 4.3 节所述的预取顺序，并分配可复用的权重和梯度缓冲区。捕获会同时记录通用矩阵乘法、权重全聚合、梯度归约散射和累加，以及它们的流和事件依赖。因此，重放会保留逐 GEMM 等待和通信重叠（图 10（a））。每个缓冲区在重放过程中都保持其地址不变，并且只有在其



（a）保留 GTP 调度



（b）动态路由，固定容量

图 10 | 借助 GTP 实现全迭代 CUDA 图。(a) 捕获一次；使用固定缓冲区地址重放 GTP 依赖关系。(b) 各次重放中的令牌数量会变化，但缓冲区地址和容量保持固定，从而避免主机端调整大小；分段宽度表示本地专家的令牌数量 e_i。

之前的读取者完成读取后，缓冲区才会被重新使用。借助对称用户缓冲区，这些缓冲区来自 NCCL 注册内存池（第 4.2.3节）。在每个训练步中，新批次会在重放前复制到静态输入缓冲区中。与即时执行一样，DDP 在副本之间的通信也会被捕获到计算图中：更新后权重分片的全聚合与前向传播重叠，梯度归约散射与反向传播重叠。捕获的依赖关系确保每个更新后的分片都在 GTP 对其执行全收集之前到达，并确保 GTP 通信和归约散射在优化器运行之前完成。随后，优化器会即时运行，或在其自身的计算图中运行。

捕获动态 MoE 路由。路由决策和每专家令牌数量可能在训练步之间发生变化，而图重放要求缓冲区地址保持稳定。HybridEP [17] 和融合分组专家内核会预分配一个由本地专家共享的每 GPU 令牌容量，并在同一预算内以分页存储方式保存激活值 [71]。路由决策和令牌数量仍然是动态的，但分发、专家计算和激活值存储会使用预分配容量，无需在重放期间由主机端调整缓冲区大小（图 10(b)）。这种方式以容量填充和存储余量换取与计算图兼容的执行。如果路由所需容量超过预分配的令牌容量或分页存储容量，MCore 会丢弃本次尝试、删除计算图，并以即时执行方式重新运行同一批次，采用无丢弃分发且不使用分页存储；下一训练步会重新捕获计算图。容量上限针对令牌存储；GTP 仍会重放记录下来的专家权重预取顺序。

4.5. 分片权重上的 AdamW 和 Muon

AdamW [31] 独立更新每个元素，因此可以直接作用于存储分片。Muon [22]根据完整矩阵计算每次更新，而没有任何单个 GPU 存储完整矩阵，因此 GTP 必须决定在哪里执行该计算。第 5.8节衡量其对训练吞吐量的影响。

4.5.1. AdamW

MCore 的分布式优化器将每个存储分片视为普通参数，并在持有该分片副本的 c/r GPU 之间对其优化器状态进行分片。因此，每个 GPU 都持有$t^{1}\cdot r\cdot \frac{r}{c}=\frac{1}{n}$ 的每个权重的优化器状态，即使 $\overline{r}<\bar{c}$ 且权重本身仅分布在 $t\cdot r$ 个 GPU 上（第 2.3 节）。复制权重（第 4.1.1 节）的优化器状态则分布在 r 少的 GPU 上。用于裁剪的全局梯度范数会对每个元素仅计数一次，不计入其他副本上的重复元素和填充行。每次更新后，GTP 都会重新量化

每个存储分片，使其恢复为存储格式（第 4.2.1 节）。第 5.2 节比较了使用和不使用 GTP 时，AdamW 下的训练损失与验证损失。

4.5.2. Muon

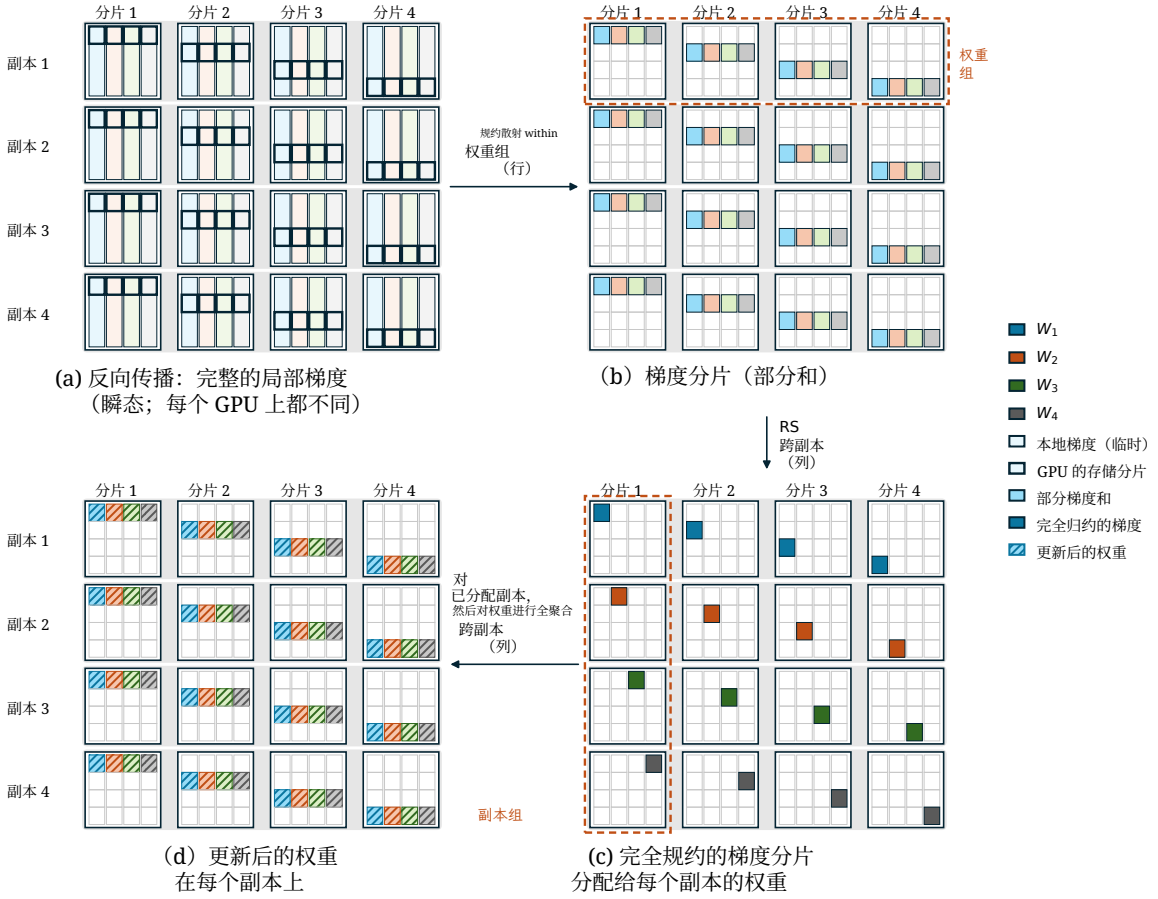


图 11 | 采用填充布局的 Muon 分层梯度归约，该布局会在副本间对梯度分片执行归约散射（16 个 GPU，$t = 1$，$r = 4$）。每个副本由四个 GPU 组成一个权重组，而四个权重（W_1–W_4）各有四个副本。顺时针方向：（a）临时完整本地梯度；（b）在权重组内分片的部分和；（c）分配到指定副本上的完全归约分片；（d）在副本间全聚合的更新后权重。在（a）中，粗边框标示存储分片。颜色用于区分权重；（a）和（b）中的梯度值在不同副本间各不相同。虚线边框标示集合通信组。

与 AdamW 不同，Muon 不会独立更新每个元素：它会对每个权重的整个 动量矩阵（即权重梯度的运行平均值）进行正交化，并执行固定次数的牛顿–舒尔茨（NS）迭代。然而，在 GTP 下，每个权重矩阵及其梯度都被拆分到 $p = t \cdot r$ 个 GPU 上：t 个张量并行分区，每个分区再拆分到由 r 个 GPU 组成的权重组中。因此，在分片权重上运行 Muon 需要额外通信，还需要选择由哪些 GPU 来计算每个矩阵更新。

MCore 的分布式优化器会将优化器状态分片到每个权重的 c/r 副本之间：它将每个矩阵的更新分配给一个副本，即矩阵的指定副本，其 p GPU 各自保存该矩阵动量的一个 $1/p$分片。梯度首先在每个权重组内进行规约散射，使每个 GPU 获得每个矩阵梯度的分片，该分片仅对该权重组的词元求和。随后，跨副本的梯度归约会将这些部分分片在所有副本上求和。一个

全归约会将每个矩阵的完全归约分片发送给每个 GPU，尽管每个副本只需要其负责更新的矩阵分片。归约散射通过仅向每个 GPU 提供其梯度缓冲区一个切片的完全归约值，避免了这种额外通信。它要求梯度缓冲区以与优化器状态相同的方式进行分片：每个 GPU 的切片必须恰好包含其副本负责更新的矩阵的梯度分片。 **填充布局** 通过按照负责更新它们的副本对每个 GPU 的梯度分片进行分组，并将各组填充至相同大小，确保了这一点，因此可以使用归约散射（图 11）。 **紧凑布局**用于我们的 DSV4-Pro Muon 评估，省略了这种填充并使用全归约。

在两种布局中，分配到的副本负责计算更新，然后每个优化器步骤一次，在各副本之间对更新后的权重分片执行全收集（图 11（c–d））。在分配到的副本内，每个矩阵的动量仍然分布在 p 各个 GPU 上，但 NS 作用于完整矩阵。MCore 支持三种在这些分片上运行 NS 的方式：

- **复制 NS** 对动量执行全收集，并在每个 p GPU 上运行 NS，从而精确复现未分片的 Muon。增加 p 并不会减少其每 GPU 的 NS 工作量（图12（a））。
- **分布式 NS**保持动量分片，并在每次 NS 迭代中对格拉姆矩阵执行全归约。每个 GPU 根据自己的分片计算部分格拉姆矩阵，然后通过全归约将它们求和。格拉姆矩阵在未分片维度上为方阵，因此当该维度较小时，其全归约开销最小。
- **按层分片的 NS** 为每个矩阵分配一个 *NS 主节点*，该主节点是 p GPU 之一，并使用能够平衡 NS 成本的贪心分配方式 [12]。打包的可变大小全互交换（A2Av）将动量分片发送到各主节点，由主节点分别计算一次更新，然后返回更新分片（图 12（b）），类似于 TorchTitan 中基于所有者的更新 [62]和 FlexShard [10]。

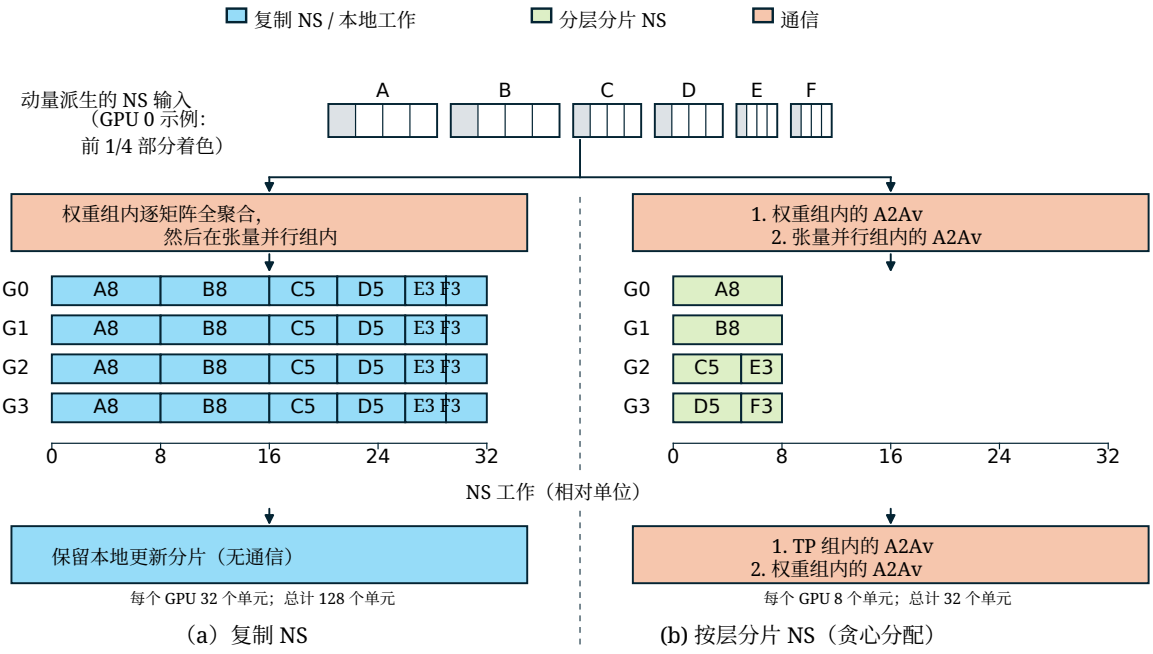


图 12 | 层分片消除了复制 NS 工作量。以带有 TP2 的示例性 GTP4 为例（$p = 4$、 $t = 2$、$r = 2$）：（a）逐矩阵全聚合使每个 GPU 都能计算全部六个更新（32 个单元/GPU）；（b）打包的可变大小全互换（A2Av）将输入路由到经过贪心均衡的归属 GPU，在那里每个更新仅运行一次（8 个单元/GPU），随后返回更新分片。张量宽度遵循示例性的 NS 成本尺度；阴影四分之一部分表示 GPU 0 的输入分片。

这三种方法分别在冗余计算（复制 NS）、每次 NS 迭代中的通信（分布式 NS），以及临时内存和负载均衡（按层分片的 NS）之间进行权衡：每个主节点都持有完整矩阵，且矩阵不能跨主节点分割。复制 NS 可作为数值参考；分布式 NS 与其之间的差异仅来自浮点舍入，而在我们的测试中，使用默认内核时，按层分片的 NS 在位级别上与其完全一致。

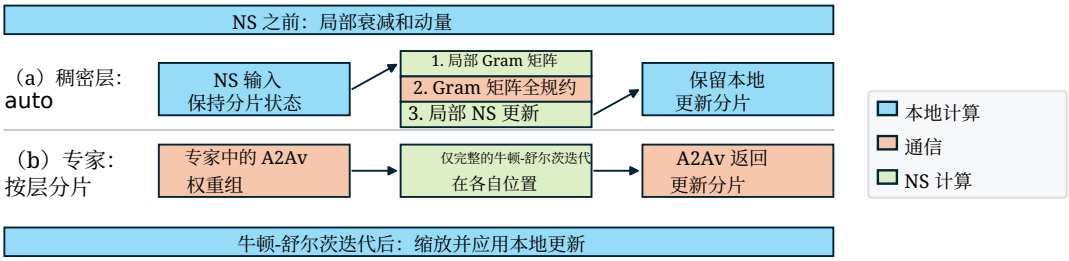


图 13 | 混合 Muon 策略。(a) 对稠密矩阵进行分布式 NS（成本模型选择此方案时）；每次 NS 迭代都会重复步骤 1–3。(b) TP1 下按层分片的专家。

在实际运行中，我们会分别为稠密权重和专家权重选择 NS 的运行方式（混合 *Muon*；图 13）。每个副本只分配到少量大型稠密矩阵，层分片难以在其 p GPU 之间实现良好均衡，因此 MCore 会根据矩阵形状、峰值算力和 NVLink 带宽构建成本模型，为每个稠密矩阵选择重复执行或分布式 NS。每个副本会分配到许多形状相同、易于均衡的专家矩阵，因此专家权重采用分层分片 NS。这样，每个优化器步骤只需进行少量打包的全对全通信，就能替代每次 NS 迭代中的 Gram 矩阵全规约，从而避免通信经过较慢的域间链路。

4.6. 检查点保存

GTP 通过 MCore 的分布式检查点保存功能来保存模型和优化器状态，该功能基于 PyTorch 的分布式检查点（DCP）库 [50]。每个保存的分片都会记录其在完整权重中的偏移量，因此可以使用不同的 TP、GTP、EGTP 和数据并行度来加载检查点。低精度权重以 BF16 格式保存。加载时会根据新配置填充逻辑行（第 4.2.2 节）；如果某个 GPU 上存储的分片完全落在填充区域内，则该 GPU 只会接收零值。

5. 评估

本节旨在回答以下有关 GTP 的主要问题：

- GTP 是否能达到与基线训练运行相同的损失曲线（第5.2节）？
- GTP 扩展至 12,288 个 GPU 时的表现如何（第5.3节）？
- 在最多 1,024 个 GPU 上，GTP 在吞吐量和内存方面与全分片数据并行和三维并行相比表现如何（第5.4–5.5节）？
- 第 3节和第 4节中的设计与实现选择如何影响吞吐量和内存（第 5.6–5.10节）？

5.1. 实验设置

硬件。我们使用英伟达 GB300 GPU，每个节点配备四块，每块约有 298 GB 高带宽内存（HBM）。这些 GPU 组成 72 GPU 的 NVLink 域 [46]，每个 GPU 的 NVLink 带宽为 900 GB/s，标称域间网络带宽为每个 GPU 100 GB/s。

实现与软件。GTP 已集成到 MCore 和 Transformer Engine 中（第 4 节）。我们使用 PyTorch 2.13、Transformer Engine 2.19 开发版、CUDA 13.3 和 NCCL 2.31.2。我们使用 Emerging Optimizers 0.3。1 [23, 39] 用于实现 Muon。

基线。 我们将 GTP 与 TorchTitan 中的 MFSDP [43] 和 PyTorch FSDP2 [51, 54]进行比较。二者都会对训练状态进行分片，并在每个模块执行前后重新物化参数。我们还将其与三维并行（三维并行包括 TP、PP 和 DP；对于混合专家层，还结合使用 EP）进行比较，后者采用了基于 Megatron Bridge [40]调整的方案。每项吞吐量比较均匹配模型配置、GPU 数量、序列长度、全局批量大小、计算精度，以及激活卸载和重计算策略。VPP 表示虚拟（交错式）流水线并行 [35]。后缀表示某种并行方式的并行度；例如，GTP64 表示 64 路密集参数分片，EGTP4 表示 4 路专家参数分片。未指定的并行度均为 1。MBS 和 GBS 分别表示以序列数计的微批次大小和全局批次大小，GA 表示每个训练步中的梯度累积微批次数。除非另有说明，性能基准测试均使用随机初始化、AdamW、MXFP8 精度以及强制混合专家负载均衡。

模型	总计参数	激活参数（完整/代理）	层	专家 / top-k	h/ h'	序列长度
NT3-Nano [36]	31.6B	3.2B	52 / –	128 / 6	2688 / 1856	8192
DSV3 [6]	671B	37B	61 / 30	256 / 8	7168 / 2048	4096
DSV4-Pro [7]	1.6T	49B	61 / 30	384 / 6	7168 / 3072	4096
Internal-2T	2T	130B	164 / 55	512 / 10	3072 / 10240	8192； 2^{16}–2^{20}

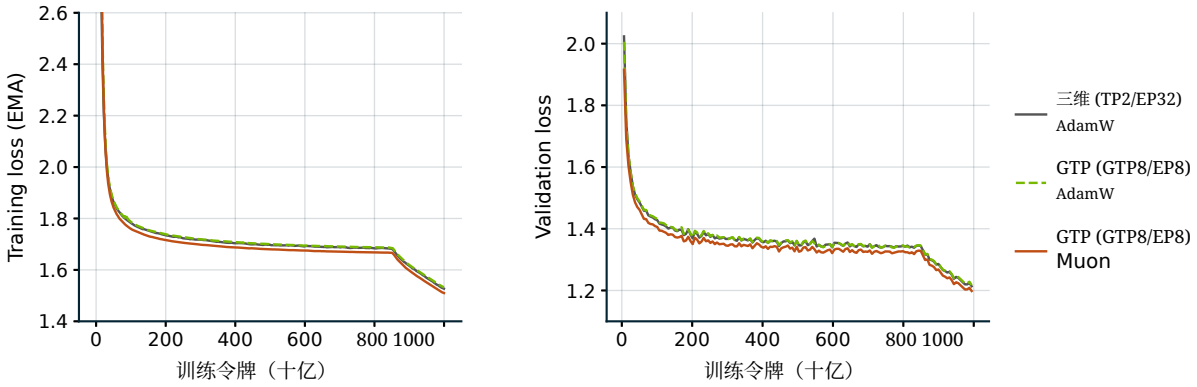
表 3 | 基准模型配置。参数数量指完整模型的参数量；活跃参数指每个词元使用的参数。隐藏维度为 h，每个专家的中间维度为 h'（第 2.1 节）。代理保留层维度；– 表示没有代理。层数不包括多词元预测（MTP）层 [6]：DSV3 和 DSV4-Pro 各有一个 MTP 层；在横向扩展实验中，Internal-2T 有两个 MTP 层，而在代理和长上下文实验中没有 MTP 层。序列长度以词元为单位。

工作负载。我们在 DeepSeek-V3 (DSV3) 上评估 GTP [6], DeepSeek-V4-Pro(DSV4-Pro) [7],和 Internal-2T（一个内部 2T 参数模型）。表3汇总了完整模型配置和代理模型的深度。我们分别用 $K = 2^{10}$ 和 $M = 2^{20}$表示词元数和批次大小；例如，8K 表示 8,192 个词元。为了适配更少的 GPU，部分比较使用层数更少但层维度不变的代理模型。我们在 Nemotron-3-Nano (NT3-Nano) 上检查正确性 [36]。

性能指标。我们报告**词元/秒/GPU**和**模型 TFLOP/秒/GPU** [41], ，后者统计前向传播和反向传播的浮点运算次数，不包括重计算。为了比较 Megatron 和 TorchTitan，我们使用词元/秒/GPU，因为两者原生的模型浮点运算量估算器有所不同，包括因果注意力计算方式 [44, 55]。吞吐量取预热期后各训练步的平均值。每 GPU 内存表示设备内存总用量，包括 PyTorch 缓存分配器已预留但当前未分配的内存，单位为十进制 GB；我们报告所采样 GPU 中观测到的最大用量。

性能分析指标。 显式通信是指在性能分析中的 GPU 上至少有一个通信内核运行且没有计算内核运行的总时间。除反向传播阶段的全聚合外，我们将其表示为步时间的比例；对于反向传播阶段的全聚合，则表示为任意反向传播阶段全聚合运行期间所占的比例。通信类别可能相互重叠，因此其暴露比例之和不等于总量。

5.2. 损失曲线



（a）训练损失　　（b）验证损失

图 14 | Nemotron-3-Nano 在 1T 个词元上的损失曲线（8K 个序列，GBS768）。AdamW 运行对比了 768 个 GPU 上的三维并行与 256 个 GPU 上的 GTP。训练损失采用指数移动平均进行平滑；验证损失每 1,000 个训练步评估一次。GTP 使用密集参数分片（GTP8），不使用专家参数分片。额外的 GTP–Muon 曲线使用不同的优化器，且没有匹配的三维并行基线。

图14 比较了使用和不使用 GTP 时 Nemotron-3-Nano 的训练损失与验证损失。GTP AdamW 运行会直接对存储分片应用 AdamW，具体如第 4.5.1 节所述。在经过 158,900 个训练步（约 1T 个词元）的 Nemotron-3-Nano 训练中，GTP 与基线之间的验证损失平均绝对差为 0.0022 纳特，平均绝对相对差为 0.16%，统计于 158 个匹配的验证检查点。两次 AdamW 运行均使用 MXFP8；基线在 768 个 GPU 上使用 TP2 和 EP32，而 GTP 在 256 个 GPU 上使用 GTP8 和 EP8。图中还包括使用 Muon 的 GTP，以展示其对优化器的支持；由于没有匹配的三维并行基线，因此不应将其较低的损失归因于 GTP。我们还在 100B 和 500B 参数的内部模型上，通过 EGTP 获得了匹配的损失曲线。

5.3. 大规模吞吐量

（a）弱扩展
每个 *GPU* 一个序列

GPU	GBS	每秒万亿次浮点运算总计	每个 GPU（每秒百亿亿次浮点运算）	扩展效率
4,096	4,096	1,079	4.42	–
8,192	8,192	1,084	8.88	100.5%
12,288	12,288	1,060	13.03	98.2%

（b）强扩展
固定全局批次大小为 *12,288* 个序列

GPU	全局批次大小	每秒万亿次浮点运算 总计	每个 GPU（每秒百亿亿次浮点运算）	扩展效率
6,144	12,288	1,156	7.10	–
12,288	12,288	1,060	13.03	91.7%

表 4 | Internal-2T 大规模训练吞吐量。GBS 表示序列数。每 GPU 吞吐量是指在一次运行的各个日志记录间隔中，每个 10 步间隔的平均吞吐量的中位数。总吞吐量为每 GPU 吞吐量乘以 GPU 数量，单位为模型 EFLOP/s。

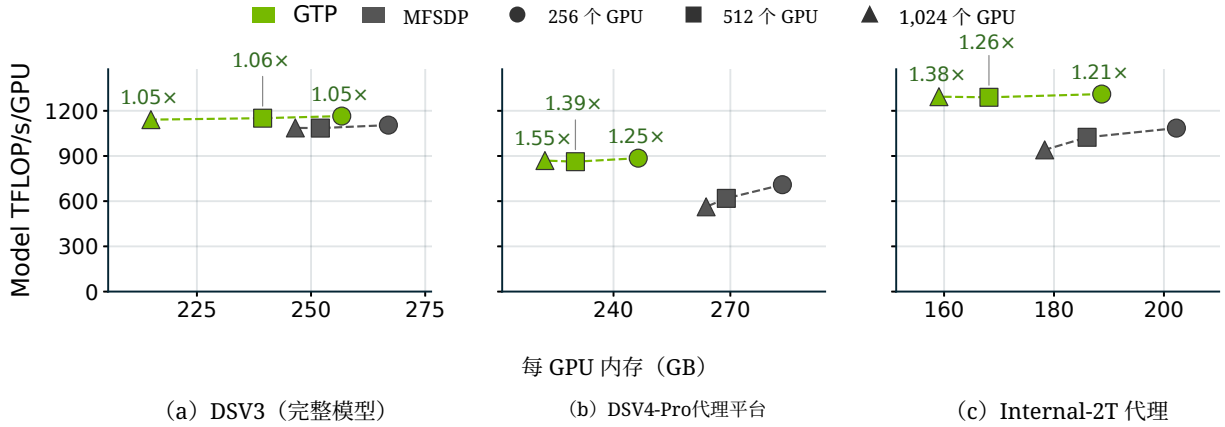
借助 GTP，Internal-2T 训练达到 **13.0 模型 EFLOP/s**（1,060 TFLOP/s/GPU），在 12,288 个 GPU 上采用 PP1、MBS1 和 8K 序列 且无需强制负载均衡（表 4）。这些完整模型运行采用 Muon（动量 0.9，使用 Polar Express 系数进行 16 次 NS 迭代 [1]），并使用即时执行。扩展规模会增加数据并行副本的数量，同时保持 GTP64、EGTP8 和 EP64 不变。当每个 GPU 使用一个序列（ GA1 ）时，从 4,096 个 GPU 扩展到 12,288 个 GPU 的弱扩展

GPU 的效率为 **98.2%**。在固定 GBS 为 12,288 个序列的情况下，将 GPU 数量从 12,288 减半至 6,144（将 GA 加倍至 2）可使每GPU吞吐量提高 9%。

5.4. 与 FSDP 实现的比较

我们将 GTP 与两种 FSDP 实现进行了比较：在三个模型上从 256 个 GPU 弱扩展至 1,024 个 GPU 的 MFSDP，以及 TorchTitan 中的 PyTorch FSDP2；后者是在 256 个 GPU 上针对 DSV3 代理进行的匹配比较。GTP 的吞吐量达到 MFSDP 的 1.05–1.55× 倍，同时每 GPU 内存低 4–16%；吞吐量达到 FSDP2 的 1.04× 倍，同时每 GPU 内存低 16%。

5.4.1. 与 ***MFSDP*** 的弱扩展



（a）DSV3（完整模型）　（b）DSV4-Pro代理平台　（c）Internal-2T 代理

图 15 | GTP 与 MFSDP（GA1）的弱扩展吞吐量和内存：(a) 完整 DSV3，(b) DSV4-Pro 代理平台，以及 (c) Internal-2T 代理。吞吐量单位为模型 TFLOP/秒/GPU；标签显示在相同 GPU 数量下 GTP 相对于 MFSDP 的加速比。虚线按照 GPU 数量顺序连接每种方法的数据点，以展示弱扩展趋势。MFSDP 基线基于 MCore 0.20.0 开发代码 [43]，并包含针对具体模型的扩展和修复。

我们在完整 DSV3、DSV4-Pro 和 Internal-2T 代理平台上，将 GTP 和 MFSDP 从 256 个 GPU 弱扩展至 1,024 个 GPU（图15(a–c)）。DSV3 和 DSV4-Pro 使用 MBS2 和全迭代 CUDA 图；Internal-2T 代理使用 MBS1 和即时执行。所有比较均使用 EP64。GTP 在 NVLink 域内的 64 个 GPU 上分片稠密权重；MFSDP 在 64 个 GPU 上分片参数和梯度，并在所有 GPU 上分片优化器状态。GTP 还使用 EGTP4，而 MFSDP 不会在专家并行之外进一步分片专家权重。

在 256 个 GPU 上，GTP 的吞吐量分别达到完整 DSV3、DSV4-Pro 代理平台和 Internal-2T 代理平台上 MFSDP 吞吐量的 **1.05**×、**1.25**× 和 **1.21**×（图15(a–c)）。在 256–1,024 个 GPU 的范围内，GTP 的每 GPU 内存分别比 MFSDP 低 4–13%、13–16% 和 7–11%。

GTP 相较于 MFSDP 的优势。 在评估的 MFSDP 配置中，反向钩子会等待一个 Transformer 块（DSV3 和 DSV4-Pro）的所有参数全收集完成，或对于 Internal-2T，等待一对连续层（线性注意力层或全局 softmax 注意力层，以及其后的混合专家层）的所有参数全收集完成，然后才开始该块的反向计算。相较之下，每个 GTP 通用矩阵乘法只需等待自身的权重（第 3.1 节）。这些 MFSDP 配置不会在专家并行之外进一步分片专家权重，因此从不对专家权重执行全收集，且 GFC1 从不等待 GFC2 的权重。在使用 256 个 GPU 的 DSV4-Pro 上，性能分析记录显示，GTP 的反向传播全聚集时间中有 20% 暴露出来，而 MFSDP 为 80%；同时，GTP 的反向传播缩短了 18.3%。在

DSV3 上，均衡路由下每个微批次中每个专家的令牌数多于 DSV4-Pro 和 Internal-2T（分别为 16,384、8,192 和 10,240）。每个专家的令牌数越多，每个全收集字节所对应的计算量就越大（公式 4），从而留给 GTP 通过调度改进重叠的空间更小。

GTP 的优势会随着规模扩大而更加显著。从 256 个 GPU 扩展到 1,024 个 GPU 时，GTP 在 DSV3、DSV4-Pro 和 Internal-2T 上的弱扩展效率分别为 **98.0%、98.2% 和 98.7%**。MFSDP 在 DSV4-Pro 和 Internal-2T 上的效率仅为 79.3% 和 86.7%，因此在 1,024 个 GPU 上，GTP 在这些模型上的吞吐量分别达到 MFSDP 的 1.55× 倍和 1.38× 倍。在 DSV4-Pro 上，从 256 个 GPU 扩展到 1,024 个 GPU 时，两种方法的计算时间变化均不到 2%，因此 MFSDP 的步时间增长是由显式通信时间增加造成的，且主要发生在反向传播阶段。MFSDP 在 1,024 个 GPU 上的反向传播显式全聚集时间远高于在 256 个 GPU 上的时间（性能分析步中为 1.26 秒对 0.34 秒），且在不同训练步之间波动很大。在 512 个 GPU 上，这一时间在不同训练步之间为 390 至 932 毫秒，其中几乎所有差异都耗费在等待每个组中的最后一个 GPU 开始每次全聚集上。GTP 的反向传播显式全聚集时间从 256 个 GPU 到 1,024 个 GPU 几乎保持不变。

5.4.2. 与 *PyTorch FSDP2* 的匹配对比

我们在 TorchTitan [51, 54]上使用 256 个 GPU 的 DSV3代理，将 GTP 与 PyTorch FSDP2 进行比较，因为完整的 DSV3 无法适配所评估的 FSDP2 配置。两者均使用 MBS2、GBS512、GA1、EP64 和 64 路密集分片。GTP 使用 EGTP4；TorchTitan（搭配本地密集网格适配器）也将专家权重分为四路。GTP 和 TorchTitan 均将多头潜在注意力（MLA）的下投影设为副本。对于分片不符合 MXFP8 对齐要求的稠密权重，TorchTitan 会退回使用 BF16全聚合；其专家全聚合则使用 MXFP8。

方法	词元/秒 每个 GPU	相对的 rate	内存 (GB/GPU)
PyTorch FSDP2（TorchTitan）	8,503	1.00×	219
GTP	8,862	1.04×	183

表 5 | 256 个 GPU 上 DSV3代理的吞吐量和内存。相对速率以 FSDP2 为基准进行归一化。

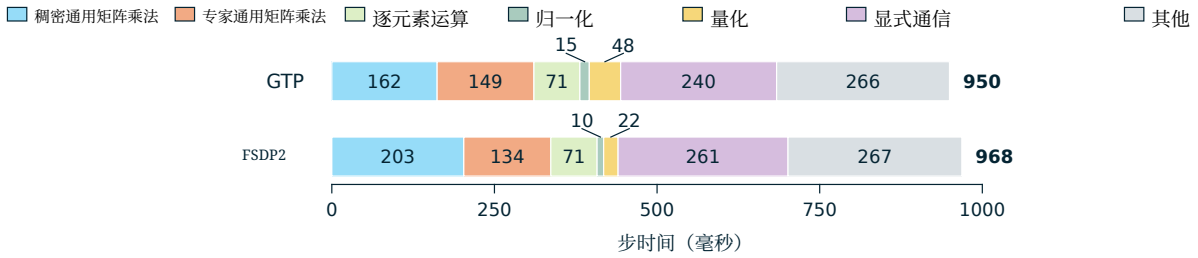


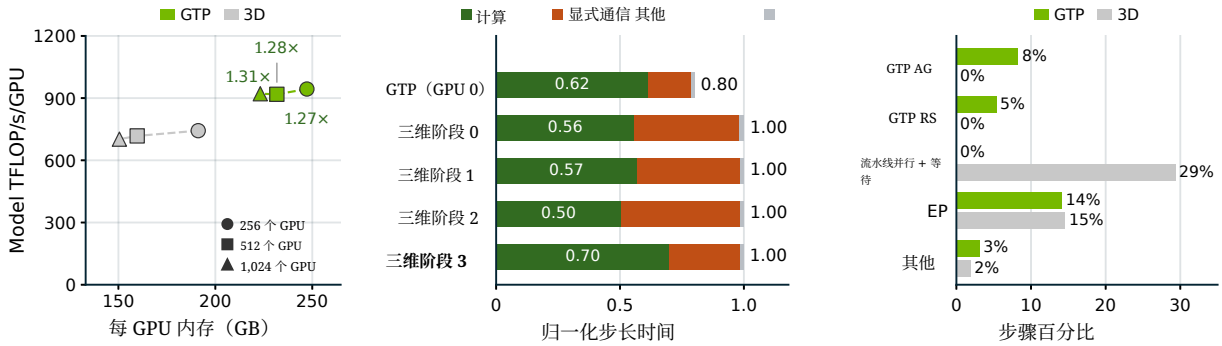
图 16 | 256 个 GPU 上 DSV3代理的步时间分解。步中的每个时间区间都归入一个类别，因此各分段之和等于步时间。量化统计独立内核；FSDP2 的专家GEMM还包括融合的 EP 令牌交换和量化。逐元素运算涵盖其余的逐点运算和归约内核。其他包括注意力、优化器工作量、复制操作和空闲时间。

吞吐量高于 FSDP2，内存占用低于 FSDP2。GTP 达到 1。04× FSDP2 的吞吐量（表 5）。完整步性能分析记录显示，稠密GEMM和显式通信耗时更少，但专家GEMM和独立量化耗时更多（图 16）。量化耗时的差异（48 对比

22 毫秒）部分反映了融合边界：TorchTitan 的 Dist-MoE后端 [33] 将 EP 交换和输入量化纳入专家GEMM，而 GTP 则分别执行 HybridEP内核 [17]和量化器。归一化耗时分别为 15 毫秒和 10 毫秒，而两种方法的剩余逐元素运算耗时均约为 71 毫秒。

与 FSDP2 相比，GTP 的 GPU 内存占用减少了 16%（每个 GPU 为 183 GB，而非 219 GB）。GTP 通过 MXFP8 权重全聚合，并在全聚合缓冲区和归约散射缓冲区最后一次使用后将其释放，从而限制临时存储空间；TorchTitan 对未对齐的稠密分片执行 BF16 全聚合时，则需要更大的缓冲区。

5.5. 与三维并行的比较



(a) 吞吐量和内存　(b) 步时间分解　(c) 显式通信

图 17 | ：DSV4-Pro 代理平台上的 GTP 与三维并行（GTP 使用 MBS2；三维并行使用 MBS1）。(a) 在 256–1,024 个 GPU 上的弱扩展；标注显示 GTP 的加速比。(b–c) 在 GBS1K 下分别进行的 256-GPU 运行的性能分析记录：(b) 相对于 3D 阶段 0 归一化的步时间；(c) 显式通信占各方法步时间的比例（GTP 采用 GPU 0；三维并行采用阶段 0）。其他： (b) 中的副本和空闲时间；(c) 中的 DP 通信及剩余同步；流水线并行包括对等端等待。

我们在 256–1,024 个 GB300 GPU 上，使用 DSV4-Pro代理平台比较 GTP 与三维并行。在弱扩展中，我们将每个 GPU 的工作量固定为每个训练步 4 个 4K 序列，因此在 256 个 GPU 上得到 GBS1K。三维并行采用 TP1、PP4、VPP4 和 EP64，每个阶段分别包含 8、8、7 和 7 层；GTP 采用 GTP64、EGTP4 和 EP64。两种方法均使用全迭代 CUDA 图，将核心注意力激活值卸载到主机内存，并选择性地重新计算 MLA上投影和流形约束超连接（mHC） [69]。

GTP 与 PP 面临不同的微批次权衡。 我们在相同的硬件内存约束下，调整每种方法的 MBS 以获得更高吞吐量。更大的微批次可以提高两种方法的 GEMM效率，并帮助 GTP 隐藏权重全聚合。然而，在 GBS 固定时，更大的微批次会使 PP 用于填充流水线的微批次减少，并增加激活内存。在 256 个 GPU 和 GBS1K 下：

- **在微批次大小相同的情况下，GTP 可节省内存。**与三维并行相比，GTP 在 MBS1 和 MBS2 下分别少使用**11% 和 13% 的内存**（每个 GPU 分别为 169 对 191 GB 和 247 对 285 GB）。GTP 不会为多个在途微批次保留激活值，并且相比 TP1 配合 PP4，GTP 能将稠密层权重划分得更细。
- **GTP 以更高的内存使用量换取更高的吞吐量。**将 MBS 从 1 增至 2，可使 GTP 的吞吐量从 601 提高到 944 模型 TFLOP/秒/GPU，内存使用量则从 169 增至 247 GB/GPU。对于三维并行，同样的调整会使吞吐量从 743 降至 698 模型 TFLOP/秒/GPU，同时使内存使用量从 191 增至 285 GB/GPU。

在扩展对比中，我们对 GTP 使用 MBS2，对三维并行使用 MBS1，因此 GTP 每个训练步包含两个微批次，而三维并行的每条流水线包含 16 个微批次。采用这些设置后，GTP 的吞吐量达到**1.27–1.31× 倍于三维并行**，覆盖 256–1,024 个 GPU，同时使用**多出 29–48% 的内存**（图17（a））。

阶段不平衡会拖慢三维并行。 为考察这一优势，我们使用了两种方法在 256 个 GPU 和 GBS1K 下的性能分析记录（图 17（b–c））。每 GPU 计算时间相近：GTP 每步约 2.3 秒，而三维并行基线的流水线阶段平均每步为 2.1 秒。不过，这些阶段并不均衡：最后阶段的计算时间比阶段平均值高出**20%**，而流水线的运行速度取决于最慢的阶段。其他阶段会花费这部分差值进行等待，这表现为流水线并行传输和对等端等待：阶段 0 的每步中有 29% 用于此，相比之下，理想的阶段均衡气泡仅为 4.5% [35]。GTP 无需平衡阶段，因此其步长**缩短了 20%**，尽管其权重全聚合和梯度归约散射分别占用了该步的 8% 和 5%。

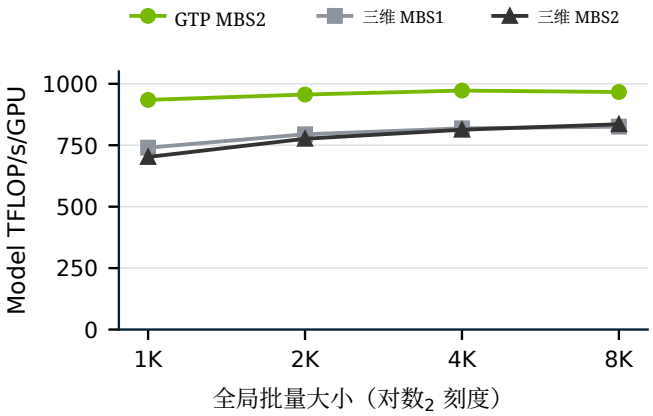


图 18 | 在 256 个 GB300 GPU 上，使用 4K 序列时，随着 GBS 变化，DSV4-Pro代理平台的吞吐量；GTP 使用 MBS2，三维并行使用 MBS1 和 MBS2。

更大的全局批次会缩小 GTP 的吞吐量优势。 在 256 个 GPU、4K 序列的条件下，将 GBS 从 1K 增加到 8K，可使三维并行（在 MBS1 和 MBS2 中取最佳值）的最高吞吐量提升 13%，而 GTP 仅提升 3%，因为更大的批次能将流水线填充与排空分摊到更多微批次上（图 18）。GTP 的优势从 **26% 缩小到 16%**，但在每个测量的 GBS 下仍然更快。剩余差距源于上文所述的阶段不平衡。改变 MBS 并不会改变这一点：使用 MBS2 时，三维并行在不同 GBS 下相较于使用 MBS1 的性能从慢 5% 到快 1% 不等，并且每个 GPU 约多使用 90 GB 内存。这些运行与图 17中的运行相互独立，因此绝对数值略有不同。

5.6. 长上下文训练

为评估长上下文训练，我们在 512 个 GPU 上，将完整 Internal-2T 的序列长度从 8K 调整到 1M 词元，同时保持每个训练步处理 400万令牌（表 6）。随着序列长度增加，我们将 CP度数从 1 提高到 128，使每个 GPU 始终处理 8K 词元。所有测试配置均使用 AdamW、GTP64、EGTP8、EP64、MBS1，以及相同的激活重计算和卸载策略。

使用 CP128 时，完整的 Internal-2T 运行以每个 GPU 283 GB 的显存，使用 100 万词元序列执行训练步。从 8K 到 100 万，模型吞吐量下降 30%，从 **1,258 降至 883 TFLOP/s/GPU**，而步时间则从 **5.1 秒升至 35.8 秒**（表6）。

序列长度	上下文并行	全局批次大小	Step 时间（秒）	内存（GB/GPU）	模型 TFLOP/s/GPU
8K	1	512	5.1	272	**1,258**
64K	8	64	7.0	280	1,127
256K	32	16	12.4	290	1,018
1M	128	4	35.8	283	**883**

表 6 | 在 512 个 GPU 上，每步固定处理 400 万词元时的长上下文结果，并强制采用均衡路由和即时执行。全局批次大小表示每步处理的序列数。

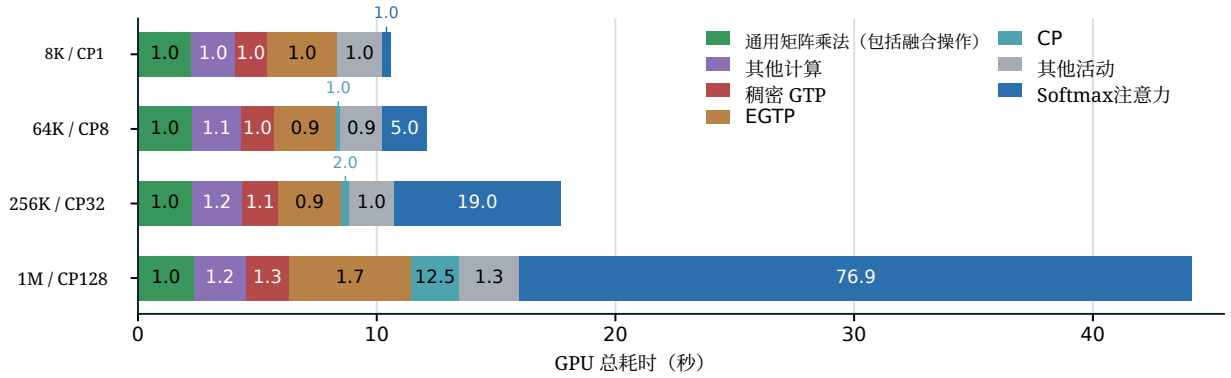


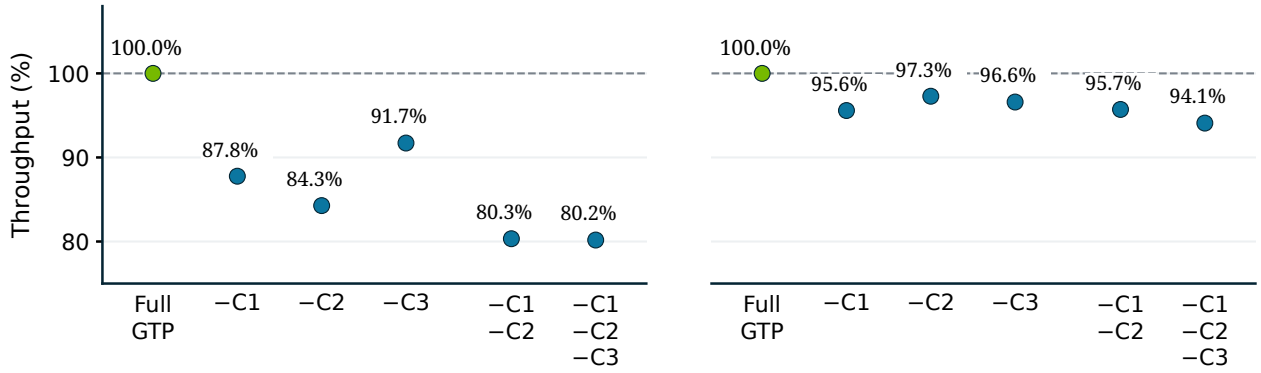
图 19 | 在 512 个 GPU 上进行 Internal-2T 长上下文扫描的 GPU时长分解，**未扣除重叠部分**。宽度之和表示通用矩阵乘法和复制时长；总时长并非步骤耗时。数字标签显示各组件相对于其 8K 值的归一化时长，但 CP 除外：由于 8K 没有 CP通信，CP 以 64K 为基线。其他计算组别包括通用矩阵乘法和注意力之外的计算；其他活动组别包括 HybridEP、副本间的梯度同步、其余集合通信和复制。

Softmax注意力推动了计算量的增长。在每步词元数固定时，softmax 注意力需要 T 词元 $O(TL)$ 序列长度为 L 时的计算量。Softmax注意力耗时从 15.3× 64K 增至 1M，序列长度增长了 16×（图 19）。在整个测试范围内，通用矩阵乘法耗时变化不到 5%；而以 BF16 运行的 Softmax注意力占 GPU 总活动时间的比例从 3% 增至 64%，压缩了用于更快的 MXFP8 GEMM 的时间占比。

外层CP会增加跨域流量。在 256K 下，每个 CP32 组都位于一个 NVLink 域内。在 1M 下，每个序列跨越两个 64-GPU 权重组，从而在两个 NVLink 域之间形成 CP128，而稠密 GTP64 保持在各自域内（第 4.1.2节）。

- **CP通信不断增加，并且在 1M 下很少与计算重叠**。从采用 CP8 的 64K 到采用 CP128 的 1M，CP总时长不断增加 12.5×（图19），这是因为 softmax 注意力键值对以及线性注意力层的循环状态的交换需要跨越更多序列分区，并通过域间速度较慢的 InfiniBand（IB）链路。在 1M 下，CP 时长中有 99.8% 没有并发计算，但其中 47.6% 与其他通信重叠。
- **在 1M 配置下，权重全聚合耗时更长，部分原因是需要等待其他 GPU**。在 1M 配置下，稠密 GTP 和 EGTP 的总耗时达到 1.3× 以及 1.7× 其 8K 配置中的数值（图 19），尽管权重载荷和集合通信次数均未改变。每个 EP64组占满一个 NVLink 域，因此每个 EGTP8组横跨全部八个域，也就横跨四个 CP128组；其全聚合需要等待这些组中最慢的一个，并与 CP 共享 IB链路。在没有并发计算的情况下，EGTP 增加的耗时约为 35.8 秒步耗时中的 1 秒。

5.7. 逐 GEMM 调度消融



(a) DSV4-Pro代理平台　　(b) Internal-2T 代理

图 20 | 在 256 个 GPU 上，对 (a) DSV4-Pro 代理平台和 (b) Internal-2T 代理进行调度消融，并以各模型完整 GTP（100%）为基准进行归一化。数据点表示完整 GTP、分别移除 C1、C2 和 C3，以及依次移除 C1 和 C2、再移除全部三项的结果。

我们在两个基准模型上，分别单独移除并累计移除独立的专家预取链（C1）、消费者专属等待（C2）和专家权重合并（C3），同时固定所有其他设置（图20（a–b））。两者均使用 GTP64、EGTP4、EP64 和 GA1；DSV4-Pro 使用 MBS2 和 GBS512，而 Internal-2T 使用 MBS1 和 GBS256。DSV4-Pro 使用全迭代 CUDA 图；Internal-2T 使用即时执行。

移除 C1 后，GFC1 和 GFC2 共用一条专家预取链。移除 C2 后，使用了预取权重的通用矩阵乘法还必须等待该权重的全收集流中已排队的所有任务完成，其中包括为其他通用矩阵乘法进行的预取。移除 C3 后，每个本地专家权重都会单独进行全收集，而不是与输入同一分组通用矩阵乘法的其他权重合并处理。

每种机制都能提升两种工作负载的吞吐量。 分别移除 C1、C2 和 C3 后，DSV4-Pro 的吞吐量会下降**12.2%、15.7% 和 8.3%**，而 Internal-2T 的吞吐量会下降**4.4%、2.7% 和 3.4%**；在 DSV4-Pro 上，C2 带来的单项损失最大；在 Internal-2T 上，则是 C1 带来的损失最大。

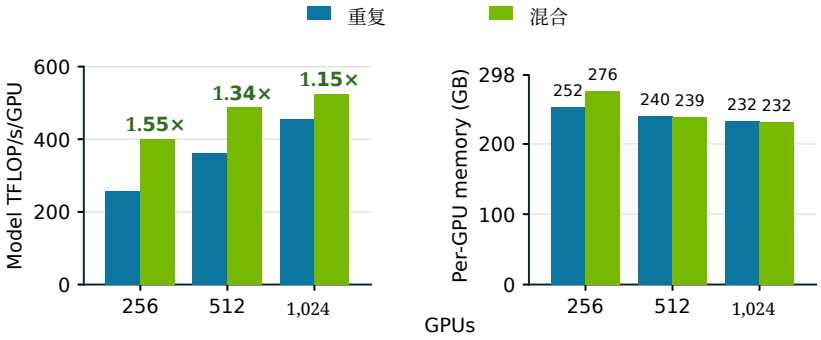
DSV4-Pro 上的收益更大。 移除全部三种机制后，吞吐量会下降 **DSV4-Pro 上下降 19.8%**，而 **Internal-2T 上下降 5.9%**。在均衡路由下，Internal-2T 每个微批次中每个专家处理的词元多 25%（10,240 个，而非 8,192 个），因此每个全收集权重对应的算术运算量更大（公式 4）。它较小的调度损失与更多计算能够掩盖通信开销这一点相符，不过两者的模型和执行模式也都不同。

各项收益并非简单相加。 单独移除各机制造成的损失总计为 DSV4-Pro 上的 36.2% 和 Internal-2T 上的 10.5%，但移除全部三种机制仅分别造成 19.8% 和 5.9% 的损失。在移除 C1 和 C2 后，再移除 C3 会使 DSV4-Pro 的吞吐量从完整 GTP 的 80.3% 降至 80.2%；这一差异太小，若不进行多次运行便无法确认。Internal-2T 的吞吐量则会从 95.7% 降至 94.1%。

5.8. 分片权重上的 Muon

我们比较了在 DSV4-Pro代理平台上，256、512 和 1,024 个 GPU 规模下的混合 Muon（第 4.5.2 节）与复制 NS（图 21）。这三种规模均使用 8K 序列、GTP64、EGTP4、EP64、MBS1 和 GA1。

两种 Muon 模式均使用 0.95 的动量、十次 NS 迭代以及 DeepSeek-V4 的系数 [7]。CUDA 图已禁用。



（a）模型吞吐量　　（b）每 GPU 内存

图 21 | DSV4-Pro代理平台上的重复 Muon 和混合 Muon（8K；GBS 等于 GPU 数量；每个数据点运行一次）。（a）模型吞吐量，其中标注了混合方案的令牌吞吐量加速比。（b）每 GPU 峰值内存。

混合 Muon 在每种规模下都能提升吞吐量，但增益逐渐减小。 混合 Muon 将令牌吞吐量提升了 1.55× **在 256 个 GPU 时，** 1.34× **在 512 个 GPU 时，以及** 1.15× **在 1,024 个 GPU 时**，随着 GPU 数量和 GBS 同步扩展（图 21（a））。这些运行测量的是合并配置，未将稠密层和专家部分分开。

副本数量越多，每 GPU 的 NS 工作量越少。 在 256、512 和 1,024 个 GPU 配置下，每个稠密权重分别有 4、8 和 16 个副本，每个专家权重分别有 1、2 和 4 个副本，而每 GPU 令牌数和分片度保持不变。每个副本更新的矩阵份额更小（第 4.5.2 节），因此在两种模式下，每个 GPU 执行的 NS 更新都更少。优化器工作量减少会缩短步时，而前向与反向传播工作量基本保持不变，因此每 GPU 吞吐量提高。每个 GPU 的 NS 工作量减少，也会缩小混合 Muon 的相对优势（图 21（a））。

层分片带来的额外内存开销会随着规模扩大而缩小。 在 256 个 GPU 配置下，混合 Muon 使用的设备内存比复制 NS 多约 23 GB（图 21（b））。每个 NS 主节点都会暂时保存完整的 FP32 矩阵和交换缓冲区（第 4.5.2 节）。随着 GPU 数量增加，分配给专家权重的每个四 GPU 权重组中的专家矩阵数量从 372 降至 186，再降至 93，从而缩小这些缓冲区。在 512 和 1,024 个 GPU 配置下，两种模式的设备内存占用大致相当。

5.9. 对称用户缓冲区

我们在 Internal-2T 代理上，以 256 个和 512 个 GPU（图 +22）比较稠密 GTP、关闭 UB、UB RS 和 UB 全聚合（第 4.2.3 节）。专家权重归约散射仍采用全对全路径，DDP 的梯度集体通信不使用 UB。包括即时执行在内的所有其他设置在各种模式下均保持不变。

更高的吞吐量与更低的内存使用量。 在 256 个 GPU 上，UB RS 和 UB AG+RS 相比关闭 UB，吞吐量分别提升 4.0% 和 4.4%，每 GPU 内存分别减少 20% 和 21%。在 512 个 GPU 上，两种模式的吞吐量均提升约 5%，每 GPU 内存减少约

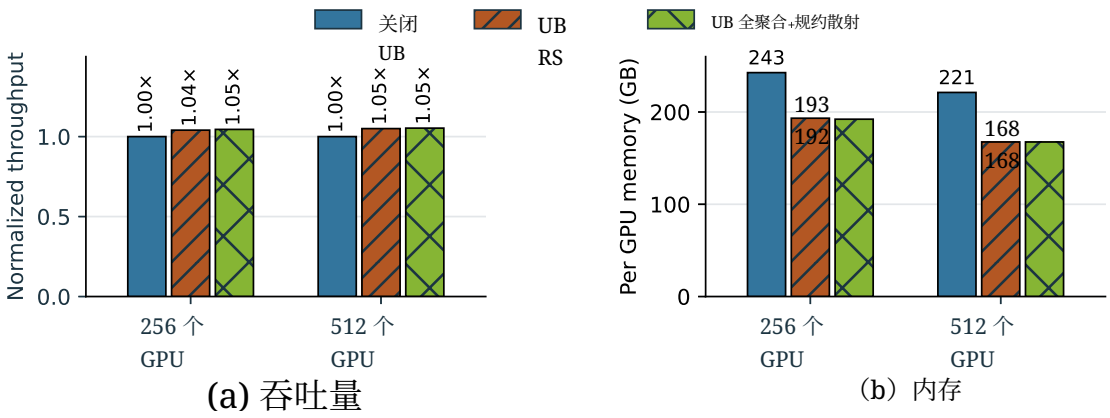


图 22 | 在 256 个和 512 个 GPU 上，Internal-2T 代理的密集用户缓冲区模式：(a) 各 GPU 数量下相对于关闭 UB 的归一化吞吐量；(b) 每 GPU 内存（GB）。

24%。在这两种规模下，UB RS 已获得大部分收益，而注册全聚合缓冲区带来的额外收益很小。内存节省来自避免全对全路径的接收缓冲区（第 4.2.3节）。

5.10. 分片粒度与填充权衡

对权重进行更细粒度的分片会缩小每个 GPU 的存储分片，但 MXFP8 和 NVFP4 要求每个分片的行数必须是其块大小的倍数，因此增加的填充可能会缩小甚至完全抵消这些节省（第 4.2.2 节）。表 7 量化了 TP1 下相对于每个通用矩阵乘法逻辑大小所增加的权重元素。这些百分比描述的是权重存储和集合通信载荷，不包括缩放因子、优化器状态和临时缓冲区。

GEMM	逻辑形状	MXFP8			NVFP4	
		GTP32	GTP64	GTP128	GTP64	GTP128
DSV3						
注意力输出投影	$7,168 \times 16,384$	0.0%	14.3%	14.3%	0.0%	14.3%
共享专家 FC2	$7,168 \times 2,048$	0.0%	14.3%	14.3%	0.0%	14.3%
Q下投影（如果分片）	$1,536 \times 7,168$	33.3%	33.3%	166.7%	33.3%	33.3%
DSV4-Pro						
注意力输出投影	$7,168 \times 16,384$	0.0%	14.3%	14.3%	0.0%	14.3%
共享专家 FC2	$7,168 \times 3,072$	0.0%	14.3%	14.3%	0.0%	14.3%
KV 投影（如果是分片的）	$512 \times 7,168$	100.0%	300.0%	700.0%	100.0%	300.0%

表 7 | TP1 下 DSV3 和 DSV4-Pro 权重的计算填充开销。共享专家使用稠密 GTP。我们的 GTP 基准测试将 DSV3 的 MLA 下投影和 DSV4-Pro 的 KV 权重保持为副本；标记为“若分片”的行显示反事实成本。我们不运行 GTP128 或 NVFP4。

选择性分片可避免不成比例的填充。 在 MXFP8 和 GTP64 下，两个模型中的注意力输出和共享专家 FC2 矩阵乘法都会产生 **14.3%** 的填充。对 DSV3 的 Q下投影和 DSV4-Pro 的 KV 投影进行分片，分别会产生 **33.3%** 和**300%** 的填充。是否对权重进行分片应取决于其形状。

填充可能会抵消单个权重进一步节省的内存。 在 MXFP8 下，将任一模型的注意力输出投影从 GTP32 改为 GTP64，可使每个 GPU 存储的元素减少 43%，同时使填充后的有效载荷增加 14.3%。如果对 DSV3 的 Q下投影进行分片，GTP64 和 GTP128 都会在每个 GPU 上存储 32 个对齐行：填充将从 33.3% 增至 166.7%，使集合通信有效载荷翻倍，却不会进一步节省存储空间。在 GTP64 下，NVFP4 可消除注意力输出投影的填充，但如果对 Q下投影进行分片，仍会产生 33.3% 的填充。

填充会增加存储和通信开销。GTP 会在通用矩阵乘法前移除填充行，因此通用矩阵乘法的形状和浮点运算次数保持不变。填充会扩大存储分片和集合通信载荷；其对吞吐量的影响，除了取决于未隐藏的传输开销外，还可能受到内存压力和通信争用的影响。

较小的专家分片度可以避免行填充。在 TP1 下，DSV3 和 DSV4-Pro 中路由FC1和FC2权重在 **EGTP2、EGTP4 和 EGTP8 下无需进行 MXFP8 行填充**：它们的输出维度可被 $32 \times 8 = 256$ 行整除。分别配置稠密 GTP 和 EGTP，可以在不让专家承担同样对齐开销的情况下，实现较宽的稠密分片。

6. GTP配置经验

GTP 的通用性带来了许多配置选择，而最佳选择取决于模型、批量大小等训练超参数，以及集群。本节将简要讨论如何做出这些选择。

6.1. 隐藏参数全收集

参数分片以全聚合通信流量为代价，减少了存储权重所需的内存。将全聚合与计算重叠，可以部分隐藏这项开销。释放出的内存还可用于增大微批次，从而增加可用于隐藏全聚合的计算量。

每个专家的词元数决定了重叠预算。公式4说明了为什么配置 GTP 时每个专家的词元数很重要：词元越多，可用于覆盖每次权重全聚合的计算就越多。在均衡路由下，每个微批次中每个专家的词元数从 DSV4-Pro 的 8,192 到 DSV3 的 16,384 不等；第5.4.1 节将这些差异与观察到的调度收益联系起来。更大的微批次、更长的本地序列以及更高的专家并行度会提高 s_e；梯度累积则不会，因为它会同时重复计算和全聚合。提高 EGTP 并行度 r_e 只会让全收集部分 $1 - 1/r_e$ 缓慢趋近于 1；提高 r_e 的主要代价是，跨 NVLink 域的全聚合会导致 B_r^e大幅降低。对于稠密权重，我们采用域内放置，使全聚合通过 NVLink 进行（第4.1节），相较于跨域放置减少了全聚合时间。

调度会限制重叠。 公式4 中的高比值代表机会，而非保证：调度必须在每个权重的消费者运行之前交付该权重。GFC1 和 GFC2 需要在不同时间获取各自的权重；采用消费者专属等待（C2）时，只有当某个 GEMM 的其他输入已就绪而其自身权重尚未就绪时，该 GEMM 才会停顿（第 3.3节）。在 DSV4-Pro 上，在输入词元数和全聚合数据量不变的情况下，仅禁用消费者专属等待（C2）就会使吞吐量降至完整 GTP 的 84.3%（第 5.7节）。

内存限制了 s 的增长空间。 增加 s 会提高激活内存。增加专家并行可以在不增加每个 GPU 的输入词元数的情况下提高每个专家的词元数，但会改变分发缓冲区和本地专家数量；将专家并行限制在一个 NVLink 域内，还会将其并行度限制为该域的大小。预取权重也会占用内存，从而限制前瞻范围（第3.1节）。重计算或卸载可以释放内存，但会增加计算或传输成本。

6.2. 选择并行度与放置

配置 GTP 意味着选择 TP、重新物化度（r 用于稠密权重，r_e 用于专家权重）和 CP度数，并确定每组 GPU 相对于 NVLink 域的放置位置。每种选择都需要在通信与内存之间进行权衡；平衡点取决于每 GPU 令牌数和层形状。

TP 度数。 TP 的激活通信量随 t增长，而参数通信量则会缩小；在纯带宽模型假设下，公式 3 给出了使组合通信时间最小的 TP 度数。每 GPU 令牌数较多时适合较小的 TP 度数，而令牌数较少、层较宽或 TP 链路较快时适合较大的度数；对于混合专家模型，h' 表示每个专家的中间维度，而不是专家数量。

重新物化度。 较大的重新物化度会将每个 GPU 存储的权重和梯度缩减至每个稠密权重的 $1/(t\cdot\ r)$ 以及每个专家权重的 $1/r_e$，但每次全聚合随后都会跨越更多 GPU。共同持有每个稠密权重一个副本的 $t\cdot r$ 个 GPU 应保持在同一个 NVLink 域内，这样每微批次权重全聚集就能通过高速链路完成。专家权重则可以使用更大的 r_e，跨越多个域（第 4.1.3 节）：它们的全聚合会在独立的链（C1）上预取，并且不会随词元数增长，前提是公式 4 中的比率在较低的跨域带宽下仍保持较高。对于较小的权重，当分片节省的内存很少或会引入不成比例的填充时，可以继续使用副本（第 4.1.1 节和 5.10 节）。

CP度数。 在序列长度和 MBS 固定时，提高 CP度数会减少每个 GPU 上的词元数及其激活值存储。在 GPU 数量以及 TP 和 PP 度数也固定的情况下，可独立划分的批次数量会减少。若保持 GBS 不变，则需要更多梯度累积微批次。这些额外的微批次会重复进行权重全聚合，而每次 GEMM 中词元数减少也可能降低计算效率。此外，注意力后端还必须在更大的 CP组上进行通信。对于长序列，按照序列长度的比例提高 CP度数，可以保持每个 GPU 上的词元数和微批次数量不变，正如我们的长上下文实验所示（第 5.6节）。

EP 和 CP 的放置。 EP 应保持在一个 NVLink 域内（例如，在一个包含 72 个 GPU 的域中使用 EP64 [46]），因为分发与合并会为依赖计算提供输入，并且会随着本地词元数的增加而增长。在激活内存允许的情况下，CP 也应保持在一个 NVLink 域内。当 CP 必须跨越多个域时，稠密权重的每个权重组仍应保持在一个 NVLink 域内（第 4.1.2节），而 CP 随后会与 EGTP 共享域间网络（第5.6节）。

6.3. 与流水线并行的权衡

内存。 PP 和 GTP 都会按照各自并行度的比例，减少每个 GPU 上的参数和优化器状态占用。在混合专家模型中，大多数参数都是专家权重，因此相关比较应关注 PP 并行度与 EGTP 并行度。GTP 还会按照其跨越的 GPU 数量的比例减少激活内存，但会为重新物化的权重及其

梯度增加固定的缓冲区开销。PP 会造成内存不对称：尽管每个阶段只为自身的层存储激活值，但第一阶段需要为数量等于阶段数的在途微批次保留这些激活值，因此其激活内存并不会缩减。

吞吐量。 流水线并行的主要吞吐量开销是流水线填充与排空，这一开销会分摊到训练步中的微批次上，因此会随着微批次数量增加而缩小。虚拟流水线通过为每个 GPU 分配多个更小的模型分块来缩短填充与排空时间，但代价是需要更多的点对点通信 [35]。流水线并行还要求各阶段的计算保持均衡；对于层类型成本不同的混合模型，这一点更难实现，而采用虚拟流水线时则更加困难，因为每个分块包含的层更少，可用于平衡的空间也更小。广义张量并行的主要吞吐量开销是其权重全聚合，这一操作成本较高，但当每个微批次包含足够的计算量且通信经过仔细调度时，可以将其隐藏起来（公式 4）。在我们的测量中，随着填充与排空开销得到分摊，更大的全局批次缩小了广义张量并行相对于三维并行的优势；但阶段不平衡使三维并行在所有测得的批次大小下都更慢（第 5.5节）。

在 GTP 和 PP 之间进行选择。 当激活内存占主导，或每个微批次包含足够的计算量以隐藏其全聚合时，单独使用 GTP 更为合适。当全局批次足够大，能够摊销填充和排空开销且各阶段可以实现均衡时，或者当无法隐藏 GTP 的全聚合时，加入 PP 会更有帮助。

7. 相关工作

统一分片抽象。 GSPMD [70]和 Mesh-TensorFlow [59]将数据并行、张量并行及其他形式的并行表示为设备网格上张量维度的分片注释，而 GSPMD 的编译器会将这些注释传播到每个算子。Alpa [76] 会与流水线阶段联合搜索此类算子内计划。GTP 将 DP、TP、SP、CP 和 FSDP 视为分片与重新物化选择，这与上述思路一致。我们关注这一视角对通信调度的影响：将参数重新物化视为 GEMM 一个输入上的预处理步骤，包括融合算子内部的 GEMM，并在即时执行框架中使其能够高效扩展至大规模。

FSDP 实现。 ZeRO-3 [57] 和 PyTorch FSDP [75] 会对参数、梯度和优化器状态进行分片，并按模块重新物化参数。MFSDP [43], PyTorch FSDP2 [53],veScale-FSDP [67], 和 FlexShard [10] 提供了用于预取、分桶和参数分组的控制项，而 DeepSpeed ZeRO-3 会按照跟踪的训练步骤中记录的模块顺序预取参数 [8]。这些系统源于数据并行，并共享基于单元的调度模型。它们在单元的构成方式，以及所提供的预取、分桶和保留控制方面存在差异。第 3.1 节讨论了该模型对融合专家混合层产生的开销。更细粒度的单元（例如，为 FC1 和 FC2 分别设置 MFSDP 单元）可以应对其中一些开销，但必须针对每个模型进行配置。我们在第 5.4节将 GTP 与 MFSDP 和 PyTorch FSDP2 进行评估比较。

veScale-FSDP 和 FlexShard 是另外两种近期的 FSDP 实现，重点关注参数如何在 GPU 之间分布。veScale-FSDP 使用 RaggedShard 布局规划和 DBuffer 视图来对通信进行分组、对齐存储并减少复制 [67]。FlexShard 将存储与优化器所有权对齐，以避免 Muon 更新时重新分布，同时支持块对齐的 FP8 布局和可配置的通信组 [10]。其所有者分配依据参数数量平衡整个矩阵，而 GTP 的按层分片 NS 则依据 NS 成本进行平衡。

MiCS [74], ZeRO++ [64]和 HSDP [75] 将参数全收集限制在高速链路上的子组内。GTP 的权重组同样将权重全收集保留在高速链路上。GTP 还独立于权重组对数据进行分区，因此权重全收集无需在完整的 CP 组上执行，并且一个长序列可以跨越多个权重组（第 4.1节）。

三维并行。 三维并行结合了张量并行、流水线并行和数据并行，已被广泛用于训练大型模型 [15, 34, 35], 但其流水线并行部分有三个弱点。除非每个训练步包含许多微批次，否则流水线填充与排空会让 GPU 闲置。各阶段必须保持均衡，否则较快的阶段就要等待较慢的阶段。每个阶段还需为多个在途微批次保留激活值。以往的研究致力于降低这些开销：PipeDream 使用基于性能分析的分区来平衡流水线阶段 [34,第 3 节。1], 而 Alpa 则联合搜索异构层的阶段分区和设备分配 [76, 第 5 节和第 8 节。1]。Megatron-LM 的虚拟流水线并行通过交错排列模型分块来减少理想情况下的流水线气泡，但会增加阶段间通信 [35,第 2.2 节。2]。GTP 可与流水线并行结合；不采用流水线并行的配置可避免阶段不平衡和流水线气泡，但需要足够的内存来存放重新物化的权重，并需要足够的计算来掩盖重新物化的通信开销（第 6.3节）。

重叠张量并行通信。 张量并行中通信的激活值是前序计算的输出，因此，要重叠这类通信，就需要将通用矩阵乘法和通信拆分成更小的部分（第 2.1节）。Wang等人将集合通信与其依赖计算一同分解 [66], T3借助硬件支持，触发生成该数据的通用矩阵乘法进行通信 [49], FLUX将分块通信融合进通用矩阵乘法内核 [4],而Domino则将输入和权重切分为彼此独立的部分 [65]。相比之下，GTP的参数全收集不依赖前序计算，因此可以在使用这些参数的通用矩阵乘法启动之前发起，无需拆分任何计算；当 $t > 1$ 时，这些技术仍可用于GTP的激活通信。

混合专家层并行化。 GShard [25], 、DeepSpeed-MoE [56], 、Tutel [16], 和MegaBlocks [11]采用专家并行、针对不断变化的专家工作负载进行运行时适配以及专用内核等技术，加速混合专家层。我们的实现基于MCore中的MoE并行折叠，该方法将注意力层与混合专家层的并行映射解耦 [28]; GTP进一步支持稠密层权重和专家权重采用不同的重新物化度（GTP和EGTP）。MegaScale-MoE针对每个模块定制并行策略，并在生产环境中重叠混合专家通信 [21], ；COMET则以细粒度重叠专家并行通信与专家计算 [73]。这些系统提高了令牌交换和专家计算的效率；GTP可与专家并行及这些优化结合使用。

8. 结论

本文展示了，对于线性层，五种常用的并行方式（DP、TP、SP、CP 和 FSDP）都是我们称为广义张量并行（GTP）的新框架的特例。在该框架中，参数和激活值均可进行分区，并且二者都可以重新物化。将参数重新物化视为对 GEMM 的一个输入执行的预处理步骤，而不是整个 Transformer 块之前的步骤，可以让我们按每个 GEMM 调度全聚合。借助我们在 MCore 和 TE 中的实现，一个内部 2T 参数的混合专家模型在 12,288 个 GB300 GPU 上达到了每秒 13.0 百亿亿次浮点运算的模型性能。在 256–1,024 个 GPU 上，我们的实现在三个混合专家模型上实现了 1.05–1.55× M FSDP 吞吐量

，以及在 DSV4-Pro 代理平台上调优后的三维并行方案吞吐量的 1.27–1.31× 倍。

致谢

感谢 Xiaowei Ren、Yashaswi Karnati、Peter Dykas、Zhongbo Zhu、Zhengyang Wang、Hongx- iao Bai、Yuzhong Wang、Sangkug Lym 和 Michael Andersch 对本工作的贡献。我们也感谢未在此逐一列出的同事，以及开源社区对共同开发的功能所提供的反馈和持续参与。

参考文献

[1] Noah Amsel、David Persson、Christopher Musco 和 Robert M. Gower。《The Polar Express：最优矩阵符号方法及其在 Muon 算法中的应用》。*arXiv* 预印本 *arXiv:2505.16932*，2025年。

[2] Jimmy Lei Ba、Jamie Ryan Kiros 和 Geoffrey E. Hinton。《层归一化》。*arXiv* 预印本 *arXiv:1607.06450*，2016年。

[3] Tom B.Brown、Benjamin Mann、Nick Ryder、Melanie Subbiah、Jared Kaplan、Prafulla Dhariwal、Arvind Neelakantan、Pranav Shyam、Girish Sastry、Amanda Askell、Sandhini Agarwal、Ariel Herbert-Voss、Gretchen Krueger、Tom Henighan、Rewon Child、Aditya Ramesh、Daniel M. Ziegler、Jeffrey Wu、Clemens Winter、Christopher Hesse、Mark Chen、Eric Sigler、Mateusz Litwin、Scott Gray、Benjamin Chess、Jack Clark、Christopher Berner、Sam McCandlish、Alec Radford、Ilya Sutskever 和 Dario Amodei。《语言模型是少样本学习器》。载于神经信息处理系统进展（*NeurIPS*），2020年。

[4] Li-Wen Chang、Wenlei Bao、Qi Hou、Chengquan Jiang、Ningxin Zheng、Yinmin Zhong、Xuanrun Zhang、Zuquan Song、Chengji Yao、Ziheng Jiang、Haibin Lin、Xin Jin 和 Xin Liu。《FLUX：通过内核融合在 GPU 上实现快速的基于软件的通信重叠》。*arXiv* 预印本 *arXiv:2406.06858*，2024年。

[5] Aakanksha Chowdhery、Sharan Narang、Jacob Devlin、Maarten Bosma、Gaurav Mishra、Adam Roberts、Paul Barham、Hyung Won Chung、Charles Sutton、Sebastian Gehrmann、Parker Schuh、Kensen Shi、Sasha Tsvyashchenko、Joshua Maynez、Abhishek Rao、Parker Barnes、Yi Tay、Noam Shazeer、Vinodkumar Prabhakaran、Emily Reif、Nan Du、Ben Hutchinson、Reiner Pope、James Bradbury、Jacob Austin、Michael Isard、Guy Gur-Ari、Pengcheng Yin、Toju Duke、Anselm Levskaya、Sanjay Ghemawat、Sunipa Dev、Henryk Michalewski、Xavier Garcia、Vedant Misra、Kevin Robinson、Liam Fedus、Denny Zhou、Daphne Ippolito、David Luan、Hyeontaek Lim、Barret Zoph、Alexander Spiridonov、Ryan Sepassi、David Dohan、Shivani Agrawal、Mark Omernick、Andrew M. Dai、Thanumalayan Sankaranarayana Pillai、Marie Pellat、Aitor Lewkowycz、Erica Moreira、Rewon Child、Oleksandr Polozov、Katherine Lee、Zongwei Zhou、Xuezhi Wang、Brennan Saeta、Mark Diaz、Orhan Firat、Michele Catasta、Jason Wei、Kathy Meier-Hellstern、Douglas Eck、Jeff Dean、Slav Petrov 和 Noah Fiedel。《PaLM：借助 Pathways 扩展语言建模》。机器学习研究杂志，24(240):1–113，2023。

[6] DeepSeek-AI。《DeepSeek-V3 技术报告》。*arXiv* 预印本 *arXiv:2412.19437*，2024年。

[7] DeepSeek-AI。《DeepSeek-V4：迈向高效百万词元上下文智能》。*arXiv* 预印本 *arXiv:2606.19348*，2026年。

[8]DeepSpeed 贡献者。《DeepSpeed ZeRO-3 参数协调器与配置》。 https://github.com/deepspeedai/DeepSpeed/tree/ff34be65ee1656ddca9a4b31fda3e817bbb34313/deepspeed/runtime/zero，2026年。修订版本 ff34be65ee16。[9] William Fedus、Barret Zoph 和 Noam Shazeer。《Switch Transformers：通过简单高效的稀疏性扩展至万亿参数模型》。机器学习研究杂志，23(120)：1–39，2022。[10]FlexShard 贡献者。《FlexShard：可插拔的参数分片与优化器布局》。 https://github.com/meta-pytorch/flex_shard/tree/233e90691a55，2026年。修订版本 233e90691a55。[11] Trevor Gale、Deepak Narayanan、Cliff Young 和 Matei Zaharia。《MegaBlocks：使用混合专家进行高效稀疏训练》。载于《机器学习与系统会议论文集》，第 5 卷，2023。[12] Ronald L. Graham。《多处理时序异常的界限》。SIAM 应用数学杂志，17(2)：416–429，1969。[13] Khaled Hamidouche、John Bachan、Pak Markthub、Peter-Jan Gootzen、Elena Agostini、Sylvain Jeaugey、Aamir Shafi、Georgios Theodorakis 和 Manjunath Gorentla Venkata。《面向 NCCL 的 GPU 发起的网络通信》。arXiv 预印本 arXiv:2511.15076，2025年。[14] Jordan Hoffmann、Sebastian Borgeaud、Arthur Mensch、Elena Buchatskaya、Trevor Cai、Eliza Rutherford、Diego de las Casas、Lisa Anne Hendricks、Johannes Welbl、Aidan Clark、Thomas Hennigan、Eric Noland、Katherine Millican、George van den Driessche、Bogdan Damoc、Aurelia Guy、Simon Osindero、Karén Simonyan、Erich Elsen、Oriol Vinyals、Jack Rae 和 Laurent Sifre。《计算最优大语言模型训练的实证分析》。载于《神经信息处理系统进展》，第 35 卷，第 30016–30030 页。Curran Associates, Inc.，2022。[15] Yanping Huang、Youlong Cheng、Ankur Bapna、Orhan Firat、Mia Xu Chen、Dehao Chen、HyoukJoong Lee、Jiquan Ngiam、Quoc V. Le、Yonghui Wu 和 Zhifeng Chen。《GPipe：使用流水线并行高效训练巨型神经网络》。载于《神经信息处理系统进展》（NeurIPS），2019年。[16] Changho Hwang、Wei Cui、Yifan Xiong、Ziyue Yang、Ze Liu、Han Hu、Zilong Wang、Rafael Salas、Jithin Jose、Prabhat Ram、Joe Chau、Peng Cheng、Fan Yang、Mao Yang 和 Yongqiang Xiong。《Tutel：大规模自适应混合专家》。载于《机器学习与系统会议论文集》，第 5 卷，2023。[17] HybridEP 贡献者。《HybridEP：混合专家并行通信后端》。 https://github.com/deepseek-ai/DeepEP/tree/hybrid-ep，2026年。DeepEP 的 HybridEP 分支；访问于 2026 年 10 月。[18]InfiniBand 贸易协会。《关于 InfiniBand》。 https://www.infinibandta.org/about-infiniband/，2026年。访问于 2026 年 10 月。[19] InfiniBand 贸易协会。《RoCE 倡议》。https://www.infinibandta.org/roce-initiative/，2026年。访问于 2026 年 10 月。[20] Sam Ade Jacobs、Masahiro Tanaka、Chengming Zhang、Minjia Zhang、Shuaiwen Leon Song、Samyam Rajbhandari 和 Yuxiong He。《DeepSpeed Ulysses：通过系统优化实现超长序列 Transformer 模型训练》。arXiv 预印本 arXiv:2309.14509，2023。

[21] Chao Jin、Ziheng Jiang、Zhihao Bai、Zheng Zhong、Juncai Liu、Xiang Li、Ningxin Zheng、Xi Wang、Cong Xie、Qi Huang、Wen Heng、Yiyuan Ma、Wenlei Bao、Size Zheng、Yanghua Peng、Haibin Lin、Xuanzhe Liu、Xin Jin 和 Xin Liu。《MegaScale-MoE：生产环境中混合专家模型的大规模高效通信训练》。arXiv 预印本 arXiv:2505.11432，2025年。[22] Keller Jordan、Yuchen Jin、Vlado Boza、Jiacheng You、Franz Cesista、Laker Newhouse 和 Jeremy Bernstein。《Muon：神经网络隐藏层的优化器》。https://kellerjordan.github.io/posts/muon/，2024年。[23] Mikail Khona、Aditya Vavre、Boxiang Wang、Deyu Fu、Hao Wu、Mike Chrzanowski、Bryan Catanzaro、Dheevatsa Mudigere、Jeff Pool、Michael Lightstone、Mohammad Shoeybi、Mostofa Patwary、Nima Tajbakhsh 和 Tijmen Blankevoort。《SOAP、Muon 及更进一步：推动大语言模型预训练规模扩展》。arXiv 预印本 arXiv:2607.20548，2026年。[24] Vijay Anand Korthikanti、Jared Casper、Sangkug Lym、Lawrence McAfee、Michael Andersch、Mohammad Shoeybi 和 Bryan Catanzaro。《减少大型Transformer模型中的激活重计算》。载于《机器学习与系统会议论文集》，第5卷，第341–353页，2023年。[25] Dmitry Lepikhin、HyoukJoong Lee、Yuanzhong Xu、Dehao Chen、Orhan Firat、Yanping Huang、Maxim Krikun、Noam Shazeer 和 Zhifeng Chen。《GShard：通过条件计算和自动分片扩展巨型模型》。载于国际学习表征会议（ICLR），2021年。[26] Shen Li、Yanli Zhao、Rohan Varma、Omkar Salpekar、Pieter Noordhuis、Teng Li、Adam Paszke、Jeff Smith、Brian Vaughan、Pritam Damania 和 Soumith Chintala。《PyTorch分布式：加速数据并行训练的实践经验》。载于《VLDB基金会学报》，13(12):3005–3018，2020年。[27] Shenggui Li、Fuzhao Xue、Chaitanya Baranwal、Yongbin Li 和 Yang You。《序列并行：从系统视角看长序列训练》。载于第61届计算语言学协会年会（ACL）论文集，第2391–2404页，2023年。[28] Dennis Liu、Zijie Yan、Xin Yao、Tong Liu、Vijay Korthikanti、Evan Wu、Shiqing Fan、Gao Deng、Hongxiao Bai、Jianbin Chang、Ashwath Aithal、Michael Andersch、Mohammad Shoeybi、Jiajie Yao、Chandler Zhou、David Wu、Xipeng Li 和 June Yang。《MoE并行折叠：使用Megatron Core实现高效大规模混合专家模型训练的异构并行映射》。arXiv 预印本 arXiv:2504.14960，2025年。[29] Hao Liu、Matei Zaharia 和 Pieter Abbeel。《利用分块Transformer实现近无限上下文的环形注意力》。载于国际学习表征会议（ICLR），2024年。[30] Llama 团队，Meta AI。《Llama 3 模型家族》。arXiv 预印本 arXiv:2407.21783，2024年。[31] Ilya Loshchilov 和 Frank Hutter。《解耦权重衰减正则化》。载于国际学习表征会议（ICLR），2019年。[32] Meta AI。《Llama 4 模型家族：原生多模态人工智能创新新时代的开端》。https://ai.meta.com/blog/llama-4-multimodal-intelligence/，2025年。[33] Meta PyTorch 贡献者。《Dist-MoE：分布式混合专家内核》。https://github.com/meta-pytorch/dist_moe，2026年。访问于2026年10月。

[34] Deepak Narayanan、Aaron Harlap、Amar Phanishayee、Vivek Seshadri、Nikhil R. Devanur、Gregory R. Ganger、Phillip B. Gibbons 和 Matei Zaharia。《PipeDream：用于深度神经网络训练的广义流水线并行》。载于第27届ACM操作系统原理研讨会（SOSP）论文集，2019年。[35] Deepak Narayanan、Mohammad Shoeybi、Jared Casper、Patrick LeGresley、Mostofa Patwary、Vijay Korthikanti、Dmitri Vainbrand、Prethvi Kashinkunti、Julie Bernauer、Bryan Catanzaro、Amar Phanishayee 和 Matei Zaharia。《使用Megatron-LM在GPU集群上高效训练大规模语言模型》。载于国际高性能计算、网络、存储与分析会议（SC）论文集，2021年。[36] NVIDIA。《Nemotron 3 Nano：面向智能体推理的开放、高效混合专家混合式Mamba-Transformer模型》。https://research.nvidia.com/labs/nemotron/files/NVIDIA-Nemotron-3-Nano-Technical-Report.pdf，2025年。技术报告。[37] NVIDIA。《使用NVFP4预训练大语言模型》。arXiv 预印本 arXiv:2509.25149，2025年。[38] NVIDIA。《CUDA编程指南：CUDA 图》。 https://docs.nvidia.com/cuda/cuda-programming-guide/04-special-topics/cuda-graphs.html，2026年。访问于2026年11月12日。[39] NVIDIA。《Emerging Optimizers》。 https://github.com/NVIDIA-NeMo/Emerging-Optimizers，2026年。版本0.3.1。[40] NVIDIA。《Megatron Bridge：DeepSeek-V4-Pro GB300性能配方》。 https://github.com/NVIDIA-NeMo/Megatron-Bridge/blob/c36bd9a02e54538a0aa9b8be7ac2145a5d1f2ca4/src/megatron/bridge/perf_recipes/deepseek/gb300/deepseek_v4.py，2026年。修订版c36bd9a02e54。[41]NVIDIA。《Megatron Bridge：性能摘要》。https://docs.nvidia.com/nemo/megatron-bridge/latest/performance-summary.html#performance-metrics，2026年。访问于2026年10月。[42] NVIDIA。《Megatron-Core广义张量并行实现》。 https://github.com/NVIDIA/Megatron-LM，2026年。访问于2026年11月17日。[43]NVIDIA。《Megatron-Core MFSDP：实现及评估基线的上游基础版本》。https://github.com/NVIDIA/Megatron-LM/commit/725949abbfa0878b2d84997a07cc7d959ee55624，2026年。MCore 0.20.0开发基线，基于main分支；提交725949abbfa0878b2d84997a07cc7d959ee55624。评估所用代码检出版本包含特定模型的扩展和修复。[44]NVIDIA。《Megatron-LM：DeepSeek MLA模型浮点运算量估算器》。https://github.com/NVIDIA/Megatron-LM/blob/92641a421f40eeb40fc83f3f13ec83ca652c8b13/megatron/training/training.py#L1160-L1171，2026年。修订版 92641a421f40。[45]NVIDIA。《NCCL用户指南：用户缓冲区注册》。https://docs.nvidia.com/deeplearning/nccl/user-guide/docs/usage/bufferreg.html，2026年。访问于2026年10月。[46]NVIDIA。《NVIDIA GB300 NVL72》。https://www.nvidia.com/en-us/data-center/gb300-nvl72/，2026年。访问于2026年10月。

[47] 英伟达。Transformer Engine。 https://github.com/NVIDIA/TransformerEngine, 2026年。访问于2026年10月。[48] NVIDIA TransformerEngine 贡献者。Transformer Engine FSDP2 量化权重全聚合的实现与测试。 https://github.com/NVIDIA/TransformerEngine/tree/2e959f352c8e2d4b2709e6775867f2c7494656f3，2026年。修订版本 2e959f352c8e。[49] Suchita Pati、Shaizeen Aga、Mahzabeen Islam、Nuwan Jayasena 和 Matthew D. Sinclair。《T3：面向计算与集合通信细粒度重叠的透明跟踪与触发》。载于第29届 ACM 编程语言与操作系统架构支持国际会议（ASPLOS）论文集，2024年。[50] PyTorch。分布式检查点 – torch.distributed.checkpoint。 https://docs.pytorch.org/docs/stable/distributed.checkpoint.html，2026年。访问于2026年11月17日。[51] PyTorch 贡献者。FSDP2 预取与集合通信完成实现。 https://github.com/pytorch/pytorch/tree/ad8c2fc386e4f19c8335e097980671a4dbb15c26/torch/distributed/fsdp/_fully_shard，2026年。修订版本 ad8c2fc386e4（PyTorch 2.16.0.dev20261022）。[52]PyTorch 贡献者。PyTorch FSDP2 参数分片实现。 https://github.com/pytorch/pytorch/blob/be90fe2665c2/torch/distributed/fsdp/_fully_shard/_fsdp_param.py，2026年。 `_fsdp_param.py`，修订版本 be90fe2665c2；于2026年10月12日检查了不均匀分片填充实现。[53] PyTorch 贡献者。torch.distributed.fsdp.fully_shard。 https://docs.pytorch.org/docs/2.14/distributed.fsdp.fully_shard.html，2026年。PyTorch 2.14 API 文档。[54] PyTorch 贡献者。TorchTitan FSDP 分组与显式预取配置。 https://github.com/pytorch/torchtitan/blob/6cdda6cacb4ce25d609aa6f38947dfc69c674bcc/torchtitan/distributed/fsdp.py，2026年。TorchTitan 0.2.2；修订版本 6cdda6cacb4c。[55] PyTorch 贡献者。TorchTitan：每个词元的二次注意力浮点运算次数。https://github.com/pytorch/torchtitan/blob/6cdda6cacb4ce25d609aa6f38947dfc69c674bcc/torchtitan/models/common/flops.py#L20-L51，2026年。修订版本 6cdda6cacb4c。[56] Samyam Rajbhandari、Conglong Li、Zhewei Yao、Minjia Zhang、Reza Yazdani Aminabadi、Ammar Ahmad Awan、Jeff Rasley 和 Yuxiong He。《DeepSpeed-MoE：推进混合专家推理与训练，赋能下一代人工智能规模》。载于第39届国际机器学习会议（ICML）论文集，第162卷，PMLR，第18332–18346页，2022。[57] Samyam Rajbhandari、Jeff Rasley、Olatunji Ruwase 和 Yuxiong He。《ZeRO：面向万亿参数模型训练的内存优化》。载于国际高性能计算、网络、存储与分析会议（SC）论文集，第1–16页，2020年。[58] Bita Darvish Rouhani、Ritchie Zhao、Ankit More、Mathew Hall、Alireza Khodamoradi、Summer Deng、Dhruv Choudhary、Marius Cornea、Eric Dellinger、Kristof Denolf、Dusan Stosic、Venmugil Elango、Maximilian Golub、Alexander Heinecke、Phil James-Roxby、Dharmesh Jani、Gaurav Kolhe、Martin Langhammer、Ada Li、Levi Melnick、Maral Mesmakhosroshahi、Andres Rodriguez、Michael Schulte、Rasoul Shafipour、Lei Shao、Michael Siu、Pradeep Dubey、Paulius Micikevicius、Maxim Naumov、Colin Verrilli、Ralph Wittig、Doug Burger 和 Eric Chung。《面向深度学习的微缩放数据格式》。arXiv 预印本 arXiv:2310.10537，2023年。

[59] Noam Shazeer、Youlong Cheng、Niki Parmar、Dustin Tran、Ashish Vaswani、Penporn Koanantakool、Peter Hawkins、HyoukJoong Lee、Mingsheng Hong、Cliff Young、Ryan Sepassi 和 Blake Hechtman。《Mesh-TensorFlow：面向超级计算机的深度学习》。载于神经信息处理系统进展（NeurIPS），2018年。[60] Noam Shazeer、Azalia Mirhoseini、Krzysztof Maziarz、Andy Davis、Quoc Le、Geoffrey Hinton 和 Jeff Dean。《惊人规模的神经网络：稀疏门控专家混合层》。载于国际学习表征会议（ICLR），2017年。[61] Mohammad Shoeybi、Mostofa Patwary、Raul Puri、Patrick LeGresley、Jared Casper 和 Bryan Catanzaro。《Megatron-LM：使用模型并行训练数十亿参数语言模型》。arXiv 预印本 arXiv:1909.08053，2019年。[62]TorchTitan 贡献者。TorchTitan 分布式 Muon 实现。https://github.com/pytorch/torchtitan/blob/cc0ad1f/torchtitan/distributed/flex_shard/dist_muon.py，2026年。dist_muon.py，修订版本 cc0ad1f149cc，于2026年10月12日检查。[63] Ashish Vaswani、Noam Shazeer、Niki Parmar、Jakob Uszkoreit、Llion Jones、Aidan N. Gomez、Łukasz Kaiser 和 Illia Polosukhin。《Attention Is All You Need》。载于神经信息处理系统进展（NeurIPS），2017年。[64] Guanhua Wang、Heyang Qin、Sam Ade Jacobs、Xiaoxia Wu、Connor Holmes、Zhewei Yao、Samyam Rajbhandari、Olatunji Ruwase、Feng Yan、Lei Yang 和 Yuxiong He。《ZeRO++：面向大模型训练的极高效集合通信》。载于国际学习表征会议（ICLR），2024年。[65] Guanhua Wang、Chengming Zhang、Zheyu Shen、Ang Li 和 Olatunji Ruwase。《Domino：通过通用张量切片与重叠消除大语言模型训练中的通信》。arXiv 预印本 arXiv:2409.15241，2024年。[66] Shibo Wang、Jinliang Wei、Amit Sabne、Andy Davis、Berkin Ilbeyi、Blake Hechtman、Dehao Chen、Karthik Srinivasa Murthy、Marcello Maggioni、Qiao Zhang、Sameer Kumar、Tongfei Guo、Yuanzhong Xu 和 Zongwei Zhou。《通过分解在大型深度学习模型中重叠通信与依赖计算》。载于第28届 ACM 编程语言与操作系统架构支持国际会议（ASPLOS）论文集，第93–106页，2023年。[67] Zezhou Wang、Youjie Li、Zhiqi Lin、Jiacheng Yang、Cong Xie、Guanyu Feng、Zheng Zhong、Ziyue Huang、Hongyu Zhu、Zhi Zhang、Yanghua Peng 和 Xin Liu。《veScale-FSDP：可扩展的灵活高性能 FSDP》。载于《机器学习与系统会议论文集》，第8卷，第763–776页。MLSys，2026年。[68] Xilun Wu、Chien-Chin Huang、Less Wright、Tianyu Liu、Will Constable 和 Gokul Nadathur。《突破障碍：在 PyTorch 中使用上下文并行训练序列长度达 100 万的长上下文大语言模型》。 https://discuss.pytorch.org/t/215082，2025年1月。PyTorch 论坛。[69] Zhenda Xie、Yixuan Wei、Huanqi Cao、Chenggang Zhao、Chengqi Deng、Jiashi Li、Damai Dai、Huazuo Gao、Jiang Chang、Kuai Yu、Liang Zhao、Shangyan Zhou、Zhean Xu、Zhengyan Zhang、Wangding Zeng、Shengding Hu、Yuqing Wang、Jingyang Yuan、Lean Wang 和 Wenfeng Liang。《mHC：流形约束超连接》。arXiv 预印本 arXiv:2512.24880，2025年。[70] Yuanzhong Xu、HyoukJoong Lee、Dehao Chen、Blake Hechtman、Yanping Huang、Rahul Joshi、Maxim Krikun、Dmitry Lepikhin、Andy Ly、Marcello Maggioni、Ruoming Pang、Noam Shazeer，

Shibo Wang、Tao Wang、Yonghui Wu 和 Zhifeng Chen。《GSPMD：面向机器学习计算图的通用且可扩展并行化》。*arXiv* 预印本 *arXiv:2105.04663*，2021年。

[71] Zijie Yan、Hongxiao Bai、Xin Yao、Dennis Liu、Tong Liu、Hongbin Liu、Pingtian Li、Evan Wu、Shiqing Fan、Li Tao、Robin Zhang、Yuzhong Wang、Shifang Xu、Jack Chang、Xuwen Chen、Kunlun Li、Yan Bai、Gao Deng、Nan Zheng、Vijay Anand Korthikanti、Abhinav Khattar、Ethan He、Soham Govande、Sangkug Lym、Zhongbo Zhu、Qi Zhang、Haochen Yuan、Xiaowei Ren、Deyu Fu、Tailai Ma、Shunkang Zhang、Jiang Shao、Ray Wang、Vasudevan Rengasamy、Rachit Garg、Santosh Bhavani、Xipeng Li、Chandler Zhou、David Wu、Yingcan Wei、Ashwath Aithal、Michael Andersch、Mohammad Shoeybi、Jiajie Yao 和 June Yang。《使用 Megatron Core 实现可扩展的混合专家模型训练》。技术报告 arXiv:2603.07685，英伟达，2026年。第 2 版。

[72] Biao Zhang 和 Rico Sennrich。均方根层归一化。载于神经信息处理系统进展（*NeurIPS*），第 32 卷，2019年。

[73] Shulai Zhang、Ningxin Zheng、Haibin Lin、Ziheng Jiang、Wenlei Bao、Chengquan Jiang、Qi Hou、Weihao Cui、Size Zheng、Li-Wen Chang、Quan Chen 和 Xin Liu。COMET：面向混合专家的细粒度计算与通信重叠。载于机器学习与系统会议论文集，2025年。

[74] Zhen Zhang、Shuai Zheng、Yida Wang、Justin Chiu、George Karypis、Trishul Chilimbi、Mu Li 和 Xin Jin。MiCS：公有云上超大型模型训练的近线性扩展。*VLDB*基金会论文集，16(1)：37–50，2022。

[75] Yanli Zhao、Andrew Gu、Rohan Varma、Liang Luo、Chien-Chin Huang、Min Xu、Less Wright、Hamid Shojanazeri、Myle Ott、Sam Shleifer、Alban Desmaison、Can Balioglu、Pritam Damania、Bernard Nguyen、Geeta Chauhan、Yuchen Hao、Ajit Mathews 和 Shen Li。PyTorch FSDP：全分片数据并行扩展方面的实践经验。*VLDB*基金会论文集，16(12)：3848–3860，2023。

[76] Lianmin Zheng、Zhuohan Li、Hao Zhang、Yonghao Zhuang、Zhifeng Chen、Yanping Huang、Yida Wang、Yuanzhong Xu、Danyang Zhuo、Eric P. Xing、Joseph E. Gonzalez 和 Ion Stoica。《Alpa：自动化分布式深度学习的算子间与算子内并行》。发表于第 *16* 届 *USENIX* 操作系统设计与实现研讨会（*OSDI 22*），第 559–578 页。USENIX 协会，2022。