arXiv:2603.02885v1 [cs.DC] 3 Mar 2026

MuxTune：通过时空骨干网络复用，在多租户数据中心实现高效的多任务大语言模型微调

薛春雨[†*]、潘毅[†*]、崔伟豪[†§]、Quan Chen[†]、张书来[†]、何炳生[§]、郭敏义[†]
[†]上海交通大学 [§]新加坡国立大学

摘要

参数高效微调（PEFT）被广泛用作数据中心大语言模型（LLM）定制化微调 API 的后端。服务提供商为各个参数高效微调任务分别部署实例，由此造成显著的资源利用低效，包括：（1）小规模原生 PEFT 算子导致图形处理器利用率不足；（2）并行执行中的通信延迟和数据依赖导致设备停顿。为解决这些问题，本文提出 MuxTune，这是一种微调系统，可实现多个参数高效微调任务的高效并发执行。其核心思想是以时空方式在独立任务间复用骨干网络，从而提升利用率并减少停顿。基于统一的参数高效微调表示实现灵活的模块化骨干网络共享，MuxTune 提出了一种分层协同调度方案，并结合任务级、算子级和数据级优化。具体而言，该方案通过空间与时间复用相结合的方式融合任务，并以两级混合并行协调多任务算子执行。此外，MuxTune 采用基于分块的数据对齐，以减少任务间无效词元。实验结果表明，与三种最先进的基线方法相比，MuxTune 的吞吐量最高提升 2.33×，内存减少 5.29×。

1 引言

预训练-微调范式在大语言模型（LLM）领域迅速兴起 [32, 32, 35, 37, 72, 87]。企业和开发者利用领域特定数据集，将预训练模型微调为定制模型，从而生成更贴合上下文的回答 [7, 12, 15, 17, 75, 79]。随着新兴硬件不断发展、成本日益上升，对许多开发者而言，本地微调已变得越来越难以承受。在此背景下，大语言模型服务提供商推出公共微调 *API* [1–3, 6], 让开发者能够在共享的 GPU 数据中心中远程微调大语言模型。鉴于这些

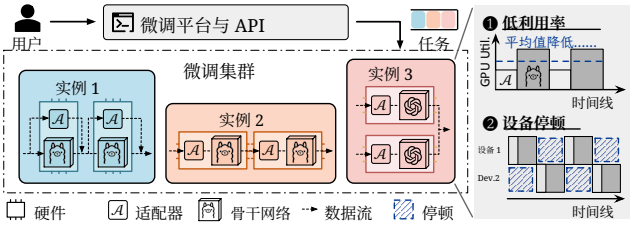


图 1：远程提交任务和微调大语言模型的工作流程。每个实例分别处理 1 个不同 PEFT 类型的任务，其骨干网络并行部署在 2 个图形处理器上。

API 采用基于令牌的定价模型 [1, 6], ，因此，使用最少的资源提升微调效率至关重要。

为降低服务供应成本并适应多样化的下游领域，参数高效微调（PEFT）被广泛用作这些 API 的后端之一 [1,2,6,43]。图 1 展示了微调服务的一个示例工作流程。PEFT 任务会将小规模可训练*适配器* 附加到预训练大语言模型 *骨干网络*的目标算子上。只有适配器会进行微调，而骨干网络参数则保持冻结 [47, 64]。如图所示，开发者使用微调 API 创建 PEFT 任务，而集群调度器分别调度任务、部署实例（硬件和骨干网络），并启动参数高效微调程序。

然而，现有的参数高效微调框架（例如 HuggingFace PEFT [18],NeMo [44]）存在明显的资源利用低效问题，主要源于两个原因。首先，参数高效微调会引入一些规模较小但不可忽视（就延迟而言）的算子，例如 LoRA 下投影 [47],, 使平均 GPU 利用率最多降低 1.47×(❶ 见图1）。与预训练相比，一些特定领域的参数高效微调语料具有更短的序列和更小的批次大小 [82, 88], , 进一步加剧了 GPU 利用不足。其次，在阶段内和 [78]阶段间 [49, 65] 并行中（❷ 见图1），PEFT 适配器会加剧由算子间通信延迟和数据依赖所导致的设备停顿。由于没有骨干权重梯度，参数高效微调不支持无停顿流水线，例如 DeepSeek-V3 DualPipe [63]和 ZeroBubble ZB-H2 [70]），后者会将反向传播拆分为输入

*贡献相同。

和权重梯度，以实现更精细的调度。通信重叠技术 [26, 86]，通过将计算算子拆分为更小的分块来与通信重叠，但容易导致 GPU 利用不足，最终增加端到端延迟（§2.2）。随着 GPU 计算能力的增长速度超过互连带宽，上述低效问题进一步加剧——NVIDIA H100 GPU 通过 NVLink 连接时与通过 PCIe 4.0 连接的 A40 GPU 相比， 3.18× 模型浮点运算利用率（MFU）存在差距 [53] 。

为了提升 GPU 利用率并缓解设备停顿，在多租户数据中心中，我们通过复用大语言模型骨干网络，将优化目标从单任务扩展到多任务协同调度。我们提出通过时空骨干网络复用来解决这些低效问题：通过空间批处理和时间交错，让多个任务共享单一骨干网络。尽管一些复用策略已应用于预训练或服务 [33, 76, 95], 但直接将其用于参数高效微调场景存在关键局限。

具体而言，改编自多 LoRA 服务的基于批处理的空间复用（例如 SLoRA [76],Punica [34]）可能会加剧设备停顿并引入无效计算，因为参数高效微调工作负载与服务存在根本差异（§2.3）。另一方面，受近期预训练和服务系统（例如 Centauri [33], NanoFlow [95]）启发的时间复用和重叠技术，虽然能缓解部分通信瓶颈，却会进一步降低 GPU 利用率。直接采用空间或时间复用方法，不仅无法有效解决这些资源利用低效问题，还可能彼此冲突。

因此，我们提出了**MuxTune**，这是一种资源高效的多任务参数高效微调系统，可在数据中心实现空间与时间复用的骨干网络。然而，在系统设计和调度方面仍存在诸多挑战。在任务级层面，如何在各任务之间进行空间复用与时间复用的混合决策，以抑制其固有的权衡，并非易事。在算子层面，在混合并行条件下，高效编排空间与时间复用中的复杂算子执行仍具有挑战性。在数据层面，如何高效对齐来自空间批处理任务的变长序列，同时减轻任务间无效
词元（即零填充）需要经过谨慎设计。

MuxTune 的核心思想是在统一的参数高效微调表示之上，通过分层设计统一空间与时间复用，从而联合优化利用率并减少设备停顿。该方法首先将多种参数高效微调类型模块化为统一表示，使骨干网络能够灵活共享，同时保持一致的训练行为（§3.2）。基于这一基础，MuxTune 采用分层多任务协同调度方案，并针对各层设计专门的解决方案。在任务级，MuxTune 引入混合任务（“hTask”）抽象，以应对时空权衡：同一 hTask 内的任务通过空间批处理提升利用率，不同 hTask 则通过时间交错来隐藏停顿。借助基于流水线的成本

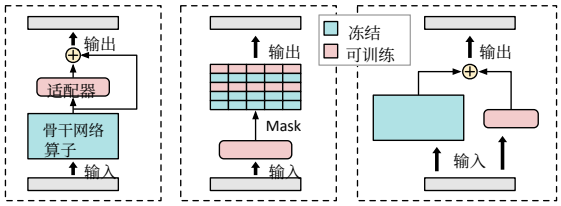


图2：PEFT算法的典型类别。

模型，使用动态规划（DP）算法确定将任务融合为多个 hTask 的最优方式（§3.3）。在算子层，MuxTune 统筹共享的模块化骨干网络上各 hTask 的算子执行。通过求解跨阶段间并行与阶段内并行的两级调度问题[1]，它能够在混合并行下协调复杂依赖关系，生成细粒度、无停顿的执行调度（§3.4）。在数据层，MuxTune 通过序列打包和基于块的分区，在每个空间融合的 hTask 内对齐各任务的可变长度数据（§3.5）。本文的主要贡献如下：

• 我们揭示了数据中心中参数高效微调工作负载的资源利用低效问题，并指出了实现高效多任务并发执行的机遇与挑战。

• 我们介绍 MuxTune，这是一种高效且可扩展的微调系统，通过分层的空间-时间骨干网络复用支持多任务参数高效微调模型。

• 我们提出模块化的骨干网络共享方案，以实现灵活且可扩展的时空复用。在此基础上，我们设计了细粒度的分层协同调度方案，整合任务、算子和数据级优化，以系统性地提升资源效率。

我们构建了 MuxTune，并在配备 NVIDIA A40 和 H100 GPU 的测试平台上，使用四种大语言模型和三个参数高效微调数据集对其进行评估。实证结果表明，与三种最先进的基线方法相比，MuxTune 的吞吐量最高可提升2.33×，内存最多可减少 5.29× 。MuxTune 已开源，地址为 https://github.com/sjtu-epcc/muxtune。

2 背景与动机

2.1 参数高效微调（PEFT）

基础知识。 参数高效微调通过使用可训练适配器，以经济高效的方式将预训练大语言模型适配到特定领域，而不是从头开始训练大语言模型。如图 2所示，它由三类具有代表性的方法组成 [43, 62]: （1）加性（例如适配器调优 [46]），将适配器插入模型架构中的特定位置。（2）选择性（例如差分剪枝 [41]），通过二进制掩码对部分参数进行微调。（3）重参数化（例如 LoRA [47]），对原始模型参数构建低秩变换。

[1]阶段内指数据并行（DP）、张量并行（TP） [78] 及其他变体; [56,57],阶段间指流水线并行（PP） [65]。

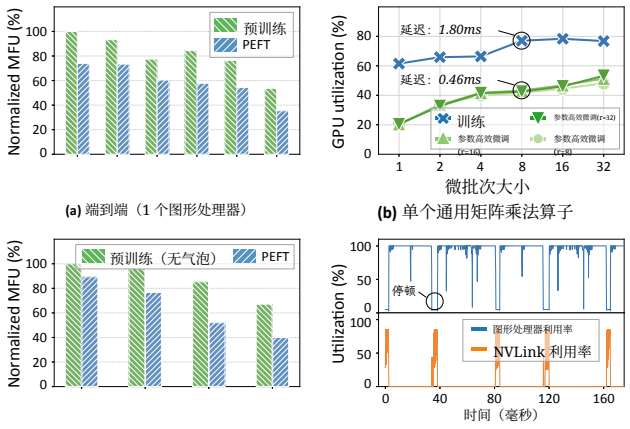


图 3：参数高效微调的低效之处（MBS：微批次大小，序列长度为 128）。（a）8 层模型在单GPU上的 MFU（全局批次大小为 32，LLA：LLaMA7B，GPT：GPT2.7B）。（b）算子利用率（形状 `[MBS,128,4096]×[4096,r],r=4096`，用于预训练）。（c）完整模型在多GPU上的 MFU（全局批次大小为 128）。（d）图形处理器和 NVLink 利用率。

特征。 数据中心中的参数高效微调工作负载呈现出鲜明的特征。首先，参数高效微调支持灵活地配置和附加适配器，而不会影响骨干网络，因为适配器管理的是独立参数。

其次，许多参数高效微调任务会在相同类型的骨干网络上进行微调，因为开发者有数千名 [9]，但可用的骨干网络却很少（例如，OpenAI 有 7 个 [1], 3，Gemini 有 [6]）。这些任务保留了彼此独立且长度可变的批次（即数据异构性），同时共享骨干网络算子（适配器相关算子除外）（即骨干同质性）。一些基于参数高效微调的多模态研究还提出，使用相同语料库在单一骨干网络上微调多个适配器，并将每个适配器作为独立任务 [42, 69]。

第三，服务提供商在提供微调 API 时，会定义、实例化并执行模型，因此能够完全访问模型的具体信息。这不同于传统的基础设施提供商，后者只负责分配硬件和运行脚本，而任务则由其框架在内部进行优化。

最后，与预训练相比，一些领域特定的参数高效微调语料（如情感分析 [82] 和短文本生成 [88],）具有更短的序列长度（例如，64）和更小的语料库规模 [4, 50]。为确保模型的泛化能力，先前工作 [47, 80] 主张在模型微调时采用更小的批次大小（例如，GPT-3 [47]使用 128）。此外，由于不同参数高效微调语料所关注的领域各不相同，其序列长度也存在显著差异 [12, 15, 79]。

2.2 参数高效微调工作负载的低效率

参数高效微调工作负载在先进图形处理器上通常存在资源利用低效（§5.1），如图 3 所示，采用两项指标：(1) *GPU* 利用率，用于衡量占用率，针对

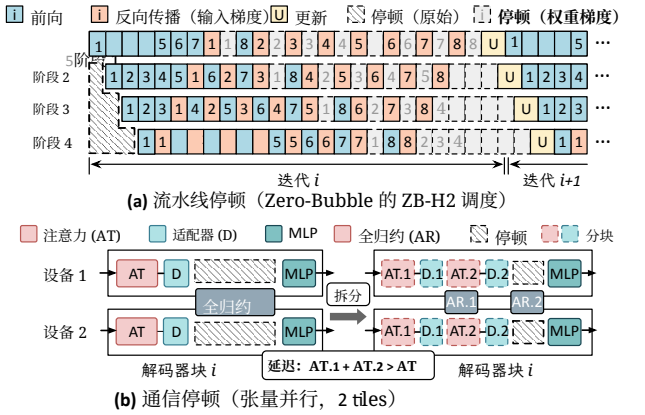


图 4：模型并行下参数高效微调中的设备停顿。

流式多处理器（SMs） [24]；（2）模型浮点运算次数利用率 （MFU），用于衡量端到端效率 [85]。

利用率不足。 图 3（a）显示，与模型预训练相比，参数高效微调的单 GPU MFU 更低（最高低至 1.47×）。主要原因在于，参数高效微调省略了计算密集型的骨干权重梯度，并引入了小规模算子，例如 LoRA 下投影 [47] 和前缀微调的可学习向量 [60]。例如，LoRA秩（最大为 64）64.0×小于 LLaMA7B 的隐藏层大小（4096）。图 3（b）显示，这些小型算子会带来不可忽略的延迟（例如，预训练中为 0.46ms，而预训练中为 1.80ms）以及 GPU 利用不足（差距最高达 40.9%）。由于它们附着于不连续的骨干算子（例如，每个解码器块中的 `qkv-proj` 算子 [84]），严格的相互依赖关系阻碍了直接进行垂直或水平融合。此外，由于参数高效微调语料通常具有更短的序列和更小的批次大小（§2.1），输入规模受到限制，进一步削弱了GPU并行计算的效率。更糟糕的是，我们在不同 GPU 架构上开展的实验（配置如图 3（a）所示）表明，在 NVIDIA V100、A40 和 RTX6000 上，参数高效微调的平均 MFU 为0.84×、 0.68×、 0.59× 的预训练水平，说明更高端的硬件反而会加剧利用不足问题。

设备停顿。在模型并行中，并行化算子之间的通信延迟和数据依赖会导致设备停顿。PEFT适配器会增加阶段延迟和阶段内通信开销，从而加剧停顿 [76]。更糟的是，诸如无停顿流水线 [63, 70] 和通信重叠 [26, 86],等先进训练优化无法直接应用于PEFT。这会进一步降低多GPU执行期间的MFU，降幅最高可达 1.65× （图3(c)，劣于`1-GPU` 情形）。图3(d)展示了存在显著停顿时，4-GPU张量并行中的GPU和NVLink利用率。下文将展示两种停顿类型，并讨论PEFT如何加剧这些停顿。

首先是 流水线停顿 （也称为气泡），这是由流水线冲刷和阶段间依赖导致的（图 4（a））。先前工作（如 DualPipe [63],ZB-H2 [70]）通过将反向传播拆分为输入

和权重梯度，并通过精细调度实现近零气泡流水线。由于缺少骨干权重梯度，参数高效微调（PEFT）本身无法支持这种技术。如图所示，与原有停顿不同，由省略权重梯度引起的停顿会随微批次数量线性增长，因此无法摊销。在参数高效微调中直接采用 DualPipe 会使吞吐量降低 1.16×，相比 1F1B [65]。

第二种是张量并行中的通信停顿，由通信延迟和算子依赖导致（图4（b））。先前工作 [26, 86] 通过将计算分解为更小的分块以便与通信重叠，来减少这些停顿。如图所示，由于 PEFT 本身就存在利用率不足的问题，这种分解会加剧该问题（利用率下降 24。5%），并使整体延迟增加 1.17×（对于使用 2 个图形处理器的 GPT2.7B）。PEFT 中还存在其他类型的停顿，例如由参数同步引起的停顿 [74, 92] 。

2.3 直观的复用方法

粗粒度空间复用会导致可扩展性差、性能提升有限。提高利用率的一种直接方法是通过 NVIDIA MPS 将多个 PEFT 任务共同部署 [23] 或使用多流 [21], , 类似于 [48, 66]。然而，这种方法的效用受内存限制（图❶ 中的5）。为说明这一点，我们分析了 LoRA LLaMA7B（批大小为 8，序列长度为 128）的内存构成：骨干网络参数和激活值分别占用 13。4GB 和 4。3GB，而总内存占用为 18。1GB。使用 4 个 A40 GPU（每个 48GB）时，在不进行并行化的情况下最多只能共同部署 8 个任务，因此无法扩展到更大的模型或支持更多并发任务。此外，由于缺乏对算子执行的细粒度控制，停顿减少效果不理想，并可能产生任务间干扰（例如， 2.5×中报告的性能下降 [95]）。

基于批处理的空间复用会导致停顿加剧和计算效率低下。 以往的多 LoRA 服务系统 [34, 76] 采用简单的批处理方式，将多个请求合并起来并发计算。虽然这种方法适用于服务，但与参数高效微调工作负载并不匹配（❷ 见图 5）。首先，反向传播会带来流水线停顿问题，而仅前向传播服务通常不存在这一问题 [30]。过度批处理会加剧设备停顿，并损害端到端性能（§3.3）。其次，批处理带来的收益逐渐递减

与计算受限的参数高效微调相比，服务的解码阶段受内存限制，因此收益更大 [29]。最后，服务中使用的连续批处理 [90] 避免了填充的需要。尽管填充对于对齐参数高效微调中的可变长度序列仍不可或缺，但计算效率低下（§3.5）。

时间复用和重叠会导致 GPU 利用率进一步降低。受近期预训练和服务系统的启发，将其他任务的通信与计算在时间上重叠的做法， [33, 95],对于参数高效微调（PEFT）也适得其反（❸ 见图 5）。具体而言，时间复用会依次执行每个任务的算子

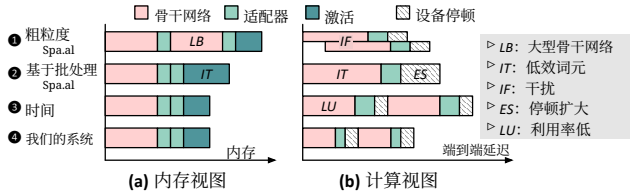


图5：直观的复用方法概览。

，而不是利用批处理机会来提高算子内利用率。鉴于参数高效微调工作负载的输入规模通常有限，过度增大单个任务的微批次大小也可能并不可行。

2.4 我们的方法与挑战

我们发现，可以通过时空骨干网络复用来克服这些局限：通过空间维度上的批处理和时间维度上的交错，让多个任务共享骨干网络。这一范式形成了一个复杂的优化空间，包含多个相互耦合的维度，例如，在调度多任务流水线执行的同时，确定最优的时空组合。为降低优化的复杂性，我们提出将该空间进行层次化分解，划分为三个层级，并分别应对各层级特有的挑战：

任务级：应对空间-时间权衡。 空间复用与时间复用之间存在固有的权衡（§3.3）。这给动态调度策略的设计带来了首要挑战：如何智能地在空间上批处理任务以提高利用率，同时在时间上交错执行任务，以隐藏流水线停顿和通信停顿。

算子层：无停顿多任务编排。如果没有算子级执行控制，粗粒度空间复用就无法缓解停顿。在混合并行模式下，跨空间复用和时间复用任务进行算子编排，是一个复杂的双层问题（§3.4）。这需要量身定制的算法和细粒度协调，才能实现无停顿执行和高 GPU 利用率。

数据层：缓解任务间的无效计算。对具有可变长度序列的任务进行空间复用，自然需要沿序列维度对齐。诸如零填充或将序列打包成长序列等简单策略，要么会将计算浪费在无效词元上，要么会降低效率。如果缺乏周密的对齐设计，这些影响会削弱多任务复用带来的收益。

3 系统设计

3.1 概述

MuxTune 是一个高效且可扩展的多任务参数高效微调系统，作为大语言模型微调 API 的后端，用于提升多租户 GPU 数据中心中的资源效率。MuxTune 的核心理念是以时空方式在相互独立的参数高效微调任务之间复用骨干网络

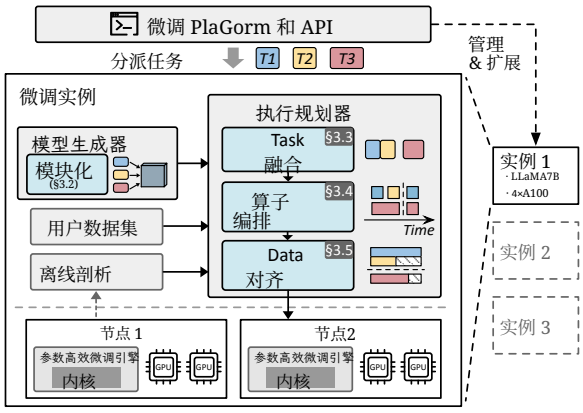


图6：MuxTune 的架构概览。

通过灵活的模块化骨干网络共享，以及细粒度的分层多任务协同调度。

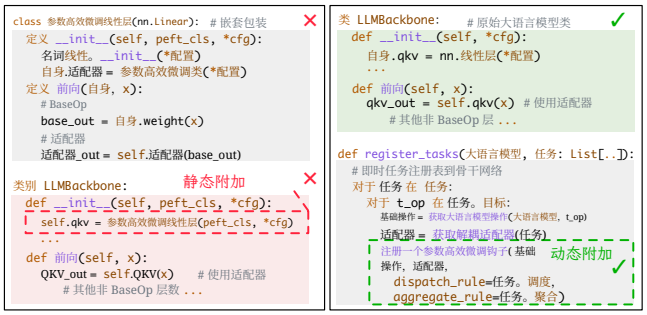
图 6 展示了 MuxTune 的架构，它包含三个主要模块：模型生成器、规划器和参数高效微调引擎，旨在应对§2.4 中的挑战。首先，用户配置其参数高效微调任务（例如，骨干网络类型），并通过微调 API 提交任务。集群调度器会根据调度策略（例如基于预算的 [19]），将具有相同骨干网络的任务调度到运行中的实例，或创建新实例。对于已调度的任务，模型生成器 基于模块化构建包含多任务适配器的参数高效微调模型（§3.2）。利用该参数高效微调模型、用户数据集和性能分析数据，执行规划器 通过自适应组合空间与时间复用，将任务融合为混合任务（§3.3）。随后，规划器在两级混合并行（§3.4）下，统筹这些混合任务的细粒度算子执行。从数据角度看，数据批次以流式方式加载，并在空间批处理任务之间对齐（§3.5）。运行时，参数高效微调引擎 通过高效融合且重叠的内核并发执行已调度的参数高效微调任务。

3.2 骨干网络共享

MuxTune 采用灵活的模块化骨干网络共享方式，将具有不同工作负载的 PEFT 任务共同部署。本小节介绍 PEFT 模块化，以及如何在多任务子模块间共享骨干网络，以实现高效复用。

PEFT 模块化。 如图7(a) 所示，现有的单任务框架 [18,44] 静态地 将适配器注入大语言模型骨干网络，即直接将某个任务的适配器视为大语言模型层。虽然这种静态实现直观易懂，但在多任务 PEFT 场景中无法支持适配器数量和类型不断变化的动态工作负载，因为应对任务即时到达或完成事件需要从头重新初始化模型。

相比之下，MuxTune 采用解耦适配器，以支持灵活的用户定制，并通过集群调度器实现高效扩展。具体而言，通过考察§2.1 中的主流 PEFT类型，我们将通用的 PEFT工作流抽象为



(a) 静态嵌套适配器实现　　**(b)** 即时适配器附加

图7：现有的静态嵌套适配器实现，以及我们基于模块化的动态适配器挂接。

四个子模块：(i) *BaseOp* 是骨干网络中的一种算子，可能会附加适配器，例如 QKV 和线性投影（不包括注意力）。(ii)适配器 描述 PEFT算法（例如 LoRA [47]），并由用户定制。(iii) *Dispatch* 定义多任务数据分发规则，为 *BaseOp*和适配器准备输入张量。(iv) *Aggregate* 定义数据聚合规则，用于收集来自*BaseOp* 和 适配器的输出张量。

动态多任务骨干网络共享。 如图 7(b) 所示，与以往框架不同，MuxTune 在保持对大语言模型骨干网络非侵入性的同时，提供了一个`register_tasks()` API，用于处理任务即时到达事件。这是多任务骨干网络共享的基石。当集群调度器将新任务分配给一个运行中的实例时，本地模型生成器会主动调用此 API，将任务注册到复用骨干网络上，无需付出代价高昂的模型重新初始化（§4）。此后，MuxTune 会自动执行下文介绍的任务级、算子级和数据级优化，并在运行时高效执行生成的多任务 PEFT 模型。

隔离与收敛保证。 在通过多任务骨干网络共享提升效率的同时，MuxTune 为任务间隔离和一致的模型收敛提供严格保证。具体而言，MuxTune 安全地实例化大语言模型骨干网络和用户定义的适配器（§2.1），从而防止大多数运行时错误（例如语义错误），如 [51]所述。通过细粒度内存建模和算子编排，MuxTune 还缓解了任务间性能干扰和内存溢出风险。

在 MuxTune 中，时间交错任务中的*BaseOp*和适配器在时间上天然隔离，不会影响收敛。对于空间批处理任务，*BaseOp*沿批次维度拼接，并呈现数学隔离。以一个通用矩阵乘法*BaseOp* 及两个适配器为例，*BaseOp* 的前向计算为：

$$BW_g = [B_1, B_2]_b W_g = [B_1 W_g, B_2 W_g]_b, \quad (BaseOp \text{ fwd}) \quad (1)$$

其中，B_i 是任务i的批次，B 是拼接后的批次，而W_g 是通用矩阵乘法权重。在反向传播中，每个B_i 都独立计算损失并进行反向传播——

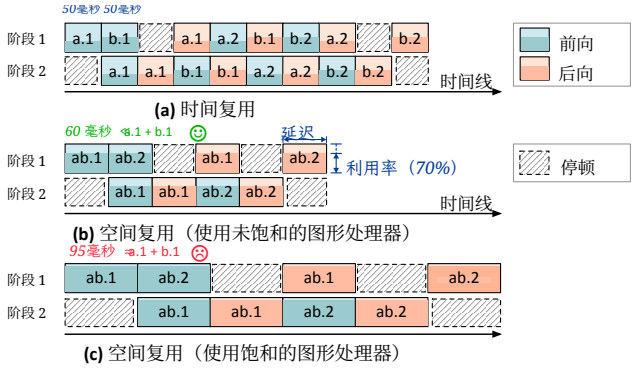


图8：空间与时间复用的权衡。x.i 表示（混合）任务 x 的第i个微批次。

而相同*BaseOp*的计算则会合并成批处理，如下所示：

$$G^{in} = [G_1^{in}, G_2^{in}]_b \leftarrow [G_1^{out}, G_2^{out}]_b W_g^{\mathrm{T}}, \quad (BaseOp \text{ bwd}) \quad (2)$$

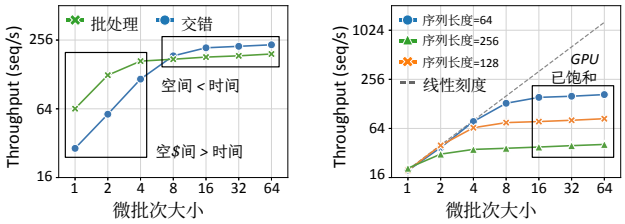
其中，G_i^{in}和G_i^{out}分别是任务i的输入梯度和输出梯度。G^{in}是输入到*BaseOp*的临时梯度缓冲区。其他不可批处理算子（例如适配器）以融合方式独立计算（§3.4.3）。这种隔离既能保持收敛一致（例如均方偏差为 0.07），又能避免任务间的数值故障传播（例如学习率过大导致梯度出现 NaN）。

3.3 任务融合

借助模块化表示和灵活的接口，MuxTune 能够以不同方式复用骨干网络：（1）通过执行批处理的*BaseOp*、*Dispatch*、融合后的适配器和*Aggregate*，在空间上批处理参数高效微调任务；或（2）通过依次执行每个任务的*BaseOp*、*Dispatch*、适配器和*Aggregate*，在时间上交错执行各任务。

时空权衡。 图 8 展示了这两种复用方案。基于 交错的时间复用 按顺序计算各个任务，并在任务之间重叠停顿与计算，但使用独立的 适配器 且输入规模有限时，可能导致图形处理器利用不足（§2.2）。基于批处理的空间复用 将 *BaseOp* 进行批处理，并融合 适配器 与独立任务，以提高利用率。然而，过度批处理会延长算子（或阶段）的延迟，并加剧并行执行中的停顿。

最优复用决策取决于任务输入规模、参数高效微调参数和并行策略。如图 9(a) 所示，当图形处理器未饱和时，对任务进行空间批处理可获得更好的性能，而在较高 GPU 利用率下，时间交错更为可取。这一转变源于超出 GPU 饱和后过度空间批处理所引起的收益递减（图 9(b)）。例如，理想情况下批处理 8 个任务，每个任务的微批次大小为 8、序列长度为 128，吞吐量仅提升 1.12×，远低于预期的 8× 增益。值得注意的是，上述分析仅



(a) 两种方案的权衡　(b) 批处理的次线性扩展

图 9：两种方案的定量分析。(a) 在具有 4-GPU流水线的 16 层 LLaMA7B 上运行 2 个任务（4 个微批次，序列长度 64）。(b) 在 1 个图形处理器上运行 8 层 LLaMA7B 的 1 个任务。

涉及两个任务，而扩展到更多任务会以指数级提高此类复用决策的复杂度。

混合任务抽象。 鉴于调度的复杂性，要得出最优的复用决策，需要对所有任务进行分层调度。我们引入混合任务抽象（简称“hTask”），以统一这两种复用方案；它由独立任务融合而成。在一个 hTask 内部，任务以空间方式融合并进行批处理；而在多个 hTask 之间，其计算则以时间交错的方式执行。

任务融合被表述为一个装箱问题。最初，M 个任务 $T = \{T_1, T2, ..., T_M\}$ 被共同部署在骨干网络上（共 S 个阶段，每个阶段有 $N_g^{(s)}$ 个图形处理器）。由于延迟与输入规模正相关（这是由 §2.1 中描述的骨干网络同质性所致），我们按照 token 数量（第 i 个任务的 token 数量记为 n_i）升序排列 T。令 $H_{i\to j}$ 表示包含任务 i 至 j 的 hTask。所有任务统一采用 C 个微批次，该值通过经验确定。每个任务的序列都会填充至最大长度 l_i，详见 §3.5。

代价模型。 我们构建了一个代价模型，用于评估混合并行下的端到端延迟和内存占用。在参数高效微调中，同一阶段的前向和反向传播具有相近的延迟（因为不存在权重梯度），因此我们将 hTask $\mathcal{H}_{i\to j}$ 在第 s 个阶段（阶段。）的延迟建模为：

$$\begin{aligned} L^{(s)}(\mathcal{H}_{i\to j}) = & \sum_{o\in St.^{(s)}} t_o\left(\sum_k n_k\right)/N_g^{(s)} \quad (BaseOp \text{ Lat.}) \\ & + \sum_{\{a\}\in St.^{(s)}} \max\left\{\begin{array}{l}\sum_k u_a \cdot t_a(n_k), \\ \max_k t_a(n_k)\end{array}\right\}, \quad (Adapter \text{ Lat.}) \end{aligned} \quad (3)$$

其中，第一行对分片到$N_g^{(s)}$个图形处理器上的*BaseOp*的延迟进行建模（$k \in [i, j]$）。$t_o(x)$表示包含x 个词元的计算算子 o 的延迟（通信按 §3.4.2 所述进行重叠）。第二行估算融合适配器 $\{a\}$ 的延迟，其中$u_a(x)$和$t_a(x)$分别表示包含x 个词元时*Adapter a* 的图形处理器利用率和延迟。我们使用加权和$u_a \cdot t_a(x)$来估算水平融合（§3.4.3）后的总体延迟，同时将其限制在适配器的最大单独延迟以内，以避免瓶颈效应。在包含S 个阶段的流水线中，$\mathcal{H}_{i\to j}$ 的端到端延迟为：

$$L(\mathcal{H}_{i\to j}) = 2\sum_{s=1}^{S-1}\{L^{(s)}(\mathcal{H}_{i\to j})\} + 2C\max_{1\le s\le S} L^{(s)}(\mathcal{H}_{i\to j}), \quad (4)$$

其中，第一项估算预热和排空阶段的延迟总和，而第二项则对包含C 个微批次 [52]。的稳态阶段的总体延迟进行建模。

至于内存占用，我们发现它在参数高效微调中主要由三部分组成：(i) 骨干网络参数 M_b，(ii) 任务 i 的输入梯度 $M_g^{(i)}$，以及 (iii) 激活值 $M_a^{(i)}$。对于 1F1B 流水线，每阶段的最大内存估计为：

$$M_{stage} = [M_b + \sum_{i=1}^{M} M_g^{(i)}]/S + \sum_{i=1}^{M} M_a^{(i)}(b_i, l_i), \tag{5}$$

其中前两项与输入规模无关（例如，微批次大小 b_i）。第三项最多累积到S份，且正比于 b_i和序列长度 l_i。在实践中，M_g 通常复用 M_a [28] 的已分配内存。该内存模型用于评估混合任务（hTask）在构建过程中是否会导致内存溢出（OOM）问题。经 §5.3 评估，它与实测内存占用的缩放规律精确吻合。

基于动态规划算法的任务融合。基于成本建模，我们采用动态规划（DP）算法，以在将 M 个任务装箱为 N 个混合任务时最小化端到端流水线延迟。我们推导出状态转移方程，其中 $F(m,n)$ 表示将前 m 个任务打包为 n 个混合任务时的最小端到端延迟（$m \in [1,M], n \in [1,N], m \geq n$）：

$$\begin{aligned} F(m,n) &= \min_{n-1 \leq i \leq m} \{F(i,n-1) + L(\mathcal{H}_{(i+1)\to k})/S\}, \\ F(m',1) &= L(\mathcal{H}_{1\to m'}), \quad \forall m' \in [1,M], \end{aligned} \tag{6}$$

其中，$H^{(i+1)\to k}$ 对 F(m, n) 的影响通过平均阶段延迟来估算，以满足动态规划的最优子结构。这是因为稳态阶段通常主导端到端延迟，在该阶段，$H^{(i+1)\to k}$ 会增加一次前向—后向传播 [61]。最优融合方案推导为 $F^* = \min_{1\leq N\leq M}\{$ F(M, N)$\}$。值得注意的是，最小化端到端延迟也能平衡各混合任务之间的负载，因为流水线受其最慢阶段的限制 [45]。虽然时间复杂度为 $O(M^2(S+M))$，但当每个骨干网络包含的任务数量适中时，该算法开销极小（§5.3），并且可通过对 N 进行并行计算来加速。

3.4 算子编排

任务融合后，MuxTune 会在算子[2] 粒度上编排混合任务执行，涵盖空间与时间复用，以及阶段内和阶段间并行。

解耦双层编排。鉴于阶段内并行和阶段间并行各自独特的特点，我们通过将 N 混合任务$H = \{H_1, H_2..., H_N\}$划分为 P个分桶 $G = \{G_1, G_2, ..., G_P\}$。同一桶中的混合任务在一个流水线时钟周期内交错执行（即阶段内并行），而不同桶则跨流水线时钟周期交错执行（图 10）。

[2]为统一阶段内和阶段间并行，我们使用“算子”同时表示层（例如通用矩阵乘法）和阶段粒度计算。

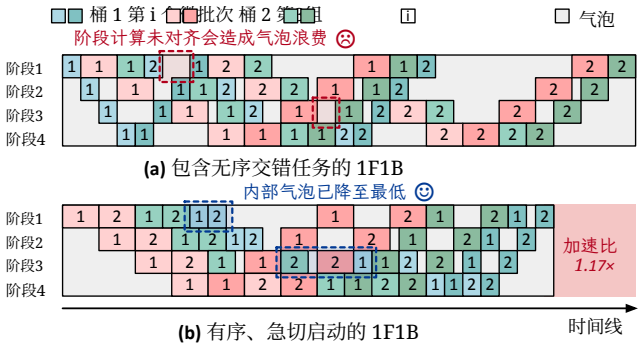


图 10：阶段间编排示例。浅色/深色分别表示 hTask桶的前向/反向传播。

负载均衡混合任务分组。为了高效确定最优分组策略，我们基于如下观察将 hTask 分组与算子编排解耦：在给定固定的 P 时，均衡的工作负载会带来更少的内部气泡和更低的端到端延迟。我们遍历P，从 1（将所有混合任务归为一组）到N（每个桶中一个混合任务），通过最小化桶间方差来优化$\mathcal{G}$：

$$\begin{aligned} \mathcal{G}^*(P) = &\arg\min_{\mathcal{G}=\{\mathcal{G}_1,...,\mathcal{G}_P\}} \sum_{j=1}^{P} |L^{(1)}(\mathcal{G}_j) - \overline{L^{(1)}(\mathcal{G})}|^2, \\ &\text{s.t.} \quad L^{(1)}(\mathcal{G}_j) = \sum_i L^{(1)}(\mathcal{H}_i), \ \forall \mathcal{H}_i \in \mathcal{G}_j, \end{aligned} \tag{7}$$

其中，$L^{(1)}(\mathcal{H}_i)$ 表示 hTask $\mathcal{H}_i$ 的第一阶段延迟，这是因为该步骤侧重于平衡各组之间的工作负载。随后，我们调用下面的编排过程，对每个固定的 P 建模端到端延迟，并选择具有最小延迟的最优分组策略$\mathcal{G}^*$。

3.4.1 阶段间编排

为了优化 hTask 桶$\mathcal{G} = \{\mathcal{G}_1, ..., \mathcal{G}_P\}$的阶段间执行，目标是调度$\{\mathcal{G}_j\}_{1\leq j\leq P}$的微批次，以最小化端到端流水线延迟。

计算同质性。微批次调度涉及一个可重入流水车间问题 [39],其中 $\{\mathcal{G}_j\}_{1\leq j\leq P}$会在前向和反向传播中被各阶段重复处理。不同于以往采用复杂调度方法的训练工作，[52],我们观察到 参数高效微调中的计算同质性：(i) 在每个桶 $\mathcal{G}_j$ 中，微批次在各次迭代中保持一致的形状（§3.5），而不同桶之间的形状可能不同；(ii) 对于桶 $\mathcal{G}_j$，由于不存在权重梯度，前向和反向传播具有相同的执行时间。

这些特性使我们能够缩减调度空间并降低算法开销。具体而言，借助 (i)，MuxTune 采用统一的流水线模板，以结构化方式执行多任务迭代，而不是依赖代价高昂的逐次迭代实时调度。借助 (ii)，如图10所示，MuxTune 通过交错执行延迟相近的桶的前向和反向传播，高效地最大限度减少内部气泡（例如图10(b) 中下方的蓝色区块）。

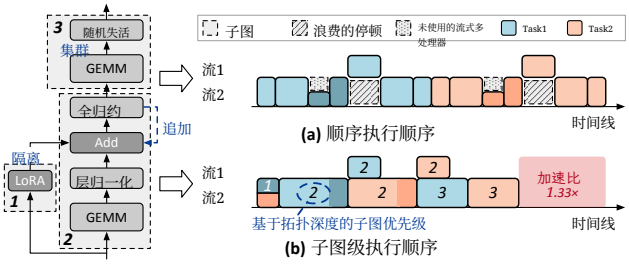


图 11：阶段内编排示例。左侧是一个计算图示例，其中较深色的方块表示适配器。

结构化流水线模板。流水线模板会根据桶信息（例如阶段延迟和内存占用）动态生成，以支持结构化多任务执行。具体而言，为减少气泡，模板生成会通过以下三条规则扩展 1F1B 流水线：(1) 按$\mathcal{G}$的$L^{(1)}(\mathcal{G}_j)$降序排列，使$\mathcal{G}_j$能够填补$\mathcal{G}_{j-1}/\mathcal{G}_{j+1}$（图10(b)）。(2) 将同一桶的微批次连续排列，因为它们的延迟完全匹配。(3) 在内存限制 (§3.3) 内尽可能积极地启动更多微批次，以确保每个阶段有足够的待处理批次 [96]。通过这些规则，MuxTune 在 1.17× 的同时提高效率，并保持结构化执行。我们在附录 §A 中提供了详细的最优性分析，证明这种结构化流水线始终能减少最后阶段的内部气泡，从而实现近乎最优的执行。

3.4.2 阶段内编排

为优化 hTask 桶 $\mathcal{G}_j$ 内的阶段内执行，我们对混合任务 $\{\mathcal{H}_i\} \in \mathcal{G}_j$ 的计算算子和通信算子进行调度，以缓解设备停顿，这等价于最小化阶段延迟，如图11所示。

依赖关系感知的图构建。 每个混合任务的计算图都是一个有向无环图（DAG），因此自然可以将跨混合任务的阶段内编排视为一个多 DAG 调度问题 [68]。MuxTune 使用子图 作为最小编排单元，因为模型执行是按顺序进行的，而较长的计算算子序列有利于实现通信的充分重叠。

子图的构建采用 依赖关系感知 的方式。MuxTune 首先通过聚集连续的计算算子，并将每个通信算子追加到其依赖的算子上，将每个 DAG 划分为若干子图（图 11 左侧）。小规模适配器被隔离为独立子图。然后，MuxTune 根据每个子图的拓扑深度为其分配优先级。这既确保了交错执行，也遵循了由适配器融合所产生的跨图依赖（§3.4.3）。值得注意的是，子图这一概念与基于垂直融合的内核和 CUDA 图技术兼容 [10, 20]。

子图调度。给定多个分段的有向无环图，MuxTune 将 Kahn 算法 [54] 扩展为多有向无环图

算法 1 基于优先级的子图调度

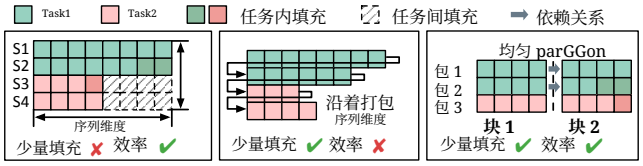
```
输入: 混合任务 H = {H_1, ..., H_n}
函数 SubgraphSchedule(H)
    P ← 优先级队列()
    for H_i ∈ H 做                                  ▷ 初始化队列
        P.enqueue(GetZID子图(H_i.图))
    启动调度_ ← ∅; t ← 0
    当 P ≠ ∅ 时执行
        子图 ← P.dequeue(max_lat)                   ▷ 最高优先级
        有向无环图 ← 子图.parent_图                  ▷ 父有向无环图
        有向无环图.移除(子图)
        P.入队(GetZID子图(有向无环图))
        启动_调度.记录(<子图, t >)
        t ← t + 子图.延迟                           ▷ 更新计时器
    返回 启动_调度
```

变体，并以延迟感知的方式进行调度。算法1 概述了调度过程。给定初始混合任务后，它会维护一个优先级队列$\mathcal{P}$，用于跟踪每个有向无环图中的零入度子图（第 3–5 行）。每次迭代中，算法会筛选出优先级最高的子图，并选取累积延迟最长的子图（根据其内部算子计算），以最大限度地与在途通信算子重叠。随后，该子图会从其有向无环图中移除，同时将新出现的零入度子图加入队列（第 6–13 行）。通过启动_调度，MuxTune 能够高效协调流水线阶段内多任务子图的运行时执行（§4）。

3.4.3 适配器融合与通信重叠

为了提高小规模、原生支持参数高效微调的算子之图形处理器利用率，我们采用水平适配器融合，并对图形处理器资源进行细粒度管理（§4），因为适配器无法跨任务直接进行批处理。该融合策略包含三种情况：(1) 对于同一 hTask 内的空间批处理任务，将其适配器进行融合；(2) 对于同一桶内的 hTask（即在阶段内并行中交错执行），如果每个 hTask 都只分配了单个任务，则在不妨碍任务间重叠的情况下融合其适配器。如图11所示，`LoRA`算子中的`Task1` 和`Task2` 可以进行融合，因为它们不在全归约算子的同一子图中。相反，加法算子无法融合，因为融合会在全归约算子之前引入全局同步，而这些算子属于`Task1` 和`Task2`。(3) 不会跨桶进行融合（即在阶段间并行中交错执行）。

在阶段内编排中跨不同流重叠计算与通信时（图11），我们观察到，通信内核使用过多协作线程数组会降低计算内核的效率，而协作线程数组过少则无法充分利用 NVLink 连接。为解决这一权衡，我们采用 NVLink SHARP [27]，将归约卸载到 NVSwitch 中，仅使用少量协作线程数组预算即可维持接近峰值的链路带宽。因此，网络内核仅使用 8 个协作线程数组，就能与其他任务的计算完全重叠。



(a) 零填充至最大长度 **(b)** 打包序列 **(c)** 基于分块的对齐

图 12：现有数据对齐策略和我们的基于分块的对齐示例（Si：第i个序列）。依赖关系表示因果注意力计算中的 KV 缓存复用。

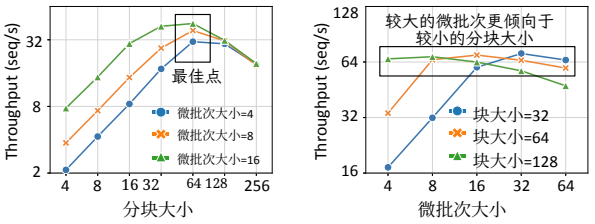
3.5 数据对齐

要执行混合任务（hTask），其内部参数高效微调任务的数据批次需要沿序列维度对齐。一种常见策略是将所有序列零填充至全局最大长度（图12(a)）。这会产生大量任务间无效词元（即不包含语义信息的词元），从而浪费计算和内存资源。另一种用于预训练的工业级方案是将序列打包成更长的序列。然而，近期研究 [31, 52] 发现，仅依赖打包会降低微调效率，因为注意力掩码会导致序列间的注意力计算被浪费。因此，微调 API 通常要求将序列填充至最大长度，并向用户收取任务内零填充词元的费用 [3]。然而，在 MuxTune 中，任务间无效词元源自协同调度中的数据对齐，无法向用户收费。因此，MuxTune 专注于减少任务间无效词元。

通过基于分块的对齐革新打包方式。为减少任务间填充，MuxTune采用双步对齐策略，在不损害模型质量的前提下最大限度地提高效率（图12(c)）。首先，针对每个任务，分别在单个全局批次内自适应地打包序列，以确保不影响模型收敛（例如，包 1/包 2 用于`Task1`，而包 3 用于`Task2`）。此步骤将各任务的序列转换为一组更长、更密集的打包序列。其次，MuxTune将打包序列均匀划分为大小相等的分块（例如，图12(c)中的分块大小为 4）。对于长度超过分块大小的序列（例如，包 1），MuxTune借助KV缓存复用，将其分散到多个连续分块中 [61]。此步骤不仅减少了冗余的跨序列注意力计算，还将过长的打包序列拆分为较短的序列，以支持更细粒度的流水线，从而有助于提高吞吐量并减少峰值激活内存。

分块大小的确定需要在计算效率与填充缩减之间进行权衡，如图所示13。较小的分块大小可以减少填充词元，但可能导致利用率不足并增加 KV 缓存访问；相反，过大的分块则不利于填充缩减，并且会降低……

阶段延迟。在实践中，我们将分块大小设为所有序列长度的最大公约数中最大的 2 的幂，并设定一个最小阈值（通常为 64），以避免利用率不足。



(a) 将一个微批次划分的多个分块 **(b)** 将多个微批次划分为固定大小分块大小（Global-BS=128）

图13：量化基于分块的对齐（1个任务，16层LLaMA7B，4-GPU流水线，序列长度256）。

4 实现

我们基于 Megatron-LM，使用 14K 行 Python 代码构建了 MuxTune [78]和 PyTorch [28],，以及 2K 行 C 代码++和 CUDA 实现内核。MuxTune 支持采用混合并行进行部署，包括流水线（例如 GPipe [49], 1F1B [65],交错式 1F1B [67]）、Megatron 张量并行 [78], 和数据并行策略（例如 PyTorch 分布式数据并行 [59],全分片数据并行 [92]）。MuxTune 通过 PyTorch 适配器挂接到*BaseOp*，并利用钩子机制，[14],将*Dispatch* 和 *Aggregate*子模块的逻辑封装到钩接的 *BaseOp* 函数中，从而实现动态挂接。

分组内核。我们基于 NVIDIA CUTLASS 和 CuTe [81] 实现 MuxTune 的内核，用于跨任务适配器的分组计算。在每个内核中，我们首先按各任务适配器的浮点运算次数和内存访问量成比例地分配线程块。然后，我们将每个适配器算子分解为分块，并把访问算子权重同一部分的分块分配给同一个已分配的线程块。该实现在增强 SMs 间负载均衡的同时降低内存读写开销。

子图执行。我们使用 TorchFX [16] 将参数高效微调模型追踪为多任务中间表示图，并执行子图级编排（§3.4）。在此基础上，我们采用多流和 CUDA 原语（例如`Event.synchronize()`）来协调独立任务之间的算子执行。

离线剖析与系统开销。鉴于硬件和骨干网络可预先访问，我们对典型算子配置进行离线剖析以构建代价模型。这得益于 PyTorch 的*Dispatch*机制，该机制可确保对相同输入形状、数据类型和硬件进行一致的内核选择 [13]。因此，我们通过避免劳动密集型 GPU 操作，将调度开销限制在 10 秒以内；与通常持续 3-70 小时的微调任务相比，这一开销微不足道。

5 评估

5.1 实验设置

测试平台。我们在三种服务器配置上评估 MuxTune：(1) 测试平台 A：1 个节点，配备 4 个 NVIDIA A40 GPU（48GB）、Intel

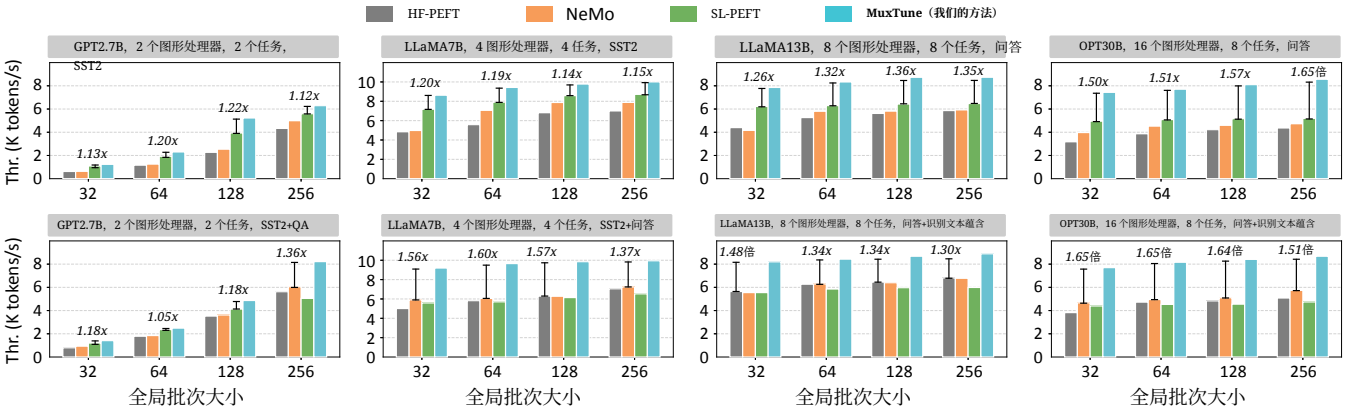


图14：不同全局批次大小、骨干模型和硬件配置下的系统吞吐量（每秒处理的词元数）。各子图上方（灰色方框中）展示了详细工作负载。

表1：实验中使用的模型配置。#图形处理器表示每个模型使用的图形处理器数量，另有说明的除外。

模型	#层数	隐藏维度	#注意力头数	#图形处理器
GPT3-2.7B [32]	32	2560	32	2
LLaMA2-7B/13B [83]	32/40	4096/5120	32/40	4/8
OPT-30B [91]	48	7168	56	16

Xeon Silver 4310 CPU 和 NVLink；(2) 测试平台-B：8 个节点，每个节点配备 2 个 NVIDIA A40 GPU（48GB）、Intel Xeon Gold 5318Y CPU 和 Mellanox ConnectX-5 [22]（100Gb/s InfiniBand）。(3) 测试平台-C：1 个节点，配备 8 个 NVIDIA H100 GPU（80GB）、Intel Xeon Platinum 8457C CPU 和 NVLink。

模型与数据集。我们使用表1中的四种代表性大语言模型开展实验。我们实现了三种参数高效微调类型：LoRA [47]（主要使用）、适配器微调 [46]和差分剪枝 [41]。我们使用三个序列长度各异的数据集，包括 SST2 [79], OpenBookQA [12],和识别文本蕴含 [15]。由于每个任务的序列都会填充至其最大长度，我们分析数据集分布，并将 SST2、问答和识别文本蕴含的序列分别填充（或截断）至 64、128 和 256。在评估中，我们采用两种数据集组合（§2.1）：(1)均匀：同一骨干网络上共置的各任务共享同一个数据集；(2)非均匀：这些任务采用不同的数据集。该测试平台的规模和工作负载足以评估系统性能，因为参数高效微调所需内存远少于预训练，且输入规模有限（因此不需要大规模数据并行）。**B**

aselines. W比较 MuxT宽度 ith three baselines: (1)*HuggingFace PEFT* [18]（HF-PEFT）：一个用户友好的库，用于将各种大语言模型适配到下游任务，并支持量化等内存优化。(2)*NeMo Megatron* [44]（NeMo）：一个基于 Megatron-LM 构建的人工智能框架 [78]，支持高效内核和可扩展的并行策略，以便使用参数高效微调来适配大语言模型。(3)*SLoRA-PEFT* [76]（SL-PEFT）：它采用 SLoRA 的技术，如骨干网络共享以及参数高效微调中的仅批处理，支持同时微调多个任务。

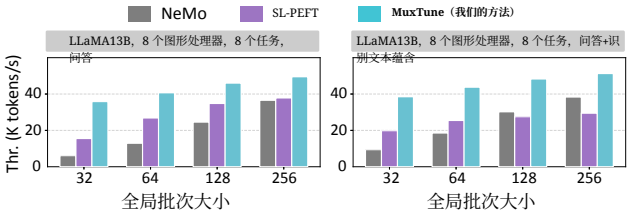


图 15：不同全局批量大小下 H100 GPU 的吞吐量。配置与上述实验保持一致。

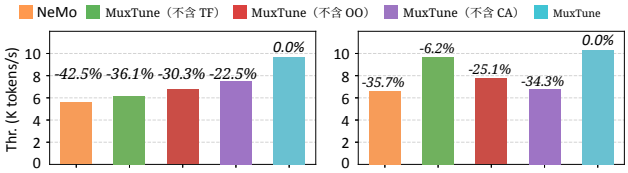
并行选择。我们在所支持的策略中对 MuxTune 和基线方法进行网格搜索，以寻找最优并行（§4）。对于 2 个图形处理器和 4 个图形处理器的情况，搜索分别定位阶段内并行和阶段间并行。对于图形处理器更多且工作负载更重的情况，则选择混合并行（每个节点内采用阶段内并行，跨节点采用流水线）。

5.2 端到端性能

系统吞吐量。图14 展示了 MuxTune 和基线系统在各种工作负载下的端到端吞吐量。在均匀场景中，MuxTune 的吞吐量相较于 HF-PEFT、NeMo 和 SL-PEFT 分别最高提升 2.33×，1.87×和 1.64×。对于轻量级工作负载，MuxTune 通过自适应地对多个任务进行空间融合，提升利用率，从而超越基线方法。得益于 MuxTune 能够重叠多任务算子以减少停顿，随着图形处理器数量增加，性能提升幅度也随之增大。

在非均匀场景中，MuxTune 相较于三个基线方法的吞吐量分别提升 2.23×、1.83× 和 1.85×。对于 HF-PEFT 和 NeMo，其提升幅度与均匀场景一致，因为它们分别执行各项任务，不会产生额外的零填充令牌。对于 SL-PEFT，MuxTune 的提升幅度高于均匀场景，因为 SL-PEFT 会产生大量零填充令牌，浪费计算和内存资源。

先进硬件上的性能。我们进一步评估了 MuxTune 在 H100 GPU 上相对于两种基线方法的表现。如



(a) 2 个任务，4 个微批次，SST2+问答 **(b)** 4 个任务，8 个微批次，问答+识别文本蕴含

图 16：性能分解。TF、OO 和 CA 分别表示任务融合、算子编排和数据对齐。

表 2：实验中使用的随机生成配置（数据集、批大小）对应的任务工作负载（WL）。

任务顺序	#1	#2	#3	#4	#5	#6	#7	#8
WL-A	SST2	QA	QA	SST2	SST2	SST2	QA	QA
WL-B	识别文本蕴含	SST2	识别文本蕴含	SST2	SST2	识别文本蕴含	识别文本蕴含	识别文本蕴含
批大小	4	2	4	4	8	2	4	4

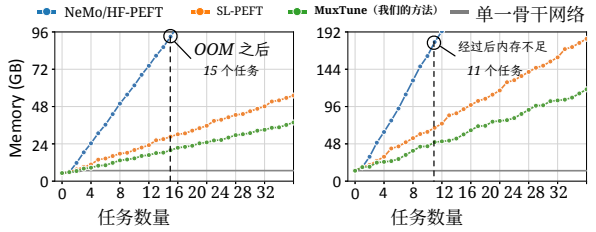
图 15所示，在5.29×和 2.31×方面，MuxTune 在均匀场景（左图）中优于 NeMo 和 SL-PEFT；在 3.69× 和 1.94× 方面，MuxTune 在非均匀场景（右图）中也有提升。除了上述原因外，与 A40 相比，MuxTune 在 H100 上取得更显著的性能提升，主要得益于 H100 更强的计算能力。它放大了单任务参数高效微调框架固有的利用率不足问题，同时释放了 MuxTune 在多任务优化方面的更大潜力。

5.3 消融研究

性能分解。图 16 展示了在使用 LLaMA7B、4-GPU 流水线和全局批次大小为 128 时，MuxTune 各组件的影响。对于轻量级工作负载（图 16(a)），禁用三个组件会分别使吞吐量下降 36.1%、30.3% 和 22.5%。这是因为图形处理器利用率提高（得益于计算融合）、设备停顿减少（得益于通信重叠），以及跨任务的无效词元得到缓解（得益于基于分块的对齐）。对于更重的工作负载（图 16(b)），数据对齐对整体吞吐量的影响日益显著，禁用后会导致吞吐量下降 34.3%。相反，由于图形处理器已达到饱和，禁用任务融合仅会使吞吐量下降 6.25%。由于微批次增多会减少设备停顿，禁用算子编排会导致吞吐量下降 25.1%，略低于轻量级工作负载下的降幅。

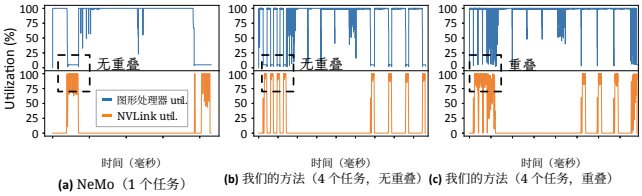
内存效率分析。我们仅使用表 2中列出的两种参数高效微调任务工作负载，评估 MuxTune 与其他基线方法的内存效率。如图 17所示，我们通过重复运行这些工作负载四次，逐步提交 32 个参数高效微调任务，每次迭代执行 1 个微批次。我们对 WL-A 使用 GPT2.7B 骨干网络，对工作负载 B 使用 LLaMA7B，二者的内存消耗分别为 5.2GB 和 13.4GB。

在图 17(a) 中，在 2 个 A40 GPU（每个 48GB）上进行张量并行时，MuxTune 最多可实现 4.67× 和 1.44×



(a) GPT3-2.7B（2-GPU 张量并行） **(b)** LLaMA2-7B（4-GPU流水线）

图 17：不同任务数量下的内存占用。



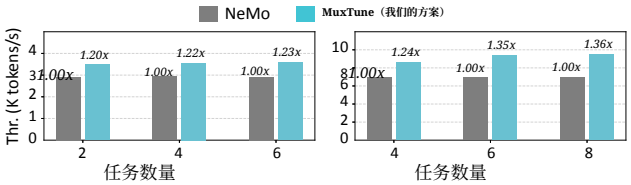
(a) NeMo（1 个任务） **(b)** 我们的方法（4 个任务，无重叠） **(c)** 我们的方法（4 个任务，重叠）

图 18：采用 4 路 GPU 张量并行时，单层的图形处理器和 NVLink 利用率。(b) 和 (c) 中的任务采用交错执行。

相较于 NeMo/HF-PEFT（运行 15 个任务后内存不足）和 SL-PEFT，内存分别减少了 5.29× 和 1.46× 。在不受内存限制的情况下（即扩展至 32 个任务），与这些基线方法相比，MuxTune 进一步减少了 5.29× 和 1.46× 内存占用。原因有两点：(1) MuxTune 可灵活地在各任务间共享内存密集型骨干网络，其内存占用为 13.4GB，而激活值占用为 4.3GB，其他部分占用 0.4GB（图17(b)）；相比之下，NeMo 和 HF-PEFT 会为每个任务复制一个骨干网络；(2) MuxTune 可减少任务间无效词元，并受益于细粒度流水线（§3.5），而 SL-PEFT 会过度批处理任务，导致大量填充和更高的峰值激活内存。对于更大的骨干网络和更多的图形处理器（图 17(b)），相较于 NeMo/HF-PEFT（运行 11 个任务后内存不足）和 SL-PEFT，MuxTune 可将内存占用减少 3.57× 和 1.37×，表明 MuxTune 相比基线系统具有更好的可扩展性。

算子编排的效率。我们展示了 MuxTune 与 NeMo 基线的图形处理器计算和网络带宽利用率对比。图 18 使用 Nsight 工具包 [25] 对张量并行下的图形处理器和 NVLink 利用率进行分析。如图 18(a) 所示，NeMo 通过顺序启动算子来执行单个任务。由于计算受通信阻塞，图形处理器平均利用率保持在 82。5%，单层延迟为 43。2 毫秒。图 18(b) 展示了 4 个任务交错执行但不重叠时的结果。延迟线性增加至 172。5 毫秒，而图形处理器利用率几乎保持不变，为 84。7%。图 18(c) 中，MuxTune 实现了计算与通信的完全重叠。图形处理器不再受阻，利用率达到 97。8%，较基线 1.19×有所提升，同时，解码器层处理 4 个任务的延迟降至 156。2 毫秒。

我们还评估了不同并行策略和工作负载下的端到端性能，如图所示——



(a) 张量并行（4 个图形处理器） **(b)** 1F1B 流水线（4 个图形处理器）

图 19：不同任务数量下算子编排的吞吐量（LLaMA7B，序列长度为 128、64、32）。（a）1 个大小为 8 的微批次，（b）8 个大小为 8 的微批次。

19 所示，仅启用骨干网络共享和算子编排。图19（a）展示了张量并行中任务间重叠的优势。随着任务数量增加，得益于计算与通信的重叠，MuxTune 的吞吐量比 NeMo 高出 1.20×、 1.22×和 1.23× 。图19（b）展示了流水线编排的效果。与 NeMo 相比，MuxTune 的吞吐量提升了 1.24×、 1.35×和 1.36×，因为跨任务交错执行阶段计算能有效减少流水线气泡。值得注意的是，微批次数量较少（流水线气泡更多）时，MuxTune 也能实现更高的提升（例如， 1.59× 使用 4 个微批次时）。

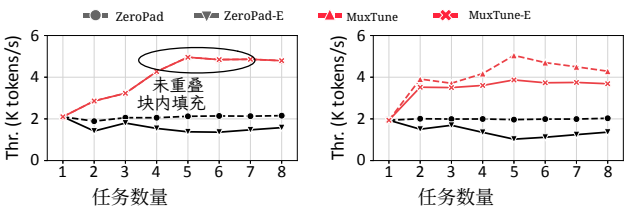
数据对齐的有效性。我们使用指标有效吞吐量 [8],来评估基于分块的数据对齐。该指标衡量原始词元（不包括任务间零填充的词元）的吞吐量，反映了服务提供商的经济收益（§3.5）。我们使用表2 中的工作负载，评估采用和不采用块内零填充的情况。

图20 展示了逐步将任务加入单个混合任务（含一个微批次）时的吞吐量。图20(a)采用与 SST2 匹配的 64 分块大小时，MuxTune 避免了块内填充，吞吐量最高提升 2.33× ，有效吞吐量则比 SL-PEFT 高出 3.59× 。这是因为 MuxTune 能有效减少任务间填充，同时避免图形处理器利用不足。图20(b)中，序列长度的差异更大，MuxTune 为分块大小为 128 的 SST2 任务引入了块内零填充。即便如此，与 SL-PEFT 相比，MuxTune 的整体吞吐量仍提升了最高 3.77× ，有效吞吐量提升了 2.57×。

5.4 可扩展性与调度研究

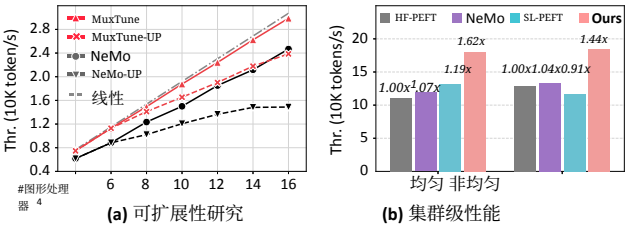
可扩展性研究。 图 21(a) 展示了采用两种扩展策略时系统的可扩展性（LLaMA7B，全局批次大小为 128，微批次大小为 8，n 个任务，使用 n 个图形处理器）：(1)“仅向上扩展”（-UP）：随着工作负载增加，为活跃实例扩展资源（即增加分配的图形处理器）；(2)“先向上再向外扩展”：先向上扩展，若工作负载继续增加，则向外扩展（即复制新实例）。对于“仅向上扩展”策略，尽管扩展呈次线性（§

3.3),MuxTune 通过有效提高利用率并减少设备停顿，取得了比 NeMo 1.61×更高的吞吐量。对于“先向上再向外扩展”



(a) 工作负载-A（SST2 + 问答） **(b)** 工作负载-B（SST2 + 识别文本蕴含）

图20：1 个混合任务在不同任务数量和数据集数量下的吞吐量（LLaMA7B，4-GPU流水线）。ZeroPad 表示像 SL-PEFT 中那样对所有序列进行零填充。-E 表示有效吞吐量，未标注时表示总体吞吐量。



(a) 可扩展性研究 **(b)** 集群级性能

图21：MuxTune 的系统可扩展性，以及生产级工作负载下的集群级性能。

策略，两种系统都实现了近线性扩展，而 MuxTune 的吞吐量仍最高可提升 1.28×。

集群级性能。 我们进一步将 MuxTune 集成到集群调度中，并在生产级工作负载下对其进行评估。由于缺乏公开的参数高效微调跟踪数据，我们采用了一周的费城跟踪数据 [51]。任务平均持续时间和标准差分别为 372.6 分钟和 612.9 分钟，平均到达率为每分钟 2.59 个任务。我们在一个配备 128 个图形处理器的模拟集群中重放参数高效微调工作负载，使用先到先服务调度器，并分别采用 均匀和 非均匀 组合、LLaMA7B骨干网络，以及为每个任务随机生成的配置。如图21(b) 所示，在 均匀 情况下，与 HF-PEFT、NeMo 和 SL-PEFT 基线方法相比，MuxTune 分别将集群吞吐量提升了 1.61×、 1.51×和 1.36× 。非均匀 情况下，MuxTune 相较于 SL-PEFT 进一步实现了 1.58×更高的集群吞吐量，这凸显了在变长序列工作负载下融合多个任务时，基于分块的数据对齐的有效性。

6 讨论与未来工作

对集群调度策略的普适性。除了评估中使用的先到先服务调度器（§5.4）之外，MuxTune 的系统设计具有可扩展性，可作为其他集群调度策略的微调后端，例如基于预算的 [19],基于任务优先级的 [66], 以及感知 SLO 的调度 [40]。例如，在考虑任务优先级时，调度器可以将低优先级任务共置，以提高实例级吞吐量，同时为高优先级任务分配专用资源

，从而保证任务级延迟。此外，骨干网络类型相同的任务可以共置以实现骨干网络多路复用，而类型不同的任务则应调度到不同实例上。我们将对“多路复用感知”调度的深入探索留待未来工作。

面向性能指标优化的可扩展性。MuxTune 的优化侧重于通过提升资源效率和减少每任务内存占用，最大限度地提高实例级吞吐量。除此之外，MuxTune 还能通过减少设备停顿造成的浪费并缩短所有共置任务的总体耗时，实现更高的能源效率 [36]。用户可以扩展我们的方法，以优化其他性能指标，例如集成准入控制（即限制分派到微调实例的任务数量），从而确保所有共置任务都能在用户指定的服务等级目标内完成。此外，用户可以在满足服务等级目标要求的同时，自适应地调整硬件频率，以进一步提升能源效率 [36, 71]。

7 相关工作

多适配器大语言模型系统。近年来，多适配器大语言模型系统取得了显著进展 [34,38, 76, 89, 94]。PetS [94] 提出了一个用于并发参数高效微调任务服务的统一框架。Punica [34], SLoRA [76],和 dLoRA [89] 致力于多 LoRA 服务系统的高效调度与内核实现。与这些工作不同，MuxTune 面向参数高效微调场景下的多任务执行优化，通过分层协同调度最大限度地提高资源利用率并减少设备停顿。

提升 GPU 利用率的技术。多项研究通过跨多个机器学习应用进行资源分配与管理，探索了 GPU 利用率优化，包括通过时间 [5, 77] 和空间 [11, 21, 23] 共享来复用计算单元。其他研究 [26, 58] 则将小型内核横向或纵向融合为较大的内核，以充分利用硬件。然而，由于内存密集型骨干网络（限制了可扩展性）、缺乏算子级执行控制（会造成任务间干扰），以及同一任务的适配器之间存在相互依赖，这些方法无法直接应用于参数高效微调。

大语言模型并行化与减少设备停顿的框架。 近年来，大语言模型并行化领域得到了深入研究，包括流水线 [49, 65],张量 [78], 和数据并行 [73, 74], 以及自动并行优化 [73, 93]。为解决这些并行策略引入的设备停顿问题，DeepSeek-V3 [63] 和ZeroBubble [70]拆分反向传播以减少流水线气泡。基于重叠的方法 [26, 86] 将计算分解，并使其与通信重叠。TeraPipe [61] 对词元级计算进行流水线化，以优化长序列训练。这些技术不适用于参数高效微调，因为缺少权重梯度计算，并且可能出现 GPU 利用不足。

8 结论

MuxTune是一个资源高效的系统，用于优化多租户数据中心中并发多任务参数高效微调的执行。MuxTune的核心是以时空方式在独立任务之间复用骨干网络。MuxTune将PEFT任务模块化，以实现灵活的骨干网络共享，并设计了包含任务、算子和数据级优化的分层多任务协同调度方案，以提高GPU 利用率并减少设备停顿。实验结果表明，与三种基线方法相比，MuxTune的吞吐量最高提升 2.33×，内存减少5.29×。

致谢

我们谨向匿名审稿人以及我们的导师 Jiarong Xing 致谢，感谢他们提出的宝贵意见。本研究得到国家自然科学基金（62232011）和上海市自然科学基金（24ZR1430500）的资助。Quan Chen 是本文的通讯作者（chen-quan@cs.sjtu.edu.cn）。

参考文献

[1] OpenAI 微调平台。 https://platform.openai.com/docs/guides/fine-tuning/，2023年。[2] OpenPipe 微调快速入门。 https://docs.openpipe.ai/features/fine-tuning/quick-start，2023年。[3] Together AI 微调命令行界面。 https://docs.together.ai/docs/fine-tuning-cli，2023年。[4] 使用 Hugging Face 和 LoRA 在单个 Amazon SageMaker 图形处理器上训练大语言模型。 https://aws.amazon.com/cn/blogs/machine-learning/train-a-large-language-model-on-a-single-amazon-sagemaker-gpu-with-hugging-face-and-lora/，2023年。[5] 阿里云 CGPU。 https://www.alibabacloud.com/help/en/elastic-gpu-service/latest/what-is-the-cgpu-service，2024年。[6] 使用 Gemini API 进行微调。 https://ai.google.dev/gemini-api/docs/model-tuning，2024年。[7] 开始自定义大语言模型 (LLM)。 https://learn.microsoft.com/en-us/azure/ai-services/openai/concepts/customizing-llms，2024年。[8] 有效吞吐量。https://en.wikipedia.org/wiki/Goodput，2024年。

[9] 为微调 API 引入视觉功能。 https://openai.com/index/introducing-vision-to-the-fine-tuning-api/, 2024年。

[10] Nvidia Apex 扩展。 https://nvidia.github.io/apex/，2024年。

[11] Nvidia 多实例图形处理器。 https://www.nvidia.com/en-us/technologies/multi-instance-gpu/, 2024年。

[12] OpenBookQA 数据集。 https://huggingface.co/datasets/allenai/openbookqa，2024年。

[13] PyTorch 分发器。 http://blog.ezyang.com/2020/09/lets-talk-about-the-pytorch-dispatcher/, 2024年。

[14] PyTorch 钩子机制。 https://pytorch.org/docs/stable/generated/torch.nn.Module.html, 2024年。[15] 文本蕴含识别数据集。 https://huggingface.co/datasets/SetFit/rte，2024年。[16] Torch FX。 https://pytorch.org/docs/stable/fx .html，2024年。[17] 理解自定义大语言模型：2024年指南。 https://botpenguin.com/blogs/understanding-custom-llm-models，2024年。[18] HuggingFace PEFT 库。 https://huggingface.co /docs/peft/index, 2025年。[19] Kubernetes。 https://kubernetes.io/, 2025年。[20] Nvidia CUDA 图 API。 https://docs.nvidia.com/cuda/cuda-runtime-api/group__CUDART__GRAPH. html，2025年。[21] Nvidia CUDA 流管理。 https://docs.nvidia.com/cuda/cuda-runtime-api/group__CUDA RT__STREAM.html，2025年。[22] Nvidia Mellanox ConnectX-5。 https://www.nvidia .com/en-us/networking/ethernet/connectx-5/, 2025年。[23] Nvidia 多进程服务。 https://docs.nvidia. com/deploy/mps/index.html, 2025年。[24] Nvidia Nsight Compute 自定义指南。 https: // docs.nvidia.com/nsight-compute/CustomizationGuide/index.html，2025年。

[25] NVIDIA Nsight Systems。 https://developer.nvidia.com/nsight-systems，2025年。

[26] NVIDIA Transformer Engine。 https://github.com/NVIDIA/TransformerEngine，2025年。

[27] NVLink 和 NVSwitch。 https://www.nvidia.com/en-us/data-center/nvlink/，2025年。

[28] PyTorch 基金会。https://pytorch.org，2025年。

[29] Amey Agrawal、Nitin Kedia、Ashish Panwar、Jayashree Mohan、Nipun Kwatra、Bhargav Gulavani、Alexey Tumanov 和 Ramachandran Ramjee。《借助 Sarathi-Serve 驯服大语言模型推理中的吞吐量—延迟权衡》。载于第 *18* 届 *USENIX* 操作系统设计与实现研讨会（*OSDI 24*），第 117–134 页，圣克拉拉（加利福尼亚州），2024 年 7 月。USENIX 协会。

[30] Reza Yazdani Aminabadi、Samyam Rajbhandari、Am- mar Ahmad Awan、Cheng Li、Du Li、Elton Zheng、Olatunji Ruwase、Shaden Smith、Minjia Zhang、Jeff Rasley 等。DeepSpeed-Inference：以空前规模实现 Transformer 模型的高效推理。载于*SC22*：国际高性能计算、网络、存储与分析会议，第 1–15 页。IEEE, 2022年。[31] Yushi Bai、Xin Lv、Jiajie Zhang、Yuze He、Ji Qi、Lei Hou、Jie Tang、Yuxiao Dong 和 Juanzi Li。LongAlign：大语言模型长上下文对齐方案。*arXiv* 预印本 *arXiv:2401.18058*，2024年。[32] Tom Brown、Benjamin Mann、Nick Ryder、Melanie Subbiah、Jared D Kaplan、Prafulla Dhariwal、Arvind Neelakantan、Pranav Shyam、Girish Sastry、Amanda Askell 等。语言模型是少样本学习者。神经信息处理系统进展，33：1877–1901，2020年。[33] Chang Chen、Xiuhong Li、Qianchao Zhu、Jiangfei Duan、Peng Sun、Xingcheng Zhang 和 Chao Yang。Centauri：通过通信分区实现大模型训练中通信与计算重叠的高效调度。载于第 *29* 届 *ACM* 国际编程语言与操作系统架构支持会议论文集，第 *3* 卷，第 178–191 页，2024年。[34] Lequn Chen、Zihao Ye、Yongji Wu、Danyang Zhuo、Luis Ceze 和 Arvind Krishnamurthy。Punica：多租户 LoRA 服务。机器学习与系统论文集，6：1–13，2024年。[35] Marcin Chrapek、Anjo Vahldiek-Oberwagner、Marcin Spoczynski、Scott Constable、Mona Vij 和 Torsten Hoefler。筑牢基础：云端基础模型部署的实用隐私与安全，2024年。

[36] Jae-Won Chung、Yile Gu、Insu Jang、Luoxi Meng、Nikhil Bansal 和 Mosharaf Chowdhury。降低大模型训练中的能源浪费。载于*ACM SIGOPS* 第 *30* 届操作系统原理研讨会论文集，SOSP '24，第 144–159 页，美国纽约州纽约，2024年。计算机协会。

[37] Jacob Devlin、Ming-Wei Chang、Kenton Lee 和 Kristina Toutanova。Bert：用于语言理解的深度双向 Transformer 预训练。*arXiv* 预印本 *arXiv:1810.04805*，2018年。

[38] Wei Gao、Weiming Zhuang、Minghao Li、Peng Sun、Yonggang Wen 和 Tianwei Zhang。Ymir：数据中心基础模型微调工作负载的调度器。载于第*38*届 *ACM*国际超级计算会议论文集，第259–271页，2024年。

[39] Stephen C Graves、Harlan C Meal、Daniel Stefek 和 Abdel Hamid Zeghmi。重入式流水车间的调度。《运营管理杂志》，3(4):197–207，1983年。

[40] Diandian Gu、Yihao Zhao、Yinmin Zhong、Yifan Xiong、Zhenhua Han、Peng Cheng、Fan Yang、Gang Huang、Xin Jin 和 Xuanzhe Liu。Elasticflow：用于分布式深度学习的弹性无服务器训练平台。载于第*28*届国际编程语言与操作系统架构支持会议论文集，第*2*卷，第266–280页，2023年。

[41] Demi Guo、Alexander M Rush 和 Yoon Kim。基于差分剪枝的参数高效迁移学习。*arXiv* 预印本 *arXiv:2012.07463*，2020年。

[42] Avelina Asada Hadji-Kyriacou 和 Ognjen Arand- jelovic。Context-PEFT：高效的多模态多任务微调。*arXiv* 预印本 *arXiv:2312.08900*，2023年。

[43] Zeyu Han、Chao Gao、Jinyang Liu、Sai Qian Zhang 等。大型模型的参数高效微调：全面综述。*arXiv* 预印本 *arXiv:2403.14608*，2024年。

[44] Eric Harper、Somshubra Majumdar、Oleksii Kuchaiev、Li Jason、Yang Zhang、Evelina Bakhturina、Vahid Noroozi、Sandeep Subramanian、Koluguri Nithin、Huang Jocelyn、Fei Jia、Jagadeesh Balam、Xuesong Yang、Micha Livne、Yi Dong、Sean Naren 和 Boris Gins- burg。NeMo：面向对话式人工智能和大语言模型的工具包。

[45] John L Hennessy 和 David A Patterson。《计算机体系结构：定量方法》。Morgan Kaufmann，2017年。

[46] Neil Houlsby、Andrei Giurgiu、Stanislaw Jastrzebski、Bruna Morrone、Quentin De Laroussilhe、Andrea Ges- mundo、Mona Attariyan 和 Sylvain Gelly。面向自然语言处理的参数高效迁移学习。载于国际机器学习大会，第 2790–2799 页。PMLR，2019年。

[47] Edward J Hu、Yelong Shen、Phillip Wallis、Zeyuan Allen-Zhu、Yuanzhi Li、Shean Wang、Lu Wang 和 Weizhu Chen。LoRA：大语言模型的低秩适配。*arXiv* 预印本 *arXiv:2106.09685*，2019年。

[48] Qinghao Hu、Meng Zhang、Peng Sun、Yonggang Wen 和 Tianwei Zhang。Lucid：面向深度学习训练作业的非侵入式、可扩展且可解释的调度器。载于第*28*届 *ACM* 国际编程语言与操作系统架构支持会议论文集，第*2*卷，第457–472页，2023年。

[49] Yanping Huang、Youlong Cheng、Ankur Bapna、Orhan Firat、Dehao Chen、Mia Chen、HyoukJoong Lee、Jiquan Ngiam、Quoc V Le、Yonghui Wu 等。GPipe：利用流水线并行高效训练巨型神经网络。神经信息处理系统进展，第32卷，2019年。

[50] Mathav Raj J、Kushala VM、Harikrishna Warrier 和 Yogesh Gupta。《企业大语言模型微调：实用指南与建议》，2024年。

[51] Myeongjae Jeon、Shivaram Venkataraman、Amar Phan- ishayee、Junjie Qian、Wencong Xiao 和 Fan Yang。《面向深度神经网络训练工作负载的大规模多租户 GPU 集群分析》。载于*USENIX* 年度技术会议，第 947–960 页，2019年。

[52] Chenyu Jiang、Zhen Jia、Shuai Zheng、Yida Wang 和 Chuan Wu。《Dynapipe：通过动态流水线优化多任务训练》。载于第十九届欧洲计算机系统会议论文集，第 542–559 页，2024年。

[53] Ziheng Jiang、Haibin Lin、Yinmin Zhong、Qi Huang、Yangrui Chen、Zhi Zhang、Yanghua Peng、Xiang Li、Cong Xie、Shibiao Nong、Yulu Jia、Sun He、Hongmin Chen、Zhihao Bai、Qi Hou、Shipeng Yan、Ding Zhou、Yiyao Sheng、Zhuo Jiang、Haohan Xu、Haoran Wei、Zhang Zhang、Pengfei Nie、Leqi Zou、Sida Zhao、Liang Xiang、ZheruiLiu、Zhe Li、XiaoyingJia、JianxiYe、Xin Jin 和 Xin Liu。《Megascale：将大语言模型训练扩展至 10,000 多个图形处理器》，2024年。

[54] A. B. Kahn。大型网络的拓扑排序。*ACM*通讯，5(11):558–562，1962年11月。

[55] Richard M Karp。组合问题之间的可约性。载于整数规划五十年*1958-2008*：从早期到最新进展，第219–241页。施普林格，2009。

[56]Vijay Korthikanti、Jared Casper、Sangkug Lym、Lawrence McAfee、Michael Andersch、Mohammad Shoeybi 和 Bryan Catanzaro。《减少大型 Transformer 模型中的激活值重计算》，2022年。

[57] Dmitry Lepikhin、HyoukJoong Lee、Yuanzhong Xu、De- hao 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。*arXiv* 预印本 *arXiv:2006.16668*，2020年。

[58] Ao Li、Bojian Zheng、Gennady Pekhimenko 和 Fan Long。GPU内核的自动水平融合。载于*2022 IEEE/ACM* 国际代码生成与优化研讨会（*CGO*），第14–27页。IEEE，2022年。

[59] Shen Li、Yanli Zhao、Rohan Varma、Omkar Salpekar、Pieter Noordhuis、Teng Li、Adam Paszke、Jeff Smith、Brian Vaughan、Pritam Damania等。PyTorch 分布式：加速数据并行训练的经验。*arXiv* 预印本 *arXiv:2006.15704*，2020年。

[60] Xiang Lisa Li 和 Percy Liang。前缀微调：优化用于生成的连续提示。*arXiv* 预印本 *arXiv:2101.00190*，2021年。

[61] Zhuohan Li、Siyuan Zhuang、Shiyuan Guo、Danyang Zhuo、Hao Zhang、Dawn Song 和 Ion Stoica。TeraPipe：用于训练大规模语言模型的令牌级流水线并行。在国际机器学习会议，第 6543–6552 页。PMLR，2021年。

[62]Vladislav Lialin、Vijeta Deshpande 和 Anna Rumshisky。缩小规模以实现扩展：参数高效微调指南。*arXiv* 预印本 *arXiv:2303.15647*，2023年。

[63] Aixin Liu、Bei Feng、Bing Xue、Bingxuan Wang、Bochao Wu、Chengda Lu、Chenggang Zhao、Chengqi Deng、Chenyu Zhang、Chong Ruan 等。DeepSeek-V3 技术报告。*arXiv* 预印本 *arXiv:2412.19437*，2024年。

[64] Yongkang Liu、Yiqun Zhang、Qian Li、Shi Feng、Dal- ing Wang、Yifei Zhang 和 Hinrich Schütze。HiFT：一种分层全参数微调策略。*arXiv* 预印本 *arXiv:2401.15207*，2024年。

[65] Deepak Narayanan、Aaron Harlap、Amar Phanishayee、Vivek Seshadri、Nikhil R Devanur、Gregory R Ganger、Phillip B Gibbons 和 Matei Zaharia。Pipedream：面向深度神经网络训练的通用化流水线并行。载于第 *27* 届 *ACM* 操作系统原理研讨会论文集，第 1–15 页，2019年。

[66]Deepak Narayanan、Keshav Santhanam、Fiodar Kazhamiaka、Amar Phanishayee 和 Matei Zaharia。面向深度学习工作负载的异构性感知集群调度策略。载于第 *14* 届 *USENIX* 操作系统设计与实现会议论文集，第 481–498 页，2020年。

[67] Deepak Narayanan、Mohammad Shoeybi、Jared Casper、Patrick LeGresley、Mostofa Patwary、Vijay Korthikanti、Dmitri Vainbrand、Prethvi Kashinkunti、Julie Bernauer、Bryan Catanzaro 等。使用 Megatron-LM 在图形处理器集群上进行高效的大规模语言模型训练。载于高性能计算、网络、存储与分析国际会议论文集，第 1–15 页，2021年。

[68] Pál András Papp、Georg Anegg 和 Albert-Jan N Yzelman。BSP 模型中的有向无环图调度。载于 计算机科学理论与实践当前趋势国际会议，第 238–253 页。施普林格，2025年。

[69] Jonas Pfeiffer、Aishwarya Kamath、Andreas Rücklé、Kyunghyun Cho 和 Iryna Gurevych。AdapterFusion：用于迁移学习的非破坏性任务组合。*arXiv* 预印本 *arXiv:2005.00247*，2020年。

[70] Penghui Qi、Xinyi Wan、Guangxing Huang 和 Min Lin。零气泡流水线并行。*arXiv* 预印本 *arXiv:2401.10241*，2023年。

[71] Haoran Qiu、Weichao Mao、Archit Patke、Shengkun Cui、Saurabh Jha、Chen Wang、Hubertus Franke、Zbigniew Kalbarczyk、Tamer Ba¸sar 和 Ravishankar K. Iyer。使用μ-Serve实现功耗感知深度学习模型服务。发表于*2024*年 *USENIX* 年度技术会议（*USENIX ATC 24*），第75–93页，圣克拉拉，加利福尼亚州，2024 年 7 月。USENIX 协会。

[72] Alec Radford、Jeffrey Wu、Rewon Child、David Luan、Dario Amodei、Ilya Sutskever 等。语言模型是无监督多任务学习器。*OpenAI* 博客，1(8):9，2019年。

[73] Samyam Rajbhandari、Jeff Rasley、Olatunji Ruwase 和 Yuxiong He。ZeRO：面向训练万亿参数模型的内存优化。载于*SC20*：高性能计算、网络、存储与分析国际会议，第1–16页。IEEE，2020年。

[74] Jie Ren、Samyam Rajbhandari、Reza Yazdani Aminabadi、Olatunji Ruwase、Shuangyan Yang、Minjia Zhang、Dong Li 和 Yuxiong He。 {ZeRO-Offload}：让 {十亿级} 模型训练惠及大众。发表于*2021*年 *USENIX* 年度技术会议（*USENIX ATC 21*），第 551–564 页，2021年。

[75] Baptiste Roziere、Jonas Gehring、Fabian Gloeckle、Sten Sootla、Itai Gat、Xiaoqing Ellen Tan、Yossi Adi、Jingyu Liu、Tal Remez 等。Code Llama：面向代码的开放基础模型。*arXiv* 预印本 *arXiv:2308.12950*，2023年。

[76] Ying Sheng、Shiyi Cao、Dacheng Li、Coleman Hooper、Nicholas Lee、Shuo Yang、Christopher Chou、Banghua Zhu、Lianmin Zheng、Kurt Keutzer 等。S-LoRA：为数千个并发 LoRA 适配器提供服务。*arXiv* 预印本 *arXiv:2311.03285*，2023年。

[77] Lin Shi、Hao Chen、Jianhua Sun 和 Kenli Li。vCUDA：虚拟机中的 GPU 加速高性能计算。*IEEE Transactions on Computers*，61(6):804– 816，2011年。

[78] Mohammad Shoeybi、Mostofa Patwary、Raul Puri、Patrick LeGresley、Jared Casper 和 Bryan Catanzaro。Megatron-LM：利用模型并行训练数十亿参数的语言模型。*arXiv* 预印本 *arXiv:1909.08053*，2019年。

[79] Richard Socher、Alex Perelygin、Jean Wu、Jason Chuang、Christopher D Manning、Andrew Y Ng 和 Christopher Potts。基于情感树库的语义组合性递归深度模型。载于*2013*年自然语言处理实证方法会议论文集，第1631–1642页，2013年。

[80] Andreas Steiner、Alexander Kolesnikov、Xiaohua Zhai、Ross Wightman、Jakob Uszkoreit 和 Lucas Beyer。如何训练视觉变换器？视觉变换器中的数据、增强与正则化，2022年。

[81] Vijay Thakkar、Pradeep Ramani、Cris Cecka、Aniket Shivam、Honghao Lu、Ethan Yan、Jack Kosaian、Mark Hoemmen、Haicheng Wu、Andrew Kerr、Matt Nicely、Duane Merrill、Dustyn Blasig、Fengqi Qiao、Piotr Ma-jcher、Paul Springer、Markus Hohnerbach、Jin Wang 和 Manish Gupta。CUTLASS，2023年1月。

[82] Nirmalya Thakur、Vanessa Su、Mingchen Shao、Kesha A. Patel、Hongseok Jeong、Victoria Knieling 和 Andrew Bian。一个用于分析 YouTube、TikTok 及其他来源上有关2024年麻疹疫情视频的情感分析带标签数据集，2024年。

[83] Hugo Touvron、Louis Martin、Kevin Stone、Peter Albert、Amjad Almahairi、Yasmine Babaei、Nikolay Bashlykov、Soumya Batra、Prajjwal Bhargava、Shruti Bhosale 等。Llama 2：开放基础模型与微调聊天模型。*arXiv* 预印本 *arXiv:2307.09288*，2023年。

[84] Ashish Vaswani、Noam Shazeer、Niki Parmar、Jakob Uszkoreit、Llion Jones、Aidan N Gomez、Łukasz Kaiser 和 Illia Polosukhin。注意力就是你所需要的一切。神经信息处理系统进展，第30卷，2017年。

[85] Borui Wan、Mingji Han、Yiyao Sheng、Zhichao Lai、Mo-fan Zhang、Junda Zhang、Yanghua Peng、Haibin Lin、Xin Liu 和 Chuan Wu。Bytecheckpoint：面向大语言模型开发的统一检查点系统。*arXiv* 预印本 *arXiv:2407.20143*，2024年。

[86]Shibo Wang、Jinliang Wei、Amit Sabne、Andy Davis、Berkin Ilbeyi、Blake Hechtman、Dehao Chen、Karthik Srinivasa Murthy、Marcello Maggioni、Qiao Zhang 等。通过分解在大型深度学习模型中将通信与依赖计算重叠。在第 *28* 届 *ACM* 国际编程语言与操作系统架构支持会议论文集，第 *1* 卷中，第 93–106 页，2022年。

[87] Yu Wang. 面向自然语言处理的预训练-微调范式教程。*arXiv* 预印本 *arXiv:2403.02504*，2024年。

[88] Ziwen Wang、Jie Wang、Haiqian Gu、Fei Su 和 Bojin Zhuang。《个性化社交媒体短文本的自动条件生成》，第 56–63 页。施普林格国际出版，2018年。

[89] Bingyang Wu、Ruidong Zhu、Zili Zhang、Peng Sun、Xu- anzhe Liu 和 Xin Jin。 {dLoRA}：为 {LoRA}{大语言模型}服务动态编排请求与适配器。载于第 *18* 届 *USENIX* 操作系统设计与实现研讨会（*OSDI 24*），第 911–927 页，2024年。

[90] Gyeong-In Yu、Joo Seong Jeong、Geon-Woo Kim、Soo- jeong Kim 和 Byung-Gon Chun。Orca：面向基于 {Transformer} 的生成模型的分布式服务系统。载于第 *16* 届 *USENIX* 操作系统设计与实现研讨会（*OSDI 22*），第 521–538 页，2022年。

[91] Susan Zhang、Stephen Roller、Naman Goyal、Mikel Artetxe、Moya Chen、Shuohui Chen、Christopher De- wan、Mona Diab、Xian Li、Xi Victoria Lin 等。OPT：开放式预训练 Transformer语言模型。*arXiv* 预印本 *arXiv:2205.01068*，2022年。

[92] Yanli Zhao、Andrew Gu、Rohan Varma、Liang Luo、Chien-Chin Huang、Min Xu、Less Wright、Hamid Sho- janazeri、Myle Ott、Sam Shleifer 等。PyTorch 全分片数据并行：扩展全分片数据并行的实践经验。*arXiv* 预印本 *arXiv:2304.11277*，2023年。

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

[94] 周哲、魏学超、张洁静和孙光宇。{PetS}：用于 {参数高效} Transformer服务的统一框架。载于*2022*年*USENIX* 年度技术会议（*USENIX ATC 22*），第489–504 页，2022年。

[95] 朱侃、赵一龙、赵亮宇、左格非、Yile Gu、谢德东、高宇飞、徐沁宇、唐天、Zihao Ye 等。《NanoFlow：迈向最优大语言模型服务吞吐量》。*arXiv* 预印本 *arXiv:2408.12757*，2024年。[96] Yonghao Zhuang、赵和煦、Lianmin Zheng、Zhuohan Li、邢爱克、何启荣、Joseph E. Gonzalez、Ion Stoica 和 Hao Zhang。《关于优化模型并行通信》，2024年。

结构化流水线模板的最优性分析

从理论上证明一种流水线调度的最优性并非易事，更何况微批次具有异质性，即微批次大小和序列长度各不相同。穷举所有可能的候选流水线调度，可以表述为一个整数线性规划（ILP）问题，其包含 $S\sum_i n_i$ 个约束；已证明该问题为 NP难 [55]，时间复杂度为$e^{O(S\Sigma_{ni})}$，其中 n_i 是任务i的微批次数量。然而，在我们的参数高效微调场景中，每个 hTask 桶中的微批次形状一致，并且在每个流水线阶段，前向和反向传播的延迟相同（§3.4.1）。这些特性为我们降低问题复杂度提供了机会。下面，我们从理论上讨论所提出流水线模板的最优性。

我们的讨论聚焦于如何在 1F1B 范式下实现多任务流水线最优性（即最小化端到端延迟）[65],，因为它是应用最广泛且效率最高的流水线之一。1F1B 流水线的执行可分为三个阶段：预热、稳态和排空阶段。在多任务 1F1B 流水线中，我们遵循 [52,65]，首先介绍一个基本引理：

引理1 端到端流水线延迟等于三个阶段（T_{warm}、T^{steady}、T_{drain}）的延迟之和，而 T_{warm}/T_{drain} 可按以下方式计算：$(S-1)t_1/(S-1)t_P$。

在上述引理中，C 表示微批次数量，P表示混合任务（hTask）桶的数量，S 表示流水线阶段数。t_i 表示第 i 个桶中微批次的阶段延迟。t_1 和t_P 分别表示按序排列的首个和最后一个 hTask 桶的前向（后向）阶段延迟。然后，如图22（b）和（c）所示，我们给出下一个引理如下：

引理2 在稳态阶段，所有微批次都必须经过最后阶段进行一次前向和一次反向传播，即 $T_{steady} \geq 2C\sum_{i=1}^{P} t_i$。

此后，我们可以将T^{steady}与$T_{warmup}+T_{drain}$之间的延迟比计算为：

$$\frac{T_{steady}}{T_{warmup}+T_{drain}} \geq \frac{2C\sum_{i=1}^{P} t_i}{(S-1)(t_1+t_P)}. \tag{8}$$

由于我们以负载均衡方式构建hTask桶（即最小化桶间方差），此处我们通过假设$t_i = t_{i-1} = t \quad \forall i \in [2,P]$来简化该方程：

$$\frac{T_{steady}}{T_{warmup}+T_{drain}} \geq \frac{2CPt}{2(S-1)t} = \frac{CP}{S-1}. \tag{9}$$

在常见实践中，微批次的数量被设置为远大于S，以降低流水线气泡比例（例如，

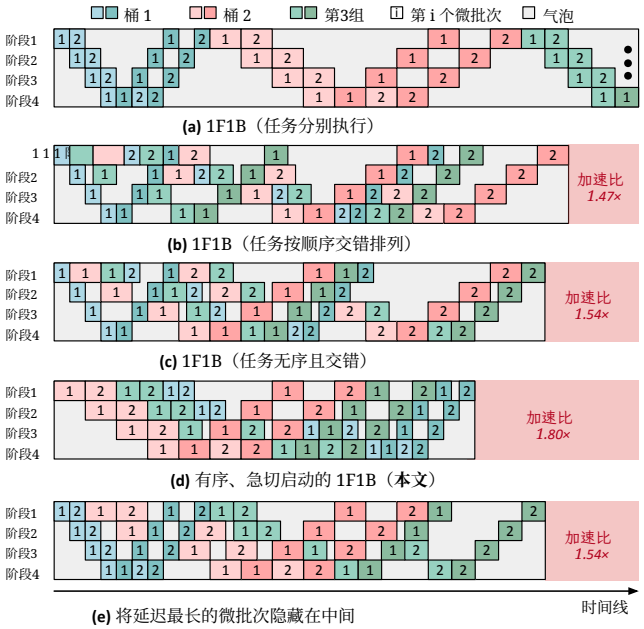


图22：各种多任务 1F1B 流水线调度。

4× 在 GPipe 中 [49] 和 Alpa 中 [93]）。此外，在我们的多任务骨干网络共享场景中，hTask 桶的数量通常也远大于S。例如，在第5.3节的“内存效率分析”中，4 张 A40 GPU 可在单个 LLaMA7B 骨干网络上通过 4-GPU流水线同时容纳 32 个以上的 PEFT任务。也就是说，当P 可设置为最多 32 个，此时$S = 4$。因此，我们给出如下定理：

定理1 在多任务*PEFT*场景中，稳态阶段的延迟 *(T^{steady})* 通常主导端到端流水线延迟。

因此，为了证明我们提出的流水线模板接近最优，**我们只需证明T^{steady}被最小化为$2C\sum_{i=1}^{P} t_i$，即最后阶段不存在流水线内部气泡，如图22(d)**所示。

为了证明这一结论，我们注意到，模板生成需要按延迟降序对 hTask 桶 $\mathcal{G} = \{\mathcal{G}_1, \mathcal{G}2, \ldots, \mathcal{G}_P\}$进行排序（同一桶中的微批次连续排列），同时主张尽可能积极地启动更多微批次。在此背景下，我们给出第三条引理如下：

引理 3 在最后阶段中$\mathcal{G}j$的最后一个微批次完成反向传播之前，倒数第二个阶段中$\mathcal{G}j{+}1$的第一个微批次的前向传播始终已准备就绪。

如图 22(d) 所示，该引理的依据是：(1) 前向和反向传播的延迟相同，(2) $\mathcal{G}j{+}1$ 的阶段延迟始终短于$\mathcal{G}j$，并且 (3) 1F1B 流水线始终优先处理已就绪的反向传播。基于该引理，我们给出以下定理：

定理 2 在我们提出的流水线模板中，一旦第一个微批次在最后阶段开始前向传播，最后阶段就会“保持忙碌”，直至最后一个微批次完成反向传播。

因此，**我们已经完成了对所提出的流水线模板的近似最优性证明**。需要注意的是，最优性证明建立在以下前提之上：对于我们的微批次即时启动方案，设备内存始终充足，因为在多任务参数高效微调场景中，内存占用大幅降低（§5.3）。在实践中，我们在考虑内存限制的情况下实现流水线模板生成，即如果内存成本模型报告至少一个阶段会出现内存溢出问题，就延迟启动微批次。

在图 22(e) 中，我们进一步展示了将最长的微批次隐藏在中间（而非按降序排序）的效果。如图所示，尽管T_{warmup} 和T_{drain}有所降低，但这种修改破坏了定理 2，因此导致端到端流水线延迟变差。