arXiv:2303.04759v1 [cs.LG] 8 Mar 2023

RAF：面向深度学习模型训练的整体编译 ng

Cody Hao Yu
亚马逊网络服务公司

Haozheng Fan
亚马逊网络服务公司

Guangtai Huang
亚马逊网络服务公司

Zhen Jia
亚马逊网络服务公司

Yizhi Liu
亚马逊网络服务公司

Jie Wang
亚马逊网络服务公司

Zach Zheng
亚马逊网络服务公司

Yuan Zhou
亚马逊网络服务公司

Haichen Shen
Scroll Foundation

Junru Shao
OctoML ，*Inc*

Mu Li
亚马逊网络服务公司

Yida Wang
亚马逊网络服务公司

摘要

随着深度学习在现代应用中日益普及，许多深度学习框架应运而生，帮助深度学习从业者快速开发和训练DNN模型。同时，近年来训练大型深度学习模型已成为一种趋势，训练吞吐量和内存占用也变得至关重要。因此，使用编译器优化训练工作负载势在必行，也日益受到关注。然而，现有的深度学习编译器（DLC）主要面向推理，在训练工作负载中并未整合自动微分和自动混合精度等整体优化。

本文介绍了 RAF，一款面向训练的深度学习编译器。与现有的深度学习编译器不同，RAF 接受前向模型，并在内部生成训练图。因此，RAF 能够系统地整合图优化，以提升性能、节省内存并支持分布式训练。此外，为了追赶手工编写的内核库和张量编译器所达到的最先进性能，RAF 提出了算子方言机制，以无缝集成所有可能的内核实现。我们证明，通过内部生成训练图和采用算子方言机制，我们能够进行整体优化，并在 GPU 上针对热门 Transformer模型，实现优于 PyTorch [34]（即时执行和 TorchScript 模式）、XLA [49],和 DeepSpeed [14] 的训练吞吐量，或支持更大的批量大小。

1 引言

近年来，深度学习已广泛应用于现代应用中，涵盖计算机视觉 [14,21,44,45],、自然语言处理 [12,37,46],和语音识别 [7,11,50]等领域。随着深度学习模型及其训练数据集不断增大，高效训练深度学习模型变得愈发重要。如今，深度学习从业者通常依赖 TensorFlow [1] 和 PyTorch [34],等深度学习框架来描述和训练模型。然而，深度学习框架最初并非为高效模型训练而设计。相反，它们旨在提供友好的交互体验，以便于模型设计。在性能方面，框架通常会调用 cuDNN [10] 等手工编写的内核库来执行计算密集型算子。这受限于内核库所能提供的功能，也错失了算子之间进行全局优化的机会（例如算子融合和分解）。

此外，现代深度学习框架的设计也限制了分布式训练系统的发展 [24, 40, 51], , 而这类系统通常构建于这些框架之上。以 DeepSpeed 为例， [40],它是一个构建于 PyTorch 之上的最先进水平（SOTA）分布式系统：尽管 DeepSpeed 实现了采用 ZeRO [39] 内存优化技术的分布式引擎，以支持超大模型训练，但其设备级执行引擎仍是原生 PyTorch 运行时，且不具备图优化。因此，在分布式训练 [1]过程中，DeepSpeed 无法通过充分隐藏设备间通信的延迟来全局优化模型执行。

因此，采用深度学习编译器（DLC）作为深度学习框架的后端引擎，已成为实现高效模型训练的一种颇具前景的方案，因为它们能够系统地优化整个模型图，并为硬件平台生成内核代码。例如，PyTorch 将 TorchScript 集成作为模型推理的编译器后端，最近又扩展了其功能，使其也支持训练工作负载，并通过 nvFuser 得到增强 [36]。不过，TorchScript 本身并不涵盖分布式计算，而是依赖 PyTorch 中的其他组件来实现。XLA [49] 是最著名的处理模型训练的深度学习编译器，一些深度学习框架已采用 XLA 作为其编译引擎。例如，JAX [6]使用 XLA 即时（JIT）编译纯粹且静态组合（PSC）的子程序；PyTorch 也支

[1]PyTorch 提供了“钩子”，开发者可通过它们在每个张量或模块执行前后注入回调函数，因此可以利用钩子预取张量，以掩盖通信延迟。然而，若不分析整个模型图，就难以针对各种模型架构制定最优的预取计划。

持对其跟踪模型使用 XLA [43] 来通过编译实现模型训练。然而，现有的大多数编译器（如 XLA）并非以整体方式设计，无法实现全栈优化。例如，XLA 依赖框架执行部分图级操作，例如自动微分和自动混合精度（AMP）。此外，XLA 无法保证使用最佳可用的内核实现。

我们认为，要实现最佳的深度学习模型训练性能，必须全面掌控从图级到算子级的整个软件栈。本文提出 RAF，这是一种基于编译器的系统，可通过整体优化为深度学习模型训练提供编译支持。我们使用 RAF 展示以下几点。

#1：整体优化。RAF 跟踪来自 [2] PyTorch 等框架的原生深度学习模型 [34]，生成所需的训练图，并将其一路编译为可执行文件，涵盖图操作与优化、算子级内核代码生成以及分布式并行实现。我们将在第 5 节说明整体优化能够带来更好的性能。

#2：三阶段图优化。RAF 首次将三类图级优化整合到同一个编译器栈中，包括图生成（例如通过自动微分生成反向图）、表达式优化（例如常量折叠）以及执行顺序优化（例如内存规划）。相应地，RAF 将图级优化系统地抽象为三个阶段，并为每个阶段选择最合适的中间表示（IR），以简化开发。现有深度学习编译器要么对所有优化遍都使用数据流 [8]，要么都使用 A 范式（ANF）[49] IR，以求简便。我们将说明：1）在各个优化阶段采用合适的 IR 形式能够提高开发效率；2）在数据流形式与 ANF 形式之间转换 IR 时，可以保留语义。

#3：面向内核库和张量编译的可扩展后端。为了利用不同后端的手工编写的内核，RAF 引入了算子方言机制，用于将每个算子（可以是单个算子，也可以是融合计算子图）分派给高性能内核库或张量程序编译器。我们展示了
通过智能调度高质量的手工编写的内核，并持续改进张量编译器，RAF 能够始终赶上最新的最先进水平性能。

本文的贡献总结如下：

- 我们设计并实现了 RAF 训练编译器，通过整体优化，将原始模型一路转换为可执行程序。

- 我们在以下环节中进行全面的图级优化

[2]原始模型是用户编写的模型，不包含任何特定于框架的操作，例如自动微分和自动混合精度。

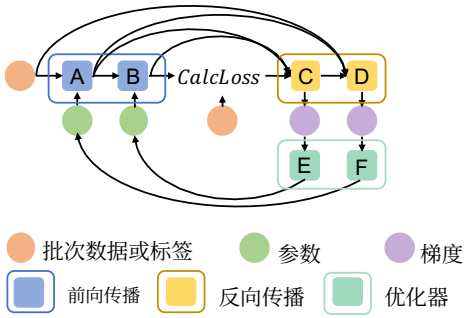


图 1：深度学习训练工作负载。

三个阶段——图生成、表达式优化和执行顺序优化，并在每个阶段采用最合适的中间表示，以提高开发效率。

- 我们引入了一种算子方言机制，以实现智能算子分派，并支持在多个平台上使用第三方内核库和张量编译器。

- 我们使用热门 Transformer 模型对 RAF 进行了评估，结果表明，RAF 在单块 GPU 上的性能优于最先进水平，或在多块 GPU 上实现更大的批量大小。

RAF 的源代码已开源，地址为 https://github.com/awslabs/raf。

2 背景

本节首先在 第 2.1 节分析训练工作负载与推理工作负载之间的差异，随后介绍深度学习编译器（DLC），并探讨支持训练工作负载所面临的挑战，详见 第 2.2 节。

2.1 深度学习训练工作负载

本文聚焦于编译技术，以提升深度学习模型训练的吞吐量并降低内存占用。我们结合图 1说明推理与训练工作负载之间的差异。推理工作负载由深度学习模型定义的前向图以及已训练参数组成。与此同时，训练工作负载由以下部分组成：1）前向图（**A**和**B**）；2）损失计算；3）反向图（**C** 和**D**）；4）优化器（**E** 和**F**）；以及 5）正在训练的可学习参数。这些差异使训练工作负载更难通过深度学习编译器进行优化。

首先，在推理工作负载中，由于参数值经过训练后便固定不变，因此会被视为常量。因此，数据布局（例如行优先或列优先）和数据类型（例如

全精度、半精度或量化）转换等相关优化所带来的性能开销可以通过常量折叠消除。然而，对于训练工作负载中正在训练的参数，情况并非如此。其次，与通常只需要单个或少量批次并侧重于延迟优化的推理工作负载不同，训练工作负载侧重于缩短收敛时间 [26]，这可以通过增大批量大小和提高吞吐量来实现。因此，优化内存占用 [9,13,20,22] 以支持更大的批量大小，以及优化分布式机制 [40,51,53] 以实现更高的吞吐量，是训练工作负载的关键。

2.2 深度学习编译器

与直接将算子映射到手工编写的内核库来执行的深度学习框架不同，Apache TVM [8] 和 XLA [49],等深度学习编译器（DLC）充当后端引擎，将深度学习框架中的模型图转换为特定的中间表示（IR），应用一系列图级和算子级优化，并生成可执行程序。然而，DLC 要实现最优性能或应用于训练工作负载，仍面临挑战：

挑战 1：优化范围。除了前向推理，训练图还包括反向传播和权重更新。现有 DLC 会优化深度学习框架生成的完整训练图。然而，正如我们将在第 3.1.1 节中所说明的，框架生成的训练图可能不适合 DLC，从而导致性能未达最优。

挑战 2：中间张量。与大多采用直线型数据流、因此可立即释放中间张量的推理图不同，在训练工作负载中，前向推理生成的许多中间张量必须实体化并保留，以便在反向传播中进行梯度计算。例如，节点**A** 和**B** 在图 1 中的输出张量在推理时可以很快释放，但在训练时必须一直保留到节点**C** 和**D** 执行。这些生命周期较长的中间张量带来两项挑战：1）内存容量和优化变得至关重要。2）前向图中生成和使用该张量的算子（例如节点**A** 和**B**）不再能够融合，从而大幅减少融合机会。

挑战 3：分布式处理。虽然大多数推理工作负载仍以单个设备为目标，但面向推理的现代深度学习编译器
工作负载缺乏分布式处理支持，无法用于训练工作负载；由于深度学习模型和训练数据集的规模，这些工作负载可能需要横向扩展。

3 系统设计

现有深度学习编译器（DLC）专注于在不改变给定中间表示语义的前提下对其进行优化，从而限制问题

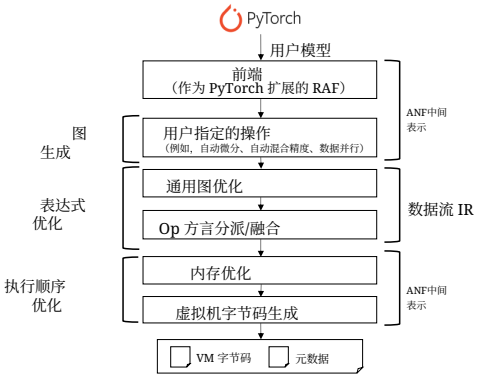


图2：RAF 编译流程。

范围，因此它们只包含表达式优化和执行顺序优化。因此，这些深度学习编译器可以为所有编译器遍选择一种中间表示格式，以简化框架设计。例如，XLA 的所有编译器遍 [49] 都作用于 ANF中间表示。同时，Apache TVM [8]让几乎所有编译器遍都作用于数据流 IR[3]，，且缺乏执行顺序优化。

另一方面，为了纳入整体优化，RAF 接受普通前向传播（即推理）模型的中间表示，并生成用户指定的图（例如反向传播和自动混合精度（AMP） [30] 图）。因此，为了处理新引入的复杂性，我们将 RAF 中的编译器遍分为三个阶段：图生成、表达式优化和执行顺序优化。图2展示了 RAF 编译流程的概览。第一阶段将用户指定的模型操作应用于 ANF 中间表示中的图。例如，它可以添加由自动微分生成的反向传播（第 3.1.1 节）、优化器（例如随机梯度下降 [5] 和 Adam [19]），执行自动类型转换以支持自动混合精度（AMP） [30] 训练（第 3.1.2 节），或执行自动并行（第 3.1.3 节）以支持 ZeRO [39] 数据并行。由于 ANF 中间表示中的表达式通过 `let`绑定到变量，且所有 let 语句构成一个隐含执行顺序的序列，因此可以轻松地在与变量关联的中间表示中编码额外信息，例如自动微分中的链式法则信息，以及用于原地更新的参数别名。

接下来，在优化阶段，我们首先将中间表示从 ANF 转换为数据流形式，并应用一系列中间表示变换遍来优化表达式。这是因为 ANF 中对表达式顺序的强制约束，使开发者更难实现专注于优化表达式而非执行顺序的变换遍，例如算子融合和表达式简化。相比之下，数据流 IR 以数据流图的方式构建模型，表达式作为后续表达式的参数嵌入其中。因此，很容易实现基于模式的

[3]虽然 TVM 也会将数据流 IR 转换为 ANF，但这仅用于虚拟机执行。

和基于规则的数据流 IR 表达式变换。

最后，在执行顺序优化阶段，我们再次将中间表示转换回 ANF，并应用一系列遍次。例如，分布式优化遍次（第 3.1.3 节），包括集体通信算子优化和计算通信重叠，也用于支持超大模型训练。同时，内存优化遍次也会减少峰值内存消耗，以支持更大的批量大小，其中包括内存占用感知的执行顺序调度（第 3.1.4 节）和自动重计算（第 3.1.5 节）。请注意，由于数据流 IR 不会使用变量限定表达式，因此我们会将额外信息编码到 ANF 中，再将其存入数据流 IR 的边或节点属性，并在转换回来时恢复这些信息。

除了图级优化，RAF 还包含算子级优化。具体而言，为了支持可扩展后端，使 kernel 库（例如 cuBLAS 和 CUTLASS [32]）以及张量编译（例如 TVM [8]）都能轻松集成，我们使用方言分派遍来确定每个算子应使用哪个后端。详情见第 3.2.1 节。此外，我们还应用融合遍次（详见第 3.2.2 节），将方言算子融合为闭包，以减少内核调用和算子间通信开销。

本节余下部分将介绍图级和算子级优化中的关键系统组件设计，并分享相关见解。

3.1 整体图级优化

3.1.1 自动微分

在 RAF 中，我们实现了一个自动微分（Autodiff）过程，采用基于闭包的方法进行源代码变换 [35] 以执行求导。Autodiff 会将前向图中的每个算子替换为相应的伴随函数，该函数根据输出的导数计算输入的导数。具体而言，每个伴随函数都会返回原始函数相对于输入的偏导数，以及一组按顺序排列的、相对于输出的偏导数。

自包含自动微分的优势在于伴随函数在数据依赖和算子组合方面具有灵活性。对于数据依赖，伴随函数通常同时接收原始算子的输入

x 和输出 y 作为参数来推导梯度，这会在前向图中与这两个张量都产生数据依赖关系。例如，**D** 在图 1 中的节点需要节点C的输出（**dy**）、节点A的输入（x）和输出（**y**）。然而，对于某些算子，只需x 或 y 就足以计算梯度。以y =双曲正切(x) 为例，梯度可以通过$grad$ $= 1 - 2 \times$双曲正切(x) 或 $grad = 1 - 2 \times y$计算。因此，

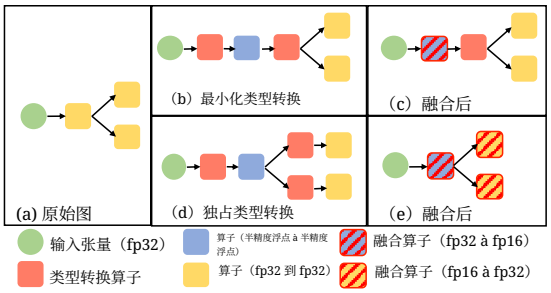


图 3：为自动混合精度插入最小化或专用类型转换算子对融合的影响。

我们只需使用x 或y 来计算梯度，而非两者兼用；这不仅能降低峰值内存，还能增加相应前向传播算子的融合机会。

对于算子组合，直观而言，将一个反向算子分解为一系列小算子，能够为算子融合提供更多机会。例如，双曲正切_$dx(y) = 1 - 2 \times y$ 可以分解为先乘后减，而不是封装成一个 tanh_dx 算子。然而，即使牺牲融合机会，内核库在某些反向算子（如 Conv2D_dx）上的表现也可能优于编译器。借助灵活的伴随函数，RAF 可以针对每种情况选择最佳方案。

3.1.2 自动混合精度

训练大型模型时，通常会使用半精度来支持更大的批量大小，但精度损失仍可能导致训练时间性能低下。因此，现代深度学习框架提供了一种折中方案：大部分计算采用半精度，而数值敏感的计算则保持全精度。这称为自动混合精度（AMP） [30]。然而，算子能否以半精度计算，不仅取决于其算术表达式，还取决于其实现。因此，直接使用由框架处理的 AMP 计算图可能会导致 DLC 出现正确性问题，正如我们将在第 5 节中展示的那样。

在 RAF 中，我们定义了自己的数值敏感算子列表，并利用自有的 AutoCast 遍为图插入类型转换算子，以进行混合精度训练。除了保证数值正确性外，AutoCast 还会优化性能。具体而言，尽管最小化插入的类型转换算子数量很直接，但我们发现这并不总是最佳策略。以图 3为例，如果 AutoCast 像图 3b 那样将类型转换算子数量降至最低，那么共享的类型转换就无法融合到任何一个使用它的算子中，如图 3c 所示。相比之下，图 3d 中的专用类型转换算子可以融合，如图 3e 所示。由于融合后的类型转换操作几乎没有性能开销，因此可以相应地获得更好的端到端性能。RAF 中的 AutoCast 遍会考虑

算子融合机会，并为能够融合其逐元素前驱和后继算子的使用方插入专用类型转换算子。

3.1.3 分布式优化

RAF 目前支持数据并行[4]，以实现分布式训练。其核心思想受 ZeRO 启发 [39]：将优化器状态和梯度分区，在扩展规模时减少单个设备上的内存占用。不过，与 [40] DeepSpeed 在 PyTorch 之上手动实现 ZeRO 不同，RAF 会自动应用多个变换遍，分析单设备的 IR 图，并插入所需的集合操作，使其能够扩展到多个设备。

与手动实现（即 DeepSpeed）相比，我们基于编译器的解决方案具有以下优势。首先，所实现的变换遍通常适用于所有模型和优化器，因此在出现新模型和优化器时，能显著减少工程工作量。其次，加入更多高级优化也很容易。例如，在为分布式训练插入集合操作时，我们利用 CUDA 流来重叠计算与通信，从而隐藏设备间通信开销。此外，我们还对集合操作进行水平融合，以提高带宽利用率并降低通信开销。正如我们将在 第 5 节中展示的，通过自动应用这些优化，RAF 在一个包含 15 亿参数、运行于 8 个 NVIDIA A100 GPU 上的自定义编码器模型上，性能比 DeepSpeed 高出 $\sim 14\%$。

3.1.4 算子调度

将数据流 IR 转换为 ANF 时，从正确性角度看，任何有效的拓扑序都是有效的执行顺序。然而，某些执行顺序可能会导致峰值内存消耗大幅增加，甚至超过动态随机存取存储器的容量。例如，如果过早执行生成大型张量的算子，就必须让这个大型张量在内存中占用空间很长时间。

为降低某个执行顺序的峰值内存，算子调度遍会分析每个算子的内存占用，以确定其执行时机。形式化地，设该算子产生的张量总大小为p，该算子执行后可释放的张量总大小为c，当$p - c < 0$时，算子调度器会将该算子提前执行；当$p - c > 0$时，则会将其延后执行。因此，降低内存消耗的算子会尽可能早地执行，反之亦然。

[4]模型并行和流水线并行仍是未来的研究方向 .

3.1.5 重新物化

如果算子调度后峰值内存仍超过容量，另一种广泛使用且能使模型适配硬件的优化方法是重新物化 [9],：在内存不足之前释放一些中间张量，并在之后需要时重新生成它们。

PyTorch 等深度学习框架依赖模型开发者来确定前向传播中的检查点（即不会重新生成的中间张量），并在反向传播时重放两个检查点之间的所有算子。然而，这种方法有两个缺点。首先，模型开发者必须手动识别检查点。其次，一旦启用检查点，即使内存预算充足，两个检查点之间的每个算子也都必须重放，从而造成显著的性能开销。

另一方面，RAF 展示了基于编译的重新物化更加实用且灵活。具体而言，RAF 中的重新物化遍会遍历 ANF 中间表示，并通过存活性分析分析内存占用。为尽量减少重计算开销，当某个执行点的估计内存消耗超过给定预算时，重新物化遍会对当前存活的中间张量应用成本函数，该函数考虑以下因素：1）重计算延迟；2）剩余使用次数；3）大小。随后，它会识别出一个或多个中间张量，并将其存活期拆分为两个区间，其中第二个区间将通过重放相应的算子来初始化。因此，这种方法带来了两项好处。首先，由于这一过程完全自动化，开发者无需向模型中插入检查点。其次，RAF 只会重新物化满足内存预算所必需的张量，因此与手动设置检查点相比，其性能开销得到了控制。

3.2 算子级优化

3.2.1 算子方言

一个平台可能有多种内核实现，但不存在一种库/编译器能够在所有算子上都优于其他实现。例如，根据我们在 NVIDIA V100 GPU 上的评估，尽管 cuDNN 和 TVM 都为 NVIDIA GPU 提供了特定的 CUDA 内核，但在 Conv2D反向传播中，cuDNN 相比 TVM 实现了 1.8× 的加速比；而在 softmax反向传播内核中，TVM 则能够实现最高达 100×的加速比（通过自动调优）。因此，要实现最佳性能，采用灵活的内核选择机制是不可避免的。更糟糕的是，越来越多的后端实现现在引入了算子融合功能，例如 cuDNN v8 [31] 和 CUTLASS [32]，同时支持不同的融合模式，因此，如 [18] 所述，这一问题变得更加具有挑战性。

受 MLIR [23] 引入中间表示方言的启发，我们引入算子方言来解决这一问题。类似

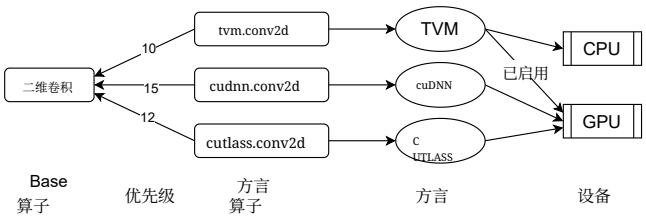


图 4：与某个方言关联的方言算子会以一定的分派优先级注册到基础算子。方言仅在特定设备上启用。

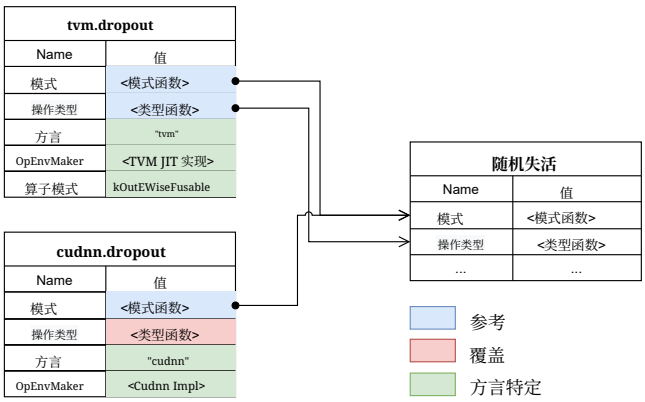


图5：基础算子和方言算子的属性表。基础算子的属性可以被继承（蓝色）、覆盖（红色）和自定义（绿色）。

于 MLIR 中的方言语言，我们将每种后端实现注册为中间表示中的一个新方言，例如“cudnn”“tvm”等，并允许每种方言自定义算子属性。与将所有后端都视为方言的 MLIR 不同，RAF 自带一组预定义算子，作为基础算子。通过共享基础算子的部分算子属性（例如类型关系函数），可以减少添加新方言算子所需的工作。此外，用户无需像 MLIR 那样编写每一对方言之间的转换代码。尽管如此，它仍然提供了灵活性，允许方言算子通过属性表自定义语义，如图5所示；更重要的是，它支持为每种方言设置不同的融合模式（详见第 3.2.2 节）。

在图 4中，方言算子会以一定的分派优先级注册到基础算子。开发者可以根据以往经验预先定义该优先级，也可以通过性能分析结果得出。方言算子还与某个方言相关联。例如，`tvm.conv2d` 属于方言`tvm`，并以优先级 10 注册到基础算子`conv2d`。此外，方言也可以仅在特定设备上启用。例如，方言`tvm` 可用于 CPU 和 GPU；而方言`cudnn` 和`cutlass` 仅适用于 GPU。如果在 RAF 中禁用了某个方言，则该方言中的所有方言算子都不会被纳入中间表示，也不会在运行时进行分派。

每个算子（包括基础算子和方言算子）都有各自的属性表。基础算子的属性表包含其方言算子之间共享的通用属性，例如参数模式、类型关系函数等，如图5所示。方言算子的大多数属性可以继承自基础算子，因此可显著减少开发人员添加新方言算子所需的工作量（图5中的蓝色项目）。另一方面，方言算子可以在基础算子的基础上覆盖某些属性值，并拥有自身特有的属性（分别为红色和绿色项目）。例如，在图5中，`cudnn.dropout` 具有不同的类型关系函数，因为 cuDNN 实现需要为状态预留额外空间。

3.2.2 算子融合

算子融合将多个算子组合为单个内核，并且通过避免中间结果在缓存和主存之间的数据传输，可大幅减少内核执行时间。它对于 GPU 和专用加速器尤其有效。如前所述，多个后端都具备算子融合能力，但它们的融合模式彼此不同。例如，基于模板的库 CUTLASS [32] 能够融合通用矩阵乘法（GEMM）及其后续的逐元素激活函数。另一方面，作为深度学习编译器，Apache TVM 根据其融合规则融合算子。尽管 TVM 的性能可能无法优于手工编写的内核，但在某些情况下，它能够以厂商库未覆盖的方式有效融合算子，从而获得更好的端到端性能，如 [8]所述。在 RAF 中，我们的目标是制定一个可受益于所有后端的融合方案。例如，在这一特定情况下，当模式匹配时，我们倾向于选择 CUTLASS 方案，因为它具有手工优化的高性能；其余算子则可由 TVM 融合并优化。

为了适应所有后端不同级别的融合能力，RAF 将算子融合优化分为两类：基于模式的融合 和基于规则的融-合。例如，CUTLASS 和 cuDNN 后端的融合属于基于模式的类别，因为它们仅支持有限的融合模式（例如，通用矩阵乘法（GEMM）后接逐元素激活）。另一方面，TVM 属于基于规则的融合，因为它基于融合规则 [8]执行通用算子融合。

对于基于模式的融合，RAF 为开发者提供了注册融合模式的接口。得益于 第 3.2.1 节中描述的算子方言，我们可以为每个模式指定特定方言，以指明融合算子使用哪个后端进行代码生成。每个模式

模式 / 规则	方言	优先级
二维卷积 +偏置 + ReLU（半精度浮点）	CUTLASS	15
二维卷积 + 偏置 + ReLU	cuDNN (>=8)	12
二维卷积	cuDNN	10
二维卷积 + 偏置 + ReLU	CUTLASS	8
TVM 融合规则	tvm	5

表 1：融合模式和规则的优先级。

```
%1 = raf.op.conv2d(%x: f16, %w: f16);
%2 = raf.op.relu(%1);
%3 = raf.op.add(%2, %y);
%4 = raf.op.multiply(%3, %z);
```

(a) 融合前。

```
%1 = 函数(%p0, %p1, 方言="cutlass") {
  %2 = raf.op.cutlass.二维卷积(%p0, %p1);
  %3 = raf.op.cutlass.ReLU(%2)
};
%4 = %1(%x: f16, %w: f16);
%5 = raf.op.add(%4, %y);
%6 = raf.op.multiply(%5, %z);
```

(b) 应用 CUTLASS 融合模式后。

```
%1 = 函数(%p0, %p1, 方言="cutlass") {%2 = raf.op.cutlass.conv2d(%p0, %p1);%3 = raf.op.cutlass.relu(%2)};%4 = %1(%x: f16, %w: f16);%5 = 函数(%p0, %p1, %p2, 方言="tvm", 原语=1) {%6 = raf.op.tvm.add(%p0, %p1);%7 = raf.op.tvm.multiply(%6, %p2)};%8 = %5(%4, %y, %z);
```

(c) 应用 TVM 融合规则后。

图 6：用于说明融合的中间表示示例。

每种融合模式都关联一个优先级，用于决定这些融合模式的应用顺序。我们也为基于规则的融合设置优先级，算子融合通道会按照优先级从高到低，一次应用一个融合模式或融合规则。模式匹配的顺序会产生不同的融合图，并可能影响性能。虽然目前的优先级由开发人员根据启发式方法设定，但我们计划根据性能分析结果推导优先级，并基于融合模式和融合规则搜索所有可能的图划分方案，以找到最优的融合计划。

我们通过图 6 中的示例，说明算子融合如何发挥作用，以及如何应用融合模式和规则。表 1 给出了几个与二维卷积相关的融合模式示例，以及带有优先级的 TVM 融合规则。首先，融合表中适用于 float16 数据类型的 CUTLASS 融合模式具有最高优先级。优化通道会在程序中找到一个匹配项，然后将匹配的子图替换为一个方言为`cutlass`并包含函数调用的融合算子。随后，中间表示从图 6a 转换为图 6b。之后，在转换后的程序中，除 TVM 融合规则外，找不到其他适用的模式。图 6c 展示了融合优化后的最终程序。

4 实现

除了上一节介绍的编译器设计外，本节还将介绍 RAF 的前端和运行时，使其成为一个端到端的深度学习系统。

4.1 RAF 作为 PyTorch 扩展

我们观察到，用户倾向于使用 PyTorch 提供的交互式界面 [34] 来编写和调整用于训练的深度学习模型。因此，我们希望将 RAF 与 PyTorch 的用户界面和编程范式集成起来。然而，这颇具挑战，因为 PyTorch 采用命令式风格的编程范式，以提供最佳易用性，并通过动态解释和执行模型图中的每个算子来运行模型。另一方面，RAF 等基于编译的框架则旨在一次性编译整个模型图，以便在执行前应用图级优化，从而提升性能。这种编程范式与 PyTorch 采用的范式大相径庭，也使用户难以打印中间结果或修改模型。因此，要使用 RAF 运行现有模型或训练脚本，需要付出大量工作。

为了尽量减少用户在训练任务中启用 RAF 的工作量，RAF 将 Lazy Tensor Core (LTC) [43] 作为其命令式前端。该前端被设计为 PyTorch 的扩展，可以轻松插入官方发布的 PyTorch 软件包，而无需修改上游 PyTorch 代码库。LTC 的核心思想是在 PyTorch 中注册一个新的后端。与其他后端中的算子在执行时生成输出张量不同，LTC 中的算子会生成一个惰性张量，它只是一个连接其输入惰性张量的符号。这样一来，当用户以命令式方式执行 PyTorch 模型时，实际上不会执行任何操作，而只会对算子进行跟踪和记录，从而获得完整的模型图。实际的编译和执行会在调用 LTC 同步 API 时进行。

图 7 展示了如何通过 LTC 启用 RAF，仅需修改 PyTorch 模型中的 2～3 行代码。具体来说，在第 4 行，用户将模型从 CPU 移动到 惰性 设备，该设备在导入 RAF 时注册为由 RAF 负责后端。然后，需要在第 7 行调用 RAF 专用 API 来封装前向模型。这有助于我们捕获包含控制流在内的完整模型结构。完成上述修改后，模型便处于惰性模式。在第 9～12 行期间，这意味着

```
1 导入 torch，raf2从 torch.optim 导入 Adam3模型 = # 初始化一个 torch 模型 4 模型 =
模型.to("lazy") 5 模型.train () 6 优化器 = Adam(模型. 参数 ()) 7 模型 =raf.jit.script(模型) 8
对于数据中的 inp、label_loader (): 9 输出 = 模型(inp.to("lazy")) 10 损失 = 计算_损失 (
输出，label.to("lazy")) 11 输出.backward () 12 优化器.step () 13 raf.mark_step() 14
```

图 7：RAF 前端编程模型的代码片段。

这些算子在访问时不会执行，而只是以符号的形式暂存为 RAF IR，形成一个图。随后，收集到的 IR 图会经过优化和编译，并在用户于 L13 创建的同步点执行。

4.2 运行时虚拟机

模型图的 IR 优化完成后，我们会将其编译为虚拟机（VM）字节码，由 RAF VM 运行时执行。我们选择虚拟机而非图解释器，原因如下。首先，虚拟机可扩展以支持非 DAG 执行（例如包含控制流的模型）。其次，它能够动态管理动态形状所需的内存。

VM 执行流程如下所述。在启动训练循环之前，RAF 会初始化一个 VM 实例，以加载字节码、初始化寄存器和内存池等。每次训练迭代中，VM 都会解释字节码，以操作寄存器、管理内存池并调用相应的内核。具体而言，当首次访问某个算子或闭包（即融合算子）时，VM 会调用一个特定的后端，该后端由 IR 变换期间的方言分派遍决定，并执行即时编译（JIT）以生成并缓存可执行二进制文件。我们将在第 5 节中展示，由于 JIT 编译只会产生一次性开销，因此不会影响用户体验，因为训练模型通常需要数千到数百万次迭代。

5 评估

在本节中，我们使用 Amazon EC2 GPU 实例评估 RAF 在热门 Transformer模型上的表现，包括 BERT [12],RoBERTa [25] 和 GPT-2 [37] 。这些模型来自 Huggingface transformer library [48], 未经修改。

我们选择 PyTorch 1.12.1 和 CUDA 11.3 下的即时执行模式与 TorchScript 作为单设备训练的基线。我们还选择 DeepSpeed 0.5.10 用于分布式训练评估。此外，我们还将 RAF 与 PyTorch/XLA 1.12+8baddab（于 2022年05/09 发布）进行评估对比，因为它同样利用 Lazy Tensor Core (LTC) [43] 将 PyTorch 模型转换为 XLA 以进行编译。需要注意的是，由于 JAX [6] 不支持自动混合精度（AMP），且其底层编译引擎也是 XLA，因此我们在评估中不使用 JAX，而是利用 PyTorch/XLA 的结果进行评估和分析。因此，我们可以清楚地说明：1）编译技术有助于提高训练效率；2）自包含的反向图和可扩展后端可能是实现更佳性能的关键。

5.1 吞吐量评估

我们首先评估处理一个小批次的吞吐量，以回答以下问题：

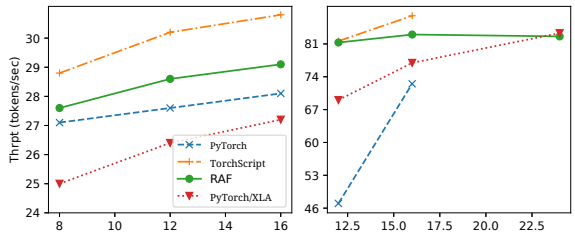
1. RAF 相较于 PyTorch 和 PyTorch/XLA 能实现多大的加速比?
2. 整体编译优化如何帮助实现更高的吞吐量?

我们在一块配备 16 GB 设备端动态随机存取存储器的英伟达 V100 GPU 上进行此评估。我们使用 WikiText-2 [27] 作为训练数据集。模型及对应的分词器来自 Huggingface [48] , 序列长度设为 512。我们使用 PyTorch 实现的 Adam 优化器 [19] , 学习率为 1e-5，eps 为 1e-6。

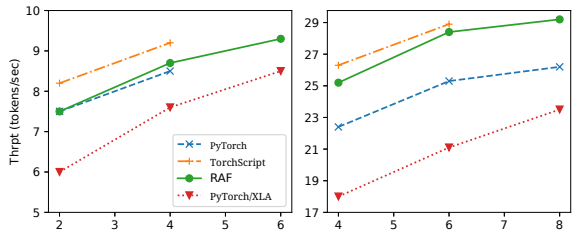
图 8 展示了[5] 所有模型在不同批量大小下的吞吐量。对于每个模型，我们评估了全精度训练（左）和 float16 自动混合精度（AMP）[6] （右）时的吞吐量。可以看到，RAF 相比 PyTorch 即时执行模式支持更大的批量大小，并实现更高的吞吐量。我们没有在 PyTorch 模型中启用用户自定义的梯度检查点，因为即使内存充足，它也会持续带来 ∼2× 延迟开销。

与此同时，我们观察到，在禁用用户自定义的梯度检查点时，无论是在 eager 模式还是 torchscript 模式下，RAF 支持的批量大小都比 PyTorch 更大。这是因为 RAF 会分析模型的内存占用，并自动调用重计算（第 3.1.5 节），以释放前向传播中的一些中间张量，并在反向传播时重新计算它们。由于 RAF 只重新计算最少数量的张量，因此在大多数情况下，延迟开销较小，我们仍能观察到吞吐量提升，例如 BERT-large、RoBERTa-base 和 RoBERTa-large。另一方面，我们还观察到

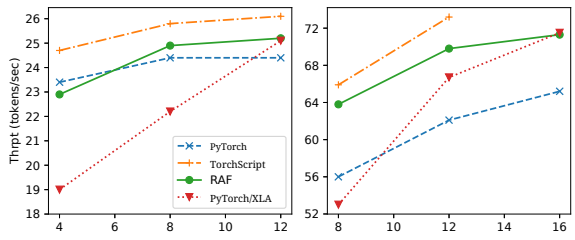
[5]一次性编译开销未计入其中，相关讨论见第 5.2 节。[6]自动混合精度（AMP）能够比半精度实现更好的收敛，因此，当模型可以在单个设备上运行时，使用 AMP 训练模型更为可取。



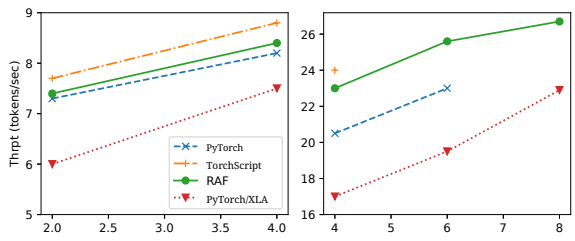
(a) BERT基础版（未区分大小写）



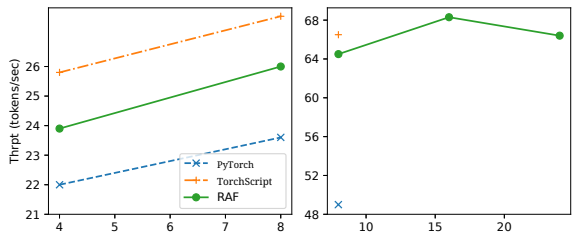
(b) BERT-large-uncased



(c) RoBERTa-base



(d) RoBERTa-large



(e) GPT-2

图 8：吞吐量（纵轴）与批量大小（横轴）。缺失的数据点表示发生了内存溢出。由于 GPT-2 在 PyTorch/XLA 上的结果不正确，因此忽略了这些结果。

对于 BERT-base AMP 和 GPT-2 等少数模型，随着批量大小达到最大值，吞吐量会下降。这是因为在较大的批量大小下，这些模型的峰值内存占用远超 GPU 动态随机存取存储器的容量（16 GB）。为了满足内存约束，RAF 必须重新计算更多张量，因此随着批量大小增大，吞吐量提升幅度会有所减小。

此外，我们可以看到，RAF 在大型模型上优于 PyTorch/XLA（BERT-large 有 3.4 亿个参数，RoBERTa-large 有 3.55 亿个参数）。这再次表明，掌控模型训练图的生成与执行全过程能够获得更好的性能。另一方面，我们忽略了 GPT-2 在 PyTorch/XLA 上的结果，如图 8 所示，因为其未能生成正确结果，这一点将在下一小节中验证。

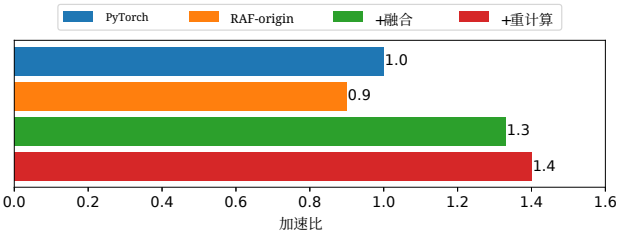


图9：RAF 与 PyTorch 在 BERT-large-uncased 上的消融研究。

为了更好地了解算子融合和内存优化对训练吞吐量的贡献，我们使用 BERT-large 模型进行了消融研究，结果见图9。所有实验均采用能够实现最佳吞吐量的批量大小。可以看出，由于存在各种开销（例如内核启动、内存分配等），RAF-origin 在未进行任何优化时甚至不如 PyTorch。通过算子融合消除这些开销后，RAF 实现了 30% 的加速比。此外，将批量大小从 4 增加到 6 会触发重新物化，且开销极小（重计算 181 个算子的延迟仅增加 2.5%），由于 GPU 利用率更高，吞吐量因此又提升了 10%。

5.2 训练时间性能评估

在本次评估中，我们尝试回答两个问题：

1. RAF 能否得出数值正确结果，并具备与最先进的方法（即 PyTorch）相同的收敛能力？ 2. 训练大型模型时，如何降低一次性编译开销？

我们设计了以下实验来回答这些问题：尽管 MLPerf [26] 建议进行完整的

从头预训练，并将达到目标指标所需的时间作为训练时间性能进行衡量，但这在多设备多节点平台上耗时很长，不适用于我们的评估。为节省时间，我们基于 Hugging Face transformers 库中的预训练模型进行继续预训练。具体来说，我们首先使用原生 PyTorch 训练模型 10 个训练轮次，并将 PyTorch 在第 10 个训练轮次结束时的最终损失用作目标损失，以此训练 RAF 和 PyTorch/XLA。换言之，我们训练 RAF 和 PyTorch/XLA 必要的训练轮次，直到它们达到 PyTorch 在第 10 个训练轮次时取得的损失。请注意，对于所有训练任务，我们都使用相同的随机种子和超参数，以进行公平比较。这是合理的，因为 RAF 和 PyTorch/XLA 都旨在无缝支持 PyTorch 模型和编程范式。

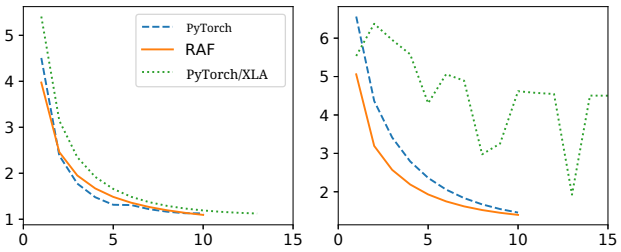


图 10：使用自动混合精度继续预训练 BERT-large（左）和 GPT-2（右）时，损失（纵轴）与训练轮次（横轴）的关系。

三个框架的损失变化趋势如图 10所示。请注意，由于 RAF 执行整体优化，PyTorch 与 RAF 中包含反向传播、优化器和自动混合精度的中间表示并不完全相同，因此收敛趋势有所不同。然而，如图所示，RAF 与 PyTorch 达到目标损失所需的训练轮次相同（BERT-large 为 1.128，GPT-2 为 1.458），这证明 RAF 保持了数值正确性。

另一方面，从图 10 可以看出，尽管 PyTorch/XLA 在 BERT-large 上达到了与 PyTorch 相同的目标损失，但在 GPT-2 上未能做到，且损失随时间波动。深度学习从业者通常会调整学习率和动量等超参数来避免这一问题；但由于我们是在已训练参数的基础上继续预训练，并且 PyTorch 和 RAF 都能呈现平稳的损失趋势，因此我们认为 PyTorch/XLA 应该能够使用相同的超参数训练这些模型。因此，PyTorch/XLA 很可能在优化 GPT-2 时未能保持数值正确性。

同时，在表2中，我们报告了训练时间以及模型设置时间（即一次性编译）的分钟数。尽管与 PyTorch 相比，RAF 存在一次性编译开销，但仍能在更短的时间内达到目标损失。具体而言，给定首个 epoch 的耗时t_0，（其中包含编译开销）以及平均其余训练轮次的已用时间t_a，我们可以估算总计

表2：持续预训练的训练时间。数值为训练时间（模型设置时间），单位为分钟。

框架	BERT-Large	GPT-2
RAF	27.9 (6.2)	11.9 (4.4)
PyTorch	29.8 (∼0)	12.6 (∼0)
PyTorch/XLA	40 (5.3)	不适用 (2.3)

在N个训练轮次后的训练时间：$T = t_0 + (N-1) \times t_a$。鉴于在 PyTorch 和 RAF 上训练 BERT-large 与 GPT2 一个训练轮次的平均已用时间分别为 178 秒对 130 秒和 75 秒对 57 秒，当N 分别大于 8 和 9 时，RAF 的训练时间性能优于 PyTorch。因此，对于大型语言模型而言，RAF 能够提供优于 PyTorch 的训练时间性能，因为它们的预训练需要更多训练轮次才能收敛。

5.3 数据并行评估

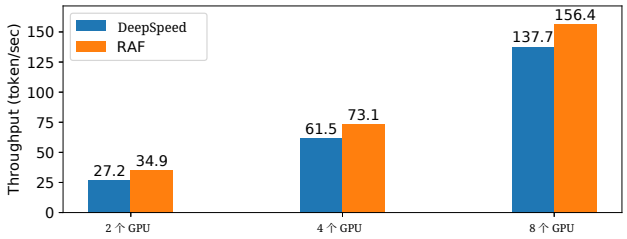


图11：在多块GPU上训练拥有15亿参数的定制语言模型时的吞吐量对比。

我们使用 图11 来展示数据并行下的大模型训练。基线是基于 PyTorch 的 DeepSpeed [40][7],ZeRO [39] 实现。在这项实验中，我们评估了在配备 8 块英伟达 A100 GPU（每块 GPU 配有 40 GB 动态随机存取存储器）的 Amazon EC2 p4d实例上训练一个专有的、拥有 15 亿参数的定制 Transformer模型时的吞吐量。如图所示，得益于 ∼ 14第 3.1.3 节所述的优化，RAF 的吞吐量提升了 %第 3.1.3 节。

为了进一步说明第3.1.3节中提到的优化，我们对上述实验进行了消融研究，如图12所示。此外，图13展示了每次迭代执行的详细分解。如图所示，通过使用 CUDA流重叠计算与通信，RAF能够达到与 DeepSpeed 手动优化相当的吞吐量。此外，从图13可以看出，重叠中的通信由大量集合操作构成，因此产生了较长的延迟和同步开销。因此，在

[7]PyTorch/XLA 对数据并行的支持尚不稳定，因此我们在此次评估中略过了这一点。

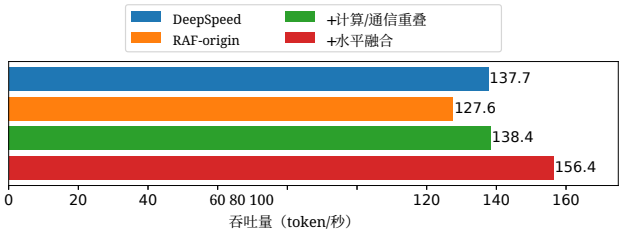


图12：RAF 对 DeepSpeed 在多块GPU上的消融研究。

启用水平融合以聚合全收集和归约散射等集合操作后，融合的性能分析结果显示，通信延迟显著降低，并且可以被完全隐藏。这表明，在框架层，自动编译器解决方案能够达到与手动优化相同或更好的性能。

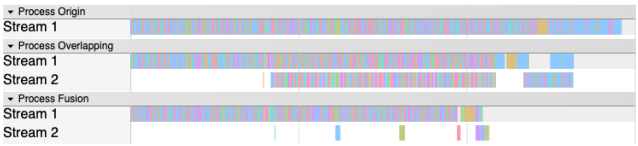


图13：使用包含15亿个参数的自定义模型时，RAF 的算子级性能分析（多块GPU）。

6 相关工作

深度学习编译器XLA [49] 是近年来最引人注目的深度学习编译器。TensorFlow 等深度学习框架 [1] 和 JAX [6] 采用 XLA 作为其编译器后端。与此同时，近期的一项工作 Lazy Tensor [43], 提出了惰性张量核心，用于弥合 PyTorch 与 XLA 之间的差距。尽管该工作的重点 [43] 是 Google TPU，但其开源框架 PyTorch/XLA 也支持 GPU 后端。与 RAF 相比，XLA 依赖框架生成的训练图，因此牺牲了图级灵活性。XLA 对所有优化都采用 A-范式（ANF）中间表示，并将算子分派至内核库（例如 cuDNN [10]）或代码生成。因此，实现某些优化或接入新兴的手工编写的内核十分繁琐。

此外，还有一些现有的编译器可用于优化深度学习工作负载。Halide [38]提出了一种领域专用语言和编译器，最初用于图像处理流水线。随后，它被应用于深度学习工作负载 [2]。TVM [8],是一个开源的深度学习编译器，与 Halide 有许多相似之处，但更专注于深度学习，并采用一种低级中间表示，能够明确表达内存布局、并行化模式、局部性和硬件原语等。Tiramisu [3],是一种面向稠密和稀疏内核的多面体编译器，借助调度语言执行循环优化和数据布局变换。Astra [42]专注于矩阵乘法等计算密集型内核；而将其他工作（如逐元素算子融合）交由框架完成。

分布式训练 以分布式方式训练大型模型时，有几种并行策略——数据并行和模型并行。对于数据并行，ZeRO [39] 是最先进的工作之一，它减少了优化器状态、梯度和参数中重复存储的张量。借助 ZeRO 内存优化，DeepSpeed [40]成功训练了一个拥有 1000 亿个参数的 Transformer 模型。MiCS [51] 进一步改进了 DeepSpeed，通过优化数据通信来适应公有云上的有限带宽。另一方面，以上系统都基于 Transformer 模型和 PyTorch，因此不易扩展和泛化。同时，ZeRO 已在 RAF 中实现为编译器遍，我们的评估结果表明，其效率优于 DeepSpeed。

对于模型并行，Megatron-LM [41] 通过手动划分 Transformer 模型来进行分布式训练。GPipe [15]是一个流水线并行库，它将由一系列层组成的网络划分到多个设备上进行训练。另一方面，现有针对分布式训练编译的工作寥寥无几。Alpa [53] 基于 JAX [6]和 XLA 构建，用于优化算子间与算子内并行。与针对整体优化的 RAF 不同，Alpa 专注于分布式平台上的图级调度，未来可以将这一思路集成到 RAF 中。

深度学习编译器优化虽然针对深度学习工作负载的编译器优化种类繁多，但鉴于融合和重新物化对训练工作负载的重要性，我们主要讨论这两者。算子融合的方法可分为三类：基于模式、基于规则和基于依赖的融合。固定的基于模式的融合限制了可融合的算子组合。NVIDIA CUTLASS [32],、cuDNN [31], 、TensorRT [33],和 Intel oneDNN [16]均采用这种方法。然而，模式需要预先手动定义，因此难以适应新算子。相比之下，TVM [8],、XLA [49],、DNNFusion [29], 和 FusionStitching [54] 采用基于规则的融合，为算子定义一组规则，从而有助于发掘更多融合机会。不过，基于规则的融合有一个缺点：难以灵活地支持多个后端。最后，基于依赖的融合利用通用的依赖分析（例如多面体分析 [4, 28, 47, 52]）实现最强的通用性，但由于涉及 NP 难算法而面临可扩展性问题。在 RAF 中，我们采用基于算子方言的融合机制，将基于模式的融合与基于规则的融合结合起来，兼取两者之长。

重新物化 通过在反向传播阶段重新计算中间张量，降低模型训练的峰值内存。Chen等人 [9] 提出了一种启发式方法，用于训练一个n层

线性网络，其内存成本为 $O(\sqrt{n})$：将网络划分为多个片段，并且在前向传播阶段仅存储各片段的输出。Gruslys 等人 [13] 提出了一种动态规划方法，用于为循环神经网络选择检查点。Kumar等人 [22] 将重新物化建模为一个优化问题，并通过树分解求解；而 Checkmate [17] 则采用整数线性规划（ILP）。DTR [20] 是一种运行时解决方案，它维护张量元数据，并使用精心设计的成本函数，在模型执行期间动态驱逐和重新物化张量。另一方面，DTR 会给运行时带来内存分析开销，而且在缺乏动态性的模型上，性能可能不如静态方法。RAF 采用的方法与 DTR 类似，但在编译时执行。这样，我们便能够进行完整的存活性分析，从而实现比 DTR 更低的开销。

7 结论

本文介绍了 RAF，这是一款用于训练的深度学习编译器。RAF 接受原始模型，并在内部生成训练图，包括自动微分和混合精度，从而实现针对性能、内存和分布式训练的整体优化。RAF 具有算子方言机制，可集成第三方内核库和张量编译器，确保持续跟进最先进水平的内核实现。评估结果表明，在保持数值正确性的同时，RAF 在 GPU 上运行流行的 Transformer 模型时，可实现更高的训练吞吐量或支持更大的批量大小。RAF 的源代码已开源，网址为https://github.com/awslabs/raf。

参考文献

[1] Martín Abadi、Paul Barham、Jianmin Chen、Zhifeng Chen、Andy Davis、Jeffrey Dean、Matthieu Devin、Sanjay Ghemawat、Geoffrey Irving、Michael Isard 等。Tensorflow：一种大规模机器学习系统。载于第 *12* 届 *USENIX* 操作系统设计与实现研讨会（*OSDI 16*），第 265–283 页，2016 年。

[2] Andrew Adams、Karima Ma、Luke Anderson、Riyadh Baghdadi、Tzu-Mao Li、Michaël Gharbi、Benoit Steiner、Steven Johnson、Kayvon Fatahalian、Frédo Durand 等。通过树搜索和随机程序学习优化 Halide。*ACM* 图形学汇刊（*TOG*），38(4):1–12，2019。

[3] Riyadh Baghdadi、Jessica Ray、Malek Ben Ro mdhane、Emanuele Del Sozzo、Abdurrahman Akkas、Yunming Zhang、Patricia Suriana、Shoaib Kamil 和 Saman Amarasinghe。Tiramisu：一种用于表达快速且可移植代码的多面体编译器。在*2019 IEEE/ACM* 国际代码生成与优化研讨会（*CGO*）论文集，第 193–205 页。IEEE，2019 年。

[4] Uday Bondhugula、Albert Hartono、Jagannathan Ramanujam 和 Ponnuswamy Sadayappan。一种实用的自动多面体并行化器和局部性优化器。载于第 *29* 届 *ACM SIGPLAN* 编程语言设计与实现会议论文集，第 101–113 页，2008。

[5]Léon Bottou。利用随机梯度下降进行大规模机器学习。载于*COMP-STAT'2010* 论文集，第 177–186 页。Springer，2010。

[6]James Bradbury、Roy Frostig、Peter Hawkins、Matthew James Johnson、Chris Leary、Dougal Maclaurin、George Necula、Adam Paszke、Jake VanderPlas、Skye Wanderman-Milne 和 Qiao Zhang。JAX：Python+NumPy 程序的可组合变换，2018。

[7] William Chan、Daniel Park、Chris Lee、Yu Zhang、Quoc Le 和 Mohammad Norouzi。SpeechStew：简单地混合所有可用的语音识别数据，训练一个大型神经网络。*arXiv* 预印本 *arXiv:2104.02133*，2021。

[8] Tianqi Chen、Thierry Moreau、Ziheng Jiang、Lianmin Zheng、Eddie Yan、Haichen Shen、Meghan Cowan、Leyuan Wang、Yuwei Hu、Luis Ceze 等。TVM：一种用于深度学习的自动化端到端优化编译器。发表于第 *13* 届 *USENIX* 操作系统设计与实现研讨会（*OSDI 18*），第 578–594 页，2018 年。

[9] Tianqi Chen、Bing Xu、Chiyuan Zhang 和 Carlos Guestrin。以次线性内存成本训练深度网络。*arXiv* 预印本 *arXiv:1604.06174*，2016 年。

[10] Sharan Chetlur、Cliff Woolley、Philippe Vandermersch、Jonathan Cohen、John Tran、Bryan Catanzaro 和 Evan Shelhamer。cuDNN：用于深度学习的高效原语。*arXiv* 预印本 *arXiv:1410.0759*，2014 年。

[11] Yu-An Chung、Yu Zhang、Wei Han、Chung-Cheng Chiu、James Qin、Ruoming Pang 和 Yonghui Wu。W2v-bert：结合对比学习与掩码语言建模的自监督语音预训练。载于*2021* 年 *IEEE* 自动语音识别与理解研讨会（*ASRU*），第 244–250 页。IEEE，2021 年。

[12] 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。

[13] Audrunas Gruslys、Rémi Munos、Ivo Danihelka、Marc Lanctot 和 Alex Graves。面向时间反向传播的高效内存方法。《神经信息处理系统进展》，第 29 卷，2016 年。[14] Kaiming He、Xiangyu Zhang、Shaoqing Ren 和 Jian Sun。用于图像识别的深度残差学习。载于 IEEE 计算机视觉与模式识别会议论文集，第 770–778 页，2016 年。[15] 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。[16] 英特尔。oneAPI 深度神经网络库（oneDNN）。https://github.com/oneapi-src/oneDNN。[在线；访问日期： 2022]。[17] Paras Jain、Ajay Jain、Aniruddha Nrusimha、Amir Gholami、Pieter Abbeel、Joseph Gonzalez、Kurt Keutzer 和 Ion Stoica。Checkmate：通过最优张量重物化突破内存墙。《机器学习与系统会议论文集》，2:497–511，2020 年。[18] Byungsoo Jeon、Sunghyun Park、Peiyuan Liao、Sheng Xu、Tianqi Chen 和 Zhihao Jia。Collage：深度学习后端的自动集成。arXiv 预印本 arXiv:2111.00655，2021 年。[19]Diederik P Kingma 和 Jimmy Ba。Adam：一种随机优化方法。arXiv 预印本 arXiv:1412.6980，2014 年。[20] Marisa Kirisame、Steven Lyubomirsky、Altan Haan、Jennifer Brennan、Mike He、Jared Roesch、Tianqi Chen 和 Zachary Tatlock。动态张量重物化。arXiv 预印本 arXiv:2006.09616，2020 年。[21] Alex Krizhevsky、Ilya Sutskever 和 Geoffrey E Hinton。使用深度卷积神经网络进行 ImageNet 分类。载于《神经信息处理系统进展》，第 1097–1105 页，2012 年。[22] Ravi Kumar、Manish Purohit、Zoya Svitkina、Erik Vee 和 Joshua Wang。面向深度网络的高效重物化。《神经信息处理系统进展》，第 32 卷，2019 年。[23] Chris Lattner 、 Mehdi Amini、Uday Bondhugula、Albert Cohen、Andy Davis、Jacques Pienaar、River Riddle、Tatiana Shpeisman、Nicolas Vasilache 和 Oleksandr Zinenko。MLIR：扩展面向特定领域计算的编译器基础设施。载于 2021 年 IEEE/ACM 国际代码生成与优化研讨会（CGO），第 2–14 页。IEEE，2021 年。

[24] 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。[25] Yinhan Liu、Myle Ott、Naman Goyal、Jingfei Du、Mandar Joshi、Danqi Chen、Omer Levy、Mike Lewis、Luke Zettlemoyer 和 Veselin Stoyanov。《RoBERTa：一种稳健优化的 BERT 预训练方法》。arXiv 预印本 arXiv:1907.11692，2019。[26] Peter Mattson、Christine Cheng、Gregory Diamos、Cody Coleman、Paulius Micikevicius、David Patterson、Hanlin Tang、Gu-Yeon Wei、Peter Bailis、Victor Bittorf 等。MLPerf 训练基准。《机器学习与系统会议论文集》，2：336–349，2020。[27] Stephen Merity、Caiming Xiong、James Bradbury 和 Richard Socher。《指针哨兵混合模型》。arXiv 预印本 arXiv:1609.07843，2016。[28] Ravi Teja Mullapudi、Vinay Vasista 和 Uday Bondhugula。《PolyMage：图像处理流水线的自动优化》。ACM SIGARCH 计算机体系结构新闻，43(1)：429–443，2015。[29] Wei Niu、Jiexiong Guan、Yanzhi Wang、Gagan Agrawal 和 Bin Ren。《DNNFusion：通过先进的算子融合加速深度神经网络执行》。载于第 42 届 ACM SIGPLAN 编程语言设计与实现国际会议论文集，第 883–898 页，2021。[30]英伟达。深度学习的自动混合精度。https://developer.nvidia.com/automatic-mixed-precision。[在线；访问日期2022]。[31]英伟达。cuDNN 8.x.x 版本。https: //docs.nvidia.com/deeplearning/cudnn/release-notes/rel_8.html。 [在线； 2020]。[32] 英伟达。CUTLASS：CUDA 线性代数子程序模板。https://github.com/NVIDIA/ cutlass。 [在线；访问日期 2017]。[33]英伟达。Nvidia TensorRT。https://developer. nvidia.com/tensorrt。 [在线；访问日期 2022]。[34] Adam Paszke、Sam Gross、Francisco Massa、Adam Lerer、James Bradbury、Gregory Chanan、Trevor Killeen、Zeming Lin、Natalia Gimelshein、Luca Antiga 等。《PyTorch：一种命令式风格、高性能的深度学习库》。载于《神经信息处理进展》

处理系统，第 8024 页 –8035，2019。

[35] Barak A Pearlmutter 和 Jeffrey Mark Siskind。《函数式框架中的反向模式自动微分：Lambda，终极反向传播器》。*ACM* 程序设计语言与系统汇刊（*TOPLAS*），30(2):1–36，2008。

[36]PyTorch。https://pytorch.org/blog/introducing-nvfuser-a-deep-learning-compiler-for-pytorch/。[在线； 2022]。

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

[38] Jonathan Ragan-Kelley、Connelly Barnes、Andrew Adams、Sylvain Paris、Frédo Durand 和 Saman Ama-rasinghe。Halide：一种用于优化图像处理流水线中并行性、局部性和重新计算的语言及编译器。*ACM SIGPLAN* 通讯，48(6):519–530，2013。

[39] 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。在 *SC20*：高性能计算、网络、存储与分析国际会议，第 1–16 页。IEEE，2020。

[40] Jeff Rasley、Samyam Rajbhandari、Olatunji Ruwase 和 Yuxiong He。DeepSpeed：系统优化助力训练参数超过 1000 亿的深度学习模型。在 第 *26* 届 *ACM SIGKDD* 知识发现与数据挖掘国际会议论文集，第 3505–3506 页，2020。

[41] 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。

[42] Muthian Sivathanu、Tapan Chugh、Sanjay S Singapuram 和 Lidong Zhou。Astra：利用可预测性优化深度学习。在第二十四届编程语言与操作系统架构支持国际会议论文集，第 909–923 页，2019。

[43] Alex Suhan 、 Davide Libenzi、Ailing Zhang、Parker Schuh、Brennan Saeta、Jie Young Sohn 和 Denys Shabalin。LazyTensor：将即时执行与领域专用编译器相结合。*arXiv* 预印本 *arXiv:2102.13267*，2021。

[44] Christian Szegedy、Wei Liu、Yangqing Jia、Pierre Ser-manet、Scott Reed、Dragomir Anguelov、Dumitru Erhan、Vincent Vanhoucke 和 Andrew Rabinovich。迈向更深层的卷积网络。在 *IEEE* 计算机视觉与模式识别会议论文集，第 1–9 页，2015。

[45] Christian Szegedy、Vincent Vanhoucke、Sergey Ioffe、Jon Shlens 和 Zbigniew Wojna。重新思考用于计算机视觉的 Inception 架构。载于*IEEE* 计算机视觉与模式识别会议论文集，第 2818–2826 页，2016 年。

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

[47] Sven Verdoolaege、Juan Carlos Juega、Albert Cohen、Jose Ignacio Gomez、Christian Tenllado 和 Francky Catthoor。面向 CUDA 的多面体并行代码生成。*ACM Transactions on Architecture and Code Optimization (TACO)*，9(4):1–23，2013。

[48] Thomas Wolf、Lysandre Debut、Victor Sanh、Julien Chaumond、Clement Delangue、Anthony Moi、Pierric Cistac、Tim Rault、Rémi Louf、Morgan Funtowicz 等。Hugging Face Transfor mers：最先进的自然语言处理。*arXiv* 预印本 *arXiv:1910.03771*，2019。

[49] XLA 团队。XLA——TensorFlow 的编译器，2017年3月。

[50] Yu Zhang、James Qin、Daniel S Park、Wei Han、Chung-Cheng Chiu、Ruoming Pang、Quoc V Le 和 Yonghui Wu。推动半监督学习在自动语音识别中的极限。*arXiv* 预印本 *arXiv:2010.10504*，2020。

[51] Zhen Zhang、Shuai Zheng、Yida Wang、Justin Chiu、George Karypis、Trishul Chilimbi、Mu Li 和 Xin Jin。MiCS：在公有云上训练超大型模型的近线性扩展。*arXiv* 预印本 *arXiv:2205.00119*，2022年。

[52] Jie Zhao、Xiong Gao、Ruijie Xia、Zhaochuang Zhang、Deshi Chen、Lei Chen、Renwei Zhang、Zhen Geng、Bin Cheng 和 Xuefeng Jin。Apollo：通过逐层优化实现基于自动分区的算子融合。机器学习与系统会议论文集，4:1–19，2022年。

[53] Lianmin Zheng、Zhuohan Li、Hao Zhang、Yonghao Zhuang、Zhifeng Chen、Yanping Huang、Yida Wang、Yuanzhong Xu、Danyang Zhuo、Joseph E Gonzalez 等。Alpa：为分布式深度学习自动实现算子间和算子内并行。*arXiv* 预印本 *arXiv:2201.12023*，2022年。

[54] Zhen Zheng、Pengzhan Zhao、Guoping Long、Feiwen Zhu、Kai Zhu、Wenyi Zhao、Lansong Diao、Jun Yang、

和 Wei Lin。FusionStitching：提升深度学习工作负载中内存密集型计算的性能。*arXiv* 预印本 *arXiv:2009.10924*，2020。