9. 深度学习编译器 - 分布式并行训练

前面几节介绍了深度学习编译器涉及到的前端、图优化、高效 KERNEL 的实现以及运行时等内容，对推理（Inference）和训练都适用。接下来几节我们分析下训练过程所涉及的一些独特问题，本节主要讨论分布式训练中的并行模式。

9.1 神经网络训练的基本原理

神经网络可以抽象为一个带参数的函数：y = f(x, w)，其中 x 为输入，y 为输出，w 为参数。神经网络训练过程就是根据一系列样本数据 (x_i, y_i) (1 <= i <= N) 学习参数 w 的过程。该过程通常是迭代完成的，即将样本数据分为 若干 batch, 每步迭代给定一个 batch 数据，根据参数 w 当前取值和 batch 数据更新参数的值。例如在当前最广泛使用的梯度下降算法中，根据样本数据计算 w 的梯度。之后根据梯度更新 w， 然后进入下一步迭代。对神经网络而言，W 的梯度是通过反向传播计算得到的。在现有的深度学习框架/编译器框架中，W 的梯度的计算和更新会直接在计算图中体现。上面提到的每步，在文献中通常称作一个 STEP，对全部样本数据进行循环通常称作一个 Pass 或者一个 Epoch。

从系统的角度，神经网络训练主要关注以下几个目标：

1. 系统吞吐，即单位时间内能够处理的样本数。虽然推理也关注吞吐，但是推理同时也关注延时，而训练任务对延时相对不敏感。因此，推理任务 Batch 大小受延时约束，而训练任务可以通过使用较大的 Batch 来提升系统吞吐。
2. 系统能够支持的模型大小。随着以 Transformer 为基础的模型在 NLP、CV 等领域取得 STOA 的效果， 神经网络参数 W 的规模在不断增加，从几百 MB 增加到数百 GB。参数规模的增加为训练系统的设计带来了新的挑战，也使得能够支持的模型大小成为衡量训练系统的一个重要指标。
3. 收敛特性。训练系统最终的目标是获得参数 W 的值，因此能否快速收敛，以及收敛之后模型的效果也是衡量训练系统的另一个重要指标。这点通常和训练系统能够支持的优化算法有关。

简言之，分布式训练系统的目标就是通过多个设备协作，在保证收敛特性的前提下，提升训练系统的吞吐和可以支持的模型大小。本节主要关注通过并行提升训练系统吞吐的方法。这里的多个设备可以是单台服务器内的多张加速卡，也可以是多台服务器。

分布式并行训练有不同的实现方法，从大的方面可以划分为三类：数据并行、模型并行和流水线并行。数据并行和模型并行属于通过并行加快一个 Batch 计算的方法，流水线并行则是通过并行同时计算多个 Batch，从而提升系统的整体吞吐。这三种并行模式也可以相互组合形成混合并行模式。

9.2 数据并行

数据并行的思路是将一个Batch 分成若干小的 MiniBatch，然后交由不同设备计算。每个设备在计算时，首先获取最新的参数，然后对分配到该设备的 MiniBatch 进行前向反向计算，得到参数的梯度。不同 Minibatch 梯度汇总得到该 Batch 对应的梯度，然后更新参数。梯度汇总和更新可以通过不同的方法实现，例如可以使用分布式参数服务器，也可以使用 ALL-REDUCE 等（此处暂不深入，后续再用专门的章节讨论）。更新完成后，进入下一个 STEP 计算。

数据并行是一种典型的 SPMD (Single Program Multiple Data)，好处是不需要对计算图进行任何变化，实现起来比较简单。缺点是每个设备需要存储完整的参数，需要的内存/显存较多。当模型参数增大，单个计算设备存储不下时，数据并行方案就会失效（有一些针对训练过程中需要显存的优化方案，可以缓解这种情况，例如 Materialization 等，此处暂不深入讨论）。除此之外，数据并行需要对各个设备计算得到的梯度进行汇总，当参与设备数较多或者网络带宽较差时，网络会成为数据并行的瓶颈，限制数据并行可以达到的加速比。同时，对于 GPU 等大规模并行计算设备，通常 Batch 越大吞吐越高。因此，采用数据并行方案时，分给单个设备的 MiniBatch 不能太小。换言之，设备数越多，数据并行方案中每个 STEP 需要的 Batch 就越大。随着 Batch 增加，模型的收敛性会发生变化，需要使用一些特殊的优化算法，这也是限制数据并行扩展性的另一因素。

目前 Tensorflow、PyTorch 等框架都支持数据并行，特别是单机多卡下的数据并行，使用起来非常方便。这种模式也是在实际应用中使用最广泛的。

9.3 模型并行

正如名字所暗示的，模型并行的基本思路是对模型进行拆分，每个设备负责一部分模型的计算。如果两个设备负责的部分不存在相互依赖，这两部分就可以并行计算，从而达到加速的效果。除此之外，每个设备只需要存储一部分参数，因此模型并行相当于将参数分布到多个设备存储，因此能够支持更大模型的训练。

不同于数据并行对样本进行划分这一种模式，模型并行可以从多个粒度对模型进行拆分。第一个粒度是 OP 层面，将不同的 OP 分配给不同的设备。第二个维度是 OP 内，将一个 OP 拆分到多个不同的设备。比如，对于 Matmul, 如果参数的规模特别大，单个设备存储不下，则需要将其拆分到多个设备。

在对一个 OP 进行拆分时，可以从 Tensor 的不同维度进行切分。比如对于矩阵乘：C_{[m,n]}=A_{[m,k]} *B_{[k,n]} ，假设 B 为参数。有两种拆分方法，第一，从 Reduce 维度，也即 k 所在维度对 B 进行拆分。第二，从输出隐层的维度，也即 n 所在维度拆分。如果采用第一种，从 k 维度拆分，相当于将矩阵乘法运算中的乘累加拆分到了两个设备，在各自计算完成之后还需要再进行一次 Reduce 操作。综上，模型并行下会有两类跨设备的数据传输，第一类是 OP 的计算结果，当一个设备需要的输入数据是由另一个设备计算得到的，就需要跨设备传输；第二类是一个 OP 由多个设备完成计算，需要跨设备传输各自的计算结果进行汇总。

整体来看，深度学习框架对模型并行的支持还不够完善。Tensorflow 和 PyTorch 支持在代码中显示的将某个 OP 放置在给定的设备上。在运行时会这些指示对计算图进行拆分，本质上是属于 MPMD (Multi Program Multi Data) 的一种并行范式。Mesh-Tensorflow 库类比数据并行的思路，将模型的参数也看作是数据，支持显示的从不同的维度对 Tensor 进行切分。在切分后，不同设备上运行的 Graph 仍然是一样的，只是数据不同。这种方法是通过 SPMD 实现模型并行的

一种方案，且可以同时支持数据并行和模型并行的混合并行模式，但是这种方案不能支持将不同的 OP 交给不同的设备计算，灵活性受限。

模型并行和数据并行可以组合实现混合并行。模型并行和混合并行都需要决定如何划分模型，使模型训练性能最优。在论文 “BEYOND DATA AND MODEL PARALLELISM FOR DEEP NEURAL NETWORKS“，作者对混合并行进行了统一抽象，将不同的并行模式看作从不同的维度对输出 Tensor 进行切分，以此为基础，通过 Tuning 和搜索的方式获取最优切分方式。作者通过 FlexFlow 框架实现了混合并行。上述方案主要是为了对模型训练加速。随着超大模型的快速发展，另外一些研究从支持训练超大模型的角度提出了新的模型并行方案。代表性工作包括 GSPMD: General and Scalable Parallelization for ML Computation Graphs 和 PATHWAYS: ASYNCHRONOUS DISTRIBUTED DATAFLOW FOR ML。

9.4 流水线并行

数据并行和模型并行通常是指对一个 STEP 的计算进行并行，而流水线并行则是对多个 STEP 的计算进行并行。通过将一个 STEP 的计算分为多个 STAGE，不同设备负责不同 STAGE 的计算，然后通过流水线的方式完成多个 STEP 的计算。例如，当设备 d 完成第 n 个 Batch STAGE d 的计算之后，将数据发给设备 d + 1, 由设备 d + 1 计算 STAGE d + 1，与此同时，设备 d 计算第 n+1 个 Batch 的 STAGE d。这种方式一方面可以提高设备计算资源的利用效率。另一方面，每个设备只负责部分模型的计算，因此所需要的存储空间相对较小，通过增加 STAGE 数量，可以支持更大模型的训练。从这个意义上讲，流水线并行也可以看作是一种特殊的模型并行方式。

对于机器学习训练任务，流水线并行面临两方面的问题。第一，训练任务包括前向计算和反向计算，反向计算过程需要使用前向计算的结果。这就意味着流水线不能只是单向流动，当流水线上的最后一个设备完成最后一个 STAGE 的前向计算和反向计算后，需要将梯度传回之前的设备，用于参数梯度的计算和更新，并依次类推直到第一个设备。第二，训练算法和参数更新的问题。通常的训练任务中，上一个 STEP 计算完成后，会根据梯度更新模型参数，在下一个 STEP 的计算中使用最新的参数，且前向计算和反向计算使用的模型参数是一致的。对于流水线并行，由于多个 STEP 同时在多个设备进行计算，每个 STEP 在进行前向计算时所使用的参数和反向计算时的参数不一定相同。上一个 STEP 梯度更新后的参数不能直接用于下一个 STEP 的前向计算。这种困难对训练算法提出了新的挑战。

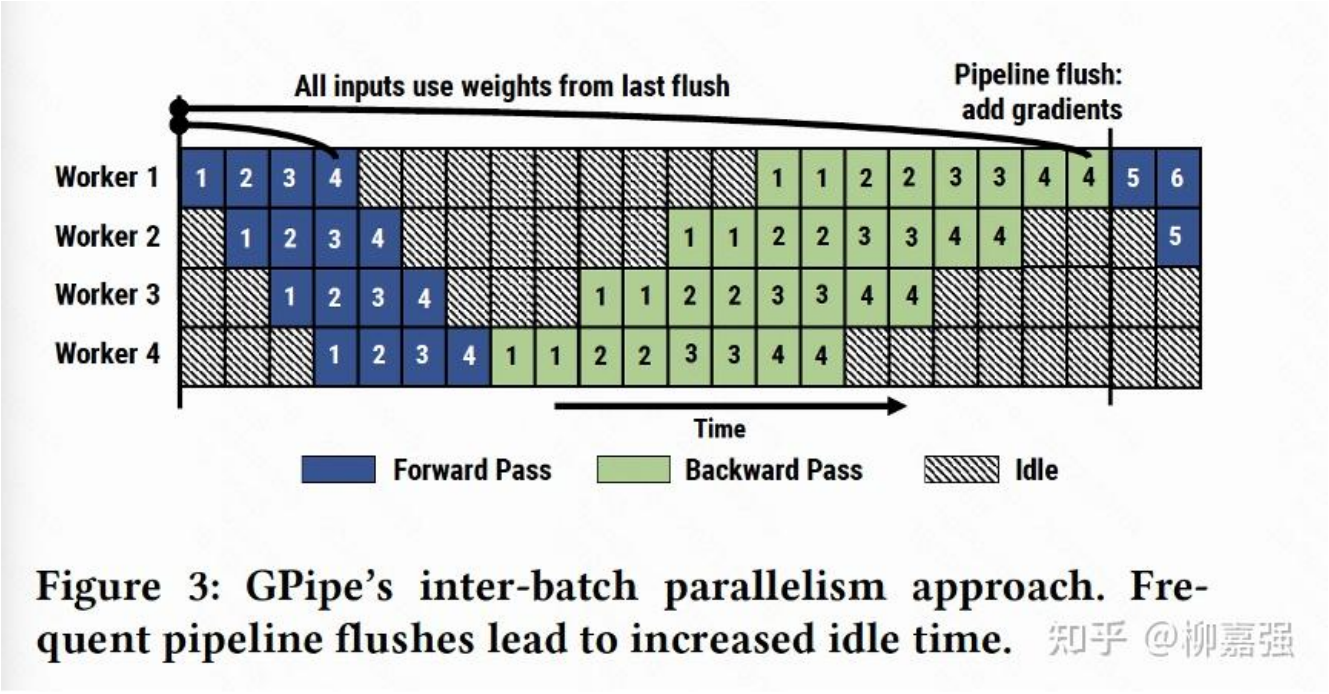

Figure 3: GPipe's inter-batch parallelism approach. Frequent pipeline flushes lead to increased idle time. 知乎 @柳嘉强

（图片摘自论文 PipeDream: Generalized Pipeline Parallelism for DNN Training）

上图是论文 GPipe: Efficient Training of Giant Neural Networks using Pipeline Parallelism. 提出的一种流水线并行解决方案。该方案中，每个流水线阶段的前向和反向由同一个 Worker 负责，在任意时刻，每个 worker 只会有一个前向或反向的计算任务。每个 STEP 会包括若干个 Batch，数量和 Worker 的数量一致，这些 Batch 会按流水线的方式依次完成前向和反向计算，在计算完成之后，再统一更新参数。这种方式保证了每个 Batch 计算时都使用了最新的参数，且在前向和反向计算中是一致的。

GPipe 的方案，虽然不存在参数不一致的问题，但如上图所示，在训练过程中有很多气泡，导致设备在很多时候是空闲的。为了解决这个问题，论文 PipeDream: Generalized Pipeline Parallelism for DNN Training 提出了一种新的流水线并行方案，如下图所示，这种方案在一开始也有气泡，但进入稳定状态之后，每个Worker 通过穿插执行前向和反向计算，填补了流水线中的气泡，从而能够提高设备的利用效率。

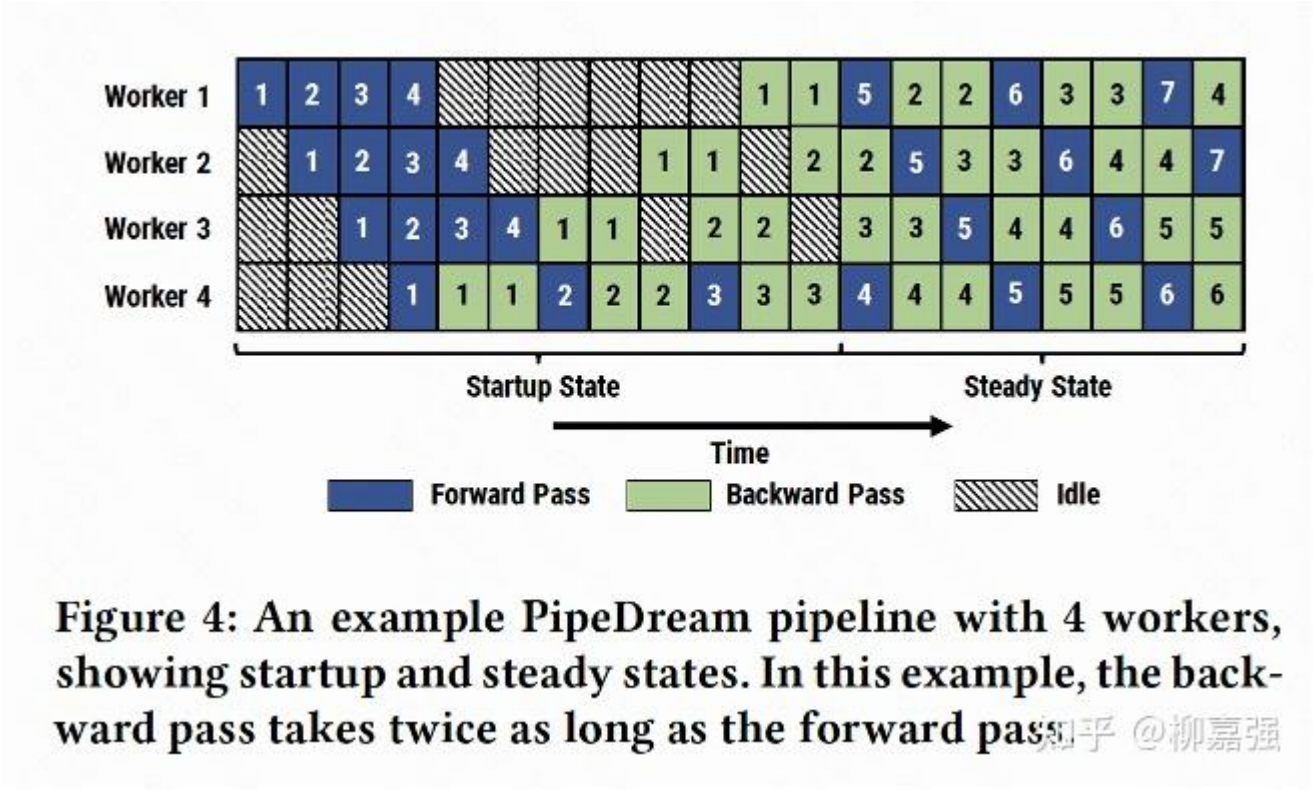

Figure 4: An example PipeDream pipeline with 4 workers, showing startup and steady states. In this example, the backward pass takes twice as long as the forward pass.

然而，上述方案前向计算和反向计算可能会使用不同版本的参数。举例来说，第 5 个 Batch 在进行第一个 STAGE 的前向计算时，Batch 1 的反向计算已经完成，因此参数是基于 Batch1 梯度更新之后的参数；但是第 5 个 Batch 在进行第一个 STAGE 的反向计算时，Batch 2、3、4 的参数也已经完成更新，因此参数是基于 Batch1-4 的梯度更新之后的参数。PipeDream 解决参数不一致问题的方法是维护多个版本的参数，保证同一个 Batch 在同一个 STAGE 内，前向计算和后向计算使用的参数是一致的。每个 Batch 在进行前向计算时，可以使用当前 STAGE 的最新参数，或者通过作者提出的 “Vertical Sync” 技术，使用和前一个 STAGE 相同版本的参数。例如，Batch 5 在进行第一个 STAGE 的计算时，使用基于 Batch 1 的梯度更新后的参数。在进行第二个 STAGE 的计算时，最新的参数是使用基于 Batch1-2 的梯度更新后的参数，此时，Batch5 可以选择使用最新的参数，也可以选择使用 STAGE2 中稍旧版本的参数，也即只基于 Batch1 的梯度更新后的参数，这样的好处是和 STAGE1 的参数保持版本一致，对收敛有帮助。

PipeDream 解决了流水线气泡和参数一致性的问题，但是由于每个 Worker 需要维护多个版本的参数，导致其内存开销较大，不能训练大模型。为此，作者进一步在论文 Memory-Efficient Pipeline-Parallel DNN Training 中提出了 PipeDream-2BW 的方案。如下图所示，该方案中，每个 Worker 会对不同 Batch 的梯度进行累积，然后每隔 N 个 Batch 进行一次更新，当 N 设置为 STAGE 数量时，每个 Worker 只需要维护两个版本的参数。

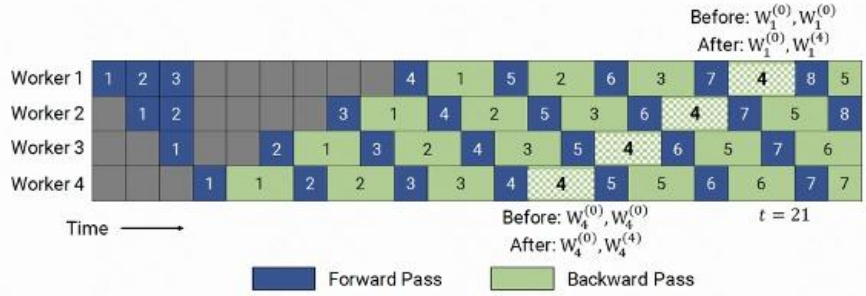

Figure 2. Timeline showing PipeDream-2BW's double-buffered weight update (2BW) scheme with time along x-axis. *Without loss of generality*, backward passes are assumed to take twice as long as forward passes. PipeDream-2BW only stashes two weight versions at every worker, reducing the total memory footprint while no longer requiring expensive pipeline stalls. $W_i^{(v)}$ indicates weights on worker i with version v (contains weight gradient generated from input v). New weight versions are generated in checkered green boxes. $W_4^{(4)}$ is first used for input 9's forward pass.