分布式机器学习框架的范式演进：从算子编排到命令式执行

——基于 TensorFlow 与 PyTorch 系统架构的研读报告

姓名：刘远航　学号：2023202275

一、 引言

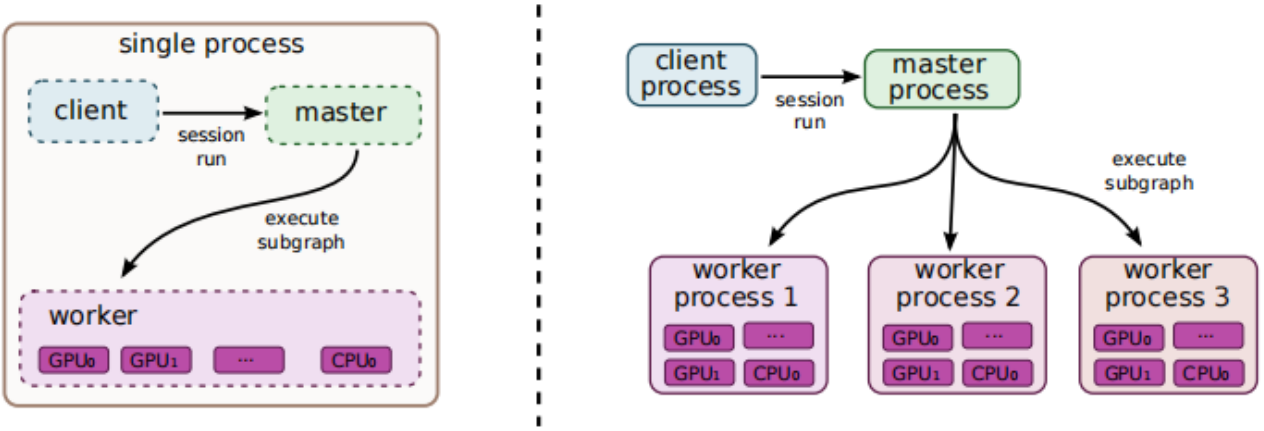
在并行计算课程上，我们小组做了以分布式机器学习为主题的汇报。其中，我负责分布式机器学习中基本架构的部分。在查阅资料的过程中，我注意到了日常我们跑深度学习经常遇到的两种技术框架——Pytorch和Tensorflow，前者非常流行，在学术界广受推崇；后者主要应用于工业级实现。二者有截然不同的设计哲学，给予我非常大的启发，因此我阅读了相关的两篇论文并撰写了这篇汇报。

在计算机体系结构向异构计算全面转型的背景下，分布式机器学习框架已实质上演变为支撑大规模算力调度的专用操作系统。TensorFlow（2015）与 PyTorch（2019）两篇论文不仅记录了两个主流框架的技术细节，更反映了并行计算领域在应对大规模深度学习任务时，关于如何平衡执行效率与开发灵活性的深层思考。通过对两篇论文的对比研读，我们可以清晰地观察到分布式系统如何从追求极致的静态优化，逐步转向以开发者为中心、兼顾硬件特性的务实架构。

二、 TensorFlow：以计算图为核心的静态并行机制

TensorFlow 的设计哲学建立在Dataflow Graph的抽象基础之上。这种设计的核心思想是将**计算过程与执行环境解耦**。在TensorFlow看来，机器学习模型不是一段普通的程序，而是一组预先定义的算子依赖网络。

这种高度抽象为并行计算带来了显著优势。由于计算图是静态的，系统可以在实际执行前进行深度优化。例如，论文中提到的算子融合和静态内存预分配减少了运行时的系统开销。更重要的是在处理异构硬件时，TensorFlow 引入了一种自动化的分布式编排机制。其主控端可以利用启发式算法，根据算子的计算密度和张量大小，自动将计算图切分为多个子图，并自动插入Send/Receive节点处理跨设备通信，实现了计算与通信的拓扑感知调度，如下图所示。

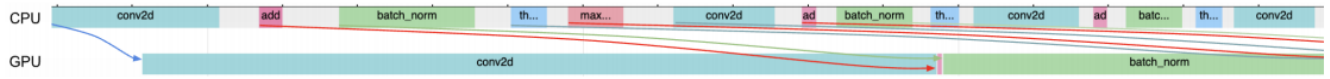


这种**先编排、后执行**的模式，本质上是并行计算中典型的**空间换时间**策略。通过静态分析插入 Send/Receive 节点，TensorFlow 成功地将复杂的网络通信隐匿于图层之下。然而，正如后续研究所指出的，这种对静态结构的依赖，虽然在生产环境的大规模部署中表现稳健，却在科研探索阶段造成了极大的调试阻碍。

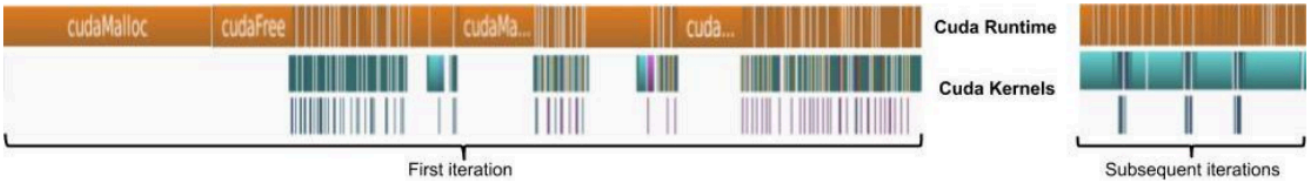
三、 PyTorch：命令式风格下的异步执行优化

与 TensorFlow 不同，PyTorch 的核心思想是“**模型即程序**”。它摒弃了静态图的中间层，直接采用 Python 的命令式风格进行即时执行。PyTorch 论文的核心论点在于：**高性能并不一定非要依赖复杂的静态编译优化，通过精巧的运行时设计，动态框架依然可以实现卓越的并行效率。**

在我看来，PyTorch 系统架构中最具启发性的部分在于其**对控制流与数据流的异步分离**。在单机多卡或分布式训练中，CPU 负责逻辑控制和任务分发，而 GPU 负责实际的数据计算。PyTorch 通过异步执行机制，允许 CPU 线程快速跑在前面排队分发任务，而 GPU 则在后台饱和式地处理任务队列。论文中的执行追踪示意图直观地展示了这种 CPU 调度与 GPU 执行的重叠，这是并行计算中**掩盖延迟**思想的典型应用。



此外，PyTorch 在内存管理上的务实策略也值得关注。针对 GPU 显存频繁申请与释放导致的碎片化和延迟问题，PyTorch 设计了自定义的缓存分配器。这种设计不仅保证了即时执行的灵活性，也确保了显存利用率能与静态预分配框架相媲美。这种从底层硬件特性出发的优化证明了在分布式并行系统中，微观层面的资源调控或许会比宏观层面的图优化更具实战价值。



四、 TF和Pytorch的共同点

纵观两篇论文，我发现虽然它们在编程接口上不同，但在分布式并行的底层逻辑上却存在着相似之处：

（1）**异构计算的抽象：** 无论是 TensorFlow 的设备放置算法，还是 PyTorch 的显式设备转移，都是在解决如何将张量运算映射到 CPU、GPU 甚至 TPU 上的核心问题。

（2）**通信模式的回归：** 早期 DML 框架深受大数据处理思路影响，多采用异步的参数服务器模式。而随着模型参数量的激增，两篇论文都开始更多地探讨同步并行与集体通信（如 AllReduce），这标志着分布式机器学习正在向HPC的经典并行范式回归。

（3）**互操作性与扩展性：** 两者都强调了与现有生态（如 NumPy、C++ 内核）的互操作性。PyTorch 的多进程共享内存机制与 TensorFlow 的分布式运行时，都是为了在 Python 环境下突破全局解释器锁的限制，以实现真正的并行。

五、 个人体会

通过阅读这两篇论文，我对分布式系统的构建有了更深理解。

首先，PyTorch 的成功给了我很大启发。很多时候，我们过度迷信编译器级别的全局优化，却忽略了硬件本身的特性。PyTorch 通过重叠 CPU/GPU 负载、优化显存分配器等手段，在不破坏用户编程习惯的前提下实现了极高性能。这提示我，在未来的并行程序设计中，应首先考虑如何最大限度地压榨硬件的异步并行能力。

其次，系统的灵活性是第一生产力。TensorFlow 1.x 版本的复杂性曾拖累了其生态发展，而 PyTorch 凭借 Pythonic 的接口迅速占领市场。在分布式系统设计中，“用户友好”不应被视为次要因素。一个难以调试的并行系统，其维护成本可能会抵消它带来的所有计算效率提升。

最后，显存墙与带宽墙是分布式并行的最终挑战。研读过程中我发现，无论框架如何演进，分布式训练的瓶颈始终在内存容量和网络带宽上。当前大模型的兴起对并行策略提出了更高要求，如流水线并行和 3D 并行。我意识到，未来的研究方向不应仅限于算子加速，更应关注如何通过拓扑感知的调度算法，减少分布式节点间冗余的数据流动。

六、参考文献

【1】[NeurIPS'19] Adam Paszke , Sam Gross , Francisco Massa, et al., “PyTorch: An Imperative Style, High-Performance Deep Learning Library”（PyTorch）

【2】[OSDI'16] Martín Abadi , Ashish Agarwal , Paul Barham, et al., “TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems”（TensorFlow）