计算机系统应用 ISSN 1003-3254, CODEN CSAOBN
Computer Systems & Applications,2022,31(9):114−126 [doi: 10.15888/j.cnki.csa.008672]

E-mail: csa@iscas.ac.cn
http://www.c-s-a.org.cn
Tel: +86-10-62661041

Kubernetes 集群上深度学习负载优化①

陈　培, 王　超, 段国栋, 王德奎, 王　斌, 王文潇, 孙辽东, 荆荣讯, 邢良占, 刘慧兴, 姬贵阳

(浪潮电子信息产业股份有限公司, 济南 250101)
通信作者: 王　超, E-mail: chaowang@inspur.com

摘　要: 人工智能技术的快速发展和在云原生上部署应用高效等优点让越来越多的开发者和互联网企业将人工智能应用部署在 Kubernetes 集群上, 但 Kubernetes 并不是主要针对深度学习而设计, 对深度学习这个特定领域需要做定制优化. 本文针对具有一定规模的 Kubernetes 集群上部署深度学习负载的场景, 设计和实现了一系列优化方案, 主要从深度学习所要求的数据处理、graphics processing unit (GPU) 计算、分布式训练等几个方面进行优化, 本文提出的优化方案覆盖了数据处理、计算等方面, 这些技术极大简化人工智能负载在规模化云原生平台上的部署难度和提高运行效率, 同时从实践上来看也验证了以上技术对人工智能应用有着显著的提升作用.

关键词: Kubernetes; 深度学习; 分布式训练; CUDA; 负载优化; 人工智能

引用格式: 陈培,王超,段国栋,王德奎,王斌,王文潇,孙辽东,荆荣讯,邢良占,刘慧兴,姬贵阳.Kubernetes 集群上深度学习负载优化.计算机系统应用,2022,31(9):114–126. http://www.c-s-a.org.cn/1003-3254/8672.html

Optimization of Deep Learning Workload on Kubernetes Cluster

CHEN Pei, WANG Chao, DUAN Guo-Dong, WANG De-Kui, WANG Bin, WANG Wen-Xiao, SUN Liao-Dong, JING Rong-Xun, XING Liang-Zhan, LIU Hui-Xing, JI Gui-Yang

(Inspur Electronic Information Industry Co. Ltd., Jinan 250101, China)

Abstract: Owing to the rapid development of artificial intelligence (AI) technologies and the efficient deployment of AI applications on cloud-native platforms, an increasing number of developers and internet companies deploy AI applications on Kubernetes clusters. However, Kubernetes is not designed chiefly for deep learning, which, as a special field, requires customized optimization. This study designs and implements a series of optimization schemes, mainly from the perspectives of data processing, graphics processing unit (GPU) calculation, and distributed training that deep learning requires, for the scenario of deploying deep learning workloads on Kubernetes clusters of a certain scale. The proposed optimization schemes involve data processing and calculation. These technologies reduce the difficulty in deploying AI workloads on large-scale cloud-native platforms and improve operational efficiency greatly. Moreover, the practice also verifies their significant improvement effect on AI applications.

Key words: Kubernetes; deep learning; distributed training; CUDA; load optimization; artificial intelligence

近年来人工智能技术快速发展, 尤其是深度学习方面取得了诸多令人瞩目的成就, 而 Kubernetes 作为下一代分布式系统的主流, 作为云原生的新生力量, 其发展也是十分迅速, Kubernetes 有着完善的组件和工具生态系统, 能够减轻应用程序在公有云或私有云中运行的负担, 并且可以和任何场景结合, 另外 Kubernetes 的插件化、组件化开发方式能够支持更多定制化的设计开发工作, 这些优点让越来越多的开发者和互联网

① 收稿时间: 2021-12-04; 修改时间: 2022-01-04; 采用时间: 2022-01-20; csa 在线出版时间: 2022-06-16

企业将人工智能应用部署在 Kubernetes 集群上. 但由于 Kubernetes 并不是主要针对深度学习设计, 对深度学习这个特定领域需要做定制优化.

目前针对 Kubernetes 集群部署深度学习应用已有很多优化尝试, 如国内腾讯的 Gaia 调度系统[1,2] 能够细粒度使用 GPU 资源, 但是并没有针对数据集使用和分布式训练进行优化. 对于 GPU 虚拟化使用, NVIDIA 推出了 Multi-Process Service[3] 和 virtual GPU (vGPU)[4] 两种方案, 但 MPS 具有故障传递限制[5] 而且 vGPU 需要授权使用, 其他的很多开源方案在具体实践应用上都有一定的局限性.

本文针对具有一定规模的 Kubernetes 集群上部署深度学习负载的场景, 设计和实现了一系列的优化方案, 并且已经在实际生产环境中实践, 取得了良好的效果.本文从深度学习所要求的数据处理、graphics processing unit (GPU) 计算、分布式训练等几个方面进行优化, 主要优化方面有以下几点: 针对目前人工智能应用只能占用整数 GPU 卡资源, 难以实现 GPU 卡资源多任务复用的场景, 提出 GPU 多任务共享调度技术, 能够实现多种应用共享同一张 GPU 卡资源, 极大限度的挖掘 GPU 计算力, 提升 GPU 的使用效率; 随着训练数据集规模的快速增长, 提出训练数据集预加载技术能快速提高数据集读取速度进而提高单机和分布式的训练速度; Kubernetes 的原生调度系统和策略并不能很好的满足目前人工智能场景, 因此提出了针对 non uniform memory access (NUMA) 特性[6]、数据集亲和性的优化调度技术. 本文提出的优化方案覆盖了数据处理、计算等方面, 以上技术极大简化人工智能负载在规模化云原生平台上的部署难度和提高运行效率, 同时从实践上来看也验证以上技术对人工智能应用有着显著的提升作用.

1 基于缓存机制的数据读取加速技术

云原生上部署深度学习应用时, 会遇到对接不同底层存储的情况, 诸多实验表明, 不同存储系统对深度学习训练性能是有着不同程度的影响. 训练开始时, 存储系统的性能会极大的影响训练数据的加载和后续训练数据的持续读取, 如 network file system (NFS)[7] 等传统存储系统对于海量数据和海量小文件数据的读写性能不是很高, 而在深度学习训练场景 (图片、视频、语音等) 中的训练数据大多是以文件形式而存在, NFS 系统在很多实际场景都会使用, 这样会很大程度上影响训练数据的读取速度进而影响整体训练性能. 高性能存储系统如 BeeGFS[8] 等, 如果在高并发、大数据量和网络带宽有限制情况下也会出现性能下降问题. 因此, 加速训练过程的数据前期读取和简化在云原生上对不同存储的对接亦是业界热门问题, 本文提出在 Kubernetes 集群上利用本地高性能存储设备进行数据缓存进而提高训练过程前、中的数据读取速率, 大大减少对存储系统和网络的依赖, 另外根据深度学习任务特性进行针对性优化, 与直接使用传统存储系统的表现相比这些改进提高了训练性能.

Kubernetes 集群上完成对数据集缓存需要对训练数据进行很多的针对性处理和设计, 本文对 Kubernetes 集群上数据集本地和节点缓存两种情形设计了一套缓存系统, 该缓存系统要求需要对数据集生命周期具有细粒度的管理特性. 数据集缓存系统共分为两种方式, 即本地模式和节点缓存模式. 本地模式即直接使用从存储系统中读取和使用数据集, 不做任何缓存, 数据读取速率则完全依赖于存储系统性能或者网络带宽. 节点缓存模式即本文提到的数据缓存加速机制实现, 在运行训练任务的节点上进行数据集缓存和管理, 这样在训练时存储系统性能和网络带宽就不会是瓶颈, 如果底层存储介质为高性能设备如 SSD、NVMe 则会有更大的增益.

本文针对训练使用数据集的过程做了主要以下优化工作:

(1) 设计 dataset-agent 实现在 Kubernetes 内数据集的本地和节点缓存两种使用模式.

(2) 针对大数据集和海量小文件数据缓存过程进行了优化并提高效率.

(3) 简化使用过程和能够对接不同存储, 使用者无需关注底层存储系统.

两种数据集使用方式的架构如图 1 和图 2 所示.

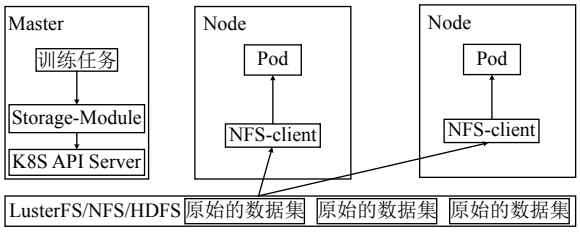


图 1 本地模式架构图

本地模式数据读取为直接使用存储系统, 不在本文中详述, 整体架构如图 1 所示, 实现逻辑为管理节点

的 Kubernetes API Server 接收到用户任务请求后下发训练任务数据集名称、存储位置等信息给 Kubernetes, 然后 Kubernetes 的调度器调度训练 Pod 到相应节点上进行训练, 其中数据集的加载和管理则完全依靠集群内的存储系统进行管理, 如上所述, 性能和使用逻辑则完全依靠存储系统, 由于分布式和共享存储的训练数据读取很大程度上会与网络环境强相关, 因此对训练性能会有一定程度的影响, 取决于网络设备性能和并发量, 节点缓存模式则避免上述问题.

节点缓存模式的架构如图 2 所示, 各个计算节点上都会部署 dataset-agent 服务即数据集代理服务, 主要管控数据集缓存的生命周期, 包括数据集缓存创建、删除、更新等信息. 其实现机制为在实际训练开始前提前将训练用数据集缓存到被调度使用的节点, 这样在实际训练时, 训练数据则为本地数据, 不再受存储系统或者网络性能的影响. 具体实现逻辑为 Kubernetes 管理节点接收到训练任务请求后, 在承载训练任务的 Pod 中通过 init-container 形式将训练所需的信息如数据集名称等进行封装, Kubernetes 调度 Pod 成功后, init-container 首先会将数据集信息下发到部署到该节点上的 dataset-agent, 随后 dataset-agent 进行校验 (数据一致性) 决定是否访问存储系统进行数据集拉取进行缓存.

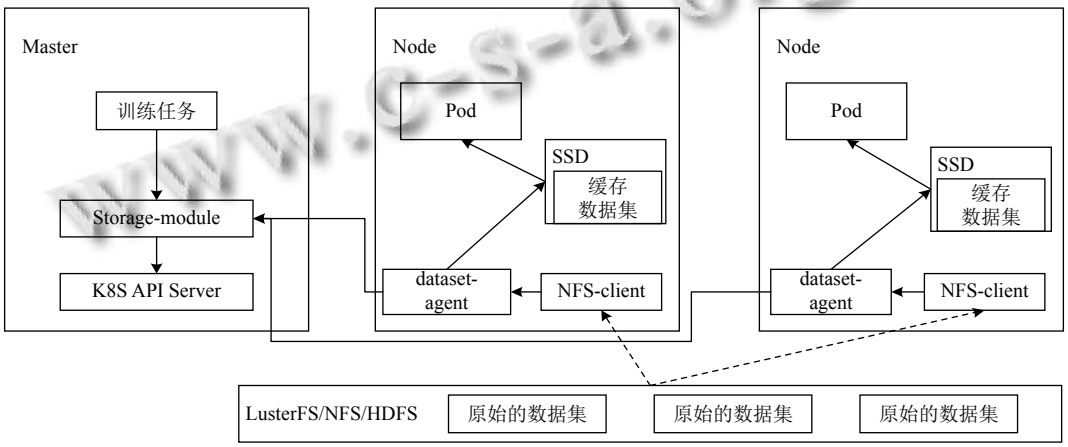


图 2 节点缓存架构图

如果该节点进行了节点数据缓存, 即在训练节点上的存储设备上就会有相应的训练数据.数据缓存结束后, 随即启动训练程序 Pod 同时将缓存的数据集进行挂载, 这样就能直接利用本地存储介质进行数据读取和训练, 消除了存储系统和网络的影响, 提高训练速度, 经测试该系统能够支持多种存储系统包括 NFS、LusterFS[9]、BeeGFS、HDFS[10] 等.

节点缓存技术在部署到 Kubernetes 集群生产环境中时遇到很多实际问题, 为此针对如下一些主要情况做了优化处理.

(1) 实际过程中一个训练任务会出现挂载多个数据集的情况, 所以将这样一对多的组合作为一个请求任务进行管理, 同一训练任务在多次中只会有一条任务数据, 避免了由于 init-container 重启导致多次发起数据集缓存请求造成重复数据的情况.

(2) 数据集一致性比对, 主要是针对缓存数据集前的校验工作, 针对数据集名称相同, 通过比对原始数据集文件摘要和节点中数据集缓存的文件摘要来确认数据集和需要的数据集缓存是否相同, 这种比对规则的前提, 即假定节点中的数据集缓存不会变更, 也就是在缓存前生成的文件摘要是准确不变的, 详细流程见图 3.

(3) 数据集缓存和镜像拉取并行处理, 在节点没有即将加载运行的镜像情况下, 被调度到该节点的 Pod 相当一段时间是在进行镜像拉取工作, 由于该段时间资源 (CPU、内存等) 已经分配, 但是并没有在实际使用这些资源, 因此可以有效利用 Pod 调度成功并且运行前这一时间段内 Pod 的空闲资源来提升拉取数据集的效率, 这样相对较小的数据集在镜像拉取完成的同时也缓存完成.

(4) 自动清理数据集缓存, 当计算节点空间不足时, 清理缓存数据释放空间就变得十分必要, 系统根据既定的规则自动清除, 清除策略发生在新训练任务缓存数据集, 但缓存空间不足时, 系统亦会清除缓存空间中

未在使用的数据集, 且长时间未使用的数据集也会作为清除对象被自动清除, 当清除的空间能够满足新数据集缓存时候, 即停止清除, 这里需要清除空间的大小是所需数据集大小的 1.2 倍.

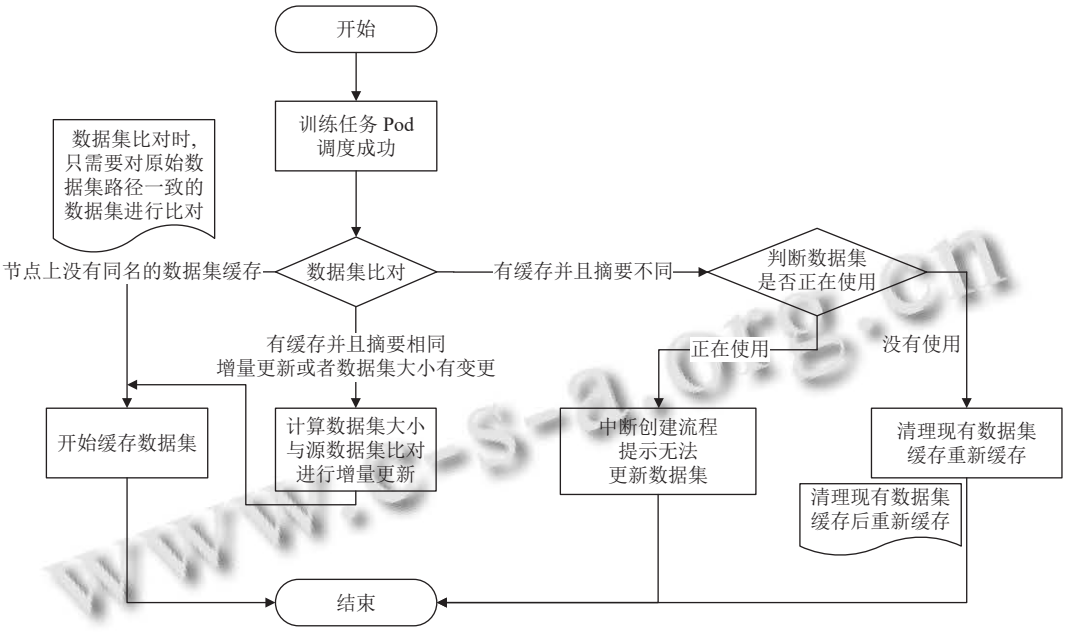


图 3 数据集一致性比对流程示意图

(5) 支持多种存储系统下的数据集操作, 即将相关配置信息放到 Pod 的 yaml 的环境变量中来解决.

(6) 多个数据集缓存进程并发, 一个节点上的 dataset-agent 可以处理多个数据集缓存操作来提高效率, 同时每个 dataset-agent 也做了多线程处理, 提高处理速度.

(7) 海量小文件数据集缓存优化处理, 采用对待缓存数据进行压缩后再进行传输完成节点缓存, 这样极大提高带宽的利用率, 实际测试中能够有效减少因为网络传输带来的数据缓存延迟, 聚合传输测试出文件个数为 5 万打一个包, 每个包的大小约为 800 MB 左右, 该设置下效率较高, 性能比命令传输提高 6–8 倍. 聚合传输可根据实际业务场景调整聚合传输相关参数, 解决机器资源占用、传输效率问题.

(8) 特别针对 NFS 系统, 通过使用 NFS-RDMA 技术[11] (如图 4 所示) 在小文件传输方面性能提升 2 倍左右. 通过 NFS-RDMA V3 协议在小文件传输上提高约 1.5 倍.

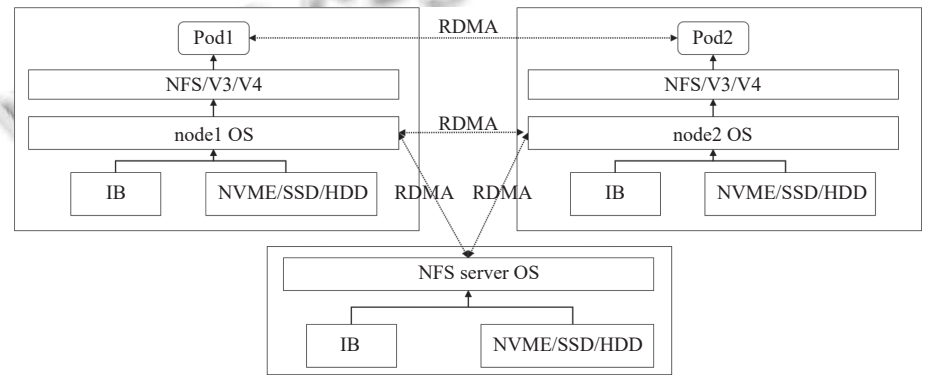


图 4 NFS-RDMA 技术应用示意图

2 vCUDA-GPU 虚拟化技术

GPU 有着强大的处理并行计算任务能力, 其单一芯片上集成大量的计算核心的架构设计使得 GPU 对于计算敏感性任务尤其适用. 当前 GPU 已经广泛应用

于视觉、自然语言处理等领域来加速处理过程.

CUDA (compute unified device architecture) 是 NVIDIA 针对 GPU 架构提供的一套针对 GPU 的通用 API 平台, 用户可以通过 CUDA 简单和快速的使用 GPU 以达到加速效果. 无论在公有云或者私有云, GPU 设备已经广泛部署并使用, 起到了显著的效果. 实际使用中 GPU 使用实例的类型也是多种多样, 对底层计算资源的要求也不同. 作为底层提供计算资源支持的平台对于用户来说应该是透明的, 需要做到按需索取和使用. 随着深度学习业务广泛铺开和 GPU 的架构快速迭代, 对于 GPU 计算资源的需求也变的多样, 从单卡训练到分布式训练, 从独占使用到多任务共享. 但 Kubernetes、GPU (除 Ampere 架构[12]) 和 CUDA 原生并不支持 GPU 细粒度调度和使用. 基于容器的 GPU 虚拟化的技术在 Kubernetes 上容器化使用 GPU 也同时面临如下一些具体问题: 需要指定 GPU 设备[13]; 只能独享该整个 GPU 设备[14], 不能多任务共享; 单一 GPU 使用容器间只能共享主机内存[15].

本文提出一种能够在互相隔离的容器间进行共享同一 GPU 设备内存的方法来提高 GPU 的利用率.该方法不需要更改用户的镜像或者训练代码即可达到 GPU 虚拟化的目的. 通过自定义修改的 Kubernetes device plugin 可以实现按显存大小来分配 GPU 资源, 即 GPU 可按显存大小粒度进行调度使用, 不再局限于整卡级别的粒度, 同时进程间可以做到隔离, 保证用户应用不会互相影响. 除此之外利用 unified memory 技术[16]实现显存的超分使用, 即在实际训练过程中可以保证超出 GPU 总显存量进行训练, 在适量超出 GPU 显存容量后保障较大模型正常训练.

由于 docker 出于安全对权限做了限制导致 NVML 接口[17] 在容器内无法查询正在 GPU 上运行的进程, 为此本文针对 GPU 上正在运行进程的查询机制做了优化, 实现在容器内可以查询正在 GPU 上运行的进程信息, 可以正确的显示该容器内运行进程而不是主机上的进程以保证进程安全和访问安全.

本文针对 GPU 虚拟化主要做了以下工作:

(1) 通过 GPU sharing device plugin 实现在 Kubernetes 内细粒度调度 GPU 任务.

(2) 封装 CUDA driver API 实现 GPU 虚拟化使用.

(3) 添加显存超分使用, 即超出 GPU 总显存量可以继续进行训练.

(4) 优化 NVML 查询 GPU 进程机制使得在容器内正确显示 GPU 上运行的进程信息.

2.1 Device plugin

Kubernetes 的 device plugin 插件的主要用途为将计算资源信息 (如 GPU, RDMA, FPGA[18] 等) 发布给集群并无需修改 Kubernetes 核心代码, 图 5 展示了基本的 device plugin 与 kubelet 通讯过程, 主要通过两个步骤实现.

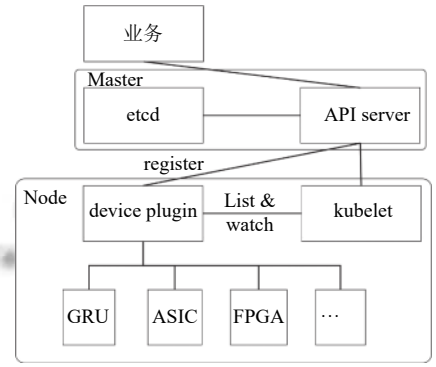


图 5 Kubernetes device plugin 和各组件关系

(1) 资源发现: 首先每种扩展的资源类型都作为一个 device plugin 形式展现. Device plugin 通过 gRPC 服务注册到 kubelet 上. 注册成功后, device plugin 将其所管理的设备列表发送给 kubelet. 最后 kubelet 负责将这些扩展资源发布给 Kubernetes master;

(2) 资源分配: 当用户申请资源时, 调度器会将相应的 Pod 调度到具有所申请扩展资源的节点上. 所在节点的 kubelet 会将设备使用请求发送给 device plugin. 然后 device plugin 将相应的扩展资源分配给 Pod. 但针对 GPU 等设备, 直接使用开源 device plugin 并不能针对 GPU 内存进行细粒度的使用和分配.

本文中将现有的一些扩展设备资源 (如 GPU 等) 的 device plugin 进行优化, 实现了以下功能: 基于 Kubernetes 标准的 device plugin 机制, 支持接入多种 AI 计算资源; 多种可调度的资源在业务上统一建模, 以资源名称、数量、类型等; 信息描述接入集群的异构实现, 实现统一的调度、运维管理; 实现多 device plugin 管理插件, 由一个 device plugin 实现多个异构资源的注册、分配等, 且 plugin 的资源使用仅需要 0.1 CPU/0.3 GB 内存, 降低运维成本; 实现 GB 粒度的资源管理以及 GPU 复用场景下的资源管理.

2.2 GPU 虚拟化设计和实现

整体 vCUDA 架构设计和流程如图 6 所示, 主要由 3 部分组成: GPU sharing device plugin (以下简称 GS device plugin), 调度器 scheduler 和 vCUDA library.

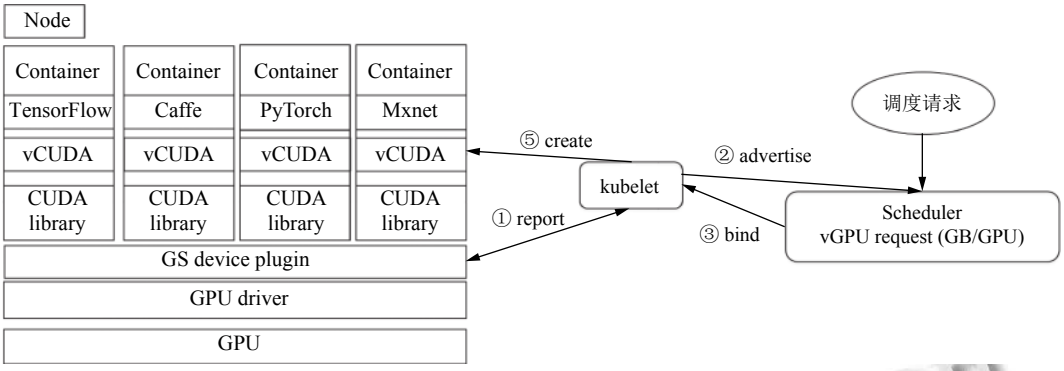


图 6 vCUDA 架构和流程示意图

(1) GS device plugin

其中经过修改和优化的 GS device plugin 在各个节点上运行负责建立虚拟 GPU 设备和与 kubelet 进行通讯. GS device plugin 发现设备上报时将 GPU 显存视为一种资源进行上报, 这样 GPU 显存也可以作为可调度的 Kubernetes 集群资源进行使用.

(2) 调度器 scheduler

调度器为 GS device plugin 提供其所申请的调度服务, 调度成功后调度器会返回包含所分配 GPU 信息的响应.

(3) vCUDA library

在运行的 Pod 中 vCUDA 负责实际的内存控制. vCUDA 库通过挂载的方式与运行的 Pod 进行绑定. 当容器中应用开始运行时, vCUDA 通过对训练过程中内存相关的 API 进行劫持从而实现内存大小的控制和隔离, 主要由以下几个部分组成:

(1) vCUDAManager: vCUDA library 的总控制, 对于 CUDA 的操作均需要通过该类对象, 单例运行只初始化一次. 其中主要包括 cudaManager、nvmlManager、gpuMemoryManager 和 dlsym 的 map 管理.

(2) cudaManager: 管理所有 CUDA API 的劫持, 主要是 cuMalloc 类似的函数, 当分配显存时调用此类的接口来控制 OOM 问题.

(3) nvmlManager: NVML API 的劫持管理类, 主要是获取 NVIDIA GPU 卡上各个进程运行的详细信息, 如显存和进程 PID.

(4) gpuMemoryManger: 记录各个 GPU 卡的显存利用信息, 当分配显存时会调用此类 API 判断是否 OOM.

GPU 虚拟过程主要通过 GPU 的显存申请和分配来实现. 本文中以 1 GB 显存作为基本粒度, 一个最终的内存分配单元作为一个虚拟 GPU 设备.当用户申请一个规定大小 (GB 粒度) 的虚拟 GPU 调度请求后, 调度器会将请求发布到给个节点上的 kubelet, 由于 GS device plugin 已经将其所管理的设备列表和资源信息发送给 kubelet, 因此 GS device plugin 在收到所分配的 Pod 为虚拟 GPU 的请求后将 Pod 所要创建的 allocateResponse 返回给 kubelet 进行资源创建即可, 其中包括基本的 Pod 环境变量, Pod 挂载卷配置 (例如 NVIDIA 驱动, CUDA 库, vCUDA 库) 和相应设备.

另外通过 dlsym 劫持函数的 map 对象中, 针对 NVML 库设置单独劫持进行处理主要是为了防止其他应用通过 dlsym 来调用 CUDA 和 NVML 的 API, 例如 nvidia-smi 命令, 在使用 CUDA 劫持库时始终保持结果一致性. 具体使用到的 CUDA 和 NVML API 如表 1 和表 2 所示.

基于不同场景, 本文中提到 GPU 虚拟化实现如下场景支持:

(1) 基于 GPU 显存隔离的 GPU 虚拟化

资源调度模块将多个 GPU 训练任务调度到同一张 GPU 卡, 通过设置的显存粒度大小, 对 GPU 显存进行切片, 来限制每个任务对 GPU 显存大小的使用. GPU 任务通过设置的显存粒度切片, 来达到不同任务所用显存互相隔离的效果.

设置好需要使用的显存粒度大小后, 具体使用时就把 GPU 卡整块显存按预置显存粒度大小分割为多个粒度切片进行隔离. 如图 7 上半部分场景所示, 4 个 GPU 卡, 每个 GPU 卡的显存大小为 24 GB, 系统设置显存粒度大小为 8 GB, 这样, 每个 GPU 卡就可以分割成 3 个 8 GB 细粒度的 GPU 切片. 当作业运行时, 作业会按 GPU 切片请求 GPU 资源, 像图中 task1 就是要

求 4 个 GPU 切片, 每个切片大小为 8 GB. 然后系统资源调度器会针对显存隔离场景作业执行相应的调度策略, 给 task1 分配了 GPU0、GPU4、GPU2、GPU3 卡中各 1 个显存切片. 再看单个 GPU 卡中运行的多个作业, 各作业所使用的显存是互相隔离的, 如图中 GPU0 卡运行的 task1、task3、task4.

表 1 CUDA driver API 使用情况

API类别	CUDA driver API	描述
Memory-related APIs	cuMemAlloc	Allocate device memory.
	cuMemAllocManaged	Allocate memory that be automatically managed by the unified memory system.
	cuMemAllocPitch	Allocate pitched memory.
	cuMemFree	Free device memory.
	cuArrayCreate	Create a 1D or 2D array.
	cuArray3DCreate	Create a 3D CUDA array.
	cuMipmappedArrayCreate	Create a CUDA mipmapped array.
	cuPointerGetAttribute	Return information about a pointer.
	cuGetErrorString	Get the string description of an error code enum name.
	cuInit	Initialize the CUDA driver API.
	cuCtxCreate	Create a CUDA context.
Device-related APIs	cuDeviceTotalMem	Return the total amount of memory on the device
	cuMemGetInfo	Get free and total memory.
	cuDeviceGetUuid	Return an UUID for the device.

表 2 NVML API 使用情况

API类别	NVML API	描述
Device-related APIs	nvmlDeviceGetComputeRunning-Processes	Get information about processes with a compute context on a device.
	nvmlDeviceGetHandleByUUID	Acquire the handle for a particular device, based on its globally unique immutable UUID associated with each device.
	nvmlDeviceGetIndex	Retrieve the NVML index of this device.
	nvmlDeviceGetMemoryInfo	Retrieve the amount of used, free and total memory available on the device, in bytes.
	nvmlDeviceGetComputeRunning-Processes	Get information about processes with a compute context on a device.

(2) 基于 unified memory (UM) 显存隔离的 GPU 内存使用优化

NVIDIA 的 Pascal[19] 之后架构均已经支持 UM 技术, 得益于启用 UM 机制, 将主机部分内存分配给 UM 管理, UM 可以用来统一管理分配作业请求的 GPU 显存资源, 此时 UM 就扩展了系统中 GPU 可用显存大小, 可以满足运行更多的作业和对较大模型的支持. 图 7 下半部分场景所示显存粒度大小为 8 GB 时, 如图所示按 UM 管理系统 UM 显存容量为 24 GB (其中 8 GB 为系统内存), 这样在资源调度器可以执行 UM 场景下的调度策略, 即 16 GB 显存大小的 GPU 卡就可以同时运行 3 个 GPU 作业 (如图 7 中 task1、task2 和 task3), 每个作业使用 8 GB 显存粒度. 此时, UM 扩展了系统可用显存容量, 支持运行更多的 GPU 作业.

3 基于 AI 训练特点深度优化的调度策略

3.1 节点内 NUMA 亲和性调度

NUMA (非统一的内存访问) 是一种在多 CPU 系统上可用的技术, 允许不同的 CPU 以不同的速度访问不同部分的内存. 任何直接连接到 CPU 的内存都被认为是“本地的”, 并且可以快速地访问. 没有直接连接到 CPU 的任何内存都被认为是“非本地的”, 并且将有可变或较长的访问时间, 这取决于必须通过多少个互连才能到达目标. 在现代系统中, “本地”和“非本地”内存的概念也可以扩展到外围设备, 如 NIC 或 GPU. 为了实现高性能, 应该在分配 CPU 和设备尽可能的使它们能够访问相同的本地内存.

NUMA 拓扑管理就是用来处理实现高性能下的 CPU 与 GPU 等设备资源分配. 云原生上的很多深度学习负载除了依赖 GPU 计算能力之外, 对于某些特殊的应用同样对 CPU 也有着较强的依赖, 如数据前处理、数据搬运等过程. 因此在调度任务时, 考虑到 NUMA 拓扑特性, 将最优化的 NUNA 组合分配给任务, 则能起到提升整体集群的运行效率和资源利用率的效果.

总体上, 节点内 NUMA 亲和性调度遵循以下流程, 如图 8 所示: 当 Pod 创建的时候, 节点 kubelet 中的

topology manager 根据配置的 policy 策略来调度 Pod, 其中有如下调度策略:

(1) none (默认): 无策略, 极容易产生所调度的 Pod 使用的 CPU、内存和网卡分别位于不同的 NUMA node 上.

(2) best-effort: 根据当前 Pod 的资源请求, 尽量满足 Pod 分配的资源, 不满足的就随意划分.

(3) restricted: 严格保证 Pod 资源请求, 如果资源不满足 Pod 的 affinity 需求, Pod 就会进入 terminated 状态.

(4) single-numa-node: 满足 policy 的 Pod 请求都会从一个单独的 NUMA node 内进行分配, 如果不满足, Pod 会进入 terminated 状态. 这个跟前两个的区别是, 前两个可以请求从两个 NUMA node 都分配资源.

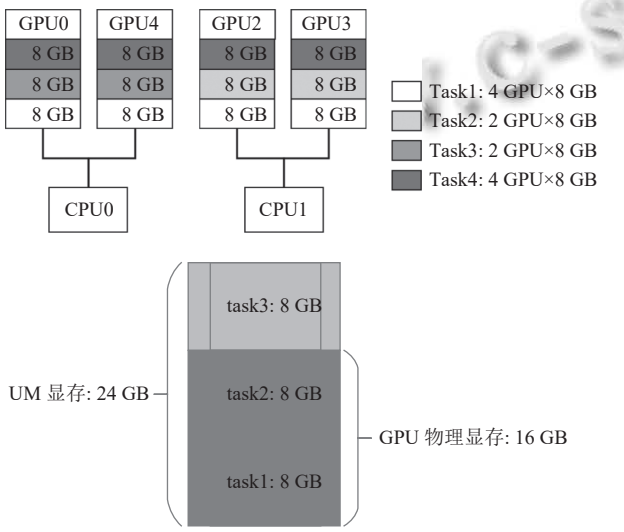


图 7 vCUDA 适用场景说明

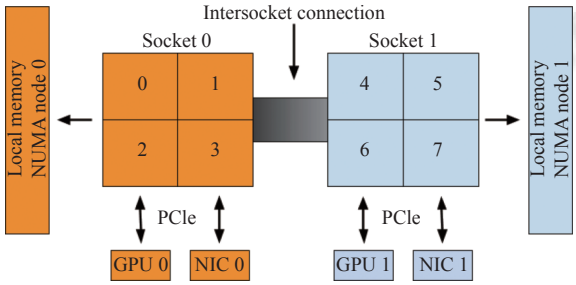


图 8 NUMA 亲和性示意图

3.2 模块交互

本文提出的 NUMA 拓扑管理通过 topology-Manager 和 deviceManager、cpuManager、GPU 设备之间交互来完成, 具体各模块功能描述和交互过程如下和图 9 所示.

(1) NUMA 拓扑管理器 topologyManager: 运行资源分配算法, 进行 CPU 和 GPU 资源的分配候选集的构造生成、合并优选和输出最佳分配结果; 同时更新维护 CPU 和 GPU 已分配状态信息.

(2) CPU 管理器 cpuManager: 提供获取 NUMA 节点信息接口; 按分配算法执行具体的 CPU 分配操作; 更新 CPU 分配状态.

(3) 设备管理器 deviceManager: 提供 GPU 设备的 NUMA 信息; 按分配算法执行具体的 GPU 分配操作, 选取包含必选设备中的最优设备组合, 具体说明如下: 如当前节点 kubelet 传递可用设备是 [GPU-0, GPU-1, GPU-2, GPU-3], 考虑 NUMA 亲和性的必选集 [GPU-2, GPU-3] 作为候选, 而实际业务需求为 3 个 GPU 卡, 则 GPU 插件会根据内置 GPU 通信打分规则 (见表 3)(该规则基于 GPU 与 GPU 间最优连接方式而确定, 由于 NVLink 具有 GPU 间最佳的传输效率, 因此该连接方式分数为 100 且随链数依次累加, 其他连接方式依次递减). 按照上述打分规则从 [GPU-0, GPU-1] 中选取最优的一个 GPU (如 GPU-0 (具有与 GPU2 或 GPU3 NVLink 连接)) 之后, 即组合 [GPU-0, GPU-2, GPU-3] 为最终分配组合; 其他功能如实现与 GPU 插件之间的业务接口和更新 GPU 分配状态等.

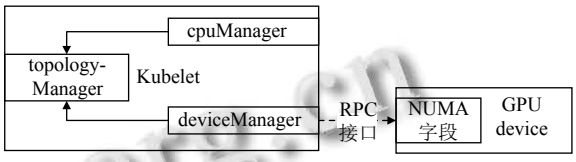


图 9 Kubelet 与 device plugin 交互示意图

表 3 Device plugin 内置通信打分规则 (GPU 对)

GPU link type	Score
P2P link, cross CPU socket	10
P2P link, same CPU socket	20
Host PCI bridge	30
Multiple PCI switches	40
Single PCI switch	50
Same board	60
Single NVLink[20]	100
Two NVLinks	200
Three NVLinks	300
Four NVLinks	400
Five NVLinks	500

(4) 快照管理器 checkpointManager: 提供进行 CPU 和 GPU 分配状态信息的生成、更新和删除接口.

3.3 针对 AI 训练的 Kubernetes 集群数据集亲和性调度

在集群上运行大规模数据集训练时, 会遇到如网

络带宽的限制影响数据集读取速率、训练 Pod 所在计算节点没有数据集缓存等情况导致需要重新进行缓存配置, 这些情况在大数据集和复杂 (文件类型多样、海量小数据等) 情况下会极大延长训练时间和占用网络带宽, 因此本文针对数据集的使用提出一种数据集亲和性调度策略来提高本文第一部分提到的数据集缓存技术使用效率和间接减轻集群间由于数据集传输带来的带宽占用.

基本的数据集亲和性准则为尽量选择作业 Pod 所需数据集完全匹配命中节点缓存数据集的节点. 对于未命中或部分命中的节点则忽略数据集亲和性策略, 具体则由图 10 举例说明数据集亲和性, 如集群中有 Node1 和 Node2 两个节点, 其中 Node1 上已缓存有数据集 A, Node2 上已缓存有数据集 B.如果作业需要数据集 A, 则调度器需要在 Node1 满足预选条件时, 并完全匹配作业所需数据集时才能将作业调度到 Node1 上. 如果作业需要数据集 A, Node1 不满足预选条件, Node2 满足预选条件, 则调度器会将作业调度分配到 Node2 上, 即数据集亲和性调度是一种优选算法策略.

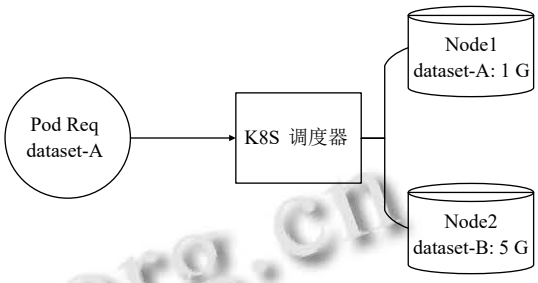


图 10 数据集亲和性场景

在进行数据亲和性调度时会对待选节点进行打分, 具体的得分计算方法如下.

节点缓存数据集得分公式:

$$\begin{cases} 0, \forall matchDataSet \neq podRqDS \text{ or } UpdateDatssetInUse \\ \sum NodeMatchDataSet(dsname), \forall matchDataSet = podRqDS \text{ and } UpdateDatasetInUse \end{cases}$$

$$\forall node \in clusterNodeList, \forall matchDataSet \in (podRqDS \cap nodeCacheDataSet)$$

其中, *node* 为集群节点选择器, 根据节点名称参数 *nodename* 选择节点; *clusterNodeList* 为集群节点列表; *NodeMatchDataSet* 为当前节点匹配作业要求数据集; *dsname* 为数据集名; *podRqDS* 为 Pod 所需的数据集; *UpdateDatasetInUse* 表示使用中的数据集需要更新; *nodeCacheDataSet* 为该节点上已经缓存的数据集.

公式说明:

NodeMatchDataSet 为当前节点匹配作业要求数据集, 当 Pod 请求的数据集和当前节点缓存数据集的交集如果匹配则累计该数据集, 否则过滤;

如果节点已缓存数据集并未完全匹配作业数据集, 则节点得分为 0;

如果作业要更新节点上正在使用的数据集, 则得分为 0;

如果作业所需数据集全部命中节点缓存数据集, 且数据集可用, 则节点数据集得分为完全匹配数据集个数, 即作业所需数据集个数;

如果解析作业所需数据集和节点数据集参数异常, 则节点得分为 0.

此分值为节点数据集原始得分, 后面还需经归一化处理后, 才能跟其它优选策略共同作用调度结果. 该得分值越高, 优先级越高, 得分越低, 优先级越低. 如果节点没有或者只有部分作业 Pod 所需的数据集, 则其得分为 0; 如果节点上有 Pod 所需要的全部数据集, 则得分为最高值. 如果出现多个节点的数据集亲和性得分相同且都为最高分, 调度器可参考其它优选策略进行优选.

4 实验与分析

文中提到的 vCUDA、NUMA、数据集亲和性调度策略已经在具体的生产环境中得到验证, 以下测试将具体展示其性能效果.

4.1 vCUDA 实验

分别采用 YOLOv3[21]、ResNet50[22]、BERT[23] 在视觉、自然语言处理领域的典型模型进行测试 vCUDA 性能效果.

实验环境如下:

模型: 采用 YOLOv3, ResNet50 和 BERT 模型, 网络模型包含了主要的图像处理、自然语言处理且具有相对复杂的网络结构能够充分利用 GPU 计算能力.

训练数据: 采用 COCO[24], ImageNet large scale visual recognition challenge (ILSVRC 2012)[25] 训练数据集 (约 130 G, 120 万张训练图片), SQuAD1.1[26].

训练框架: TensorFlow[27], Darknet[28].

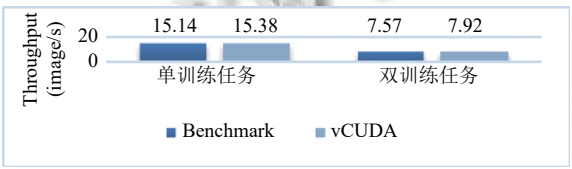
GPU: Tesla V100S 32 GB.

实验内容如下:

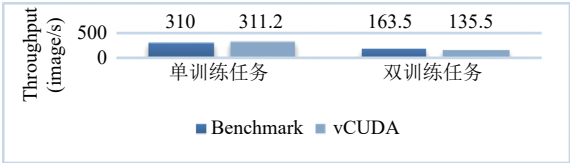
在单 GPU 上进行单个任务和两个任务测试, 采用直接在单 GPU 上调度训练任务 (即不限制内存使用和隔离) 和使用 vCUDA 进行内存限制和隔离进行对比.

实验结果和分析如下:

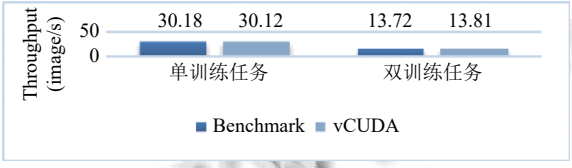
如图 11 所示单任务训练对比结果显示在 GPU 上只有一个训练任务时, 与 benchmark 结果比较, 采用 vCUDA library 对训练没有影响, 即 CUDA 的劫持方案并没有造成训练性能上的损失; 在同一个 GPU 上同时运行两个训练任务时, 采用 vCUDA 方案的训练效果和使用原生 CUDA 方案没有较大差异, 以上结果显示采用 vCUDA 方案可以进行细粒度显存使用和隔离的同时并保证同一块 GPU 上进行多任务的提交和训练, 以及性能保证.



(a) Darknet-YOLOv3, 数据集: COCO



(b) TensorFlow-ResNet50, 数据集: ILSVRC 2012



(c) TensorFlow-BERT, 数据集: SQuAD1.1

图 11 单任务训练对比结果

4.2 NUMA 亲和性实验

为验证同 NUMA 内计算资源对训练任务带来相应的影响, 设计如下实验.

实验环境如下:

GPU: A100 40 GB;

CPU: Xeon Platinum 8 268 @ 2.9 GHz, 2 Sockets, Cores per socket: 24;

存储系统: NFS.

实验内容如下:

(1) 采用 TensorFlow 框架和 ResNet50 使用 ILSVRC 2012 数据集进行测试, batch_size=418, steps=500, GPU 与同 NUMA node 和不同 NUMA node 的 CPU 绑定进行测试.

(2) 采用 PyTorch[29] 和 Transformer[30] 使用 wmt14_en_de 数据集[31] 进行测试, GPU 与同 NUMA node 和不同 NUMA node 的 CPU 绑定进行测试.

采用 TensorFlow 和 ResNet50 使用 ILSVRC 2012 数据集在 GPU 与同 NUMA node 和不同 NUMA node 的 CPU 绑定条件下的测试结果如表 4, 采用 PyTorch 和 Transformer 使用 wmt14_en_de 数据集在 GPU 与同 NUMA node 和不同 NUMA node 的 CPU 绑定条件下测试结果如表 5 所示.

表 4 在 ILSVRC 2012 数据集上采用 TensorFlow 和 ResNet50 的 Throughput 测试结果 (image/s)

CPU数目	GPU数目=1	GPU数目=2
无限制 (benchmark)	860	1 715
4, 同NUMA内绑定	861	1 712
4, 不同NUMA内绑定	861	1 710
8, 同NUMA内绑定	861	1 714

表 5 在 wmt14_en_de 数据集上采用 PyTorch 和 Transformer 的 Throughput 测试结果 (image/s)

CPU数目	GPU数目=1	GPU数目=2
无限制 (benchmark)	11.16	9.79/GPU
4, 同NUMA内绑定	11.37	9.82/GPU
4, 不同NUMA内绑定	11.36	9.81/GPU
8, 同NUMA内绑定	11.38	9.88/GPU

从表 4 和表 5 可以看出采用和 CPU 相同的 NUMA 的 GPU0 和 GPU1 进行测试时, NUMA 绑定在使用 GPU 数量不多的情况下训练方面提升有限, 多 GPU 训练时绑定会有一定提升, 且同 NUMA 内 GPU、CPU、内存配合使用效果最好, 同时 CPU 和内存也不要跨 NUMA node 使用.

4.3 数据集缓存和亲和性调度实验

针对数据集缓存和亲和性调度采用海量小文件的训练场景, 即使用 ILSVRC 2012 数据集的原生 JPEG 格式数据和 ResNet50 模型进行训练性能测试, 具体实验信息如下:

实验环境:

GPU: A100 40 GB;

训练框架、模型: PyTorch, ResNet50;

数据集: ILSVRC 2012 数据集 (JPEG 格式);

存储系统: Lustre;

网络: 100 Gb InfiniBand 网络.

实验内容:

(1) 采用 Horovod 分布式训练框架[32], PyTorch 作为后端进行训练, ResNet50 使用 ILSVRC 2012 数据集进行测试, batch_size=256, 训练数据集读取方式采用网络读取存储系统中的文件方式和节点缓存模式进行对比.

(2) 采用 PyTorch 框架的 DistributedDataParallel (DDP)[33] 方式进行训练, ResNet50 使用 ILSVRC 2012 数据集进行测试, batch_size=256, 训练数据集读取方式采用网络读取存储系统中的文件方式和节点缓存模式进行对比.

采用 Horovod 和 ResNet50 使用 ILSVRC 2012 数据集在数据集读取方式采用网络读取存储系统方式和节点缓存模式进行训练的对比结果如表 6 所示, 采用 PyTorch 的 DDP 和 ResNet50使用 ILSVRC 2012 数据集在训练数据集读取方式采用网络读取存储系统方式和节点缓存模式进行训练的对比结果如表 7 所示.

从表 6 和表 7 数据可以看出在节点缓存模式下无论是使用 Horovod 还是 PyTorch-DDP 方式进行训练, 其训练效果均要比直接使用网络读取存储系统中的训练数据集好, 并且在多 GPU 任务下的表现更加明显. 直接使用存储系统中文件由于在网络、CPU 和 GPU 之前的数据拷贝操作会使 GPU 计算资源处于闲置状态, 因此会影响训练效果, 降低训练速度, 图 12 和图 13 展示了在以上两种测试中 GPU 使用情况, 其中图 12(a) 为使用 2 个 GPU 训练的情况, 图 12(b) 和图 13 为使用 8 个 GPU 训练的情况. 可以看出节点缓存模式下训练框架较充分使用了 GPU 计算资源, 使得 GPU 一直处于比较平均且正常的使用率, 在 PyTorch-DDP 方式下尤为明显, 而通过网络使用存储系统的情况下可以看到 GPU 使用有相当的空置状态, 此种情况会在高并发读取数据情况下变得尤为突出, 如分布式训练等.由此可见, 采用节点缓存方式能够有效减小网络上的传输压力并同时提高训练效果.

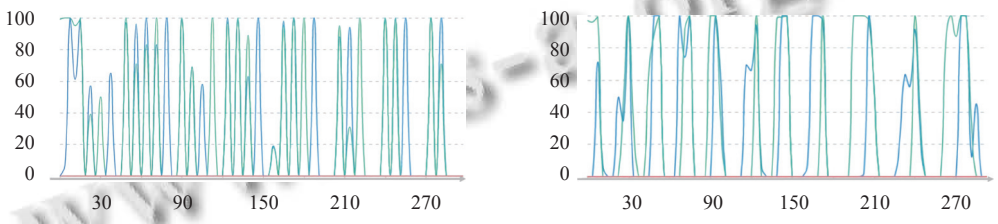
实际应用环境中配合使用数据集调度策略可以充分利用大规模集群中节点上已缓存的数据资源, 进而提高训练性能和集群计算资源的利用率.

表 6 在 ILSVRC 2012 数据集上采用 Horovod 和 ResNet50 的测试结果

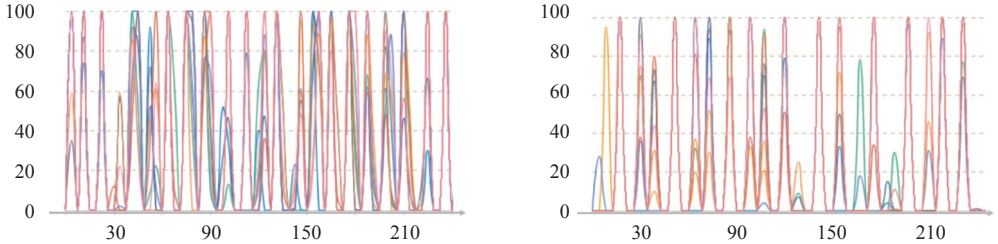
GPU数目	直接使用存储系统中文件	使用节点缓存模式
2	3.20 s/iteration	2.30 s/iteration
8	3.05 s/iteration	1.87 s/iteration

表 7 在 ILSVRC 2012 数据集上采用 PyTorch 的 DDP 和 ResNet50 的测试结果

GPU数目	直接使用存储系统中文件	使用节点缓存模式
8	867.61 images/s	1 017.96 images/s

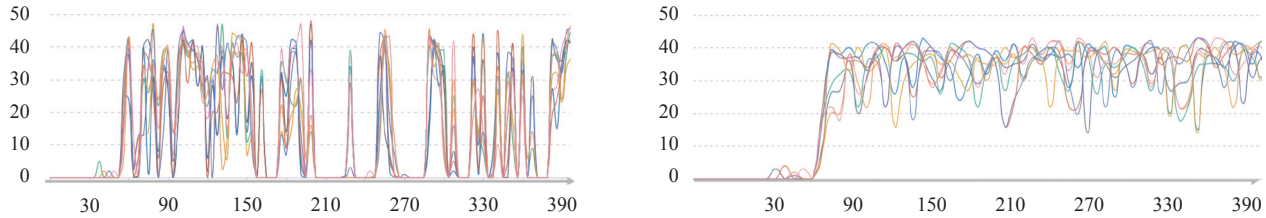


(a) 使用 2 个 GPU 训练



(b) 使用 8 个 GPU 训练

图 12 在 ILSVRC 2012 数据集上采用 Horovod-PyTorch 和 ResNet50 的 GPU 利用率 (纵轴为 GPU 利用率 (%), 横轴为时间 (s))
左: 直接使用存储系统; 右: 节点缓存模式



(a) 直接使用存储系统 (b) 节点缓存模式

图 13 在 ILSVRC 2012 数据集上使用 PyTorch-DDP 和 ResNet50 的 GPU 利用率 (纵轴为 GPU 利用率 (%), 横轴为时间 (s))

5 结束语

本文针对在 Kubernetes 集群上部署深度学习应用所遇到的一些问题, 对数据、计算方面提出了一系列优化方案和设计, 并结合实际场景进行了测试, 整体上达到了预期效果. 其中数据集缓存和亲和性调度能够极大的减少由于存储系统和网络环境限制带来的数据读取速率慢问题; vCUDA 技术能够解决部分场景下的 GPU 共享要求, 并且利用 UM 机制扩大显存使用; 另外 NUMA 亲和性调度和针对海量小文件的优化技巧在实际测试和生产场景中均能够提供可观的整体性能提升.

参考文献

1 Gu J, Song SB, Li Y, *et al*. GaiaGPU: Sharing GPUs in container clouds. Proceedings of 2018 IEEE International Conference on Parallel & Distributed Processing with Applications, Ubiquitous Computing & Communications, Big Data & Cloud Computing, Social Computing & Networking, Sustainable Computing & Communications. Melbourne: IEEE, 2018. 469–476. [doi: 10.1109/BDCloud.2018.00077]

2 Song SB, Deng LL, Gong J, *et al*. Gaia scheduler: A Kubernetes-based scheduler framework. Proceedings of 2018 IEEE International Conference on Parallel & Distributed Processing with Applications, Ubiquitous Computing & Communications, Big Data & Cloud Computing, Social Computing & Networking, Sustainable Computing & Communications. Melbourne: IEEE, 2018. 252–259. [doi: 10.1109/BDCloud.2018.00048]

3 NVIDIA. Multi-process service. https://docs.nvidia.com/deploy/mps/index.html. (2021-07-01).

4 Wu H, Liu W, Gong YF, *et al*. Safe process quitting for GPU multi-process service (MPS). Proceedings of the 2020 IEEE 40th International Conference on Distributed Computing Systems. Singapore: IEEE, 2020. 1169–1170. [doi: 10.1109/ICDCS47774.2020.00125]

5 NVIDIA. Virtual GPU (vGPU). https://www.nvidia.com/en-us/data-center/graphics-cards-for-virtualization/. (2021-09-01).

6 Manchanda N, Anand K. Non-uniform memory access (NUMA). http://www.cs.nyu.edu/~lerner/spring10/projects/NUMA.pdf. (2013-12-28).

7 Sandberg R, Golgberg D, Kleiman S, *et al*. Design and implementation of the Sun network filesystem. Proceedings of the Summer 1985 USENIX Conference. USENIX, 1985. 119–130.

8 Fraunhofer ITWM. BeeGFS: A hardware-independent POSIX parallel file system. https://www.beegfs.io/c/. (2021-09-10).

9 Corbet J. Lustre 1.0 released. https://lwn.net/Articles/63536/. (2003-12-17).

10 Wikipedia. Hadoop distributed file system (HDFS), and a processing part which is a MapReduce programming model. https://en.wikipedia.org/wiki/Apache_Hadoop. (2006-04-01).

11 NetApp and Open Grid Computing. NFS over RDMA. https://www.kernel.org/doc/Documentation/filesystems/nfs/nfs-rdma.txt. (2008-05-29).

12 Wikipedia. Ampere is the codename for a graphics processing unit (GPU) microarchitecture developed by Nvidia. https://en.wikipedia.org/wiki/Ampere_(microarchitecture). (2021-05-14)

13 Herrera A. NVIDIA GRID: Graphics accelerated VDI with the visual performance of a workstation. Nvidia Corporation. (2014-05-01).

14 NVIDIA docker. https://www.nvidia.com/en-us/search/?page=1&q=NVIDIA%20Docker&sort=relevance. (2018-08-23).

15 Kang D, Jun TJ, Kim D, *et al*. ConVGPU: GPU management middleware in container based virtualized environment. Proceedings of 2017 IEEE International Conference on Cluster Computing. Honolulu: IEEE, 2017. 301–309.

16 NVIDIA. Unified memory for CUDA beginners. https://

developer.nvidia.com/blog/unified-memory-cuda-beginners/. (2021-07-01).
17 NVIDIA. NVIDIA management library. https://developer.nvidia.com/nvidia-management-library-nvml. (2021-07-30).
18 Wikipedia. Field-programmable gate array. https://en.wikipedia.org/wiki/Field-programmable_gate_array. (2021-11-01).
19 Wikipedia. Pascal is the codename for a GPU microarchitecture developed by Nvidia. https://en.wikipedia.org/wiki/Pascal_(microarchitecture). (2021-08-01).
20 Wikipedia. NVLink is a wire-based serial multi-lane near-range communications link developed by Nvidia. https://en.wikipedia.org/wiki/NVLink. (2021-08-01).
21 Redmon J, Farhadi A. YOLOv3: An incremental improvement. arXiv: 1804.02767, 2018.
22 He KM, Zhang XY, Ren SQ, *et al*. Deep residual learning for image recognition. Proceedings of 2016 IEEE Conference on Computer Vision and Pattern Recognition. Las Vegas: IEEE, 2016. 770–778 .
23 Devlin J, Chang MW, Lee K, *et al*. BERT: Pre-training of deep bidirectional transformers for language understanding. Proceedings of 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies. Minneapolis: Association for Computational Linguistics, 2019. 4171–4186.
24 Microsoft. COCO is a large-scale object detection, segmentation, and captioning dataset. https://cocodataset.org. (2021-09-30).
25 Stanford Vision Lab, Stanford University, Princeton University. ImageNet is an image database organized according to the WordNet hierarchy. https://www.image-net.org/challenges/LSVRC/. (2021-07-30).
26 Stanford University. The Stanford question answering dataset. https://rajpurkar.github.io/SQuAD-explorer/. (2021-07-30).
27 Google. TensorFlow is an end-to-end open source platform for machine learning. https://www.tensorflow.org. (2021-07-30).
28 Joseph R. Darknet is an open source neural network framework written in C and CUDA. https://pjreddie.com/darknet/. (2018-12-30).
29 Facebook. An open source machine learning framework that accelerates the path from research prototyping to production deployment. https://pytorch.org. (2021-07-30).
30 Vaswani A, Shazeer N, Parmar N, *et al*. Attention is all you need. Proceedings of the 31st International Conference on Neural Information Processing Systems. Long Beach: Curran Associates Inc., 2017. 6000–6010.
31 ACL 2014 9th Workshop on Statistical Machine Translation. Machine translation on WMT2014 English-German. https://paperswithcode.com/sota/machine-translation-on-wmt2014-english-german. (2014-06-01).
32 Uber. Horovod is a distributed deep learning training framework for TensorFlow, Keras, PyTorch, and Apache MXNet. https://horovod.ai/. (2021-07-30).
33 Facebook. DistributedDataParallel (DDP). https://pytorch.org/tutorials/intermediate/ddp_tutorial.html. (2021-10-30).

(校对责编: 孙君艳)