基于分布式缓存加速容器化深度学习的优化方法

张凯，车漾
阿里巴巴科技（北京）有限公司，北京 100102

摘要
使用GPU运行容器化深度学习模型训练任务，性能往往受限于数据加载和预处理效率。很多GPU计算资源浪费在等待从远程存储服务读取数据的过程中。首先介绍了基于容器和分布式缓存技术加速深度学习训练的方法，以及使用Alluxio和Kubernetes实现的系统架构和初步优化手段；然后阐述了TDCS及其训练任务与缓存数据互感知的协同调度策略；接着在Kubernetes容器集群中实现了TDCS，增强了分布式缓存加速大规模深度学习训练的可扩展性；最后用ResNet50图像分类模型训练任务进行性能验证。实验结果表明，相较于直接从远程存储服务中读取数据，TDCS可对运行在128块NVIDIA V100 GPU设备上的分布式训练任务实现2~3倍加速。

关键词
深度学习；分布式缓存；协同调度；Alluxio；容器

中图分类号：TP311　　文献标识码：A　　doi: 10.11959/j.issn.2096-0271.2021054

Method of accelerating deep learning with optimized distributed cache in containers

ZHANG Kai, CHE Yang
Alibaba Technology (Beijing) Co., Ltd., Beijing 100102, China

Abstract
When using GPU to train deep learning models with large-scale dataset, the data loading and preprocessing stages often decrease overall performance notably. Lots of GPU computing resources are wasted on waiting for loading data from remote storage. Firstly, the methods of accelerating deep learning training with container and distributed cache were introduced. The architecture and initial optimization of such training system, which was implemented with Alluxio and Kubernetes, were introduced as well. Secondly, the task and data co-located scheduling (TDCS) and the co-located scheduling policy were elaborated. Thirdly, TDCS was implemented in Kubernetes cluster, which made the acceleration result more extensible. Finally, the result of training ResNet50 image classification model on 128 NVIDIA

V100 GPU devices demonstrates that the proposed methods can bring 2 to 3 times speed up comparing with load data from remote storage directly.

Key words
deep learning, distributed cache, co-located scheduling, Alluxio, container

1 引言

最近10年，深度学习技术和应用发展风起云涌，百花齐放。尤其在机器视觉、自然语言理解、语音识别等领域，一大批成熟的人工智能模型和服务正落地应用于各行各业。越来越多的行业发掘出大量需求，需要利用企业日积月累的海量业务数据，高效便捷地训练深度学习和机器学习模型。

与此同时，云计算在IaaS、PaaS和SaaS等层次已经发展得相当成熟，并逐步进入云原生时代。在云原生技术体系中，以Docker为代表的Linux容器技术和以Kubernetes为代表的容器编排管理技术在应用开发、交付、运维的生命周期中逐渐成为业界的事实标准。同时，应用微服务架构的自动化部署，运行时动态弹性伸缩，完善的日志、监控、告警等运维体系等，都成为现代软件开发运维的基础需求。

深度学习和人工智能应用往往需要使用大量GPU、张量处理器（tensor processing unit，TPU），甚至神经元处理器（neural processing unit，NPU）这类定制的高性能异构计算设备来迭代处理海量的训练数据，最终得到满足准确率和性能要求的模型。由于数据规模较大、模型复杂度较高，训练过程一般较漫长。因此，优化GPU等异构计算设备的利用效率、加快训练数据处理速度，成为提升深度学习和人工智能应用生产率和成本优化的关键因素。

Kubernetes容器编排管理技术对于GPU、TPU、NPU等高性能异构计算设备资源的支持日趋成熟。考虑到云计算在成本优化、资源规模和弹性伸缩方面的优势，配合Docker容器技术擅长的轻量级和标准化应用交付能力，业界主流的生产人工智能模型和应用的技术方案会采用容器集群结合云计算平台的GPU资源的架构，进行分布式深度学习模型训练。

在“云上”执行训练任务，通常会在带有GPU设备的计算服务节点上进行。训练数据会被存放在各种异构的云存储服务中，如对象存储服务、远程文件系统服务等。这种典型的计算和存储分离的分布式计算架构有利于增强计算和存储的弹性扩展能力，但同时也会带来数据访问时延增大的问题。如果考虑数据敏感性、隐私保护和安全合规的要求，训练数据通常也会被存放在私有数据中心中。这样势必要求深度学习模型训练平台采用混合云的架构，统一使用“云上”计算资源来访问“云下”数据。而这种“本地存储+云上计算”的训练模式进一步增大了计算和存储分离架构带来的远程数据访问时延，进而导致深度学习模型训练整体性能的显著下降。

本文首先概述深度学习训练系统中

典型的数据访问方法；然后提出基于容器和分布式缓存技术的深度学习训练系统架构，以及训练任务与缓存数据互感知的协同调度方法TDCS（task and data co-located scheduling）；之后介绍使用Alluxio加速训练的实验结果和性能瓶颈，并针对Alluxio进行参数调优，获得了初步性能提升；笔者还提出TDCS任务与数据协同调度方法，并给出TDCS的具体策略设计和实现细节；最后提供多组实验数据验证TDCS的效果，实验表明TDCS有效增强了容器环境下分布式缓存系统的可扩展性，可以显著加速深度学习训练。

2 访问训练数据的典型方法

为了规避计算和存储分离架构带来的数据访问时延，典型做法是在开始模型训练之前，将训练数据复制到GPU计算节点本地的磁盘存储中，如普通机械硬盘或者NVMe SSD等高速存储设备；或者将数据提前复制到部署在计算节点上的分布式存储系统中，如Ceph、GlusterFS。随后计算任务就相当于从本地读取训练数据，有效缓解了数据访问时延对GPU计算性能的影响。然而，这种额外的数据迁移过程会面临如下问题。

- 把训练数据复制到GPU节点的方式低效且难以管理。如果是手动复制，非常容易出错。
- 深度学习训练数量很大，且可能持续增加。“云上”GPU计算节点配置的磁盘容量有限，经常出现无法存放全量训练数据的情况。
- 将训练数据存放在多个GPU计算节点上的分布式存储系统内，可以解决数据容量问题，但分布式存储系统自身的运维成本和难度都很大；并且，存储系统与计算节点耦合，本身也会产生计算、网络、I/O等本地资源的争抢和干扰问题。

3 基于分布式缓存的容器化深度学习模型训练系统

业界已经有一些工作通过基于缓存机制的方法[1-3]来解决上述复制训练数据到计算节点方案存在的问题。与这些工作不同，本文设计并实现了一种Kubernetes容器编排技术和分布式缓存技术相结合的深度学习模型训练系统。该系统架构可以满足云计算环境下大规模容器化深度学习系统对训练任务的加速要求。该系统的基础架构如**图1**所示。

基于容器和分布式缓存的深度学习模型训练系统架构的核心模块包括如下几个。

- Kubernetes。Kubernetes是业界非常主流的容器集群管理和编排引擎。其系统架构和API已经成为容器平台的事实标准。Kubernetes非常广泛地用于构建深度学习训练平台。它提供了通过容器兼容不同深度学习计算框架的灵活性，动态调度训练任务和按需扩展GPU资源的能力，以及完善的集群管理能力。Kubernetes中最重要的子模块是Kube-scheduler。该子模块负责在Kubernetes容器集群中统一调度深度学习训练任务和分布式缓存系统。本文通过扩展Kube-scheduler，实现了TDCS方法。
- Kubeflow。Kubeflow是Kubernetes技术生态中最流行的支持机器学习和深度学习任务的开源项目。Kubeflow的目标是使机器学习、深度学习的完整工作流程在Kubernetes集群上能够简便地部署、高效地运行，并且使整个工作流可移植和可扩展。Kubeflow为主流的深度学习框架TensorFlow[4-5]的分布式训练提供了两种

支持模式，分别是参数服务器模式和Ring Allreduce[6]模式。Kubeflow同时支持训练任务使用CPU和GPU运行。

● 分布式缓存系统。分布式缓存系统设计的初衷是解决大数据处理流水线中，不同计算框架在通过磁盘文件系统（如HDFS）交换数据时，系统I/O操作方面的瓶颈[7]。业界已存在一些用于实现分布式缓存系统的技术。本文设计的深度学习训练系统选择使用Alluxio作为分布式缓存技术的具体实现。Alluxio是面向混合云环境的开源数据编排与缓存系统[8]。它利用多个计算节点本地的内存和磁盘资源构成统一的分布式缓存系统，从而为计算任务提供就近、可重用的数据访问服务。

为了提升深度学习的训练性能，本文先对Alluxio进行系统参数调优，获得初步的训练加速效果。为了继续将缓存加速效果扩展到更大规模的训练场景，本文结合Kubernetes容器集群的调度和弹性伸缩能力、Alluxio缓存引擎的部署和运行机制，提出了一种训练任务与缓存数据互感知的协同调度方法TDCS，并在容器化深度学习系统中进行实现。

TDCS通过多层资源分配策略来感知任务与数据间的关联性，从而实现训练任务和缓存数据的双向亲和性调度。通过感知任务运行时拓扑，实现弹性伸缩缓存容量和数据分布策略。在经典的图像分类模型训练实验中，TDCS可以实现2~3倍的训练加速，并且随着计算资源的增加，TDCS能够持续加速训练任务。本文对使用128块GPU卡的训练任务进行实验，发现TDCS依然带来了可观的性能提升。

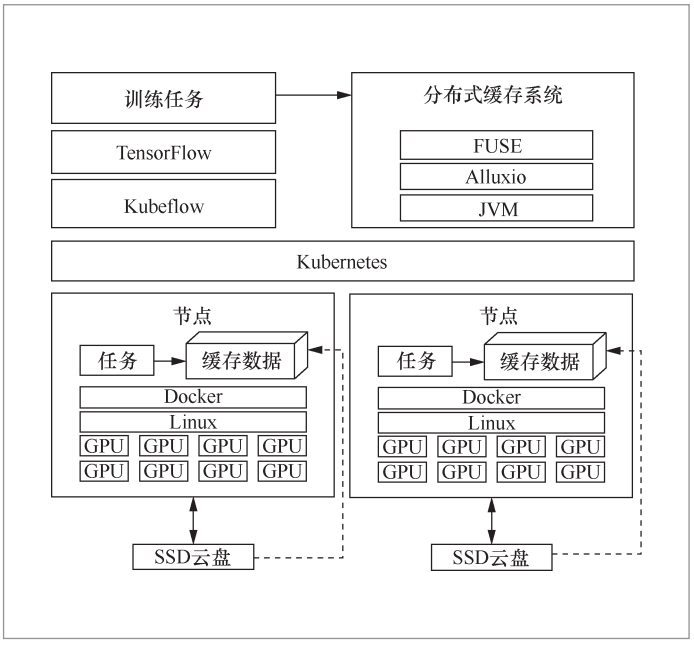

图1 基于容器和分布式缓存的深度学习模型训练系统的基础架构

4 初步优化分布式缓存加速训练

使用分布式缓存加速诸如深度学习的大规模数据处理任务，在获得性能提升的同时，会产生其他方面的问题，如整体成本的上升[9]、加速效果的波动等。本文实现了基于Kubernetes容器编排技术和Alluxio分布式缓存技术的深度学习训练系统。实验发现，Alluxio分布式缓存可以在一定条件下加速训练任务。但在扩大任务规模、增加参与计算的GPU资源后，缓存加速效果很快达到瓶颈，无法随资源量的增加而持续提升。如果要实现资源量和训练任务性能的线性加速，则需要对缓存系统进行优化。

下面首先介绍使用Alluxio加速深度学习训练任务的实验方法；然后对实验结果进行分析，提出影响加速效果的原因和优化手段，并验证初步优化的结果。

4.1 实验环境

本文使用阿里云托管Kubernetes服

务，部署含有多个GPU节点的Kubernetes容器集群作为实验环境。使用4台服务器，每台服务器上有8块NVIDIA Tesla V100 GPU卡，共提供32块GPU卡的算力。训练程序通过Alluxio提供的POSIX接口读取远程数据存储，并缓存到本地Alluxio缓存中。每台服务器可为Alluxio提供40 GB内存存储，实验环境共提供160 GB分布式缓存空间。

使用图像分类模型ResNet50[10]与图像识别数据集ImageNet[11]进行模型训练实验。ImageNet数据集包含128万余张图片及其标注数据，总数据量约为144 GB。使用TensorFlow作为深度学习训练框架，Horovod[12]作为分布式训练通信框架。每块GPU卡处理训练数据的batch_size为256。训练任务执行标准TensorFlow benchmark测试程序，当达到预定训练轮数时，任务自动停止，同时检查训练所得模型的图像分类精确度以及训练速度（每秒处理图片数）。

4.2 实验结果

对实验中模型精确度达到业界标准情况下的训练速度结果进行评估。分别使用合成数据（即synthetic）和Alluxio缓存系统访问存储服务中的真实数据（即Alluxio）进行模型训练，并对比性能结果，如**图2**所示。横轴表示任务使用GPU卡数量，纵轴表示训练速度。合成数据是训练程序在内存中生成的数据，节省了I/O开销，可作为训练性能的理论上限。

由**图2**可以看出，使用1~2块GPU卡训练时，两者性能差距在可以接受的范围内；但当GPU卡增加到4块时，二者性能差距比较明显；当GPU卡数量达到8块时，使用Alluxio缓存系统访问存储服务中的数据的训练速度下降至使用合成数据的28.77%。通过监控观测到GPU服务器的系统资源利用率并不高，这说明使用Alluxio默认配置实现的分布式缓存系统无法支持GPU训练规模化扩展。这成为深度学习训练平台能力的瓶颈。

通过分析Alluxio的系统架构和实现方法，发现造成无法支撑训练任务性能弹性扩展的主要原因有如下3个。

- Alluxio的文件操作需要进行多次远程过程调用（remote procedure call，RPC）才能获取到数据。
- 默认情况下，Alluxio优先将全量数据缓存在本地节点。当本地缓存容量饱和时，就会驱逐已缓存数据。如果被驱逐数据很快又被任务读回，就会导致同一份数据频繁写入和删除，形成数据震荡。
- 缓存系统提供的用户空间文件系统（filesystem in userspace，FUSE）接口虽然易于使用，但是读写性能并不理想。在深度学习训练任务多线程高并发读写下，需要优化FUSE的配置参数。

4.3 优化Alluxio系统参数

在分析了上述性能瓶颈后，本文首先调整一系列Alluxio系统参数。实验结果表明，Alluxio缓存系统的性能得到一定程度的提升。

4.3.1 优化FUSE文件系统参数

Linux系统上的FUSE实现分为两层，包括运行在用户态的libfuse和运行在内核态的FUSE模块。本文分别进行调优。第一个优化手段是升级Linux内核版本。高版本Linux系统内核对FUSE模块内置了许多优化，比如Linux Kernel 4.19版的FUSE读性能比3.10版提升20%。第二个优化手段是调整libfuse的参数，包括如下

两种。

- 增大参数entry_timeout和attr_timeout的值，避免libfuse本地读取文件的inode和dentry信息频繁过期，导致大量读文件操作穿透到Alluxio的远程元数据管理节点。
- 适当调大libfuse线程池参数max_idle_threads的值（默认为10），避免训练任务高并发访问缓存数据时，libfuse可用线程频繁地创建和销毁，引入过多CPU开销。

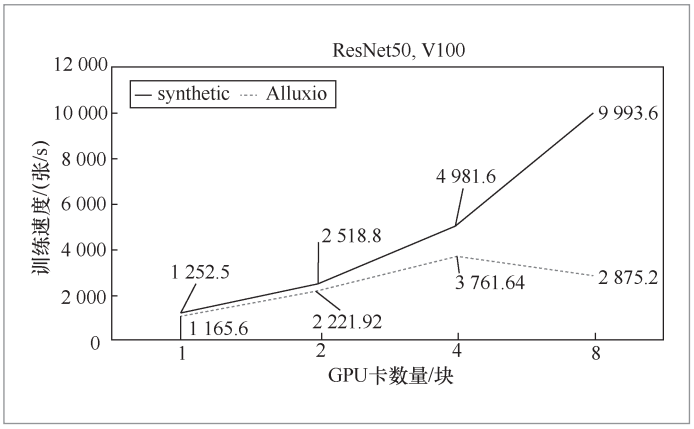

图 2 使用合成数据和 Alluxio 缓存系统访问存储服务中的真实数据训练的速度对比

4.3.2 优化Alluxio参数

深度学习训练任务读取的训练数据量往往很大，并发压力也非常大。本文针对缓存数据驱逐策略、元数据操作效率和容器环境下客户端读取缓存数据的方法，进行多组Alluxio系统参数配置。

第一组是参数优化，避免Alluxio频繁驱逐缓存数据，造成数据震荡，包括如下3种。

- alluxio.user.ufs.block.read.location.policy。默认值为alluxio.client.block.policy.LocalFirstPolicy，表示优先将数据保存在Alluxio客户端节点。本文设置为alluxio.client.block.policy.LocalFirstAvoidEvictionPolicy。同时设置alluxio.user.block.avoid.eviction.policy.reserved.size.bytes参数，保证在节点本地固定保留适量缓存数据不被驱逐。
- alluxio.user.file.passive.cache.enabled。默认值为true，导致Alluxio节点额外复制已经在其他节点上缓存过的数据。本文将该属性置为false，避免多余的本地缓存副本。
- alluxio.user.file.readtype.default。默认值为CACHE_PROMOTE，会导致缓存数据在节点的各存储层间（如内存与磁盘）迁移。本文将该参数值改为CACHE，避免数据迁移和数据加锁开销。

第二组也是参数优化，配置Alluxio客户端缓存远程文件元数据的策略。客户端缓存元数据信息，可避免频繁地对Alluxio服务端进行RPC访问，具体包括如下4种。

- alluxio.user.metadata.cache.enabled，设置为true，表示开启客户端元数据缓存。
- alluxio.user.metadata.cache.max.size，设置客户端可缓存元数据的最大值。
- alluxio.user.metadata.cache.expiration.time，设置客户端元数据缓存的有效时间。
- alluxio.user.worker.list.refresh.interval，设置Alluxio服务端同步所有缓存节点状态的时间间隔，本文设为2 min。

第三组是配置容器本地访问Alluxio缓存系统的方法。容器环境下Alluxio支持两种短路读写方式来实现本地访问缓存数据，包括UNIX Socket和直接文件访问。本文使用直接文件访问方式，使Alluxio客户端读取本地缓存节点的性能更优。

4.3.3 优化Alluxio容器中的JVM参数

通过Kubernetes部署Java应用容器，如果不指定容器请求的CPU资源量，容器内Java虚拟机（Java virtual machine，JVM）识别到的可用CPU核数可能有误，进而影响容器内的Alluxio可用线程数。本文将JVM参数-XX:ActiveProcessorCount设置为容器内Alluxio进程可使用的CPU资源数量。

本文还调整了JVM垃圾回收（JVM garbage collection，JVM GC）和JIT（just in time）相关参数，包括-XX:ParallelGCThreads、-XX:ConcGCThreads和-XX:CICompilerCount，将其设置为较小值，避免线程频繁切换，影响Alluxio进程的性能。

4.4 优化结果分析

优化上述系统参数后，在一台配置8块GPU卡的服务器上再次训练ResNet50模型。如**图3**所示，训练速度明显加快，与使用合成数据的训练速度仅相差3.3%左右。并且，这样的加速效果可以接近线性地扩展到4台服务器（共32块GPU卡）的实验中。

然而，实验中发现，继续增加GPU资源后，缓存加速的扩展性瓶颈再次出现。这使得优化Alluxio系统参数的方法只能对使用少于32块GPU卡的ResNet50训练任务产生良好的加速效果。同时，本文前述实验都使用阿里云SSD云盘作为数据存储。虽然SSD云盘可以提供300 MB/s以上的I/O吞吐性能，但是单块云盘容量有限，

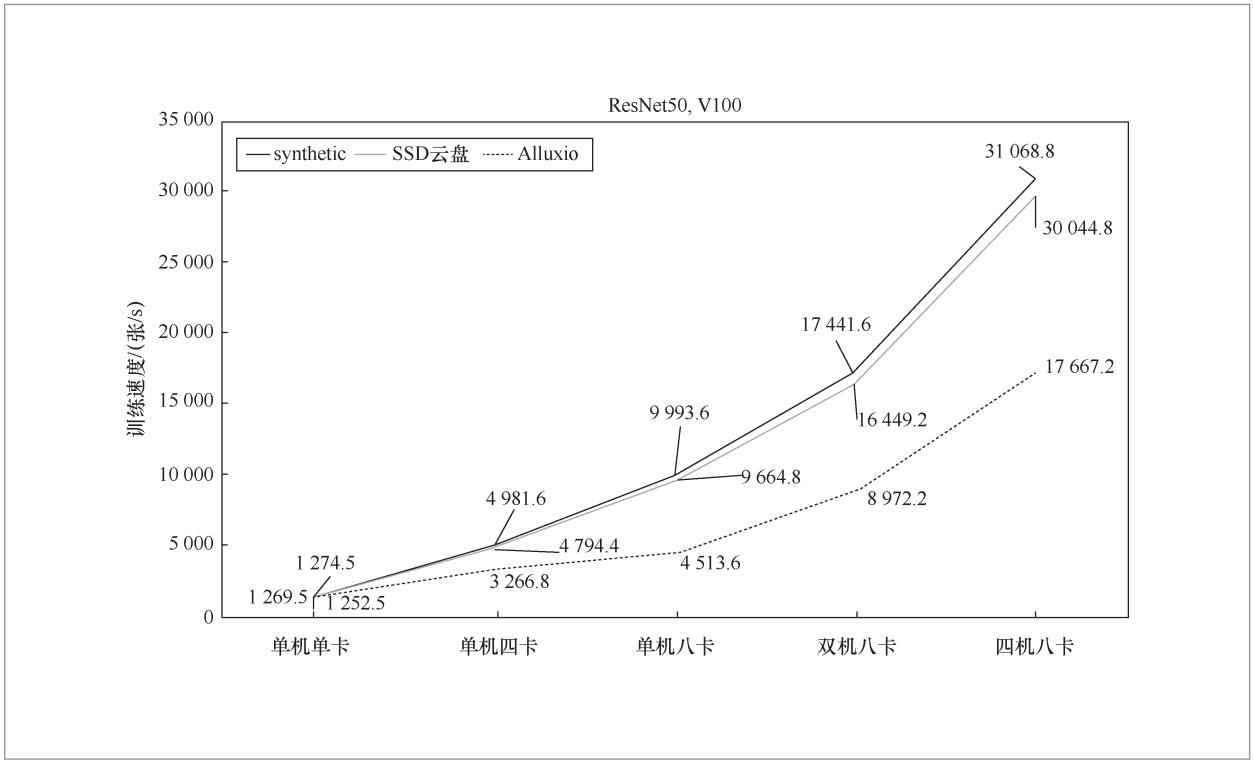

图 3 优化分布式缓存系统加速分布式训练的实验结果

且成本较高。如果在更大量级的训练数据集上执行更大规模的分布式训练任务，SSD云盘并不是最佳选择。实际上，在大规模分布式训练场景中，使用具有较高性价比的阿里云对象存储服务（object store service，OSS）是更常用的方案。本文后续实验也将采用OSS作为训练数据集的存储方案。

5 任务与数据协同调度——TDCS

目前，典型分布式缓存技术实现多用静态架构进行部署，比如Alluxio。在初始化部署之后，缓存系统的拓扑和容量就固定了。在运行过程中，Alluxio不会主动对缓存数据的访问频率和系统资源的使用情况进行动态扩展，无法降低系统压力。Alluxio也不会感知上层访问应用，无法做到根据不同应用状态和数据访问特点同步调整缓存系统的部署拓扑和缓存策略。因此，在前面的实验中，尽管优化Alluxio系统的参数后，可以加速最多使用32块GPU卡的训练任务，但是，增加更多GPU卡参与计算后，受到缓存系统静态部署的限制，以及训练任务运行拓扑动态变化的干扰，加速效果再次遇到瓶颈。

经过分析深度学习训练任务的特点发现，一方面，在单个任务运行周期内以及多个任务运行过程之间，都存在计算与数据的时间相关性；另一方面，对于一个训练任务使用的计算资源，可能会按需进行弹性伸缩。例如，当集群资源空闲时，可通过扩展利用更多GPU资源，加快训练任务的整体进程；当集群资源紧张时，对低优先级任务进行资源回收。分布式缓存系统会优先部署在计算发生的本地节点上，通过就近读取，节省大量远程数据I/O损耗。这使得计算与缓存数据之间还存在空间相关性。

利用计算与缓存数据的时间和空间相关性，本文设计了深度学习训练任务与分布式缓存数据互相感知的协同调度方法TDCS。TDCS分别针对单任务、多任务和弹性伸缩任务，实现了多维度的统一管理和调度策略，具体包括训练数据预加载策略、多任务缓存数据共享策略、分布式缓存弹性策略等。笔者在前文所述基于Kubernetes容器和分布式缓存的深度学习模型训练系统之上，实现了TDCS方法。通过增加预加载控制器Preloader、缓存数据管理器CacheManager、弹性伸缩控制器CacheScaler等组件，完成训练任务和分布式缓存系统的统一管理和调度。支持TDCS方法的完整系统架构如**图4**所示。

5.1 训练数据预加载策略

对于单个训练任务，通常以多轮迭代方式执行。训练任务在给定数据集上循环执行算法，不断调整参数，使得算法提取和比对特征的准确率最终达到目标。这种迭代执行具有明显的时序特征。由于每轮迭代执行都会读取全量训练数据，各轮之间会有大量数据需重复计算。因此，通过分布式缓存，将尽可能多的训练数据保存在计算节点本地，之后就可以避免每轮计算都反复从远程存储系统读取数据造成的损耗。然而，对于首次执行的任务，还存在“冷启动”问题。第一轮迭代周期内尚未有任何数据被缓存，执行计算的同时必须从远程存储系统中拉取所有训练数据，性能明显低于后续迭代。

针对“冷启动”，TDCS设计实现了训练数据集Preloader，负责在任务计算开始之前，预先读取部分或者全部训练

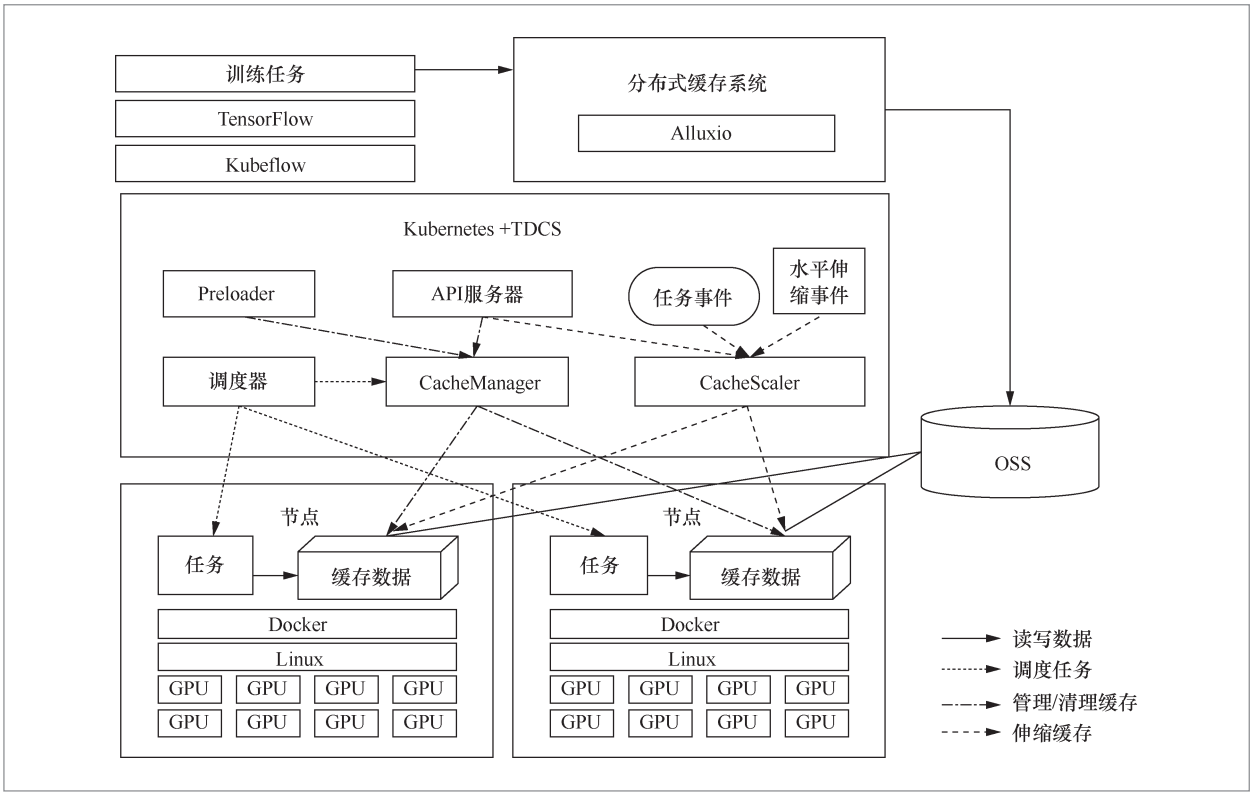

图 4 使用 TDCS 分布式缓存的容器化深度学习训练系统架构

数据进入缓存系统。TDCS允许使用者指定是否启用预加载功能，以及预加载的具体策略。策略包括启动预加载的时间、预加载数据量或其占整个训练数据集的比例，以及预加载特定部分的训练数据（如仅加载远程文件系统中特定路径下的数据）。

训练平台的用户可以提交针对某个训练数据集的预加载指令，Preloader响应并依据预加载策略，在计算启动之前就开始将部分或者全部训练数据拉取到分布式缓存系统中。预加载数据过程与用户提交的训练任务的执行是解耦的，可以完全串行，也可以部分并行。TDCS通过Preloader解决了单个训练任务的“冷启动”问题，保证整个训练过程没有明显的读取数据时延。

5.2 多任务缓存数据共享策略

深度学习在很多领域取得了非常好的效果，如图像分类、物体检测、自然语言理解等领域。一个领域内经常会有多种深度学习算法在演进。而训练同一类领域模型的多种算法经常会使用同样的数据集。比如，在视觉识别领域，几乎所有深度学习算法会将ImageNet作为训练数据的基线。对于同一种算法模型，也需要多次提交任务，使用同一份训练数据反复实验。依据每个任务的实验结果，不断调整模型结构和参数，最终得到表现稳定的模型版本。因此，在深度学习训练平台上，会有很多任务重复使用相同数据进行训练。根据这些任务提交的时序特征，可在

任务之间共享缓存数据，从而提升多个相关任务的性能。

TDCS利用多任务间的时序特征，设计了任务与缓存数据的亲和性调度策略，即把后序提交的任务分配到已缓存其目标数据的计算节点上，保证后序任务可以直接重用本地缓存数据。具体地，如果在t_i时刻提交任务T_1，使用数据集D_1训练。假设训练平台调度器Scheduler为T_1分配了计算节点集合$S_1=\{N_1,N_2,N_3\}$，缓存系统会在节点集合S_1上保存D_1的缓存D_1'。在t_j（$j \geqslant i$）时刻，提交任务T_2，也使用数据集D_1进行训练。

TDCS的任务调度器Scheduler为T_2分配计算节点的算法流程如下。这样可以最大限度地保证T_2计算与本地缓存数据的亲和性。

- Scheduler查询CacheManager后，得知集群中已存在数据集D_1的缓存数据D_1'，且D_1'部署在节点集合S_1上。
- Scheduler根据T_2的计算资源要求，在集群中找到符合执行任务条件的备选节点集合S_2，假设为$\{M_1, M_2, M_3, M_4, M_5\}$。
- Scheduler计算集合S_1与S_2的交集，得到高优先级备选节点集合S_3。
- Scheduler使用贪心算法，总是更多地选择S_3中的节点，并将其分配给任务T_2。

在实现多任务共享缓存数据调度策略时，会存在“缓存热点”和“缓存清理”问题。

- 缓存热点：考虑到在特定时期有大量类似T_2的任务需要几乎同时执行，如果多个任务同时访问缓存数据D_1'，会导致S_1内节点的访问压力过大，使这些节点成为“热点”。
- 缓存清理：考虑到计算节点本地内存和磁盘存储容量有限，缓存数据D_1'势必会从S_1节点上被逐步驱逐，或者彻底删除。任务T_2的提交时刻t_j不确定，导致缓存数据的利用率不稳定。极端情况下，每次T_2被提交的时候，D_1'都已被删除，缓存完全无法被多任务共享。

为了解决这两个问题，TDCS设计实现了缓存数据管理器CacheManager，负责根据用户指定策略，管理对缓存数据的访问和清理缓存数据。

首先，CacheManager控制缓存数据D_1'的并发访问数上限。用户可以选择配置应对并发任务数超过限制的策略。不同于参考文献[13]提出的优化缓存系统内部数据分布算法的方法，以及参考文献[14]提出的利用存储系统中数据块间的关联性指导Alluxio缓存策略CPR（cache prefet repace），TDCS避免侵入修改分布式缓存引擎，而选择在其外部增加控制访问缓存任务量和缓存供给量的逻辑，实现缓存负载稳定和均衡。由CacheManager与Scheduler配合，动态调整超限并发任务的资源分配，以及缓存数据副本数量。为此，CacheManager提供了Suspend和Clone两种策略。

- Suspend：表示暂时挂起新任务对D_1'的访问。此时，CacheManager会通知Scheduler，将超限任务放入等待队列。当并发访问D_1'的任务数量降到上限以内时，CacheManager会通知Scheduler将等待队列中的任务移出队伍，将其尽量调度到D_1'所在的节点上。
- Clone：表示CacheManager在集群中复制一份新的缓存数据D_1''，供新任务读取。Scheduler查询CacheManager后，得到D_1存在可被访问的缓存数据D_1''及其所处节点集合S_i。根据最大限度地保证计算与缓存数据亲和性的原则，新任务会被尽可能多地调度到S_i的节点上，避免出现缓存热点。

其次，CacheManager支持3种缓存清理策略，包括：锁定资源，不清理缓存；

定时清理；最近最少使用（least recently used，LRU）清理。

用户可以根据计算节点的可用存储资源量、任务提交计划、多任务的相关性等因素，配置清理策略。例如，在特定时段，存在许多使用相同训练数据的任务需要执行，为了保证总是有缓存数据用于加速训练计算读取，可以选择不清理缓存，也可以选择定时清理，具体时间间隔可以根据任务执行计划和可用存储资源量来设置。而对于集群中存在多个训练数据集的缓存情况，可以选择CacheManager按照LRU算法，优先清理最近最少访问的缓存数据。还可以扩展CacheManager模块，以支持更多复杂的缓存清理策略，比如参考文献[15]提出的兼顾最近访问时间和历史访问频率的缓存替换策略等。

5.3 分布式缓存弹性策略

通常训练平台内会有很多任务同时运行，它们共享了集群计算资源，尤其是宝贵的GPU资源。为了提升GPU利用率，训练平台调度器会动态调整分配给任务的GPU资源。例如，回收低优先级任务占用的部分GPU节点，将其分配给高优先级任务。训练过程中，任务需要具有弹性伸缩的能力。为高优先级任务动态分配更多计算节点时，若将分布式缓存系统弹性扩展到这些新增节点上，则可以扩大缓存总容量，进而支撑更大规模的并发读取，提升任务整体的训练性能。

TDCS设计了CacheScaler，负责弹性扩展、收缩分布式缓存系统，使缓存系统的部署位置与训练任务的运行节点保持同步，保证计算所需缓存数据总是在本地就可读取。CacheScaler可通过如下3种触发方法来弹性伸缩分布式缓存系统。

- API触发伸缩活动。用户可以调用CacheScaler的API，主动触发扩展和收缩缓存。
- 事件触发伸缩活动。CacheScaler通过事件订阅机制，发现、响应训练任务的计算资源量变化事件，自动触发扩展和收缩缓存。
- 监控触发伸缩活动。本文的训练平台基于Kubernetes容器编排和管理技术构建。利用集群监控系统，根据监控指标的变化情况，调用Kubernetes的HPA（horizontal pod autoscaler）组件，主动伸缩缓存系统。典型的监控指标是缓存系统容量使用率。当缓存系统容量使用率（即缓存数据量的占比）达到给定阈值时，就会自动触发扩展缓存；反之，当缓存系统容量使用率长期保持低水平时，可以选择通过HPA触发收缩缓存。

TDCS通过CacheScaler、Kubernetes集群监控和HPA等组件支持分布式缓存的弹性策略，使缓存数据能够更灵活地配合训练任务弹性伸缩，既提供了高训练性能，也保证了较高的资源利用率。实验表明，TDCS帮助Alluxio支持从使用8块NVIDIA V100 GPU卡计算的深度学习训练任务扩展到使用128块NVIDIA V100 GPU卡的任务。

6 TDCS实验结果

为了在前文实验基础之上，验证TDCS进一步提升训练任务性能的效果，本文使用相同的ResNet50图像分类模型和ImageNet图像识别数据集，在不同数量的GPU资源上测试了多组分布式训练任务。如**图5**所示，4组实验数据分别使用8、32、64、128块NVIDIA Tesla V100 GPU卡进行分布式训练。每组实验都对比了使用4种不同缓存配置的训练速度。

配置1：不使用缓存，任务直接读取远程OSS数据。

配置2：使用默认配置Alluxio读取OSS数据。

配置3：在配置2的基础上增加TDCS训练数据预加载策略。

配置4：在配置3的基础上增加TDCS多任务缓存数据共享策略。

其中，使用配置3、4的实验中，都使用了TDCS分布式缓存弹性策略。

总体来看，使用TDCS协同调度增强的分布式缓存之后，实验任务的训练速度能达到直接读取OSS数据的任务速度的2.2~3.7倍。在所有实验中，使用默认配置的Alluxio读取训练数据后，都会比不使用缓存的训练速度有大幅提升，加速比范围为25%~200%。在增加使用TDCS训练数据预加载策略后，相比仅使用默认Alluxio的加速比会提升7%~102%。在继续增加使用TDCS多任务共享缓存调度策略后，训练速度依然还有4%~7%的提升。特别地，当使用默认配置的Alluxio加速8块GPU卡上的训练任务时，由于计算资源不足，已经出现瓶颈，继续增加使用TDCS的协同调度策略并不能进一步明显地提升训练性能。

7 结束语

本文针对云环境下，计算、存储分离架构带来的数据读取性能瓶颈问题，提出基于分布式缓存结合Kubernetes容器技术加速深度学习训练的方法和架构。在应用Alluxio分布式缓存系统，并进行Alluxio系统参数优化后，获得初步的训练加速效果。受限于Alluxio静态部署的系统架构，加速效果无法扩展到大规模分布式训练任务，进而提出TDCS训练任务与缓存数据协同调度方法和其在Kubernetes容器集群架构下的系统实现。通过增加训练数据预加载策略、多任务缓存数据共享策略以及分布式缓存弹性策略，TDCS可以保证在容器环境下给深度学习训练任务带来2~3倍的加速，并且加速效果可以扩展到使用128块NVIDIA GPU卡的大规模分布式训练场景。

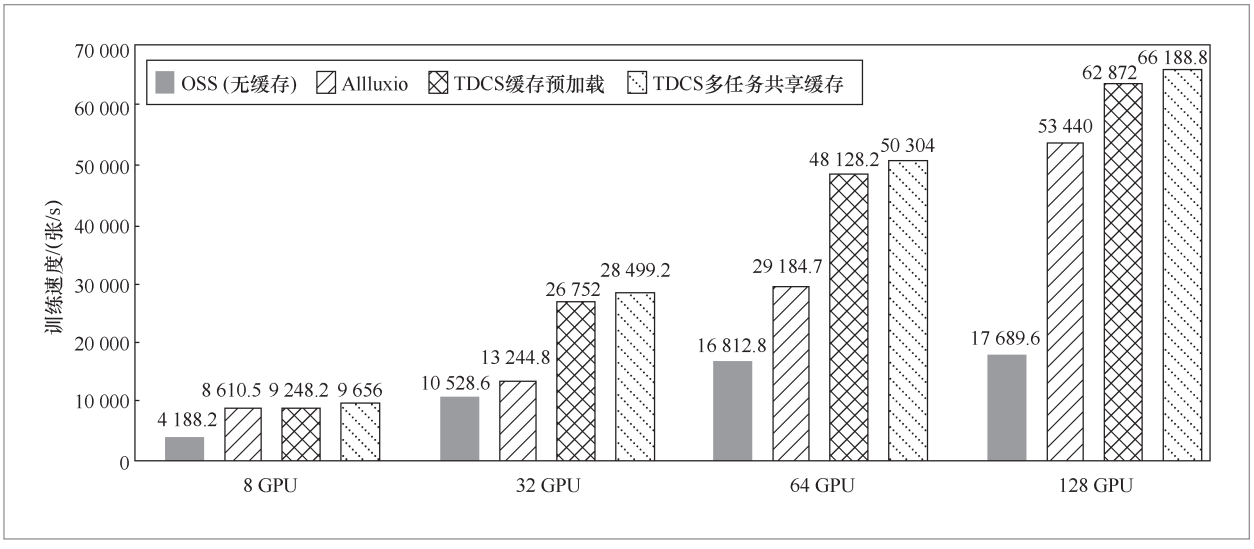

图 5 使用 TDCS 协同调度分布式缓存加速 ResNet50 模型训练的实验结果对比

参考文献:

[1] PINTO C, GKOUFAS Y, REALE A, et al. Hoard: a distributed data caching system to accelerate deep learning training on the cloud[J]. arXiv preprint, 2018, arXiv:1812.00669.

[2] KUMAR A V, SIVATHANU M. Quiver: an informed storage cache for deep learning[C]// Proceedings of the 18th USENIX Conference on File and Storage Technologies (FAST'20). Berkeley: USENIX Association, 2020: 283–296.

[3] WANG L P, YE S G, YANG B C, et al. DIESEL: a dataset-based distributed storage and caching system for large-scale deep learning training[C]// Proceedings of the 49th International Conference on Parallel Processing. New York: ACM Press, 2020: 1–11.

[4] ABADI M, AGARWAL A, BARHAM P, et al. TensorFlow: large-scale machine learning on heterogeneous distributed systems[J]. arXiv preprint, 2016, arXiv: 1603.04467.

[5] ABADI M, BARHAM P, CHEN J M, et al. TensorFlow: a system for large-scale machine learning[J]. arXiv preprint, 2016, arXiv:1605.08695.

[6] PATARASUK P, YUAN X. Bandwidth optimal all-reduce algorithms for clusters of workstations[J]. Journal of Parallel and Distributed Computing, 2009, 69(2): 117–124.

[7] LI Z W, YAN Y L, MO J T, et al. Performance optimization of in-memory file system in distributed storage system[C]// Proceedings of the 2017 International Conference on Networking, Architecture, and Storage. Piscataway: IEEE Press, 2017.

[8] LI H Y, GHODSI A, ZAHARIA M, et al. Tachyon: reliable memory speed storage for cluster computing frameworks[C]// Proceedings of the ACM Symposium on Cloud Computing. New York: ACM Press, 2014: 1–15.

[9] CHANG X, ZHA L. The performance analysis of cache architecture based on Alluxio over virtualized infrastructure[C]// Proceedings of the 2018 IEEE International Parallel and Distributed Processing Symposium Workshops. Piscataway: IEEE Press, 2018: 515–519.

[10] HE K M, ZHANG X Y, REN S Q, et al. Deep residual learning for image recognition[C]// Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition. Piscataway: IEEE Press, 2016.

[11] DENG J, DONG W, SOCHER R, et al. ImageNet: a large-scale hierarchical image database[C]// Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition. Piscataway: IEEE Press, 2009.

[12] SERGEEV A, BALSO M D. Horovod: fast and easy distributed deep learning in TensorFlow[J]. arXiv preprint, 2018, arXiv:1802.05799

[13] LIU Z X, BAI Z H, LIU Z M, et al. DistCache: provable load balancing for large-scale storage systems with distributed caching[C]//Proceedings of the 17th USENIX Conference on File and Storage Technologies. Berkeley: USENIX Association, 2019: 143–157.

[14] DONG W J, WEN D X, ZHANG Z. Optimization of cache strategy based on Alluxio remote scenario[J]. Application Research of Computers, 2018, 35(10): 3025–3028.

[15] 杨青霖, 吴桂勇, 张广艳. 分布式存储系统中的数据高效缓存方法[J]. 大数据, 2021, 7(2): 147–157.
YANG Q L, WU G Y, ZHANG G Y. An approach to buffering data efficiently in distributed storage systems[J]. Big Data Research, 2021, 7(2): 147–157.

作者简介

张凯（1981- ），男，阿里巴巴科技（北京）有限公司高级技术专家，主要研究方向为云计算、容器、深度学习、分布式系统。

车漾（1982- ），男，阿里巴巴科技（北京）有限公司高级技术专家，主要研究方向为云计算、容器、分布式缓存、机器学习系统。

收稿日期： 2021-02-01

通信作者： 张凯，wsxiaozhang@sina.com