Capuchin:基于张量的深度学习的GPU内存管理

宣鹏, 宣华世 *, 胡林代 *.
金海, 马伟良, 熊倩, 杨帆, 钱学海‡
华中科技大学计算机科学与技术学院大数据技术与系统国家工程研究中心、服务计算技术与系统实验室
⋆微软亚洲研究院, ‡南加州大学

摘要

近年来，深度学习在各个领域获得了前所未有的成功，成功的关键在于更大更深的*深度神经网络*（DNNs）取得了非常高的精度。另一方面，由于GPU全局内存是一种稀缺资源，大型模型在训练过程中对内存的要求也带来了巨大的挑战。这一局限性限制了DNN架构的探索灵活性。

在本文中，我们提出了*Capuchin*，一个基于张量的GPU内存管理模块，通过张量驱逐/重取和重新计算来减少内存占用。*Capuchin*的主要特点是，它根据运行时跟踪的动态张量访问模式做出内存管理决策。这种设计的动机是观察到在训练迭代期间对张量的访问模式是有规律的。基于识别出的模式，我们可以利用总的内存优化空间，并对*何时和如何*执行内存优化技术进行细粒度和灵活的控制。

我们在一个广泛使用的深度学习框架Tensorfow中部署了*Capuchin*，并表明*Capuchin*可以在6个最先进的DNN中比原来的Tensorfow减少高达85%的内存占用。特别是在NLP任务BERT中，*Capuchin*的最大批处理量超过了Tensorfow和梯度检查点的处理量。
7倍和2.1倍。我们还表明，*Capuchin的*性能优于vDNN和梯度检查点，最高可达286%。
和55%，在同样的内存超额认购下。

* 通讯作者。

ASPLOS '20，2020年3月16-20日，瑞士洛桑。
ACM ISBN 978-1-4503-7102-5/20/03.. . $15.00
https://doi.org/10.1145/3373376.3378505

CCS概念 - **软件及其工程** → **内存管理**； - **计算机系统组织** → *单指令，多数据。*

关键词深度学习训练；GPU内存管理；张量访问

ACM参考格式。
彭宣，石宣华，戴虎林，金海，马伟良，熊倩，杨帆，钱学海。2020.*Capuchin*:基于张量的深度学习的GPU内存管理。在*第二十五届国际编程语言和操作系统架构支持会议（ASPLOS '20）论文集，2020年3月16-20日，瑞士洛桑。*https://doi.org/10.1145/3373376.3378505

1 简介

深度学习应用是计算和内存密集型的。*深度神经网络*（DNN）如今已经非常流行，因为几种新兴的异构硬件提供了巨大的计算能力，如TPU[17]、ASIC和GPU--目前最普遍的训练选项。越来越深、越来越广的DNN的训练对板载的GPU内存构成了巨大的挑战。例如，最新的BERT[9]有768个隐藏层，在训练中消耗73GB的内存（批量大小为64）。然而，高带宽的GPU板载内存是一种稀缺资源：最新的最强大的NVIDIA GPU V100的板载内存最大只能达到32GB，而商业云中的主流GPU类型，例如P100，只有16GB的板载内存。这种限制限制了探索更先进的DNN架构的灵活性。

先前的工作[13, 24]表明，DNN训练中的主要内存足迹是由于中间层输出，即特征图。这些特征图是在前向传播中产生的，并在后向传播中再次使用。因此，主要的深度学习框架，如Tensorfow[2]、MXNet[4]和Pytorch[23]通常在GPU内存中维护这些特征图，直到它们在后向传播计算中不再需要。然而，在前向传播和后向传播中，对同一特征图的两次访问之间通常存在很大的差距，这就造成了存储中间结果的高内存消耗。

为了应对这一挑战，有两种主要技术可以减少内存占用：交换和重新计算。这两种方法的设计原理都是在前向传播中释放特征图的内存，在后向传播中重新生成中间特征图。它们的不同之处在于重新生成的方式。具体来说，*交换*利用CPU的DRAM作为一个更大的外部存储器，在GPU和CPU之间异步地来回复制数据；而*重新计算则*通过重放前向计算来获得所需的中间特征图。这两种方法都不会影响训练精度。另一种方法是在训练过程中使用低精度计算以节省内存消耗。然而，要分析低精度计算对最终训练精度的影响并不容易。本文不考虑这个选项。

先前的工作[5, 24, 31]根据计算图和不同层的特征进行逐层的GPU内存管理。然而，神经网络的一个层是由许多低级数学函数组成的高级计算抽象，即使是一个有几十层的神经网络也可能在其相关的计算图中包含成千上万的节点。例如，ResNet-50中有3000多个节点，BERT中有7000个节点，当它们被转换为Tensorfow的内部计算图时。因此，这种粗粒度的内存管理限制了优化空间。此外，他们通过对计算图的*静态*分析选择交换和重新计算。例如，基于卷积层计算成本高、耗时长的假设，vDNN[24]选择了卷积层的输入作为交换目标，希望增加交换与卷积层计算的重叠。SuperNeurons[31]也避免了卷积层的重新计算以减少开销。

静态分析导致了三个问题：（1）硬件和输入大小的多样性使得预测计算时间变得很困难，即使是同一类型的层也有很大的不同。因此，根据层的类型静态地确定内存优化目标将限制内存优化的潜力。(2) 基于粗粒度的"定性"信息的决定不能量化特定内存操作的开销，因此很难对内存优化候选者进行优先排序或在交换和重新计算之间做出选择。(3) DNNs正在持续快速发展，例如，从*卷积神经网络*（CNN）和*循环神经网络*（RNN）到变形器和*图形神经网络*（GNN），甚至还有用户定义的操作。对于新类型的DNN，先验知识是不可用的。此外，基于计算图的内存管理对于执行前没有计算图的深度学习框架来说效果并不好，例如Pytorch[23]。和Tensorfow的eager模式[3]。因此，这种方法并不通用，不能应用于所有的框架。

在本文中，我们提出了*Capuchin*，一个基于张量的GPU内存管理模块，用于深度学习框架，通过张量驱逐/重取和重新计算减少内存占用。*Capuchin*的主要特点是，它根据运行时跟踪的张量访问模式做出内存管理决策，每个张量都有一个独特的ID。这种方法在张量的粒度上实现了内存管理，允许通过特定的张量访问模式来触发具体的决定。*Capuchin*不仅做出了张量驱逐的决定，而且还确定了预取和重新计算的时间。该模块对用户来说是跨平台的，其实现只需要底层深度学习框架的一些支持。从本质上讲，*Capuchin*通过收集运行时张量访问的反馈信息来迭代修正内存管理策略。*Capuchin*的设计是基于两个关键的观察。首先，所有的深度学习框架都是基于数据流执行模型的，其中的处理程序是基于*张量操作的*。与传统程序的内存访问的重用和定位分析类似，深度学习训练中的张量访问也表现为数据重用和某些访问模式。因此，我们认为，动态跟踪细粒度的张量访问是一种基本的和通用的技术，可以实现有效的内存管理优化。本文证明了这一基本思想可以在主要的计算机系统之上有效地实现。深度学习框架。

其次，深度学习应用的特性确保了我们方法的有效性。训练过程是由*数百万个*具有明确边界的*迭代*组成的，张量访问在不同的迭代中具有*规律性和重复的*访问模式。这意味着分析时间和张量访问模式可以很容易地揭示出具有具体指导意义的内存优化机会，例如，是否以及何时选择驱逐/重新获取或重新计算。此外，这样的内存管理策略可以及时应用于下一次迭代，并从运行时反馈中反复修改。据我们所知，*Capuchin*是第一个可以应用于命令式编程环境的计算图无关的内存优化模块。

我们在广泛使用的深度学习框架Tensorfow的基础上实现了*Capuchin*。我们在P100 GPU上评估了*Capuchin*，结果显示。(1) 与Tensorfow和OpenAI的梯度检查点[1]相比，*Capuchin*可以适应更大的批处理规模，分别为9.27倍和2.14倍；(2) *Capuchin*比vDNN实现了3.86倍的速度提升。我们还表明，在Tensorfow的急切模式下，*Capuchin*在在这种模式下，批处理规模增加了2.71倍，而其他作品都无法优化内存。

2 背景介绍

2.1 深度学习模型训练

DNN是使深度学习成功的关键因素，通常由一个输入和输出层以及中间的多个隐藏层组成。各种类型的层被用于不同的目的和应用，如卷积层、池化层、图像分类中的激活层、*自然语言处理*（NLP）中的注意层。随着深度学习的应用越来越多，DNN模型也在快速、持续地发展。即使考虑到现有的层，这些层之间的连接也可以任意探索，以寻求适合各种问题的神经网络架构，例如ResNet[11]。

DNN训练的目标是找到一组适当的模型参数，以满足一个*目标函数*，它表达了模型的理想属性，如在图像年龄分类中识别物体类别时产生的误差。训练过程通常包括数以百万计的迭代，每个迭代由两个计算阶段组成，即前向传播和后向传播。在前向传播中，输入特征图、当前层的权重和偏置产生输出特征图，成为下一层的输入数据。前向传播的结论是通过比较输出层的输出和地面真实标签来计算损失。后向传播从输出层开始，反向遍历各层以优化权重和偏置。有许多用于后向传播算法的操作计时器[20]，包括最常用的*随机梯度白化*（SGD）、*动量*[36]和*亚当*[18]等。

为了实现高吞吐量，训练样本被*分批*送入计算阶段。事实也证明，随着批次的增加，训练往往能达到更高的精度，最近的工作[10]显示，使用的批次高达8K。

内存的使用。 在训练过程中，GPU内存主要被三部分消耗：特征图，即前向传播的输出；梯度图，即后向传播的输出；以及卷积工作区，即卷积算法所需的额外内存空间。与它们相比，模型权重消耗的内存非常小，而且通常在GPU内存中持久存在，可以持续更新。在这三部分中，后两部分是临时性的内存使用，可以在当前计算完成后立即重新租赁。然而，最近的一些工作[24, 31]试图通过动态选择卷积工作区来提高性能。类似的启发式方法也在TensorFlow和cuDNN[6]中实现，并在我们的评估中（第6节）考虑。在前向传播和后向传播中都需要特征图。然而，在前向和后向阶段的计算中，两个使用点之间存在着很大的时间差。由于这个原因，内存特征图的消耗成为先前工作中的主要优化目标。

2.2 深度学习框架的执行模式

目前流行的深度学习框架通常基于命令式或声明式编程。这两种模式
命令式程序类似于Python和C++程序。
克，在执行过程中进行计算。在Tensorflow中，它被称为*Eager Execution*。PyTorch将其作为默认和唯一的执行模式。
相比之下，声明式程序是基于*图形执行（Graph Execution）*的，在这种模式下，当函数被定义时，实际的计算并没有发生。它被许多框架所支持，包括Tensorflow、CNTK[26]和Cafe[14]。在下文中，我们将更详细地比较这两种方法。

急切模式。 命令式编程环境立即评估操作，而不需要构建图。在这种模式下，操作立即返回具体的值，而不是构建一个计算图，以便以后执行。显然，这种方法使部署和调试一个新模型变得很方便。特别是，急切模式在学术界很受欢迎，因为它可以快速建立一个新模型的原型。顺应这一趋势，TensorFlow宣布从2.0版本开始，eager模式将成为其默认的执行模式。另一方面，由于解释Python代码的开销和缺乏图的优化，这种模式可能比执行等价图要慢。

图模式。 在这种模式下，一个模型的计算被抽象为*计算图*，指定计算节点之间的数据流，由表示张量流的边连接。计算图是在执行开始前建立的，而实际的计算只在必要时才被安排，因此它也被称为懒人编程。当程序被转换为内部计算图时，某些优化（如剪枝和常数折叠）可以应用于程序。因此，内存消耗和训练性能可能比急切模式更好。

基于以上的讨论，我们认为，由于命令式编程环境的流行和便利性，高效的内存管理是非常重要的。

3 观察和激励

3.1 静态分析的局限性

交换中的同步化开销。 之前的工作中，交换的性能取决于交换时间和试图重叠交换的层的执行时间。交换时间由内存大小和CPU与GPU之间的通道带宽决定，而层的执行时间则取决于GPU模型和它的输入

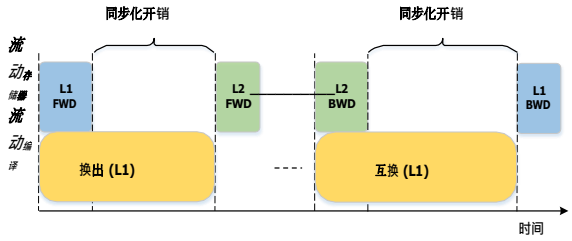


图1. Vgg16上VDN的同步化开销

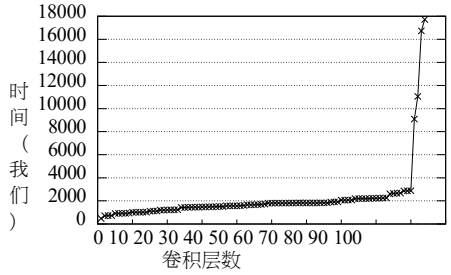


图2. InceptionV3卷积层的执行时间

规模。我们使用配备PCIe 3.0 ×16的P100 GPU对VDN在Vgg16上的执行情况进行了分析（批量大小为230），并将最大尺寸内存的交换时间线显示为图1。每个操作的执行时间与每个块的长度成正比。我们可以看到，换出/换入的时间是重叠层执行时间的3倍多，因此引入了明显的同步开销。在每次换出/换入时普遍存在的同步开销，总的性能损失达到了41.3%.

这种性能损失是由于对静态分析的两个方面的fxed控制：（i）换出后何时释放内存；（ii）何时预取。当当前层的执行时间不足以重叠数据传输时，同步开销是巨大的，而且这种开销在后续迭代中持续存在。

同一类型的层的执行时间变化。

先前的工作根据鉴于卷积的高计算成本做出内存管理决层类型。策。vDNN[24]试图将数据传输与卷积层的计算重叠，而Chen等人[5]则避免了重新计算卷积层。这是基于经验的假设，即卷积层的计算很耗时。我们观察到在一个神经网络中不同的卷积层的执行时间有很大的不同，如图2所示，在P100 GPU上运行InceptionV3。最小和最大的执行时间是474us和17727us。分别大到37倍。在这94个卷积层中，95.7%的执行时间都小于3ms以上，而左边的四个则超过8ms。

这带来了两个问题：(i)在强大的GPU上将数据传输与对话层的计算完美地重叠起来仍然很困难--在PCIe 3.0 ×16下，3ms只能将数据传输重叠在40MB以内。(ii)鉴于卷积层在CNN中占据了大多数层。完全忽略卷积层作为一个重新计算的目标将极大地限制内存优化空间。事实上，重新计算低开销的卷积层是相当可行的，因为其计算时间（小于3ms）只占一个迭代时间（超过1s）的3‰以下。

3.2 机会。常规张量访问模式

深度学习训练包括数百万次的迭代。为了了解张量的访问模式，我们在ResNet-50中选择了三个张量，并在P100 GPU上分析了它们在迭代5、10、15的访问时间戳（与每个迭代开始的时间有关），结果如图3所示。张量的访问明显遵循一个规律，也就是说，在一个迭代中出现的次数和时间戳大部分是固定的。T1出现了四次，其中两次是在前向阶段，另外两次是在后向阶段，而T2和T3出现了六次。同一张量访问在不同迭代中的时间差异小于1ms。

虽然结果只显示了在图形模式下运行的CNN任务的行为，但我们发现其他类型的工作负载，如语音、NLP和使用急切模式，也表现出类似的模式。利用深度学习训练中的常规张量访问方式，*Capuchin*得出了智能内存管理策略。首先，我们可以清楚地识别那些可以用来减少内存占用的张量。其次，张量访问的时间间隔是做出内存管理决策的有效信息，这在第4.3节中讨论。在图3中，很明显，对T1进行交换比对T2和T3进行交换更有利，因为T1的两个连续访问的时间间隔更大。这个决定更有可能减少开销（假设三个张量有相同的大小）。

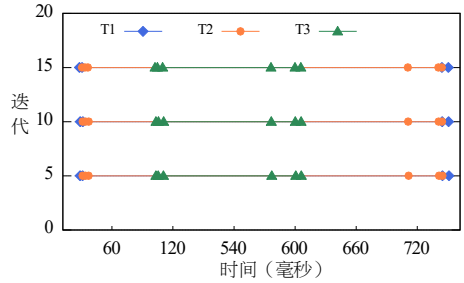


图3. ResNet-50张量访问的时间线

4 *Capuchin*内存管理 机制

4.1 设计目标

我们试图在*Capuchin*的设计中实现两个目标。首先，我们应该尽最大努力使*尽量减少开销。*

众所周知，现代DNN的训练可能需要几天甚至几周的时间，即使是适度的性能开销（如50%）也会因为使用昂贵的GPU资源而大大增加训练的总成本。因此，*Capuchin*的design必须通过内存超额订阅来尽量减少性能开销。第二，该框架应该*一般和引起少量的代码修改*为不同的深度学习框架。它可以通过识别框架之间的一般设计特征来实现。

4.2 设计概述

由于张量访问模式的规律性，我们把的训练分为两个阶段。(1) *衡量的执行*：。执行第一个小批次（迭代），在这里我们观察动态张量访问序列，在此基础上可以做出张量内存管理决策；以及（2）*指导性执行*：在第一个迭代之后，根据使用观察到的张量访问模式生成的内存管理策略进行训练。

*Capuchin*的基本目标是支持当前框架中无法实现的大批量。因此，在整个执行过程中，*Capuchin*应该能够继续执行，即使发生*内存不足（OOM）*和*访问失败*。为了实现这一目标，系统需要在Null指针上拦截张量访问，然后触发张量驱逐。我们把这种*被动模式*称为执行在张量的驱逐中，张量的内存将被*腾需覆盖*到CPU内存中，张量的ID将被记录在框架中，以便以后再次访问时可以从CPU内存中换入。

从本质上讲，被动模式的行为类似于传统操作系统中的虚拟内存，它*以页*的粒度进行换入/换出操作，并使用磁盘作为CPU内存之外的扩展空间，以提供大的虚拟内存空间的错觉。相比之下，我们的系统以张量粒度运行，将CPU内存作为有限的GPU内存的扩展缓冲区。在被动模式下，即使所需的内存大于可用的GPU内存，*Capuchin*也能继续执行，只有一个例外。当一个计算函数将张量A作为输入并产生张量B作为输出时，如果A和B的大小加起来已经大于整个GPU内存，我们的被动模式无法处理这种情况，因为没有机会进行张量交换。然而，我们相信这种情况是极其罕见的。

在测量的执行过程中，*Capuchin*保留了张量的访问序列和每个张量的附加信息，包括*访问计数*、*时间戳*、*产生张量的操作和它的输入张量*，以便重新计算决定。在被动模式下，张量A的驱逐总是由另一个张量B的访问触发的，此时内存不足，无法为B分配空间。

类似于缓存替换。然而，驱逐（或换出）需要时间，而且是在执行的关键路径上。为了提高性能，*Capuchin*利用了缓存一致性协议中*"自我验证"*的类似想法。具体来说，根据观察到的张量访问模式，该框架可以决定在一次访问之后*主动地*驱逐张量A。

到自己身上。与按需互换相比，主动的eviction可以为将来的其他张量*提前*释放内存，并隐藏掉换出的开销。

当一个被驱逐的张量再次被访问时，它需要被*重新生成*。如前所述，*Capuchin*考虑了两种方法--交换和重新计算。与主动驱逐类似，以尽量减少张量重新生成的开销。我们也可以*在*使用*前*启动张量的生成。在关键问题是要选择一个合适的时间点，既不能太早，否则会不必要地增加内存压力；也不能太晚，否则会在执行关键路径中暴露出重新生成的问题。为了决定正确的时间点，我们需要确定交换和重新计算的成本。

在讨论详细的机制之前，我们首先要确定几个术语。*evicted-access*是指在计算中使用张量访问后触发的自我否定。*后置访问*是指张量从GPU内存中被驱逐后的第一个张量访问，即张量被驱逐后的访问。直观地讲，在驱逐和回访之间的较长间隔提供了更好的机会来隐藏张量的重新生成开销。值得注意的是，当执行回访时，张量可能在GPU内存中，也可能不在。如果没有主动的张量重新生成，张量肯定不在GPU内存中--在关键路径中增加交换的开销。为了减少这种开销，张量A的重新生成可以由另一个称为*in-trigger的*张量操作提前触发。它可以是张量的驱逐访问和返回访问之间的一个任意的访问。当它没有被指定时，我们可以认为在访问失败后，back-access被作为触发器来带入张量。

在下文中，我们将根据*Capuchin*的一般方法来回答这些自然问题。

- 如何估计互换和重新计算的成本？
- 要驱逐哪些张量？
- 何时驱逐并重新生成？
- 如何重新生成（互换与重新计算）？

4.3 估算互换和重新计算的收益

对于这两种张量重生成方法，即交换和重新计算，目标是尽可能地隐藏开销，以尽量减少回访的等待时间。一般来说，对于交换，我们应该增加交换和正常计算之间的重叠；对于重新计算，我们应该选择便宜的操作来重新计算。实现这些目标的关键是*对计算时间和交换时间的准确估计*。如图4所示，*Capuchin*根据测量执行中的访问时间剖析来估计开销。

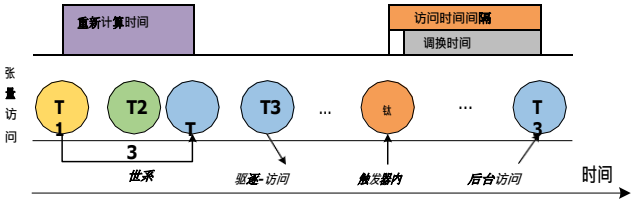


图 4.一个评估张量的交换和重组的例子。T_3在其第二次访问中被驱逐。T_1是它的一个输入。

*SwapTime*可以通过将张量的内存大小除以PCIe带宽来计算。PCIe带宽相当稳定，在具有16个通道的PCIe 3.0下可以达到12GB/s。为了量化交换张量的潜力，我们引入了*自由时间（FT）*的指标。

$$FT = SwapInStartTime - SwapOutEndTime \quad (1)$$

*SwapOutEndTime*等于被驱逐的访问时间加上*SwapTime*，*SwapInStartTime*表示预取此张量的时间。理想的*SwapInStartTime*可以通过从回访时间中减去*SwapTime*而计算出来。

对于重新计算，挑战在于如何估计重新计算一个操作的成本，这与操作算法、设备计算能力和输入大小密切相关。由于并行内核计算之间的干扰，Hyper-Q技术使问题更加复杂，几乎不可能通过静态分析来估计。因此，*Capuchin*通过记录张量的线程和运行时访问时间来*测量*测量执行期间的重新计算时间。具体来说，一个操作的再计算时间可以通过组合输出和输入张量的访问时间来获得。通过这种方式，评估从任意重新计算源开始重新计算张量的开销是可行的。

与交换类似，我们定义*每秒节省内存（MSPS）*来表示重新计算张量的有利程度，也就是说，更高的内存减少和更少的重新计算时间将导致更高的*MSPS*。

$$MSPS = \frac{记忆的保存}{重新计算时间} \quad (2)$$

给定一个张量，节省的内存是一个常数，但重新计算的时间却随着我们开始计算这个张量的地方而变化。接下来，我们讨论如何计算交换和重新计算的时间。

4.4 确定张量再生成的成本

本节讨论了如何估计张量重新生成的成本。目标是选择适当的时间来启动操作，即交换或重新计算，以便不增加内存压力（太早）或增加执行中的等待时间（太晚）。我们考虑交换和分别进行重新计算，并将在第4.5节讨论两者的选择。

对于交换，我们可以通过计算*SwapInStartTime*（后端访问时间减去*SwapTime*）来推断出预取一个张量的最新时间，然后我们可以从张量访问列表中的后端访问反向追踪，寻找第一个张量访问。其时间早于*SwapInStartTime*。这个计算是基于理想的情况，没有考虑到所有的执行环境和其他张量。在现实中，这个时间可能并不准确，或者需要稍作调整。首先，in-trigger不应该设置在峰值内存的时间范围内的某一点。这样做很可能会导致其他张量的驱逐，类似于被动模式下的按需交换。其次，由于钉住的内存传输完全占据了单向的PCIe通道，在其前面的交换完成之前，一个交换不能开始。因此，实际的交换可能比in-trigger晚发生。

为了减少动态调度和多次交换之间的PCIe干扰造成的不确定性，我们在*Capuchin*中引入了反馈驱动的调整，在运行时动态调整张量的触发时间。关键的见解是在张量的回访中得到运行时的反馈，如果张量仍在被换入，就意味着应该提前调整in-trigger时间。该机制可以通过在张量数据结构中保留交换状态而轻松实现。详细的数据结构设计将在第5.2节讨论。在我们的实验中，根据反馈，在下一次迭代中，in-trigger被提前调整为其交换时间的5%。

重新计算与交换不同，交换只需要PCIe资源，可以与当前计算重叠。相反，重新计算也需要GPU计算资源，它是否能与当前的计算重叠，取决于当时的空闲GPU资源。根据我们的观察，当GPU内存不足时，GPU的计算资源通常也会耗尽。出于这个原因，我们没有设置重新计算的触发器，而是以按需的方式进行。

我们仍然需要从测量的执行中估计重新计算的时间，以获得张量选择的MSPS用于驱逐。它比交换时间更复杂，因为张量世系中的依赖性。取T的线型$_1 \rightarrow$以$T_2 \rightarrow T_3 \rightarrow T_4$为例，假设$T_1$，$T_2$，和$T_4$是重新计算的*候选人*。重新计算$T_4$的成本取决于重新计算的来源。如果对$T_3$的最后一次访问是在前向传播中对$T_4$的计算，那么在重新计算$T_4$时，$T_3$一定是不可用的，因为它在最后一次访问后将被驱逐出GPU内存。在这种情况下，T_4的重新计算将从T_2开始。即使T_2也是一个候选人，在计算中我们认为T_2在GPU内存中，这意味着我们不需要从T_1开始重新计算。或者，当T_3也在后向传播中被访问时，我们比较T_3的最后访问时间和T_4的后访问时间。如果前者较大，我们就认为T_3是可用的。

在T_4 '的重新计算，而T_4 可以从T_3 重新计算；否则，T_3 本身也需要重新计算。基于这种方法，我们可以通过两个可用的信息来生成每个候选张量的重新计算成本：（1）世系中张量的寿命，以确定一个张量（不是重新计算候选张量之一）是否可以作为源；以及（2）世系中的张量是否是重新计算候选张量--
它们被假定在GPU内存中。上述计算的关键观察点是，一旦一个张量被确认为重新计算，它将影响其他候选张量的MSPS。测量的执行可以得到（1），接下来我们将在第4.5节讨论候选人的生成和迭代MSPS更新算法。

4.5 选择内存优化

在本节中，我们考虑如何在选定的一组张量的交换和重新计算之间做出选择。基于前面的讨论，很明显，交换可以在很大程度上与计算重叠，引入少量或没有开销；而重新计算肯定会产生性能损失，因为它需要计算资源。因此，我们选择交换作为第一选择，直到我们无法选择一个内触发器来完美隐藏预取开销。在这一点上，我们将考虑这两个选项，分别比较交换或重新计算所有考虑中的张量的最小开销。将选择开销较少的张量。该程序显示在算法1中。接下来我们详细解释每一个步骤。

算法1：混合政策

输入：张量访问序列（tas），mem_saving

```
候选人← IdentifyAndSortCandidates(tas)。
为候选人中的t做
    ChooseSwapIntrıɔɔer (t );
    s_overhead ← SwapOverhead(t )。
    r _overhead ← RecomputeOverhead(t )。
    如果s_overhead < r _overhead 那么
        交换(t )。
    结束
    否则
        重新计算(t )。
    结束
    mem_savinɔ ← mem_savinɔ − t .mem;
    如果mem_savinɔ≤0，那么
        休息。
    结束
    UpdateCandidates(candates)。
结束
```

根据在被动模式下的测量执行中获得的张量访问序列，我们将一个张量插入到*evic-tion候选集*，如果。(i)张量的访问次数超过1；(ii)张量的访问发生在张量生活的内存使用峰值期。在分析过程中，我们可以跟踪张量的分配和删除时间，以推断出内存的使用情况。

接下来，我们确定张量*驱逐集*，其中包含使用交换的十个张量，这些张量被认为要被驱逐。基于驱逐候选集中的张量访问列表，我们根据两个连续的张量访问之间的FT长度生成一个排名列表，假设张量在两次访问之间被交换掉。由于较长的FT提供了更多的机会来隐藏张量交换的开销，我们从列表的顶部选择10个访问对进行驱逐，没有开销。被选中的张量被从驱逐候选集中删除。使用这种方法，驱逐集包含所选张量的*{被驱逐的访问，后方访问}*对的列表。在引导执行过程中，*Capuchin*将在evicted-access处触发对特定张量的驱逐，并通过第4.4节中讨论的使用交换开销确定的in-trigger重新生成用于back-access的张量。请注意，在GPU内存中进行整个训练所需的内存减少可以从测量的执行中得到
-
在被动模式下，这个数量是被驱逐的张量的总内存大小。如果使用交换所减少的内存大于这个数量，我们就在evic-tion集合中选择了足够的张量，这个过程就可以终止了。否则，我们将进一步考虑交换和重新计算留在候选集中的张量。

为了用重新计算来选择被驱逐的张量，我们需要用第4.4节中描述的方法来计算当前候选集中所有张量的MSPS。重新计算的主要问题是，一旦某个张量被选入驱逐集，候选集中其他张量的重新计算源就会受到影响，最终导致MSPS的改变。我们通过以下方式将张量迭代插入到驱逐集。(1)
更新：根据当前的候选集和驱逐集计算当前候选集中张量的MSPS；(2)
选择：将具有最大MSPS的张量插入驱逐集，并将其从候选集中删除。为了说明选择张量对MSPS的影响，让我们考虑一个例子。在$T_1 \rightarrow T_2 \rightarrow T_3 \rightarrow T_4$ 中，假设候选集是{T_1 ，T_2 ，T_4 }，我们选择T_2 首先插入到驱逐集。我们需要更新MSPS的
其他张量进行下一次选择。对于T_4 ，重新计算的来源将不是T_2 ，而是T_2 的来源T_1 ，这将影响T_4 的重新计算时间。此外，所有以T_4 为源的重新计算目标也应该从T_1 开始。这使得从T_1 到T_4 的计算重复多次。对于T_1 ，T_2 和所有来源包括T_2 的再计算目标将多次重复T_1
的再计算。我们通过在*MSPS*计算中多次加入重复的重新计算的时间来揭示这种额外的重新计算开销

的T_4和T_1，它们分别对应于
算法2的第23-28和30-32行。

在选择一个张量进行重新计算后，将其开销与所选张量的开销进行比较，最终选择开销较小的张量。另一个张量则被插回候选集，供进一步考虑。

算法2：重新计算策略 **输入：**
候选者，mem_saving **输出：**
重新计算目标

```
1 InitMSPS（候选人）。
2 recomps ← {}。
3 当mem_saving>0时，做
4     t ← MaxMSPS(候选者)。
5     ext _ct ← 1。
6     foreach rp in recomps do
7         如果t∈rp.srcs，那么
8             rp.srcs .删除(t )。
9             rp.srcs .Add(t . srcs)。
10            ext _ct += 1;
11        结束
12    结束
13    recomps .加(t )。
14    候选人.删除(t)。
15    mem_saving −= t .mem;
16    /* 更新候选人的MSPS */
17    foreach cand in candidates do
18        如果t∈cand.srcs，那么
19            cand . srcs .删除(t )。
20            cand . srcs .Add(t . srcs)。
21            cand . rp_time += t . rp_time;
22            蜡烛 ..... ext _time ← 0。
23            foreach rp in recomps do
24                如果cand∈rp.srcs，那么
25                    蜡烛。Ext _time += 蜡烛。
Rp_time。
26                结束
27            结束
28            蜡烛.UpdateMSPS()。
29        结束
30        如果cand∈t.srcs，那么
31            蜡烛...ext _time = ext _ct ∗ 蜡烛...rp_time。
32            蜡烛.UpdateMSPS()。
33        结束
34    结束
35结束
```

5 卡普金的实施

本节解释了*Capuchin*的实现，我们首先讨论了对底层深度学习框架的要求，并展示了系统的架构，其中有两个例子

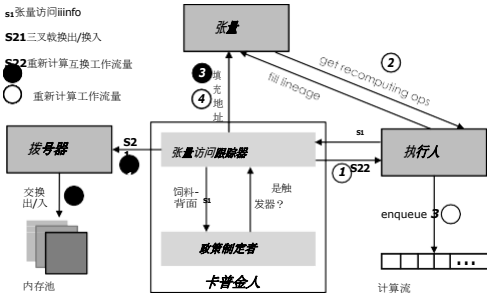


图5.*Capuchin*系统结构

关键模块。然后我们提出两个性能优化方案。最后我们讨论了与GPU架构有关的具体问题。

5.1 框架要求

*Capuchin*假设底层框架有这两个模块。

执行器。它指的是基本计算操作的处理单元。通常情况下，输出可以由一个张量矢量和一个计算内核作为输入产生。为了获得张量的访问顺序，我们可以在张量的编译前后使用RecordTensorAccess函数。为了支持*Capuchin*的重新计算服务，我们采用了一种基于线上的重新计算，类似于Spark的RDD线上。*Capuchin*的世系不是在执行前为了容错而建立的，而是在运行时生成的。给定一个张量的唯一ID作为参数，我们可以搜索到最接近的输入张量进行重新计算。

Allocator。它是用于动态GPU分配的原始cudaMalloc和cudaFree的包装器，它通常为高层模块提供两个方法，即Allocate和Deallocate。一般来说，当一个张量的引用计数达到零时，Deallocate将被自动调用。
为了支持*Capuchin*的交换服务，分配器需要支持两个额外的方法，即SwapOut和SwapIn。这两个方法接受一个地址作为参数，然后根据目标设备在GPU或CPU上分配一个同等大小的内存。最后，一个特定设备的复制操作将被调用，例如，在NVIDIA GPU中，运行时API cudaMemcpyAsync负责这一操作。

5.2 系统结构

图5显示了*Capuchin*的系统结构。为了监控张量的访问，*Capuchin*在*张量*结构中增加了额外的栅栏，如清单1所示。tensor_id是张量的唯一ID，用于在多个迭代中定位同一张量，因为张量的底层内存地址在多个迭代中可能是不同的。因此，tensor_id可以确保某些内存优化被应用到

在接下来的迭代中正确的张量。access_count和timestamp表示张量被访问的次数和最近一次张量访问的timestamp。这对{tensor_id, access_count}可以指定一个特定的张量访问，可能会触发一个内存优化策略。Tensor的状态栏里有五个状态：IN、SWAPPING_OUT、OUT、SWAPPING_IN和RECOMPUTE。对于
如果张量被驱逐以备将来重新计算，张量的内存将被简单地释放，所以只需要三种状态，即IN、OUT和RECOMPUTE。根据这个结构，*Capuchin*可以在运行时检查张量的状态，以调整内存管理策略，这在第4节中已经描述过。**输入**和**操作_名称**共同构建了张量的线程，用于重新计算。

*Capuchin*由两个模块组成。*张量访问跟踪器*（TAT）和*策略制定者*（PM）。TAT与深度学习框架中的*Executor*、*Tensor*和*Allocator*交互，以跟踪张量访问并执行与特定张量访问相关的内存管理操作。PM负责根据TAT提供的张量访问信息做出内存管理策略决定。

```
class Tensor {
  string tensor_id;
  int access_count;
  int timestamp; int
  status ;
  // 用于重新计算
  向量 < 张量 * > inputs ; // 输入张量 string
  operation_name ; // 产生的操作。
   这个张量
  ...
}
```

清单1.张量结构

张量访问跟踪器（TAT） 维护一个10个访问列表，其中每个元素包括{张量_id, access_count, timestamp}。当一个张量被生成时（它的第一次访问），访问计数被初始化为1，当前时间的时间戳被存储在时间戳中。每次张量被访问时，这三个值将被记录并插入张量访问列表中。

TAT的设计有两个目的。首先，它支持按需的内存交换，以克服测量执行中的*内存不足（OOM）*和*访问失败*。当它们发生时，系统将触发张量驱逐，并在Null指针上拦截张量访问。其次，它跟踪张量访问模式，以便PM可以根据一个迭代中完整的张量访问序列做出并动态地调整内存优化决策。

为了支持*按需交换*，TAT工作在图6所示的*被动模式*。当检测到OOM时，TAT将在张量访问列表中寻找一个或多个有近似大小的张量，从一开始就被驱逐，直到当前分配成功。然后，被驱逐的张量的内存将被同步复制到CPU内存中，{tensor_id, cpu_addr}一对将被保存在TAT中。之后

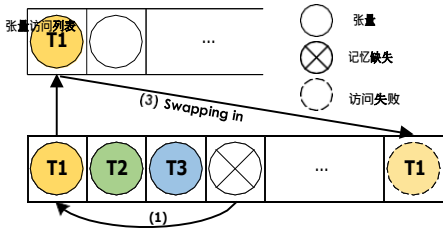


图6.张量访问跟踪器的被动模式

驱逐一个张量，当这个张量被再次访问时，访问失败将发生。这时，TAT将提供这个张量对应的CPU地址，并将数据复制到GPU。

在测量执行过程中，由于张量访问信息处于被动状态，主要问题是识别完整迭代之间的张量访问边界。在一些框架中，如Tensorfow，这个信息是直接知道的，因为迭代号被表示为步骤ID。对于其他框架，如PyTorch，每个迭代没有明确的ID，但当同一张量生成两次时，边界仍然可以被识别。由于被动模式下的张量访问时间包括按需交换时间，我们需要从张量访问时间中减去这个时间，以获得张量访问发生的实际时间点，假设GPU内存为无穷大。虽然需要一个锁来保持张量访问序列的完整性，但TAT中的调用都是异步的，不在关键路径中。6.3.2节展示了低开销的情况。PM根据张量访问顺序确定内存优化策略。由TAT跟踪的特定张量访问将触发相应的内存优化操作，这些操作在*Allocator*或*Executor*中执行，取决于交换或重新计算。

5.3 优化

脱钩的计算和互换。在之前的工作中[22, 24, 31]，每次张量被换出后，只有当使用该张量的当前计算和该张量的换出都结束后，才能开始下一次计算。这是因为计算和数据传输是耦合的，也就是说，计算需要在张量的换出时同步数据传输。这如图7的左边所示。当当前的计算不足以重叠换出时，这样的同步化就会引入非显著的过度。然而，所有接下来的计算，包括使用该张量的当前计算，都可以与张量的换出同时进行。在图7中，这意味着前向计算*FWD2*和*FWD3*可以与*SwapOut1*重合。为了利用这个机会，我们在张量的换出时将计算和数据传输解耦，只在OOM发生时同步最早的未完成的换出。这样一来，张量换出的开销就会大大减少。

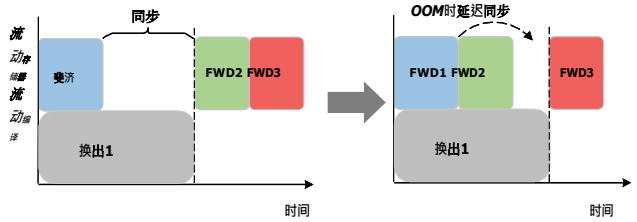


图7.解耦计算和交换

集体再计算。它是一种用一个重新计算来存储多个重新计算目标张量的机制。以$T_1 \rightarrow T_2 \rightarrow T_3 \rightarrow T_4$为例（$T_2$至$T_4$
为再计算目标）。在前向传播中释放了这三个张量的内存后，T_4 将首先在后向传播中被重新计算。此时，T_2和T_3 还没有被触发开始重新计算--当开始T_4的重新计算时，它们不在GPU内存中。
因此，T_4 的重新计算将从T_1 开始（假设T_1现在在GPU内存中），并在这个过程中产生T_2 ，T_3。考虑到接下来需要T_2 和T_3，我们可以考虑在重新计算T_4
的过程中存储它们。这实质上是在节省内存和重新计算的效率之间做一个权衡。我们保留的张量越多，内存压力就越大，但在最好的情况下，重新计算的复杂性将从$O(n^2)$减少到$O(n)$。

然而，要在执行之前做出决定是很困难的。
因为我们不知道有多少可用的内存来保留多个再计算目标。因此，我们在运行时动态地应用集体再计算优化。在前面的例子中，在重新计算T_4
的过程中，我们将保留T_2
，一旦它被计算出来。当继续计算T_4
时，如果内存足够，T_2
仍将被保留；否则，其内存将被释放。通过这种方式，我们可以在一个重新计算过程中尽可能多地保留最新的重新计算张量。

5.4 GPU特有的问题

访问时间的计算。
我们需要张量的访问时间来评估交换和重新计算的有效性。在Tensorfow中，CPU的处理与GPU的运算是并行的，以便尽可能地使GPU饱和。这意味着，计算功能在将内核排入GPU流后立即返回。如果我们记录这个计算前后的时间，我们只能得到CPU进程的时间，而不是真正的GPU处理时间。为了解决这个问题，我们通过*CUDA Profiling工具接口（CUPTI）来*测量，这也是在Tensorfow中实现的（在 会 话
中启用`run_metadata`）。由于只做一次，所以产生的少量开销是可以接受的。

*Capuchin*的函数也不应该立即执行，而是应该延迟到流中的适当位置执行。例如，当一个换出操作被一个特定的张量访问触发时，它不应该在那个时候执行换出操作，而是要等到GPU到达相应的点。此外，内存优化操作应该异步执行，以避免关键路径上的块执行。请注意，重新计算

异步和延迟的操作。
对于在GPU上运行的内核，CPU只是将其排入GPU流。相应地。

这就像正常的计算一样，所以它自然是一个异步的和延迟的操作。因此，我们只需要关心Tensorfow中的交换问题。*Capuchin*通过利用*CUDA流*和*CUDA事件*来实现这一目标。通过在必要时将CUDA事件插入到CUDA流中并监控CUDA事件的状态，*Capuchin*只在相应的CUDA事件完成后执行函数。我们使用异步内存拷贝API（即`cudaMemcpyAsync()`）和两个独立的CUDA流来实现换出/换入。

6 评价

6.1 方法论

实验设置。我们的实验是在一台配备了NVIDIA Tesla P100 GPU，双2.60GHz英特尔至强CPU E5-2680 v4处理器，28个逻辑核心，256GB内存，PCIe 3.0 ×16，运行Ubuntu 16.04的服务器上进行的。CUDA工具包的版本是9.0，cuDNN是7.3.1。Tensorfow的ver-1.11版本，其中*Capuchin*也是基于该版本进行修改的。

工作负载。我们在7个最先进的深度学习工作负载上评估了*Capuchin*，如表1所示。我们对这些CNN使用合成数据，而不是ImageNet[8]，以便更好地揭示内存优化的效果，因为数据预处理可以帮助训练在CPU中受限[35]--
不能完全显示出我们的方法的好处。BERT[9]是由谷歌人工智能语言提出的，在各种NLP任务中取得了最先进的成果。BERT的基本版本包括768个隐藏层和1.1亿个参数。

表1.工作负载

	建筑学	数据集	运行模式
VGG16 [27]	有线电视新闻网	合成的	图形
ResNet-50 [11]	有线电视新闻网	合成的	图形&渴望
ResNet-152 [11]	有线电视新闻网	合成的	图形
InceptionV3 [30]	有线电视新闻网	合成的	图形
InceptionV4 [30]	有线电视新闻网	合成的	图形
密集网[12]	有线电视新闻网	合成的	渴望
BERT [9]	转换器	古腾堡[19]	图形

基线。第一个基线是*TF-ori*，即Tensorfow的原始版本1。我们还将*Capuchin*与其他先进的框架进行比较，这些框架都是基于计算的。

[1] Tensorfow已经在静态内存分配策略框架中使用了内存共享和就地操作的操作技术，如MxNet，因为动态分配会自动释放内存，而MxNet则会自动释放。

图。没有现有的内存优化可以应用于急切模式，所以我们只将*Capuchin*与Tensorfow的原始急切模式进行比较。

在图模式下，第二条基线是vDNN[24]，它装载了适当的内存并使用静态预取策略来重叠数据传输和计算时间。我们实现了vDNN的加载和预取策略。vDNN在Tensorfow中使用其预先定义的操作符进行交换。

由于vDNN是为CNN设计的，所以它不能在BERT上使用。第三条基线是OpenAI的梯度检查点[1]，它是对Chen等人在Tensorfow上的重新计算策略[5]的重新实现。它包括两种模式，*内存*和*速度*。点，旨在实现$O(sqrt(n))$的内存使用。该系统通过自动识别*衔接点进行*工作在图中，即在移除时将图分成两个不相连的部分的张量，然后对这些张量的适当数量进行检查点。速度模式试图通过对所有通常计算成本较高的操作，即卷积和矩阵乘法的输出进行检查点来最大化训练速度。因此，除了CNN，这种模式对其他神经网络不起作用。

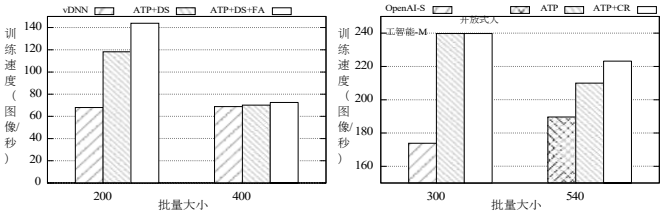
6.2 细分分析

我们分别评估了*Capuchin*在交换和推荐方面的机制。

互换。 我们比较了只在*Capuchin*上启用交换的vDNN的训练速度，并展示了每种机制的性能改进。我们在InceptionV3上进行了实验，结果如图8（a）所示。这里，DS（*延迟同步*）指的是解耦计算和交换优化；ATP（*基于访问时间的程序*）指的是启用测量执行；FA指的是反馈驱动的触发时间调整。请注意，*Capuchin*不知道神经网络的结构，也不知道它的计算图，因此我们不能在换出/换入时进行逐层同步，必须启用延迟同步。

vDNN在InceptionV3上能达到的最大批量是400。在这个批处理规模下，评估显示*Capuchin*只提高了5.5%的训练速度。我们发现，需要撤离的内存总量约为25GB，换出/换入时间分别为1.97s和2.60s（设备到主机的带宽比主机到设备的带宽快一点）。因此，可能与数据传输重叠的计算时间只有大约2.0s。然而，总的数据传输时间是计算时间的两倍多，所以计算时间远远不够与数据传输相重叠。因此，在大批量的情况下，交换带来的改进是非常有限的。随着

不再需要，而且原地操作已经被整合到算法的实现中。



(a) InceptionV3上的互换　　**(b)** 在ResNet-50上进行重新计算

图8.分解分析

在相对较小的200个批次中，细分分析显示ATP+DS提高了73.9%的训练速度，而FA在ATP+DS的基础上进一步提高了21.9%。

重新计算。我们通过在*Capuchin*中只启用再计算来比较OpenAI的两种再计算模式的训练速度，并分别介绍每种机制的性能改进。我们在ResNet-50上进行了实验，结果如图8（b）所示。

OpenAI的重新计算的内存模式被表示为*OpenAI-M*，而速度模式被表示为*OpenAI-S*。x轴表示OpenAI-S和OpenAI-M能达到的最大批量大小，分别为300和540。我们只启用ATP，然后启用*集体再计算*（CR）来评估*Capuchin*在这两种批量大小下的再计算。我们可以看到，OpenAI-S的训练速度比OpenAI-M还要低8.3%，这也说明根据层的类型来选择内存优化目标是不合适的。在批处理量为300时，单独启用ATP和同时启用ATP和CR时，训练速度保持不变。这是因为在所有的重新计算过程中只有一个目标张量，因此集体重新计算并没有提供改进。因此，所有的改进都是由于ATP，与OpenAI-S相比，它的性能提高了37.9%。在批量大小为540的情况下，*Capuchin*比OpenAI-M高出17.8%，其中ATP贡献了10.7%的性能改进，CR贡献了另外7.1%。

6.3 图形模式的评价

6.3.1 减少内存足迹

我们用批量大小来表示内存占用的减少程度。由于OpenAI有两种模式，我们选择较大的一种作为代表。

表2列出了原始的Tensorfow、vDNN、OpenAI和*Capuchin*所能达到的最大批量大小。我们观察到，*Capuchin*总是能够达到最大的批处理量。大小。与原始的Tensorfow相比，*Capuchin*将ResNet-50的最大批处理量提高了9.27倍，而将Tensorfow的最大批处理量提高了1.5倍。平均为5.49倍。特别是对于BERT，*Capuchin*实现了7倍的批量大小改进。与第二好的公司相比(VDNN在Vgg16上表现最好，而OpenAI在Vgg16上获胜。

表2.图表模式下的最大批处理量

模型	栗子	vDNN	兴业银行	*卡普金人*
Vgg16	228	272	260	**350**
共和国网-50	190	520	540	**1014**
ResNet-152	86	330	440	**798**
InceptionV3	160	400	400	**716**
InceptionV4	88	220	220	**468**
BERT	64	-	210	**450**

其他的），*Capuchin*仍然可以实现批量大小的提升，对于BERT来说可以达到2.14倍，平均为1.84倍。这要归功于*Capuchin*的张量式内存管理，它可以提取所有的内存优化机会。在所有工作负载中，对Vgg16的改进是最小的。
因为Vgg16的层数最少，其中有几个层需要巨大的内存来计算，例如，它的第一个ReLU层需要大约6GB。这种僵硬的内存占用要求不能通过交换或重新计算来优化。

6.3.2 业绩

在这一节中，我们评估了*Capuchin*与原始Tensorfow、vDNN和OpenAI的性能。在内存和速度模式之间，我们选择性能更好的OpenAI来表示。

运行时间的开销。首先，我们测量了*Capuchin*由于运行时信息跟踪而产生的运行时间开销。我们运行了原始Tensorfow可以容纳的批量大小，测量了20次迭代的平均训练速度。结果如图9所示。前两个（在Vgg16上是三个）批次大小是原始Tensorfow可以运行的。在Tensorfow的最大批处理规模下，*Capuchin*在所有工作负载中引入的开销都小于1%，平均为0.36%。在相对较小的批次规模下，ResNet-
152的最大开销为1.6%，平均为0.9%。这意味着当GPU计算量大时，*Capuchin*引入的开销可以忽略不计。

性能比较。由于*Capuchin*会在运行时对策略进行修正，我们将忽略这些迭代，只对策略稳定前的性能进行评估（通常在50次迭代内）。至于性能基线，我们使用原始Tensorfow能达到的最大批量下的训练速度。但是Vgg16有一个例外，当批次从208增加到228时，性能下降了25%，如图9（a）所示。这是由于一些卷积层由于内存的限制而退回到一个较慢的卷积算法。因此，我们把批处理量为208时的性能作为Vgg16的基线。

图9总结了*Capuchin*与基线的性能比较。在所有的神经网络中，*Capuchin*一直表现出最好的性能，第二名是"A"。OpenAI是最好的，而VDNN是最差的。vDNN和OpenAI的性能在不同的批处理规模下几乎保持不变，这是因为它们的静态内存优化策略。对于CNN工作负载，vDNN在ResNet网络上表现出极高的性能损失，ResNet-
152的性能损失高达74.4%，ResNet-
50为70.0%。这种极度的性能损失是由于在强大的P100 GPU上，层间的同步开销是巨大的，因为计算时间非常快，不能与数据传输时间重叠。OpenAI的速度模式在批量较小的Inception网络上表现出更好的性能，而内存模式在其他神经网络上总是表现出更好的性能。因此，我们可以清楚地看到，在OpenAI的最大批处理规模下，OpenAI在InceptionV3/4上的性能下降了。与vDNN和OpenAI相比，*Capuchin*
在BERT上可分别实现高达3.86倍和1.55倍的性能提升，其中1.55倍的提升是在BERT上实现的。

总的来说，*Capuchin*可以实现平均每项工作的75%。在第6.3.1节中，负载的最大批处理量的性能损失在26%以内。我们可以清楚地看到，随着批量大小的增加，*Capuchin*的性能也在慢慢恶化。当批量大小只比TF-ori的最大批量大小增加20%时，*Capuchin*的策略只包括交换候选人，在所有的工作负载中提供3%的性能下降。当它进一步增长时，该策略由交换和重新计算组成。然而，我们观察到*Capuchin*在Vgg16和BERT上非但没有表现出性能下降，反而有一些性能改进，如图9（a）和图9（f）所示。在Vgg16上，这是因为一些卷积层在较大的批处理规模下由于内存限制而退回到较慢的卷积算法。另一方面，*Capuchin*在前向传播中释放了大量的内存，从而获得了更多的自由内存来选择更快的卷积算法。对于BERT，我们观察到它的GPU利用率在批量大小为48时从31.7%上升到37.2%。
→
64.此外，我们发现在*Capuchin*中，批处理规模为200时，GPU的利用率为73.7%。因此，这个性能
改善的结果是更多的计算使GPU饱和。

6.4 对急切模式的评价

6.4.1 减少内存足迹

表3显示了TF-ori和*Capuchin*在急切模式下可以达到的最大批量大小。ResNet-
50的最大批处理量在急切和图形模式下分别为122和190。

这是由于在eager模式下缺乏一些优化技术，例如修剪和恒定折叠，这些技术只能在图模式下应用。总之，*Capuchin*在ResNet-
50和DenseNet上分别实现了2.46倍和2.71倍的批处理规模增量。

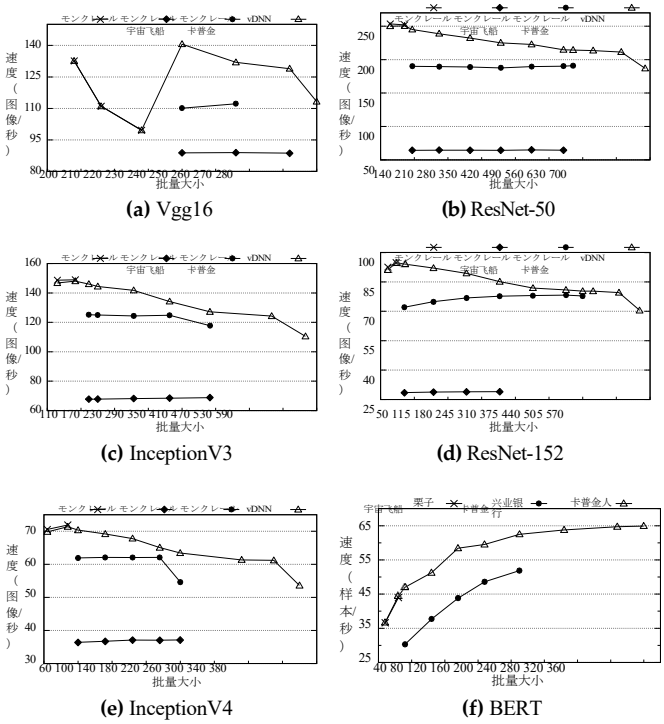
表3.急切模式下的最大批量大小

模型	Tensorfow	*卡普金人*
里斯网-50	122	**300**
密网	70	**190**

6.4.2 业绩

运行时间的开销。 *Capuchin*在ResNet-50和DenseNet上分别引入了1.5%和2.5%的运行时间开销。这个开销比图模式高一点，因为在eager模式下，操作的处理是顺序的，而在图模式下，有很多节点可以并行处理。因此，频繁的*锁定/解锁*会使执行停滞。

性能比较。 如图10(a)所示，当*Capuchin*将批量大小提高83.6%时，ResNet-50的性能开销为23.1%。由于急切模式并不像图形模式那样有效，所以性能下降的幅度比图形模式大一些。另一方面，我们可以看到，在图10(b)中，DenseNet在增加批处理规模时表现出了性能的提升。这与BERT类似。通过分析，我们观察到，GPU的利用率在在批处理规模为60→70时，性能从41.8%上升到45.3%。提高GPU利用率带来的性能改善掩盖了重新计算所带来的开销。

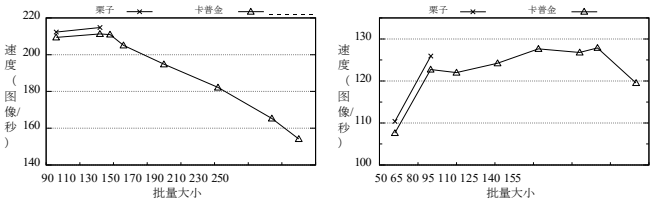


(a) Vgg16 **(b)** ResNet-50

(c) InceptionV3 **(d)** ResNet-152

(e) InceptionV4 **(f)** BERT

图9.图表模式下的性能



(a) ResNet50 **(b)** DenseNet

图10.渴望模式下的性能

7 相关工作

深度学习框架支持。 在目前主要的深度学习框架中，数据并行已经被广泛采用。每个GPU都有自己的网络副本。虽然它可以通过减少每个GPU的批处理量来减少GPU的内存占用，但最近的研究[34]表明，由于频繁的模型聚合，训练管道将在通信方面受到限制。模型并行化将整个神经网络分割给多个GPU，每个GPU分别负责自己的部分计算。数据和模型并行的权衡在[15, 28, 29, 32, 33]中得到分析。这种方法与*Capuchin*是正交的。

计算图依赖技术。 主要的大部分工作是基于computation graph进行内存优化，包括三类，即交换。

vDNN[24]、超级神经元[31]和[22]在前向阶段将数据交换给CPU，并在后向传播时预取数据，这些都是通过在实际运行前对原始计算图插入相应操作来实现的。[16]提出了一种层内内存重用和层间数据迁移的方法。

减少内存的占用。他们试图将这些数据重叠起来与计算时间的转移，这些都是基于对CNN不同层的特性的先验知识，如Conv层很耗时，而Pool、ReLU层的计算成本很低。然而，这个特性取决于设备的计算能力、GPU-CPU通道的链接速度和输入大小，这不是一个静态的属性。因此，当计算量不足以覆盖*陈等人*[5, 31]提出了一种重新计算的方法。

在前向阶段释放廉价计算的内存，并通过重新计算将其取回。然而，他们也忽略了同类型层之间的变化，因此不能充分提取内存减少的机会。CDMA[25]和Gist[13]通过利用特定神经网络架构的稀疏性来压缩数据，如ReLU-Pool，这些都是无损压缩方法。

Gist[13]也引入了一种有损压缩策略，即在前向阶段降低精度，在后向传播时将其还原。然而，这些工作是在算法层面上，与我们的工作是正交的。

计算图不可知技术。[7, 21]试图通过利用主机内存作为扩展内存来虚拟化GPU的内存。然而，这些工作对深度学习训练的特点并不敏感，因此，由于GPU和CPU之间的按需数据传输的巨大开销，导致性能不佳。[37]通过简化训练过程，采用了良好的内存交换算法。然而，这些工作都没有意识到计算的信息，因此，他们失去了在强大的GPU下有效减少内存的重要机会，也就是通过重新计算。

8 结论

本文提出了*Capuchin*，一个基于张量的GPU内存管理模块，通过张量驱逐/重取和重新计算来减少内存占用。*Capuchin*的主要特点是，它根据运行时跟踪的动态张量访问模式做出内存管理决策。这种设计的动机是观察到在训练过程中对张量的访问模式是有规律的。基于识别出的模式，我们可以利用总的内存优化空间，并对*何时和如何*执行内存优化技术进行灵活的控制。我们在Tensorfow中部署了*Capuchin*，并表明*Capuchin*可以在6个最先进的DNN中比原来的Tensorfow减少高达85%的内存占用。特别是，对于NLP任务BERT。*Capuchin*的最大批处理量比Tensorfow和梯度检查点的性能分别高出7倍和2.1倍。我们还表明，在相同的内存超额订购条件下，*Capuchin*比vDNN和梯度检查点的性能高出286%和55%。

鸣谢

我们真诚地感谢匿名审稿人的建设性意见。我们感谢沈西鹏和周永銮的宝贵意见。这项工作得到了国家重点研发计划（No.2017YFC0803700）和国家自然科学基金（No.61772218和No.61832006）的支持。
它还得到了美国国家科学基金会CCF-1750656和CCF1919289资助的部分支持。

参考文献

[1] Gradient-checkpointing. https://github.com/cybertronai/gradient-checkpointing.

[2] Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., Kudlur, M., Levenberg, J., Monga, R., Moore, S. 。Murray, D. G., Steiner, B., Tucker, P., Vasude- van, V., Warden, P., Wicke, M., Yu, Y., and Zheng, X. TensorFlow: A System for Large-Scale Machine Learning.In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)* (2016), vol. 16, USENIX Association, pp. 265-283.

[3] Agrawal, A., Modi, A. N., Passos, A., Lavoie, A., Agarwal, A., Shankar, A., Ganichev, I., Levenberg, J., Hong, M., Monga, R., and Cai, S. Tensorfow eager。一个多阶段的、嵌入python的dsl用于机器学习。In *Proceedings of the 2nd Conference on Systems and Machine Learning (SysML'19)* (2019).

[4] Chen, T., Li, M., Li, Y., Lin, M., Wang, N., Wang, M., Xiao, T., Xu, B. , Zhang, C., and Zhang, Z. Mxnet:a fexible and efcient machine learning library for heterogeneous distributed systems. *arXiv preprint arXiv:1512.01274*（2015）.

[5] Chen, T., Xu, B., Zhang, C., and Guestrin, C. Training deep nets with sublinear memory cost. *arXiv preprint arXiv: 1604.06174* (2016).

[6] Chetlur, S., Woolley, C., Vandermersch, P., Cohen, J., Tran, J., Catanzaro, B., and Shelhamer, E. cudnn: Efcient primitives for deep learning. *arXiv preprint arXiv: 1410.0759* (2014).

[7] Cui, H., Zhang, H., Ganger, G. R., Gibbons, P. B., and Xing, E.P.
Geeps:在分布式GPU上用GPU专用参数服务器进行可扩展的深度学习。在*第十一届欧洲计算机系统会议*（2016）上，ACM，第4页。

[8] Deng, J., Dong, W., Socher, R., Li, L.-J., Li, K., and Li, F.-F.Imagenet：一个大规模的分层图像数据库。In *Proceedings of IEEE Conference on Computer Vision and Pattern Recognition* (2009), pp. 248- 255.

[9] Devlin, J., Chang, M.-W., Lee, K., and Toutanova, K. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. *arXiv preprint arXiv:1810.04805*（2018）.

[10] Goyal, P., Dollár, P., Girshick, R., Noordhuis, P., Wesolowski, L., Kyrola, A., Tulloch, A., Jia, Y., and He, K. Accurate, large minibatch sgd。在1小时内训练imagenet。*arXiv预印本arXiv:1706.02677*（2017）。

[11] He, K., Zhang, X., Ren, S., and Sun, J. Deep residual learning for image recognition.In *Proceedings of the IEEE conference on computer vision and pattern recognition* (2016), pp. 770-778.

[12] Huang, G., Liu, Z., Van Der Maaten, L., and Weinberger, K. Q. 密集连接的卷积网络。In *Proceedings of the IEEE conference on computer vision and pattern recognition* (2017), pp. 4700- 4708.

[13] Jain, A., Phanishayee, A., Mars, J., Tang, L., and Pekhimenko, G. Gist: Efcient data encoding for deep neural network training.In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)* (2018), IEEE, pp.776-789.

[14] Jia, Y., Shelhamer, E., Donahue, J., Karayev, S., Long, J., Girshick, R., Guadarrama, S., and Darrell, T. Cafe。用于快速特征嵌入的卷积结构。在*第22届ACM国际多媒体会议*（2014）*论文集中*，ACM，第675-678页。

[15] Jia, Z., Zaharia, M., and Aiken, A. Beyond data and model parallelism for deep neural networks.在*第二届系统和机器学习会议（SysML'19）论文集*（2019年）。

[16] Jin, H., Liu, B., Jiang, W., Ma, Y., Shi, X., He, B., and Zhao, S. Layer- centric memory reuse and data migration for extreme scale deep learning- ing on many-core architectures.*ACM Transactions on Architecture and Code Optimization (TACO) 15*, 3 (2018), 37.

[17] Jouppi, N. P., Young, C., Patil, N., Patterson, D., Agrawal, G., Bajwa, R., Bates, S., Bhatia, S., Boden, N., Borchers, A., Boyle, R. 。Cantin, P., Chao, C., Clark, C., Coriell, J., Daley, M., Dau, M., Dean, J., Gelb, B., Ghaemmaghami, T. V., Gottipati, R., Gulland, W., Hagmann, R., Ho, C. R., Hogberg, D. 。Hu, J., Hundt, R., Hurt, D., Ibarz, J., Jaffey, A., Jaworski, A., Kaplan, A., Khaitan, H., Kille- brew, D., Koch, A., Kumar, N., Lacy, S., Laudon, J., Law, J., Le, D. 。
Leary, C., Liu, Z., Lucke, K., Lundin, A., MacKean, G., Maggiore, A., Mahony, M., Miller, K., Nagarajan, R., Narayanaswami, R., Ni, R,Nix, K., Norrie, T., Omernick, M., Penukonda, N., Phelps, A., Ross, J., Ross, M., Salek, A., Samadiani, E., Severn, C., Sizikov, G., Snel-
ham, M., Souter, J., Steinberg, D., Swing, A., Tan, M., Thorson, G. 。Tian, B., Toma, H., Tuttle, E., Vasudevan, V., Walter, R., Wang, W., Wilcox, E., and Yoon, D. H. In-Datacenter Performance Analysis of a Tensor Processing Unit。In *Proceedings of the 44th Annual*

International Symposium on Computer Architecture, ISCA 2017, Toronto, ON, Canada, June 24-28, 2017（2017）, pp.

[18] Kingma, D. P., and Ba, J. Adam: A method for stochastic optimization.
arXiv预印本arXiv:1412.6980（2014）。

[19] Lahiri, S. Complexity of Word Collocation Networks:A Preliminary Structural Analysis.In *Proceedings of the Student Research Workshop at the 14th Conference of the Association of European Chapter for Computational Linguistics* (Gothenburg, Sweden, April 2014), Association for Computational Linguistics, pp. 96-105.

[20] LeCun, Y., Bottou, L., Bengio, Y., and Haffner, P. 基于梯度的学习应用于文档识别。*IEEE论文集，86,* 11（1998），2278-2324。

[21] Li, C., Ausavarungnirun, R., Rossbach, C. J., Zhang, Y., Mutlu, O., Guo, Y., and Yang, J. A framework for memory oversubscrip- tion management in graphic processing units.In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems* (2019), ACM, pp. 49-63.

[22] Meng, C., Sun, M., Yang, J., Qiu, M., and Gu, Y. Training deeper models by gpu memory optimization on tensorfow.在*NIPS*（2017）*的ML系统研讨会论文集中*。

[23] Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A. 。Kopf, A., Yang, E., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., Bai, J.和Chintala, S. Pytorch:一个命令式的、高性能的深度学习库。In *Advances in Neural Information Processing Systems* (2019), pp. 8024-8035.

[24] Rhu, M., Gimelshein, N., Clemons, J., Zulfiqar, A., and Keckler, S.W.
vdnn:虚拟化的深度神经网络，用于可扩展的、内存有效的神经网络设计。In *The 49th Annual IEEE/ACM Inter- national Symposium on Microarchitecture* (2016), IEEE Press, p. 18.

[25] Rhu, M., O'Connor, M., Chatterjee, N., Pool, J., Kwon, Y., and Keckler, S. W.
压缩DMA引擎。利用激活稀疏度来训练深度神经网络。In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)* (2018), IEEE, pp. 78-91.

[26] Seide, F., and Agarwal, A.
CNTK：微软的开源深度学习工具包。在*第22届ACM SIGKDD知识发现与数据挖掘国际会议*（2016）*论文集中*，ACM。
pp.2135-2135.

[27] Simonyan, K., and Zisserman, A. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv: 1409.1556* (2014).

[28] Song, L., Chen, F., Zhuo, Y., Qian, X., Li, H., and Chen, Y. Accpar:用于异构深度学习加速器阵列的张量划分。在*2020年IEEE高性能计算机国际研讨会上 Architecture (HPCA)*（2020年），IEEE。

[29] Song, L., Mao, J., Zhuo, Y., Qian, X., Li, H., and Chen, Y. Hypar。迈向深度学习加速器阵列的混合并行。In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)* (2019), IEEE, pp.56-68.

[30] Szegedy, C., Vanhoucke, V., Ioffe, S., Shlens, J., and Wojna, Z. Rethinking the inception architecture for computer vision.In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition* (2016), pp. 2818-2826.

[31] Wang, L., Ye, J., Zhao, Y., Wu, W., Li, A., Song, S. L., Xu, Z., and Kraska, T.
Superneurons:用于训练深度神经网络的动态gpu内存管理。在*ACM SIGPLAN通告*（2018），第53卷，ACM，第41-53页。

[32] Wang, M., Huang, C.-c., and Li, J. Unifying data, model and hy- brid parallelism in deep learning via tensor tiling. *arXiv preprint arXiv:1805.04170*（2018）.

[33] Wang, M., Huang, C.-c., and Li, J. Support very large models using automatic datafow graph partitioning.In *Proceedings of the Fourteenth EuroSys Conference 2019* (2019), ACM, p. 26.

[34] Xiao, W., Bhardwaj, R., Ramjee, R., Sivathanu, M., Kwatra, N. 。Han, Z., Patel, P., Peng, X., Zhao, H., Zhang, Q., Yang, F., and Zhou, L.Gandiva: Introspective cluster scheduling for deep learning.在*第13届USENIX操作系统设计研讨会论文集 和实施（OSDI 18）*（2018），第595-610页。

[35] Xiao, W., Han, Z., Zhao, H., Peng, X., Zhang, Q., Yang, F., and Zhou, L.为基于gpu的深度学习工作调度cpu。In *Proceedings of the ACM Symposium on Cloud Computing* (2018), ACM, pp. 503-503.

[36] Yu, X., Loh, N. K., and Miller, W. A new acceleration technique for the backpropagation algorithm.In *IEEE International Conference on Neural Networks* (1993), IEEE, pp. 1157-1161.

[37] Zhang, J., Yeung, S. H., Shu, Y., He, B., and Wang, W. Efcient memory management for gpu-based deep learning systems. *arXiv preprint arXiv:1903.06631*（2019）.