

**ARK:分布式深度学习的GPU驱动的代码执行框架**

学生姓名： 张智皓

学生学号： 7020230508

专 业：计算机科学与技术专业

任课教师： 王耀彬老师

学院(部)：计算机科学与技术学院

**ARK:分布式深度学习的GPU驱动的代码执行框架**

**摘要**

现代最先进的深度学习(DL)应用程序倾向于扩展到大量并行GPU。不幸的是，我们发现GPU之间的集体通信开销通常是限制分布式DL性能的关键因素。它通过频繁的小数据块传输来充分利用网络带宽，这也会导致GPU上大量的I/O开销，从而干扰GPU的计算。其根本原因在于基于CPU的通信事件处理效率低下，以及无法通过GPU线程控制GPU内部的DMA引擎。

为了解决这个问题，我们提出了一个GPU驱动的代码执行系统，它利用GPU控制的硬件DMA引擎来进行I/O卸载。我们的自定义DMA引擎管道多个DMA请求，以支持高效的小数据传输，同时消除了GPU核上的I/O开销。与现有的仅由CPU启动的GPU DMA引擎不同，我们让GPU线程直接控制DMA操作，这导致了一个高效的系统，GPU驱动自己的执行流，并在没有CPU干预的情况下自主处理通信事件。我们的原型DMA引擎实现了从消息大小小到8KB(更好的吞吐量3.9倍)的行速率，只有4.3µs的通信延迟(更快9.1倍)，同时它对GPU上的计算几乎没有干扰，在实际训练工作负载中实现了更高的1.8倍的全减少吞吐量。

**目录**

[摘要 2](#_Toc5210)

[1.介绍 4](#_Toc10563)

[2.背景与动机 5](#_Toc24332)

[2.1.分布式DL中的小数据传输 5](#_Toc3930)

[2.2.外部执行控制开销 5](#_Toc15330)

[2.3.GPU端控制的I/O开销 7](#_Toc9890)

[3.ARK框架设计 8](#_Toc23369)

[3.1.GPU控制的DMA引擎 8](#_Toc19666)

[3.2.循环内核和虚拟CTA 9](#_Toc16171)

[3.3.离线调度 10](#_Toc29427)

[3.4.限制 11](#_Toc22545)

[4.实现 12](#_Toc1157)

[4.1.DMA引擎的实现 12](#_Toc5164)

[4.2.循环内核实现 14](#_Toc27873)

[5.评估 14](#_Toc14259)

[5.1.实验设置 14](#_Toc21657)

[5.2.DMA引擎性能 15](#_Toc20803)

[5.3.避免通信干扰 16](#_Toc16098)

[5.4.离线调度评估 17](#_Toc29214)

[5.5.张量平行推理 18](#_Toc12112)

[5.6.流水线并行训练 19](#_Toc20858)

[6.未来工作以及相关工作 20](#_Toc618)

[7.总结 20](#_Toc31557)

# 1.介绍

现代机器学习(ML)应用程序倾向于利用越来越多的加速器(特别是在这项工作中的GPU)[19,26]。最先进的深度学习(DL)算法通常需要扩展到数千个GPU，以获得更高的吞吐量和准确度[26]。不幸的是，这给整个系统带来了巨大的通信开销，延迟或干扰了数值计算，从而损害了GPU的使用。

通信开销主要体现在两个方面。首先，在大多数流行的DL算法中广泛采用的集体通信(如all-reduce、split -gather、all-to-all等)，通常将传输的数据分割成多个小块，用于流水线或发送到多个不同的目的地。当我们向外扩展时，块的大小趋于变小，这不利于有效利用网络带宽。其次，GPU常用的通信库，如NCCL[32]和RCCL[5]，通常会在GPU上产生严重的I/O开销。这是因为它们通常利用内存映射I/O (MMIO)来在GPU之间复制数据，这会消耗大量的GPU资源(即，核心周期和L2缓存/DRAM带宽)。我们观察到GPU上集体通信和数字计算的并行执行会严重干扰——在我们使用BERT-Large[10]的训练实验中，并行计算的吞吐量下降了45%，而仅达到峰值通信吞吐量的53.6%(详见2.3节)。

不幸的是，现有系统很难同时解决这两个问题(例如，小块的大传输延迟和GPU上的I/O开销)。我们可以通过将I/O转移到硬件DMA引擎而不是使用GPU线程的MMIO来避免I/O开销。然而，目前通用GPU上的DMA引擎都是由CPU线程发起的，通常是在通信的关键路径上注册CPU的控制。这将导致CPU - GPU同步开销，增大通信延迟，特别是对小数据块传输的吞吐量不利。事实上，人们可以在一个流行的DL框架中观察到数百个通信延迟，因为它利用了DMA引擎。类似地，如果不使用DMA引擎来进行数据块的通信，那么GPU上的通信就会受到高I/O开销的影响。

本文提出了GPU驱动的ARK系统，一种通信驱动的DL系统设计。GPU驱动系统的核心思想是在不受外部设备控制的情况下实现GPU代码的自主执行控制。这种机制通过允许GPU线程直接与远程GPU通信，而不需要任何外部控制信号，将每个GPU核的计算能力紧密地连接在一起，从而实现了低延迟通信。同时，为了避免GPU的I/O开销，我们设计了一个GPU控制的DMA引擎。具体来说，我们的自定义DMA引擎是直接由GPU线程启动的，这避免了沉重的MMIO，而无需CPU干预。

我们的评估表明，我们的DMA引擎原型对小消息特别有利，在低延迟(CPU干预下更快9.1倍)下实现了高通信吞吐量(8KB消息比cudaMemcpy高3.87倍)。此外，它不会干扰GPU的计算，GPU的计算和通信吞吐量都比使用基于MMIO的库提高[5,32](BERT-Large[10]训练中的全减少速度快1.8倍，见章节5.3)。

为了实现GPU驱动系统，我们还提出了一种高效的GPU自主执行调度器。我们的关键观察是，在线动态调度是不必要的，因为DL工作负载通常在运行时是确定的。相反，我们提出了虚拟协作线程阵列(vCTA)框架来抽象离线GPU调度。离线调度允许在重用现有前端接口和GPU内核实现的同时，消除后台的运行时调度开销。

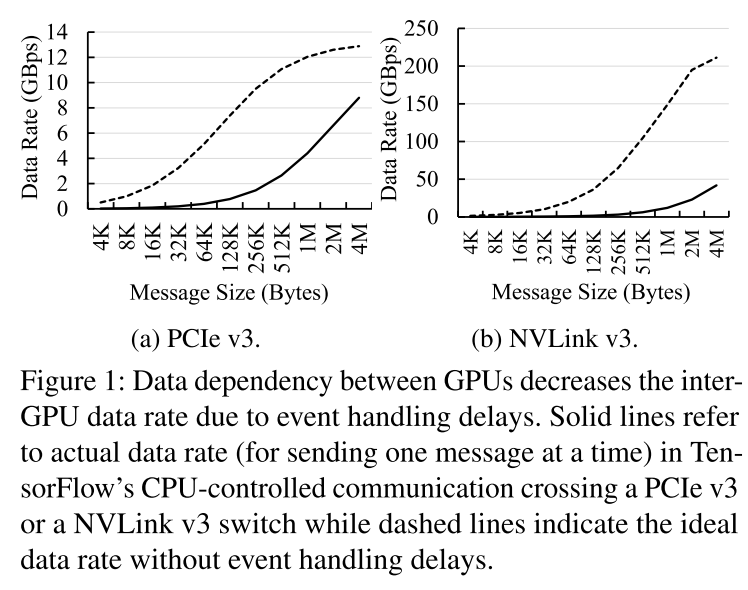
ARK为数据并行、张量并行和流水线并行支持高效灵活的并行执行模型。我们的评估表明，ARK在训练和推理方面都有显著的性能提高，分别实现了2.5倍和3.6倍的吞吐量提高。

# 2.背景与动机

本节解释现有的GPU间通信技术及其局限性。

## 2.1.分布式DL中的小数据传输

集体通信由多个通信原语组成，可在多个GPU之间并发地交换数据，被广泛应用于分布式DL中实现各种并行方法。常用的用例包括用于数据并行的all-reduce，用于张量并行的split-and-gather[22,40]，以及用于专家并行的all-to-all。随着使用的GPU数量的增加，在集体通信中，单位数据传输的大小变得越来越小，因为它将本地数据分割成多个块，交付给不同的GPU。这种较小的传输大小使得集体通信的整体性能高度依赖于每次数据传输前后的控制平面开销。不幸的是，我们发现无论是CPU控制的通信还是GPU控制的通信，控制平面的开销都非常大(参见第2.2节和第2.3节)。此外，现有的解决方案(例如，张量融合[39])批处理大量数据以避免小的传输，不能完全解决这个问题，因为它们通过故意延迟数据传输来权衡计算吞吐量。



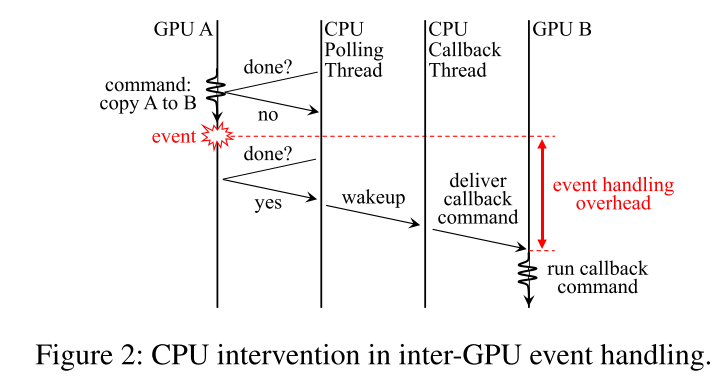
## 2.2.外部执行控制开销

现有的GPU程序执行严重依赖外部处理器(即CPU)提交GPU命令进行内核执行或数据传输。不幸的是，由于命令从主机端传递到GPU硬件队列(即流)的延迟，这种模型通常会带来很大的开销。我们可以使用传统的GPU事件接口(例如cudaEvent)来隐藏延迟，但它也会在事件处理上遭受大量的延迟。当采用GPU间通信时，我们称之为CPU控制通信(与NCCL[32]的GPU控制通信不同)，我们发现事件处理成为了超出数据传输本身的大通信延迟的主要原因。

现有的GPU程序执行严重依赖外部处理器(即CPU)提交GPU命令进行内核执行或数据传输。不幸的是，由于命令从主机端传递到GPU硬件队列(即流)的延迟，这种模型通常会带来很大的开销。我们可以使用传统的GPU事件接口(例如cudaEvent)来隐藏延迟，但它也会在事件处理上遭受大量的延迟。当采用GPU间通信时，我们称之为CPU控制通信(与NCCL[32]的GPU控制通信不同)，我们发现事件处理成为了超出数据传输本身的大通信延迟的主要原因。

### 2.2.1.控件的运行时干预

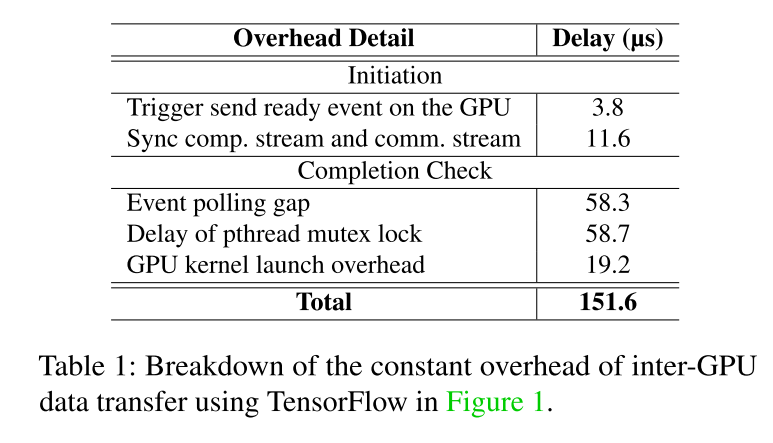
CPU可以作为一个中介，在两个通信的GPU之间传递一个事件。实际上，如果GPU位于不同的NUMA节点或不同的机器上，则需要CPU的运行时干预来进行通信。此外，一些框架(如TensorFlow)实现了一个通用接口，无论放置位置如何，GPU事件处理总是使用CPU。图2显示了GPU A将数据发送给计划运行下一个数据命令的GPU B时，CPU干预导致的事件处理开销。



我们注意到上面有三个地方。首先，CPU线程轮询GPU事件的效率很低，因为事件接口不允许CPU线程同时监视多个事件。当一个专用的繁忙等待的CPU线程被通知一个触发的GPU事件只需要大约3µs,2当一个应用程序必须运行许多并行任务时，这种方法不适用，因为这将运行许多轮询线程。相反，TensorFlow的事件轮询循环只使用一个CPU线程，平均轮询时间约为58.3µs(见表1)。其次，它需要唤醒调用被触发事件的回调函数的CPU线程。在TensorFlow中，从轮询线程释放互斥锁开始，回调线程获取互斥锁需要58.7µs的时间。如果两个线程运行在同一个CPU核上，这个延迟可以降低到5µs，但是将两个线程放在一起或合并到一个线程会增加事件轮询间隔和整体处理时间。最后，回调线程将计算命令传递给GPU B的效率很低。向GPU B发送事件信号只需要2 ~ 3µs，但我们还需要发送回调命令的二进制文件。我们可以避免额外的延迟，如果我们提前交付GPU命令，然后在CPU端触发它，但这是不支持的普通GPU。

### 2.2.2.异步控制

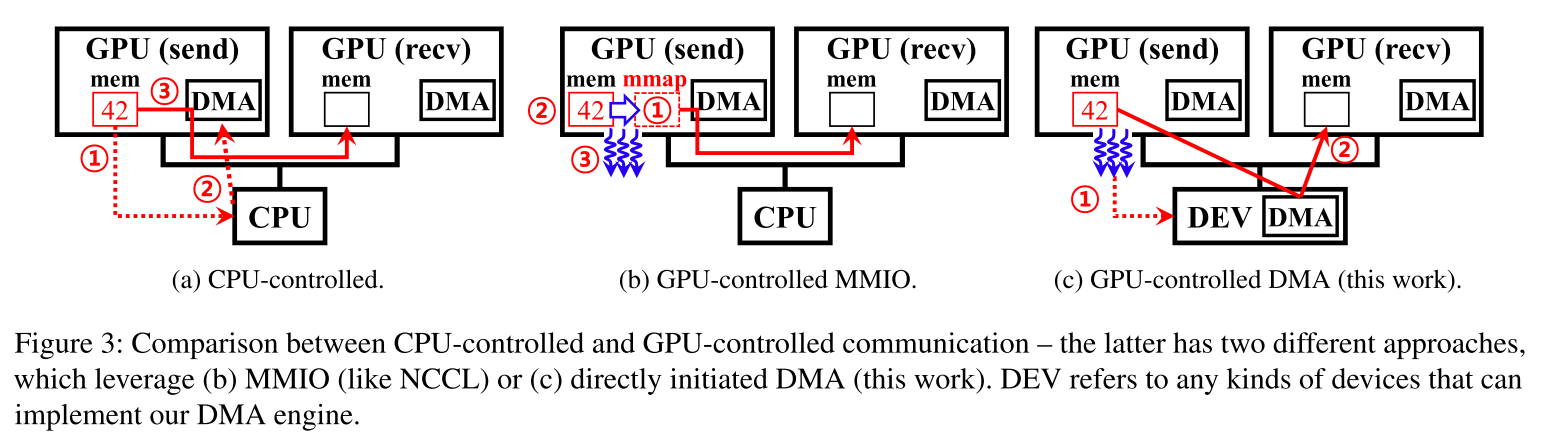
如果GPU在同一个NUMA节点下，CPU可以预留一个GPU事件异步触发，这样当事件发生时，GPU之间可以直接通信。在这种情况下，我们可以在实际事件发生之前将回调命令交给GPU B，然后使用常规的GPU事件接口(例如cudaEvent或更高级别的包装器，如CUDA Graphs[27])，通过GPU a的事件触发GPU B的回调命令。理想情况下，这应该只需要从GPU a向GPU b发送一个比特。然而，我们发现触发一个GPU事件(∼4µs)和唤醒一个相关的GPU命令(∼20µs)是令人失望的慢——它最终需要在运行时向GPU发送二进制命令。我们怀疑这是由于GPU处理事件的硬件实现效率低下。在TensorFlow中，这个开销会导致初始化一个依赖于GPU计算的传输延迟，如表1所示。



## 2.3.GPU端控制的I/O开销

既然CPU的介入会带来巨大的开销，那么如何管理与GPU本身的通信呢?NCCL[32]利用GPUDirect[31]来实现这种方式，它将GPU的内存空间公开为点对点访问，这样GPU线程就可以从另一个GPU上读写数据。由于GPU线程可以直接调用数据拷贝，它们可以在不需要CPU的情况下高效地处理通信事件。由于普通GPU硬件不允许GPU线程启动自己的DMA引擎，GPU控制的通信利用MMIO，当GPU线程写入映射数据时，它会隐式地执行DMA。图3比较了CPU控制和GPU控制的通信。前一个(图3a)采取以下步骤:1个CPU在数据准备好时被通知，2个CPU启动DMA引擎，3个DMA复制数据。另一方面，GPU控制的与MMIO的通信(图3b)如下:1 CPU在运行前创建目标GPU地址空间的内存映射(mmap)， 2运行时数据已经准备好，3 GPU线程将数据复制到mmap中，这将隐式地进行DMA复制。

不幸的是，GPU线程的数据复制经常严重干扰并行内核计算，特别是由于L2缓存污染和扭曲调度操作。具体来说，数据复制GPU线程需要将数据加载到寄存器文件中进行数据传输，但这将污染L2缓存，因为在普通GPU[34]上从DRAM读取数据时，无法绕过L2缓存。由于后者将DRAM上的数据直接复制到I/O总线(PCIe或NVLink)上，因此直接初始化DMA会导致严重的性能下降。此外，复制的线程经常发出“加载/存储”指令，使曲速调度程序繁忙，这使得其他并行计算的线程产生它们的时钟周期。尽管受影响的计算线程仅限于那些与数据复制线程共同运行翘曲调度器的线程，但它们会落后于其他线程，从而延迟整个内核。



为了分析竞争的影响，我们测量了两种不同的GPU内核，它们都只大量访问特定类型的GPU资源:分别是 L2缓存(读取1.96 TBps)和warp调度器(2.02 IPC)(所有数字在V100的GPU上测量)，同时使用8 \* V100的GPU与NCCL (v2.11.4) 64 MB的全采集内核并发运行。我们利用NVIDIA Visual Profiler (NVVP)和Nsight Compute来验证(1)L2缓存内核显示接近零的DRAM访问和L1数据缓存命中率，(2)warp scheduler内核显示接近零的L2缓存/DRAM吞吐量。我们还验证了计算核和全集核的并发性，实验过程中没有其他CPU/GPU活动。在本实验中，L2缓存和warp scheduler争用导致的慢速分别达到2.4倍和2.0倍，这使计算或并发NCCL通信变慢(当一方降级更少时，另一方往往受到更大的影响)。这一结果表明，根据并行计算内核的GPU资源使用情况，可能会出现严重的争用。

我们运行了一个微基准来评估在BERTLarge[10]模型的数据并行训练过程中NCCL all-reduce的争用情况。该模型一次执行32 MB的all-reduce，与8个GPU worker并行进行4 MB的数据传输。在一台使用8 \* V100 GPU(连接单个PCIe交换机(16 \* PCIe v3))的服务器上，并行计算吞吐量下降了45.0%，而all-reduce平均仅达到5.0 GBps，在没有干扰的情况下下降到峰值吞吐量的53.6%。在一台8 \* A100 GPU的服务器上(连接NVSwitch (NVLink v3))， all-reduce的速度更慢——并行计算吞吐量下降了14.3%，而NCCL all-reduce仅达到峰值吞吐量的30.9% (49.0 GBps)。

# 3.ARK框架设计

在本节中，我们将介绍ARK的设计，我们采用GPU驱动代码执行的方法，在没有CPU干预的情况下，避免GPU上的通信开销。

## 3.1.GPU控制的DMA引擎

我们声称，一个GPU控制的DMA引擎(图3c)可以消除通信开销，这反过来又成为我们的GPU驱动系统的基础。GPU控制的DMA引擎允许GPU线程在数据准备好时直接启动DMA操作(1)，这将立即将数据推入I/O总线，而不会浪费GPU周期(2)。我们利用现有的GPUDirect技术将GPU的物理地址空间暴露给我们的DMA引擎。

虽然GPU控制的DMA可以提供低延迟通信，而无需MMIO开销，但实现这个特性并不容易。事实上，一个理想的实现是修改GPU上现有的DMA引擎，以支持GPU控制的DMA，但这是不可行的，因为我们不能更新GPU硬件。相反，我们考虑使用一个外部设备，如图3c所示，但代价是GPU线程的额外通信延迟。

尽管有性能上的好处，但在许多现有系统中，为GPU控制的DMA引擎采用新的硬件可能代价高昂。为了提供一个临时解决方案，我们采用一种通用的DMA引擎设计，它可以在任何硬件平台(例如CPU、GPU、SmartNIC、FPGA等)或I/O总线类型(PCIe、NVLink[33]或Infinity Fabric Link (xGMI)[3])上作为软件或硬件实现。无论使用哪种平台，所有的实现都需要为GPU内核共享相同的运行时接口。DMA接口还应支持低延迟和灵活性，同时满足软硬件引擎的不同要求。

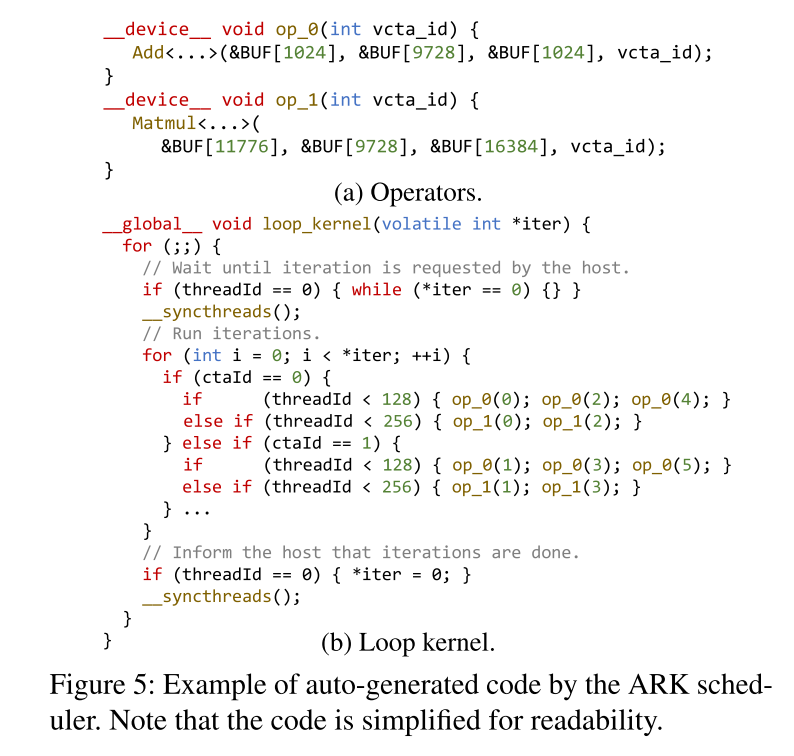
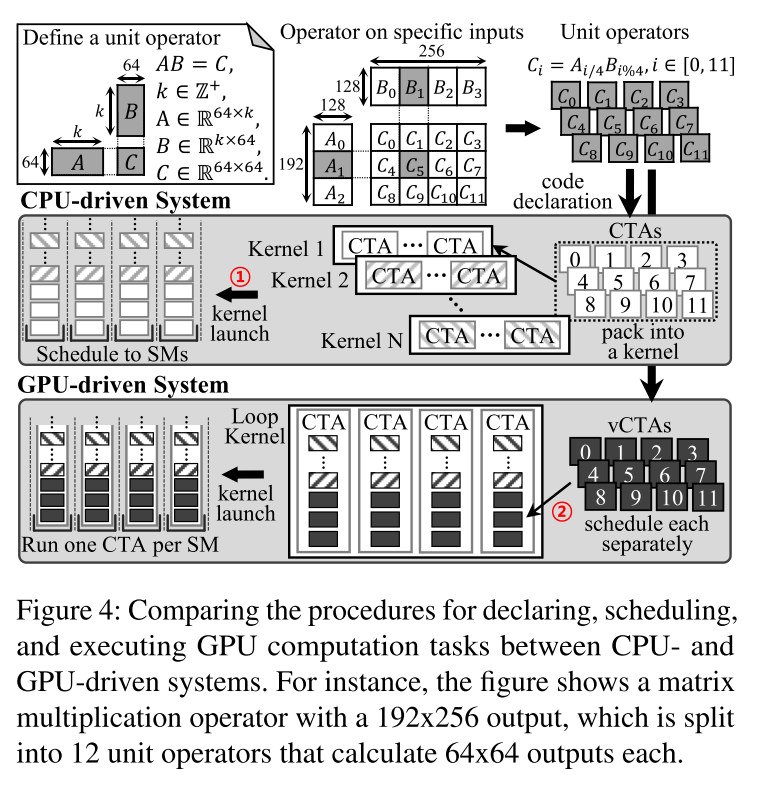
在本文中，我们提出了一个GPU控制的DMA引擎的软件实现和硬件原型。我们的软件引擎可以在任何现有的系统上工作，而不需要额外的硬件，因为它利用了主机CPU内核——繁忙等待的CPU线程从GPU读取DMA请求，并相应启动DMA。该设计符合GPU驱动系统的原理，GPU线程直接发起数据传输，而CPU线程只机械发起数据拷贝，不需要GPU事件处理和GPU资源消耗。我们的硬件引擎原型是在FPGA上实现的，我们展示了硬件部署相对于软件引擎的潜在好处。我们在第4.1节中解释了DMA引擎实现的细节。

## 3.2.循环内核和虚拟CTA

GPU控制的DMA引擎将很容易被现有系统采用，例如，NCCL可以通过启动我们的DMA引擎来取代它的MMIO。然而，现有的系统不能充分利用GPU控制通信的好处，因为通信API是由CPU启动的——CPU的干预障碍仍然存在于计算和通信之间。

为了消除这个障碍，我们提出了一个GPU驱动的代码执行系统，它可以在单个内核(称为循环内核)中运行整个DL应用程序。我们的关键观察是，在线动态调度是不必要的，因为DL工作负载通常在运行时是确定的。我们的GPU驱动系统在编译时自动将所有内核合并成一个循环内核(每个GPU一个)，并在应用程序启动时只启动一次，而不是在运行时用CPU动态地启动GPU内核。然后，循环内核在应用程序的整个生命周期中持续运行。循环内核是由代码生成器生成的，它读取DL应用程序的运行图，并自动组装相应的GPU运算符代码片段来构建循环内核代码。我们把这种代码生成称为离线调度，因为所有的GPU操作符都是通过代码静态地分布在GPU核心上，或者是流式多处理器(SMs)。离线调度让GPU有效地控制应用程序，这将最小化GPU间通信的事件处理开销。我们将在第4.2节讨论循环内核的几个技术细节。

图4显示了循环内核设计背离了传统的声明、调度和执行GPU任务的框架。在CPU和GPU驱动的系统中，GPU运算符通常定义为多个单元运算符的集合，每个单元运算符以SIMD方式计算整个输出的一部分。同时，两种系统在GPU代码中声明操作符的方式不同。CPU驱动的系统将每个单元操作符声明为一个协作线程数组(Cooperative Thread Array, CTA)8，将整个操作符声明为一个单独的内核，这需要为多个操作符启动多个内核。相反，我们的GPU驱动系统不允许使用多个内核，因为它在单循环内核中执行所有操作符。相反，它利用作为循环内核CTA一部分的单元操作符的中间声明，我们称之为虚拟CTA (vCTA)。



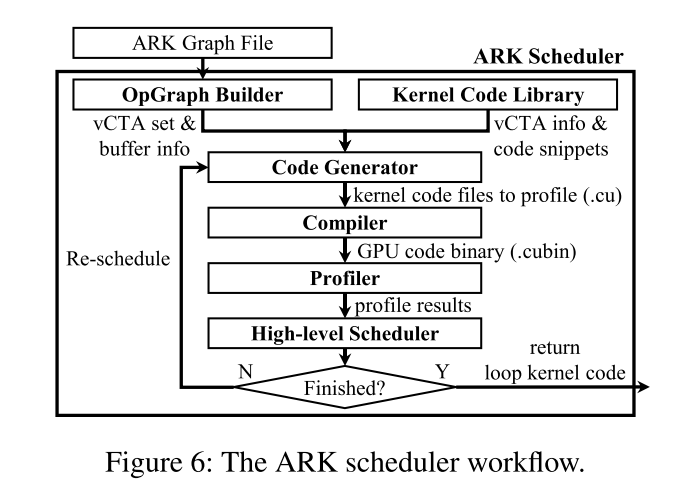
vCTA为ARK中的离线调度提供了关键的抽象，它支持软件定义的SM调度。vCTA声明一个单元操作符的代码，该操作符与循环内核中的特定SM密切相关。CPU驱动的系统依赖于在内核启动时跨SMs分发CTA的非可编程硬件调度器(图4中的①)，一个GPU驱动的系统实现了一个自定义逻辑，在CTA之间分配vCTA(图4中的②)。通过每个SM启动一个CTA，分配每个CTA使用一个SM的全部资源，ARK可以以一种可编程的方式控制vCTA的SM-affinity。这可以实现细粒度的GPU调度，这对于GPU驱动系统实现各种计算优化技术非常有用，如算子融合[17,24,35]。

将现有的代码移植到ARK是很简单的，因为ARK可以以最小的修改重用现有的GPU内核实现:将CTA ID (CUDA中的blockIdx)、线程ID (CUDA中的threadadidx)、sm本地内存地址(CUDA中的共享内存)和同步函数(例如，CUDA中的\_\_syncthreads())替换为ARK框架提供的相应常量或函数。这一修改保证了我们广泛验证的框架的正确性。

如图4所示，脱机调度将每个vCTA的代码片段写入只有特定CTA(或SM)进入的循环内核的if-branch中。由于每个CTA静态地执行计划离线的特定vCTA, GPU实际上运行一个静态while()循环，而不是动态地控制——vCTA内部繁忙轮询循环处理运行时事件。例如，在图5b中，CTA 0 (CTAId为0)和CTA 1 (CTAId为1)分别从操作符op\_0分配了三个vta，从操作符op\_1分配了两个vta。每个CTA使用256个线程，来自op\_0的vCTA由线程0 ~ 127顺序执行，而来自op\_1的任务则由线程128 ~ 255执行(这意味着每个vCTA的实现使用128个共同工作线程)。每个vCTA都是通过将特定的vCTA ID传递给GPU函数来声明的，该函数定义了一个操作符，如图5a所示。ARK的内核代码库提供了通用操作符的实现(图中的Add或Matmul)，这些操作符将数据块的地址和vCTA ID作为运行时参数框架为每个数据块分配适当的偏移量给全局GPU缓冲区(BUF)， vCTA ID定位vCTA处理的数据块的特定部分。

## 3.3.离线调度

图6显示了ARK中的调度工作流。总的来说，它读取DL模型的DAG并生成相应的循环内核代码。ARK调度器由一个高级调度器和一个分析模块组成。高级调度器实现了操作符与模块提供的分析结果的融合。在初始阶段，它构建一个OpGraph，该OpGraph指出模型中的所有操作符及其依赖关系，并生成代码来分析所需的所有类型的vCTA。然后，高级调度器使用分析结果生成它的第一个调度决策。决策可能包含多个不同的候选者，需要对它们进行分析以选择最快的一个，然后迭代整个过程以与多个其他候选者进行比较，这可能需要额外的分析。当只剩下一个候选对象时，调度器最终返回循环内核。



**1.减少剖析器中的编译。**由于代码生成器使用静态的vCTA-SM亲和度进行确定性调度，因此只需剖析vCTA的性能，就可以准确估计每个调度决策的性能(即延迟和核心资源使用)，减少了评估调度策略的编译。假设有n个并行运算符，每个运算符有m个不同的单位运算符(或称vCTA)实现，那么需要编译多达O(m^n)个不同的内核来找到最佳的融合决策。由于这个数字可能是不合理的大，现有的作品已经开发了启发式，只关注有前途的候选人[17]。。

乍一看，这似乎只需要O(nm)内核来进行vCTA计算，但它更复杂，因为当vCTA在同一SM上并发运行时，通常比串行执行时完成得更快，我们说它们具有联合效率。联合效率的产生主要有两个原因:(1)因为L1缓存命中率提高了，因为它们访问运行在同一SM上的公共内存空间，(2)因为一个vCTA的执行隐藏了另一个vCTA的内存访问(反之亦然)，从而提高了ALU和LSU的同时利用率。第一种情况经常出现在同一运营商的vCTA中，而第二种情况在大多数vCTA中普遍存在，即几乎所有的vCTA之间都具有联合效率。

考虑到联合效率，通常情况下，我们需要测量不同类型的vCTA在同一SM上共同运行时的延迟，这需要对每个vCTA进行一次内核编译。假设在一个SM中最多可以同时运行k个vCTA，那么编译次数的复杂度为。实际上，这比O(m^n)要小得多，因为k通常是一个小于等于4的小常量，这是由于SM资源的限制(最大线程数、共享内存字节数和寄存器数)。

**2.SM负载均衡代码生成器。**代码生成器通过跨SMs分发并行的vCTA来平衡它们的工作负载，从而自动最大化循环内核的SM利用率。不幸的是，由于联合效率的关系，找到最优的负载平衡是一个np -困难的问题。由于同时安排大量的vCTA，暴力搜索将花费不合理的时间。

为了解决这个问题，我们利用一个现有的图分区算法在SM上实现了一个启发式的负载平衡。图划分是一种常见的负载均衡问题，它通过切割多条边将一个图分割成给定数量的子图，同时实现两个目标:(1)平衡子图的总节点权值，(2)最小化切割边的总权值。我们将SM负载平衡问题表示为一个图分区问题。具体来说，我们首先对需要跨SMs分发的独立的vCTAs进行分组。每个组表示为一个图，其中每个节点表示一个vCTA，每个边表示连接节点(即vCTA)具有连接效率。节点权值是在SM上运行vCTA的延迟，边缘权值衡量的是联合效率，即在同一SM中同时运行两个vCTA与连续运行两个vCTA所减少的延迟的分数。

然而，它花费了太长的时间来运行分区，因为它产生了太多的边——因为几乎所有的vCTA彼此之间都有连接效率，图变得几乎是一个网格连接。为了加快算法的速度，我们采用超图表示[2]，而不是普通图，它将多个节点的等权网格连接表示为一条称为超边的单边。幸运的是，这种表示大大减少了代码生成的时间，特别是当我们使用大批处理(这会创建大量的vCTA)时，从几十小时减少到几秒。

## 3.4.限制

基于vCTA的调度采用了一种白盒方法，它假设所有的运营商都是开源的，因此ARK不能调度像cuDNN[28](类似于Rammer[24])这样的开源二进制文件。另外，ARK的离线调度器只支持静态计算图，这比PyTorch的动态图[13]灵活性差。然而，这样的限制在很多流行的框架中都很常见，包括TensorRT[35]和ONNX Runtime[25]。

# 4.实现

这一节介绍ARK的细节。

## 4.1.DMA引擎的实现

我们首先介绍了我们的DMA引擎接口，然后介绍了我们的软件和硬件DMA引擎。

### 4.1.1.接口

我们的接口设计的关键考虑是确保高通信性能，同时保持跨软件和硬件平台的接口一致。其中一个关键问题是设计GPU的DMA请求消息，我们称其为发送请求(SR)，因为它对性能和实现复杂性有重大影响。在硬件方面，接收一个大的SR，其大小超过了数据总线的宽度(在现代64位处理器中是64位)，将需要多个周期，这将需要SR缓冲区管理，重新组装分段的SRs，处理掉落的SRs(由SR缓冲区溢出引起)。在硬件上实现它们将使逻辑变得非常复杂，并增加空间成本。由于在硬件上实现它们将极大地复杂化逻辑并增加空间成本，所以我们为软件和硬件引擎共享8字节的SR设计。虽然在8字节内保存一个通用内存副本(两个地址和一个副本长度)的元数据是一个挑战，但我们通过采用少量的send/recv缓冲区来解决这个问题，它通过用一些缓冲区索引替换通用的8字节地址来减少地址空间。这是可行的，这要归功于集体通信的静态特性，其中通信实体是固定的——它允许离线预调度数据传输，这样接收者就知道哪些数据到达了哪个缓冲区，而不需要在运行时接收任何额外的元数据。同时，不同缓冲区上的DMA请求被流水线化，以实现低延迟和高吞吐量。

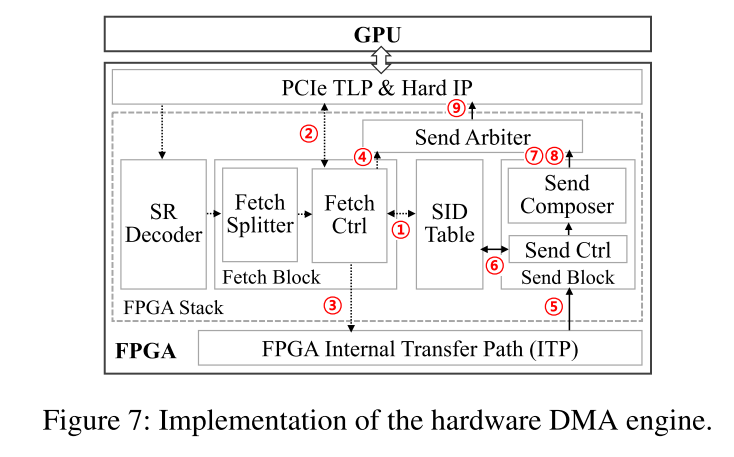
在软件方面，保持一个SR缓冲区将是更有效的，因为它将需要额外的控制，以防止覆盖以前的SR。也就是说，不像一个硬件实现，完全接收的SR可以立即触发内部DMA管道在每个周期，一个软件线程可以覆盖一个未读的SR，除非发送方(GPU)与接收方(DMA堆栈)在发送一个新的SR之前协调。这样的协调会导致额外的延迟，因为GPU需要在发送SR之前读取DMA堆栈上的一个远程标志。我们通过为SR维护一个专门的环形缓冲区来解决这个问题，GPU在发送SR之前只检查缓冲区头的本地副本，并且这个副本会被DMA堆栈异步更新。这消除了通信关键路径上的协调延迟，同时为软件和硬件引擎提供一致的SR接口。

### 4.1.2.软件引擎

我们的软件引擎使用CPU作为数据平面，GPU作为控制平面。我们实现了一个CPU线程，它忙碌等待SRs，并相应地调用cudaMemcpy或RDMA写入，也就是说，它利用了发送方GPU上现有的硬件DMA引擎。注意，这与CPU控制的通信不同，因为我们只将CPU用于数据平面操作，而控制平面(事件处理)是由GPU线程管理的。对于高吞吐量，繁忙等待循环耗尽环缓冲区中的所有SR，并调用copy一次，以便在连续内存空间上发送。此外，CPU-GPU通信使用MMIO，而不是缓慢的cudaEvent，可以发送SR、SC(发送完成)和RC(接收完成)信号，只需2 ~ 3µs。

另外，软件引擎可以使用CPU线程执行MMIO，而不是初始化硬件的DMA引擎，这样可以减少cudaMemcpy开销(例如，从CPU向GPU的DMA引擎发送一个拷贝请求)。然而，这种方法在大多数主机CPU体系结构中无法实现线路速率，因为其通过PCIe根复合体的吞吐量较差[441,44]。这个问题可能会在未来的CPU架构中得到解决，或者通过利用SmartNICs[4]上的ARM内核来解决，这是我们未来的工作。

### 4.1.3.硬件引擎



我们用FPGA实现了一个用于DMA操作的自定义硬件，与我们的软件引擎原型相比，它提供了两个好处。首先，我们的硬件引擎避免了cudaMemcpy开销带来的额外通信延迟，因为它直接执行DMA。其次，与GPU上现有的硬件DMA引擎不同，我们的自定义硬件实现了多个并行DMA操作的流水线。这有助于实现高数据速率，即使是发送小数据块。表2显示了我们在Intel Arria 10 FPGA上实现的资源使用情况。

请注意，我们的FPGA原型仅限于支持两个GPU之间的通信，它不支持NVLink，因为没有可编程硬件(或现成的设备)可以连接到NVLink。相反，我们认为它是一种概念证明，展示了理想的好处，而不是一种可以大规模部署的实用设备。更实际的实现将通过CPU、GPU或智能技术的未来发展而实现。

图7展示了FPGA上GPU间通信栈的硬件结构。与现有的GPU DMA引擎不同，我们的DMA堆栈被设计成将多个DMA请求与不同的SID同时处理。这是通过将一个长请求分解为多个短请求来实现的，避免了头线阻塞，提高了GPU同时发送多个不同数据时的PCIe吞吐量。我们将分别解释发送方和接收方堆栈如何处理每个请求。

**1.发送方。**当发送端堆栈收到一个SR时，Fetch Block读取解码后的SR并获取请求的SID，通过查找SID表(①)将其转换为物理源GPU地址。使用这个地址，Fetch Ctrl一次获取一个子请求，如果复制长度很长，它可能会获取多次请求。每个子请求从GPU读取相应的源数据，并将其存储在Fetch Ctrl的FIFO缓冲区(②)。当源数据从GPU完全读取完成后，存储的数据和子请求通过FPGA内部传输路径(FPGA Internal Transfer Path, ITP)转发给接收栈。(③)。在处理了所有来自SR的子请求后，Fetch Ctrl给Send仲裁器一个SC标志，这个标志将被写入GPU端SC标志。(④)。

**2.接收方。**接收栈从发送栈接收子请求，并将数据存储到Send Ctrl的FIFO缓冲区(⑤)。同时，子请求中的SID信息被转换为物理目的GPU地址(⑥)。Send Ctrl发送数据到目的地址，当它完成时，Send Composer发送一个RC标志给Send仲裁器，这将被写入GPU端RC标志(⑦，⑧)。

**3.资源使用和限制。**我们在Intel Arria 10 FPGA[16]上实现了DMA栈。从表2可以看出，每个栈的实现成本都很低，只使用了14253个alm和188个M20K BRAMs。请注意，我们当前的实现通过直接连接对应FPGA堆栈的FPGA ITP接口，只支持两个GPU之间的通信。我们的设计考虑利用DUA[41]来支持多个堆栈(机器内或机器间)之间的路由，但我们把它留作将来的工作。

## 4.2.循环内核实现

本节解释了在ARK中优化循环内核性能的几个细节。

**1.每线程寄存器优化。**GPU内核经常通过评估运行更多线程(获得更多并行性)和运行更少线程(每个线程有更多寄存器)(获得更多计算吞吐量)之间的权衡来调整每个SM的并发线程数。因此，循环内核也需要对其进行调优。ARK调度程序生成循环内核的多个版本，每个线程有不同数量的寄存器，并选择性能最好的一个。实际上，在NVIDIA的GPU中，由于硬件的限制，只有32、64、128和256可供选择。

**2.依赖于GPU架构。**章节3.2解释了ARK每个SM启动一个CTA，但根据GPU架构的不同，它可能会为每个SM启动两个或更多的CTA。这是因为在某些体系结构中，一个CTA可能被限制使用SM的全部资源。在这种情况下，我们需要为每个SM启动两个CTA，以使用整个SM资源。ARK调度程序自动分析循环内核的资源需求，并相应地确定每个SM的CTA数量。

**3.程序大小。**我们通过合并多个相同的单元运算符来减少循环内核的程序大小，例如，如果一个模型由多个卷积运算符组成，那么实际上只会定义几个唯一的卷积实现，这些实现在所有的运算符之间共享。因此，程序的大小仅弱地依赖于模型中操作符的数量。相反，它受制于操作员实现的总规模，这是非常有限的——例如，在单个GPU架构上，cuBLAS只提供了一个矩阵乘法实现的约10个实例，而一个循环内核可以容纳超过5000个实例。这应该涵盖任意一个DL程序，因为矩阵乘法实现的大小是DL中最流行的操作符之一。

# 5.评估

我们通过在三个不同的方面将ARK与现有的DL框架进行比较来评估它。首先，ark的快速GPU间通信有助于提高端到端吞吐量和降低DL应用的延迟。其次，在不损失GPU计算吞吐量的情况下，获得了通信上的优势。第三，ARK可以灵活地支持各种并行策略，包括数据并行、张量并行和流水线并行。

## 5.1.实验设置

**1.软件引擎。**对于使用软件DMA引擎的实验，除非另有说明，否则我们使用两个Intel Xeon Gold 6240R CPU(每个CPU 48个 lcores, 2.40 GHz)和8个NVIDIA V100 GPU。我们在机器中有两个NUMA节点，但只有一个NUMA节点承载所有GPU，即节点0连接两个PCIe v3交换机到其PCIe根节点复合体，每个交换机直接连接到4个GPU。对于多节点实验，我们使用4个Azure NDv4 SKUs[7]和32x NVIDIA A100 GPU(每个节点8个)，每个GPU专用200 Gbps的NVIDIA Mellanox HDR InfiniBand连接。

**2.硬件引擎。**对于使用硬件DMA引擎的实验，我们使用了Intel Xeon Gold 5118 CPU (24个 lcores, 2.30 GHz)、两个NVIDIA V100 GPU和一个Intel Arria 10 FPGA。GPU和FPGA都在同一个PCIe v3交换机后面。我们只在第5.2节和第5.5节的实验中使用硬件引擎。

## 5.2.DMA引擎性能

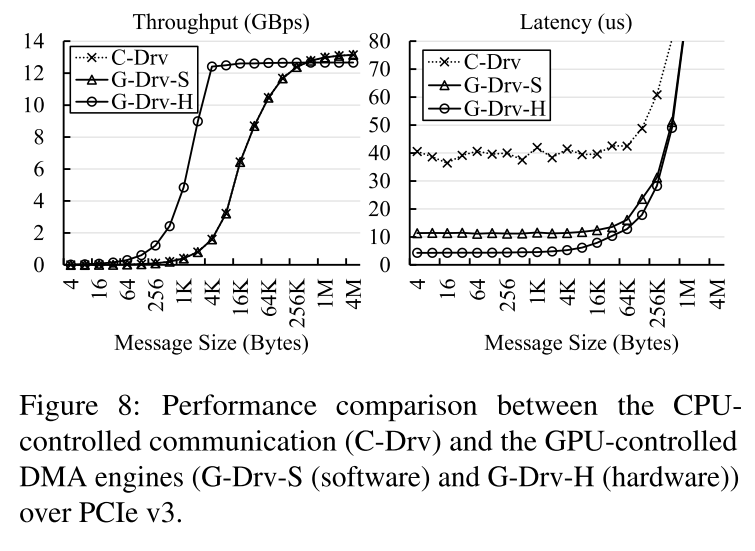


图8比较了两个GPU与DMA引擎(G-Drv-S和GDrv-H)在CPU控制通信基线(CDrv)上的通信性能。C-Drv是我们自己对典型的CPU驱动系统的最小实现，但与TensorFlow不同的是，C-Drv在事件仅由GPU使用时利用cudaEvent实现异步控制，这进一步减少了CPU-GPU的同步，以加速GPU间的通信。

我们通过同时发送许多并行消息来度量吞吐量，并报告使用不同消息大小所实现的最大吞吐量。对于延迟测量，我们实现了一个乒乓应用程序，并报告单向延迟——与吞吐量测量不同，它包括通信事件处理延迟。这个实验假设了CPU控制基线的一个有利的场景，我们可以采用异步控制(在2.2.2节中解释)。在这个场景中，单程旅行只需要触发两个GPU事件和两个流同步。

在图8的左图中，我们的软件引擎(G-Drv-S)显示了与C-Drv相同的吞吐量，因为两者都使用cudaMemcpy作为数据平面。相比之下，我们的硬件引擎(G-Drv-H)显示出了巨大的吞吐量改进，仅用8 KB消息饱和带宽，而G-Drv-S需要4 MB消息饱和。这是因为硬件DMA引擎管道处理多个DMA请求，而cudaMemcpy不能。当GPU同时向不同的目的地发送多个消息时，这种改进将特别有益，例如，为了专家并行，所有对所有通信，这是最先进的基于transformer的模型[11]的流行扩展。

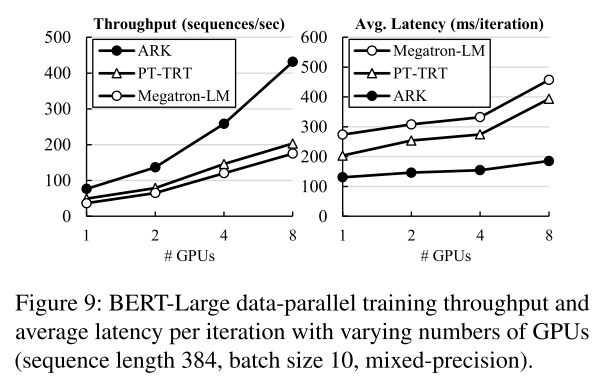
我们注意到G-Drv-H的最大吞吐量比G-Drv-S低3.68%。这是因为外部DMA堆栈需要分别向发送方和接收方GPU发送读和写请求，而发送方GPU上的本地DMA引擎只需要发送写请求。但是，由于差距很小，它不会对端到端应用程序的性能产生太大的影响。

图8的右图显示，C-Drv的单向潜伏期平均至少为~ 39.3µs。相比之下，G-Drv-S和G-Drv-H分别实现了3.5倍和9.1倍的延迟。这是因为我们的DMA引擎直接在GPU线程中处理通信事件，而C-Drv依赖cudaEvent接口来触发事件和同步流，而cudaEvent接口的开销很大。当GPU执行分割和收集中间结果以分配工作负载时，这种改进将特别有益，如张量并行[22,26]。需要注意的是，我们的DMA引擎的优点是GPU周期消耗很少。我们将在下一节中对其进行评估。

## 5.3.避免通信干扰

为了比较使用NCCL和使用我们的DMA引擎对计算和通信的干扰，我们通过训练具有代表性的NLP模型来评估ark的数据并行训练吞吐量。

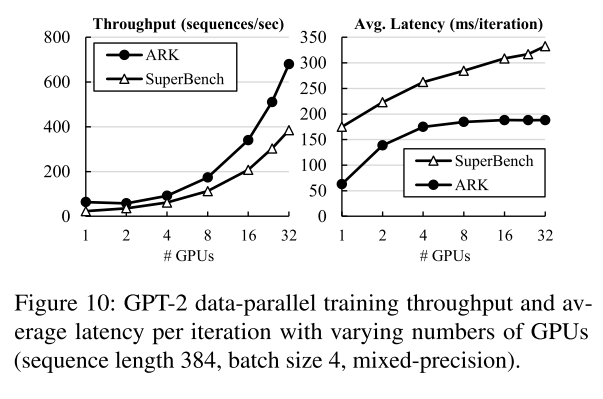
**1.基准。**PT-TRT通过采用TensorRT[35]来加速PyTorch[12]，它不能向多台机器扩展。Megatron-LM[26]是一个基于pytorch的框架，支持NLP模型的大规模训练，但我们只在这里使用单节点实验。SuperBench[42]也基于PyTorch为系统性能评估提供了正式的DL基准，我们使用PyTorch进行多节点实验。所有基线都利用NCCL[32]进行通信。



**2.单节点。**单节点实验使用8x V100 GPU训练BERT-Large[10]模型，如图9所示。从图中可以看出，在8个GPU的情况下，ARK的性能分别比Megatron-LM和PTTRT高出2.46倍和2.12倍。我们发现加速的两个原因。

首先，NCCL在反向传播过程中会对计算吞吐量产生不利影响，而ARK不会，因为它利用DMA而不是GPU线程进行数据复制。具体来说，当NCCL操作由于mmio与反向传播计算的干扰而减慢时，在8个GPU的情况下，ARK与PT-TRT之间的端到端差距达到64.5%，显示全减少吞吐量仅为5.0 GBps。我们发现，NCCL内核导致了整个反向传播计算的45.0%的延迟，从107.63 ms增加到156.02 ms。另一方面，通过直接启动DMA而不是使用MMIO，我们的DMA引擎受到了几乎为零的干扰，实现了9.10 GBps的全减少吞吐量(快1.82倍)。

其次，ARK在GPU上实现了更高效的计算。例如，对于PT-TRT大约37.8%的计算时间，它在每次迭代中执行1.2万个内存密集型内核，比如逐元素运算或intraGPU数据移动。将这些操作符作为单独的内核运行是低效的，因为这将导致不必要的内核启动和GPU内部同步。ARK在很大程度上减少了这种开销，因为它将所有的运算符调度在一个单循环内核中，类似于算子融合[17,24,35]。



**3.多节点。**多节点实验使用最多32x A100 GPU来训练GPT-2[36]模型，如图10所示。所有结果只使用InfiniBand进行通信(不使用NVLink)，并使用环约简算法。图中显示，在32个GPU的情况下，ARK的性能比SuperBench高出1.77倍。此外，虽然SuperBench的每次迭代延迟都在不断增加，但ARK的增量只是边际的。这显示了我们在NCCL上的通信堆栈的效率，它最小化了通信和计算之间的干扰。我们还发现，即使在没有通信的情况下(当使用单个GPU时)，ARK也有很大的计算优势，这将在下一节中进一步解释。

## 5.4.离线调度评估

本节展示了ARK的离线调度程序可以生成与现有DL优化技术相当甚至更好的GPU内核。我们不是在DL优化中宣称最先进的性能，而是想表明我们的GPU驱动系统的通信增益不会带来任何计算性能下降。

我们比较了使用单个GPU在不同框架下流行DL模型的推理性能。DL模型包括图像分类(ResNet-50[14]和GoogLeNet[43])、对象检测(SSD[23])和NLP (BERT-Large[10])模型。TensorFlow (TF)是ARK的主要比较目标，因为它支持像ARK这样的DL应用程序的灵活并行性。我们还将其与TensorFlow-XLA (TF- xla)[1]进行了比较，后者在TF后端实现了自动操作员融合，但这并不总是有利于性能，因为融合的内核可能比使用供应商提供的内核(如cuDNN)的性能更差。Rammer[24]和TensorRT实现了优化的算子融合，通常优于TF或TF- xla，但它们只支持有限的并行性。例如，TensorRT只支持节点内的数据并行性，通过采用它来加速其他框架，如TF和PyTorch，因为TensorRT本身不支持分布式执行。灵活的[20]为GPU内核提供了谨慎的异步控制(或提前调度)，以减少内核启动和GPU事件的运行开销。然而，正如第2.2.2节所解释的那样，异步控制被限制在处理通信开销上。《Nimble》目前也只能在单个GPU上运行。

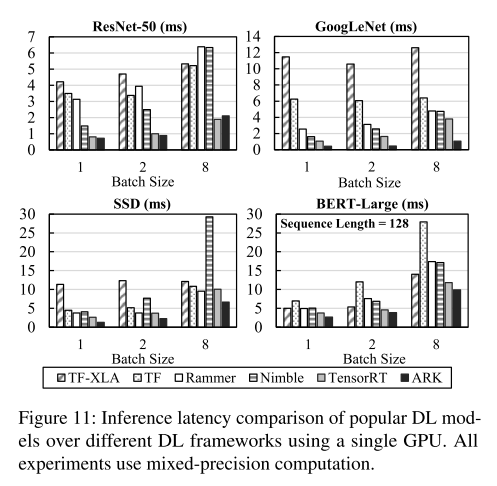


图11显示了ARK在大多数情况下对现有框架实现更快的单GPU推理。例如，方舟显示出比TensorRT低1.11x ~ 3.56x的延迟，但批量大小为8的ResNet-50却比TensorRT差~ 9.90%。这是因为我们的矩阵乘法核比本例中TensorRT中使用的cuDNN[28]核慢(注意，我们通过矩阵乘法实现卷积)。ARK目前没有实现专门用于大型矩阵乘法的vCTA(单位运算符的输出的一边大于256个元素)，所以当模型包含大型矩阵乘法时，它通常比现有的内核慢。

我们注意到，当模型包含许多并行运算符(如GoogLeNet或SSD)时，ARK的增益特别大。这是因为我们的高级调度器通过为每个并行操作符选择最佳的vCTA(或单元操作符)来最大化SM的总体利用率。具体来说，当轻量级操作员单独在GPU中运行时，我们将其安排为使用细粒度的vCTA，以便它利用更多的并发SMs。相反，当GPU因其他协同运行的操作而过载时，我们需要使用粗粒度的vCTA来更有效地利用SMs。这是因为粗粒度的vCTA同时处理更多的输入数据，因此有更多的机会更好地利用SM中的并行性。正如在3.3节中所解释的，在ARK框架中寻找性能最佳的vCTA是很容易的，因为它可以在不运行所有候选对象的情况下，精确地估计不同vCTA的性能。我们注意到其他框架没有提供类似的优化。

## 5.5.张量平行推理

本节介绍了张量并行方法的延迟改进，这种方法被称为专家混合(MoE)，它可以有效地扩展Transformer[45]架构，该架构通常用于许多流行的NLP模型[6,8,11,37]。该方法被建议将NLP模型扩展到1万亿模型参数[11,22]，但由于我们没有足够的GPU来运行整个模型，因此我们使用两个GPU来评估模型的张量并行推理。在实际操作中，这将被复制到其他GPU上，以同时应用流水线并行(用于训练或推理)和数据并行(仅用于训练)。

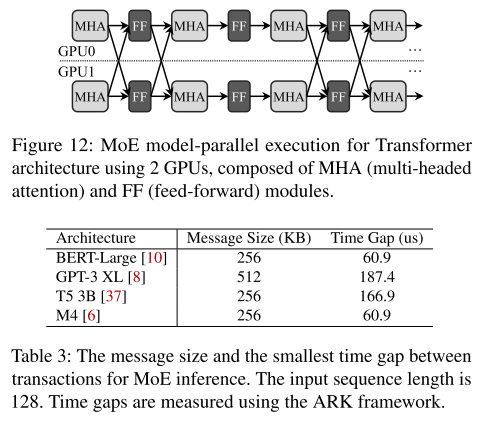


图12演示了MoE的执行。消息大小和交换之间的最小时间间隔取决于模型超参数，一些示例如表3所示。尽管我们在这里只展示了2-GPU的实验，但结果将与更大规模的实验相似，因为MoE被设计为只向选定的GPU发送一个小的常数(例如GShard[22]中的2)，而不是向所有其他GPU发送消息。

我们使用三种不同的比较基线- TF、TF- xla和C-Drv来评估ARK。注意，tensorrt加速的TensorFlow (TF-TRT)不支持模型并行性，所以这里不计算它。

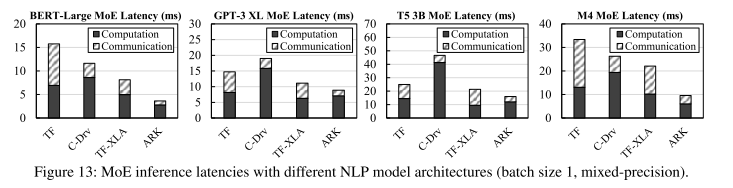


图13的结果显示，ARK的性能分别比TF和TF- xla好1.66x ~ 3.48x和1.25x ~ 2.31x。仅在通信延迟方面，ARK将其分别减少了3.68x ~ 5.65x和1.77x ~ 3.31x。总的来说，C-Drv在TF或TF- xla上实现了更好的通信延迟，但它的计算效率较低，因为它在ARK中重用GPU内核实现，但它没有受益于ARK调度程序的优化。我们还发现，与C-Drv的CPU驱动通信相比，ARK的GPU驱动通信具有显著的提速，如章节5.2所示。我们注意到，在gpt - 3xl和t53b中，ARK计算比TF-XLA慢。这是因为我们的矩阵乘法内核在这些情况下的性能比TF-XLA差，如第5.4节所述。

## 5.6.流水线并行训练

在本节中，我们将对GPT-3 [8] 6.7B模型进行训练，该模型是GPT-3通过流水线并行训练后最大的变化，能够适应8个V100 GPU的内存。该模型由32个顺序层组成，每个GPU按顺序训练4层:GPU 0读取输入数据，进行0 ~ 3层的前向传递，然后将16mb的输出传递给GPU 1，以此类推。当GPU 7完成前向传递后，继续进行31 ~ 28层的后向传递，16mb的反向传播梯度传递给GPU 6，以此类推。我们采用混合精度计算，将流水线阶段数设置为5个，每个阶段的批量大小设置为1个，序列长度设置为2048个。ARK在这个计算中使用了模拟的DMA堆栈。

本实验中，ofTF、TF- xla、Megatron-LM和ARK的训练吞吐量分别为0.35、0.47、1.69和2.38序列/ s，即ARK的训练吞吐量分别比TF、TF- xla和Megatron-LM高出6.80倍、5.06倍和1.40倍。在这种情况下，ARK的大部分改进来自GPU的计算效率，因为流水线并行训练通常会使大部分通信延迟与计算时间重叠。评估结果表明，ARK算法在提供算子融合增益的同时，支持DL算法的灵活并行性。

# 6.未来工作以及相关工作

我们希望在不久的将来，硬件的进步能够实现更高效的实现。例如，在SmartNIC上实现我们的软件DMA引擎将避免通过PCIe直接连接到GPU(例如，NVIDIA H100 CNX[9]结合了GPU和SmartNIC)的PCIe根复杂[44]的吞吐量问题，从而在SmartNIC上实现高效的MMIO。NVIDIA已经宣布了他们在smartnic上的GPU间通信硬件加速器(例如，在NVIDIA BlueField-3[29]上的all-to-all引擎)，这意味着我们的硬件引擎将来可能会实现类似的实现。此外，未来的主机CPU架构可能会解决根本的复杂问题，这将使我们的软件DMA引擎用CPU端MMIO取代cudaMemcpy，甚至更有效地，CPU上的DMA引擎(例如，Intel I/OAT[15]或AMD PTDMA[21])。

ACE[38]建议将整个集体通信逻辑转移到驻留在机器内部结构上的硬件加速器上，它不能扩展到外部网络(以太网、InfiniBand等)。我们的工作与ACE不同，因为它通常适用于任何(R)DMA网络，而且我们可以重用流行的集体通信库中的大多数现有软件逻辑。

GPUnet[18]为GPU线程提供了一个网络socket API集，并利用CPU干预来让GPU线程触发DMA。这是低效的，因为它们增加了大量的干预开销，特别是对于小消息，因为它们没有流水线处理多个DMA请求。它的吞吐量可能不是最优的，因为它在GPU上实现了通用的套接字接口，而ARK通过利用离线调度来减少运行时管理的元数据的开销。

灵活的[20]通过最小化内核的运行调度开销来加速DL执行，但它只在单个GPU上工作。由于该方法仍然依赖于cudaEvent和多流接口的CPU端控制，因此无法减少通信事件的处理开销。ARK通过让GPU线程完全控制所有的计算和通信任务来解决这个问题。

# 7.总结

本文设想了一个GPU驱动的代码执行系统，使GPU能够在DL应用程序的整个生命周期内实现自主控制。我们将GPU控制的DMA引擎置于GPU驱动系统的核心，它使GPU能够在没有任何外部控制的情况下相互通信。为了避免计算和通信之间的干扰，我们设计了DMA引擎和离线GPU调度，在不牺牲GPU计算吞吐量的前提下，使用较少的GPU资源进行通信，从而获得较高的通信性能。虽然我们的软件引擎已经显示出了与商用硬件相比的优势，但我们还提出了一种硬件引擎的概念证明，它可以显示出更高的性能，这表明随着未来商用硬件(如CPU、GPU或SmartNIC)的进步，我们的系统性能将得到进一步改善。