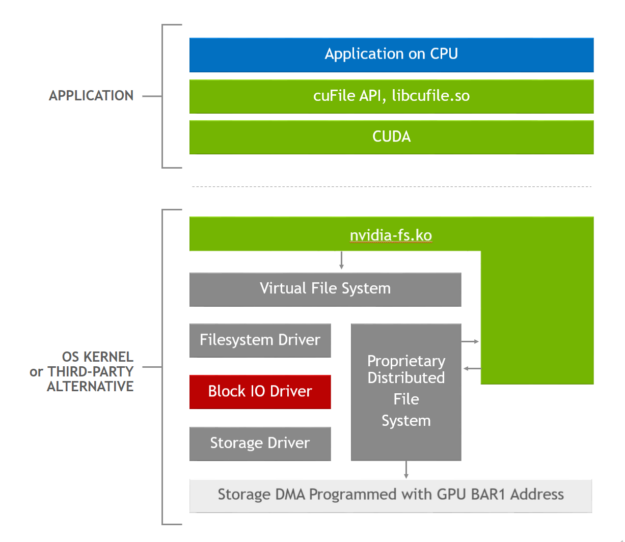
# Cuda GPU Direct Storage

以下是GDS实现的功能。首先，当前Linux实现的基本问题是通过虚拟文件系统（VFS）向下传递 GPU 缓冲区地址作为DMA目标，以便本地NVMe或网络适配器中的DMA引擎可以执行到 GPU 内存或从 GPU 内存的传输。这会导致出现错误情况。我们现在有办法解决这个问题：在 CPU 内存中传递一个缓冲区地址。

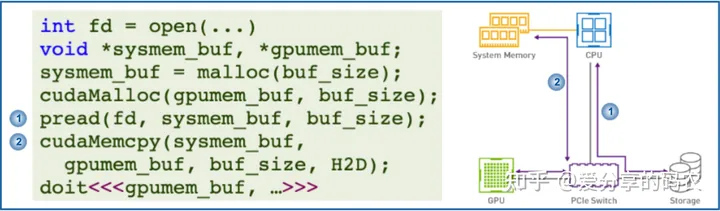
当使用cuFile API （如cuFileRead或cuFileWrite）时，libcufile。因此，用户级库捕获 GPU 缓冲区地址，并替换传递给 VFS 的代理 CPU 缓冲区地址。就在缓冲区地址用于 DMA 之前，启用 GDS 的驱动程序对nvidia-fs.ko的调用识别 CPU 缓冲区地址，并再次提供替代 GPU 缓冲区地址，以便 DMA 可以正确进行。

libcufile.so中的逻辑执行前面描述的各种优化，如动态路由、预固定缓冲区的使用和对齐。图 2 显示了用于此优化的堆栈。cuFile API 是 Magnum IO 灵活抽象体系结构原则的一个示例，它支持特定于平台的创新和优化，如选择性缓冲和 NVLink 的使用。

图 2 . GDS 软件堆栈，其中应用程序使用 cuFile API ，启用 GDS 的存储驱动程序调用 NVIDIA -fs . ko 内核驱动程序以获得正确的 DMA 地址。

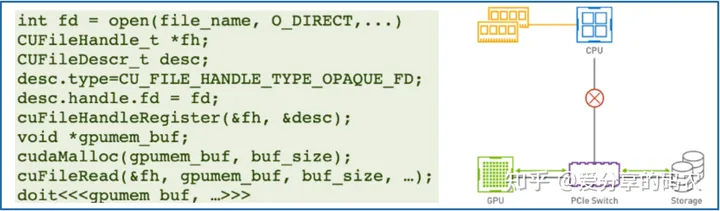
从IO读取链路来看，NVMe控制器通过DMA引擎将硬盘数据直接写入GPU显存，避免了主机内存和CPU的参与，从而实现CPU和主存的IO旁路，使IO吞吐能力不在受限于系统总线的带宽压力。

**而从编程模型的角度来看，代码结构主要进行了如下调整。**  
**(1) 主存拷贝链路**



首先通过pread函数对目标文件进行读取，将相关数据内容载入主存；然后在通过cudaMemcpy函数将目标数据从主存拷贝到显存。

(2) GPU Direct链路



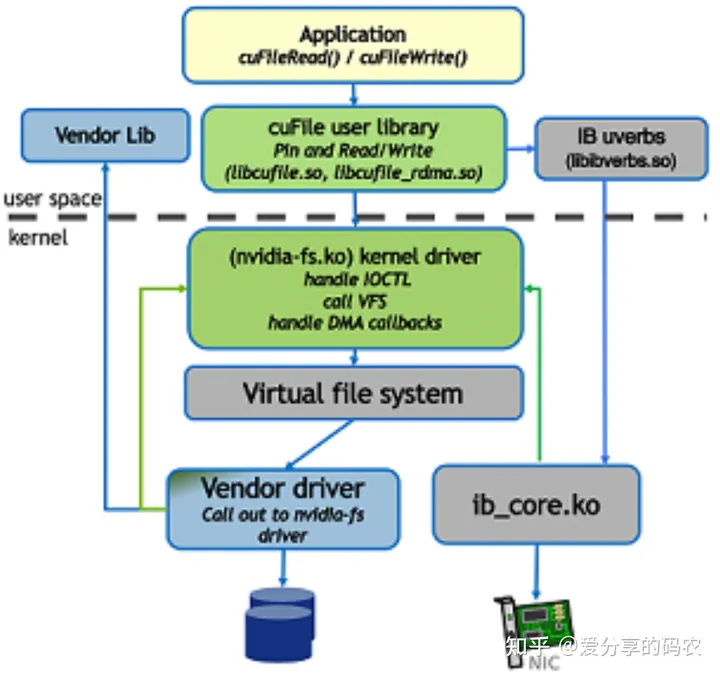
主要借助cuFileRead函数将目标文件数据直接读取到GPU显存，使其不在经由主存来做中间周转，从而降低一次内存拷贝操作，并缩减出一次PCIe Switch路由开销。

而整个链路的处理难点则在于如何能够在CPU一侧去有效识别cudaMalloc所返回的GPU虚拟地址，并能够将对应的物理空间固定住，以便NVMe控制器一侧能够通过DMA引擎来对目标数据进行传输。

由于cudaMalloc所返回的虚拟地址并不通过PageTable进行维护，所以无法通过查询PageTable来确定其对应的物理地址空间。对此，Nvidia新引入了一个内核模块(nvidia-fs)，通过它来维护每个GPU虚拟地址与对应物理地址的映射关系(借助nvidia内核驱动所提供的nvidia\_p2p\_page\_table\_t数据结构)，并对外暴露相应的接口函数，来供其他模块调用使用。

一、处理流程

以cuFileRead作为入口函数，整个IO链路的处理流程大致如图所示：



App to libcufile.so  
通过执行cuFileRead/cuFileWrite函数触发动态库的相关代码逻辑

libcufile.so to nvidia-fs.ko  
通过触发IOCTL系统调用使请求进入内核，以便内核一侧通过nvidia-fs模块来接管接下来的请求处理

nvidia-fs.ko to VFS  
对VFS的相关函数进行调用，以便将IO请求转发路由给具体的FS Impl实现(即图片中的Vendor driver)，比如beegfs的内核客户端实现

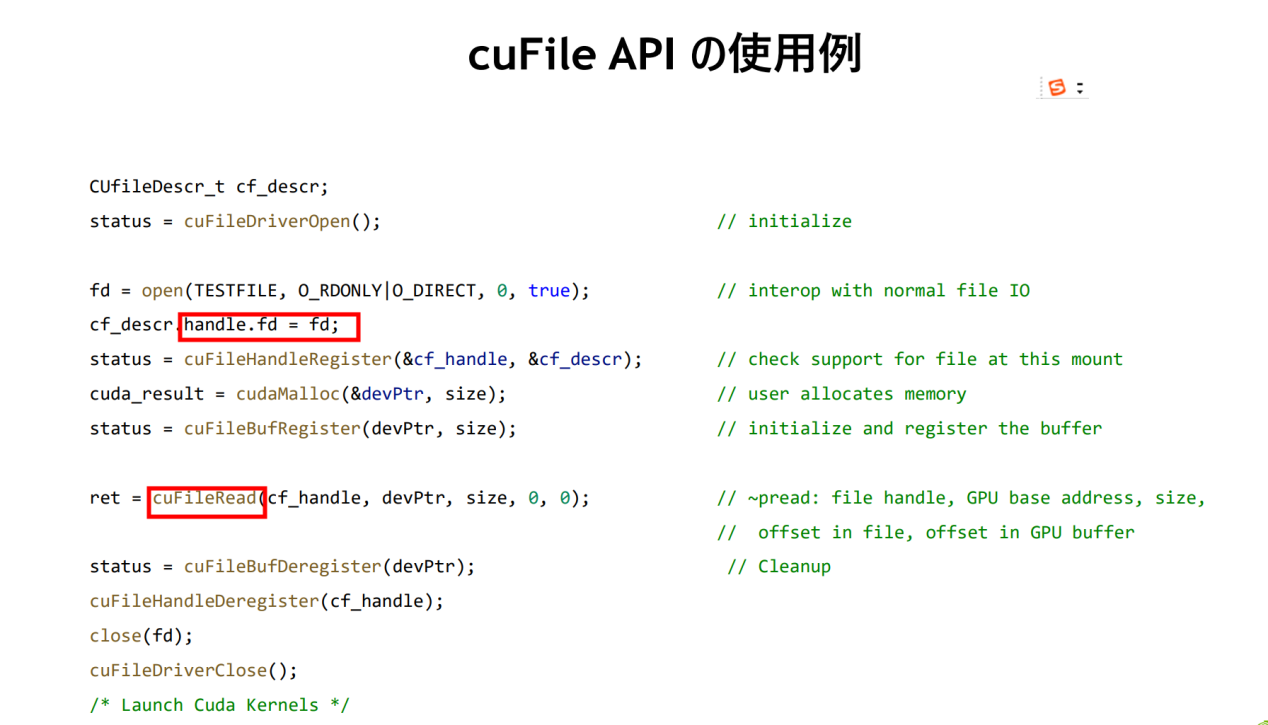
Vendor driver to nvidia-fs.ko  
通过访问nvidia-fs.ko来获取目标IO虚拟地址对应的GPU物理地址，并将对应的物理空间固定住(通过NVIDIA Kernel driver提供的nvidia\_p2p\_get\_pages函数)，以便后续进行DMA数据传输。对此，nvidia-fs.ko对外声明了nvfs\_dma\_map\_sg\_attrs函数来供第三方程序调用使用。

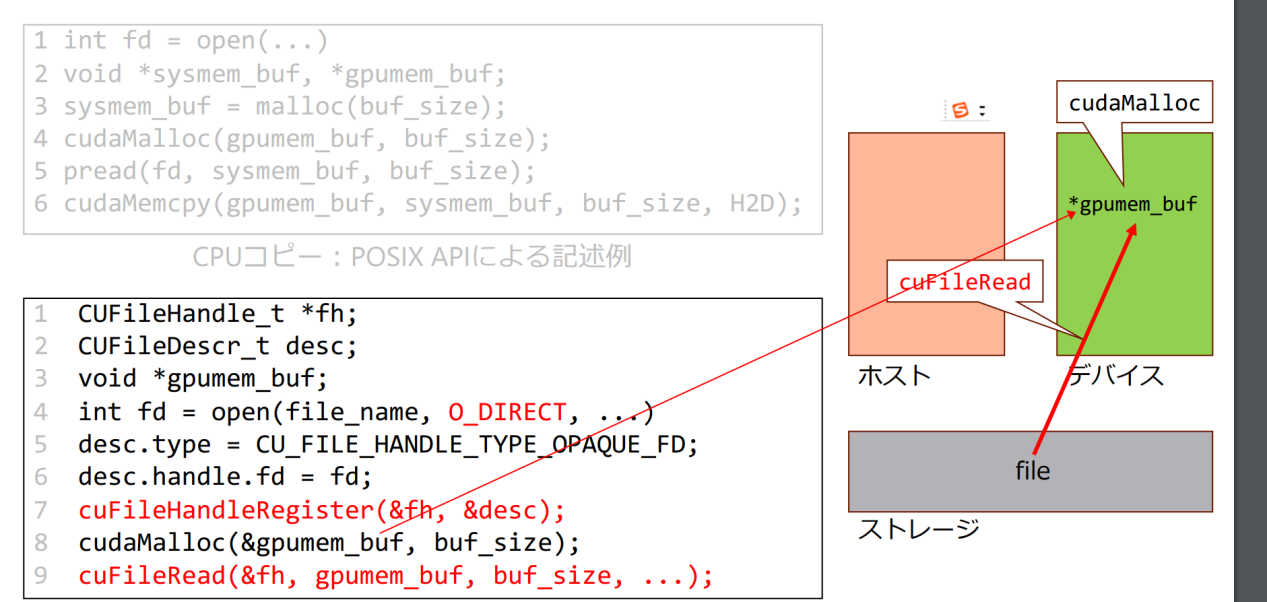
Vendor driver to DMA/RDMA engines  
通知DMA引擎将目标数据直接写入GPU显存，以此来实现CPU和主存旁路

二、注意事项

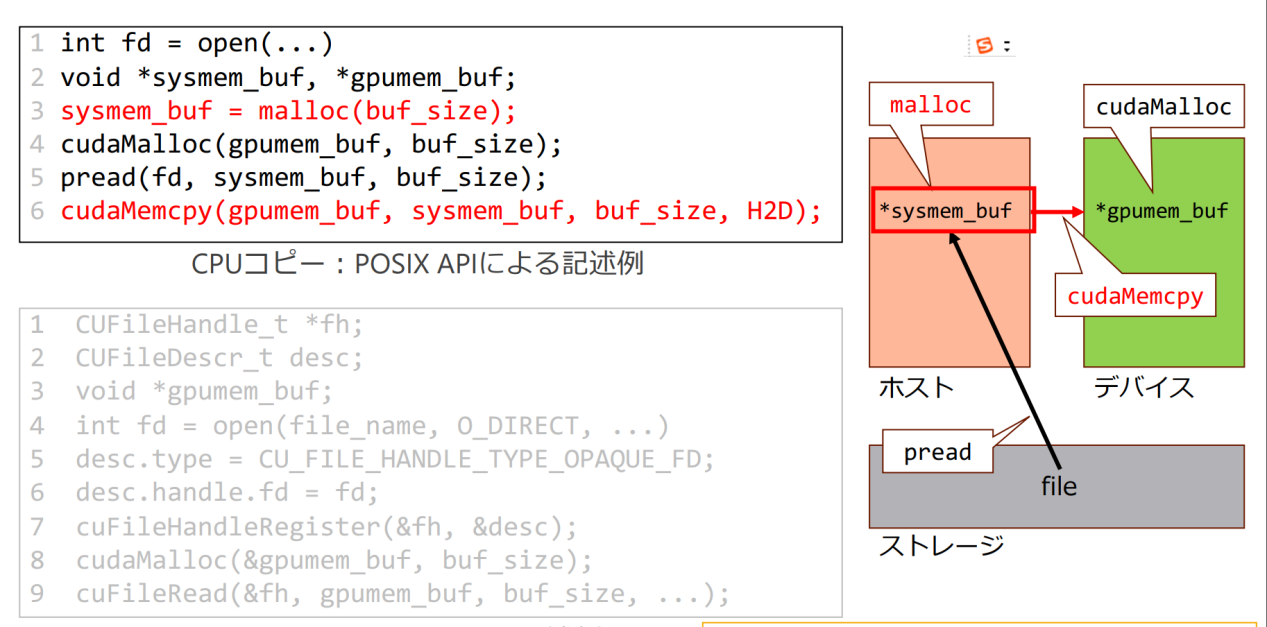
针对PCIe终端节点之间的P2P数据传输需求，内核已有一些解决方案(POC、DMA-BUF、HMM)来应对相关的场景，对此，Nvidia也在做相应的upstream处理，现有的基于nvidia-fs的处理方案很可能只是一个过渡阶段。

## Cu API



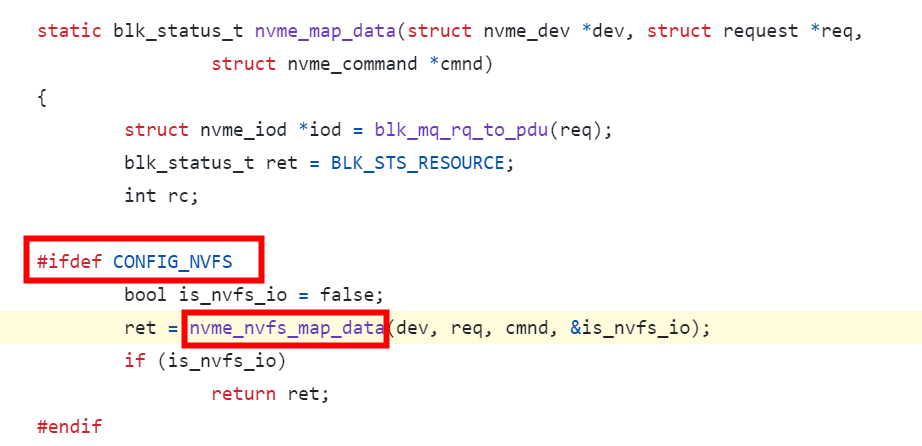


### Cuda mem cpy

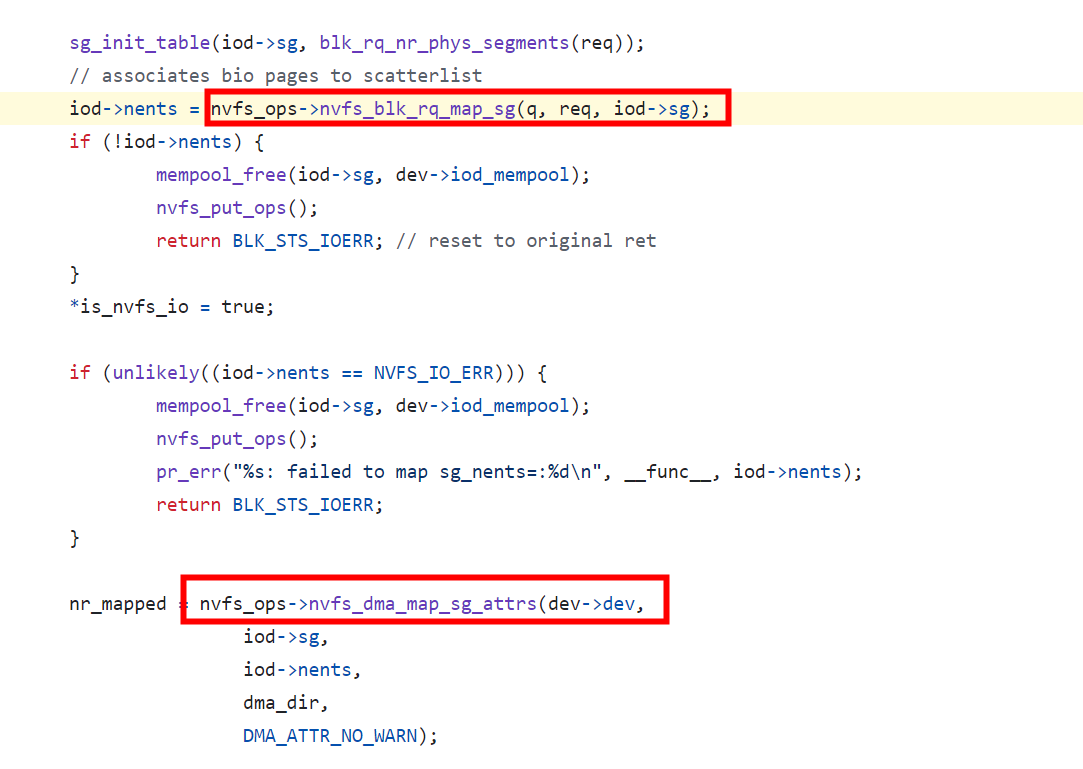


### nvfs\_dma\_map\_sg\_attrs

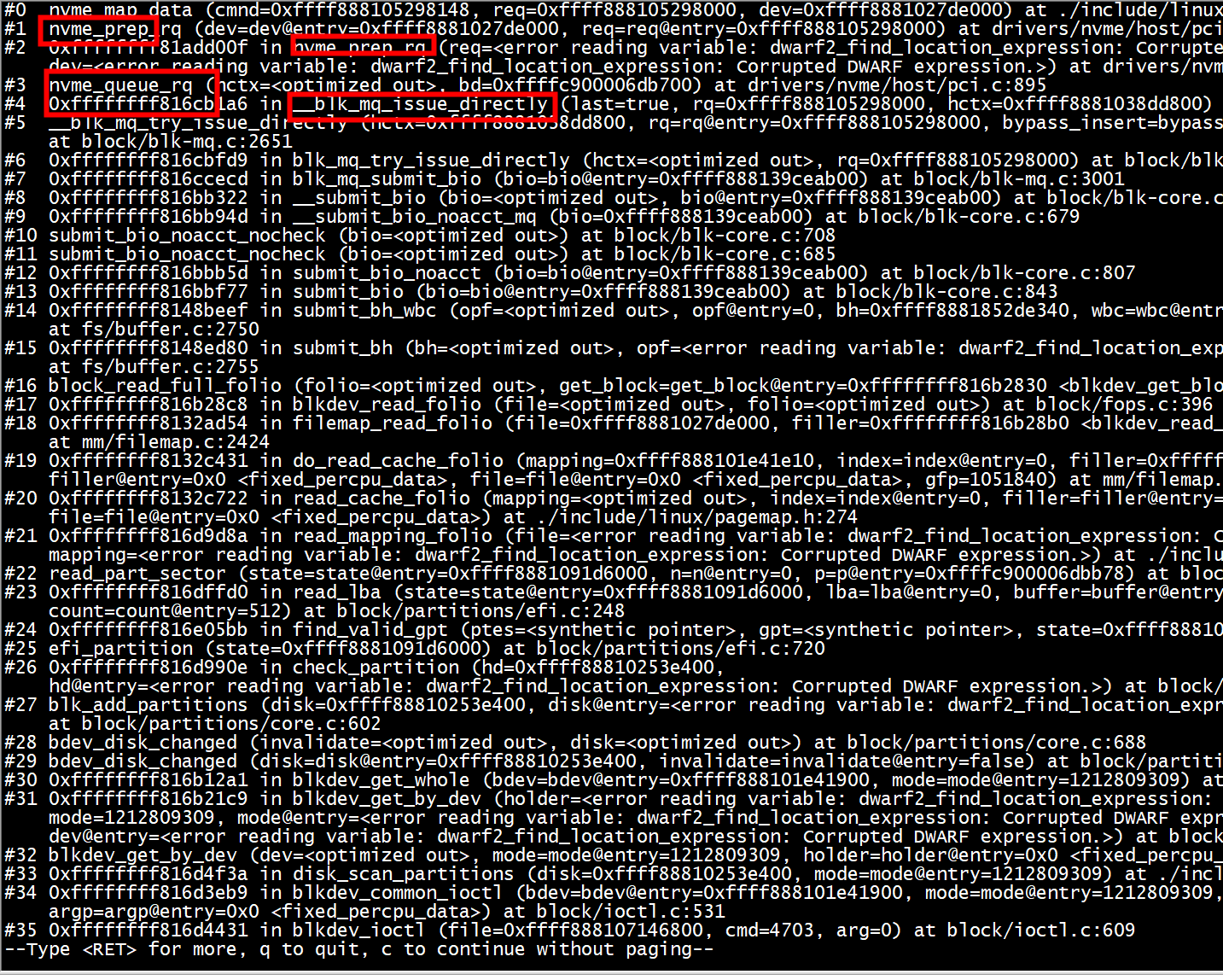
参考 mlnx-ofa\_kernel/drivers/nvme/host/nvfs.h



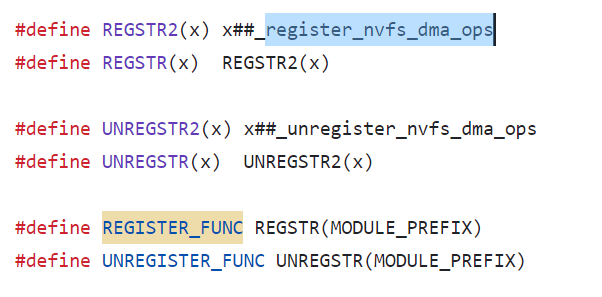
**nvme\_nvfs\_map\_data(**



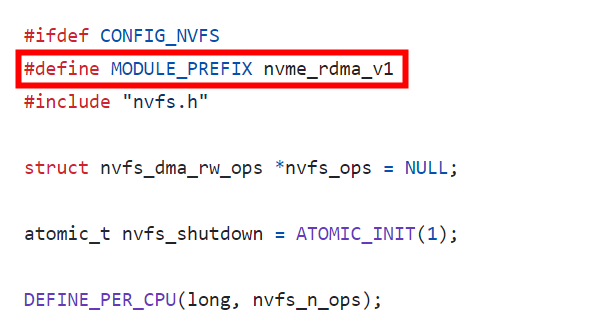
#### nvme\_map\_data

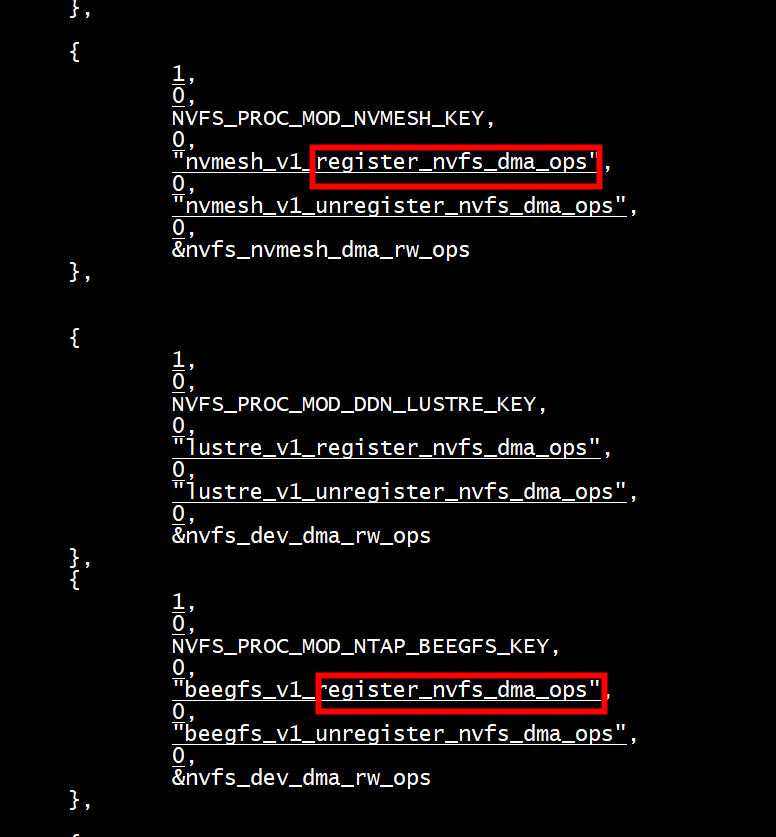


#### 注册

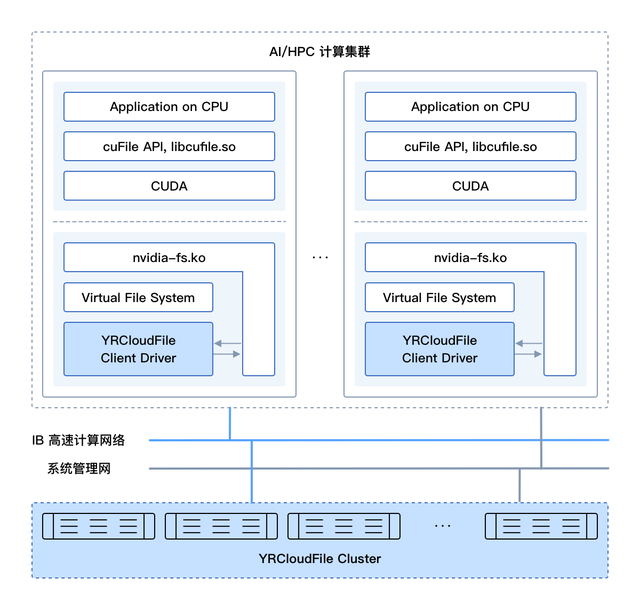


#### 注册 nvfs-rdma





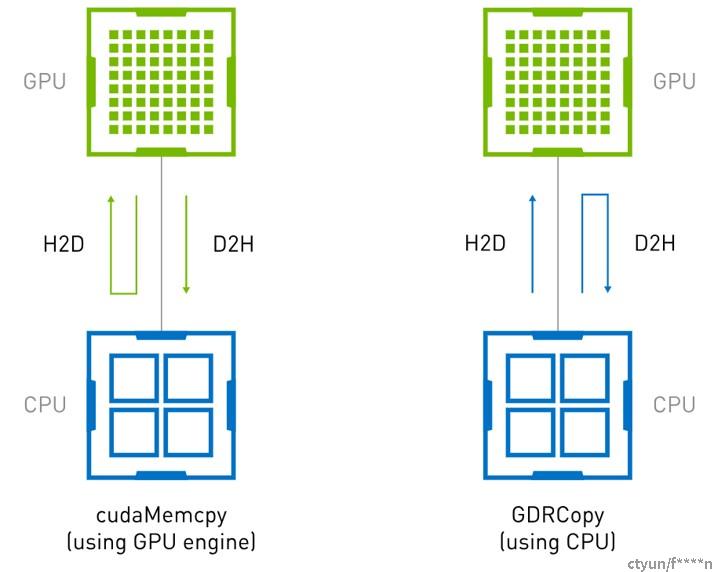
## YRCloudFile



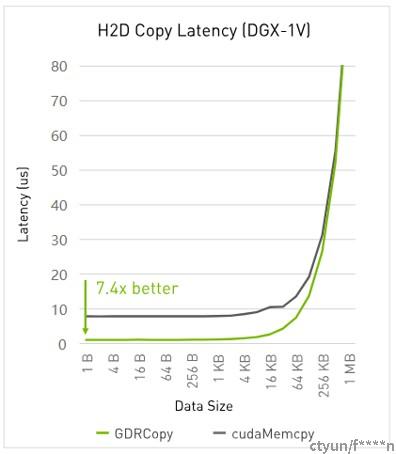
焱融分布式文件存储系统 YRCloudFile 支持 GDS 的大体流程：

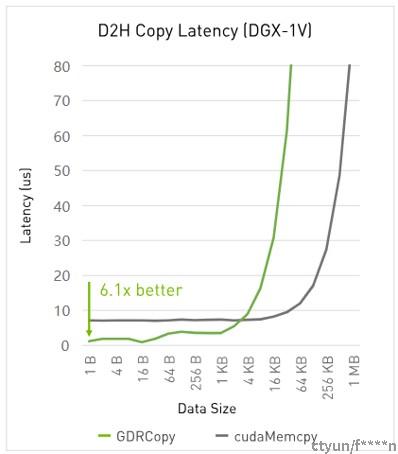
YRCloudFile 客户端向 nvidia-fs 注册后，cuFile 打开一个 YRCloudFile 集群文件，会将 nvidia-fs 和 YRCloudFile 的特定接口进行绑定，当 io 下发到 client 中，client 检测该 IO 是否是 GDS 的请求，如果是，则回调 nvidia-fs 的 map 接口，获得 sglist 请求的 dma 地址，借助底层驱动能力，实现数据的RMDA 传输。

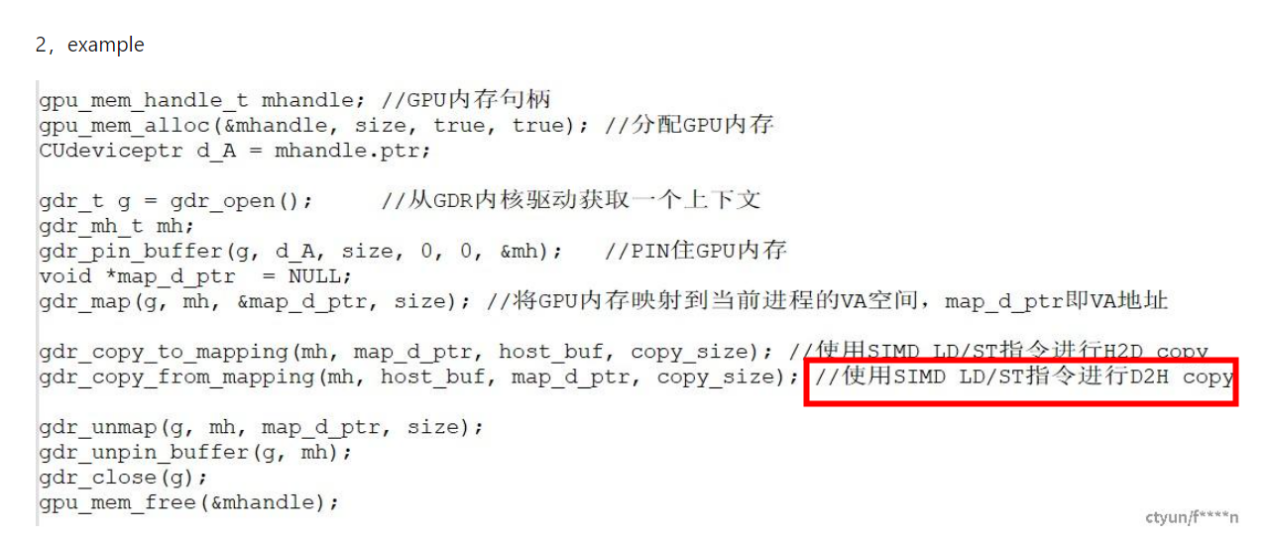
## GDRCopy介绍



GDRCopy是一个基于GPU direct RDMA技术的低时延GPU内存copy的库。如上图所示的H2D和D2H的内存拷贝，传统上采用cudaMemcpy，它实际是由GPU触发DMA引擎在CPU和GPU之间搬移内存。因为需要额外的操作DMA引擎的指令，它在小数据搬移时效率并不高。GDRCopy则允许CPU采用PCIE BAR映射的方式直接访问GPU内存，因为是直接的LD/ST指令(SIMD指令)，所以对于小数据来说效率更高，时延更低。如下图所示，H2D拷贝16KB以下的数据，cudaMemcpy需要7us，而GDRCopy只需要1us。







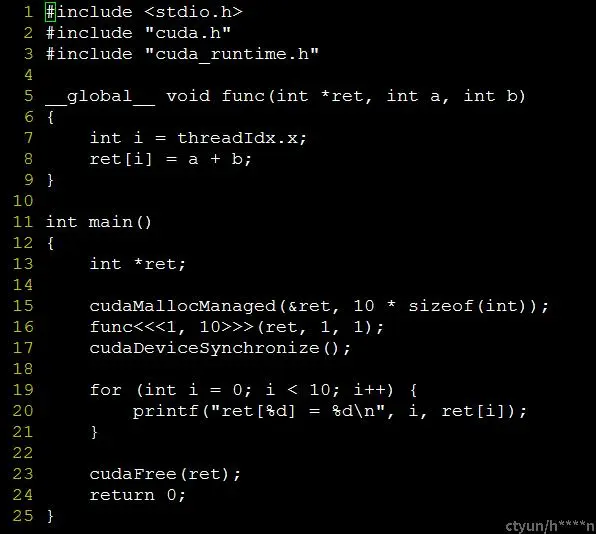
# CUDA中的UM机制与GDR实现

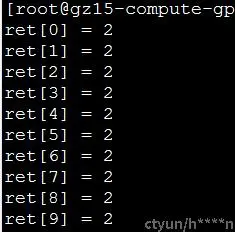
CUDA 6.0开始支持Unified Memory，在CUDA 6.0之前，由于CPU和GPU之间的地址空间是各自独立的，需要进行多次的手动分配和频繁地使用cudaMemcpy在CPU和GPU的memory之间来回拷贝内存，当实际数据结构更加复杂时，内存管理也会变得很复杂。采用UM后，可通过一个统一的指针进行内存管理，由系统自动的迁移内存，极大地减少了代码量，在很大程度上减少了程序员的工作量

使用UM机制，不能再用cudaMalloc分配GPU memory上的内存，应该使用cudaMallocManaged

cudaMallocManaged ( void\*\* devPtr, size\_t size, unsigned int flags = cudaMemAttachGlobal )

## 一、UM的例子





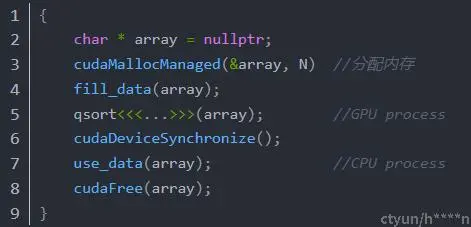
如上图程序示例图和运行结果图所示：在CPU端可以直接访问GPU内存中的ret[i]的值

直接使用cudaMallocManaged分配GPU内存，指针ret指向GPU的内存，在CPU端同样可以直接访问ret指针。

需要注意的是，UM不会消除CPU Memory和GPU Memory之间的拷贝，这部分copy工作交给CUDA执行，程序员不感知CPU Memory和GPU memory之间的数据拷贝，但copy依然是存在的。

## 二、Unified Memory page\_migration机制

GPU和CPU一样，有自己的页表和TLB，当CPU或GPU采用UM机制访问一块内存时，均可能发生page falut，从而导致memory在CPU和GPU之间来回拷贝。



上图是一个使用UM机制的例子，我们来看一下具体流程：

1 首先使用cudaMallocManaged分配GPU内存 ；

2 在fill\_data中，CPU会写这块内存，此时CPU触发page fault流程，GPU memory中的内容通过PCIE migrate到GPU中，同时为了数据保证一致性，CUDA会invalidateGPU中对应的页表项 ，这样下一次GPU访问的时候就会触发GPU端的页表项；

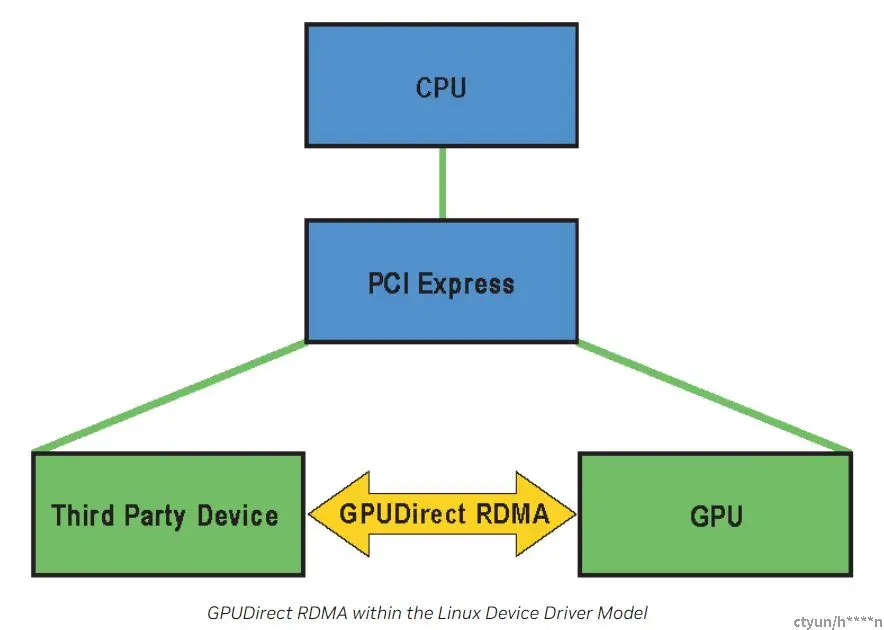
３ qsort<<<...>>>中，GPU端执行kernel函数，由于之前GPU的页表已经无效了，此时触发GPU端的page falut，数据通过PCIE从CPU migrate至GPU中。然后CPU短的页表项被invalidate；

4 use\_data中，CPU处理数据，由于CPU的页表项已经在上一步被invalidate了，于是CPU端触发page falut, 数据再从CPUmigrate至GPU。

由此我们可以看到，其实UM机制中，数据就是来回在CPU和GPU的memory中来回migrate，只是程序员在上层并不感知。

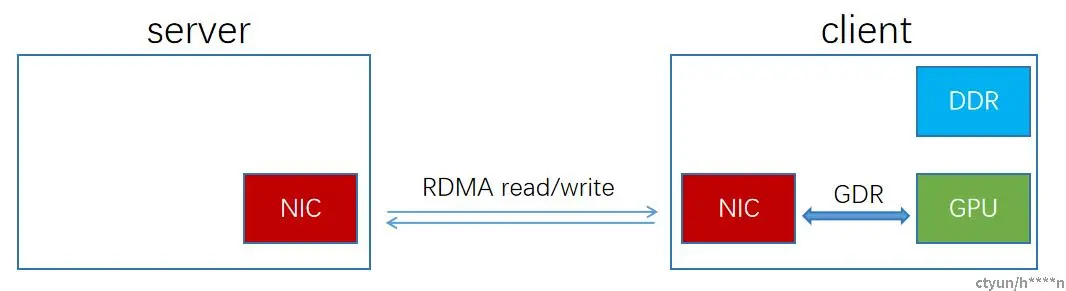
## 三、GDR(GPU Direct RDMA)实现

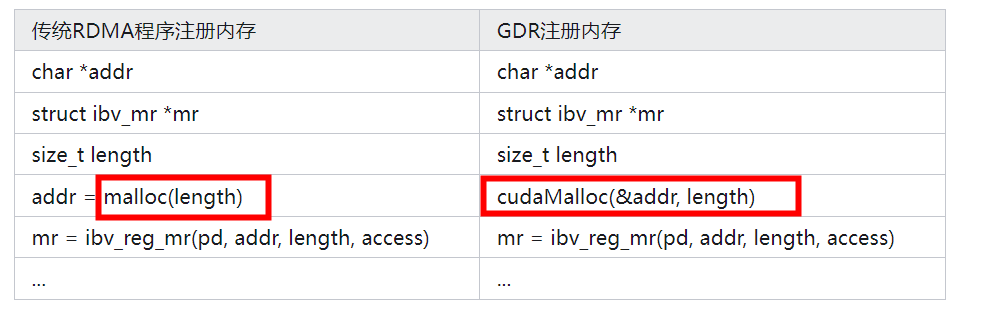
GDR，即GPU direct RDMA，可以在GPU中直接分配memory，然后在GPU和device上进行数据传输。 Device可以是NIC，storage adapters，video acquisition devices这些PCIE设备。



想要使用GDR功能，device和GPU需要在同一个RC下。

（一）GDR Example





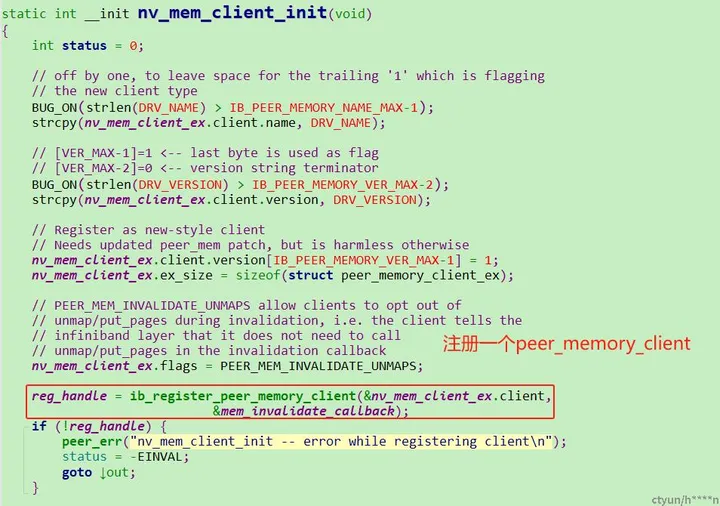
从上面的程序例子可以看到，其实GDR和传统的RDMA程序几乎没有什么不同，最主要的不同就在于，传统的RDMA程序采用malloc分配CPU memory上面的内存，而GDR程序中采用cudaMalloc分配位于GPU memory上面的内存。

（二）GDR的使用

使能GDR功能需要插入nv\_peer\_mem内核模块： insmod nv\_peer\_mem 插入nv\_peer\_mem内核模块后，调用ibv\_reg\_mr后将会在GPU memory中注册内存而不是在CPU DDR中注册memory。

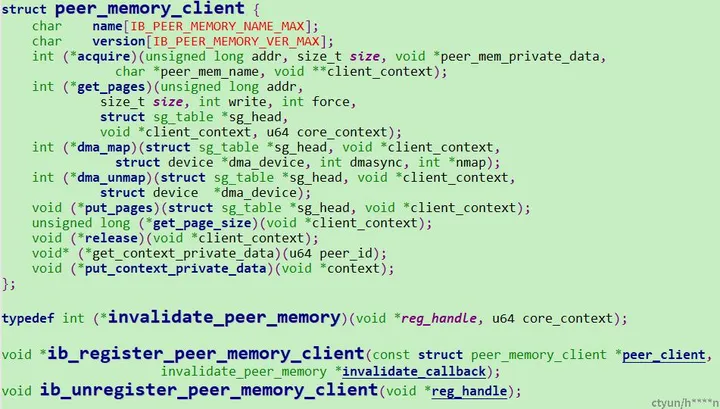
（三）GDR的实现

IB提供了一种叫做peer\_memory\_client的机制，第三方设备在自己的\_init函数中调用ib\_register\_peer\_memory\_client在ib\_core中注册了一个peer\_memory\_client。



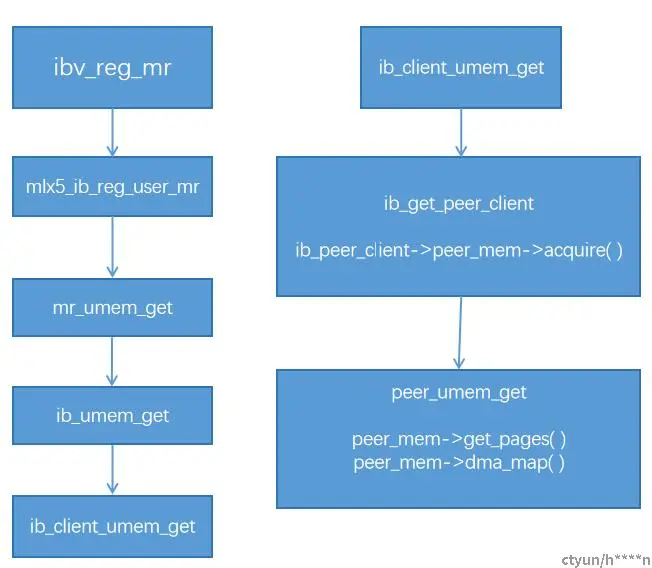
insmod nv\_peer\_mem.ko时会执行nv\_mem\_client\_init函数，其中就调用了ib\_register\_peer\_memory\_client向ib\_core注册了一个peer\_memory\_client 。

peer\_memory.h中定义了各种接口，使得注册为peer\_memory\_client的第三方device能够实现自己的get\_pages，map等相关的接口。peer\_memory\_client的实现如下：

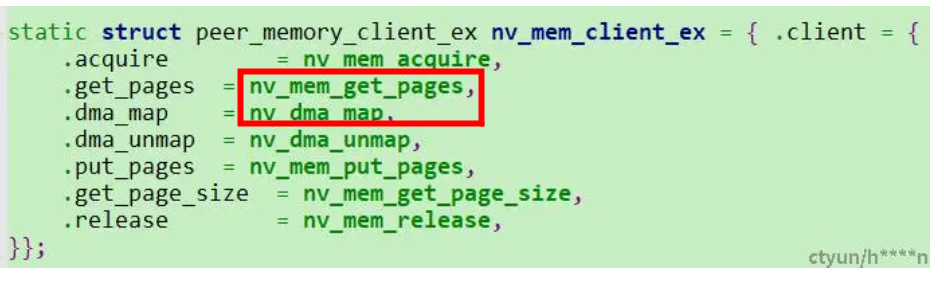


peer\_memory\_client在peer\_mem.h定义，位于mlnx-ofed-kernel-4.7\include\rdma

每个第三方设备都可以实现自己的peer\_memory\_client，然后调用ib\_register\_peer\_memory\_client往ib\_core里面注册为一个peer\_memory\_client。 每个peer\_memory\_client都是链表peer\_memory\_list中的一个元素。 这样任何实现了peer\_memory\_client接口的设备都可以实现Direct RDMA的功能。

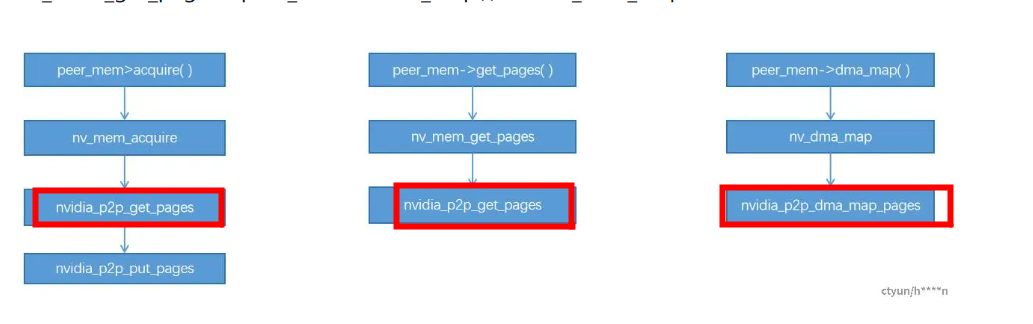


ibv\_reg\_mr的流程如上，最终会调用到ib\_client\_umem\_get函数，其中就会调用到peer\_memory\_client自己注册的acquire，get\_pages，dma\_map等函数。



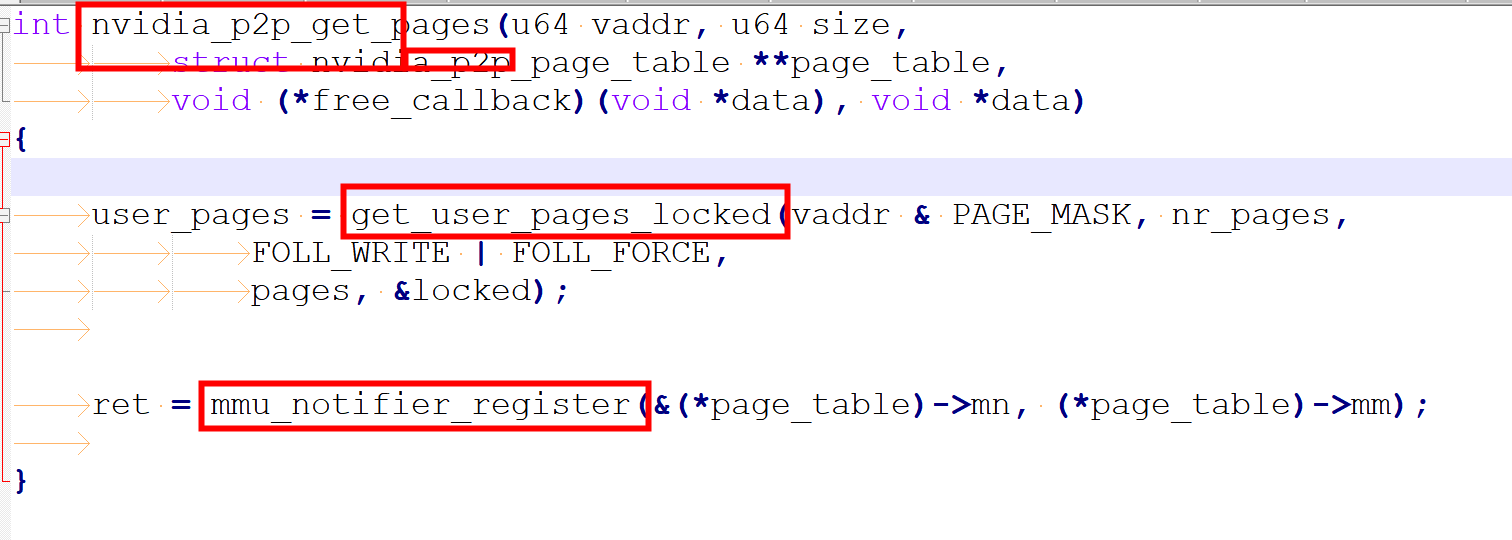
如上图所示，nv\_mem\_client\_ex中注册了自己的acuqire，get\_pages，dma\_map等函数。当insmod nv\_peer\_mem.ko时，最终执行的

peer\_mem->acquire( )就是nv\_mem\_acquire，peer\_mem->get\_pages( )就是nv\_mem\_get\_pages，peer\_mem->dma\_map( )就是 nv\_dma\_map。



* acquire函数返回1，表示ib\_core找到了对应的peer\_memory\_client，之后就由这个peer\_memory\_client来handle这块注册的内存。其中nv\_peer\_mem中的acquire最终会调用到nvidia\_p2p\_get\_pages通过虚拟页面获得物理页面，如果成功，则表明该peer\_memory\_client能够处理这块物理内存，最后调用nvidia\_p2p\_put\_pages解除VA到PA的映射。acquire主要功能就是表示这块注册的peer\_memory内存能够由该peer\_memory\_client来handle（成功返回1，失败则返回0）。
* nv\_mem\_get\_pages中调用了nvidia\_p2p\_get\_pages通过传入虚拟地址（nv\_mem\_context->page\_virt\_start），获得物理地址，并将结果保存在nv\_mem\_context->page\_table。
* nv\_dma\_map中调用nvidia\_p2p\_dma\_map\_pages获取dma\_address，page\_size等，并填入sg\_table中(用于scatter gather DMA)

#### nvidia\_p2p\_get\_pages



# Spdk GPU Direct Storage

本文来自韩国首尔国立大学以及三星电子的Jonghyun Bae等人。突出强调了深度学习训练中的内存墙问题，设计了一种将训练过程的中间结果下移到NVMe-SSD中的方法，采用这样的方法能够提高BatchSize的大小，提高训练的吞吐率，同时还减少GPU和CPU之间的打扰。

整体介绍

在深度学习训练中，由于GPU显存受限，因此通常不能够加大BatchSize的大小。但是小BatchSize通常不足以充分发挥GPU的性能，从而会造成训练吞吐率的降低。

过去的工作采用的方法通常是将GPU训练的中间结果（比如FeatureMap）下移到HostMemory之中，采用该方法能够实现大的吞吐，但是因为CPU和GPU之间来回传输数据的开销，将会造成吞吐的下降，同时还会存在使用HostMemory的计算任务的互相干扰。

本文将训练的中间结果Offloading到NVMe-SSD上，从而实现了对于大BatchSize的支持，但是由于NVMe-SSD的带宽受限，需要采用精细化调度来优化数据传输，同时使用GPU-direct的策略，直接实现GPU和SSD的数据传输，最大程度地降低对于CPU的打扰。

方法概述

内存管理 FlashNeuron通过cudaMemoryAllocate分配得到了一个比较大的连续空间，然后自己在上面完成内存管理。首先是Tensor的释放和分配，为了防止碎片化，本文将常驻内存的tensor分配在低端，将需要来回迁移的tensor放到了高端。其次是采用了压缩技术来减少和SSD交互所需要的通信量，具体来说采用的压缩技术有CSR压缩和半精度训练。

Offloading策略 Offloading的基本思路是，首先有一个profiling的过程，在该过程中，可以通过建模确定每一个下降的tensor的大小和压缩率。然后在第一阶段，首先逐个下放tensor，直到内存空间足够训练为止，在这个过程中测量下放tensor的过程所需要的时间和整个前向训练所用的时间，比较这样的两个时间，如果发现下放的用时太长了，就尝试从最后一个下放的tensor开始做松弛，然后不断地添加其他的压缩率更高的tensor直到下放tensor的用时要小于训练的用时，这样就可以将下放tensor所用的时间隐藏了。

直接SSD访问 本文的第三个创新点是使用了一个直接SSD访问的策略，不需要对于CPU产生打扰，主要就是使用了两个硬件提供的接口，一个是GDRCopy策略（可以实现GPU内存的直接访问），另一个是Intel SPDK能够实现用户态对于block-level的直接访问。写由七个步骤完成：首先发出Request，接下来分配LBA并记录，然后下发请求到队列中，队列将对应请求下发，然后使用GDRCopy+SPDK将数据连续写入到NVMe SSD上，在写入完成之后更新元数据表。在下一次写的时候，检查元数据表，确认已经更新完成。

## Peer-to-Peer Direct Storage Access

P2P-DSA 允许 GPU 和 NVMe SSDs 之间进行直接内存访问，而无需使用主机 DRAM 缓冲区，从而在 SSD 读/写过程中最大限度地减少主机的介入。P2P-DSA 是一个轻量级的层，利用 GDRCopy 和 SPDK 这两项技术，实现张量从 GPU 到 NVMe SSDs 的通信。

* GDRCopy 是一个基于 NVIDIA GPUDirect 的快速 GPU 内存复制库，它使 GPU 内存可以被其他 PCIe 设备直接访问。
* Intel 的 SPDK 则将块级 I/O 接口直接暴露给用户空间软件。

P2P-DSA 有一个元数据表，用于维护卸载到 SSD 的每个张量的元数据。元数据表包含一个长的逻辑块地址（LBA）值和一个用于检查 I/O 完成的布尔值。

具体的传输请求（从 GPU 到 SSD 的卸载）操作如下：

当调用 P2PDSA\_issue 时，P2P-DSA 从传输请求中获取索引、缓冲区和方向（写）信息。

调用逻辑块地址（LBA）分配器，为单个 SSD 设备或多个 SSD 设备（当使用多个 SSD 以提高卸载/预取带宽时）分配一组连续的块。将从 LBA 分配器分配的第一个块的 LBA 更新到元数据表的适当位置。

P2P-DSA 为每个逻辑块创建一个命令，然后将其排入命令队列。这里，NVMe 命令包含：

i) 源地址（GPU 内存地址由 GPUDirect 转换为 PCIe 总线地址），和

ii) 设备地址（使用元数据表中的 LBA 计算）。

当调用 P2PDSA\_update 时：

从软件命令队列中获取排队的命令，并将其发给 NVMe SSD，只要 NVMe 设备的提交队列有空间。

NVMe SSD 设备将执行这些请求，并在 SSD 设备和 GPU 之间进行直接数据传输。一段时间后，这些传输请求将完成，并在 NVMe 设备完成队列中更新其状态。

当再次调用 P2PDSA\_update 时，P2PDSA\_update 将清空完成队列，并通过设置相应的完成位来更新元数据表。

此时，如果应用程序调用 P2PDSA\_is\_done 来查询已经卸载的张量，它将返回 true。

以下是这个过程的七个主要步骤：

1. 发起传输请求：当 P2PDSA\_issue 被调用时，P2P-DSA （Peer-to-Peer Direct Storage Access）会从传输请求中获取索引、缓冲区和方向（写入）信息。
2. 分配逻辑块地址：调用逻辑块地址（LBA）分配器在单个或多个 SSD 设备上分配一组连续的块（当使用多个 SSD 时，可以提高卸载/预取的带宽）。然后，将从 LBA 分配器分配的第一个块的 LBA 更新到元数据表的相应位置。
3. 创建并推入命令：P2P-DSA 为每个逻辑块创建一个命令，然后将其加入命令队列。此处，一个 NVMe 命令包含：i) 源地址（GPU 内存地址由 GPUDirect 转换为 PCIe 总线地址）；ii) 设备地址（使用元数据表中的 LBA 计算）。
4. 发出命令：当调用 P2PDSA\_update 时，只要 NVMe 设备提交队列有空间，就从软件命令队列中获取排队的命令并发送到 NVMe SSD。
5. 执行请求并传输数据：NVMe SSD 设备将执行这些请求，并在 SSD 设备和 GPU 之间直接进行数据传输。
6. 清理完成队列并更新元数据表：稍后，这些传输请求将完成，其状态将在 NVMe 设备完成队列中更新。当再次调用 P2PDSA\_update 时，它将清理完成队列并通过设置相应的完成位来更新元数据表。
7. 检查已卸载的张量是否完成：如果此时应用程序调用 P2PDSA\_is\_done 来询问已卸载的张量是否完成，它将返回 true。
8. 反向路径（即从 SSD 预取数据到 GPU 内存）的执行过程类似，只是：

i) LBAs 是从元数据表中读取的，而不是被分配的；

ii) 发出的是读命令，而不是写命令。

在卸载和预取的过程中，大部分数据访问都是顺序访问，这比随机访问在吞吐量和耐用性上更有优势。

**两个关键函数**

P2PDSA\_issue 和 P2PDSA\_update 是在论文中提到的两个函数，这两个函数是 Peer-to-Peer Direct Storage Access (P2P-DSA) 技术中的关键组成部分。这两个函数的作用是处理和管理 GPU 和 NVMe SSDs 之间的直接内存访问。

P2PDSA\_issue：这个函数的作用是处理从 GPU 到 SSD 的数据传输请求。当这个函数被调用时，它会从传输请求中获取索引、缓冲区和方向（写）信息。然后，它会调用逻辑块地址（LBA）分配器，为单个 SSD 设备或多个 SSD 设备分配一组连续的块。最后，P2PDSA\_issue 会为每个逻辑块创建一个命令，然后将其排入命令队列。

P2PDSA\_update：这个函数的作用是管理 NVMe SSD 设备的完成队列。当这个函数被调用时，它会从软件命令队列中获取排队的命令，并将其发给 NVMe SSD 设备。然后，NVMe SSD 设备会执行这些请求，并在 SSD 设备和 GPU 之间进行直接数据传输。一段时间后，这些传输请求将完成，并在 NVMe 设备完成队列中更新其状态。在再次调用 P2PDSA\_update 时，它会清空完成队列，并通过设置相应的完成位来更新元数据表。

通过这两个函数，P2P-DSA 技术能够有效地管理 GPU 和 NVMe SSDs 之间的直接内存访问，从而提高数据处理的效率。

**小故事**

这个地方偏系统的实现，我们通过一个小故事来看一下他在干什么。

你可以把 GPU 想象成一个大型仓库，里面储存了大量的包裹（也就是数据）。现在，你需要将这些包裹送到 SSD，也就是你的目的地。你可以把SSD 想象成一个远在城市另一头的大型收货点。P2PDSA\_issue 就像是负责分配包裹和计划路线的调度员。

当你调用 P2PDSA\_issue 时，调度员开始工作。他会先检查包裹的详细信息，然后为每个包裹分配一个独特的追踪号（也就是逻辑块地址，LBA）。然后，他会根据追踪号和包裹的目的地，为每个包裹创建一份详细的送货单（也就是命令），然后将这些送货单排入命令队列。

接着，你调用 P2PDSA\_update，这就像是一辆大型货车，负责将包裹从仓库运送到收货点。每当货车有空位时，它就会从命令队列中取出一个送货单，然后从仓库中取出相应的包裹，直接送到收货点。在送货的过程中，货车会直接从仓库出发，直接到达收货点，不需要途径任何其他地方。

当包裹成功送达收货点后，货车会在完成队列中更新包裹的状态。然后，当你再次调用 P2PDSA\_update 时，货车会清空完成队列，并通过设置相应的完成位来更新包裹的状态。

这就是整个过程的一个比喻。通过这个比喻，你可以看到，P2PDSA\_issue 和 P2PDSA\_update 的作用就像是一个高效的物流系统，它们能够有效地管理包裹（也就是数据）的传输过程，从而提高整个系统的效率。

## 参考

[FlashNeuron: SSD-Enabled Large-Batch Training of Very Deep Neural Networks](https://www.usenix.org/conference/fast21/presentation/bae)

# PCIe P2P

[PCIe P2P](https://zhuanlan.zhihu.com/p/664434389)

[pci p2p DMA的条件](https://www.cnblogs.com/10087622blog/p/17745805.html)

P2P简称Peer-to-Peer，即点对点， PCIe P2P， 即一个EP设备直接访问另外一个EP设备。

PCIe p2p功能在SPEC中没有明确的capability定义，实质上是由PCIe路由规则而决定；例如地址路由，通过bridge设备的mem window来判断如何路由mem request，TLP的地址如果在bridge的mem window中，则TLP向bridge的下一级转发，否则向上转发。

# 内核配置

需要开启 CONFIG\_PCI\_P2PDMA 才能使用p2p的DMA

# 参考项目

1. [ssd-gpu-dma](https://github.com/enfiskutensykkel/ssd-gpu-dma)
2. [PCIe P2P test](https://zhuanlan.zhihu.com/p/386153789)