# CPU与GPU之间是如何通信的

<https://www.zhihu.com/question/22233341/answer/3422629961>

# Pcie P2p

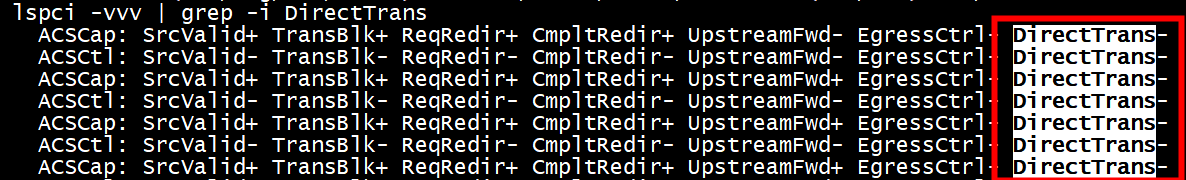
## ACS

ACS(Access Control Services)包括但不限于关注p2p，ACS包含p2p相关的配置并管控其路由。ACS协议通过设置相关的Control Bit能够决定一个TLP的生命周期，比如可以被正常路由,或者被阻塞，或者被重定向。

ACS可以被应用于RCs，Switches，Multi-Function Device和使能了sriov的pf，记住，当前就这四类。

通过在如上四类设备上上开启ACS服务，可以禁止P2P发送，强迫PCIe Switch将所有地址的访问请求送到RC，从而避开peer-to-peer访问的风险。

### DirectTrans



### acs和虚拟化

* 地址空间

在没有虚拟化的时候，系统软件的视角中，PCIe 总线地址空间就是host CPU的物理地址空间的一个子集，在BIOS给OS上报的ACPI表中，PCIe总线相关的地址空间都是CPU的物理地址空间的相关子地址空间。因此，在物理世界中，PCIe总线地址空间和CPU本地的其他内存空间（DDR）都是独立的，保证不会overlap。PCIe总线上的设备发起的各种DMA访问都可以准确无误地路由到最终真正的目的地。然而，这一切在虚拟化的时候变了。

* 功能

如图，



1.虚拟化环境下，虚拟机VM1和VM2分别绑定了设备A的VF1和设备B的VF1，在虚拟机上的驱动发起了DMA访问，分别是设备A发出了地址A的访问，设备B发出了地址B的访问。而地址A和B因为是虚拟机上的用户态/内核态直接编程，因此它们只能是GVA/GPA，并且属于是虚拟机上的OS管理的地址空间，和物理机上的OS管理的地址空间是独立的，且为了保证性能，现代虚拟化环境都会做到两者在主业务（DMA）工作的时候没有任何通信。也就是，VM上的OS认为自己管理了唯一的且是全的“物理地址空间”。

2.当设备A和B的DMA访问操作到达SW后，设备A发出的DMA访问操作的地址A命中了SW对接设备B的那个DP的prfetchable memory窗口中，同样地，设备B发出的DMA访问操作的地址B命中了SW对接设备A的那个DP的prfetchable memory窗口中，因此在SW内部直接P2P了。

3.实际上，业务编程的需要是此时VM1和VM2之间的数据流没有任何P2P的交互，VM1需要设备A将数据DMA到host CPU的内存A1中，而VM2需要设备B将数据DMA到hsot CPU的内存B1中，在Host CPU的地址空间中， A1∩B1=∅ 。

4.因为虚拟机VM1和VM2此时的dma\_allocate的地址空间都是独立的（不管是否虚拟与否），同时和物理机上的VMM管理的物理地址空间也是独立的，因此在虚拟机上申请到的DMA地址是有可能发生 A∩B≠∅ 。在数值上，A和B也可能和VMM管理的物理地址空间的某个子集有交集。这样isolation就破坏了！

5.软件能够保证的IOMMU/SMMU的group是很小的，在虚拟机多的时候根本不能用。（这里的group概念可以参考red hat相关描述）

问题就暴露出来了，随着系统越来越复杂（设备很多，虚拟机很多），于是就开始想办法解决。

因此，PCIe的ACS就自然而然地出来了。我们看到这里也会理解了ACS定义的那些功能的意义。例如，

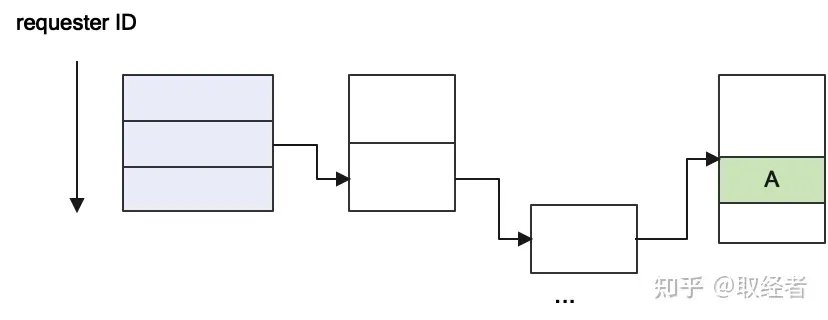
ACS P2P Request Redirect --- SW和支持P2P的RP都必须要实现，其要求设备一旦发现地址“数值”上是P2P的操作，有可能不会P2P，直接往host CPU传输。这里的为什么说有可能，是因为这个功能可能会结合ACS P2P Egress Control and ACS Direct Translated P2P这两个功能来用。后面的两个功能也好理解，因为如果设备它支持ATS，发出的DMA地址已经是PA了，那么数值上的P2P就是业务上的P2P，因此就可以直接P2P以提升性能（降低延时）。

其他的大部分功能都是按照上面的系统背景来实现的，因此我们在理解协议和实现ACS功能的时候可以灵活实现，不一定要直接翻译协议！

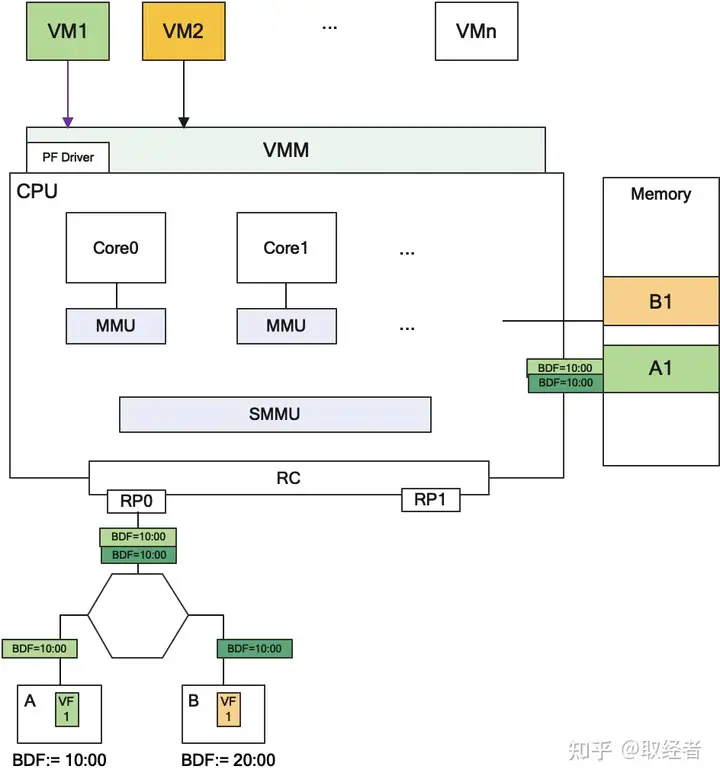
* 安全

考虑如下场景：

虚拟化的isolation：设备A只能访问VMM分配给其的内存A，同样地，设备B只能访问VMM分配给其的内存B。在现有架构体系下，是通过IOMMU/SMMU来保证isolation的。如图：

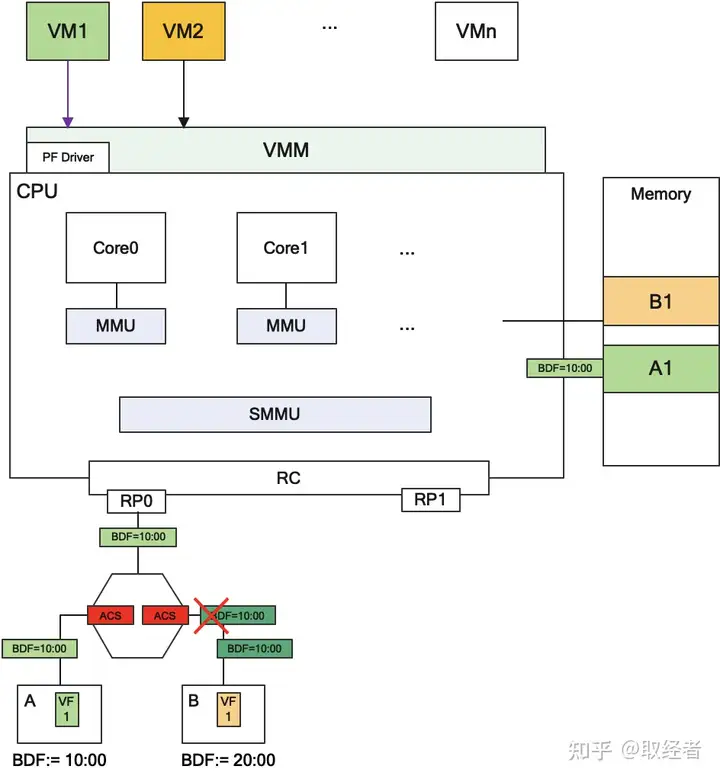


设备发出的数据流都会携带唯一的合法的requester ID，IOMMU/SMMU会基于这个ID去查页表，最终获得这个ID对应的物理地址和访问权限，从而保证isolation。



实际上，会恶意的设备或者是有bug的设备B，其可能会冒充设备A，发出携带本属于设备A上的某个function的requester ID的DMA访问操作，从而欺骗IOMMU/SMMU，去访问属于设备A的内存空间，从而造成安全漏洞。

ACS的ACS Source Validation功能便是要防止此类安全漏洞出现：在上图的SW/RP中，其会检查upstream方向的数据流的BDF是否符合预期(OS枚举时候配置的最初的拓扑，只有OS才有权限访问配置空间，这也是为什么安全的OS都不会把配置空间的访问权限开放出去)。如果不符合预期，则直接阻止数据流的继续upstream流向。



之后，ACS的所有其他功能都是围绕上的两个主题而来的，只是有些是因为做了上面的redirect后为了保证ordering不会因为路径不一样而不保证，有些是为了让拥有更多能力的设备的数据流可以继续P2P，或者系统上软件方案保证isolation下来排除上述的控制等。

## IOMMU是如何划分PCI device group的？

IOMMU的一个主要作用就是将IO设备发出的请求地址IOVA（I/O Virtual Address）转化为物理内存地址，如果没有IOMMU，那么所有的IO设备都将使用相同的物理地址空间访问物理内存。引入IOMMU后，就会引入IOVA这个地址空间，IO设备可以通过IOVA虚拟地址访问物理内存。

在虚拟化引入之前，IOMMU主要有两个功能：

1. 创建IOVA到HPA的映射，让设备能够访问任意物理内存。比如，有些I/O设备的寻址空间只有4G，但是平台的物理内存大于4G，为了让这样的I/O设备能够访问4G空间以上的内存，就需要建立一个IOVA -> HPA的映射。
2. 通过IOVA -> HPA的映射，创建内存连续的DMA访问，提高系统性能。比如，内核分配2个不相邻的4KB内存页，若没有IOMMU，需要建立两次DMA操作才能访问到这两个内存页，若有IOMMU，则可以将这两个不相邻的物理内存页映射到相邻的IOVA上，这样设备只需要建立一次DMA操作，即可访问到这两个不相邻的物理内存页。

除了以上功能，虚拟化对IOMMU的主要应用就是对设备的隔离。在引入PCIe之前，传统的PCI总线对设备的隔离是很难实现的，因为传统的PCI总线采用的是总线仲裁机制，同一时间只有一个PCI设备独占整个PCI总线，设备发出的TLP包不包含requester ID，导致RC（Root Complex）或者其他接收设备无法分辨出接收到的TLP包来自哪个设备，也就无法达到设备区分和隔离的目的。虽然PCI-X在一定程度上引入了requester ID，但是有些规则还是不够完善，还做不到完全的隔离。

对于PCIe架构而言，PCIe设备发出的所有TLP包都会包含一个requester ID（即PCI设备的Bus、Device和Function Number），这个ID可以唯一地辨别一个PCI设备，TLP的接收设备可以使用这个ID来查找IOVA的地址转换页表，这样PCIe设备就可以使用虚拟地址（IOVA）访问物理内存。这时候，对于透传给虚拟机的PCI设备，软件需要做的就是将PCIe设备所在的虚拟机的虚拟机物理地址GPA（Guest Physical Address）到主机物理地址HPA（Host Physical Address）的映射告知IOMMU，当PCIe设备发生DMA的时候，IOMMU选择目标虚拟机的GPA->HPA的映射表对地址进行转换，这样就能够让透传的PCI设备只能访问到虚拟机的物理地址空间，即分配给虚拟机的物理内存，达到设备隔离的目的。

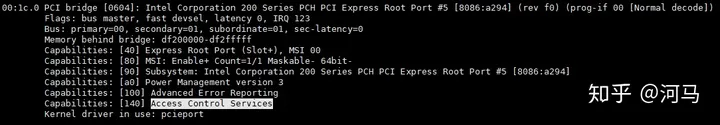
device group指的是从IOMMU角度看能够进行隔离的最小设备集。device group的划分规则包括：

1. device group中所有的设备将会共享一个IOVA地址空间。对于传统PCI总线上的设备而言，TLP中不包含Requester ID，无法进行设备区分，所以整个传统PCI总线上的设备都将被划分到同一个device group上。PCIe设备发出的TLP带有requester ID，所以可以进行设备区分，也就是可以使用独立的IOVA地址空间。
2. 从设备发出的TLP是否都能够到达IOMMU，如果设备发出的TLP可以不经过IOMMU，IOMMU就无法对TLP中包含的地址进行转换，即IOMMU无法控制设备的访问地址，无法达到隔离的目的。

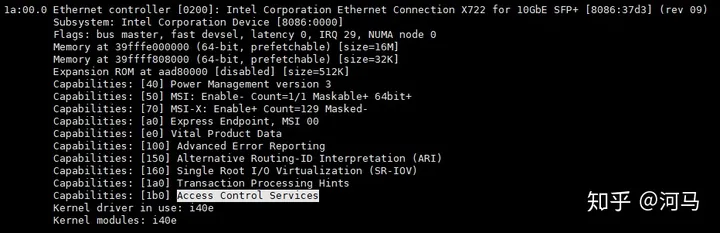
但是PCIe的另外一个peer-to-peer传输特性却给设备隔离带来的麻烦，peer-to-peer特性是指PCIe设备发出的数据包可以不一直往上提交到PCIe树的根节点，而是在中间的PCIe switch直接进行转发，转发到其他PCIe设备上。这样就会导致IOMMU无法控制到这种peer-to-peer的数据传输，达不到设备隔离的目的。比如，为了性能优化，设备驱动可以通过配置，直接将网卡接收到的数据包直接转发给存储卡，让存储卡将设备直接写到存储设备上，省去中间CPU和网卡、存储卡的交互。

PCIe ACS（Access Control Service） Extended Capability是PCIe标准中引入的用于控制对接收到的TLP（Transaction Layer Packets）进行正常的路由（即向上提交），阻塞或者是重定向转发的特性。通过控制该特性，可以让PCIe设备之间不发生peer-to-peer的数据传输，而是像传统的PCI总线一样，让设备发出的数据包都向上提交到PCIe设备树的根上，这样就可以确保PCIe设备发出的数据包都能够被IOMMU截获（可以理解为IOMMU也是位于PCIe设备树的根上）。ACS位于Root Complex，downstream port或者Muti-Function Devices（包括支持SR-IOV特性的PCIe设备），downstream port经常以PCI bridge的形式表现出来。所以，为了确保PCIe设备能够安全地隔离，需要确保PCIe设备到PCIe树根节点的路径上，所有可能执行peer-to-peer转发的downstream port、multi-function devices都需要通过ACS特性将peer-to-peer转发的功能关闭。

例如，可以在连接PCIe Root Port的PCI Bridge中找到了ACS Capability。



带有SR-IOV功能的网卡也可能提供ACS特性

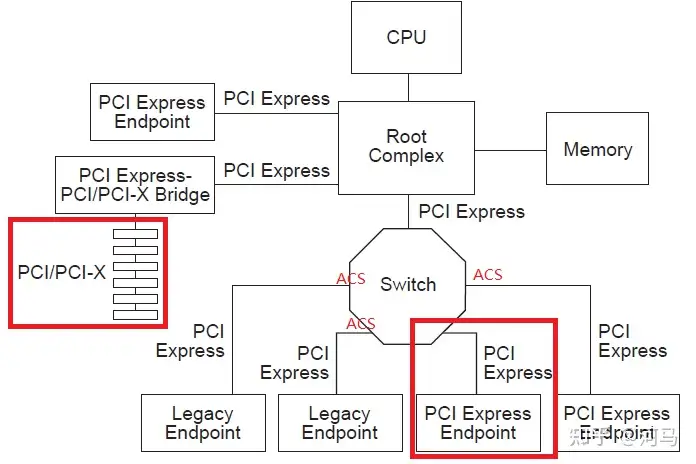


总的来说，iommu device group的划分总体来说按照以下两个规则 ：

一条传统PCI总线或者PCI-X总线上的PCI设备都划分到同一个device group。

从PCIe设备往上看，在设备通往PCIe树根节点的路径上，所有的PCIe downstream port和multi-function device都需要通过ACS特性，将peer-to-peer转发的功能关闭。若某个PCIe downstream port或multi-function未通过ACS特性关闭peer-to-peer特性，则下面的所有设备都必须归到同一个iommu group，否则该PCIe设备就可以独立成一个iommu group。

比如，以下面的拓扑图为例



红框中的PCI Express-PCI/PCI-X Bridge下面挂的设备都将被划分到同一个iommu device group中。

PCIe switch的某个downstream port未开启了ACS特性，则该PCIe switch port可能将接收到的PCIe设备发出的TLP包转发到该switch port下面的其他设备，所以为了保证完全的隔离，只能将该switch port下面的所有设备都归到同一个device group中。

系统中其余的PCIe switch port都通过ACS特性的控制确保接收到的TLP包都往上传递给root complex，所以其余的PCIe设备，每个设备都可以各自独立为一个iommu device group。

## 为什么GPU Direct RDMA不能使用IOMMU？

<https://zhuanlan.zhihu.com/p/701605928>

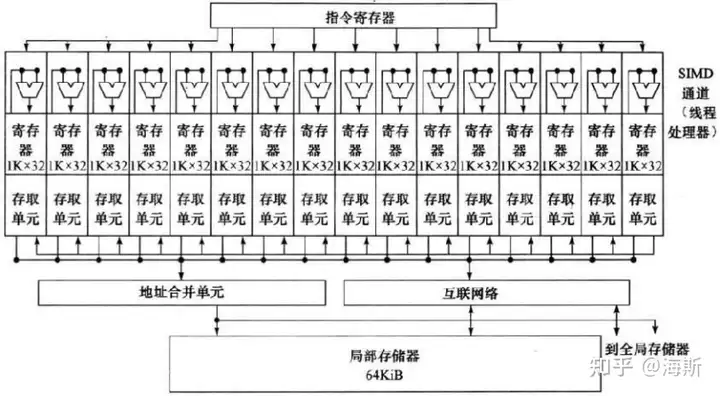
### 1 GPU的使用原理

要完整的解释清楚这个问题，我们先从源头开始——GPU在AI领域的使用原理。

GPU与CPU的设计不是同源的，二者在设计理念上存在较大的差异，下图对比了这种差异，与CPU相比，GPU控制单元较少（黄色部分），但运算单元超多（绿色部分）。

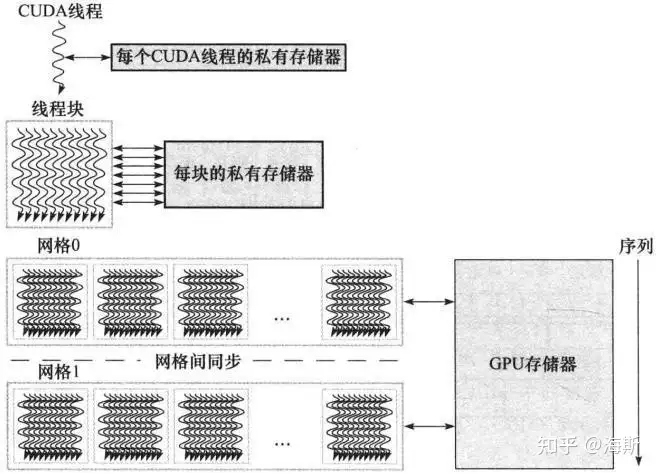


在运行上，GPU属于SIMT（单指令多线程），下图展示了16个线程处理器对不同数据同时执行相同指令的情形。



在硬件角度，GPU包含多个上图这样被称为SM的结构，SM（Stream Multiprocessor）包含多个SP（Stream Processor，也叫线程处理器），每个SP中有独立的计算单元、存取单元和寄存器。

在软件角度，CUDA把这些抽象为grid、block、thread 3级结构：GPU包含多个grid，grid包含多个block，block包含多个thread。一般block对应SM，thread对应SP，在每一级中都有自己的本地内存，分别被称为global、shared和local内存。

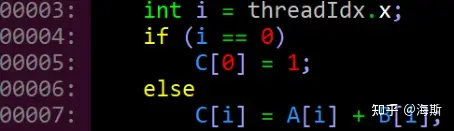


GPU执行时的SIMT模型，就是通过block来达成的：在同一时刻，block中的所有线程都执行相同的指令，但指令操作的数据不同（通过在各自SP的同名寄存器中装入不同值来实现）。block的尺寸在程序运行时配置，可以是一维、二维、三维，有两点约束：1、block的尺寸必须是warp尺寸的整数倍（warp——线程束，它是共用指令指针的一行SP，warp尺寸一般是固定的，例如32个）。2、各维乘积不能超过grid中的thread总数量。

我们可以想象这样的结构非常适合于做矩阵计算，例如：计算C[M][N] = A[M][N] + B[M][N]，如果使用CPU计算，那么需要做两层循环，执行M x N次“取数据 + 加法 + 存数据”指令。但如果使用GPU，只要创建一个M x N的二维block，将A、B装入GPU内存，只需要执行一次“取数据 + 加法 + 存数据”指令，block中的各线程就会同时计算出对应的C[i][j]。这个加速比是非常惊人的，这也就是GPU现在被广泛应用于AI训练的原因。

从上面的介绍中，我们看出GPU的本质是对计算的加速，没有GPU也可以做AI训练，只不过就是“模型训练完成日，家祭无忘告乃翁”了，目前这个加速效果让大众对AI吃了一惊，并开始影响我们的社会和生活，但这些技术仍然是第三次工业革命而产生的半导体 + 计算机技术，从技术本身看，并没有什么令人震惊的，等到数学、物理和制造工艺的再次突破，开启第四次工业革命，那才是值得我们在弥留之际对后代说一句“家祭无忘告乃翁”的。

言归正传，GPU计算很强，但这只霸王龙有着不协调的身体比例，它的控制较弱。这完全是这种计算结构导致的，并不是增加一些控制单元就能搞定的，我们看这段包含if/else的代码在GPU的block中是如何执行的。



因为block内的所有thread共用一个指令指针，所以没有办法做基于条件的跳转，这些语句要顺序执行，当执行到第5行的语句时，只有thread 0执行，而其它thread空闲，执行到第7行时，thread 0空闲，其它thread执行。我们发现，增加了if/else后，相比于原始的C=A+B的代码，它的执行时间翻倍了。实际上，这种情况正是在CUDA编程中要尽量避免出现的。要让绝大多数的thread都能有效的执行指令，这是CUDA编程的原则。举这个例子不是想将话题引向如何优化CUDA程序，那就走向数据并行编程了，这个例子的主要目的还是想说GPU善于做计算，而程序不单有计算，还有控制、输入、输出等，这些要由CPU来做。

所以，GPU编程是GPU + CPU的混合编程，一般CUDA程序的步骤：

1、分配GPU内存。

2、拷贝主机内存到GPU内存。

3、调用CUDA kernel函数来执行计算。

4、把计算完成数据拷贝回主机内存。

5、释放GPU内存。

这5步是在CPU上执行的程序，只有第3步中的CUDA kernel函数才是在GPU上执行的。

下面是一个计算矩阵相加的代码示例，第14~20行是在GPU上执行的代码，第22~43行是在CPU上执行的代码。

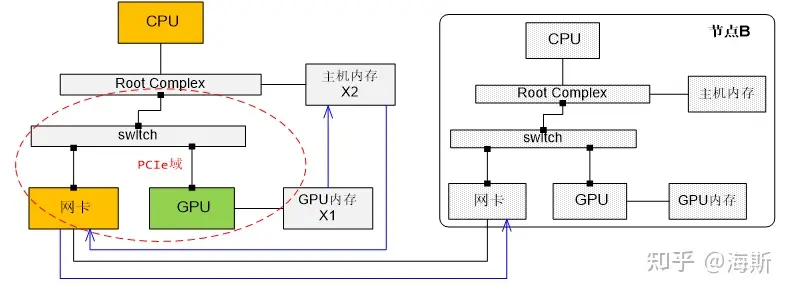


这些代码已经高度屏蔽了底层细节，使程序员更关注算法本身。实际上如果看穿这个封装的本质，GPU和CPU之间通过PCIe互联，代码中主机内存与GPU内存之间的拷贝、CPU调用GPU上的函数这些都是通过对PCIe BAR空间的读写和PCIe DMA来实现的。理解这一点，就能理解为什么CPU要向GPU内存拷贝数据，而不是GPU直接使用主机内存做计算。

### 2 智算集群中GPU互联方式

#### 2.1 传统方式

上面我们说了GPU在单台服务器内的工作方式，实际上现在的任何一个大模型的计算量都不是单台服务器能够搞定的，必须使用集群，这就涉及到计算节点之间的互联，本质上就是GPU之间的互联。对于少量GPU的互联，NVIDIA开发了nvlink + nvswitch，这项技术是真正的GPU之间的互联，它的数据路径没有PCIe参与，带宽非常高，但目前它的传输距离较短、节点数量也较少，在大型智算集群中还不能完全替换掉基于网卡 + 交换机的传统互联方式。本文的GPU Direct RDMA技术就是传统的网络互联方式中的一个加速技术，我们还是先从传统的互联方式说起。



上图中要把左侧GPU中的数据发送到右侧节点B的GPU中，流程如下：

1、CPU对GPU发起DMA请求，让它把GPU内存中X1的数据拷贝到主机内存X2中。

2、GPU执行DMA操作，将X1拷贝到X2，拷贝完成后通知CPU。

3、CPU再对网卡发起DMA请求，让它把X2数据发送到远端。

4、网卡执行DMA操作，将X2数据发送到网络上。

我们先将话题稍稍岔开，先解释一下第3步是如何执行的。

在没有RDMA之前，网卡将主机内存中的数据发向远端的典型方式是使用TCP/IP协议栈，在数据路径上需要软件参与做用户态到内核协议栈之间的数据拷贝，有了RDMA技术之后，网卡可以直接发送APP使用的主机内存中的数据，既没有软件参与的内存拷贝，也不经过软件实现的TCP/IP协议栈，而且RDMA的API语义是基于内存块的，不像TCP是基于字节流的，这种API语义能够简化APP编程，也提高了接收端代码的性能。

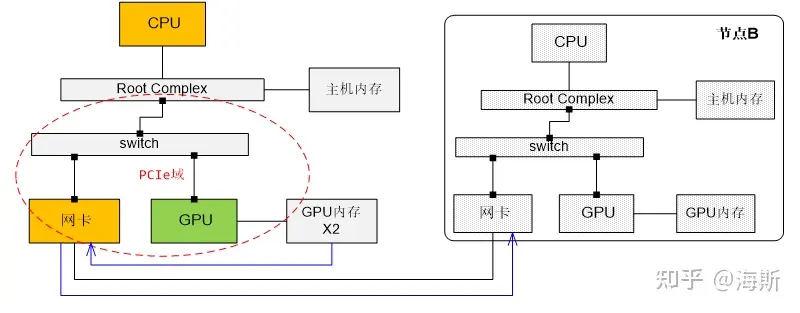
回到上文，使用RDMA，在数据路径上没有软件拷贝，但有2次串行的DMA数据拷贝，即使可以通过异步队列来隐藏这个串行开销，但所有的DMA操作都要竞争PCIe带宽，我前面没说，业界现在对PCIe的诟病中，其中一点就是PCIe带宽的增长太慢（所以NVIDIA绕开了PCIe搞nvlink）。

在接收端也有类似的2次DMA的问题。

总之，要把左侧GPU中的数据发送到右侧节点B的GPU中至少要经过4次DMA拷贝。

#### 2.2 GPU Direct RDMA

终于说到它了，我们来看GPU Direct RDMA是如何减少上述过程中的DMA次数的。



要把左侧GPU中的数据发送到右侧节点B的GPU中，流程如下：

在发送端，CPU对网卡发起DMA请求，让它把GPU内存中的X2数据发送到远端（省略了RDMA的细节）。

在接收端，网卡收到报文后将数据放入GPU内存。

这样在数据收发过程中只有2次必要的DMA拷贝。

### 3 PCIe DMA

从PCIe角度来说，GPU Direct RDMA并不是新技术，因为PCIe规范本就能支持设备之间的DMA读写。

#### 3.1 传统的PCIe DMA

以网卡对主机内存的DMA读操作为例（网卡的发包流程）。



先要明确两个重要的概念：存储器域地址 和 PCIe域地址。顾名思义，前者是CPU使用的地址，通常也被成为内存物理地址；后者是PCIe设备使用的内存地址。因为在传统的x86架构下，一般这两个地址的值是相等的，所以容易混淆。但是搞不清这两个地址的区别，就不能理解后面的内容。

流程如下：

1、CPU对网卡发起DMA读请求，让它把地址X的数据发送到网络上（这个X是PCIe域的地址，只不过CPU在设置DMA时数值上等于主机内存的物理地址）。这里忽略了通过异步队列来提高性能的实现，这个实现对网卡很重要，但对我们现在要说的DMA流程不重要。

2、网卡对地址X执行DMA读操作，它向PCIe switch发出TLP消息，类型为“存储器读请求”，消息中携带requester ID和地址X（实际上是若干个TLP消息，并且消息中携带的地址因为对齐和分片的原因，也不一定就是X，但这在本文中不重要）。

3、PCIe switch中包含两类路由表：A、根据PCIe域地址范围对应到端口，B、根据设备ID对应到端口。当该TLP到达PCIe switch后，它根据地址X查询路由表A，如果没找到，那么向上游端口转发TLP——对主机内存的访问就是这种情况，缺省上送。

4、该TLP到达RC后，它发现该地址属于内存控制器，于是使用地址X读取内存，将数据装入“读完成TLP”中，然后根据requester ID查找路由表B，将它转发到PCIe switch。

5、PCIe switch根据requester ID将读完成TLP转发给网卡，网卡最后完成组包。

对于设备间的TLP读操作，以网卡读取设备2中的内存为例：

CPU提前为设备2中的内存在PCIe域中分配地址范围AR2，然后将AR2 - 设备2 ID写入到RC并扩散到各级PCIe switch中。

在第1步中，CPU对网卡发起DMA读请求时，它设置的地址Y属于AR2。

在第3步中，PCIe switch查找路由表将TLP转发到port2。

在第4步中，TLP到达设备2后，它会把PCIe域的地址Y转换为本地地址，然后使用本地地址读取数据，组装读完成TLP。

其它流程与上面相同。

#### 3.2 基于IOMMU的PCIe DMA

IOMMU是为了支持设备硬件辅助虚拟化而引入的技术，intel称为VT-d，AMD称为AMD-Vi，ARM称为SMMU，它们的实现细节不同，但原理相同，下面以intel VT-d为例说明它的原理。

IOMMU是集成在RC中的一个部件，当设备访问主机内存时，它将设备访问时使用的地址转换为主机内存的地址。

IOMMU的3种使用场景：

1、在虚拟化场景中，guest可以使用GPA来作为DMA地址。

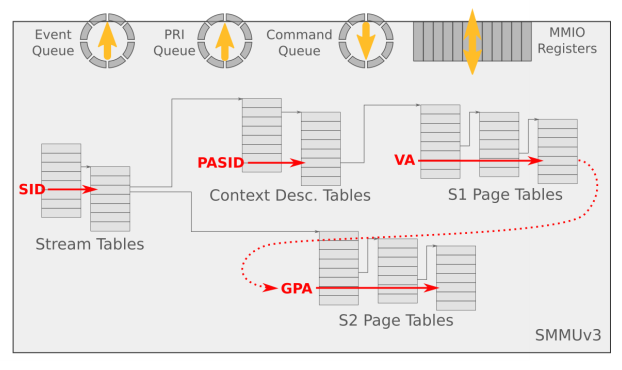
2、在裸金属场景中，host可以使用HVA作为DMA地址。

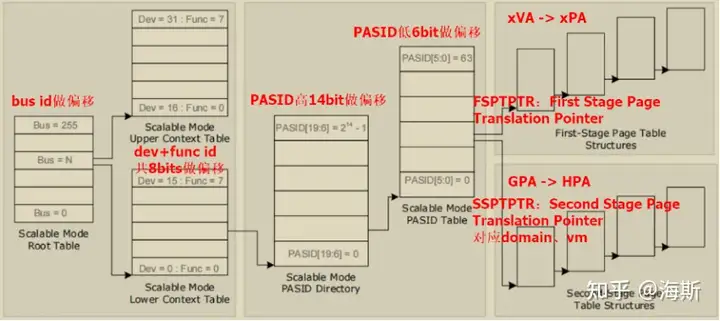
3、在虚拟化场景中，guest可以使用GVA来作为DMA地址。

这3种场景中IOMMU都起到地址转换和安全隔离的作用。

注意：这里仍然要区分两种地址域，使用X作为DMA地址的意思是说，把存储器域的地址X的值作为PCIe地址域中的值，只是值相等，但二者的含义是不同的。

下图描述了场景3中设备执行DMA读写时IOMMU从GVA到HPA的地址转换过程：





1、PCIe TLP中携带设备的requester ID和PASID。

2、IOMMU使用requester ID通过Root Table和Context Table找到PASID表，然后通过PASID找到PASID表项，里面保存着stage1和stage2转换表的入口地址。

3、使用TLP中的地址查找stage1表，输出IPA。

4、使用IPA查找stage2表，输出HPA。

其它场景可以理解为对上述流程的简化。

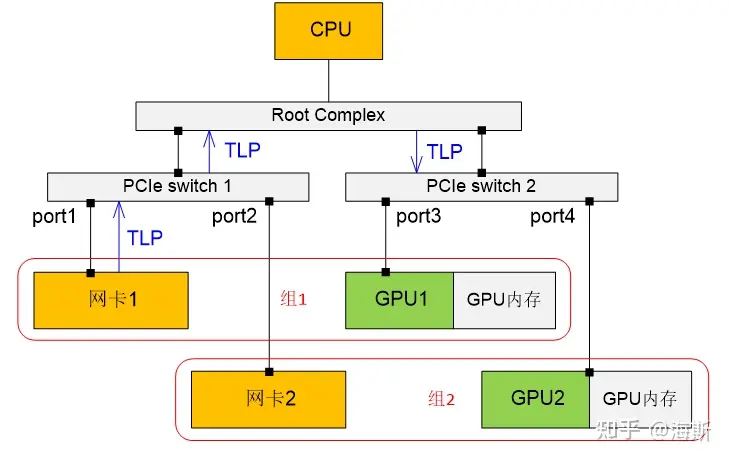
4 GPU Direct RDMA为什么不支持IOMMU

**通过上文的描述，我们能够得出GPU Direct RDMA不支持IOMMU的原因。**

原因1：IOMMU是针对主机内存的地址转换，对于设备之间的DMA读写，TLP不会经过IOMMU。

这个比较容易理解，可以参照前面的设备间DMA的流程。但大家可能会想，在GPU内部也有一个GPU MMU，用于将TLP中的地址X转换为GPU中的内存地址，利用它是否可以实现与IOMMU类似的效果呢？不行，见原因2。

原因2：读写请求TLP在PCIe桥中路由时查找的是地址范围表，这决定了两个GPU只能使用同一个PCIe域地址空间中的不同地址范围。

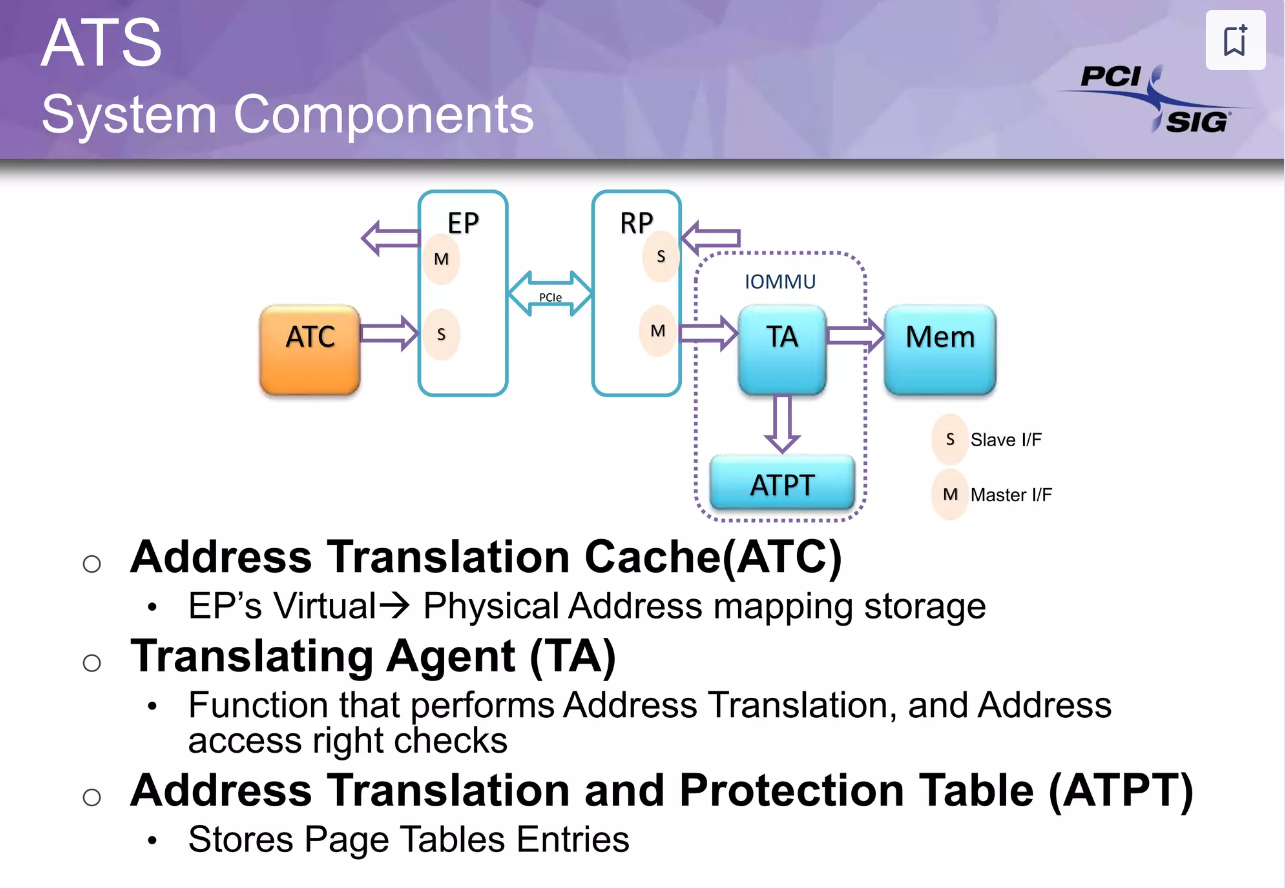


假设网卡1 + GPU1被直通给VM1，网卡2 + GPU2被直通给VM2，虚拟机的一个原则就是不感知自己是虚拟机，所以VM1和VM2独立配置GPU内存的PCIe域的地址范围，如果这两个值是相交（甚至是相同）的，那么网卡1对GPU1内存做读写请求时，当TLP到达PCIe switch 2时，PCIe switch 2根据地址查表，它无法判断出应该转发到port3还是port4

# ATS和SVM

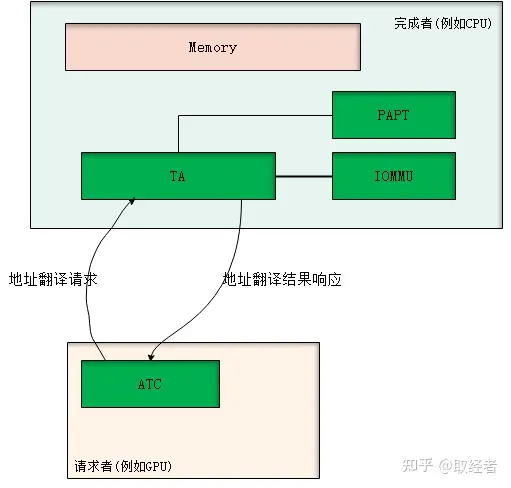
**ATS目的：**

简单理解，CPU负担过重，瓶颈日益明显，将CPU中IOMMU/SMMU的对应表项卸载到EP，EP后续查自己的表，对地址进行翻译，bypass CPU的地址翻译。





ATS全称是Address Translation Service，顾名思义，就是一个地址翻译服务机制。PCIe下的ATS是以CPU为中心，PCIe总线上的各个设备可以通过ATS机制向主机申请未翻译地址对应的物理地址映射以及响应的属性、权限等信息。



一般地，在PCIe体系下，发起地址翻译请求的设备叫请求者，也叫client，而处理地址翻译请求的（即CPU）叫完成者，也叫home。因此，对于地址翻译请求的流程如下：

1、client有数据访问需求（或者是prefetching引入的需求）；

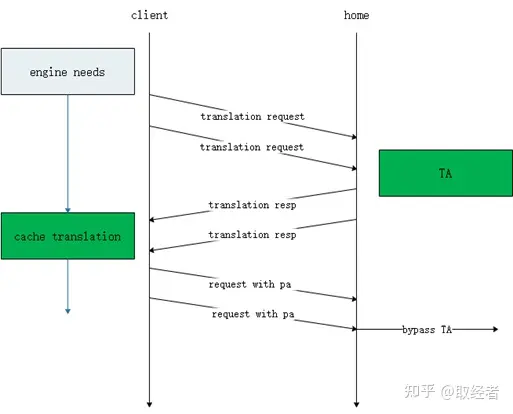
2、client发出翻译请求给home侧，携带需要访问的虚拟地址空间块和响应属性、提示信息；

3、home侧进行翻译，之后返回翻译结果给client；

4、client在本地cache翻译结果；

5、client引擎发出数据访问，查本地cache，直接发出查完后的结果访问home侧；

6、home侧直接bypass TA，访问home本地内存。



* ATS机制想要解决的问题（优势）：

1、能够分担主机（CPU）侧的查表压力，特别是大带宽、大拓扑下的IO数据流，CPU侧的IOMMU/SMMU/VT-d的查表将会成为性能瓶颈，而ATS机制正好可以提供将这些查表压力卸载到不同的设备中，对整个系统实现“who consume it， who pay for it”。

2、对于IO设备往CPU的数据流，其IOMMU/SMMU/VT-d的查表性能在整个IO性能中显得极为关键，查表性能的好坏至今影响IO性能的好坏。而根据现今大量学术界和工业界的研究表明，决定查表性能的好坏的一个最为关键的点是TLB的预测。而像传统PCIe IO数据流，在CPU侧集中式做IOMMU/SMMU/VT-d的查表，对于其TLB的预测（prefetch）是很困难，极为不友好的。因为，很多不同workload的IO流汇聚到一点，对于TLB的预测的冲击很大，流之间的干扰很大，很难做出准确的预测，从而TLB的命中率始终做不到太高，从而影响IO性能。而ATS的机制恰好提供了一个TLB的预测卸载到源头去的机制，让用户（设备）自己根据自己自身的业务流来设计自己的预测策略，而且用户彼此之间的预测模型不会受到彼此的影响，从而大大提高了用户自己的预测的准确性。抽象来看，这时候的设备更像是CPU核，直接根据自身跑的workload来预测本地的TLB，从而提升预测性能，进而提升整系统的预测性能。

ATS机制就是一个分布式的地址翻译系统，对于异构架构的地址问题也起到了很好的支持效果。例如，NVIDIA和IBM最近搞的NVLINK互联CPU和GPU，其就是用了ATS来实现SVM，这个后续我想详细介绍一下。

当然，ATS也会带来如下两个比较严重的问题：

1、安全问题。也就是有了ATS后，设备会直接发出PA来访问CPU的内存空间，而CPU这边将不再查页表，从而也不再有地址访问权限的检查。因此，对于malicious设备，或者是被入侵的设备，又或者是设备自身设计的bug，其会发出PA访问那些其没有权限访问或者是不属于其的物理地址空间，从而带来系统的isolation的破坏，系统安全的威胁等安全问题。现有机制下，目前还没有解决办法。但，本人有专利可以解决，而且本人也有更好的机制可以解决，目前保密，暂时不能公布。本人的专利的话，后面有时间可以写一下。

2、无效化问题。也就是，在CPU侧被cache的页表发生变化（例如权限发生了变化，对应的PA需要回收SWAP等）的时候，此时CPU需要发出无效化指令给对应cache过的设备（ATC）来通知这件事，这会影响CPU回收页表/物理空间的性能。这就意味着CPU侧要实时跟踪页表被谁cache走了，维护这些信息也带来了成本的开销增长。

3、对设备的设计要求比较高，可以看到，ATS在某些时刻势必增加了延时。而为了减少甚至消除影响，就需要设备设计出具有良好的ATC机制，包括ATC的预测、地址翻译请求的长度、发送翻译请求的时机等，这里的设计空间都比较大，对设备的要求都比较高。

前面的问题1和2是系统需要解决的功能问题，后者3是一个趋势。我想象中的设备将会越来越智能，设备和CPU之间将会是点对点的平等计算地位，在异构体系下，它们是平等的计算关系。它们之间SVM、具有cache coherence等，从而第3看起来不是问题，而是趋势。

好了，又侃侃而谈这些空虚的东西，具体的实现体会，我有时间可以写一下，但必须要等到我大厂的专利、产品出来后才能写。下次我想抽时间写一下异构下的SVM。

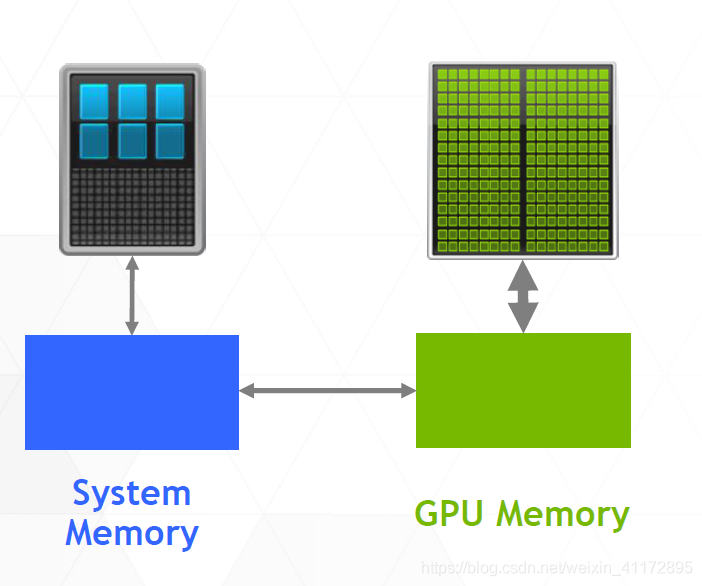
## CXL ATS 介绍

# <https://blog.csdn.net/weixin_40357487/article/details/131744826>

# CPU与GPU统一虚拟内存

## UM下的CUDA编程

在PCIE接口上插上GPU，就变成了我们Desktop形式的CPU/GPU，CPU与GPU分离，各自有属于自己的物理地址：



在这种情况下，我们比较熟悉的CUDA编程是用cudaMalloc和cudaHostMalloc分别分配device和host内存，先后显示调用cudaMemcpy进行拷贝。

但是使用的UM (Unified Memory)的cudaMallocManaged就不需要显示的将hostMem内容拷贝到deviceMem了，例子如下：

{

char \* array = nullptr;

cudaMallocManaged(&array, N) //分配内存

fill\_data(array);

qsort<<<...>>>(array); //GPU process

cudaDeviceSynchronize();

use\_data(array); //CPU process

cudaFree(array);

}

同一个指针array可以同时分别在CUDA和CPU使用，但编程框架中只显示分配了个地址数值。所以必然有底层逻辑在CPU和GPU各自独立的空间之间拷贝了数据，使其表现为程序员看似GPU和CPU使用了同一段地址。

那么问题来了，CUDA Host Runtime悄悄的做了什么。

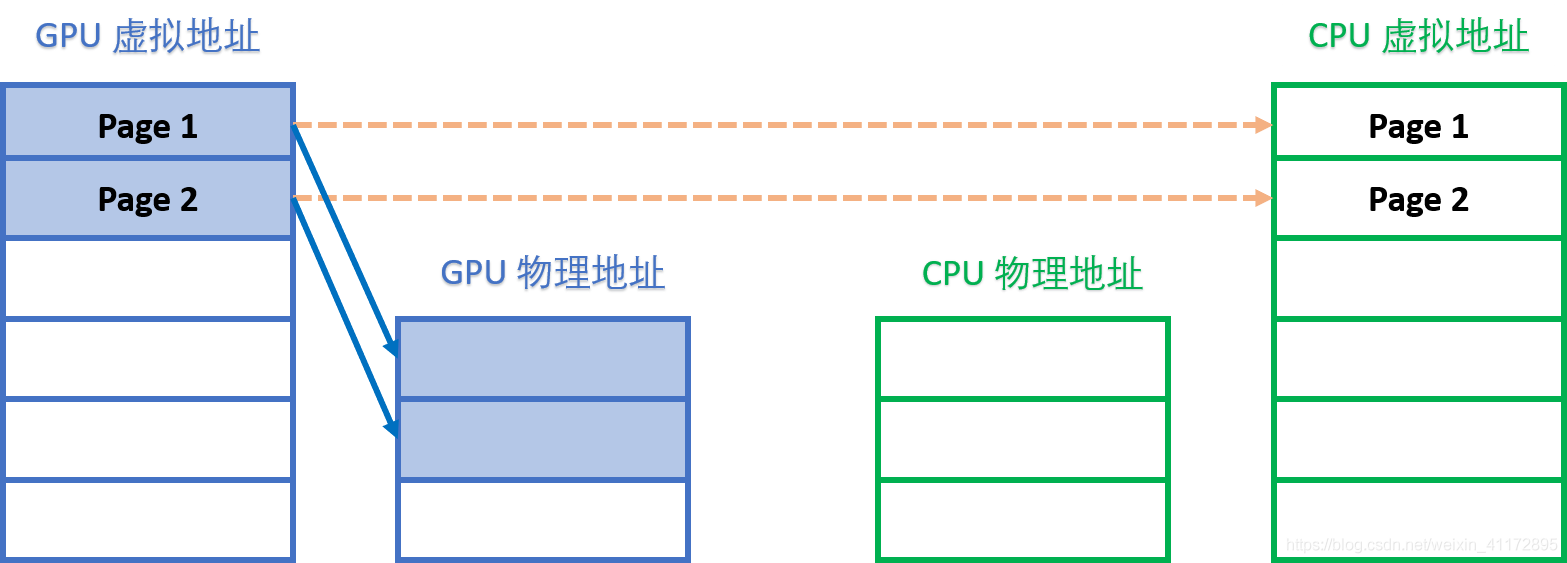
## UM的实现原理

我们以如上的程序为例子：

1. cudaMallocManaged分配CUDA内存

Pascal及以上架构的GPU是可以处理页错误(Page Fault)的，cudaMallocManaged是cudaRuntimeAPI，其不仅仅只为CPU分配内存，还将CPU端的页信息发送到GPU上:

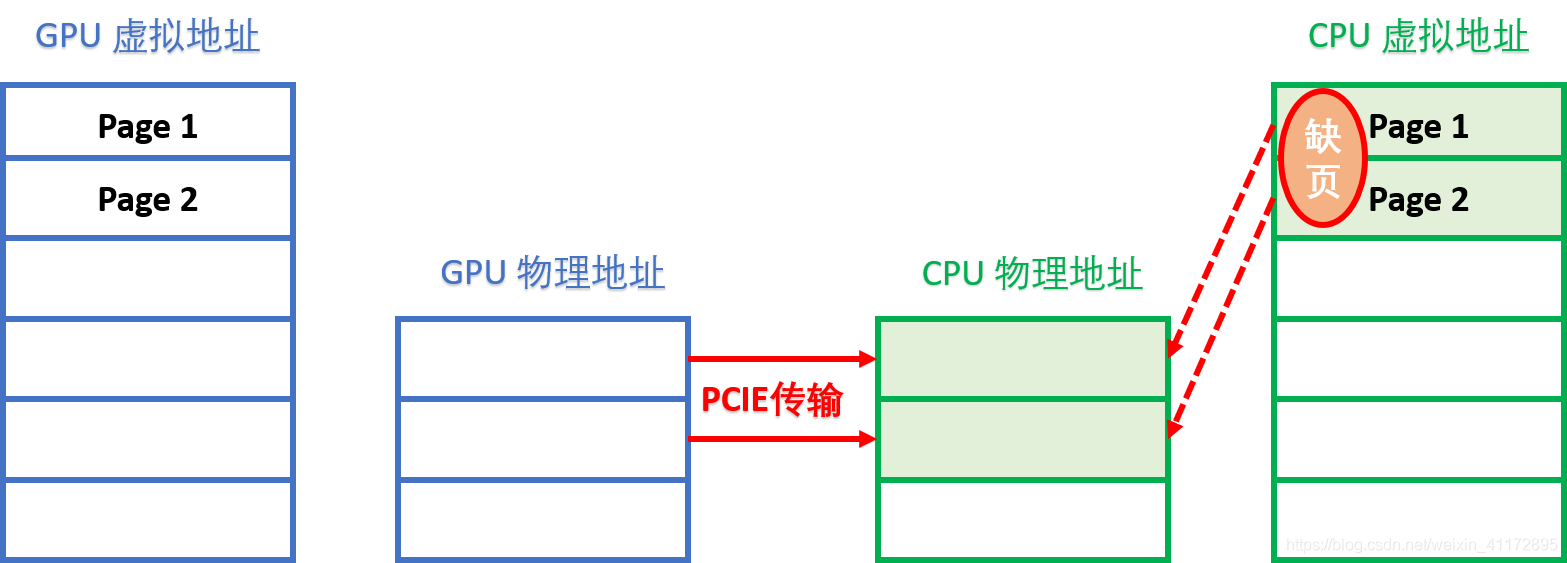
图中的例子所示，假设array占用两个内存页，内存分配在GPU上，但cudaRuntime同时在CPU端创建了指针array的内存页：



值得注意的是，由于CPU和GPU都用相同数值的array指针，所以页的页号在CPU和GPU端是相同的。

1. CPU缺页中断处理

接下来程序在CPU端调用fill\_data(array)但实际上CPU没有为array实际分配内存空间，仅仅是有保留的页表存在，所以必然会产生缺页中断：

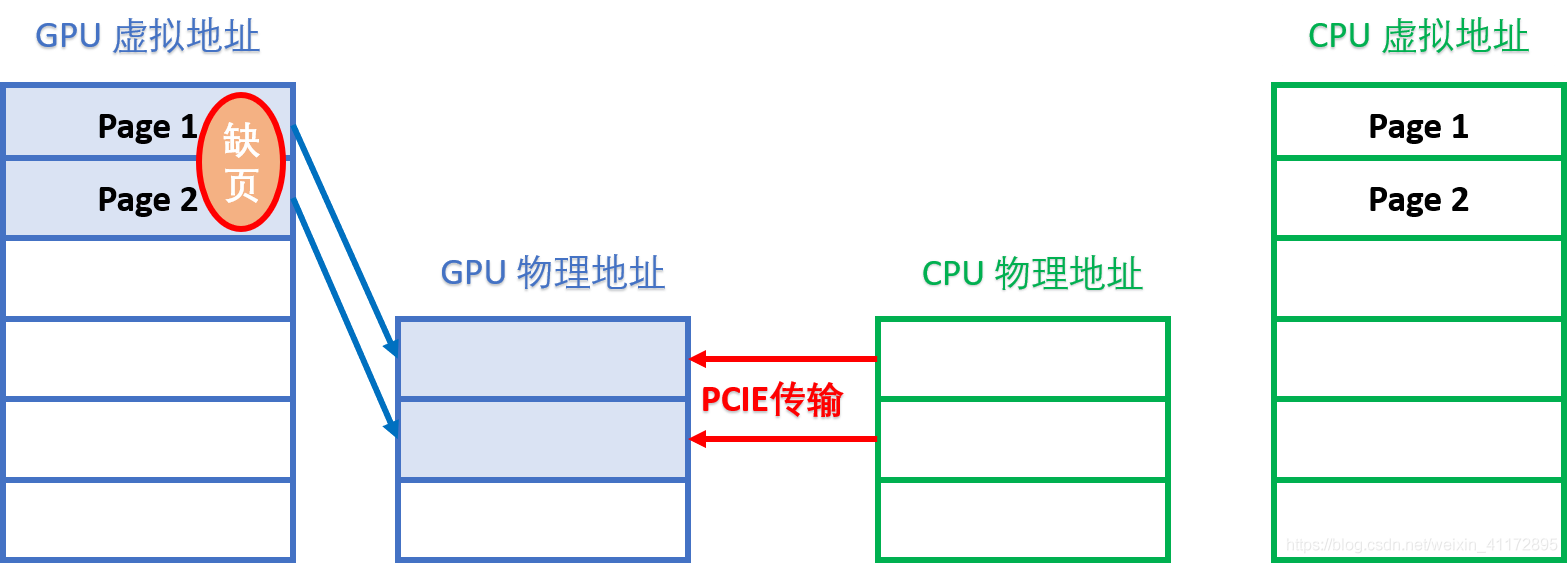


此时的缺页中断会促使GPU内存的内容通过PCIE总线migrate到CPU内存当中，待CPU处理完缺页，fill\_data才函数会继续处理。

当GPU的页传输到CPU当中，为保证数据一致性，GPU的页就标记为失效。

1. CUDA Kernel 运行：

之后程序调用cudaKernel: sort<<<....,s>>> 此时，轮到GPU发生缺页中断和数据migrate了：



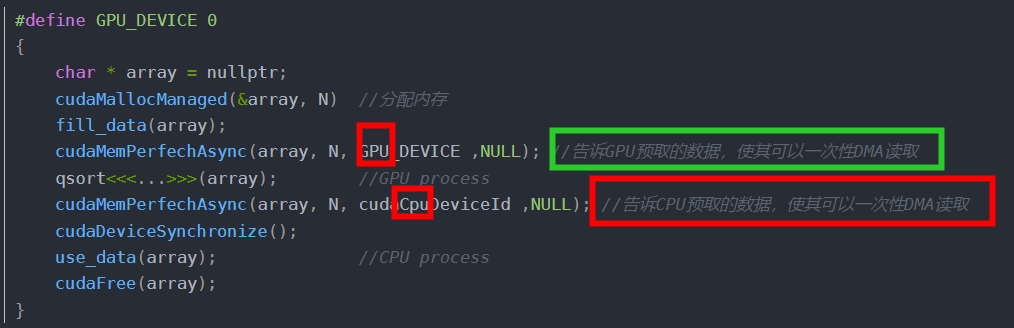
待Kernel处理完毕，cudaDeviceSyncornize等待万里，CPU再执行use\_data首先会发生缺页中断，数据再传回CPU。

## UM的性能缺陷及优化

1.用 cudaMemPrefetchAsync 避免缺页中断

毫无疑问，除去数据频繁搬运不说。这些缺页中断看着就很让人蛋疼，内存页的数据量小，页的数量很多。导致缺页中断次数多，CPU在用户态和内核态之间来回摇摆，而且内存页的传输不一定用到了DMA。

为了提升传输性能，这时就要引入cudaMemPrefetchAsync，调用DMA来异步传输，再通过cudaStream同步。这样指定的device端由于知道预取多大的数据，就不会频繁的发生缺页中断了。



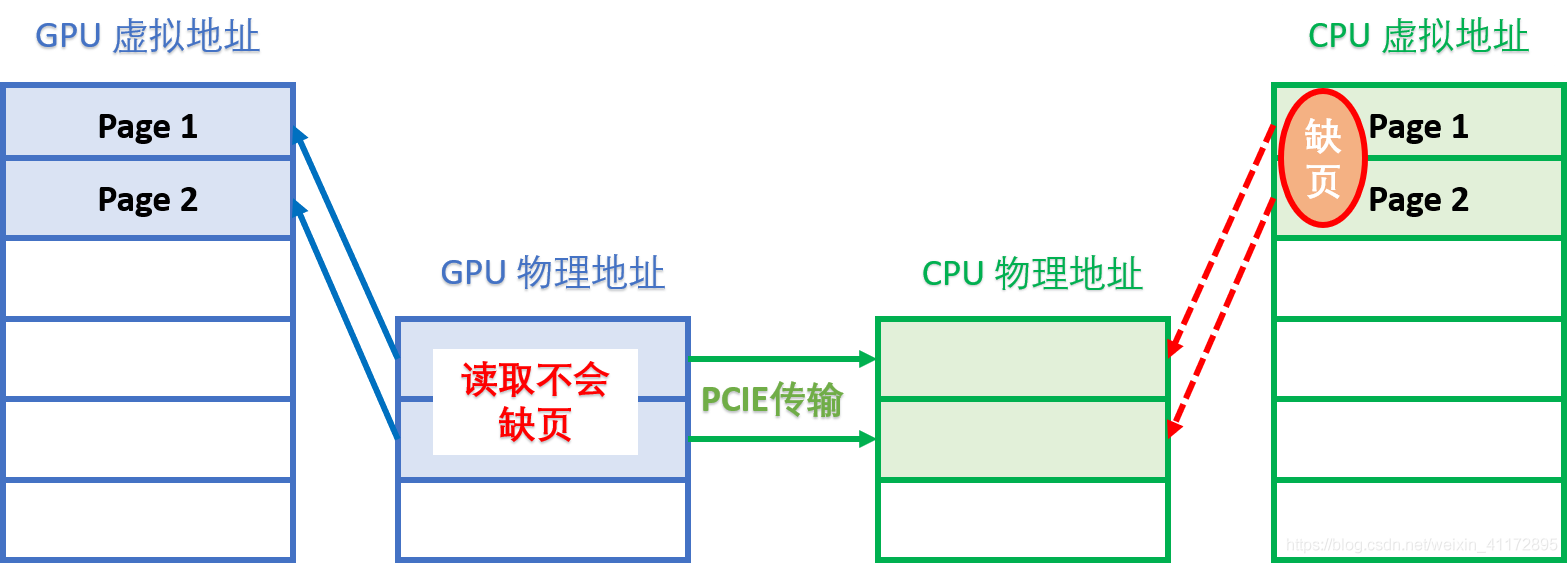
2.用cudaMemAdvise告知分配内存的特性

对于使用cudaMallocManaged开辟的内存，当cudaKernel启动时，CPU端的内存页会变为失效状态。当CPU做处理时，GPU端的内存页会变为失效状态。这么做的原因是为了保证数据一致性，但有些应用场景不要求这么严格的数据一致性，场景比如CPU和GPU都对同一片地址空间进行读取操作而没有写入操作就不存在数据竞争，CPU和GPU原本能同时进行操作，然而却被UVM子系统却杜绝了这样的并行操作。

此时就需要cudaMemAdvise上场了，它能告知一片地址空间的特性，有了先验信息，driver会在背后做更多的优化。

* A. cudaMemAdviseSetReadMostly 及 cudaMemAdviseUnSetReadMostly

这两个是cudaMemAdvise的Flag，用来为某个device设定/解除内存空间ReadMostly的特性，device所指的设备可以有一个只读副本而不发生数据迁移，当两端没有写入时，两个副本的数据是一致的。



如图所示CPU需要使用地址空间，而此地址空间已经通过cudaMemAdvise设定CPU为cudaMemAdviseSetReadMostly时，CPU产生一个只读副本(copy)，而不是数据迁移(migrate)。这样CPU/GPU端依然可以同时读取数据。当有一端写入数据时，对端数据立即失效而只在写入端存在一个副本。

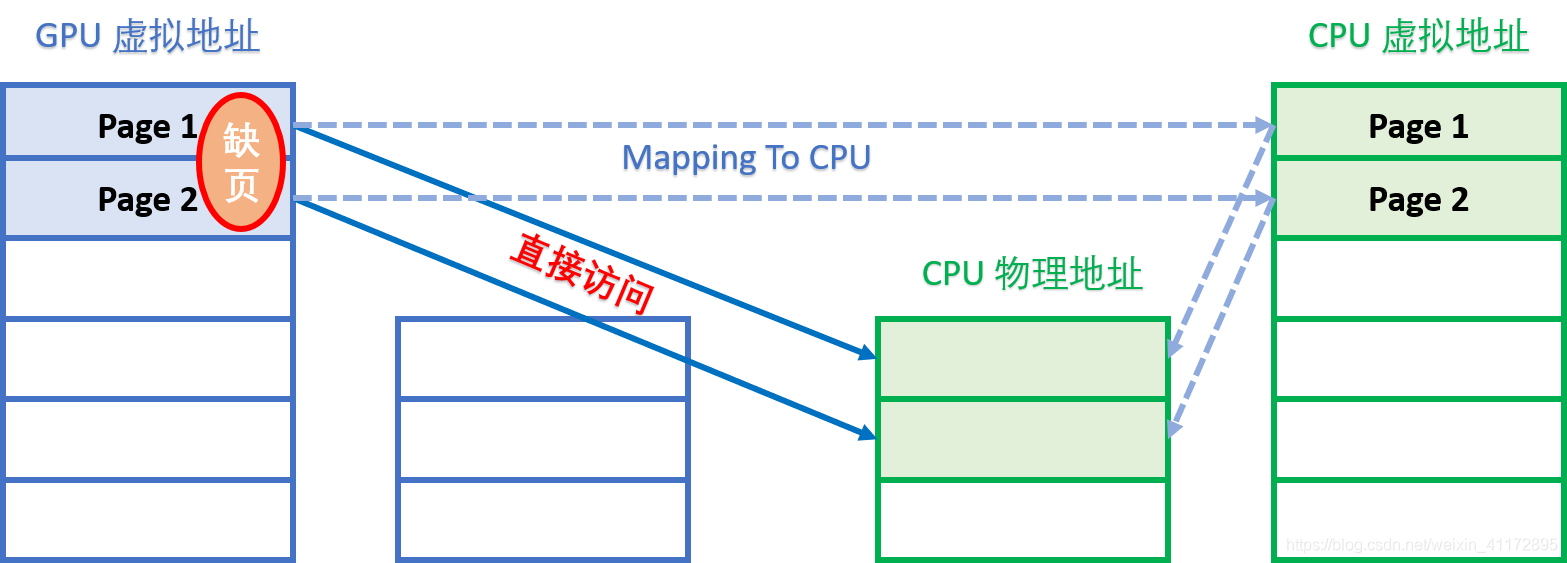


典型用法是配合cudaMemPrefetchAsync使用，这样CPU和GPU都没有缺页中断了，而且可以同时执行。



B.用 cudaMemAdvisePreferredLocation来指定数据存储位置，数据只在指定设备上一个副本，不发生迁移和拷贝 (Resist Migrations)

不发生迁移，意思是数据仅可以存储在指定的device端，缺页中断时产生到指定device的内存映射，而不复制到本地。(当不能建立地址映射表，如CPU不能访问GPU内存，此时依然会数据迁移)如图所示：





* C.cudaMemAdviseSetAccessedBy标志

与cudaMemAdvisePreferredLocation作用相似，都是Resist Migrate。用于GPU访问CPU，或GPU之间访问，地址映射表会立即建立，不用等到发生缺页中断。同时，若数据发生迁移，这其余设备的映射表自动更新。用于不关心数据Location但关心缺页中断影响性能的情况。比如Multi-GPU的Cluster。

## 内存管理 同步

在CUDA中，使用\*\*cudaMalloc()来分配设备内存，使用cudaFree()\*\*来释放设备内存。

* cudaMallocManaged 统一内存管理

\*\*统一虚拟寻址（Unified Memory）：使用cudaMallocManaged()\*\*来分配可以在CPU和GPU之间共享的内存。无需关心数据在主机或设备上。

cudaMallocManaged是一个CUDA运行时应用程序接口(API)函数，用于分配统一内存(unified memory)。统一内存是CUDA的一种内存管理模型，它提供了一个单一的、在主机和设备间共享的内存空间。

这个函数的原型如下：

\_\_host\_\_ cudaError\_t cudaMallocManaged ( void\*\* devPtr, size\_t size, unsigned int flags = cudaMemAttachGlobal );

devPtr 是一个指针，指向分配的内存的地址。

size 是请求分配的内存的字节数。

flags 是一个可选的参数，用于指示内存的附着行为。默认值是 cudaMemAttachGlobal，意味着这块内存在所有的CUDA流中都是可见的。

cudaMallocManaged函数的工作原理是基于按需页面迁移"(on-demand page migration)的机制。当GPU要访问一块统一内存时，如果这块内存当前并不在GPU的物理内存中，就会触发一个页面迁移，将数据从CPU内存迁移到GPU内存。同样，当CPU要访问一块统一内存时，如果这块内存当前在GPU内存中，就会触发一个页面迁移，将数据从GPU内存迁移到CPU内存。

页面迁移可能引发的性能开销。如果主机和设备频繁地对同一块内存进行访问，可能会导致"抖动"现象，即数据不断地在主机和设备间迁移，这会大大降低程序的性能。

* cudaDeviceSynchronize() &cudaStreamSynchronize()等待操作完成

cudaDeviceSynchronize()是一个CUDA运行时应用程序接口(API)函数，用于阻塞主机代码的执行，直到设备上所有先前的任务都完成为止。这包括内核执行以及设备与主机之间的内存传输。

这个函数在调试和性能测量中非常有用，因为它可以确保所有设备上的任务在继续执行主机代码之前都已完成。例如，如果你想测量GPU内核的执行时间，你需要在内核启动和停止之间插入cudaDeviceSynchronize()来确保内核完成执行。

然而，cudaDeviceSynchronize()应谨慎使用，因为它会阻塞主机代码的执行，这可能会导致性能下降。在生产代码中，通常优先使用非阻塞同步函数(如cudaStreamSynchronize())，这样可以在设备执行任务时让主机执行其他任务。

* cudaMemPrefetchAsync 预迁移数据

cudaMemPrefetchAsync用于管理和优化数据在主机和设备之间的迁移，此函数可以预先迁移数据。

cudaMemPrefetchAsync的工作原理主要是基于CUDA内存管理系统和硬件的协作。当调用此函数时，CUDA运行时系统会在后台发起一个数据迁移操作，将数据从源设备迁移到目标设备。

如果目标设备是GPU，系统会将数据从主机内存复制到GPU内存。这个操作通常在CPU调用cudaMemPrefetchAsync之后立即开始，但实际的开始时间取决于系统的调度策略和设备的负载情况。一旦数据迁移到GPU，GPU上的CUDA内核就可以直接访问这些数据，而无需等待从主机内存的迁移。这可以大大减少内存访问的延迟，提高程序的性能。

如果目标设备是CPU，系统会将数据从GPU内存复制回主机内存。这个操作在GPU完成其上的所有先前操作之后才开始，以保证数据的一致性。一旦数据迁回主机，CPU就可以直接访问这些数据，无需等待从GPU内存的迁移。这对于需要在CPU上进一步处理的数据非常有用。

cudaMemPrefetchAsync函数还可以接受一个CUDA流作为参数，用于控制数据迁移的执行顺序。在同一个流中，操作按照它们被提交的顺序执行。这意味着，如果在同一个流中先后调用cudaMemPrefetchAsync和一个CUDA内核，那么CUDA运行时系统会确保数据迁移完成后才开始执行内核。这种机制提供了一种在GPU和CPU之间精细控制数据流的方式。

函数的原型如下：

cudaError\_t cudaMemPrefetchAsync(const void\* devPtr, size\_t count, int dstDevice, cudaStream\_t stream = 0);

其中，

devPtr 是一个指向需要迁移的数据的指针。

count 是需要迁移的数据的字节数。

dstDevice 是目标设备。例如，可以设置为特定的GPU设备ID，或者设置为cudaCpuDeviceId表示CPU。

stream 是一个可选参数，表示CUDA流，用于异步操作。

通过使用cudaMemPrefetchAsync，可以控制何时以及如何将数据迁移至GPU或CPU，从而优化程序性能，减少内存访问的延迟。

* CUDA Stream

在CUDA编程模型中，Stream可以被看作是设备上执行的一系列命令的队列。这些命令可以包括内核的执行、内存传输等。在同一个stream中，这些命令是按照它们入队列的顺序来执行的。而不同的stream之间则可以并发执行，这使得我们可以通过合理地使用多个stream来提高程序的并行度，从而提高程序的执行效率。

在CUDA中，我们可以通过cudaStreamCreate函数来创建一个新的stream。以下是一个简单的例子：

cudaStream\_t stream;

cudaStreamCreate(&stream);

2

首先定义了一个cudaStream\_t类型的变量stream，然后调用了cudaStreamCreate函数来创建一个新的stream，这个新创建的stream的句柄被存储在stream变量中。

我们可以在调用内核或者内存传输函数时指定stream参数，例如：

myKernel<<<gridDim, blockDim, 0, stream>>>(...);

cudaMemcpyAsync(dst, src, count, cudaMemcpyHostToDevice, stream);

在这个例子中，我们在调用myKernel内核和cudaMemcpyAsync函数时分别指定了stream参数，这意味着这两个操作将会在我们创建的stream中被执行。

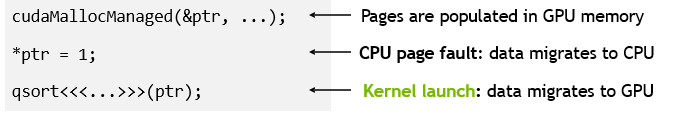
在使用完stream后，我们需要调用cudaStreamDestroy函数来释放stream占用的资源，例如：

cudaStreamDestroy(stream);// 销毁流

## nvidia gpu中的Unified Memory

### Pre-Pascal 架构上的 Unified Memory

以一个例子来说明在Kepaler/Ma

xwell架构上的UM的工作原理.

第一句代码：申请GPU显存，此时页面会在GPU上创建。

第二句代码：CPU访问内存，数据将会migrate 到CPU中。

第三句代码：GPU内核访问，数据会migrate到GPU中，并且只有在内核启动时会migrate一次。

特点：

由于页面会在内存被访问前就创建了，因此不能够超额分配（oversubscribe）。

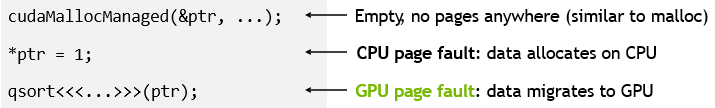
由于只有在GPU内核访问时页面才会被迁移到GPU，就会存在一次性迁移问题【这就意味只在每一次内核启动前都将花费大量时间migration上】，因此不存在按需迁移（migrate on-demand）。【本质上是由于pre-pascal unified memory不能支持页错误处理机制】

### Pascal/Valta 架构上的 Unified Memory

ON-DEMAND PAGE MIGRATION

硬件上引入PAGE MIGRATION ENGINE，支持Virtual Memory Demand Paging，其实就是gpu硬件层面支持页面错误处理机制【handle thousands of simultaneous page faults】。因为没有该特性的话，应用要提前将数据都加载到GPU显存中，这样势必会带来很大的开销。

再以一个例子来说明：



第一句代码：申请GPU显存，此时在GPU上不会创建页面。

第二句代码：CPU访问内存，出现也错误，数据将会在CPU中分配。

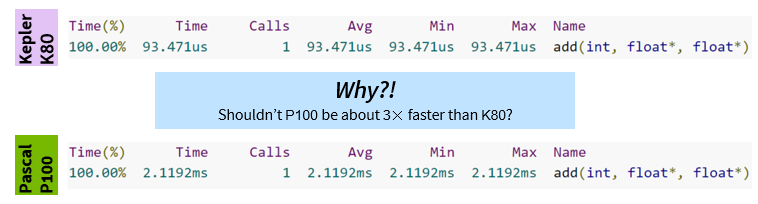
第三句代码：GPU内核访问，出现页错误，数据会migrate到GPU中。

特点：

如果系统不支持UVA，则系统会向CPU发送一个中断。

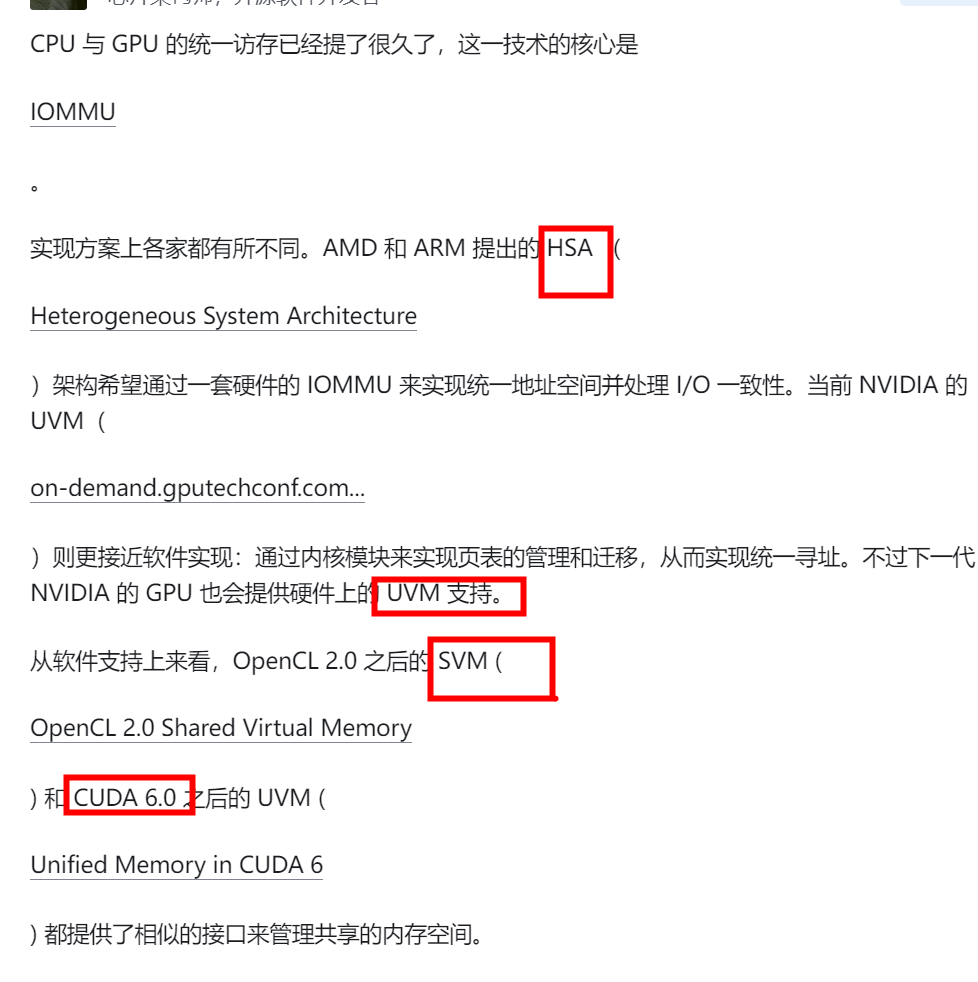
Unified Memory driver会决定会映射还是迁移数据。

实际使用怎么样呢？

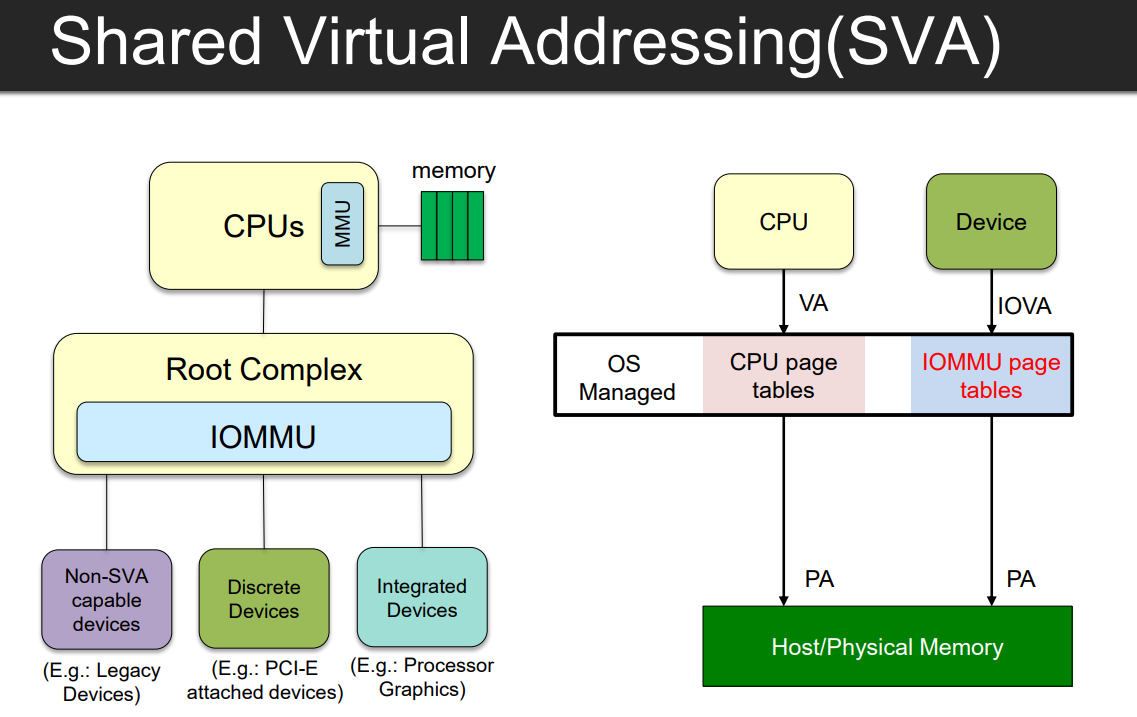
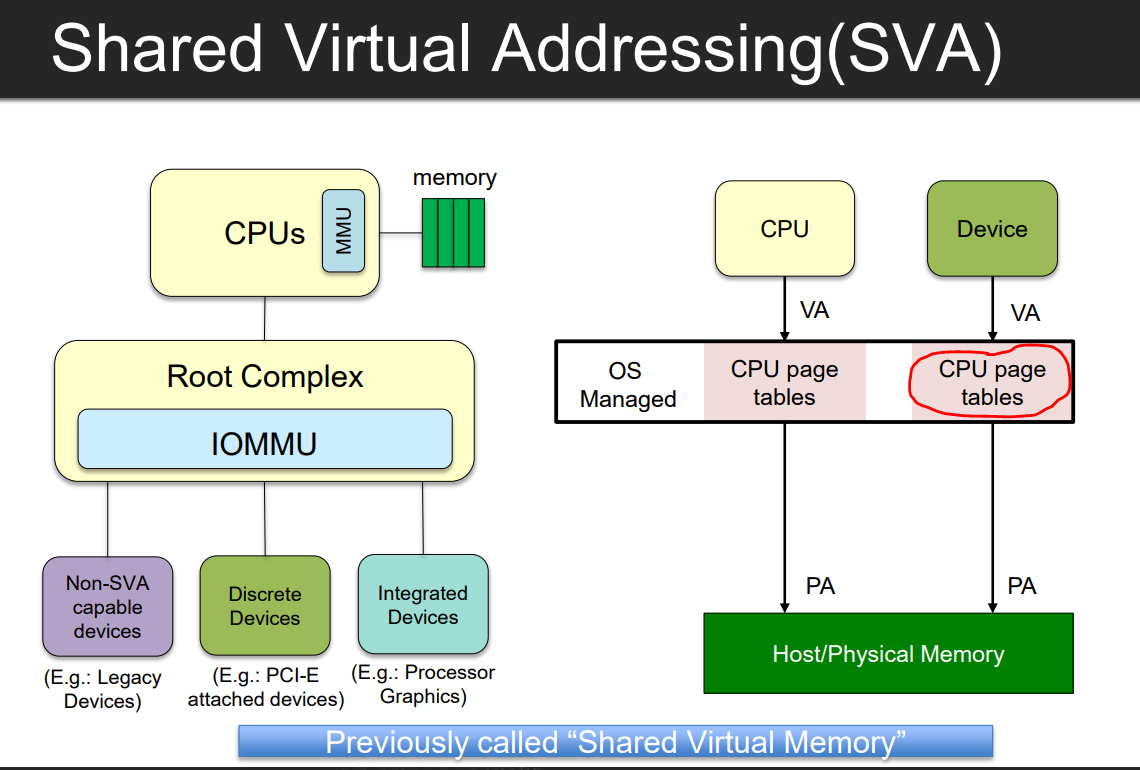


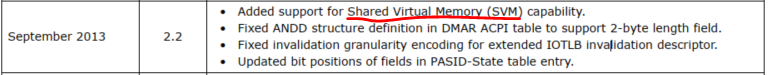
结果很让我们失望，基于P100的结果竟然比K80还慢。What’s up ???

## 基于IOMMU的CPU 与 GPU 的统一访存



### 1. Definition

Shared Virtual Addressing (SVA) allows the processor and device to use the same virtual addresses. SVA is what PCIe calls Shared Virtual Memory (SVM).  
[](https://liujunming.top/images/2022/03/23.PNG)  
[](https://liujunming.top/images/2022/03/24.PNG)

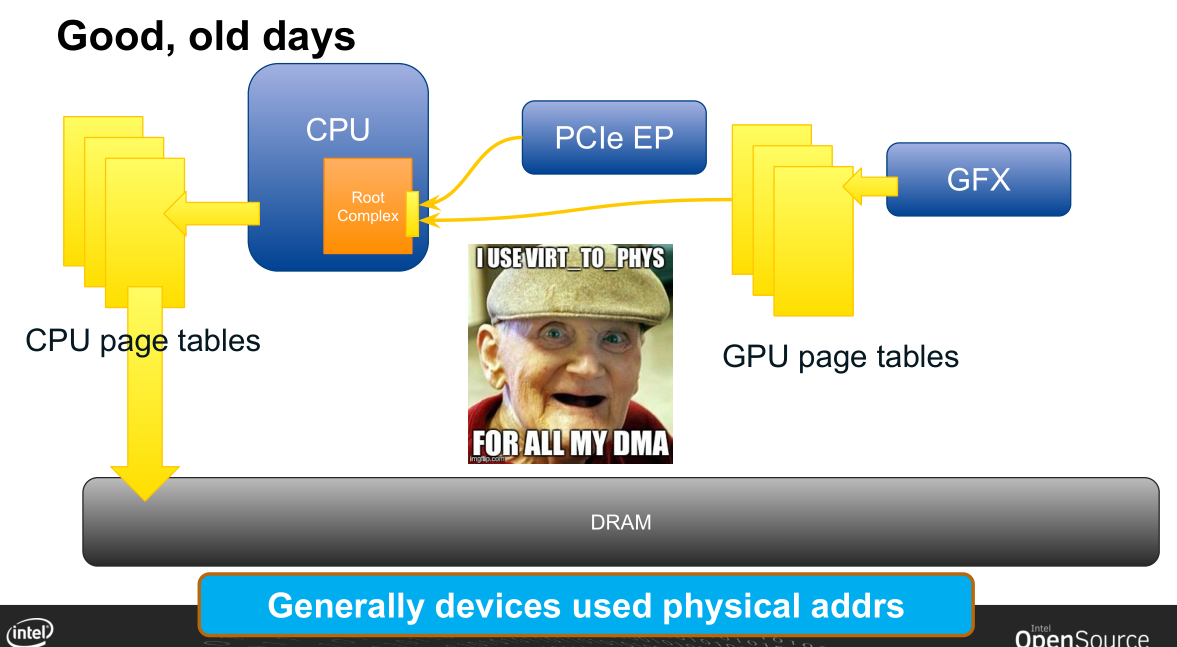
VT-d SVM: Extends root complex IOMMU to comprehend x86 page table formats  
[](https://liujunming.top/images/2022/03/31.PNG)

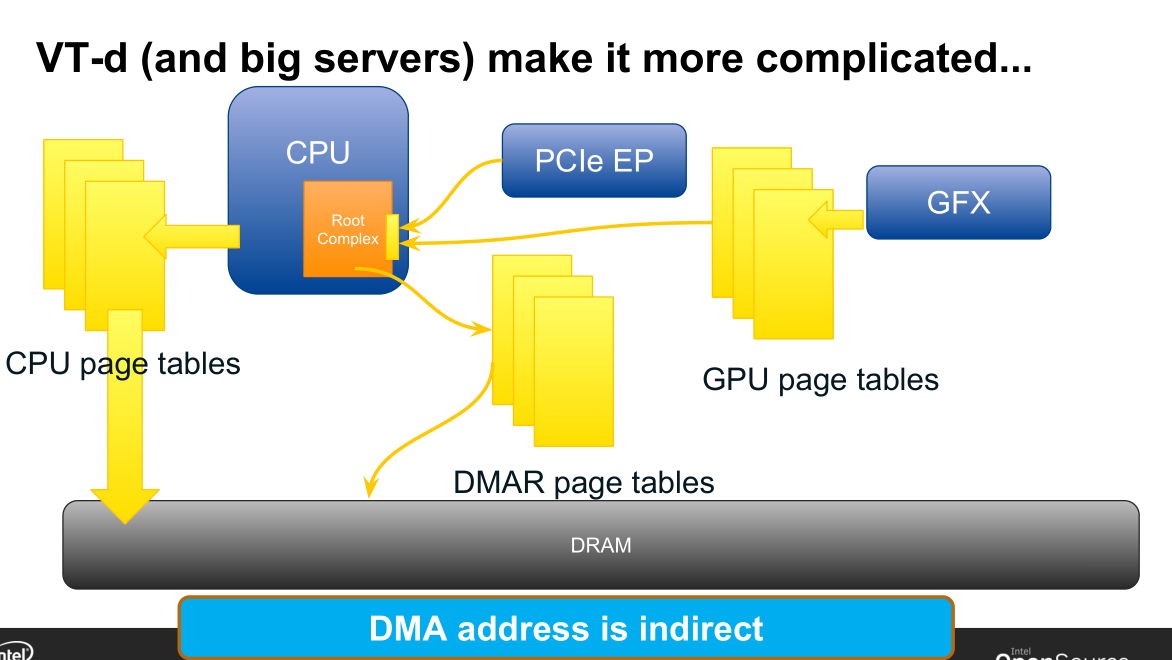
### 2. History

共享虚拟内存（SVM）技术最初是为了解决在GPU场景下，设备（GPU）和host（CPU）之间共享内存的。目的是在设备GPU和CPU之间可以直接传递指针（地址），为了让设备可以直接使用进程空间的地址，简化编程模型。我们知道通常host侧采用的地址是主机的虚拟地址（VA），而设备侧通常使用的是物理地址（PA）或IOVA。

如下图，传统内存访问的三个途径：

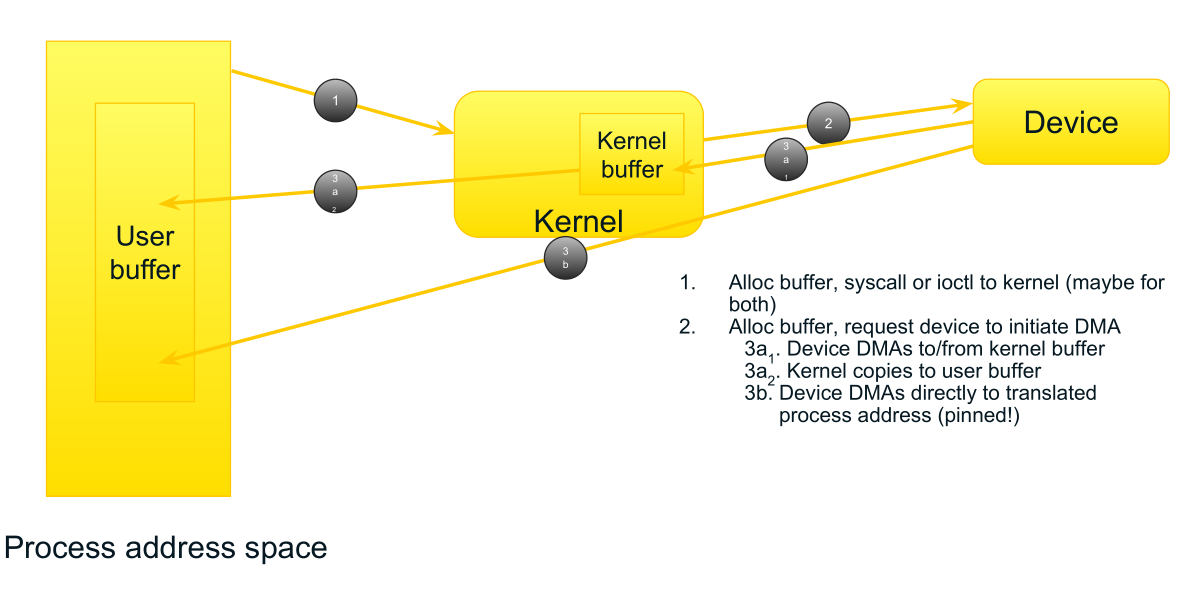
* CPU访存通过MMU（CPU的页表）将VA转为PA访问物理地址
* GPU访存通过GPU的页表访问物理地址
* PCIe设备直接使用物理地址PA访问

[](https://liujunming.top/images/2022/03/25.png)

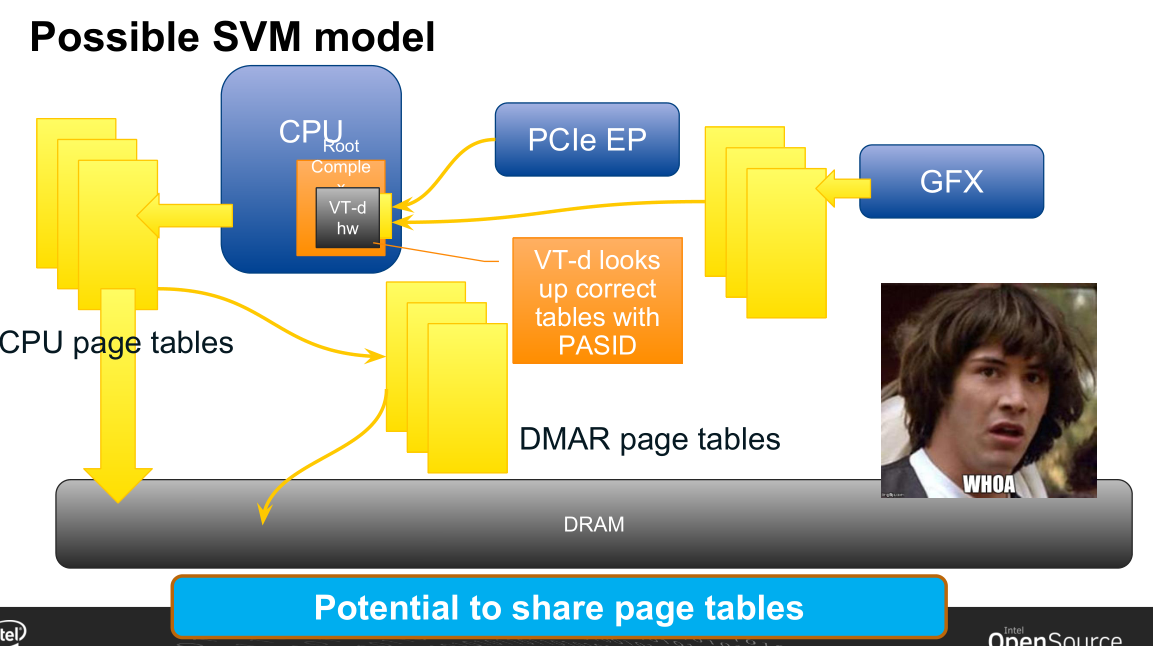
在引入了iommu（VT-d）后，如下图，PCIe设备也可以使用虚拟地址（IOVA）来访存了，也有设备自己对应的页表（iommu页表）完成IOVA到物理地址（PA）的映射。  
[](https://liujunming.top/images/2022/03/26.png)

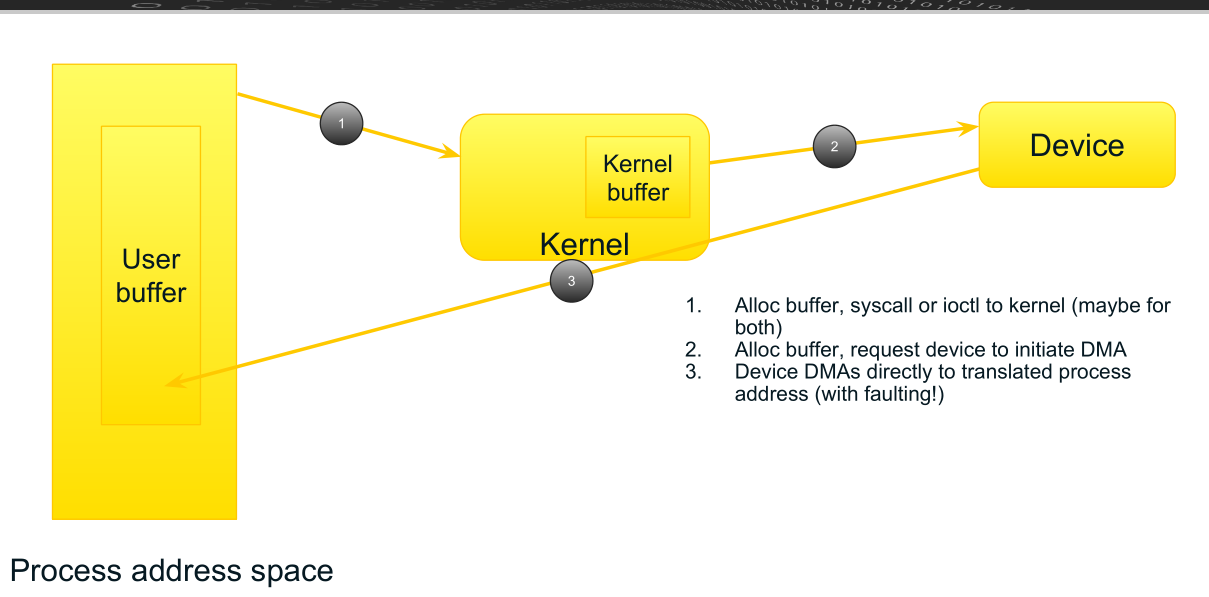
这种情况下CPU进程和设备的内存通信一般采用如下流程：

1. CPU进程分配一块内存，并采用系统调用syscall或ioctl请求内核准备接收操作
2. 内核初始化设备的DMA操作，这里面有两种情况：一种是内核重新分配一块内核空间的内存，将其物理地址传递给设备进行DMA，另一种是如果应用程序将用户空间的内存pin住（这段内存的虚拟地址空间和物理地址空间不会发生变化）则可直接将用户空间的buffer地址传递给设备进行DMA
3. 设备将数据DMA到内存，对应上面这里也有两种情况，如果是内核设置的内核buffer的地址，则设备会先将数据DMA到内核buffer，再由内核将数据由内核空间拷贝到用户空间的buffer（我们通常使用内核协议栈进行收发报文的应用程序就是这种），另一种如果用户空间直接将内存pin住，则设备直接将数据DMA到应用程序的buffer（我们采用DPDK收发报文就是这种）

[](https://liujunming.top/images/2022/03/27.png)

### 3. 引入SVM后的变化

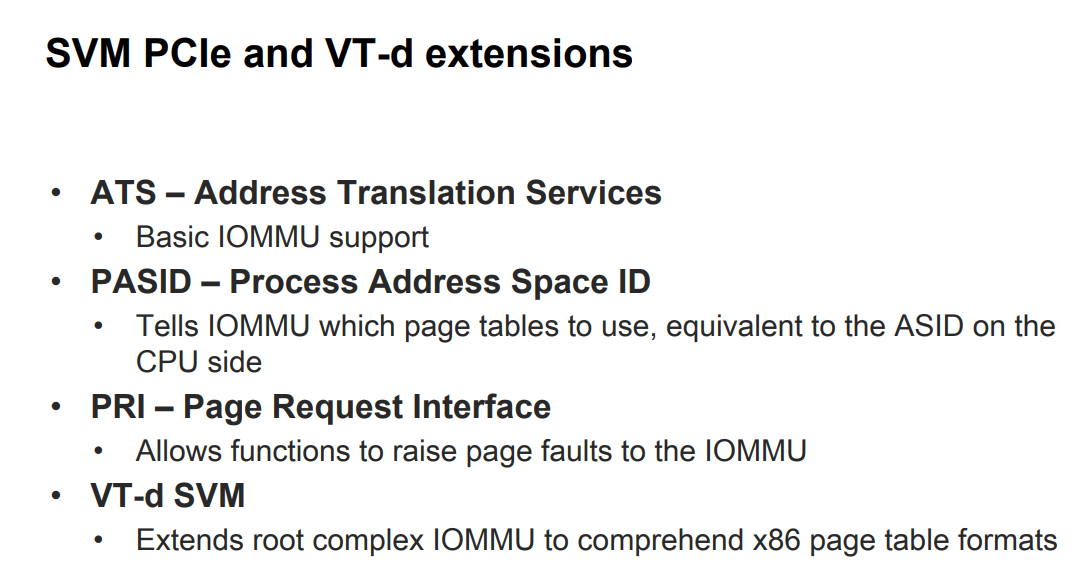
下面我们看引入SVM后的效果，最大的区别是设备访问地址在经过iommu的DMAR转换时会参考引用CPU的mmu页表，在地址缺页时同样会产生缺页中断。为什么要这样设计呢？因为要想设备直接使用进程空间的虚拟地址可能采用的有两种方法。一种是把整个进程地址空间全部pin住，但这点一般是不现实的，除非类似DPDK应用程序全部采用静态内存，否则如果进程动态分配一个内存，那么这个地址设备是不感知的。另一种方法就是采用动态映射，就像进程访问虚拟地址一样，mmu发现缺页就会动态映射，所以从设备发来的地址请求也会经过CPU缺页处理，并将映射关系同步到iommu的页表中。  
[](https://liujunming.top/images/2022/03/28.png)

有了以上的流程，CPU和设备的内存交互流程就变成了如下图所示。主要是第三步的变化，设备直接将数据DMA到进程空间的地址，并且不需要进程pin内存，**而是通过page fault触发缺页处理进行映射。**[](https://liujunming.top/images/2022/03/29.png)

### 4. 支持SVM的条件

那么支持SVM需要软硬件具备什么条件呢。首先是设备角度：

1. 要支持[PASID](https://liujunming.top/2021/11/09/Notes about-PASID-Process-Address-Space-ID/)，因为一个设备会被多个进程访问，对应多个设备DMAR页表，需要通过PASID来区分采用哪个页表
2. 支持[dma page fault](https://liujunming.top/2022/03/30/Notes-about-PCIe-Page-Request-Interface/)处理，当访问的虚拟地址引发缺页时能够等待或重试
3. 从驱动角度来说，操作设备的API需要通过PASID来区分不同进程

[](https://liujunming.top/images/2022/03/30.PNG)

参考资料:

1. [Shared Virtual Addressing (SVA) with ENQCMD](https://www.kernel.org/doc/html/latest/x86/sva.html" \t "https://liujunming.top/2022/03/30/Introduction-to-Shared-Virtual-Memory/_blank)
2. [SVM on Intel Graphics](https://archive.fosdem.org/2016/schedule/event/intel_svm/attachments/slides/1269/export/events/attachments/intel_svm/slides/1269/FOSDEM_2016___SVM_on_Intel_Graphics.pdf" \t "https://liujunming.top/2022/03/30/Introduction-to-Shared-Virtual-Memory/_blank)
3. [Shared Virtual Addressing in KVM kvm forum 2018](https://static.sched.com/hosted_files/kvmforum2018/52/kvm-forum-vSVA-yliu-jpan-jean-eric.pdf" \t "https://liujunming.top/2022/03/30/Introduction-to-Shared-Virtual-Memory/_blank)
4. [Shared Virtual Addressing in KVM](https://events19.linuxfoundation.cn/wp-content/uploads/2017/11/Shared-Virtual-Memory-in-KVM_Yi-Liu.pdf" \t "https://liujunming.top/2022/03/30/Introduction-to-Shared-Virtual-Memory/_blank)

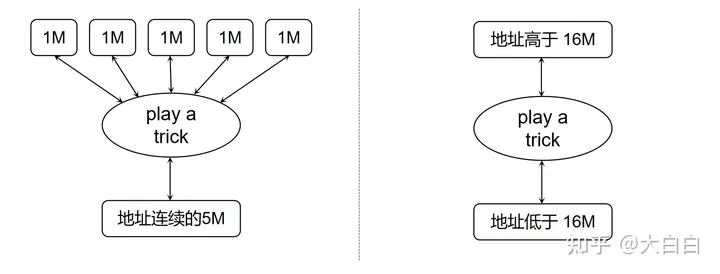
## SMMU 和 SVA 的由来

最早的时候，设备只能使用物理地址（即 PA）访问内存，用着用着我们就发现直接使用 PA 的一些问题，比如：

* 内存碎片问题：因为设备做 DMA 需要一段地址连续的内存空间，假设现在需要 5M 的内存，但是现在虽然还有不少空闲的内存，但是每段连续空闲内存都只有 1M（只是为了举例，不用纠结具体的数值），那这个内存分配就失败了（实际上操作系统还有一些其它的解决方案，比如页面迁移，但这个方案不在本文讨论的范围内，所以先不管这些方案）。
* 设备地址空间能力问题：我们首先要知道一个概念，不是有多大的内存，设备就能访问多大内存的。比如，早期的 ISA 总线上的设备只有 24 根地址线，所以最多只能访问 16M 地址空间，那么内存再大也没有用，只要地址空间超过 16M 就访问不到了。假设现在低 16M 的内存都用完了，即使高于 16M 的内存空闲再多，也不能分配给设备使用。

看上面两个场景，明明有空闲的内存却用不上，这真是急死人呀！

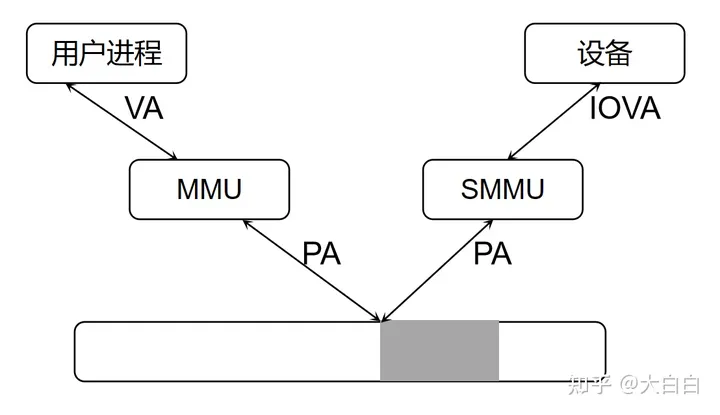
这两个场景都有一个共同特点，就是虽然有空闲的内存，但是空闲内存的地址不符合设备的要求。于是我们很容易想到一个通用的解决思路：我们 play a trick 骗设备一下，我们不要把真正的内存地址告诉设备，我们做个转换，设备需要什么样的地址，我们就告诉它什么样的地址，当设备用这个假地址来访问内存时，我们再把这个假地址翻译成真正的内存地址。如下图所示：



我们需要一个东东来完成这个骗局（就是上图中 “play a trick” 那个组件）。有人可能已经反应过来了，这个东东不就是 MMU 吗？对，这东东跟 MMU 干的活差不多，不过它跟 MMU 还是有点区别的。MMU 给 CPU 做地址翻译，这个东东给设备做地址翻译，它们的基本思路是一样的，但是实现上还是有些区别，为了避免跑题，这里先不展开分析它们的区别，先知道它们俩并不是完全一样就行。那我们得给这个东东取个名字呀，于是 SMMU 就被发明出来了。  
（说明：ARM 叫它 SMMU，intel 叫它 IOMMU。SMMU 和 IOMMU 指的是同一个东西，不同公司叫法不同而已。我们现在看到 Linux kernel 中对应的驱动模块叫 IOMMU（源码路径：./driver/iommu），只是因为 intel 先上传了代码，抢先占了名字而已。因为我是做 ARM 驱动开发的，所以默认情况下，我就跟随 ARM 的叫法。）

有了 SMMU 以后，前面两种场景都不是问题了，简直爽飞了！但还是有别的场景搞不定，比如：

我们假设现在有一个用户进程分配了一块内存空间，并让设备 DMA 把数据写到这块内存中。在有 SMMU 的情况下，这个过程可以用下图来表示：



* 首先用户进程分配一块内存空间，用户进程看到的是 VA， 该 VA 对应到 PA，然后 SMMU 将此 PA 映射到 IOVA，设备看到的是 IOVA。

那问题来了：

* 用户进程分配内存时，操作系统给进程分一个 VA，此时不一定会把 VA 映射到真正的内存空间上，可能要等到真正用内存的时候才会映射上去，也就是说用户进程拿到一个 VA，此时根本就没有对应的 PA，那谁也没法把这个根本不存在 PA 映射到 IOVA 上嘛！

还有一种情况：

* 在用户进程的生命周期中，操作系统是可以改变 VA 和 PA 的映射关系的，比如迁移页面、交换到SWAP。这边设备还在 DMA 操作原来的 PA 呢，那边操作系统吧唧一下这个 PA 的内存切去做别的事情了。这不就乱套了吗？

解决上述问题最直接的方法是强行让 VA 先跟 PA 映射好，并且把这个关系给 pin 住（即不允许操作系统中途改变这个关系）。这个方法很直接，但对用户进程使用内存的方案限制比较大。

于是有了让设备侧发起缺页的诉求。如果 VA 没有映射到 PA 上，那 IOVA 也先不映射到 PA 上，设备访问 IOVA 时，SMMU 发现没有对应的 PA ，此时再发起缺页补页的请求。但是 SMMU 和 MMU 各有各的页表，如果分别各补各的页，那怎么保证最后 VA 和 IOVA 都是映射到一个 PA 上的呢？  
于是又有了让 SMMU 和 MMU 共页表的诉求。SMMU 和 MMU 共用页表，此时 IOVA == VA。这样只要补页，那 SMMU 和 MMU 都同时补上了，如果页面迁移了，那 SMMU 和 MMU 的页表都会修改，因为本来就是一个页表嘛。  
于是 SVA 就被发明出来了（ARM 叫它 SVA，intel 叫它 SVM，SVA 和 SVM 指的是同一个特性）。

smmuv3使用了STALL特性实现sva。当translation出错时，通过Event类型来判断是否能够进行retry处理，如何符合，则通过mm进行缺页重映射，再发送CMD\_RESUME(retry)，最终完成translation。

## PASID和 SVA

<https://fishthinking.xyz/2024/03/02/pasid-sva/>

设备能通过IOMMU（带着PASID）访问到CPU地址空间还不够。CPU每次给设备下发任务是也需要携带PASID。具体的携带方式是在ENQCMD/ENGCMDS命令发送work descriptor的同时携带PASID到设备。ENQCMD是non-posted写，CPU通过返回值判断写入是否成功（例如队列满了导致写入失败）。

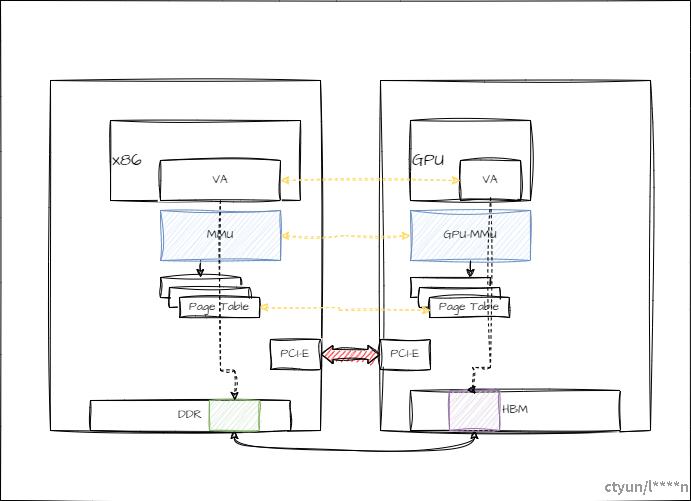
### PRI和stall模式区别

## 英特尔svm（Shared Virtual Memory）

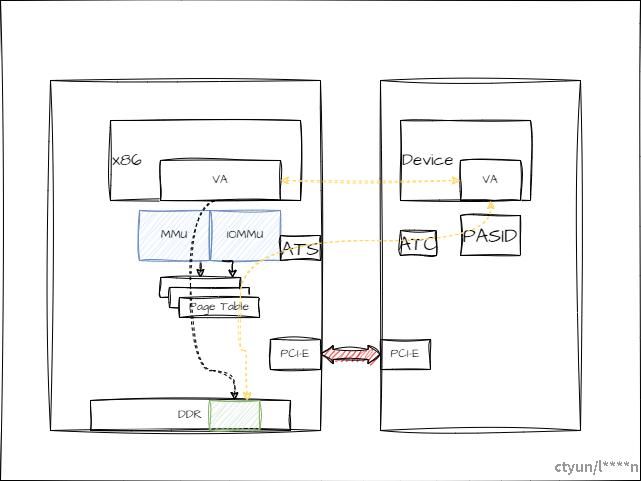
目前市面上有三种设备共享内存地址空间的技术，UVA/SVM和SVA。使用SVA可以让软件设计更加简洁 ，软件不用去管理自己地址空间的iommu mapping，设备也可以访问更复杂的对象，比如链表，树等。  设计模型也更容易实现zero copy，设备也可获得足够的信息而自主地访问系统内存空间。

* 英伟达公司实现的是UVA的技术，GPU内部加入了MMU的功能模块，GPU的MMU模块使用的转换页表是CPU转换页表的镜像。

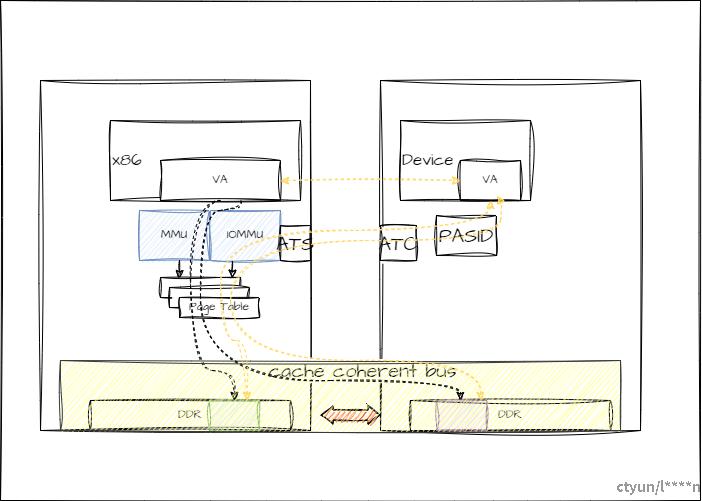
当GPU发现VA没有被记录在自己的MMU中之后，会通过PCIe总线从CPU的MMU页表中同步过来。反过来，CPU也可以从GPU内部同步页表项。



英特尔公司实现的是SVM技术，特点是通过PCI pasid绑定确定的CPU进程, 并且IOMMU访问CPU的MMU页表。当设备发现访问的地址不在设备的address translation cache (ATC) 中时，通过Adress translation service(ATS) 向IOMMU申请对应的转换页表。    
SVM实现了设备访问虚拟地址空间的能力，并且设备不需要实现复杂的MMU功能，但有一定的限制如设备上的内存无法映射到CPU的内存地址空间。



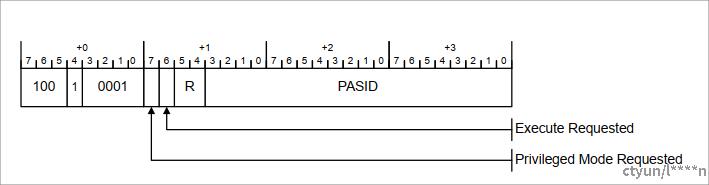
ARM公司是SVA方案，CPU和外设间的连接的是支持cache一致性协议的总线。



### DSA设备支持SVM方式

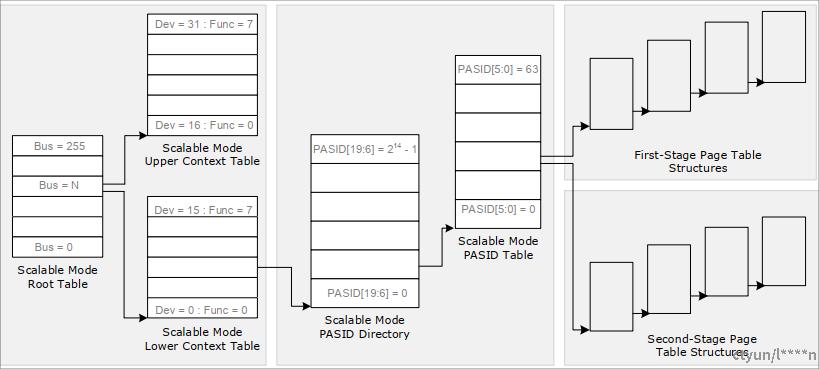
DSA设备支持的SVM模式在linux shared virtual address (SVA) 框架下进行了实现。DSA对SVM的支持是通过设备上的ATC，ATS的功能模块，PASID，page request service (PRS）来共同实现的。

PASID在PCI总线上的占用的是TLP 包头的部分，在PCI总线End-to-end数据包中使用20个bit为来传输PASID，这个值一般是iommu配置设备时指定的。具体格式可以参考PCI-spec6.0中关于PASID TLP的介绍。 PASID的分配和管理部分代码在kernel目录下的drivers/iommu/iommu-sva.c文件中。



PASID在IOMMU上面的支持也叫做scalable-mode，当scalable-mode使能的时候，会在传统的页表前加入PASID的上下文表项。

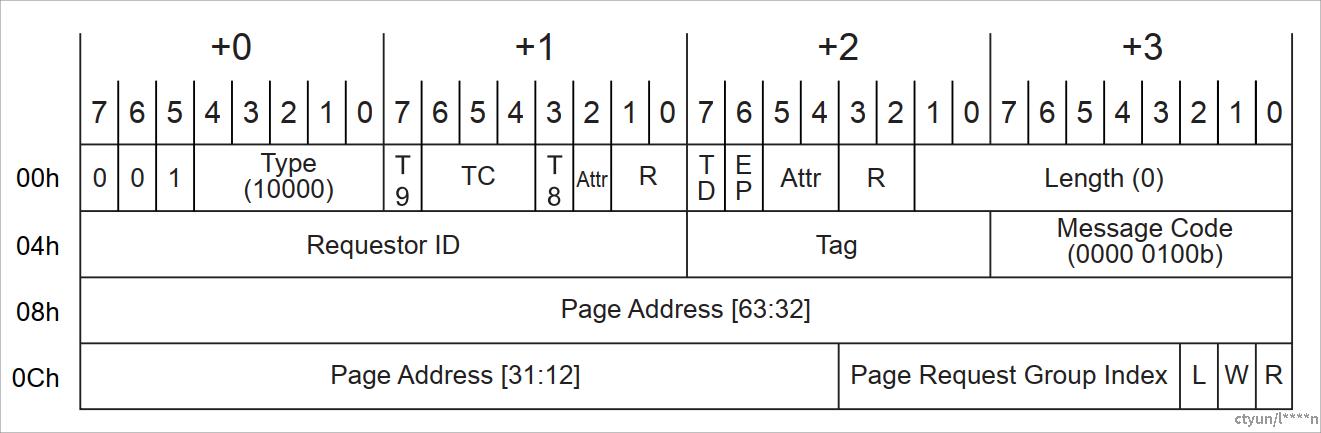
最新的VT-d spec中 Scalable Mode Address Translation 中可以看到每个设备有自己的PASID上下文，最后的PASID表会指向一级和二级页表。



ATS的实现是通过特别类型的TLP包来实现的，具体格式可以参考PCI-spec6.0第十章 Address translation service.

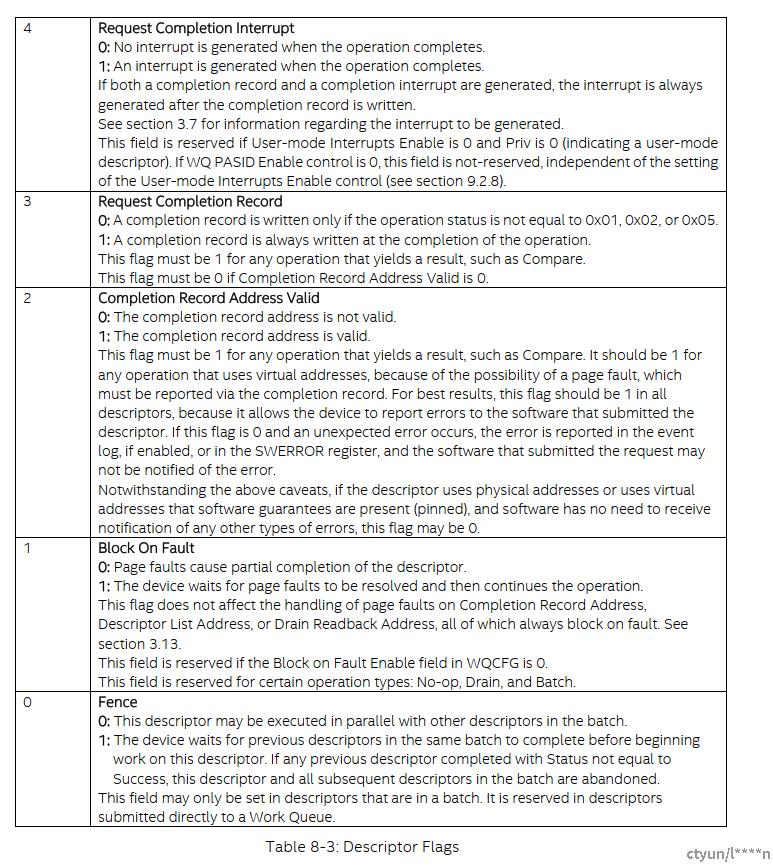


PRS的实现是通过特别类型的TLP包来实现的，具体格式可以参考PCI-spec6.0第十章 page request message相关的介绍。



### DSA与主机共享地址空间

如果是host和DSA共享内存空间，ATC/ATS，PRS两者使能之后就可以实现。 PRS对应的是描述符flag中的block on fault比特位。



DSA与虚拟主机共享地址空间

如果是guest和DSA共享内存空间， 需要平台的scalable-mode是开启的，VMM需要为VM在IOMMU设置PASID。安排到虚机的DSA vDEV也需要支持ATS/PASID/PRS 这些功能。目前DSA vdev功能仍在社区检阅中。

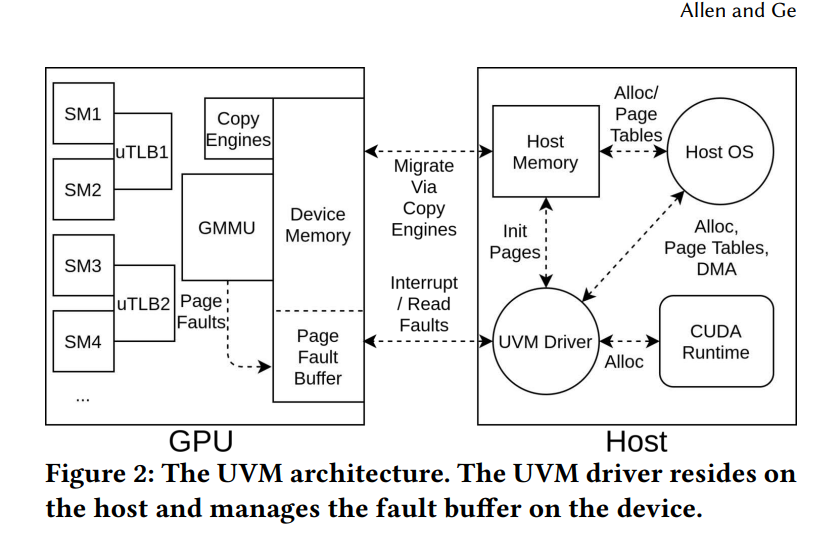
### 缺点

SVM实现了设备访问虚拟地址空间的能力，并且设备不需要实现复杂的MMU功能，但有一定的限制如设备上的内存无法映射到CPU的内存地址空间。

# MMIO与DMA的区别

[Load/Store 和缓存一致性有没有必要？](https://01.me/2023/08/cache-coherency/)

## UVM



UVM架构（如图2所示）是一种客户端-服务器架构，由一个或多个软件客户端（用户级GPU或主机侧程序）和一个服务器（主机驱动程序）组成，该服务器为所有客户端处理缺页请求。主机上的UVM驱动程序是一个开源驱动程序，依赖于nvidia驱动程序/资源管理器和主机OS进行内存管理。该驱动程序充当运行时故障处理引擎和托管内存分配的内存管理器。

GPU上的任何活动线程都可能触发缺页。缺页请求由硬件线程的相应µTLB识别和处理。线程将此情况视为任何其他未完成的内存请求，并可能继续执行不受内存依赖关系阻塞的指令。同时，缺页请求会传播到GPU内存管理单元（GMMU），GMMU会向主机发送硬件中断。GMMU将相应的缺页信息写入GPU故障缓冲区。故障缓冲区充当一个循环数组，由UVM驱动程序配置和管理。nvidia-uvm驱动程序获取缺页信息，将其缓存到主机上，并通过页面处理和迁移来处理缺页请求。

GPU通过GPU命令推送缓冲区向主机提供两个功能：主机到GPU内存复制和故障重放。作为故障处理过程的一部分，驱动程序指示GPU将页面复制到其内存中，通常使用高性能硬件“复制引擎”。一旦GPU的页表被更新并且数据被成功迁移，驱动程序就会发出故障重放，这将清除µTLB的等待状态，导致它们“重放”之前的未命中。

### GMEM--华为NPU统一内存管理UVM/SVM方案

<https://zhuanlan.zhihu.com/p/670474027>

[GPU的UM实现](https://zhuanlan.zhihu.com/p/679635240)

### CUDA中的UM机制与GDR实现

Unified Memory：

CUDA 6.0开始支持Unified Memory，在CUDA 6.0之前，需要手动的在Host和GPU之间分配内存，并在两者之间不断地进行拷贝（cudaMemcpy），即需要自己进行CPU和GPU之间地内存管理。 采用UM后，从程序员的视角，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。

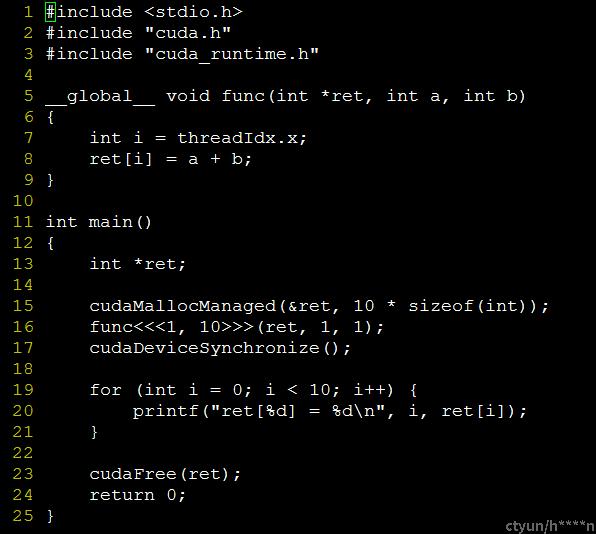
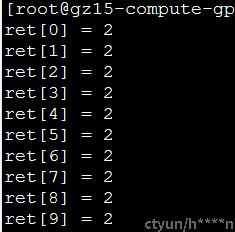
使用UM机制，不能再用cudaMalloc分配GPU memory上的内存，应该使用cudaMallocManaged

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

UM机制极大地减少了代码量，可以很大程度上减少程序员的工作量。在UM出现之前，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，会使得内存管理变得很复杂。

* 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

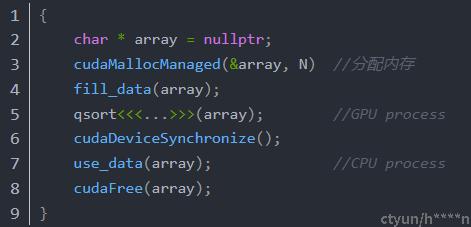
一个UM的示例程序，直接使用cudaMallocManaged分配GPU内存，指针ret指向GPU的内存，在CPU端同样可以直接访问ret指针。

UM机制使得程序员不用频繁地使用cudaMemcpy在GPU和CPU之间来回拷贝数据。

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。

* 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，CPU和GPU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，我们来看一下这个例子的具体流程：

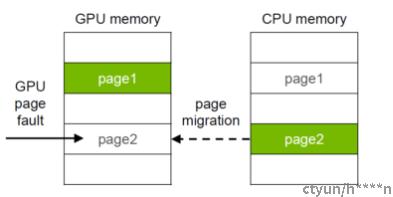
1 首先使用cudaMallocManaged分配GPU内存

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。

３qsort<<<...>>>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。



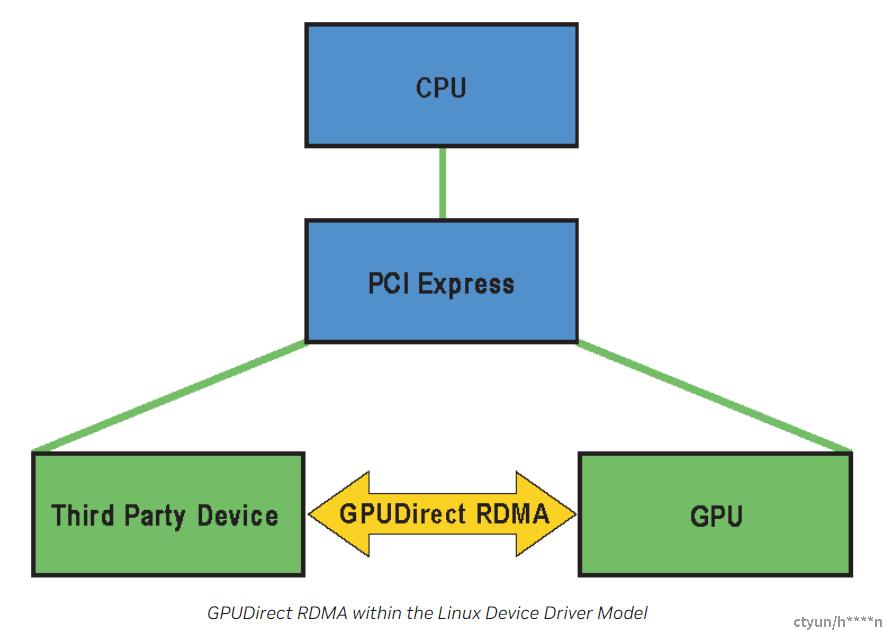
由此我们可以看到，其实UM机制中，数据就是来回在CPU和GPU的memory中来回migrate，只是程序员在上层并不感知。

* 为什么不采用zero-copy机制？

如果数据内容只有一份，那么就无法利用data locality，这样会影响性能。

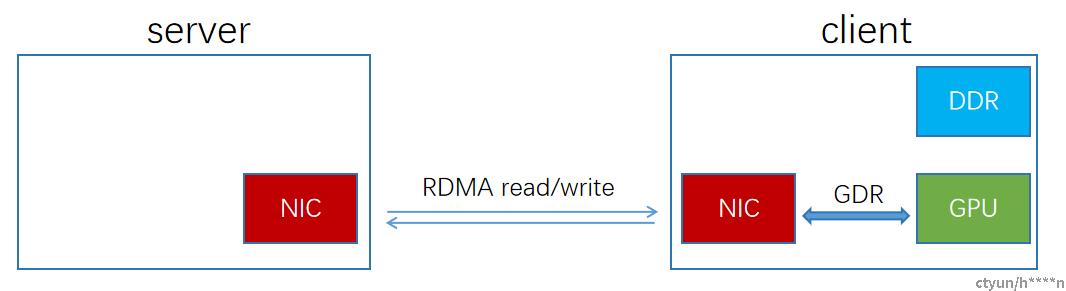
#### GDR(GPU Direct RDMA)实现：

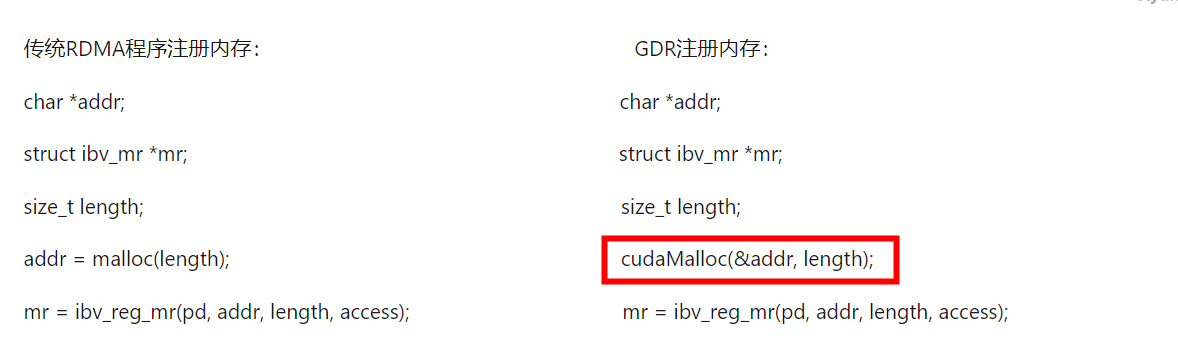
GDR，即GPU direct RDMA，可以在GPU中直接分配memory。然后直接在GPU和device上进行数据传输。 Device可以是NIC，storage adapters，video acquisition devices这些PCIE设备。



想要使用GDR功能，device和GPU需要在同一个RC下。

GDR Example：





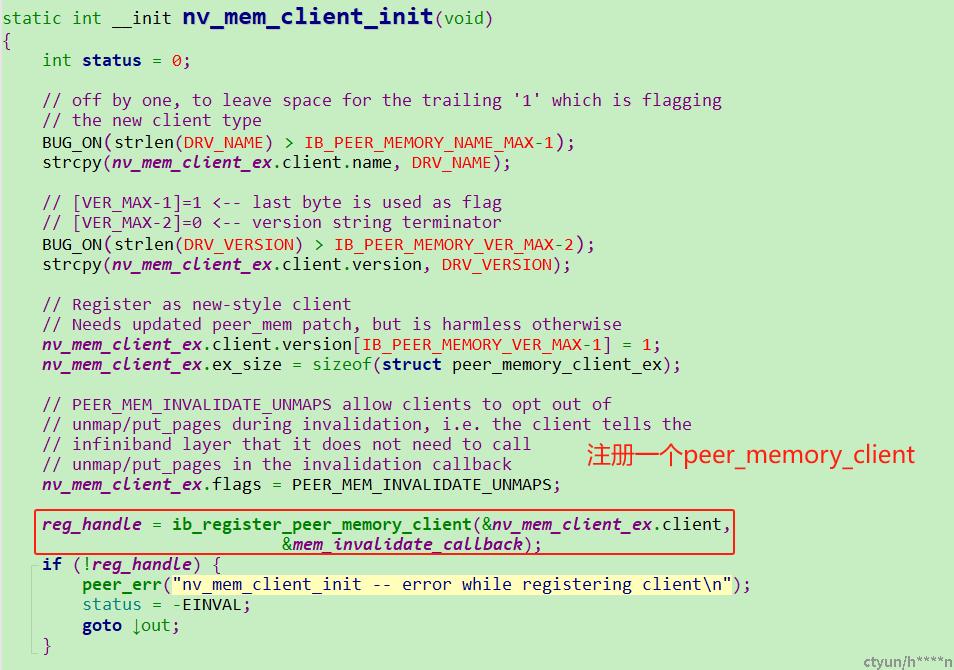
从上面的程序例子可以看到，其实GDR和传统的RDMA程序几乎没有什么不同，最主要的不同就在于，传统的RDMA程序采用malloc分配CPU memory上面的内存，而GDR程序中采用cudaMalloc分配位于GPU memory上面的内存。

* 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。

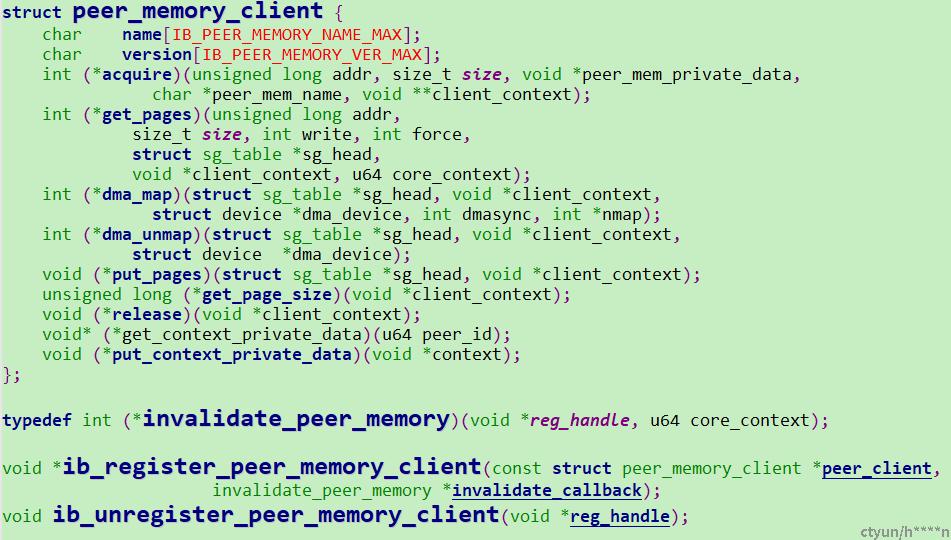
GDR的实现 ：

B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。



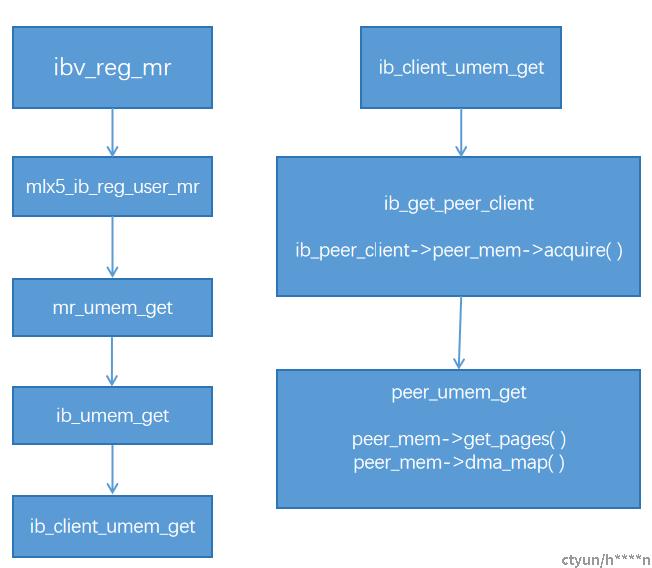
insmod nv\_peer\_mem.ko时会执行nv\_mem\_client\_init函数，其中就调用了ib\_register\_peer\_memory\_client向ib\_core注册了一个peer\_memory\_client 。

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：

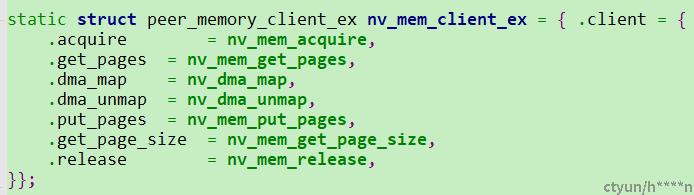


peer\_memory\_client在peer\_mem.h定义，位于mlnx-ofed-kernel-4.7\include\rdma

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。

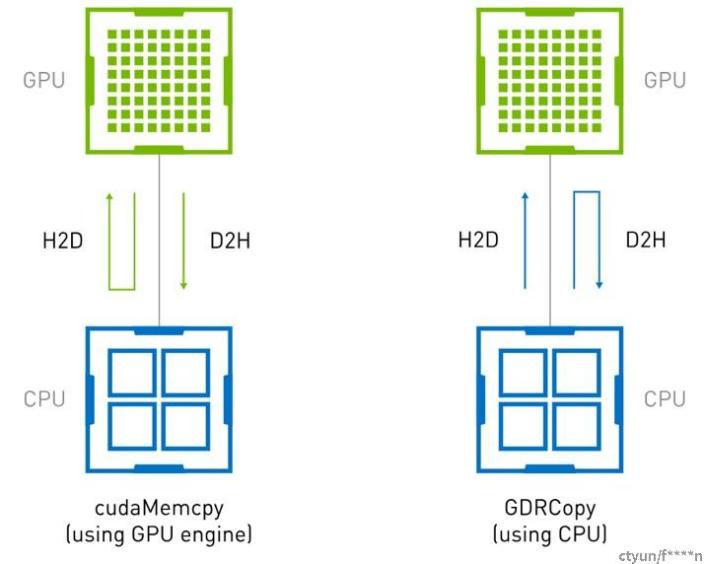
nv\_dma\_map中调用nvidia\_p2p\_dma\_map\_pages获取dma\_address，page\_size等，并填入sg\_table中(用于scatter gather DMA)

#### cudaDeviceSynchronize

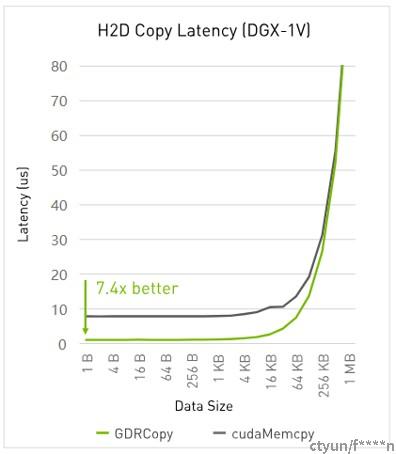
  cudaDeviceSynchronize() ：该方法将停止CPU端线程的执行，直到GPU端完成之前CUDA的任务，包括kernel函数、数据拷贝等。

# GDRCopy

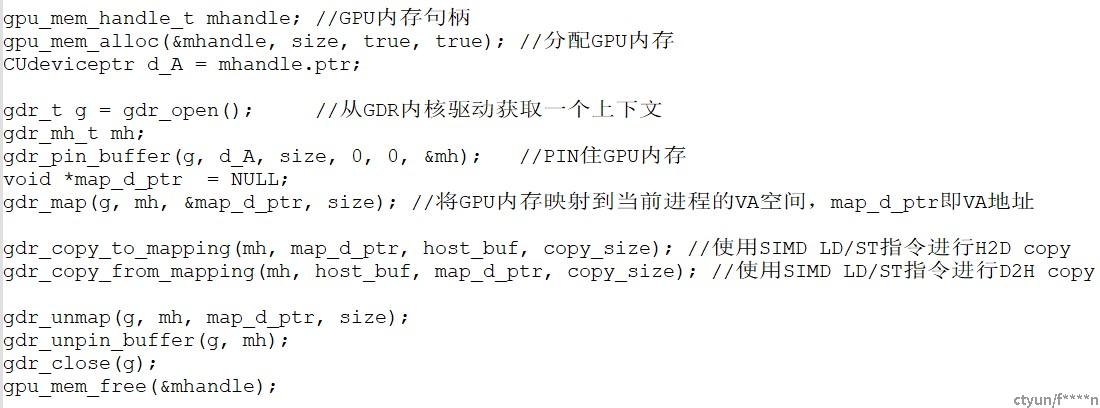
1，GDRCopy的原理



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。

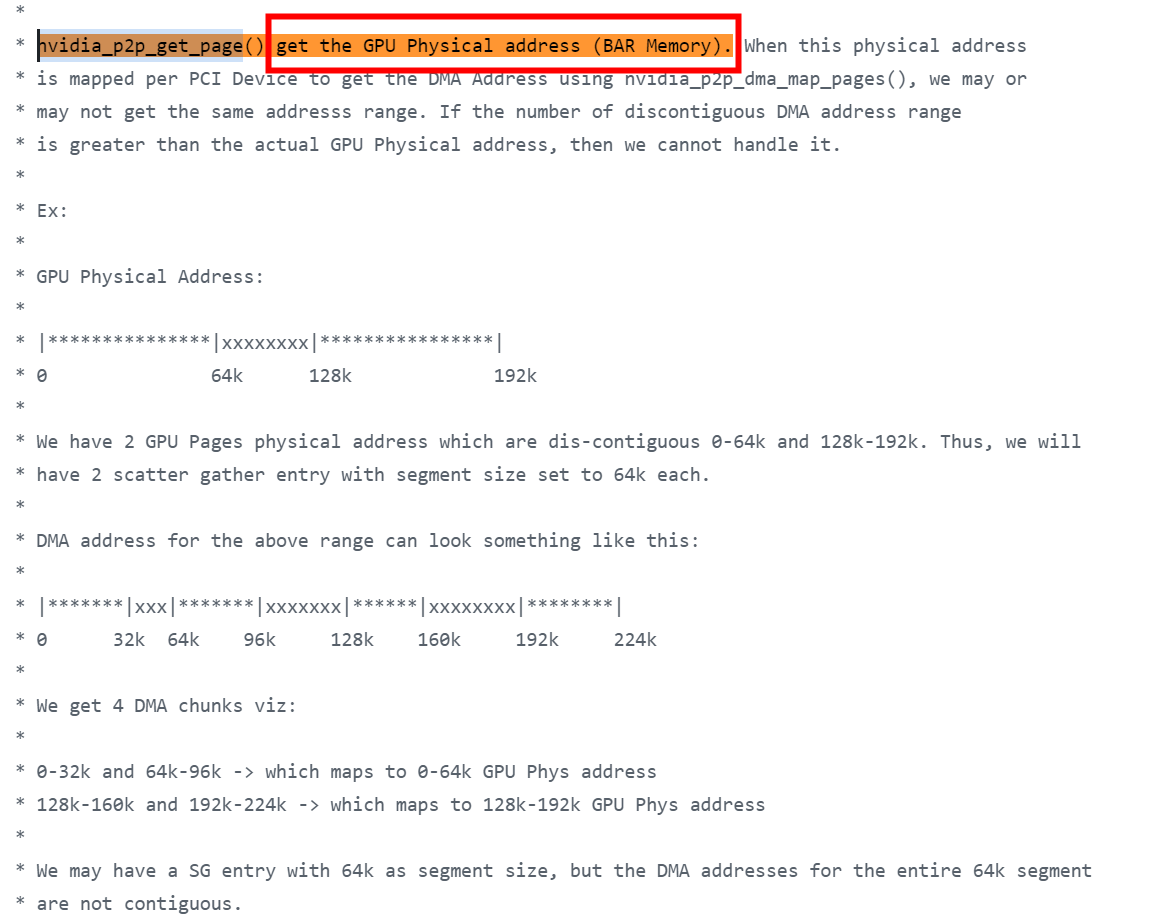
                                    

2，example



# API

## nvidia\_p2p\_get\_page



Caveats as of CUDA 6.0:

CUDA Unified Memory is not explicitly supported in combination with GPUDirect RDMA. While the page table returned by nvidia\_p2p\_get\_pages() is valid for managed memory buffers and provides a mapping of GPU memory at any given moment in time, the GPU device copy of that memory may be incoherent with the writable copy of the page which is not on the GPU. Using the page table in this circumstance may result in accessing stale data, or data loss, because of a DMA write access to device memory that is subsequently overwritten by the Unified Memory run-time. cuPointerGetAttribute() may be used to determine if an address is being managed by the Unified Memory runtime.

Every time a device memory region is pinned, new GPU BAR space is allocated unconditionally, even when pinning overlapping or duplicate device memory ranges, i.e. there is no attempt at reusing mappings. This behavior has been changed since CUDA 7.0.

# RDMA

* using memory allocation on GPU device, I have 39Gb/s RoCEv2 throughput.

cuMemAlloc(&d\_A, size);//allocation on device

mr = ibv\_reg\_mr(pd, d\_A,size, IBV\_ACCESS\_LOCAL\_WRITE | IBV\_ACCESS\_REMOTE\_WRITE|);

* using Unified Memory, I have 95Gb/s but no coherency (RDMA data are not updated on GPU, data migration happen only once during the first transfer)

cuMemAllocManaged(&d\_A, size,cudaMemAttachGlobal);

mr = ibv\_reg\_mr(pd, d\_A,size, IBV\_ACCESS\_LOCAL\_WRITE | IBV\_ACCESS\_REMOTE\_WRITE|);

* in this case, if I bind the memory region to GPU with

cuMemAdvise(d\_A, size, CU\_MEM\_ADVISE\_SET\_PREFERRED\_LOCATION, 0)

I have 39Gb/s (consistent with dot one)

* I guess that using Unified Memory requires ON DEMAND PAGING, following the slide below (http://on-demand.gputechconf.com/gtc/2018/presentation/s8474-gpudirect-life-in-the-fast-lane.pdf

2). Doing so, the HCA does not receive the RDMA packets anymore:

cuMemAllocManaged(&d\_A, size,cudaMemAttachGlobal); //Unified Memory

mr = ibv\_reg\_mr(pd, 0, SIZE\_MAX, IBV\_ACCESS\_LOCAL\_WRITE | IBV\_ACCESS\_REMOTE\_WRITE | IBV\_ACCESS\_ON\_DEMAND);