其实呢，对于在虚拟化场景中如何使用 NVMe，介绍起来会很简单，因为使用的方式或者叫使用的角度，只有 4 种：其中 3 种与 qemu 关系紧密的，再加上一种稍微有点特殊的 spdk+qemu。

如果仅用三五句话，就草草了事，未免有骗稿费的嫌疑(就是想骗也骗不成，因为压根就没有)，所以我还是打算围绕着 NVMe 多说两名，为一点儿也不熟悉 NVMe 的同学推开 NVMe 这扇门, 尽管我也仅是知道半点儿而已。
(不像以往的分享，以下相当部分内容，都来自于网上材料的东拼西凑，列位就凑合着点吧)

Let the reading BEGIN :-)

先来几句旁白。老式 sata 物理接口的 SSD 就不说了，目前消费级市场常见的 m.2 接口的 SSD 基本上分 3 种：

* 最便宜的基于 sata 总线，走 AHCI 协议的 SSD，做数据操作时，数据先至 sata 芯片，再到 CPU。一个字：慢慢慢!

* 中等价位的基于 PCIe 总线，走 AHCI 协议的 SSD，做数据操作时，虽然 PCIe 总线与 CPU 直连，但受限于 AHCI 规范的制约，不能充分发挥现代 nand flash 的优势。

* 相对最贵的基于 PCIe 总线，NVMe 的 SSD(也就是走 NVMe 协议), 做数据操作时，由于 PCIe 总线与 CPU 直连和 NVMe 得天独厚的优势，所以速度要快好多。还有其原生的并行性支持，流线型的存储堆栈，不像 AHCI 规范的每个命令都需要读取 4 个不可缓存的寄存器从而导致约 2us 的额外延迟，NVMe 无需读取寄存器就可以发出命令。籍此，相对于传统磁盘，NVMe 可为用户带来超低的 latency 和超高的 IOPS。

孰优孰劣无须多言，以后大家选购桌面级的 SSD 时，就根据自己口袋里的银子说话吧。

接下来，科普下 NVMe 及相关的常识。
我们知道 NVMe SSD,充分发挥了非易失性内存的特点，且连接在 PCIe 总线上，性能出众，目前生产环境中较常见的用法之一是使其在 Ceph 中做 Journal，可极大地提升性能。

为了支持数据中心的网络存储，也就是说数据需要安全、高效、低延迟地在存储阵列与远端应用间通信 ，于是诞生了 NVMe over Fabrics（简称 NVMf 或 NVMe-oF），实现 NVMe 标准在 PCIe 总线外的扩展，它的第一个正式版本的协议于 2016 年 6 月发布, NVMf 的贡献在于提供除 PCIe 外, 访问 NVM 的另一类途径: Fabrics，并且将 fabrics 链路上可较好地控制 latency。NVMf 支持把 NVMe 映射到多个 Fabrics 传输选项，主要包括 fc、rdma(InfiniBand、RoCE v2 和 iWARP)。

NVMf 以 NVMe 为基础，适配 Fabrics 场景，新增或删减了的一些 Command，类似 scsi, 它也同样有 Host，Target 和 Transport 这些概念。
* 发起端称作 Host
* 处理发起端请求的部分称作 Target(连接物理 NVMe 设备)
* Transport 是连接 Host 和 Target 的桥梁，可以是 fc 或 rdma。在 Fabrics 传输过程中，NVMe 命令会被相应的 Transport 封装和解析。

nvme rdma support: since kernel 4.8
nvme fc support: since kernel 4.10

与通过 FC 上承载 SCSI 命令的光纤通道协议 FCP 类似，通过 FC 上承载 NVMe 命令叫做 nvme fc，它们都是 FC-4 link services 的一种实现，但 nvme fc 的表现远远好于 FCP。

举个栗子, 一个典型的 nvme fc 的使用场景：
* 远端的 nvme 存储柜 和 用户端的 nvme fc 适配器都已经连接到光纤交换机上
* 管理员在远端的 nvme 存储柜上配置好 nvme target，暴露出若干的 nvme 存储
* 用户使用 nvme-cli，在 nvme fc host 内核模块的帮助下，发现、连接并使用这些存储

由于 NVMe 从硬件和协议角度带来的性能改进特别明显，以至于整个软件栈也可能需要做出相应改变以匹配这些变化。所以，继开源了针对网络优化的 DPDK 之后，英特尔开源了针对存储优化的 Storage Performance Developmen Kit(存储性能开发工具包，简称 SPDK)，试图将 Linux 用户态存储服务程序与底层硬件设施打通，大幅度缩短 IO 路径。

SPDK 出现的原因：
固态存储媒介正在取代旧的数据中心。这一代闪存存储相对于传统磁盘介质在性能，功耗，盘架密度上都有着巨大优势，这些优势将使闪存存储成为存储市场下一代的霸主。

市售的基于 NVMe 硬盘动辄可达到单盘 GB 级的读写带宽和十万量级的随机 IOPS，为传统磁盘介质如 SATA 固态硬盘的 5~10 倍。然而，由于 Linux 内核驱动实现与调度机制的限制，相对于 NVMe 来说，在整个 IO 事务中消耗的时间百分比就显得太多了。换言之，主流的软件定义存储系统并不能完全释放其性能，存储软件协议栈的性能和效率在存储整体系统中的地位就显得越来越关键了。

为了帮助存储设备代工厂（OEM）和独立软件开发商（ISV）整合固态存储介质，Intel 创造出一系列驱动和一个完备的，端到端参考的存储体系结构，即存储性能开发工具包，简称 SPDK。SPDK 的目标是通过使用 Intel 的网络，处理，存储技术，将固态存储介质出色的功效发挥到极致。

SPDK 带来的超高的性能基于 2 点核心技术：用户态运行和轮询模式驱动(Polled Mode Drivers, 简称 PMD)。

首先，与 DPDK 类似，设备驱动代码也运行在用户态，而不像传统的 drivers 都是运行在内核态。把设备驱动移出内核空间避免了内核上下文切换与中断处理，从而节省了大量的 CPU 负担，允许更多的指令周期用在实际处理数据存储的工作上。

其次，传统的中断式 IO 处理模式，采用的是被动的派发式工作，有 IO 需要处理时就请求一个中断，CPU 收到中断后才进行资源调度来处理 IO。举一个出租车的例子做类比，传统磁盘设备的 IO 任务就像出租车乘客，CPU 资源被调度用来处理 IO 中断就像出租车。当磁盘速度远慢于 CPU 时，CPU 中断处理资源充沛，中断机制是能对这些 IO 任务应对自如的。这就好比是非高峰时段，出租车供大于求，路上总是有空车在扫马路，乘客随时都能叫到车。然而，在高峰时段，比如周五傍晚在闹市区叫车（不用滴滴或者专车），常常是看到一辆车溜溜的近前来，而后却发现后座已经有乘客了。需要等待多久，往往是不可预知的。相信你一定见过在路边滞留，招手拦车的人群。同样，当硬盘速度上千倍的提高后，将随之产生大量 IO 中断，Linux 内核的中断驱动式 IO 处理就显得效率不高了。
操作系统的世界里，除了中断式 IO 处理的方式（即上面提到的被动的派发式工作），还有一种 IO 处理方式叫做定点轮询（polling）。还是用出租车的例子，试想在机场外出租车排队接客是怎么工作的——有一个或者多个专门的出租车道，排着一队队等候的出租车，当乘客从航站楼中一涌而出时，一辆辆出租车能够用少于十来秒的时间高效的接走一位乘客，后面的车紧接着跟上处理下一位客人。
PMD 就是按照类似的机制工作的，SPDK 中其他所有的组件也是按照这个理念设计的。专门的计算资源（特定的 CPU 核）用来主导存储设备的轮询式处理——就像专门的出租车道和车流用来处理乘客任务，数据包和块得到迅速派发，等待时间最小化，从而达到低延时、更一致的延时（抖动变少）、更好的吞吐量的效果。

那么，轮询模式驱动是否在所有的情况下都是最高效的处理 IO 的方式呢？答案是“也不尽然”。设想一下，如果航站楼里没有什么旅客出来，乘车的人稀稀拉拉的时候，我们可以看到出租车候车区等候派工的车辆长龙，这些等待的车子完全可以到市区去扫活儿，做些更有意义的事情。同样的道理，对于低速的 SATA HDD，PMD 的处理机制不但给 IO 性能带来的提升不明显，反而浪费了 CPU 资源。

其实在网络技术里面，也经历过类似的发展。从最开始的 Linux 内核 TCP/IP 驱动，到后来的 InfiniBand，再到 DPDK。在追求性能的目标下把尽可能多的工作放到用户态进行，获得了不错的效果。目前的存储介质虽然比不上网络的带宽和延迟，但是已经大大缩小了差距。SPDK 就是借助 DPDK 的设计思想推出的，从老大哥 Intel 的文档看，SPDK 的目标定位在与用户态网络结合，提供从网络到存储的融合框架，整个 I/O 路径完全绕过 Linux Kernel 的处理，交由用户程序控制。

但是，SPDK 使用 Pool 的方式来提取 Completion Command，这对于有大量 I/O 的应用很有利，但是对于 I/O 量小的应用来说，CPU 在 Pool 上会消耗大量的等待时间，此时使用内核驱动不失为一种更好的方法。

总的来说，SPDK 适合那些在高密度 I/O 的情况下最大化性能开发新的应用。而 SPDK 对此给出了让人兴奋的回报。从 Intel 分享的资料看，在多 NVMe 设备下，SPDK 在单个 CPU 上提供远超于内核驱动的性能。

终于该说主题了，如何在虚拟化场景中使用 NVMe 呢？

先声明啊，所有例子，仅适用于 NVMe，完全不涉及 NVMf，家境贫寒，无钱购置，只能看看，纸上谈兵。

1. vNVMe 的方式。即和物理硬件是否为 NVMe 完全无关，完全由 qemu 提供一个模拟的 NVMe 虚拟设备给 guest。在 guest 看来，这就是一个 NVMe 设备。可应用于开发或测试环境，以便厂商在 guest 中去开发或验证 NVMe 协议或内核驱动。

从 qemu 1.6 起，纯模拟的 nvme 设备代码就处于陆续但缓慢的更新状态。目前，libvirt 还不支持模拟的 nvme 设备，所以我们若想在 libvirt 管理的 qemu/kvm 虚拟机中使用它，只能通过 cmdline passthrough 的方式，可手动编辑虚拟机配置文件把我们想要的命令加进去，当然，列位可曾记得在 session 2 中介绍过的神器 virt-xml? 此时它可以派上用场了。

```
host5810:/home/suse/projects/virt-manager # qemu-img create -f qcow2 /storage/test/disk5.qcow2 5G
Formatting '/storage/test/disk5.qcow2', fmt=qcow2 size=5368709120 cluster_size=65536 lazy_refcounts=off refcount_bits=16
host5810:/home/suse/projects/virt-manager #
host5810:/home/suse/projects/virt-manager #
host5810:/home/suse/projects/virt-manager # ./virt-xml sles15rc4 --edit --confirm --qemu-commandline=" \
> -drive file=/storage/test/disk5.qcow2,if=none,id=mynvme5,format=qcow2 \
> -device nvme,drive=mynvme5,serial=12345"
--- Original XML
+++ Altered XML
@@ -1,4 +1,4 @@
-<domain type="kvm">
+<domain xmlns:qemu="http://libvirt.org/schemas/domain/qemu/1.0" type="kvm">
   <name>sles15rc4</name>
   <uuid>0701cbf9-1898-4eb5-adca-5b065b24e747</uuid>
   <memory unit="KiB">2097152</memory>
@@ -94,4 +94,10 @@
       <address type="pci" domain="0x0000" bus="0x00" slot="0x08" function="0x0"/>
     </memballoon>
   </devices>
+  <qemu:commandline>
+    <qemu:arg value="-drive"/>
+    <qemu:arg value="file=/storage/test/disk5.qcow2,if=none,id=mynvme5,format=qcow2"/>
+    <qemu:arg value="-device"/>
+    <qemu:arg value="nvme,drive=mynvme5,serial=12345"/>
+  </qemu:commandline>
 </domain>

Define 'sles15rc4' with the changed XML? (y/n): y
Domain 'sles15rc4' defined successfully.
host5810:/home/suse/projects/virt-manager #
host5810:/home/suse/projects/virt-manager # virsh start sles15rc4
Domain sles15rc4 started

host5810:/home/suse/projects/virt-manager #
```

在 guest 中看下 nvme 的相关信息：

```
linux-owfi:~ # lsmod | grep nvme
nvme                   36864  0
nvme_core              86016  1 nvme
linux-owfi:~ #
linux-owfi:~ # lsblk
NAME    MAJ:MIN RM SIZE RO TYPE MOUNTPOINT
vda     253:0    0   8G  0 disk
├─vda1  253:1    0   8M  0 part
└─vda2  253:2    0   8G  0 part /
nvme0n1 259:0    0   5G  0 disk
linux-owfi:~ #
linux-owfi:~ # hwinfo --disk
21: None 00.0: 10600 Disk
  [Created at block.245]
  Unique ID: KSbE.Fxp0d3BezAE
  Parent ID: HnVB.+FFPFBVXZu6
  SysFS ID: /class/block/vda
  SysFS BusID: virtio2
  SysFS Device Link: /devices/pci0000:00/0000:00:07.0/virtio2
  Hardware Class: disk
  Model: "Disk"
  Driver: "virtio-pci", "virtio_blk"
  Driver Modules: "virtio_pci", "virtio_blk"
  Device File: /dev/vda
  Device Files: /dev/vda, /dev/disk/by-path/pci-0000:00:07.0, /dev/disk/by-path/virtio-pci-0000:00:07.0
  Device Number: block 253:0-253:15
  Geometry (Logical): CHS 16644/16/63
  Size: 16777216 sectors a 512 bytes
  Capacity: 8 GB (8589934592 bytes)
  Config Status: cfg=new, avail=yes, need=no, active=unknown
  Attached to: #19 (Storage controller)

22: PCI 00.0: 10600 Disk
  [Created at block.245]
  Unique ID: wLCS.jmv_8_ti8Q2
  Parent ID: WL76.h1ftlK6gjx9
  SysFS ID: /class/block/nvme0n1
  SysFS BusID: nvme0
  SysFS Device Link: /devices/pci0000:00/0000:00:09.0/nvme/nvme0
  Hardware Class: disk
  Model: "Red Hat QEMU Virtual Machine"
  Vendor: pci 0x8086 "Intel Corporation"
  Device: pci 0x5845 "QEMU NVM Express Controller"
  SubVendor: pci 0x1af4 "Red Hat, Inc"
  SubDevice: pci 0x1100 "QEMU Virtual Machine"
  Driver: "nvme"
  Driver Modules: "nvme"
  Device File: /dev/nvme0n1
  Device Files: /dev/nvme0n1, /dev/disk/by-id/nvme-QEMU_NVMe_Ctrl_12345, /dev/disk/by-id/nvme-nvme.8086-3
  Device Number: block 259:0
  Geometry (Logical): CHS 5120/64/32
  Size: 10485760 sectors a 512 bytes
  Capacity: 5 GB (5368709120 bytes)
  Config Status: cfg=new, avail=yes, need=no, active=unknown
  Attached to: #8 (Non-Volatile memory controller)
linux-owfi:~ #
linux-owfi:~ # lspci | grep NVM
00:09.0 Non-Volatile memory controller: Intel Corporation QEMU NVM Express Controller (rev 02)
linux-owfi:~ #
```

2. device assignment，也就是 vfio 设备直通（intel vt-d 自然是 required）。将设备以独占的形式，将 host 的 NVMe 设备直通给 guest 使用。
关于如何 vfio 直通设备给 guest，就没必要再介绍了，对吧？

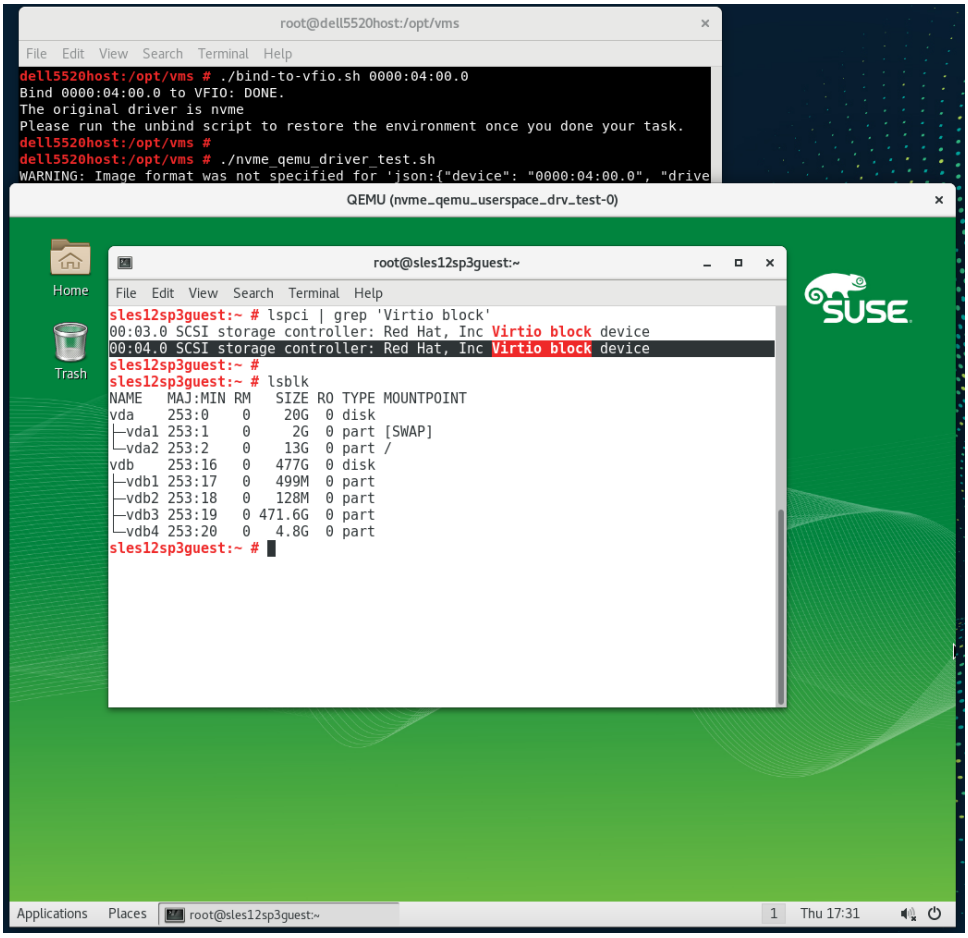
3. qemu vfio based nvme userspace driver(since qemu 2.12)，也属于独占的形式。部分模仿 spdk 的理念，在 qemu block layer 的框架下，通过 qemu nvme userspace driver 提供 nvme backend 的能力。而 front end 则由用户指定，所以 guest 看到的虚拟设备是什么，完全取决于用户。

栗：通过 qemu 2.12 引入的 vfio based nvme userspace driver，将 host 的 NVMe 设备,以 nvme backend 的方式走 vfio 分给虚拟机，front end 选择的是 virtio block。

截图中可以看到，共涉及三个脚本，bind*与 unbind*稀松平常，无任何特别，只是负责将指定的 nvme 设备的驱动与 kernel 解绑并绑定到 vfio-pci 还有 用于重新将 nvme 设备绑定到内核的 nvme 驱动，恢复环境。

第二个脚本 nvme_qemu*是启动虚拟机，内含使用 qemu nvme userspace driver 的用法。

```
dell5520host:~ # lspci -D | grep 'Non-Volatile'
0000:04:00.0 Non-Volatile memory controller: Device 1c5c:1527
dell5520host:~ #
```



此时在 host 上，我们会发现，由于 nvme 设备与内核的 nvme 驱动解绑后，host 的对应块设备木有了

```
dell5520host:/opt/vms # lsblk
NAME   MAJ:MIN RM  SIZE RO TYPE MOUNTPOINT
sda      8:0    0  1.8T  0 disk
├─sda1   8:1    0  128M  0 part
├─sda2   8:2    0    4M  0 part
├─sda3   8:3    0   16G  0 part [SWAP]
└─sda4   8:4    0  768G  0 part /
dell5520host:/opt/vms #
```

关闭虚拟机并 unbind*恢复环境后，nvme 块设备又出现了

```
dell5520host:/opt/vms # ./unbind-from-vfio.sh 0000:04:00.0 nvme
Unbind 0000:04:00.0 from VFIO: DONE.
Bind 0000:04:00.0 to nvme: DONE.
dell5520host:/opt/vms #
dell5520host:/opt/vms # lsblk
NAME        MAJ:MIN RM   SIZE RO TYPE MOUNTPOINT
sda           8:0    0   1.8T  0 disk
├─sda1        8:1    0   128M  0 part
├─sda2        8:2    0     4M  0 part
├─sda3        8:3    0    16G  0 part [SWAP]
└─sda4        8:4    0   768G  0 part /
nvme0n1     259:0    0   477G  0 disk
├─nvme0n1p1 259:1    0   499M  0 part
├─nvme0n1p2 259:2    0   128M  0 part
├─nvme0n1p3 259:3    0 471.6G  0 part
└─nvme0n1p4 259:4    0   4.8G  0 part
dell5520host:/opt/vms #
```

```
/usr/bin/qemu-kvm \
-name nvme_qemu_userspace_drv_test \
-smp 2,sockets=2,cores=1,threads=1 \
-m 4096 \
-nodefaults \
-no-hpet \
-vga cirrus \
-display sdl \
-drive file=/opt/vms/sles12sp3/disk0.qcow2,format=qcow2,if=none,id=drive-disk0-virtio \
-device virtio-blk-pci,drive=drive-disk0-virtio,id=disk0-virtio-blk,bootindex=0 \
-drive file=nvme://0000:04:00.0,if=none,id=drive-disk1-nvme \
-device virtio-blk-pci,drive=drive-disk1-nvme,id=disk1-nvme
```

4. spdk nvme，也就是 nvme driver in spdk + qemu vhost-user。既然提到了 vhost，那也就意味着 guest 看到的设备一定是 virtio 类型的喽。这种场景下，qemu 的角色变成了一个 client，spdk nvme 变成了一个 server。

玩过 dpdk + openvswitch + vhost-user-net 的童鞋们，会很容易理解这种使用方式。与 dpdk 一样，也是 spdk nvme driver 做为 server 或者叫 target, 访问 nvme 时 qemu 作为 client 的角色，通过 socket descriptor 访问在共享 hugepages 上的数据

上栗子！（假设我在 sles 15 host 上面已经编译好了 dpdk、spdk 以及 qemu 2.11）

建立 SPDK vhost target

1. 这里我们先查看下配置前的信息，然后通过 spdk 自带的脚本，方便地分配 hugepages, 将 nvme 设备与原内核驱动解绑，并重新绑定到 vfio 或 uio drivers。然后就可以在用户空间与设备进行交互了。虽然它们被 vfio_pci(基于 Intel VT-d)或 uio_pci_generic 内核模块所管理，但是这些内核模块仅负责映射设备 I/O 内存空间到用户态驱动和当设备中断被触发时去提醒用户态驱动等。

```
dell5520host:~/projects/spdk # cat /proc/meminfo | grep Huge
AnonHugePages:    264192 kB
ShmemHugePages:        0 kB
HugePages_Total:       0
HugePages_Free:        0
HugePages_Rsvd:        0
HugePages_Surp:        0
Hugepagesize:       2048 kB
dell5520host:~/projects/spdk # sysctl -a | grep huge
vm.hugepages_treat_as_movable = 0
vm.hugetlb_shm_group = 0
vm.nr_hugepages = 0
vm.nr_hugepages_mempolicy = 0
vm.nr_overcommit_hugepages = 0
dell5520host:~/projects/spdk #
```

```
dell5520host:~/projects/spdk # lspci | grep Non-Volatile
04:00.0 Non-Volatile memory controller: Device 1c5c:1284
dell5520host:~/projects/spdk #
dell5520host:~/projects/spdk # readlink /sys/bus/pci/devices/0000:04:00.0/driver
../../../../bus/pci/drivers/nvme
dell5520host:~/projects/spdk #
dell5520host:~/projects/spdk # lsblk
NAME    MAJ:MIN RM   SIZE RO TYPE MOUNTPOINT
sda       8:0    0   1.8T  0 disk
├─sda1    8:1    0   500M  0 part
├─sda2    8:2    0     8G  0 part [SWAP]
└─sda3    8:3    0 317.5G  0 part /
nvme0n1 259:0    0   477G  0 disk
dell5520host:~/projects/spdk #
```

```
dell5520host:~/projects/spdk # HUGEMEM=4096 scripts/setup.sh
0000:04:00.0 (1c5c 1284): nvme -> vfio-pci

Current user memlock limit: 0 MB

This is the maximum amount of memory you will be
able to use with DPDK and VFIO if run as current user.
To change this, please adjust limits.conf memlock limit for current user.

## WARNING: memlock limit is less than 64MB
## DPDK with VFIO may not be able to initialize if run as current user.
dell5520host:~/projects/spdk #
```

```
dell5520host:~/projects/spdk # cat /proc/meminfo | grep Huge
AnonHugePages:    264192 kB
ShmemHugePages:        0 kB
HugePages_Total:    2048
HugePages_Free:     2048
HugePages_Rsvd:        0
HugePages_Surp:        0
Hugepagesize:       2048 kB
dell5520host:~/projects/spdk # sysctl -a | grep huge
vm.hugepages_treat_as_movable = 0
vm.hugetlb_shm_group = 0
vm.nr_hugepages = 2048
vm.nr_hugepages_mempolicy = 2048
vm.nr_overcommit_hugepages = 0
dell5520host:~/projects/spdk #
```

```
dell5520host:~/projects/spdk # readlink /sys/bus/pci/devices/0000:04:00.0/driver
../../../../bus/pci/drivers/vfio-pci
dell5520host:~/projects/spdk #
```

或

```
dell5520host:~/projects/spdk # HUGEMEM=4096 scripts/setup.sh
0000:04:00.0 (1c5c 1284): nvme -> uio_pci_generic
dell5520host:~/projects/spdk # readlink /sys/bus/pci/devices/0000\:04\:00.0/driver
../../../../bus/pci/drivers/uio_pci_generic
dell5520host:~/projects/spdk #
```

2. 启动 SPDK vhost target service. 例子绑定 vhost target 到 logical cpu 0 and 1 (cpu 掩码 0x3)，socket descriptor 的路径位于/var/sles15rc4/.

Dell5520host:~/projects/spdk# **mkdir /var/sles15rc4**

```
dell5520host:~/projects/spdk # app/vhost/vhost -S /var/sles15rc4 -m 0x3
Starting SPDK v18.07-pre / DPDK 17.11.2 initialization...
[ DPDK EAL parameters: vhost -c 0x3 -m 1024 --file-prefix=spdk_pid3626 ]
EAL: Detected 8 lcore(s)
EAL: No free hugepages reported in hugepages-1048576kB
EAL: Probing VFIO support...
EAL: VFIO support initialized
app.c: 522:spdk_app_start: *NOTICE*: Total cores available: 2
reactor.c: 669:spdk_reactors_init: *NOTICE*: Occupied cpu socket mask is 0x1
reactor.c: 453:_spdk_reactor_run: *NOTICE*: Reactor started on core 1 on socket 0
reactor.c: 453:_spdk_reactor_run: *NOTICE*: Reactor started on core 0 on socket 0
```

配置 SPDK vhost target

1. 使用 construct_nvme_bdev RPC 命令创建基于 NVMe 的 block device，它将被暴露给虚拟机

```
dell5520host:~/projects/spdk # ./scripts/rpc.py construct_nvme_bdev -b Nvme0 -t pcie -a 0000:04:00.0
Nvme0n1
dell5520host:~/projects/spdk #
```

2. 创建 spdk vhost-scsi controller，用于和用户态 app 通信的 unix socket 起名为 vhost.0, qemu 将通过这个/var/sles15rc4/vhost.0 访问这个 controller. (之所以选择它当栗子是因为做为应用端的 vhost user scsi 自 qemu 2.10 开始可用。而 vhost user blk 需要 qemu 2.12，但 sles 15 中包含的 qemu 是 2.11)

```
dell5520host:~/projects/spdk # ./scripts/rpc.py construct_vhost_scsi_controller --cpumask 0x1 vhost.0
dell5520host:~/projects/spdk #
dell5520host:~/projects/spdk #
dell5520host:~/projects/spdk # ll /var/sles15rc4/
total 0
srwxr-xr-x 1 root root 0 Jun  7 14:58 vhost.0
dell5520host:~/projects/spdk #
```

3. 连接 Nvme0n1 这个 block device 到 vhost.0 vhost-scsi controller。Nvme0n1 将做为 target ID=0 的 LUN 存在于此 controller 上面。

```
dell5520host:~/projects/spdk # ./scripts/rpc.py add_vhost_scsi_lun vhost.0 0 Nvme0n1
dell5520host:~/projects/spdk #
```

该 qemu 登场了

```
dell5520host:~/projects/spdk # /opt/vms/sles15.sh
dell5520host:~/projects/spdk #
dell5520host:~/projects/spdk # cat /proc/meminfo | grep Huge
AnonHugePages:    108544 kB
ShmemHugePages:        0 kB
HugePages_Total:    2048
HugePages_Free:      512
HugePages_Rsvd:        0
HugePages_Surp:        0
Hugepagesize:       2048 kB
dell5520host:~/projects/spdk # cat /opt/vms/sles15.sh
#!/bin/bash
/usr/bin/qemu-kvm \
-name sles15 \
-m 2G \
-object memory-backend-file,id=mem0,size=2G,mem-path=/dev/hugepages,share=on \
-numa node,memdev=mem0 \
-smp 2,sockets=1,cores=2,threads=1 \
-nodefaults \
-no-hpet \
-drive file=/opt/vms/sles15rc4/disk0.qcow2,format=qcow2,if=none,id=drive-virtio-disk0 \
-device virtio-blk-pci,scsi=off,drive=drive-virtio-disk0,id=virtio-disk0,bootindex=0 \
-netdev bridge,br=br0,id=hostnet0 \
-device virtio-net-pci,netdev=hostnet0,id=net0,mac=52:54:00:c3:e2:13 \
-chardev socket,id=charserial0,path=/opt/vms/sles15rc4/serial0.socket,server,nowait \
-device isa-serial,chardev=charserial0,id=serial0 \
-monitor tcp:0.0.0.0:1234,server,nowait \
-chardev socket,id=charvhostuser0,path=/var/sles15rc4/vhost.0 \
-device vhost-user-scsi-pci,id=scsi0,chardev=charvhostuser0 \
-vga cirrus \
-display gtk &
dell5520host:~/projects/spdk #
dell5520host:~/projects/spdk # ssh 10.67.19.16
The authenticity of host '10.67.19.16 (10.67.19.16)' can't be established.
ECDSA key fingerprint is SHA256:Q3vyvQOLCYOFV6KBlIlqTvPalWeoru9qb56lgf+VCWM.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added '10.67.19.16' (ECDSA) to the list of known hosts.
Password:
Last login: Thu Jun  7 15:13:59 2018
sles15guest:~ #
sles15guest:~ # lspci | grep -i 'virtio scsi'
00:05.0 SCSI storage controller: Red Hat, Inc. Virtio SCSI
sles15guest:~ #
sles15guest:~ # lsblk
NAME   MAJ:MIN RM  SIZE RO TYPE MOUNTPOINT
sda      8:0    0  477G  0 disk
vda    253:0    0    8G  0 disk
├─vda1 253:1    0    8M  0 part
└─vda2 253:2    0    8G  0 part /
sles15guest:~ #
sles15guest:~ # exit
logout
Connection to 10.67.19.16 closed.
dell5520host:~/projects/spdk #
```

Vhost-user target 与用户态 application(也就是 qemu)通信时，要求基于 hugepages 的预分配 numa 内存，所以虚拟机内存必须基于 hugepages.
又因为 QEMU 必须与 SPDK vhost target 共享内存, 所以内存参数中加上了 share=on.

BTW,
* 当 DPDK 的版本是 2.2+时，关于虚拟机内存占用方面，较之前会有所改善
* 当使用基于 vt-d 的 vfio 时，性能较 uio 会略好些
* 当 guest kernel 使用 blk mq 时，性能会有所改善
* 对于有 NUMA 的 host，要避免 vhost target 与 qemu 中各线程被调度到不同的 socket

感兴趣的同学还可以 look look 云栖社区中一篇阿里云分享的案例。
https://yq.aliyun.com/articles/71655
以及
https://www.zhihu.com/question/265372289/answer/293318065

以上诸项，在 IO 负载很密集时，按性能排序，从高到低依次是 2, 4, 3, 1.
性能次佳的选项 4, 丧失了 qemu block 层的 features，如 mirror，backup 等。
性能最佳的选项 2, 不仅无法使用 qemu block 层的 features，还丧失了 live migration 的能力；
如果布署，取决于具体场景。个人倾向于选项 4, 这个非常适合生产环境，前途应该是灰常光明。

再插一句，Dell 已经有几款专用服务器，GPU 中集成 NVMe 驱动，实现 GPU 直接与 NVMe 通信了，如此一来 CPU 也直接成了摆设儿(虽然没有彻底踢开)

Nvdimm

关于 nvdimm，原本不在分享的计划之内，您大概也能看出来，下面的内容明显是后接上的。但因为最近组内对 NVM(非易失性内存)的热情高涨，而且国庆同学去硅谷开会带回了很多信息，所以嘛...
nvdimm 的细节我就不打算说了，大家 google 一下就可以掌握很多信息，这里只是简单梳理一下基本脉络，建立一个框架，并分享些在目前的 qemu/kvm 环境下 nvdimm 的使用.

什么是 nvm? nvdimm 又是啥玩意？它们这间有没有联系？是不是有同学傻傻分不清楚？在一个阳光明媚的夜晚，让我们开始语无伦次吧.

NVM: Non Volatile Memory，非易失存储器，掉电数据不丢失、按字节存取、存储密度高、低能耗、读写性能接近 DRAM，但读写速度不对称，读远快于写，寿命有限。
无论是传统的 2D nand flash，如 ssd，还是明日的 3D nand flash，或者下一代的 Intel 3D XPoint memory 或 ReRAM，都是 NVM. 其实如果按照'非易失存储器'的标准，磁带，软盘，硬盘等都可以算作 NVM.

NVMe: Non Volatile Memory express, 描述了连接到 PCIe 总线上的 NVM。 NVMe 协议减少了 SCSI 栈中不必要的开销。它支持比标准 SCSI 更多的队列，将队列数量从传统的 AHCI 的队列数增加到 64k。每个 NVMe 队列还可以支持 64k 个命令，而不是 AHCI 在其单个队列中支持的 32 个命令。

NVDIMM: Non Volatile Dual In-line Memory Module 非易失性双列直插式内存模块.

不严谨地讲，NVM 是存储介质；NVMe 是标准，是协议；而 nvdimm 除了闪存部分的性质与 NVMe 都是非易失性以外，就是完全不同的设备类别了。比如说一个是字节寻址，一个是块 I/O；一个是 dimm 接口中，一个是 PCIe，等等等。

SNIA(全球网络存储工业协会)于 2013 年底发布的 NVM 规范中描述了硬件接口和编程模型。定义了 NVM 的范围为 PCIe 接口的 flash 存储器，控制卡以及 NVDIMM，PCM 等可以随机访问的非易失性存储器。SNIA 提出了 block volume 和 persistent memory 两种模型，并给出了每种模型能够处理的命令集以及能完成的功能, NVDIMM pmem 就属于这种模型。Block volume 即传统的块设备模型，以块为单位进行数据传输，采用与现有的 ATA,SCSI,FC 等协议具有相同的编程方式，比如 PCIE SSD 就是此类模型。Persistent memory 是具有直接随机访问与非易失性双重特性的编程模型，可以采用传统虚拟内存管理接口，为文件系统或者数据库提供新的存储行为.

JEDEC(固态技术协会)描述了三种 nvdimm 实现：
* NVDIMM-N: 出现的较早 ,同时采用 nand flash + DRAM，CPU 的 load/store 等访存行为发生在 DRAM 上，使用 NAND Flash 做掉电保护，在掉电时，nvdimm-n 将数据从易失性的 DRAM 复制到非易失性的 nand flash(它使用小型电源或超级电容来供电)，并在恢复供电时复制回来。一般使用 8-32GB 的 DRAM+同等或更大容量的 NAND Flash。基于 DDR4 标准 ，标准中描述了 4 个特殊引脚，用于定义掉电时的备份与上电时的恢复行为。

* NVDIMM-F: 可以理解成是将 nand flash 做成 DIMM 接口的块设备,通过这种方式可以进一步避免由于 PCIe 总线引入的访问延迟, 容量一般在 100GB-1TB

* NVDIMM-P: 未来产品，基于 DDR5 标准，有点像 nvdimm-n 和 nvdimm-f 的结合，可以字节访问也可以块访问, 其能够在关机状态下保存数据，不了解。

nvdimm 的编程模型分为两种：
1、pmem 方式使用 (pmem 可以看作是 NVM 的比较好的应用领域,)
cpu 直接访问 pmem dimm，消除 PCIe 总线协议的开销。通过支持 DAX 的设备操作基于 pmem 的 mmap 时，I/O 是绕过内核的
(当我们使用 mmap 来 map 一个文件至内存时，就是告诉内核，映射一个文件至内存，然后将此内存区域暴露在进程的虚拟地址空间中。
如果一个文件是基于块存储， 当这个文件被映射时，此内存区域是字节寻址的，但底层存储还是以块的形式通信，因为即使是单个字节发生改变，整个 4k 的块会移动到存储，效率还是有瓶颈。
如果一个文件是基于 persistent memory(pmem), 此内存区域是字节寻址的，借助于内核的 dax 特性，page cache 就被绕过了。
dax 即 Direct Access ,是访问 pmem 设备时，绕过文件系统 page cache 的一种机制。support dax 的文件系统就是具备 pmem 感知能力的文件系统
)

下面一段英文描述 pmem 特别简洁，翻译都是多余的.
Byte-addressable like memory
Persistent like storage
Cache coherent
Load/store accessible—persistent memory is fast enough to be used directly by the CPU
Direct access-no paging from a storage device to dynamic random-access memory (DRAM)

2、块设备（btt／sector）方式
通过块设备方式来访问 nvdimm，其他不知

qemu 对 nvdimm 的支持仅限于 pmem 方式。前端会呈现一个虚拟 nvdimm 设备给虚拟机，俗称 vNVDIMM。(又因为 qemu 仅支持 pmem 方式，所以也可以称 vNVDIMM 为 vPMEM，意思是虚拟的 nvdimm pmem 设备)
后端可以是 host 的物理 nvdimm pmem 设备节点，也可以是 host 的内存或文件。
我的栗子中，后端还是基于大页内存的。

先检查下虚拟机内核对 nvdimm 与 dax 的支持情况。

```
sles15guest:~ # cat /proc/config.gz | gzip -d | egrep -i 'dax|nvdimm'
CONFIG_BLK_DEV_RAM_DAX=y
CONFIG_LIBNVDIMM=m
CONFIG_NVDIMM_PFN=y
CONFIG_NVDIMM_DAX=y
CONFIG_DAX=y
CONFIG_DEV_DAX=m
CONFIG_DEV_DAX_PMEM=m
CONFIG_FS_DAX=y
CONFIG_FS_DAX_PMD=y
sles15guest:~ #
```

分配 4096 个 2MB 的大页，共 8GB

```
host5810:/opt/vms # sysctl vm.nr_hugepages=4096
vm.nr_hugepages = 4096
host5810:/opt/vms #
host5810:/opt/vms # cat /proc/meminfo | grep Huge
AnonHugePages:    589824 kB
ShmemHugePages:        0 kB
HugePages_Total:    4096
HugePages_Free:     4096
HugePages_Rsvd:        0
HugePages_Surp:        0
Hugepagesize:       2048 kB
Hugetlb:         8388608 kB
host5810:/opt/vms #
```

从中分给虚拟机 4GB，做为 vPMEM 的 backend

```
host5810:/opt/vms # cat nvdimm-sles15.sh
#!/bin/bash

/usr/bin/qemu-system-x86_64 \
-name sles15rc4 \
-machine pc-i440fx-2.12,nvdimm=on,accel=kvm \
-nodefaults \
-smp 2,sockets=1,cores=2,threads=1 \
-m 2G,slots=4,maxmem=32G \
-object memory-backend-file,share,id=mem0,mem-path=/dev/hugepages,size=4G \
-device nvdimm,memdev=mem0,id=nvdimm0 \
-vga cirrus \
-display sdl \
-device ich9-usb-uhci1,id=usb1 \
-device usb-tablet,id=usb-tablet1 \
-netdev bridge,br=virbr0,id=hostnet0 \
-device virtio-net-pci,id=net0,netdev=hostnet0 \
-drive file=/opt/vms/sles15rc4/disk0.qcow2,if=none,id=drive-virtio-disk0,format=qcow2 \
-device virtio-blk-pci,id=virtio-disk0,drive=drive-virtio-disk0,bootindex=0 \
-monitor tcp:0.0.0.0:1234,server,nowait &
host5810:/opt/vms #
```

启动虚拟机后，通过 qemu monitor 查看下内存设备的信息

```
(qemu) info memory-devices
info memory-devices
Memory device [nvdimm]: "nvdimm0"
  addr: 0x100000000
  slot: 0
  node: 0
  size: 4294967296
  memdev: /objects/mem0
  hotplugged: false
  hotpluggable: true
(qemu)

(qemu) info memory_size_summary
info memory_size_summary
base memory: 2147483648
plugged memory: 4294967296
(qemu)
```

再检查下，我们添加的 vPMEM 在虚拟机中的状态正常否

```
sles15guest:~ # lsmod | egrep -i 'nvdimm|pmem'
dax_pmem               16384  0
device_dax             20480  1 dax_pmem
nd_pmem                16384  0
nd_btt                 24576  1 nd_pmem
libnvdimm             163840  4 nd_btt,nd_pmem,dax_pmem,nfit
sles15guest:~ #
sles15guest:~ #
sles15guest:~ # lsblk
NAME   MAJ:MIN RM SIZE RO TYPE MOUNTPOINT
vda    253:0    0   8G  0 disk
├─vda1 253:1    0   8M  0 part
└─vda2 253:2    0   8G  0 part /
pmem0  259:0    0   4G  0 disk
sles15guest:~ #
```

制作文件系统(ext4, xfs 等都支持 dax)，挂载并使能 dax

```
sles15guest:~ # lsblk
NAME   MAJ:MIN RM SIZE RO TYPE MOUNTPOINT
vda    253:0    0   8G  0 disk
├─vda1 253:1    0   8M  0 part
└─vda2 253:2    0   8G  0 part /
pmem0  259:0    0   4G  0 disk
sles15guest:~ #
sles15guest:~ # ll /dev/pmem0
brw-rw---- 1 root disk 259, 0 Jun  8 13:34 /dev/pmem0
sles15guest:~ #
sles15guest:~ # mkfs.ext4 /dev/pmem0
mke2fs 1.43.8 (1-Jan-2018)
/dev/pmem0 contains a ext4 file system
        last mounted on /mnt on Fri Jun  8 13:35:03 2018
Proceed anyway? (y,N) y
Creating filesystem with 1048576 4k blocks and 262144 inodes
Filesystem UUID: a9796814-afbc-44f2-9ef6-4c34c93e18a9
Superblock backups stored on blocks:
        32768, 98304, 163840, 229376, 294912, 819200, 884736

Allocating group tables: done
Writing inode tables: done
Creating journal (16384 blocks): done
Writing superblocks and filesystem accounting information: done

sles15guest:~ #
sles15guest:~ # mount -o dax /dev/pmem0 /mnt/
sles15guest:~ #
sles15guest:~ # mount | grep dax
/dev/pmem0 on /mnt type ext4 (rw,relatime,dax,data=ordered)
sles15guest:~ #
```

好，至此，从'硬件'的角度，the vPMEM 已经 ready 了，如果想开发基于 pmem 的应用，我们需要用 ndctl 工具和 NVM Library, 现已更名为 Persistent Memory Developer Kit 即 PMDK。(注：SNIA 并未描述 NVM 规范相关的 API)
比如，如果我们想从 pmem 上分配一块空间的话，传统的 malloc 是不行的，需要使用 PMDK 中提供的接口来做此事。
还有，PMDK 不可以用于 NVMe 等块寻址的设备，它是专门为字节寻址设备(也就是内存)设计的。

可是我们手头啥也没有啊，学习 PMDK 的时间成本又有些高，想个折来验证下 pmem 吧？于是接下来我打算找 NVM Library 的测试套件来帮忙 :-)

```
sles15guest:~/projects/nvml/src/test # ./RUNTESTS pmem_is_pmem
pmem_is_pmem/TEST0: SETUP (check/non-pmem/debug)
pmem_is_pmem/TEST0: PASS                        [00.047 s]
pmem_is_pmem/TEST0: SETUP (check/non-pmem/nondebug)
pmem_is_pmem/TEST0: PASS                        [00.048 s]
pmem_is_pmem/TEST0: SETUP (check/non-pmem/static-debug)
pmem_is_pmem/TEST0: PASS                        [00.047 s]
pmem_is_pmem/TEST0: SETUP (check/non-pmem/static-nondebug)
pmem_is_pmem/TEST0: PASS                        [00.046 s]
pmem_is_pmem/TEST1: SETUP (check/pmem/debug)
pmem_is_pmem/TEST1: PASS                        [00.048 s]
pmem_is_pmem/TEST1: SETUP (check/pmem/nondebug)
pmem_is_pmem/TEST1: PASS                        [00.047 s]
pmem_is_pmem/TEST1: SETUP (check/pmem/static-debug)
pmem_is_pmem/TEST1: PASS                        [00.047 s]
pmem_is_pmem/TEST1: SETUP (check/pmem/static-nondebug)
pmem_is_pmem/TEST1: PASS                        [00.046 s]
pmem_is_pmem/TEST2: SETUP (check/non-pmem/debug)
pmem_is_pmem/TEST2: PASS                        [00.047 s]
```

时间有限，公司也不富裕，所以没有深研究，也没有体验过真实硬件，先到这里吧，就此别过。