Lenovo

Configuring NVIDIA Virtual GPU (vGPU) in a Linux VM on Lenovo ThinkSystem Servers

Explains how to create vGPUs and assign the vGPUs to a VM

Provides step-by-step instructions on how to configure Linux

Describes implementation in both AMD and Intel-based servers

Provides examples driving an NVIDIA vGPU in guest OS

Xiaochun Li

Abstract

Virtual GPU (vGPU) is the method of virtualizing a GPU adapter installed in a server so that the physical GPU can be shared by multiple virtual machines running on that server. This method assigns the virtual devices to multiple guest operating systems, and the guest OSes share the performance of the single physical GPU.

This paper provides guidance on enabling vGPU and making it available to VM running in a Kernel Virtual Machine (KVM)-based OS. The paper is for Linux administrators wishing to use a vGPU in a ThinkSystem™ server to pass through to a VM. For our evaluation, we implement Red Hat Enterprise Linux 8.5 both as the host OS and guest OS, and show how to virtualize the resources of an NVIDIA T4 GPU.

At Lenovo® Press, we bring together experts to produce technical publications around topics of importance to you, providing information and best practices for using Lenovo products and solutions to solve IT challenges.

See a list of our most recent publications at the Lenovo Press web site:

http://lenovopress.com

Do you have the latest version? We update our papers from time to time, so check whether you have the latest version of this document by clicking the **Check for Updates** button on the front page of the PDF. Pressing this button will take you to a web page that will tell you if you are reading the latest version of the document and give you a link to the latest if needed. While you're there, you can also sign up to get notified via email whenever we make an update.

Contents

Introduction 3
Setup NVIDIA vGPU 3
Create NVIDIA vGPUs 8
Installing the vGPU driver on the Linux guest OS 13
Deleting a vGPU on Linux with KVM 15
Prerequisites for using NVIDIA vGPU on the NVIDIA Ampere architecture. 16
Resources 19
Authors 19
Notices 20
Trademarks 21

Introduction

Many Lenovo ThinkSystem servers support the use Graphics Processing Unit (GPU) for additional high performance video and data processing. GPUs are typically used to offload tasks from the server CPU, such as AI, VDI, and rendering tasks.

For servers running hypervisors, these GPUs can be allocated directly to a single virtual machine for exclusive access using a method called *PCI device pass through* or *GPU pass through*. The GPU appears as if it was physically attached to the guest OS running in the VM.

However, with the large-scale deployment of virtual machines, there is a greater need for GPU resources than the server can physically support, so a new feature call vGPU was developed that makes it possible to divide a physical GPU device into multiple virtual devices and assign the virtual devices to multiple guest operating systems. Multiple guest OSes then share the performance of a single physical GPU.

In this paper, we describe the process of setting up a vGPU environment. For our evaluation, we implement Red Hat Enterprise Linux 8.5 both as the host OS and guest OS, and show how to virtualize the resources of an NVIDIA T4 GPU.

Setup NVIDIA vGPU

The NVIDIA vGPU system architecture is shown in Figure 1. NVIDIA physical GPUs can support multiple virtual GPU devices (vGPUs) that can be assigned directly to guest VMs. This functionality is provided by the NVIDIA Virtual GPU Manager running under the hypervisor.

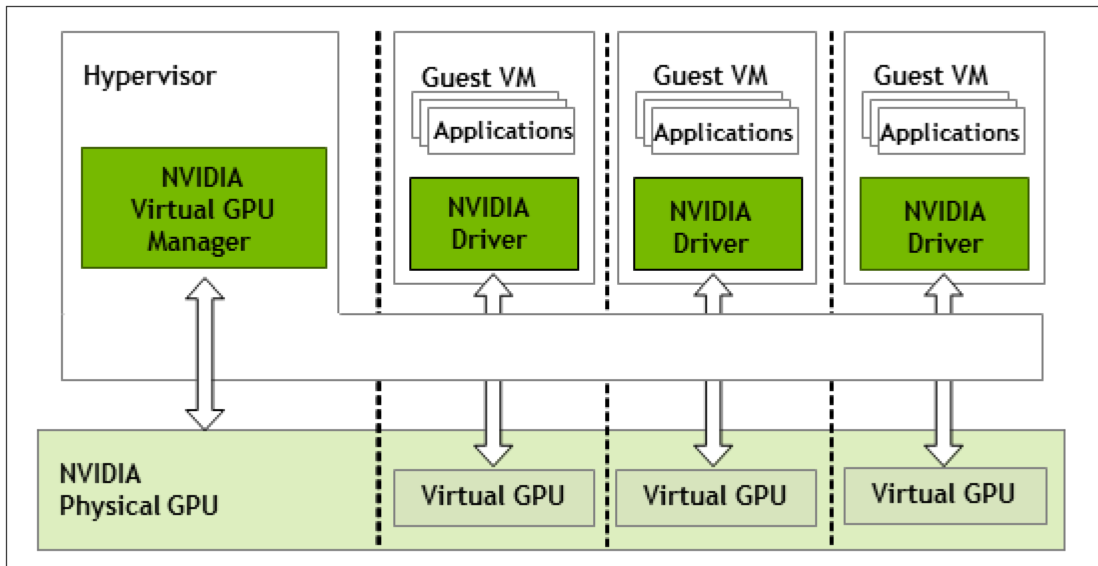

Figure 1 NVIDIA vGPU System Architecture[1]

Hypervisors take advantage of mediated device framework. The vGPU passthrough provides the direct access to a physical GPU. Two methods are supported in the framework, fast path and slow path. The purpose of the fast path is to run the performance-critical path in the NVIDIA driver and provide direct access to the GPU on the guest VM, while the slow path

[1] From https://docs.nvidia.com/grid/latest/grid-vgpu-user-guide/index.html#architecture-grid-vgpu

© Copyright Lenovo 2022. All rights reserved.

supports the non-performant management operations provided by the interface in NVIDIA virtual GPU manager.

In order to use NVIDIA vGPU on a VM, two drivers are required as shown as the dark green boxes in Figure 1 on page 3:

- NVIDIA Virtual GPU Manager running on a hypervisor
- NVIDIA vGPU driver running on a guest VM

The first step is to download NVIDIA drivers for your GPU device and then install NVIDIA Virtual GPU manager on the host OS. After the vGPU Manager is successfully installed, we need to create a mediated device on the host so that we can assign the vGPU to the VM.

Download NVIDIA GPU driver for your device

Prerequisites for NVIDIA GPUs:

- 64-bit Linux guest VMs are only supported using Q-series, C-series and B-series NVIDIA vGPU types, A-series NVIDIA vGPU types are not supported.
- Windows guest VMs are supported only using Q-series, B-series, and A-series NVIDIA vGPU types. C-series NVIDIA vGPU types are not supported.

For getting the NVIDIA vGPU drivers and installing them on your systems, please follow NVIDIA document *Grid Software Quick Start Guide* available at the following page. An NVIDIA enterprise account is needed for this download.

https://docs.nvidia.com/grid/latest/grid-software-quick-start-guide/index.html#redeeming-pak-and-downloading-grid-software

Log in and download your vGPU driver for your host OS version. For instance, you can get a compressed file “NVIDIA-GRID-RHEL-8.5-510.47.03-511.65.zip” for RHEL8.5. After extracting this file, you can see some drivers and documents on your system, as shown in Figure 2.

```
# unzip NVIDIA-GRID-RHEL-8.5-510.47.03-511.65.zip
Archive:  NVIDIA-GRID-RHEL-8.5-510.47.03-511.65.zip
  inflating: 510.47.03-511.65-grid-gpumodeswitch-user-guide.pdf
  inflating: 510.47.03-511.65-grid-licensing-user-guide.pdf
  inflating: 510.47.03-511.65-grid-software-quick-start-guide.pdf
  inflating: 510.47.03-511.65-grid-vgpu-release-notes-red-hat-el-kvm.pdf
  inflating: 510.47.03-511.65-grid-vgpu-user-guide.pdf
  inflating: 510.47.03-511.65-whats-new-vgpu.pdf
  inflating: 511.65_grid_win10_win11_server2016_server2019_server2022_64bit_international.exe
  inflating: nvidia-linux-grid-510-510.47.03-1.x86_64.rpm
 extracting: nvidia-linux-grid-510_510.47.03_amd64.deb
 extracting: NVIDIA-Linux-x86_64-510.47.03-grid.run
  inflating: NVIDIA-vGPU-rhel-8.5-510.47.03.x86_64.rpm
#
```

Figure 2 Contents of NVIDIA-GRID-RHEL-8.5-510.47.03-511.65.zip

In this zip:

- nvidia-linux-grid-510-510.47.03-1.x86_64.rpm -- vGPU driver to be installed on your VM
- NVIDIA-vGPU-rhel-8.5-510.47.03.x86_64.rpm -- NVIDIA vGPU Manager to be installed on your host OS.

Installing the NVIDIA Virtual GPU Manager on a KVM based OS

As mentioned above, "NVIDIA-vGPU-rhel-8.5-510.47.03.x86_64.rpm" is the NVIDIA Virtual GPU Manager that needs to be installed on Host OS on RHEL8.5. NVIDIA Virtual GPU Manager and guest VM drivers must be compatible. If you update vGPU Manager to a release that is incompatible with the guest VM drivers, guest VMs will boot with vGPU disabled until their guest vGPU driver is updated to a compatible version.

Before installing NVIDIA Virtual GPU Manager software, your GPU driver is display as follows through the command lspci as shown below.

```
#lspci |grep -i nvidia
4b:00.0 3D controller: NVIDIA Corporation TU104GL [Tesla T4] (rev a1)
#
# lspci -vvvnnn -s 4b:00.0|grep -i kernel
        Kernel driver in use: nouveau
        Kernel modules: nouveau
```

Figure 3 Checking the status of the GPU driver before installing vGPU Manager

Use the rpm command to install the rpm package as shown below.

```
# rpm -ivh NVIDIA-vGPU-rhel-8.5-510.47.03.x86_64.rpm
Verifying...                          ################################# [100%]
Preparing...                          ################################# [100%]
Updating / installing...
   1:NVIDIA-vGPU-rhel-1:8.5-510.47.03 ################################# [100%]
#
```

Figure 4 Installing the NVIDIA Virtual GPU Manager

After the NVIDIA Virtual GPU Manager software is successfully installed, the lspci command output should be listed below.

```
# lspci -vvvnnn -s 4b:00.0 |grep -i kernel
        Kernel driver in use: nouveau
        Kernel modules: nouveau, nvidia_vgpu_vfio, nvidia
#
```

Figure 5 Checking the status of the GPU driver after installing vGPU Manager

The driver `nvidia_vgpu_vfio` and `nvidia` are be loaded by your GPU device, but `nouveau` driver is still running. To switch to the NVIDIA dedicated driver, the inbox driver must be unbound to your GPU device. The following section will unbind the inbox driver `nouveau` on the host OS. To switch to the NVIDIA dedicated driver, the inbox driver must be unbound to your GPU device. The following section will unbind the inbox driver `nouveau` on your host OS.

Unbinding inbox driver nouveau on host OS

This section describes how to enable an NVIDIA GPU using the Linux console.

When using an assigned NVIDIA GPU in the guest OS, only the NVIDIA drivers are supported. Other drivers may not work well, therefore, it is advised that you use the

proprietary driver offered by NVIDIA. Since we have finished installing Virtual GPU manager on the above steps, the proprietary NVIDIA driver has been already installed in our host OS. Next, we only need unbind the inbox driver nouveau and reboot the system. The nvidia driver will load automatically.

Detailed steps are as follows:

1. Create the config file `/etc/modprobe.d/nvidia-blacklist.conf` and add the following two lines to it:

```
# cat /etc/modprobe.d/nvidia-blacklist.conf
blacklist nouveau
options nouveau modeset=0
```

Figure 6 Contents of nvidia-blacklist.conf

2. Back up the current `initramfs` and build a new one for the current kernel as follows. If you would like to regenerate initramfs for all installed kernel versions, you can run command **dracut** command as shown.

```
# mv /boot/initramfs-$(uname -r).img /boot/initramfs-$(uname -r)-nouveau.img
# dracut /boot/initramfs-$(uname -r).img $(uname -r)
```

Figure 7 Back up the current initramfs and build a new one

3. Restart the system.

   The system should not be loaded with the nouveau module so far. To verify that NVIDIA's proprietary driver is used in your system, the following command can help you. You should see the kernel driver in use has changed from nouveau to nvidia.

```
# lspci |grep -i nvidia
4b:00.0 3D controller: NVIDIA Corporation TU104GL [Tesla T4] (rev a1)
# lspci -vvvnnn -s 4b:00.0|grep -i kernel
        Kernel driver in use: nvidia
        Kernel modules: nouveau, nvidia_vgpu_vfio, nvidia
```

Figure 8 Verify the kernel driver in use

Checking and starting NVIDIA vGPU software

After installing NVIDIA vGPU Manager software and restarting the system, you need to perform the following three steps to verify that the software is working correctly.

1. Check whether the kernel has loaded relevant VFIO modules. The driver nvidia_vgpu_vfio depends on driver mdev, so vGPUs are created based on Linux mediated framework.

```
# lsmod |grep -i vfio
nvidia_vgpu_vfio       57344  0
vfio_mdev              16384  0
mdev                   20480  2 vfio_mdev,nvidia_vgpu_vfio
vfio_iommu_type1       36864  0
vfio                   36864  3 vfio_mdev,nvidia_vgpu_vfio,vfio_iommu_type1
#
```

Figure 9 Check the kernel for VFIO modules

2. Check if nvidia-vgpu-mgr.service is running in the host OS by issuing the following systemctl command. -- nvidia-vgpu-mgr.service is a daemon that provides an interface for vGPU configuration. If it works, you will see `active (running)` in the output of command as shown below.

```
# systemctl status nvidia-vgpu-mgr
nvidia-vgpu-mgr.service - NVIDIA vGPU Manager Daemon
   Loaded: loaded (/usr/lib/systemd/system/nvidia-vgpu-mgr.service; enabled; vendor preset: 
disabled)
   Active: active (running) since Tue 2022-03-15 10:06:13 EDT; 1h 38min ago
  Process: 3802 ExecStart=/usr/bin/nvidia-vgpu-mgr (code=exited, status=0/SUCCESS)
 Main PID: 3807 (nvidia-vgpu-mgr)
    Tasks: 1 (limit: 201560)
   Memory: 416.0K
   CGroup: /system.slice/nvidia-vgpu-mgr.service
           ··3807 /usr/bin/nvidia-vgpu-mgr

Mar 15 10:06:13 vGPUHost systemd[1]: Starting NVIDIA vGPU Manager Daemon...
Mar 15 10:06:13 vGPUHost systemd[1]: Started NVIDIA vGPU Manager Daemon.
Mar 15 10:06:14 vGPUHost nvidia-vgpu-mgr[3807]: notice: vmiop_env_log: nvidia-vgpu-mgr daemon 
started
#
```

Figure 10 Output of the command systemctl status nvidia-vgpu-mgr

3. Check if the NVIDIA driver can successfully communicate with the NVIDIA physical GPUs in your system by running command **`nvidia-smi`** as shown in Figure 11 on page 8. The command should produce a listing of GPUs in your system. In our example, the output lists a Tesla T4 on our platform and its bus id is “4b:00.0”.

```
[root@vGPUHost nvidia]# nvidia-smi
Tue Mar 15 12:19:58 2022
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 510.47.03    Driver Version: 510.47.03    CUDA Version: N/A      |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|                               |                      |               MIG M. |
|===============================+======================+======================|
|   0  Tesla T4            On   | 00000000:4B:00.0 Off |                    0 |
| N/A   39C    P8    16W /  70W |     82MiB / 15360MiB |      0%      Default |
|                               |                      |                  N/A |
+-------------------------------+----------------------+----------------------+

+-----------------------------------------------------------------------------+
| Processes:                                                                  |
|  GPU   GI   CI        PID   Type   Process name                  GPU Memory |
|        ID   ID                                                   Usage      |
|=============================================================================|
|  No running processes found                                                 |
+-----------------------------------------------------------------------------+
[root@vGPUHost nvidia]#
```

Figure 11   Output from the nvidia-smi command

Create NVIDIA vGPUs

After successfully setting up vGPU as described in the above section, you can create a vGPU on your host OS. The following steps are based on a Linux with KVM hypervisor.

Get the mdev_supported_types for your physical GPU device

After unbinding NVIDIA inbox driver and attaching NVIDIA dedicated driver described in “Unbinding inbox driver nouveau on host OS” on page 5, load the mdev driver, create a mdev_bus on the host OS and attach the physical GPU on the mdev_bus. The sysfs interface in Linux OS exposes the attributes of the devices. The attribute named `mdev_supported_types` that we want to get is also shown in sysfs. Command **`virsh nodedev-dumpxml`** can assemble sysfs information for GPU device. For instance, see the following examples.

1. Get the parameter for your configuration by using the following command. The output of the command (`pci_0000_4b_00_0` in our example) is the PCI domain bus slot and function that is used in the libvirt-compatible identifier.

```
# virsh nodedev-list --cap mdev_types
pci_0000_4b_00_0
```

Figure 12   Determining the PCI bus number

2. Use the following command to show the physical GPU attributes on your host OS, using the value you obtained in the previous step as the parameter.

```
# virsh nodedev-dumpxml pci_0000_4b_00_0
<device>
  <name>pci_0000_4b_00_0</name>
  <path>/sys/devices/pci0000:4a/0000:4a:02.0/0000:4b:00.0</path>
  <parent>pci_0000_4a_02_0</parent>
  <driver>
    <name>nvidia</name>
  </driver>
  <capability type='pci'>
    <class>0x030200</class>
    <domain>0</domain>
    <bus>75</bus>
    <slot>0</slot>
    <function>0</function>
    <product id='0x1eb8'>TU104GL [Tesla T4]</product>
    <vendor id='0x10de'>NVIDIA Corporation</vendor>
    <capability type='virt_functions' maxCount='16'/>
    <capability type='mdev_types'>
      <type id='nvidia-223'>
        <name>GRID T4-2B</name>
        <deviceAPI>vfio-pci</deviceAPI>
        <availableInstances>8</availableInstances>
      </type>
      <type id='nvidia-231'>
        <name>GRID T4-2Q</name>
        <deviceAPI>vfio-pci</deviceAPI>
        <availableInstances>8</availableInstances>
      </type>
      <type id='nvidia-228'>
        <name>GRID T4-8A</name>
        <deviceAPI>vfio-pci</deviceAPI>
        <availableInstances>2</availableInstances>
      </type>
   …
      <type id='nvidia-233'>
        <name>GRID T4-8Q</name>
        <deviceAPI>vfio-pci</deviceAPI>
        <availableInstances>2</availableInstances>
      </type>
    </capability>
    <numa node='0'/>
    <pci-express>
      <link validity='cap' port='0' speed='8' width='16'/>
      <link validity='sta' speed='2.5' width='16'/>
    </pci-express>
  </capability>
</device>
```

Figure 13 Output of the virsh nodedev-dumpxml command

If your system has two or more GPUs that support vGPU, the command above **virsh** command will display more strings. To identify which physical GPU to create a vGPU on, the following **lspci** command can help you.

```
# lspci -D |grep -i nvidia
0000:4b:00.0 3D controller: NVIDIA Corporation TU104GL [Tesla T4] (rev a1)
```

Figure 14 Using lspci to determine the correct GPU to use

From each line output, we can find GPU name and the corresponding PCI bus number. In the above output, we only have NVIDIA GPU on the platform, so there is only one line of output. The example shows the GPU named NVIDIA Tesla T4 and `0000:4b:00.0` means domain, bus, slot and function. There is a direct relationship between `0000:4b:00.0` and `pci_0000_4b_00_0`. The value `pci_0000_4b_00_0` is libvirt-compatible identifier that only replaces characters : (colon) and . (period) characters with underscores in `0000:4b:00.0`.

3. Review the output of the command, Figure 13 on page 9, in particular the contents of the following section:

   ```
   <capability type='mdev_types'>
   ```

   One of the subsections, shown in red in Figure 13 on page 9 is as follows:

   ```
   <type id='nvidia-231'>
     <name>GRID T4-2Q</name>
     <deviceAPI>vfio-pci</deviceAPI>
     <availableInstances>8</availableInstances>
   </type>
   ```

   The parameters are as follows:

   – The type ID is `nvidia-231`, which is one of the mdev_supported_types of the physical GPU. We use this value in the next steps.

   – The name is `GRID T4-2Q`. The Q in the name indicates this is a Q-series GPU.

     64-bit Linux guest VMs are only supported using Q-series, C-series and B-series NVIDIA vGPU types, A-series NVIDIA vGPU types are not supported.

     Windows guest VMs are supported only using Q-series, B-series, and A-series NVIDIA vGPU types. C-series NVIDIA vGPU types are not supported.

     We choose GRID T4-2Q because it's a Q-series type that will be assigned to a Linux guest VM.

4. Generate an universal unique identifier (UUID) with correct format for vGPU.

```
# uuidgen
e31d555b-9367-432d-87b3-482331bbd1f2
#
```

Figure 15 Generate a UUID

5. Write the UUID that you obtained in the previous step to the create file in the Linux sysfs directory for the vGPU type that you want to create.

```
# echo e31d555b-9367-432d-87b3-482331bbd1f2 >
/sys/class/mdev_bus/0000\:4b\:00.0/mdev_supported_types/nvidia-231/create
#
```

Figure 16 Contents of the create file

6. Confirm mdev device file was created for the vGPU. The mdev device file for the vGPU is added to the parent physical device directory. The vGPU is identified by its UUID, and the /sys/bus/mdev/devices/ directory contains a symbolic link to the mdev device file. See the following display, two vGPUs are created on the system.

```
# ll /sys/bus/mdev/devices/
total 0
lrwxrwxrwx. 1 root root 0 Mar 17 12:20 75ced98d-011f-4dc5-a053-3e49f569dd0e ->
../../../devices/pci0000:4a/0000:4a:02.0/0000:4b:00.0/75ced98d-011f-4dc5-a053-3e49f569dd0e
lrwxrwxrwx. 1 root root 0 Mar 17 12:19 e31d555b-9367-432d-87b3-482331bbd1f2 ->
../../../devices/pci0000:4a/0000:4a:02.0/0000:4b:00.0/e31d555b-9367-432d-87b3-482331bbd1f2
```

Figure 17 Listing the mdev devices

Alternatively, run command **virsh nodedev-list --cap mdev** to check the vGPU created on your host OS.

```
# virsh nodedev-list --cap mdev
mdev_75ced98d-011f-4dc5-a053-3e49f569dd0e
mdev_e31d555b-9367-432d-87b3-482331bbd1f2
```

Figure 18 Listing the mdev devices using virsh

Now we have a vGPU device identified by a UUID, which can be assigned to a guest VM. If desired, you can create multiple vGPUs on your system by following the above steps and add multiple vGPUs to a single VM. NVIDIA vGPU software supports up to 16 vGPUs per VM on Red Hat Enterprise Linux with KVM.

The next section will guide you on how to assign the vGPU to a gust OS.

Assign one or more vGPUs to a KVM based VM

The KVM hypervisor version and NVIDIA vGPUs support the assignment of multiple vGPUs to a single VM, so you can add multiple vGPUs to a single VM to support computing or graphics intensive applications and workloads. To see which KVM versions and vGPUs support this feature, see the sections “Linux Guest OS support” and “Multiple vGPU Support”:

https://docs.nvidia.com/grid/latest/grid-vgpu-release-notes-red-hat-el-kvm/index.html

If the guest OS is Red Hat Enterprise Linux 8.5, the host OS needs to be RHEL 8.5, 8.4 or 8.2. And on NVIDIA Tesla T4 GPU, only T4-16Q and T4-16C vGPU types have multiple vGPU support to be added to a single VM.

For adding vGPUs to VM, make sure the following prerequisites are met:

1. Guest OS version you’ve installed on your system based on KVM hypervisor must be in line with the specified host OS NVIDIA. Please refer to the above NVIDIA web page for which Guest OS version you will install.
2. The Guest OS which you want to add to the vGPUs is shut down. Use the command **virsh list --all** checking the status of all domains and command **virsh start/shutdown <domain name>** to start/shutdown a domain.

```
# virsh list --all
 Id   Name      State
-------------------------
 -    rhel8.5   shut off
#
```

Figure 19 List the state of running guest OSes

3. Follow the steps in “Get the mdev_supported_types for your physical GPU device” on page 8 to create one or more vGPUs on your system.

Add vGPUs to VM by using virsh

In order to attach vGPU to a guest OS, follow the virsh instructions below.

1. Run command **vrish edit <domain_name>** to edit the domain XML configuration file. You can obtain the domain name from the command **virsh list-all** as shown in Figure 19. The column name shows the domain name of your Guest OS.
2. Add the appropriate device XML entry to the `<devices>` sections in the XML configurations of guests that you want to get the vGPU resources. Add the following line in the form of an address element inside the source element. Use the UUID value generated by the **uuidgen** command in the previous step. Each UUID can only be assigned to one guest at a time.

```
# virsh edit rhel8.5
    <hostdev mode='subsystem' type='mdev' model='vfio-pci' >
      <source>
        <address uuid='e31d555b-9367-432d-87b3-482331bbd1f2'/>
      </source>
    </hostdev>
```

Figure 20 Adding one vGPU

Below is an example of adding two vGPUs with the UUIDs. It just adds another section that is same as the first one, but you need to following NVIDIA spec to make sure that your vGPU type is supported to be added to a single VM.

```
# virsh edit rhel8.5
    <hostdev mode='subsystem' type='mdev' model='vfio-pci' >
      <source>
        <address uuid='e31d555b-9367-432d-87b3-482331bbd1f2'/>
      </source>
</hostdev>
<hostdev mode='subsystem' type='mdev' model='vfio-pci' >
      <source>
        <address uuid='f8399521-e20b-4f44-b14f-4af09d594244'/>
      </source>
</hostdev>
```

Figure 21 Adding two vGPUs

3. Run command **virsh start <domain_name>** to start your Guest OS.

   After the Guest OS boots up successfully, you can log in with your root account and check a NVIDIA GPU on your Gust OS by command **lspci -vvvnnn|grep -i nvidia -A 15**, as listed below.

```
# lspci -vvvnnn|grep -i nvidia -A 15
07:00.0 VGA compatible controller [0300]: NVIDIA Corporation TU104GL [Tesla T4] [10de:1eb8]
(rev a1) (prog-if 00 [VGA controller])
        Subsystem: NVIDIA Corporation Device [10de:130d]
        Physical Slot: 0-6
        Control: I/O+ Mem+ BusMaster- SpecCycle- MemWINV- VGASnoop- ParErr- Stepping- SERR+
FastB2B- DisINTx-
        Status: Cap+ 66MHz- UDF- FastB2B- ParErr- DEVSEL=fast >TAbort- <TAbort- <MAbort-
>SERR- <PERR- INTx-
        Interrupt: pin A routed to IRQ 22
        Region 0: Memory at fa000000 (32-bit, non-prefetchable) [size=16M]
        Region 1: Memory at d0000000 (64-bit, prefetchable) [size=256M]
        Region 3: Memory at f8000000 (64-bit, non-prefetchable) [size=32M]
        Capabilities: [d0] Vendor Specific Information: Len=1b <?>
        Capabilities: [68] MSI: Enable- Count=1/1 Maskable- 64bit+
                Address: 0000000000000000  Data: 0000
        Kernel modules: nouveau
```

Figure 22 Output from lspci command

4. Check the physical GPU on your host OS by command **nvidia-smi** to see if there are some processes running on it. Refer to Figure 23 and compare it with the previous **nvidia-smi** output shown in Figure 11 on page 8. The bottom of the image shows a process is running now, that is, the vGPU we have added to the Guest OS.

```
[root@vGPUHost nvidia]# nvidia-smi
Thu Mar 17 16:43:39 2022
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 510.47.03    Driver Version: 510.47.03    CUDA Version: N/A      |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|                               |                      |               MIG M. |
|===============================+======================+======================|
|   0  Tesla T4            On   | 00000000:4B:00.0 Off |                    0 |
| N/A   39C    P8    16W /  70W |   1970MiB / 15360MiB |      0%      Default |
|                               |                      |                  N/A |
+-------------------------------+----------------------+----------------------+

+-----------------------------------------------------------------------------+
| Processes:                                                                  |
|  GPU   GI   CI        PID   Type   Process name                  GPU Memory |
|        ID   ID                                                   Usage      |
|=============================================================================|
|    0   N/A  N/A      9961    C+G   vgpu                             1888MiB |
+-----------------------------------------------------------------------------+
[root@vGPUHost nvidia]# _
```

Figure 23 Output from nvidia-smi showing running vGPUs

Installing the vGPU driver on the Linux guest OS

Through above preparations, now we have a vGPU on the guest OS. Guest OS can use NVIDIA vGPUs in the same manner as a physical GPU. Processes are running on a vGPU, the vGPU makes use of the physical GPU's engines, including the graphics (3D), video

decode/encode engine etc. To achieve this function, an NVIDIA driver should be loaded in the guest OS to provide direct access to the GPU for performance.

The following steps describe how to install the NVIDIA vGPU driver in a Linux Guest OS. If there are related errors, the NVIDIA vGPU software graphics driver for Linux needs to be installed.

Complete the following prerequisites before beginning:

1. Compiler toolchain
2. Kernel headers
3. If Dynamic Kernel Module Support (DKMS) is enabled, ensure that the dkms package is installed.
4. If the screen shows that you are in X server mode when installing the vGPU driver, use command **`init 3`** to exit the X server by transitioning to runlevel 3.

The steps to install the vGPU driver are as follows:

1. Copy the NVIDIA vGPU software Linux driver package you've download from NVIDIA to Guest OS as described in "Download NVIDIA GPU driver for your device" on page 4. The driver we want to use is nvidia-linux-grid-510-510.47.03-1.x86_64.rpm.
2. Use rpm tool to install this rpm package and its dependent packages, as listed below.

```
[root@localhost home]# rpm -ivh dkms-3.0.3-1.el8.noarch.rpm
warning: dkms-3.0.3-1.el8.noarch.rpm: Header V4 RSA/SHA256 Signature, key ID 2f86d6a1: NOKEY
Verifying...                          ################################# [100%]
Preparing...                          ################################# [100%]
Updating / installing...
   1:dkms-3.0.3-1.el8                 ################################# [100%]
[root@localhost home]# ls
```

```
[root@localhost home]# rpm -ivh nvidia-linux-grid-510-510.47.03-1.x86_64.rpm
Verifying...                          ################################# [100%]
Preparing...                          ################################# [100%]
Updating / installing...
   1:nvidia-linux-grid-510-1:510.47.03################################# [100%]
Running Post-install scripts
Loading new nvidia-510.47.03 DKMS files...
Building for 4.18.0-348.el8.x86_64
Building initial module for 4.18.0-348.el8.x86_64
Done.
```

Figure 24 Installing the driver

3. Restart the guest OS. The system should not be loaded with the nouveau module so far. The following command confirms if the NVIDIA's proprietary driver is used in the system. You should see the line `kernel driver in use` is changed from `nouveau` to `nvidia`.

```
# lspci |grep -i nvidia
07:00.0 VGA compatible controller: NVIDIA Corporation TU104GL [Tesla T4]
(rev a1)
# lspci -vvvnnn -s 07lspci:00.0|grep -i kernel
        Kernel driver in use: nvidia
        Kernel modules: nouveau, nvidia_drm, nvidia
```

Figure 25 Confirming the NVIDIA driver is operational

If the vGPU driver is not in a rpm format, then usually the format is a POSIX shell script executable, for example NVIDIA-Linux-x86_64-510.47.03-grid.run. In this situation, perform the following steps instead the above. These steps are the same as the NVIDIA vGPU manger software installation.

1. Run the following commands to install the vGPU driver of NVIDIA. The driver cannot be installed if the X server is running on the system, so ensure that the system starts in text mode (runlevel 3). Ensure that this driver is saved on the local disk of the target system. Installing from an external device, such as a flash drive, will cause known issues such as an installation failure.

```
# init 3
# sh nvidia_filename.run
```

Figure 26 Install the vGPU driver

2. Create a config file with any name and the name's suffix is `.conf` in the directory /etc/modprobe.d/, such as creating `/etc/modprobe.d/nvidia-blacklist.conf` file and adding the following two lines to it, so the content of the file is the following:

```
# cat /etc/modprobe.d/nvidia-blacklist.conf
blacklist nouveau
options nouveau modeset=0
```

Figure 27 Contents of /etc/modprobe.d/nvidia-blacklist.conf

3. Back up the current `initramfs` and build a new one for the current kernel as follows. If you would like to regenerate initramfs for all installed kernel versions, you can run command **`dracut --regenerated-all --force`**.

```
# mv /boot/initramfs-$(uname -r).img /boot/initramfs-$(uname -r)-nouveau.img
# dracut /boot/initramfs-$(uname -r).img $(uname -r)
```

Figure 28 Regenerate initramfs

Deleting a vGPU on Linux with KVM

For the vGPUs that you want to delete, perform this task in a Linux command shell on the Linux with KVM hypervisor host.

1. Before beginning, shut down the Guest OS which the vGPU is assigned in by command **`virsh shutdown <domain_name>`**, for example, the domain_name is `rhel8.5`, execute **`virsh shutdown rhel8.5`** to shut down the Guest OS.
2. Get the domain, bus, slot, and function of the GPU where the vGPU that you want to delete resides in, such as `0000:4b:00.0` on this example.
3. Show all the vGPU devices on you physical GPU which you get in step 2 using the following command and record the one you want to remove from you physical GPU, for example device `e31d555b-9367-432d-87b3-482331bbd1f2`.

```
# ls /sys/class/mdev_bus/0000\:4b\:00.0/ | grep -E
"[[:xdigit:]]{8}(-[[:xdigit:]]{4}){3}(-[[:xdigit:]]{12})"
e31d555b-9367-432d-87b3-482331bbd1f2
f8399521-e20b-4f44-b14f-4af09d594244
```

Figure 29 List vGPU devices

4. Write the value 1 to the remove file in directory e31d555b-9367-432d-87b3-482331bbd1f2 as follows:

```
# echo 1 > /sys/class/mdev_bus/0000\:4b\:00.0/e31d555b-9367-432d-87b3-482331bbd1f2
/remove
```

Figure 30 Removing the vGPU

Prerequisites for using NVIDIA vGPU on the NVIDIA Ampere architecture

If you are using GPUs based on the NVIDIA Ampere architecture, the following BIOS settings should be enabled on the server platform:

1. I/O Memory Management Unit (IOMMU), also known as Intel VT-d and AMD-Vi

   I/O Memory Management Unit (IOMMU) is a generic name for Intel VT-d and AMD-Vi. Some NVIDA vGPUs are only available on hardware platforms supporting either Intel Virtualization Technology for Directed I/O (VT-d) or AMD I/O Virtualization Technology (AMD-Vi). The Intel VT-d and AMD-Vi help NVIDIA vGPUs directly assign to a VM.

   IOMMU must be enabled in the host UEFI, so you should make sure that the IOMMU option is enabled in the host UEFI

2. Single Root I/O Virtualization (SR-IOV).

   The NVIDIA vGPU that supports SR-IOV resides on a physical GPU that supports SR-IOV, such as a GPU based on the NVIDIA Ampere architecture. Perform command **`nvidia-smi -q |grep -i "vgpu mode"`** to check whether the GPU supports SR-IOV. On that type of GPUs, you can create vGPUs based on that SR-IOV devices. The steps are similar to the ones described in “Create NVIDIA vGPUs” on page 8, except that the PCI bus number you specify is a virtual function bus number instead of a physical GPU PCI bus number.

IOMMU and SRIOV settings on Intel systems

VT-d stands for Intel Virtualization Technology for Directed I/O and should not be confused with VT-x Intel Virtualization Technology. VT-x allows one hardware platform to function as multiple “virtual” platforms. However, VT-d improves security and reliability of the systems and improves performance of I/O devices in virtualized environments.

The steps to activate the Intel IOMMU are as follows:

1. In System Setup (F1 at boot), enter the UEFI System Configuration and Boot Management for enabling the Intel VT for Directed I/O (VT-d).
2. From the BIOS setup menu navigate to **System Settings** → **Devices and I/O ports** → **Intel VT for Directed I/O (VT-d)** to enable the Intel IOMMU and SRIOV as shown Figure 31.

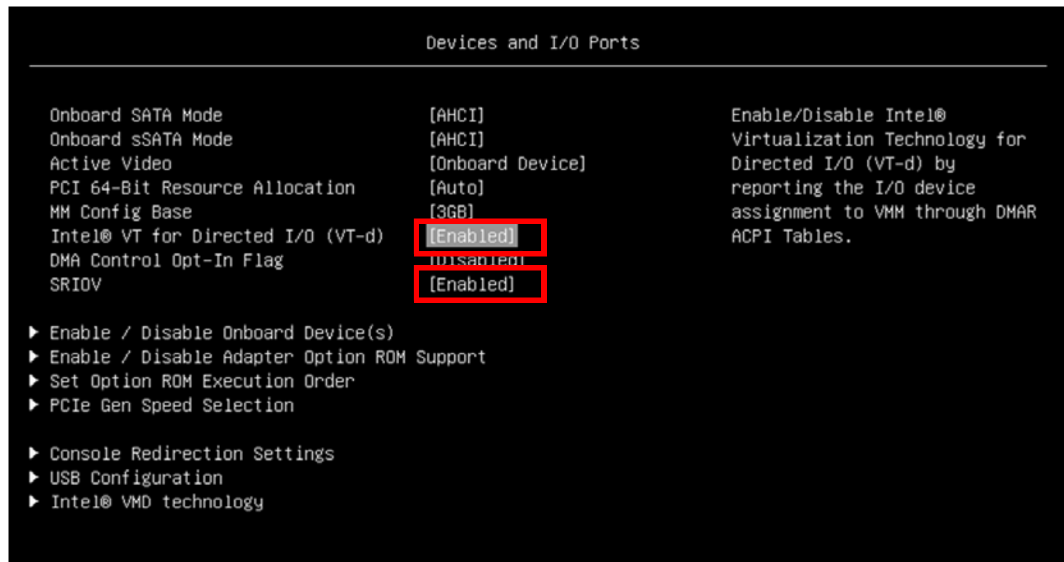

Figure 31 Enabling VT-d and SRIOV in System Setup (Intel processors)

3. Save and exit the BIOS setup menu, and then boot to the Linux OS.
4. In the OS, ensure that the IOMMU is enabled using the command **`dmesg|grep DMAR`**. If the output of the commands includes the following string, it means system has enabled VT-d by reporting the I/O device assignment to VMM through DMAR (DMA Remapping) ACPI table:

```
DMAR: IOMMU enabled
```

IOMMU and SRIOV settings on AMD systems

The AMD IOMMU specifications are required to use PCI device assignment in Linux OS. These specifications must be enabled in the BIOS. Some Lenovo systems disable these specifications by default.

The steps to activate the AMD IOMMU are as follows:

1. In System Setup (F1 at boot), enter the UEFI System Configuration and Boot Management for enabling the AMD IOMMU.
2. From the BIOS setup menu, navigate to **System Settings** > **Devices and I/O ports** > **IOMMU** to enable the AMD IOMMU and SRIOV as shown in Figure 32.

Figure 32 Enabling VT-d and SRIOV in System Setup (AMD processors)

3. Save and exit the BIOS setup menu, and then boot to the Linux OS.
4. In the OS, ensure that the IOMMU is enabled using the command **`dmesg|grep AMD-Vi`**. If the output of the commands includes the following string, it means system has enabled AMD IOMMU:

   `AMD-Vi: Interrupt remapping enabled`

Enabling IOMMU host kernel support

Currently, most released operating systems set IOMMU off by default. To enable the I/O Memory Management Unit (IOMMU) on the host OS, follow these procedures:

1. Edit the host kernel boot command line

   For an Intel VT-d system, IOMMU is activated by adding parameter `intel_iommu=on` to the kernel command line. For an AMD-Vi system, the option needed is `amd_iommu=on`.

   To enable this option, edit or add the GRUB_CMDLINX_LINUX line to the /etc/default/grub configuration file as follows:

```
# cat /etc/default/grub
GRUB_TIMEOUT=5
GRUB_DISTRIBUTOR="$(sed 's, release .*$,,g' /etc/system-release)"
GRUB_DEFAULT=saved
GRUB_DISABLE_SUBMENU=true
GRUB_TERMINAL_OUTPUT="console"
GRUB_CMDLINE_LINUX="crashkernel=auto resume=/dev/mapper/rhel-swap
rd.lvm.lv=rhel/root rd.lvm.lv=rhel/swap rhgb quiet intel_iommu=on "
GRUB_DISABLE_RECOVERY="true"
GRUB_ENABLE_BLSCFG=true
#
```

Figure 33 grup config with intel_iommu=on added

   For AMD platform, the parameter `intel_iommu=on` should be replaced by `amd_iommu=on`.

2. Regenerate the grub2 config file

To apply changes to the kernel command line, regenerate the boot loader configuration using the **grub2-mkconfig** command, and check whether the changes are effective using the command **grubby --info=0**.

3. Reboot the host OS

   For the changes to take effect to kernel driver, reboot the host machine and use the command **dmesg|grep iommu** to see one of the following messages:

```
Adding to iommu group 0
iommu: Default domain type: Passthrough (set via kernel command line)
```

Resources

- Release Notes - Virtual GPU Software R510 for Red Hat Enterprise Linux with KVM:

  https://docs.nvidia.com/grid/latest/grid-vgpu-release-notes-red-hat-el-kvm/index.html
- Installing and Configuring NVIDIA Virtual GPU Manager

  https://docs.nvidia.com/grid/latest/grid-vgpu-user-guide/index.html#installing-configuring-grid-vgpu
- Creating your NVIDIA Enterprise Account

  https://docs.nvidia.com/grid/latest/grid-software-quick-start-guide/index.html#creating-nvidia-enterprise-account
- NVIDIA vGPU Certified Servers

  https://www.nvidia.com/en-us/data-center/resources/vgpu-certified-servers/

Authors

Xiaochun Li is a Linux engineer at the Lenovo Data Center Group in Beijing, China. He specializes in the development of Linux kernel storage and memory management, as well as kernel DRM. Before joining Lenovo, he worked in INSPUR as an OS engineer for several years. With ten years of industry experience, he now focuses on Linux kernel RAS, storage, security, and virtualization.

Thanks to the following people for their contributions to this project:

- JiaJia, Lenovo Test Engineer for Linux Enablement
- Adrian Huang, Lenovo OS engineer
- Gary Cudak, Lenovo OS Architect
- David Watts, Lenovo Press

Notices

Lenovo may not offer the products, services, or features discussed in this document in all countries. Consult your local Lenovo representative for information on the products and services currently available in your area. Any reference to a Lenovo product, program, or service is not intended to state or imply that only that Lenovo product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any Lenovo intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any other product, program, or service.

Lenovo may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

Lenovo (United States), Inc.
1009 Think Place - Building One
Morrisville, NC 27560
U.S.A.
Attention: Lenovo Director of Licensing

LENOVO PROVIDES THIS PUBLICATION “AS IS” WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some jurisdictions do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. Lenovo may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

The products described in this document are not intended for use in implantation or other life support applications where malfunction may result in injury or death to persons. The information contained in this document does not affect or change Lenovo product specifications or warranties. Nothing in this document shall operate as an express or implied license or indemnity under the intellectual property rights of Lenovo or third parties. All information contained in this document was obtained in specific environments and is presented as an illustration. The result obtained in other operating environments may vary.

Lenovo may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Any references in this publication to non-Lenovo Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this Lenovo product, and use of those Web sites is at your own risk.

Any performance data contained herein was determined in a controlled environment. Therefore, the result obtained in other operating environments may vary significantly. Some measurements may have been made on development-level systems and there is no guarantee that these measurements will be the same on generally available systems. Furthermore, some measurements may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

© Copyright Lenovo 2022. All rights reserved.
Note to U.S. Government Users Restricted Rights -- Use, duplication or disclosure restricted by Global Services Administration (GSA) ADP Schedule Contract

This document was created or updated on May 10, 2022.

Send us your comments via the **Rate & Provide Feedback** form found at
http://lenovopress.com/lp1585

Trademarks

Lenovo and the Lenovo logo are trademarks or registered trademarks of Lenovo in the United States, other countries, or both. These and other Lenovo trademarked terms are marked on their first occurrence in this information with the appropriate symbol (® or ™), indicating US registered or common law trademarks owned by Lenovo at the time this information was published. Such trademarks may also be registered or common law trademarks in other countries. A current list of Lenovo trademarks is available from
https://www.lenovo.com/us/en/legal/copytrade/.

The following terms are trademarks of Lenovo in the United States, other countries, or both:

Lenovo® Lenovo(logo)® ThinkSystem™

The following terms are trademarks of other companies:

Intel, and the Intel logo are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries.

Linux is a trademark of Linus Torvalds in the United States, other countries, or both.

Windows, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

Other company, product, or service names may be trademarks or service marks of others.