Cale Rogers Follow
Internet fan boy
Sep 18, 2016 · 7 min read

# GPU Virtualization with KVM / QEMU

This is part two of our building a deep learning machine series. You can find the other posts here:

1. Building a Deep Learning Box
2. GPU Virtualization with KVM / QEMU
3. Installing Nvidia, Cuda, CuDNN, TensorFlow and Keras

If you're not using virtualization (which is possible) you can skip this entirely.

Hardware is a *critical* factor when working with GPU virtualization and this post makes hardware assumptions based on the component list from part one. Your mileage may vary along the way. There are several customizations you have to make to your system based on the hardware you have, and that is not even taking into account the varying differences between linux distributions.

**End goal**: a physical server (host) that can spin up virtual machines (guests) each with a dedicated GPU to use for ML computations.

## OS and Hypervisor Selection

We ultimately ended up choosing Ubuntu 16.04 server edition for our physical host, Ubuntu 14.04 for our guest, and KVM for our hypervisor. These are the building blocks and the rest of the guide assumes you are using them. The reasoning behind choosing 16.04 for our host was due to my familiarity with Debian based OS's and having the latest KVM / QEMU packages which was necessary for passing through GPUs.

I had worked with VMware and VirtualBox before but this was my first dive into virtualization for a data center. First I started researching the options, which I narrowed down to Xen or KVM just based on popularity, being open source, and the richness of the communities. I found that while both could theoretically work, KVM had better support for GPU virtualization so it was ultimately a no brainer.

## Other Guides

I learned a lot from forum posts (here is mine when I ran into issues) and how-to's out there on the internet. A 5 part blog series by Alex Williamson was the most helpful and informative. Looking back, this was all I needed for getting passthrough working with my hardware / setup. He is the creator and maintainer of VFIO which is the latest method for PCI passthrough so you should definitely give it a read. However, his series was based on Red Hat (he works there after all) so my much shorter guide has some differences.

Some key technologies for all of this to work are **VFIO** and **IOMMU**.

The VFIO driver is an IOMMU/device and agnostic framework for exposing direct device access to userspace, in a secure, IOMMU protected environment. In other words, this allows safe, non-privileged, userspace drivers.

What this means is that your GPUs will be bound to the VFIO driver, not an Nvidia driver. You can read more about it here.

IOMMU is a system specific IO mapping mechanism and can be used with most devices. In the Intel world this is known as VT-d.

First and foremost you need to go into the BIOS and enable VT-x and VT-d which is required for any of this to work. Once that is complete and you have Ubuntu installed, we can begin setting up KVM and installing the virtualization dependencies.

## Install KVM / QEMU / OVMF packages

```
sudo apt-get install qemu-kvm libvirt-bin bridge-utils
virtinst ovmf qemu-utils
```

**Note:** OVMF is a UEFI firmware

## Enable IOMMU

Edit the GRUB file at /etc/default/grub:

```
sudo vim /etc/default/grub
```

```
# Make this line look like this
GRUB_CMDLINE_LINUX="intel_iommu=on iommu=pt
```

```
rd.driver.pre=vfio-pci video=efifb:off"
```

```
sudo update-grub
```

**Note:** the “video=efifb:off” part was a fix for getting the primary GPU to work with passthrough and is only necessary depending on your system. I go into a little more detail about this later.

Edit the initramfs at /etc/initram-fs/modules to ensure it has VFIO modules loaded on boot:

```
sudo vim /etc/initram-fs/modules
```

```
# Add to file
vfio
vfio_iommu_type1
vfio_pci
vfio_virqfd
```

```
sudo update-initramfs -u
```

Create a new module file located at /etc/modprobe.d/local.conf. This step binds the video cards to VFIO on boot so they are not claimed by the host OS (this script also binds VFIO post boot):

```
sudo vim /etc/modprobe.d/local.conf
```

```
# Add to file
options vfio-pci ids=10de:1b80,10de:10f0
options vfio-pci disable_vga=1
```

Reboot the server.

**Note:** 10de:1b80, 10de:10f0 are specific to our GPU hardware (GTX 1080s). These numbers refer to the video card and onboard audio. You can find your specific model / vendor numbers by running:

```
lspci -nnk | grep -i nvidia
```

```
4b:00.0 VGA compatible controller [0300]: NVIDIA Corporation
Device [10de:1b80] (rev a1)
4b:00.1 Audio device [0403]: NVIDIA Corporation Device
[10de:10f0] (rev a1)
```

At this point your physical host should be setup and ready to passthrough GPUs to guest VMs. You can verify IOMMU and VFIO are working by running the following:

```
sudo find /sys/kernel/iommu_groups/ -type l
```

```
# Your output should be a long listing of lines like this
```

```
/sys/kernel/iommu_groups/0/devices/0000:ff:0b.0
/sys/kernel/iommu_groups/0/devices/0000:ff:0b.1
/sys/kernel/iommu_groups/0/devices/0000:ff:0b.2
/sys/kernel/iommu_groups/1/devices/0000:ff:0c.0
...
...
```

```
lspci -nnk
```

```
# Find your VGA controllers, it should look similar to this
```

```
4b:00.0 VGA compatible controller [0300]: NVIDIA Corporation
Device [10de:1b80] (rev a1)
    Subsystem: ASUSTeK Computer Inc. Device [1043:8591]
    Kernel driver in use: vfio-pci
```

. . .

## Creating the VMs

The next challenge is actually creating a VM that can take advantage of GPU passthrough. There are a lot of different ways to go about this, including using the GUI program, virt-manager, which Alex Williamson's guide uses. I personally used the virt-install command within a script which installs Ubuntu 14.04 based on a Debian preseed file to automate the process. You don't actually need to use the preseed file and you may or may not need to modify it to your needs.

You can also specify which GPU you want to passthrough right in the virt-install command by adding the --host-device flag similar to the following

```
sudo virt-install \
--name ubuntu-vm \
--boot uefi \
--machine q35 \
--host-device 4b:00.0 --host-device 4b:00.1 \
...
...
```

but I chose to keep it agnostic and add those details after VM creation as this better fit our flow for spinning VMs up and down consistently and in an automated way.

## Add a GPU to a VM

After VM creation the end result is an XML file with its defined attributes such as storage, RAM, CPUs, etc. There are a few necessary additions to the XML file you have to make using the “virsh edit” command to add a GPU.

Whenever you edit the VM XML file you need to ensure that your VM is shutdown (you can do that with the virsh shutdown or virsh destroy command, the latter being a forced power off). If you don’t you could run into issues with libvirt / KVM which may result in needing to reboot your host.

For a Nvidia specific issue you need to add the following code snippet:

```
virsh edit vm-name

# Add to file within <features> </features> tags
```

```
<kvm>
  <hidden state='on'/>
</kvm>
```

This code hides the fact that this is a VM sitting on a hypervisor from the guest OS / Nvidia allowing you to properly install graphics drivers within the guest.

Then you need to actually add the code that assigns the PCI device (in our case a GPU) to the VM, an example of what that looks like:

```
# Video card
```

```
<hostdev mode='subsystem' type='pci' managed='yes'>
  <source>
    <address domain='0x0000' bus='0x4b' slot='0x00'
function='0x0'/>  </source>
    <address type='pci' domain='0x0000' bus='0x00'
slot='0x05' function='0x0'/>
</hostdev>
```

```
# Onboard audio
```

```
<hostdev mode='subsystem' type='pci' managed='yes'>
  <source>
    <address domain='0x0000' bus='0x4b' slot='0x00'
function='0x1'/> </source>
    <address type='pci' domain='0x0000' bus='0x00'
slot='0x06' function='0x0'/>
</hostdev>
```

**Note:** the part to be aware of here is the “bus=0x4b” as this relates to which PCI device you want to passthrough.

## Primary GPU Workaround

Another quirk that needs to be addressed is only necessary if you are passing through your primary GPU but interesting nonetheless. This behavior occurs with the GTX 1080, but did not with a Geforce 210 in my testing, so your results may vary.

Like motherboards, GPUs have their own BIOS (aka ROM) and when the computer boots the primary GPU is using a “shadowed” copy of the ROM file. This causes issues when doing passthrough. To workaround this you need to get a copy of a non-shadowed ROM file which is specific to the GPU model. In our case we had other non-primary GPUs we could dump the ROM from. You can also try 3rd party websites that provide ROMs but I did not have success with those.

First unbind a non-primary GPU from vfio-pci (if it is bound), for example:

```
echo “0000:4b:00.0” | sudo tee /sys/bus/pci/drivers/vfio-
pci/unbind
```

Then dump the ROM contents to a file:

```
echo 0 | sudo tee
/sys/devices/pci0000:00/0000:00:03.0/0000:4b:00.0/rom
```

```
sudo cat
/sys/devices/pci0000:00/0000:00:03.0/0000:4b:00.0/rom >
gpu.rom
```

```
echo 1 | sudo tee
/sys/devices/pci0000:00/0000:00:03.0/0000:4b:00.0/rom
```

**Note:** The echo 0 and echo 1 basically just switches the ROM into a readable state and then back again if you're curious.

After you have a good ROM file you need to add this code to your VMs XML in the <hostdev> definition of the GPU which added earlier:

```
<rom file='/path/to/rom'/>
```

## Full Example

A full example of a VM XML file is located here.

At this point you should be able to SSH or console into the VM you have created and install Nvidia drivers. I used version 367.35 at the time of this writing, which you can download here, and run nvidia-smi to verify GPUs are properly detected by the system. It should look something like this:

```
root@base:~# nvidia-smi
Sat Sep 17 17:09:52 2016
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 367.35                 Driver Version: 367.35                    |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|===============================+======================+======================|
|   0  GeForce GTX 1080    Off  | 0000:00:05.0     Off |                  N/A |
|  0%   38C    P0    37W / 180W |      0MiB /  8113MiB |      0%      Default |
+-------------------------------+----------------------+----------------------+

+-----------------------------------------------------------------------------+
| Processes:                                                       GPU Memory |
|  GPU       PID  Type  Process name                               Usage      |
|=============================================================================|
|  No running processes found                                                 |
+-----------------------------------------------------------------------------+
```

## Where to go for help?

VFIO mailing list
Ubuntu Virtualization forums