Single GPU Passthrough (for Nvidia GPUs) Arch Linux

Passing through your primary GPU from your host OS to a Windows 10 VM

Contents:

Introduction

This tutorial walks you through how to passthrough your boot GPU to a guest machine on Arch Linux, this tutorial will work on other distros.

Credits

[Chirjonit] for inspiring me to create a VM when I was still running Pop! OS.
Single-GPU-Passthrough

Hardware and Software

README.md

```
CPU: i5-4570
MB: HP Generic
RAM: 16GB
Main Storage: 240GB SSD
Secondary Storage: 500GB HDD
Game Storage: 2TB HDD
GPU: NVIDIA GTX 1650
```

Monitor:

```
MSI Optix G27C2 attached to NVIDIA GPU
```

Software:

Install of Arch Linux using KDE updated using `$ sudo pacman -Syu`
Downloaded Windows 10 (version 21H2) ISO

Preparation

Enable Virtualization in BIOS

The first and most important step is to make sure that virtualization is enabled in your bios.

```
Security
└──System Security
   └──Virtualization Technology (VTx/VTd) -> Enable

File
└──Save Changes and Exit
```

Enable IOMMU in Bootloader

GRUB

Open `/etc/default/grub` `$ sudo nano /etc/default/grub`

Find the line `GRUB_CMDLINE_LINUX_DEFAULT="..."`

Add `"... intel_iommu=on"`

Press ***CTRL+O*** to save and exit.

You should `reboot` your system now.

Systemd-Boot

Open `/boot/loaders/entries/arch.conf` `$ sudo nano /boot/loaders/entries/arch.conf`

Note: some users may have `/boot/loader/entries` instead of `/boot/loaders/entries`

Find the line `options ...`

Add `... intel_iommu=on`

Press ***CTRL+O*** to save and exit.

You should `reboot` your system now.

Check & Prepare your OS

Check for IOMMU support `$ sudo dmesg | grep IOMMU`

You should get an output similar to this:

```
[    0.045131] DMAR: IOMMU enabled
[    0.104081] DMAR-IR: IOAPIC id 8 under DRHD base  0xfed91000 IOMMU 1
[    0.237596] DMAR: IOMMU feature pgsel_inv inconsistent
[    0.237597] DMAR: IOMMU feature sc_support inconsistent
[    0.237598] DMAR: IOMMU feature pass_through inconsistent
[    0.325372] AMD-Vi: AMD IOMMUv2 functionality not available on this
system - This is not a bug.
```

Install all required packages:
`$ sudo pacman -S qemu libvirt dmidecode edk2-ovmf virt-manager dnsmasq iptables-nft`

Enable libvirtd `sudo systemctl enable libvirtd`

You should `reboot` your system now.

Check/list IOMMU groupings:

```
#!/bin/bash
shopt -s nullglob
for g in /sys/kernel/iommu_groups/*; do
    echo "IOMMU Group ${g##*/}:"
    for d in $g/devices/*; do
        echo -e "\t$(lspci -nns ${d##*/})"
    done;
done;
```

This is what I get, anything you want to passthrough to your VM has to be in it's own group. You don't need to passthrough things such as `PCI bridge` or `ISA bridge` as these are non-hardware entries.

```
IOMMU Group 0:
        00:00.0 Host bridge [0600]: Intel Corporation 4th Gen Core
Processor DRAM Controller [8086:0c00] (rev 06)
IOMMU Group 1:
        00:01.0 PCI bridge [0604]: Intel Corporation Xeon E3-1200 v3/4th
Gen Core Processor PCI Express x16 Controller [8086:0c01] (rev 06)
        01:00.0 VGA compatible controller [0300]: NVIDIA Corporation TU117
[GeForce GTX 1650] [10de:1f82] (rev a1)
        01:00.1 Audio device [0403]: NVIDIA Corporation Device [10de:10fa]
(rev a1)
IOMMU Group 10:
        00:1f.0 ISA bridge [0601]: Intel Corporation Q87 Express LPC
Controller [8086:8c4e] (rev 04)
        00:1f.2 SATA controller [0106]: Intel Corporation 8 Series/C220
Series Chipset Family 6-port SATA Controller 1 [AHCI mode] [8086:8c02]
(rev 04)
        00:1f.3 SMBus [0c05]: Intel Corporation 8 Series/C220 Series
Chipset Family SMBus Controller [8086:8c22] (rev 04)
IOMMU Group 2:
        00:02.0 Display controller [0380]: Intel Corporation Xeon E3-1200
v3/4th Gen Core Processor Integrated Graphics Controller [8086:0412] (rev
06)
IOMMU Group 3:
        00:03.0 Audio device [0403]: Intel Corporation Xeon E3-1200 v3/4th
Gen Core Processor HD Audio Controller [8086:0c0c] (rev 06)
IOMMU Group 4:
        00:14.0 USB controller [0c03]: Intel Corporation 8 Series/C220
Series Chipset Family USB xHCI [8086:8c31] (rev 04)
IOMMU Group 5:
        00:16.0 Communication controller [0780]: Intel Corporation 8
Series/C220 Series Chipset Family MEI Controller #1 [8086:8c3a] (rev 04)
        00:16.3 Serial controller [0700]: Intel Corporation 8 Series/C220
Series Chipset Family KT Controller [8086:8c3d] (rev 04)
IOMMU Group 6:
        00:19.0 Ethernet controller [0200]: Intel Corporation Ethernet
Connection I217-LM [8086:153a] (rev 04)
IOMMU Group 7:
        00:1a.0 USB controller [0c03]: Intel Corporation 8 Series/C220
```

```
Series Chipset Family USB EHCI #2 [8086:8c2d] (rev 04)
IOMMU Group 8:
        00:1b.0 Audio device [0403]: Intel Corporation 8 Series/C220
Series Chipset High Definition Audio Controller [8086:8c20] (rev 04)
IOMMU Group 9:
        00:1d.0 USB controller [0c03]: Intel Corporation 8 Series/C220
Series Chipset Family USB EHCI #1 [8086:8c26] (rev 04)
```

This is what I want to passthrough:

- My back USB ports (IOMMU Group 4)
- My ethernet controller (IOMMU Group 6)
- My GPU (IOMMU Group 1)

Note that you can passthrough individual USB devices and don't have to passthrough the entire controller like I do

If your groups are not neatly divided you can use the ACS patch, although this can be risky.

Hooks

We will begin preparation by creating the hooks and associated scripts.

Create the hooks directory `$ sudo mkdir /etc/libvirt/hooks`

Get the hook helper with:

```
$ sudo wget 'https://raw.githubusercontent.com/PassthroughPOST/VFIO-Tools/master/libvirt_hooks/qemu' -O /etc/libvirt/hooks/qemu
```

Make the qemu file executable `$ sudo chmod +x /etc/libvirt/hooks/qemu`

Create required folders:

```
$ sudo mkdir /etc/libvirt/hooks/qemu.d /etc/libvirt/hooks/qemu.d/win10
/etc/libvirt/hooks/qemu.d/win10/prepare
/etc/libvirt/hooks/qemu.d/win10/prepare/begin
/etc/libvirt/hooks/qemu.d/win10/release
/etc/libvirt/hooks/qemu.d/win10/release/end
```

Ensure that your folder/file tree are correct:

```
$ sudo pacman -S tree
$ tree /etc/libvirt/hooks
```

You should see something like this:

```
├── qemu
└── qemu.d
    └── win10
        ├── prepare
        │   └── begin
        └── release
            └── end
```

Start nano via `$ sudo nano /etc/libvirt/hooks/qemu.d/win10/prepare/begin/start.sh`

Paste this script into the file via ***CTRL+SHIFT+V***

```
#!/bin/bash
## Helpful to read output when debugging
set -x

# Stop display manager
systemctl stop display-manager.service
## Uncomment the following line if you use GDM
#killall gdm-x-session

# Unbind VTconsoles
echo 0 > /sys/class/vtconsole/vtcon0/bind
echo 0 > /sys/class/vtconsole/vtcon1/bind

# Unbind EFI-Framebuffer
echo efi-framebuffer.0 > /sys/bus/platform/drivers/efi-framebuffer/unbind

# Avoid a Race condition by waiting 2 seconds. This can be calibrated to
be shorter or longer if required for your system
sleep 2

# Unload all Nvidia drivers
modprobe -r nvidia_drm
modprobe -r nvidia_modeset
modprobe -r nvidia_uvm
modprobe -r nvidia

## Load vfio
modprobe vfio
modprobe vfio_iommu_type1
modprobe vfio_pci
```

Convert `start.sh` to executable using `$ sudo chmod +x /etc/libvirt/hooks/qemu.d/win10/prepare/begin/start.sh`

Press ***CTRL+O*** to save the file

Start nano again via `$ sudo nano /etc/libvirt/hooks/qemu.d/win10/release/end/stop.sh`

```
#!/bin/bash
set -x

## Unload vfio
modprobe -r vfio_pci
modprobe -r vfio_iommu_type1
modprobe -r vfio

# Rebind VT consoles
echo 1 > /sys/class/vtconsole/vtcon0/bind
echo 1 > /sys/class/vtconsole/vtcon1/bind

nvidia-xconfig --query-gpu-info > /dev/null 2>&1
echo "efi-framebuffer.0" > /sys/bus/platform/drivers/efi-framebuffer/bind

modprobe nvidia_drm
modprobe nvidia_modeset

modprobe nvidia_uvm
modprobe nvidia

# Restart Display Manager
systemctl start display-manager.service
```

Save again using ***CTRL+O***

Convert `stop.sh` to executable using `$ sudo chmod +x /etc/libvirt/hooks/qemu.d/win10/release/end/stop.sh`

Check that your folders/files are in the correct order using `tree /etc/libvirt/hooks`

You should see something like this:

```
├── qemu
└── qemu.d
    └── win10
        ├── prepare
        │   └── begin
        │       └── start.sh
        └── release
            └── end
                └── stop.sh
```

Thats it for the scripts.

Setting up a VM

Open **Virtual Machine Manager (virt-manager)**

Go to **Edit**, then **Preferences** and tick **Enable XML Settings** under the **General** tab.

Click on **Create a new virtual machine**, select **local install**.

Click **Browse** and select your Windows 10 ISO file. It is recommended that you do not rename your Windows 10 ISO file - this way **Virtman** recognises the ISO as being a Windows 10 OS and will select the operating system automatically. If it doesn't detect it, uncheck the **Automatically detect** check box and just start typing 'Windows' and it will show Windows 10 as an option.

Select the amount of RAM you would like to passthrough.

Select your storage option and size.

Make sure that the name of the VM is `win10` exactly.

Tick **Customize configuration before install** then click finish.

Change firmware to **OVMF_CODE**

Go to **CPUs**

Click **Topology**

Tick **Manually set CPU topology**

Select the ammount of cores and threads you want for your VM.

I will be using 4 cores 1 threads in this example.

Click **Begin Installation**

Patching your VBIOS

Download your VBIOS from (https://www.techpowerup.com/vgabios/)

If you can't find your vbios here then continue to **Dumping VBIOS with NVFLASH**, otherwise continue to **Patching VBIOS**

Dumping VBIOS with NVFLASH

Download NVFlash (https://www.techpowerup.com/download/nvidia-nvflash/)

Extract NVFlash.

Switch to another TTY console with ***CTRL+ALT+FN+NUM***

Stop your display manager with `$ sudo systemctl stop display-manager.service`

Unload nvidia drivers with `$ sudo rmmod nvidia nvidia_drm nvidia_modeset nvidia_uvm`

CD to the x64 directory of NVFlash `$ cd /path/to/nvflash/x64`

Make NVFlash executable with `$ sudo chmod +x nvflash`

Run NVFlash `$ sudo ./nvflash --save /path/to/save/vbios/bios.rom`

Load nvidia drivers with `$ sudo modprobe nvidia nvidia_drm nvidia_modeset nvidia_uvm`

Start your display manager with `$ sudo systemctl start display-manager.service`

Patching VBIOS

Open `vbios.rom` with a hex editor of your choice.

Scroll down until you find **VIDEO** or search for it.

Delete everything before the `U`.

Save the vbios to another file.

Make a directory called `vgabios` in `/usr/share` `$ sudo mkdir /usr/share/vgabios`

Copy the patched romfile to `/usr/share/vgabios` `$ sudo cp /path/to/patched/vbios/patch.rom /usr/share/vgabios/patch.rom`

Note if there's nothing before the `U` this means your vbios is already patched, you can continue to **GPU Passthrough Settings and Setup**

GPU Passthrough Settings and Setup

Open your virtual machine in `Virtual Machine Manager` ***do not*** run it yet.

If you're passing through your network device remove `NIC`

Remove **USB Redirector 1** and **USB Redirector 2**

Remove **Sound ich9**

Remove **Channel spice**

Go to **Overview**

Scroll down until you find this:

```
<graphics type="spice" autoport="yes">
      <listen type="address"/>
      <image compression="off"/>
</graphics>
```

Delete this.

Delete `<audio id="1" type="spice"/>`

Scroll back up to the top and look for this:

```
<features>
  ...
</features>
```

In `<hyperv mode="custom">` add `<vendor_id state="on" value="kvm hyperv"/>`

After `</hypverv>` add this:

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

Click **Apply**.

Remove **Video QXL**

Starting VM with GPU Passthrough

Go to Boot Options and make sure your CDROM is ticked.

Click start to boot your virtual machine.

Releases

No releases published

Packages

No packages published