Heiko's Blog

Just some thoughts

Running Windows 10 on Linux using KVM with VGA Passthrough

The Need

You want to use Linux as your main operating system, but still need Windows for certain applications unavailable under Linux. You need top notch (3D) graphics performance under Windows that you can't get from VirtualBox or similar virtualization solutions. And you do not want to dual-boot into Linux or Windows. In that case read on.

Many modern CPUs have built-in features that improve the performance of virtual machines (VM), up to the point where virtualized systems are indistinguishable from non-virtualized systems. This allows us to create virtual machines on a Linux host platform without compromising performance of the (Windows) guest system.

The Solution

In the tutorial below I describe how to install and run Windows 10 as a KVM virtual machine on a Linux Mint or Ubuntu host. The tutorial uses a technology called VGA passthrough which provides near-native graphics performance in the VM. I've been doing VGA passthrough since summer 2012, first running Windows 7 on a Xen hypervisor, switching to KVM and Windows 10 in December 2015. The performance – both graphics and computing – under Xen and KVM has been nothing less than stellar!

The tutorial below will only work with suitable hardware! If your computer does not fulfill the basic hardware requirements outlined below, you won't be able to make it work.

The tutorial is not written for the beginner! I assume that you do have some Linux background, at least enough to be able to restore your system when things go wrong.

I am also providing links to other, similar tutorials that might help. Last not least, you will find links to different forums and communities where you can find further information and help.

Note: The tutorial was originally posted on the Linux Mint forum. However, limitations in terms of length and content (no. of images allowed) finally compelled me to host it on my own blog site.

Disclaimer

All information and data provided in this tutorial is for informational purposes only. I make no representations as to accuracy, completeness, currentness, suitability, or validity of any information in this tutorial and will not be liable for any errors, omissions, or delays in this information or any losses, injuries, or damages arising from its use. All information is provided on an as-is basis.

You are aware that by following this tutorial you may risk the loss of data, or may render your computer inoperable. Backup your computer! Make sure that important documents/data are accessible elsewhere in case your computer becomes inoperable.

For a glossary of terms used in the tutorial, see Glossary of Virtualization Terms.

Tutorial

Note for Ubuntu users: My tutorial uses the “xed” command found in Linux Mint Mate to edit documents. You will have to replace it with “gedit” or whatever editor you use in Ubuntu/Xubuntu/Lubuntu…

Table of Contents

Part 1 – Hardware Requirements

For this tutorial to succeed, your computer hardware must fulfill all of the following requirements:

IOMMU support

In Intel jargon its called ***VT-d***. AMD calls it variously ***AMD Virtualization***, ***AMD-V***, or ***Secure Virtual Machine (SVM)***. Even ***IOMMU*** has surfaced. If you plan to purchase a new PC/CPU, check the following websites for more information:

- Intel processors with ACS support –http://vfio.blogspot.com/2015/10/intel-processors-with-acs-support.html
- Wikipedia – https://en.wikipedia.org/wiki/List_of_IOMMU-supporting_hardware
- Intel – http://ark.intel.com/Search/FeatureFilter?productType=processors&VTD=true
- AMD – http://products.amd.com/en-us and check the processor specs.

Unfortunately IOMMU support, specifically ACS support, varies greatly between different Intel CPUs and CPU generations. Generally speaking, Intel provides better ACS or device isolation capabilities for its Xeon and LGA2011 high-end desktop CPUs than for other VT-d enabled CPUs. The first link above provides a non-comprehensive list of Intel CPUs with good IOMMU support. When I built my PC, I purchased the components specifically for supporting VGA passthrough.

Most PC / motherboard manufacturers disable IOMMU by default. You will have to enable it in the BIOS. To check your current CPU / motherboard IOMMU support and enable it, do the following:

1. Reboot your PC and enter the BIOS setup menu (usually you press F2, DEL, or similar during boot to enter the BIOS setup).
2. Search for IOMMU, VT-d, SVM, or "virtualization technology for directed IO" or whatever it may be called on your system. **Turn on VT-d / IOMMU**.
3. Save and Exit BIOS and boot into Linux.
4. Edit the /etc/default/grub file (you need root permission to do so). Open a terminal window (Ctrl+Alt+T) and enter (copy/paste): `gksudo xed /etc/default/grub` Here is my /etc/default/grub file before the edit:

```
GRUB_DEFAULT=0

#GRUB_HIDDEN_TIMEOUT=10

#GRUB_HIDDEN_TIMEOUT_QUIET=true

GRUB_TIMEOUT_STYLE=countdown
```

```
GRUB_TIMEOUT=0
GRUB_DISTRIBUTOR=`lsb_release -i -s 2> /dev/null || echo Debian`
GRUB_CMDLINE_LINUX_DEFAULT=”quiet nomodeset”
GRUB_CMDLINE_LINUX=””
```

The entry we are looking for is `GRUB_CMDLINE_LINUX_DEFAULT=”quiet nomodeset”`. We need to remove `nomodeset` and add one of the following options to this line, depending on your hardware:

Intel CPU:
`intel_iommu=on`

AMD CPU:
`amd_iommu=on`

Save the file and exit. Then type:
`sudo update-grub`

5. Now we can check that IOMMU is actually supported. **Reboot the PC**. Open a terminal window.On **AMD** machines use:
   `dmesg | grep AMD-Vi`The output should be similar to this:

   ```
   …
   AMD-Vi: Enabling IOMMU at 0000:00:00.2 cap 0x40
   AMD-Vi: Lazy IO/TLB flushing enabled
   AMD-Vi: Initialized for Passthrough Mode
   …
   ```

   Or use:
   `cat /proc/cpuinfo | grep svm`

On **Intel** machines use:
`dmesg | grep "Virtualization Technology for Directed I/O"`

The output should be this:

```
[ 0.902214] DMAR: Intel(R) Virtualization Technology for Directed I/O
```

If you do not get this output, then VT-d or AMD-V is not working – you need to fix that before you continue! Most likely it means that your hardware (CPU) doesn’t support IOMMU, in which case there is no point continuing this tutorial 😥 .

Two graphics processors

In addition to a CPU and motherboard that supports IOMMU, you need **two graphics processors (GPU)**:

1. One GPU for your Linux host (the OS you are currently running, I hope);

2. One GPU (graphics card) for your Windows guest.

We are building a system that runs two operating systems at the same time. Many resources like disk space, memory, etc. can be switched forth and back between the host and the guest, as needed. Unfortunately the GPU cannot be switched or shared between the two OS (work is being done to overcome this).

If, like me, you use Linux for the everyday stuff such as emails, web browsing, documents, etc., and Windows for gaming, photo or video editing, you’ll have to give Windows a more powerful GPU, while Linux will run happily with an inexpensive GPU, or the integrated graphics processor (IGP).

UEFI support in the GPU used with Windows

In this tutorial I use UEFI to boot the Windows VM. That means that **the graphics card you are going to use for the Windows guest must support UEFI – most newer cards do**. You can check here if your video card and BIOS support UEFI. (For more information, see here: http://vfio.blogspot.com/2014/08/does-my-graphics-card-rom-support-efi.html.)

Laptop users with **Nvidia Optimus technology**: Laptop users with Nvidia Optimus technology: Misairu_G (username) published an in-depth guide to VGA passthrough on laptops using Nvidia Optimus technology – see GUIDE to VGA passthrough on Nvidia Optimus laptops. (For reference, here some older posts on the subject: https://forums.linuxmint.com/viewtopic.php?f=231&t=212692&p=1300764#p1300634.)

Note: If your GPU does NOT support UEFI, there is still hope. You will have to look for a tutorial using the Seabios method. It's not much different from this here, but there are some things to consider. In short, if your GPU has an UEFI BIOS, use it!

Part 2 – Installing Qemu / KVM

The Qemu emulator shipped with Linux Mint 18 to 18.2 is version 2.5 and supports almost all of the latest and the greatest KVM features.

We install kvm, qemu, and some other stuff we need or want:

```
sudo apt-get update
sudo apt-get install qemu-kvm seabios qemu-utils cpu-checker hugepages
```

For **AMD Ryzen**, see here.

Part 3 – Determining the Devices to Pass Through to Windows

We need to find the PCI ID(s) of the graphics card and perhaps other devices we want to pass through to the Windows VM. Normally the IGP (the GPU inside the processor) will be used for Linux, and the discrete graphics card for the Windows guest. My CPU does not have an integrated GPU, so I use 2 graphics cards. Here my hardware setup:

- GPU for Linux: Nvidia Quadro 2000 residing in the first PCIe graphics card slot.
- GPU for Windows: Nvidia GTX 970 residing in the second PCIe graphics card slot.

To determine the PCI bus number and PCI IDs, enter:

```
lspci | grep VGA
```

Here is the output on my system:

```
01:00.0 VGA compatible controller: NVIDIA Corporation GF106GL [Quadro 2000] (rev a1)
02:00.0 VGA compatible controller: NVIDIA Corporation Device 13c2 (rev a1)
```

The first card under 01:00.0 is the Quadro 2000 I want to use for the Linux host. The other card under 02:00.0 I want to pass to Windows.

Modern graphics cards usually come with an on-board audio controller, which we need to pass through as well. To find its ID, enter:

```
lspci -nn | grep 02:00.
```

Substitute “`02:00.`” with the bus number of the graphics card you wish to pass to Windows, without the trailing “`0`“. Here is the output on my computer:

```
02:00.0 VGA compatible controller [0300]: NVIDIA Corporation Device [10de:13c2] (rev a1)
02:00.1 Audio device [0403]: NVIDIA Corporation Device [10de:0fbb] (rev a1)
```

Write down the **bus numbers** (02:00.0 and 02:00.1 above), as well as the **PCI IDs** (10de:13c2 and 10de:0fbb in the example above).

Now check to see that the graphics card resides within its own IOMMU group:

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

For a sorted list, use:

```
for a in /sys/kernel/iommu_groups/*; do find $a -type l; done
```

Look for the bus number of the graphics card you want to pass through. Here is the (shortened) output on my system:

```
…
/sys/kernel/iommu_groups/18/devices/0000:00:1f.3
/sys/kernel/iommu_groups/18/devices/0000:00:1f.2
/sys/kernel/iommu_groups/18/devices/0000:00:1f.0
/sys/kernel/iommu_groups/19/devices/0000:01:00.1
/sys/kernel/iommu_groups/19/devices/0000:01:00.0
/sys/kernel/iommu_groups/2/devices/0000:00:02.0
/sys/kernel/iommu_groups/20/devices/0000:02:00.1
```

```
/sys/kernel/iommu_groups/20/devices/0000:02:00.0
/sys/kernel/iommu_groups/21/devices/0000:05:00.0
/sys/kernel/iommu_groups/21/devices/0000:06:04.0
…
```

Make sure the GPU and perhaps other PCI devices you wish to pass through reside within their own IOMMU group. In my case the graphics card and its audio controller designated for passthrough both reside in IOMMU group 20. No other PCI devices reside in this group, so all is well.

If your VGA card shares an IOMMU group with other PCI devices, check that your motherboard BIOS is up-to-date (some vendors had bugs that have been fixed in a newer BIOS, for example the 3.30 to 4.40 BIOS update on Ryzen/AMD X390 boards). You may need to either upgrade the kernel to 4.8 or 4.15, or compile the kernel with the ACS override patch. See more below under part 12 – troubleshooting.

Next step is to find the mouse and keyboard (USB devices) that we want to assign to the Windows guest. Remember, we are going to run 2 independent operating systems side by side, and we control them via mouse and keyboard.

About keyboard and mouse

Depending whether and how much control you want to have over each system, there are different approaches:

1. Get a USB-**KVM (Keyboard/VGA/Mouse) switch**. This is a small hardware device with usually 2 USB ports for keyboard and mouse as well as a VGA or (the more expensive) DVI or HDMI graphics outputs. In addition the USB-KVM switch has two USB cables and 2 VGA/DVI/HDMI cables to connect to two different PCs. Since we run 2 virtual PCs on one single system, this is viable solution. See also my Virtualization Hardware Accessories post.
Advantages:
– Works without special software in the OS, just the usual mouse and keyboard drivers;

– Best in performance – no software overhead.
Disadvantages:
– Requires extra (though inexpensive) hardware;
– More cable clutter and another box with cables on your desk;
– Requires you to press a button to switch between host and guest and vice versa;
– Need to pass through a USB port or controller – see below on IOMMU groups.

2. Without spending a nickel you can simply **pass through your mouse and keyboard** when the VM starts.
Advantages:
– Easy to implement;
– No money to invest;
– Good solution for setting up Windows.
Disadvantages:
– Once the guest starts, your mouse and keyboard only control that guest, not the host. You will have to plug them into another USB port to gain access to the host.

3. **Synergy** (http://symless.com/synergy/) is a commercial software solution that, once installed and configured, allows you to interact with two PCs or virtual machines.
Advantages:
– Most versatile solution, especially with dual screens;
– Software only, easy to configure;
– No hardware purchase required.
Disadvantages:
– Requires the installation of software on both the host and the guest;
– Doesn't work during Windows installation (see option 2);
– Costs $10 for a Basic, lifetime license;
– May produce lag, although I doubt you'll notice unless there is something wrong with the bridge configuration.

4. "**Multi-device**" bluetooth keyboard and mouse that can connect to two different devices and switch between them at the press of a button (see for example here):
Advantages:
– Most convenient solution;
– Same performance as option 1.

Disadvantages:
– Price.
– Make sure the device supports Linux, or that you can return it if it doesn't!

I first went with option 1 for robustness and universality, but have replaced it with option 4. I'm now using a Logitech MX master BT mouse and a Logitech K780 BT keyboard. See here for how to pair these devices to the USB dongles.

Both option 1 and 4 usually require to pass through a USB PCI device to the Windows guest. I had a need for both USB2 and USB3 ports in my Windows VM and I was able to pass through two USB controllers to my Windows guest, using PCI passthrough.

For the VM installation we choose option 2 (see above), that is we pass our keyboard and mouse through to the Windows VM. To do so, we need to identify their USB ID:

```
lsusb
```

Here my system output (truncated):

```
…
Bus 010 Device 006: ID 045e:076c Microsoft Corp. Comfort Mouse 4500
Bus 010 Device 005: ID 045e:0750 Microsoft Corp. Wired Keyboard 600
…
```

Note down the IDs: 045e:076c and 045e:0750 in my case.

Part 4 – Prepare for Passthrough

We are assigning a dummy driver vfio-pci to the graphics card we want to use under Windows. To do that, we first have to prevent the default driver to bind to the graphics card.

Once more edit the /etc/default/grub file:

```
gksudo xed /etc/default/grub
```

The entry we are looking for is:

```
GRUB_CMDLINE_LINUX_DEFAULT=”quiet intel_iommu=on”
```

We need to add one of the following options to this line, depending on your hardware:

Nvidia graphics card for Windows VM:
`modprobe.blacklist=nouveau`

Note: This blacklists the Nouveau driver and prevents it from loading. If you run two Nvidia cards and use the open Nouveau driver for your Linux host, DON'T blacklist the driver!!! Chances are the tutorial will work since the vfio-pci driver should grab the graphics card before nouveau takes control of it. The same goes for AMD cards (see below).

AMD graphics card for Windows VM:
`modprobe.blacklist=radeon`
or
`modprobe.blacklist=amdgpu`
depending on which driver loads when you boot.

After editing, my /etc/default/grub file now looks like this:

```
GRUB_DEFAULT=0
#GRUB_HIDDEN_TIMEOUT=10
#GRUB_HIDDEN_TIMEOUT_QUIET=true
GRUB_TIMEOUT_STYLE=countdown
GRUB_TIMEOUT=0
GRUB_DISTRIBUTOR=`lsb_release -i -s 2> /dev/null || echo Debian`
GRUB_CMDLINE_LINUX_DEFAULT=”modprobe.blacklist=nouveau quiet intel_iommu=on”
GRUB_CMDLINE_LINUX=””
```

Finally run update-grub:
`sudo update-grub`

(There are more ways to blacklist driver modules, for example using Kernel Mode Settings. For more on Kernel Mode Setting, see https://wiki.archlinux.org/index.php/kernel_mode_setting.)

In order to make the graphics card available to the Windows VM, we will assign a "dummy" driver as a place holder: vfio-pci.

Note: If you have two identical graphics cards for both the host and the VM, the method below won't work. In that case see the following posts: https://forums.linuxmint.com/viewtopic.php?f=231&t=212692&start=40#p1174032 as well as https://forums.linuxmint.com/viewtopic.php?f=231&t=212692&start=40#p1173262.

Open or create /etc/modprobe.d/local.conf:

```
gksudo xed /etc/modprobe.d/local.conf
```

and insert the following:

```
options vfio-pci ids=10de:13c2,10de:0fbb
```

where 10de:13c2 and 10de:0fbb are the PCI IDs for your graphics card' VGA and audio part.
Save the file and exit the editor.

Some applications like Passmark require the following option:

```
echo "options kvm ignore_msrs=1" >> /etc/modprobe.d/kvm.conf
```

To load vfio and other required modules at boot, edit the /etc/initramfs-tools/modules file:

```
gksudo xed /etc/initramfs-tools/modules
```

At the end of the file add:

```
vfio
vfio_iommu_type1
vfio_pci
vfio_virqfd
vhost-net
```

Save and close the file.

We need to update the initramfs. Enter at the command line:

```
update-initramfs -u
```

Part 5 – Network Settings

For performance reasons it is best to create a virtual network bridge that connects the VM with the host. In a separate post I have written a detailed tutorial on how to set up a bridge using Network Manager.

Note: Bridging only works for wired networks. If your PC is connected to a router via a wireless link (Wifi), you won't be able to use a bridge. There are workarounds such as routing or ebtables (see https://wiki.debian.org/BridgeNetworkConnections#Bridging_with_a_wireless_NIC).

Once you've setup the network, reboot the computer and test your network configuration – open your browser and see if you have Internet access.

Part 6 – Configure hugepages

This step is not required to run the Windows VM, but helps improve performance. First we need to decide how much memory we want to give to Windows. Here my suggestions:

1. No less than 4GB.
2. If you got 16GB total and aren't running multiple VMs, give Windows 8GB-12GB, depending on what you plan to do with Windows.

For this tutorial I use 8GB. Let's see what we got:

```
hugeadm --explain
```

If you get

```
hugeadm:ERROR: No hugetlbfs mount points found
```

then you need to enable hugepages. To do so, edit /etc/default/qemu-kvm as root:

```
gksudo xed /etc/default/qemu-kvm
```

and add or uncomment

```
KVM_HUGEPAGES=1
```

Reboot!

After the reboot, in a terminal window, enter again:

```
hugeadm --explain
```

```
Total System Memory: 32180 MB

Mount Point Options
/run/hugepages/kvm rw,relatime,mode=775,gid=126

Huge page pools:
Size Minimum Current Maximum Default
2097152 0 0 0 *
…
```

As you can see, hugepages are now mounted to `/run/hugepages/kvm`, and the hugepage size is 2097152 Bytes/(1024*1024)=2MB.

Another way to determine the hugepage size is:

```
grep "Hugepagesize:" /proc/meminfo
```

```
Hugepagesize: 2048 kB
```

Here some math:
We want to reserve 8GB for Windows:
8GB = 8x1024MB = 8192MB

Our hugepage size is 2MB, so we need to reserve:
8192MB/2MB = 4096 hugepages

We need to add some % extra space for overhead (some say 2%, some say 10%), to be on the safe side I use 4500 (if memory is scarce, you should be OK with 4300, perhaps less – see comments here).

To configure the hugepage pool, open `/etc/sysctl.conf` as root:

```
gksudo xed /etc/sysctl.conf
```

Then add the following lines into the file:

```
# Set hugetables / hugepages for KVM single guest using 8GB RAM
vm.nr_hugepages = 4500
```

Save and close.

Now **reboot** for our hugepages configuration to take effect.

After the reboot, run in a terminal:

```
hugeadm --explain
```

```
Total System Memory: 32180 MB

Mount Point Options
/run/hugepages/kvm rw,relatime,mode=775,gid=126

Huge page pools:
Size Minimum Current Maximum Default
2097152 4500 4500 4500 *

Huge page sizes with configured pools:
2097152

A /proc/sys/kernel/shmmax value of 9223372036854775807 bytes may be sub-optimal.
To maximise shared memory usage, this should be set to the size of the largest
shared memory segment size you want to be able to use. Alternatively, set it to
a size matching the maximum possible allocation size of all huge pages. This can
be done automatically, using the –set-recommended-shmmax option.
```

```
The recommended shmmax for your currently allocated huge pages is 9437184000
bytes.
To make shmmax settings persistent, add the following line to /etc/sysctl.conf:
kernel.shmmax = 9437184000
```

Note the sub-optimal shmmax value. We fix it temporarily with:

```
sudo hugeadm --set-recommended-shmmax
```

And permanently by editing `/etc/sysctl.conf`:

```
gksudo xed /etc/sysctl.conf
```

and adding the following line:

```
kernel.shmmax = 9437184000
```

Note: Use the kernel.shmmax recommended by hugeadm –explain!!!

Part 7 – Install OVMF BIOS and VFIO drivers

In this tutorial I use UEFI to boot the Windows VM. There are some advantages to it, namely it starts faster and overcomes some issues associated with legacy boot (Seabios) – see note below.

Note: If you plan to use the Intel IGD (integrated graphics device) for your Linux host, UEFI boot is the way to go. UEFI overcomes the VGA arbitration problem associated with the IGD and the use of the legacy Seabios. If, for some reason, you cannot boot the VM using UEFI, you need to compile the i915 VGA arbiter patch into the kernel. For more on VGA arbitration, see here. For the i915 VGA arbiter patch, look here or under Part 15 – References.

First install OVMF:

```
sudo apt-get install ovmf
```

This will download the current Ubuntu version of OVMF and create the necessary links and directories.

Download the **VFIO driver ISO** to be used with the Windows installation from https://fedoraproject.org/wiki/Windows_Virtio_Drivers. Below are the direct links to the ISO images:

Latest VIRTIO drivers: https://fedorapeople.org/groups/virt/virtio-win/direct-downloads/latest-virtio/virtio-win.iso

Stable VIRTIO drivers: https://fedorapeople.org/groups/virt/virtio-win/direct-downloads/stable-virtio/virtio-win.iso

I chose the **latest** driver ISO.

Part 8 – Prepare Windows VM Storage Space

We need some storage space on which to install the Windows VM. There are several choices:

1. Create a **raw image file**.
   Advantages:
   – Easy to implement;
   – Flexible – the file can grow with your requirements;
   – Snapshots;
   – Easy migration;
   – Good performance.
   Disadvantages:
   – Takes up the entire space you specify.
2. Create a dedicated **LVM volume**.
   Advantages:
   – Familiar technology (at least to me);
   – Excellent performance, like bare-metal;
   – Flexible – you can add physical drives to increase the volume size;
   – Snapshots;
   – Mountable within Linux host using kpartx.
   Disadvantages:
   – Takes up the entire space specified;
   – Migration isn’t that easy.

3. Pass through a **PCI SATA controller** / disk.
   Advantages:
   – Excellent performance, using original Windows disk drivers;
   – Allows the use of Windows virtual drive features;
   – Can use an existing bare-metal installation of Windows in a VM;
   – Possibility to boot Windows directly, i.e. not as VM;
   – Possible to add more drives.
   Disadvantages:
   – The PC needs at least two discrete SATA controllers;
   – Host has no access to disk while VM is running;
   – Requires a dedicated SATA controller and drive(s);
   – SATA controller must have its own IOMMU group;
   – Possible conflicts in Windows between bare-metal and VM operation.

For further information on these and other image options, see here: https://en.wikibooks.org/wiki/QEMU/Images

Although I'm using an LVM volume, I suggest you start with the raw image. Let's create a raw disk image:

```
fallocate -l 50G /media/user/win.img
```

Note: Adjust size and path to match your needs and actual resources.

Part 9 – Check Configuration

It's best to check that we got everything:

KVM: `kvm-ok`

```
INFO: /dev/kvm exists
KVM acceleration can be used
```

KVM module: `lsmod | grep kvm`

```
kvm_intel 200704 0
kvm 593920 1 kvm_intel
irqbypass 16384 2 kvm,vfio_pci
```

Above is the output for the Intel module.

VFIO: `lsmod | grep vfio`

```
vfio_pci 45056 0
vfio_virqfd 16384 1 vfio_pci
irqbypass 16384 2 kvm,vfio_pci
vfio_iommu_type1 24576 0
vfio 32768 2 vfio_iommu_type1,vfio_pci
```

QEMU: `qemu-system-x86_64 --version`
You need QEMU emulator version 2.5.0 or newer.

Did vfio load and bind the graphics card?
`dmesg | grep vfio`

```
[ 3.294520] vfio-pci 0000:02:00.0: vgaarb: changed VGA decodes:
olddecodes=io+mem,decodes=io+mem:owns=none
[ 3.311690] vfio_pci: add [10de:13c2[ffff:ffff]] class 0x000000/00000000
[ 3.331711] vfio_pci: add [10de:0fbb[ffff:ffff]] class 0x000000/00000000
[ 9.206363] vfio-pci 0000:02:00.0: vgaarb: changed VGA decodes:
olddecodes=io+mem,decodes=io+mem:owns=none
```

It worked!

Interrupt remapping: `dmesg | grep VFIO`

```
[ 3.288843] VFIO – User Level meta-driver version: 0.3
```

All good!

If you get this message:

```
vfio_iommu_type1_attach_group: No interrupt remapping support. Use the module
param “allow_unsafe_interrupts” to enable VFIO IOMMU support on this platform
```

enter the following command in a **root terminal** (or use `sudo -i`):

```
echo "options vfio_iommu_type1 allow_unsafe_interrupts=1" >
/etc/modprobe.d/vfio_iommu_type1.conf
```

In this case you need to reboot once more.

Part 10 – Create Script to Start Windows

I’ve modified/created a script that will start the Windows VM. Copy the script below and safe it as windows10vm.sh (or whatever name you like, just keep the

.sh extension):

```
#!/bin/bash

vmname="windows10vm"

if ps -A | grep -q $vmname; then
echo "$vmname is already running." &
exit 1

else

# use pulseaudio
export QEMU_AUDIO_DRV=pa
export QEMU_PA_SAMPLES=8192
export QEMU_AUDIO_TIMER_PERIOD=99
export QEMU_PA_SERVER=/run/user/1000/pulse/native

cp /usr/share/OVMF/OVMF_VARS.fd /tmp/my_vars.fd

qemu-system-x86_64 \
-name $vmname,process=$vmname \
-machine type=q35,accel=kvm \
-cpu host,kvm=off \
-smp 4,sockets=1,cores=2,threads=2 \
-m 8G \
-mem-path /run/hugepages/kvm \
-mem-prealloc \
-balloon none \
-rtc clock=host,base=localtime \
-vga none \
-nographic \
-serial none \
-parallel none \
-soundhw hda \
-usb -usbdevice host:045e:076c -usbdevice host:045e:0750 \
-device vfio-pci,host=02:00.0,multifunction=on \
-device vfio-pci,host=02:00.1 \
-drive if=pflash,format=raw,readonly,file=/usr/share/OVMF/OVMF_CODE.fd \
-drive if=pflash,format=raw,file=/tmp/my_vars.fd \
-boot order=dc \
-drive id=disk0,if=virtio,cache=none,format=raw,file=/media/user/win.img \
-drive file=/home/user/ISOs/win10.iso,index=1,media=cdrom \
-drive file=/home/user/Downloads/virtio-win-0.1.140.iso,index=2,media=cdrom \
-netdev type=tap,id=net0,ifname=vmtap0,vhost=on \
-device virtio-net-pci,netdev=net0,mac=00:16:3e:00:01:01

exit 0
fi
```

Make the file executable:

```
sudo chmod +x windows10vm.sh
```

You need to edit the file and change the settings and paths to match your CPU and configuration. See below for explanations on the qemu-system-x86 options:

-name $vmname,process=$vmname
Name and process name of the VM. The process name is displayed when using `ps -A` to show all processes, and used in the script to determine if the VM is already running. Don't use `win10` as process name, for some inexplicable reason it doesn't work!

-machine type=q35,accel=kvm
This specifies a machine to emulate. The `accel=kvm` option tells qemu to use the KVM acceleration – without it the Windows guest will run in qemu emulation mode, that is it'll run real slow.
I have chosen the `type=q35` option, as it improved my SSD read and write speeds. See https://wiki.archlinux.org/index.php/QEMU#Virtual_machine_runs_too_slowly. In some cases `type=q35` will prevent you from installing Windows, instead you

may need to use `type=pc,accel=kvm`. See the post here. To see all options for `type=…`, enter the following command:

```
qemu-system-x86_64 -machine help
```

Important: Several users passing through **Radeon RX 480** and **Radeon RX 470** cards have reported reboot loops after updating and installing the Radeon drivers. If you pass through a Radeon graphics card, it is better to replace the -machine line in the startup script with the following line:

```
-machine type=pc,accel=kvm
```

to use the default i440fx emulation.

-cpu host,kvm=off

This tells qemu to emulate the host's exact CPU. There are more options, but it's best to stay with `host`.

The `kvm=off` option is only needed for **Nvidia graphics cards** – if you have an AMD/Radeon card for your Windows guest, you can remove that option and specify `-cpu host`.

-smp 4,sockets=1,cores=2,threads=2

This specifies multiprocessing. `-smp 4` tells the system to use 4 (virtual) processors. My CPU has 6 cores, each supporting 2 threads, which makes a total of 12 threads. It's probably best not to assign all CPU resources to the Windows VM – the host also needs some resources (remember that some of the processing and I/O coming from the guest takes up CPU resources in the host). In the above example I gave Windows 4 virtual processors. `sockets=1` specifies the number of actual CPU sockets qemu should assign, `cores=2` tells qemu to assign 2 processor cores to the VM, and `threads=2` specifies 2 threads per core. It may be enough to simply specify `-smp 4`, but I'm not sure about the performance consequences (if any).

If you have a 4-core Intel CPU with hyperthreading, you can specify `-smp 6,sockets=1,cores=3,threads=2` to assign 75% of your CPU resources to the Windows VM. This should usually be enough even for demanding games and applications.

-m 8G

The `-m` option assigns memory (RAM) to the VM, in this case 8 GByte. Same as `-m 8192`. You can increase or decrease it, depending on your resources and needs.

With modern Windows releases it doesn't make sense to give it less than 4G, unless you are really stretched with RAM. **Make sure your hugepage size matches this!**

-mem-path /run/hugepages/kvm
This tells qemu where to find the hugepages we reserved. If you haven't configured hugepages, you need to remove this option.

-mem-prealloc
Preallocates the memory we assigned to the VM.

-balloon none
We don't want memory ballooning (as far as I know Windows won't support it anyway).

-rtc clock=host,base=localtime
`-rtc clock=host` tells qemu to use the host clock for synchronization. `base=localtime` allows the Windows guest to use the local time from the host system. Another option is `utc`.

-vga none
Disables the built in graphics card emulation. You can remove this option for debugging.

-nographic
Totally disables SDL graphical output. For debugging purposes, remove this option if you don't get to the Tiano Core screen.

-serial none
-parallel none
Disable serial and parallel interfaces. Who needs them anyway?

-soundhw hda
Together with the `export QEMU_AUDIO_DRV=pa` shell command, this option enables sound through PulseAudio.
If you want to pass through a physical audio card or audio device and stream

audio from your Linux host to your Windows guest, see here: Streaming Audio from Linux to Windows.

-usb -usbdevice host:045e:076c -usbdevice host:045e:0750
`-usb` enables USB support and `-usbdevice host:`… assigns the USB host devices mouse (`045e:076c`) and keyboard (`045e:0750`) to the guest. Replace the device IDs with the ones you found using the `lsusb` command in Part 3 above!

-device vfio-pci,host=02:00.0,multifunction=on
-device vfio-pci,host=02:00.1
Here we specify the graphics card to pass through to the guest, using vfio-pci. Fill in the PCI IDs you found under Part 3 above. It is a `multifunction` device (graphics and sound). Make sure to pass through both the video and the sound part (`02:00.0` and `02:00.1` in my case).

-drive if=pflash,format=raw,readonly,file=/usr/share/OVMF/OVMF_CODE.fd
Specifies the location and format of the bootable OVMF UEFI file. This file doesn't contain the variables, which are loaded separately (see right below).

-drive if=pflash,format=raw,file=/tmp/my_vars.fd
These are the variables for the UEFI boot file, which were copied by the script to `/tmp/my_vars.fd`.

-boot order=dc
Start boot from CD (`d`), then first hard disk (`c`). After installation of Windows you can remove the "`d`" to boot straight from disk.

-drive id=disk0,if=virtio,cache=none,format=raw,file=/media/user/win.img
Defines the first hard disk. With the options above it will be accessed as a paravirtualized (`if=virtio`) drive in raw format (`format=raw`).
Important: `file=/`… enter the path to your previously created win.img file.
Other possible drive options are `file=/dev/mapper/group-vol` for LVM volumes, or `file=/dev/sdx1` for entire disks or partitions.
For a list of useful `-drive` options, see my post here.

**-drive file=/home/user/ISOs/win10.iso,index=1,media=cdrom **
This attaches the Windows win10.iso as CD or DVD. The driver used is the ide-cd driver.
Important: `file=/`… enter the path to your Windows ISO image.

Note: This option is only needed during installation. Afterwards, copy the line to the end of the file and comment it out with #.

**-drive file=/home/user/Downloads/virtio-win-0.1.140.iso,index=2,media=cdrom **
This attaches the virtio ISO image as CD. Note the different index.
Important: `file=/`… enter the path to your virtio ISO image. If you downloaded it to the default location, it should be in your Downloads directory.
Note 1: There are many ways to attach ISO images or drives and invoke drivers. My system didn't want to take a second scsi-cd device, so this option did the job. Unless this doesn't work for you, don't change.
Note 2: This option is only needed during installation. Afterwards, copy the line to the end of the file and comment it out with #.

**-netdev type=tap,id=net0,ifname=vmtap0,vhost=on **
-device virtio-net-pci,netdev=net0,mac=00:16:3e:00:01:01
Defines the network interface and network driver. It's best to define a MAC address, here `00:16:3e:00:01:01`. The MAC is specified in Hex and you can change the last `:01:01` to your liking. Make sure no two MAC addresses are the same!
`vhost=on` is optional – some people reported problems with this option. It is for network performance improvement.
For more information: https://wiki.archlinux.org/index.php/QEMU#Network and https://wiki.archlinux.org/index.php/QEMU#Networking.

Important: QEMU is evolving rapidly. Above syntax is correct for QEMU 2.5. The latest stable version is 2.11. For a more complete and perhaps updated documentation on QEMU, see https://www.qemu.org/documentation/. The QEMU user manual can be found at https://qemu.weilnetz.de/doc/qemu-doc.html.

Part 11 – Install Windows

Start the VM by running the script as root:

```
sudo ./windows10vm.sh
```

(Make sure you specify the correct path.)

You should get a Tiano Core splash screen with the memory test result.

You might land in an EFI shell. Type `exit` and you should be getting a menu. Select the boot disk and hit Enter.

Then the Windows ISO boots and asks you to:

```
Press any key to start the CD / DVD…
```

Press a key!

Windows will then ask you to:

```
Select the driver to install
```

Click “Browse”, then select your VFIO ISO image and go to “**viostor**“, open and select your Windows version, then select the “AMD64” version for 64 bit systems, click OK.

Windows will ask for the license key, and you need to specify how to install – choose “Custom”. Then select your drive (there should be only disk0) and install.

Windows may reboot several times. When done rebooting, open Device Manager and select the Network interface. Right-click and select update. Then browse to the VFIO disk and install **NetKVM**.

Windows should be looking for a display driver by itself. If not, install it manually.

Note: In my case, Windows did not correctly detect my drives being SSD drives. Not only will Windows 10 perform unnecessary disk optimization tasks, but these “optimizations” can actually lead to reduced SSD life and performance issues. To make Windows 10 determine the correct disk drive type, do the following:

1. Inside Windows 10, right-click the Start menu.
2. Select “Command prompt (admin)”.

3. At the command prompt, run:

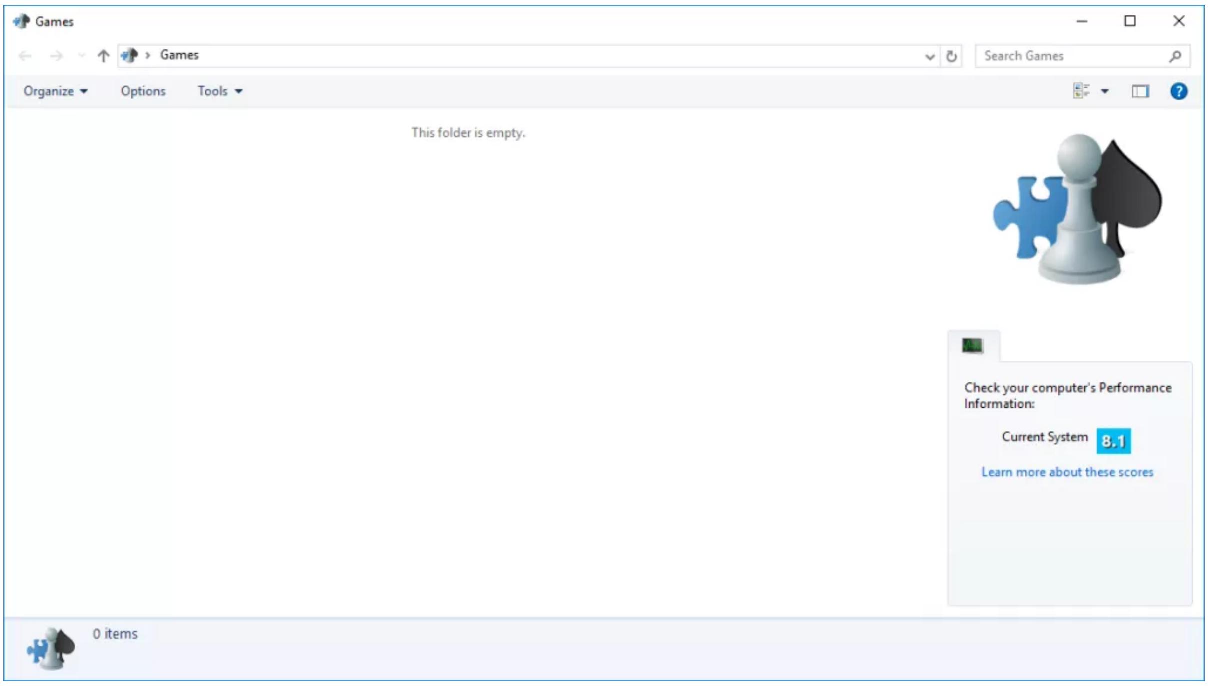
```
winsat formal
```

4. It will run a while and then print the Windows Experience Index (WEI).

5. To display the WEI, press WIN+R and enter:

```
shell:Games
```

You get the following screen:



This folder is empty.

Check your computer's Performance Information:

Current System 8.1

Learn more about these scores

WEI Windows Experience Index in Windows 10

Please share your WEI in a comment below!

To check that Windows correctly identified your SSD:

1. Open Explorer

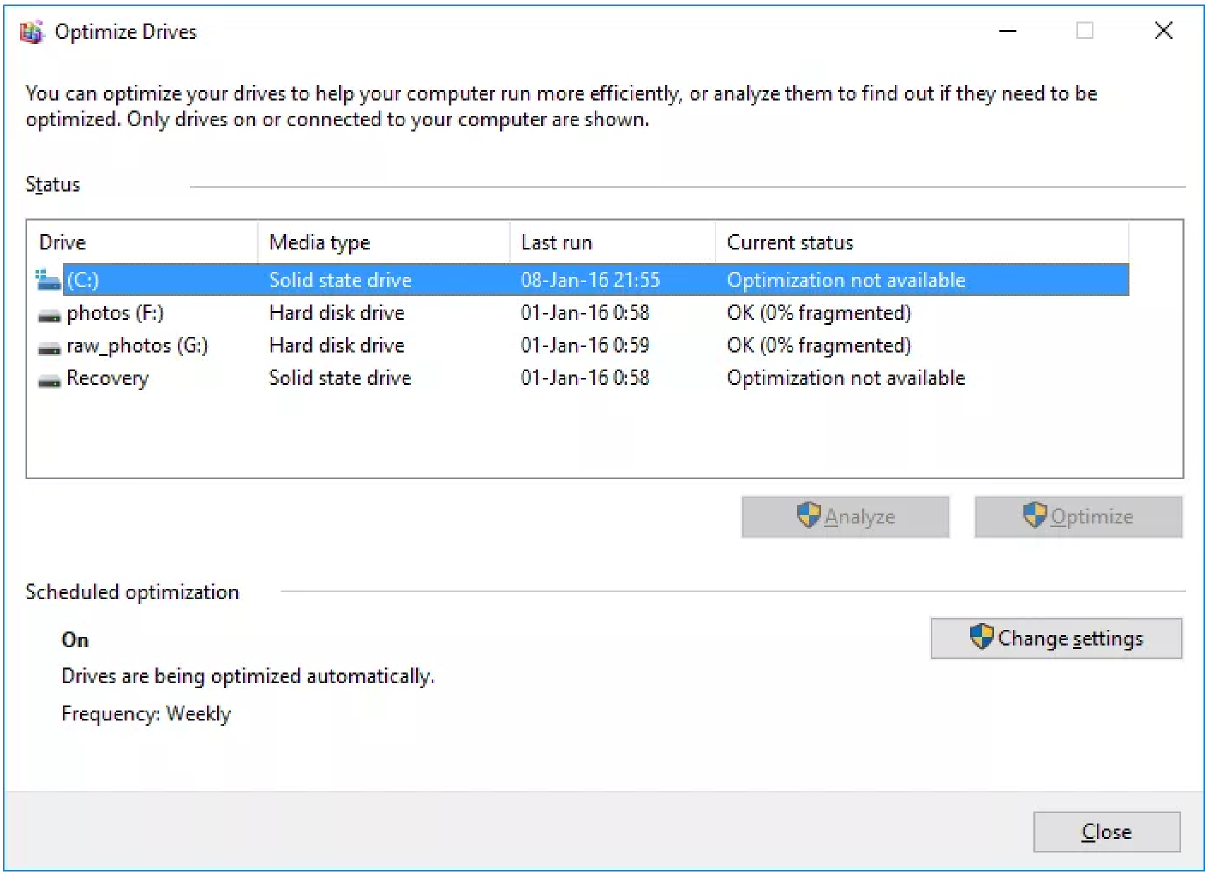
2. Click “This PC” in the left tab.

3. Right-click your drive (e.g. C:) and select “Properties”.

4. Select the “Tools” tab.

5. Click “Optimize”

You should see something similar to this:



You can optimize your drives to help your computer run more efficiently, or analyze them to find out if they need to be optimized. Only drives on or connected to your computer are shown.

Status

Drive	Media type	Last run	Current status
(C:)	Solid state drive	08-Jan-16 21:55	Optimization not available
photos (F:)	Hard disk drive	01-Jan-16 0:58	OK (0% fragmented)
raw_photos (G:)	Hard disk drive	01-Jan-16 0:59	OK (0% fragmented)
Recovery	Solid state drive	01-Jan-16 0:58	Optimization not available

Scheduled optimization

On

Drives are being optimized automatically.

Frequency: Weekly

Use Optimize Drives to optimize for SSD

In my case, I have drive C: (my Windows 10 system partition) and a “Recovery” partition located on an SSD, the other two partitions (“photos” and “raw_photos”) are using regular hard drives (HDD). Notice the “Optimization not available” 😀 .

Turn off hibernation and suspend ! Having either of them enabled can cause your Windows VM to hang, or may even affect the host. To turn off hibernation and suspend, follow the instructions for hibernation and suspend.

Turn off fast startup ! When you shut down the Windows VM, fast startup leaves the file system in a state that is unmountable by Linux. If something goes wrong, you’re screwed. NEVER EVER let proprietary technology have control over your data. Follow these instructions to turn off fast startup.

By now you should have a working Windows VM with VGA passthrough.

Part 12 – Troubleshooting

Below are a number of common issues when trying to install/run Windows in a VGA passthrough environment.

VM not starting – graphics driver

A common issue is the binding of a driver to the graphics card we want to pass through. As I was writing this how-to and made changes to my (previously working) system, I suddenly couldn't start the VM anymore. The first thing to check if you don't get a black Tianocore screen is whether or not the graphics card you try to pass through is bound to the vfio-pci driver:

```
dmesg | grep -i vfio
```

The output should be similar to this:

```
[ 2.735931] VFIO – User Level meta-driver version: 0.3

[ 2.757208] vfio_pci: add [10de:13c2[ffff:ffff]] class 0x000000/00000000

[ 2.773223] vfio_pci: add [10de:0fbb[ffff:ffff]] class 0x000000/00000000

[ 8.437128] vfio-pci 0000:02:00.0: enabling device (0000 -> 0003)
```

The above example shows that the graphics card is bound to the vfio-pci driver (see last line), which is what we want. If the command doesn't produce any output, or a very different one from above, something is wrong. To check further, enter:

```
lspci -k | grep -i -A 3 vga
```

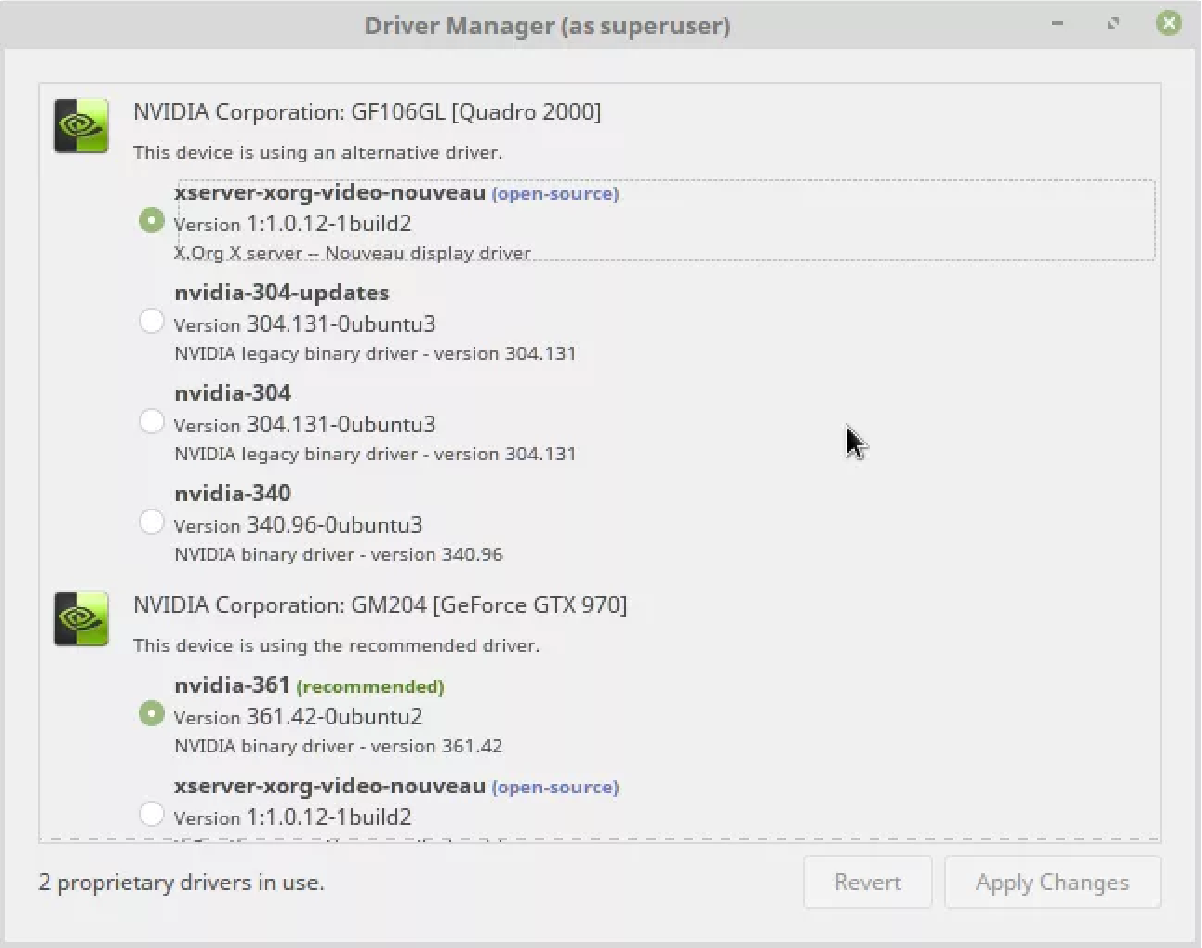
Here is what I got when my VM wouldn't want to start anymore:

```
01:00.0 VGA compatible controller: NVIDIA Corporation GF106GL [Quadro 2000] (rev a1)

Subsystem: NVIDIA Corporation GF106GL [Quadro 2000]

Kernel driver in use: nvidia

Kernel modules: nvidiafb, nouveau, nvidia_361

–

02:00.0 VGA compatible controller: NVIDIA Corporation GM204 [GeForce GTX 970] (rev a1)

Subsystem: Gigabyte Technology Co., Ltd GM204 [GeForce GTX 970]

Kernel driver in use: nvidia

Kernel modules: nvidiafb, nouveau, nvidia_361
```

Graphics card 01:00.0 (Quadro 2000) uses the Nvidia driver – just what I want.

Graphics card 02:00.0 (GTX 970) also uses the Nvidia driver – that is NOT what I was hoping for. This card should be bound to the vfio-pci driver. So what do we do?

In Linux Mint, click the menu button, click “Control Center”, then click “Driver Manager” in the Administration section. Enter your password. You will then see the drivers associated with the graphics cards. Change the driver of the graphics card so it will use the opensource driver (in this example “Nouveau”) and press “Apply Changes”. After the change, it should look similar to the photo below:



NVIDIA Corporation: GF106GL [Quadro 2000]
This device is using an alternative driver.

● xserver-xorg-video-nouveau (open-source)
Version 1:1.0.12-1build2
X.Org X server – Nouveau display driver

○ nvidia-304-updates
Version 304.131-0ubuntu3
NVIDIA legacy binary driver - version 304.131

○ nvidia-304
Version 304.131-0ubuntu3
NVIDIA legacy binary driver - version 304.131

○ nvidia-340
Version 340.96-0ubuntu3
NVIDIA binary driver - version 340.96

NVIDIA Corporation: GM204 [GeForce GTX 970]
This device is using the recommended driver.

● nvidia-361 (recommended)
Version 361.42-0ubuntu2
NVIDIA binary driver - version 361.42

○ xserver-xorg-video-nouveau (open-source)
Version 1:1.0.12-1build2

2 proprietary drivers in use.

Driver Manager in Linux Mint

BSOD when installing AMD Crimson drivers under Windows

Several users on the Redhat VFIO mailing list have reported problems with the installation of AMD Crimson drivers under Windows. This seems to affect a number of AMD graphics cards, as well as a number of different AMD Crimson driver releases. A workaround is described here: https://www.redhat.com/archives/vfio-users/2016-April/msg00153.html

In this workaround the following line is added to the startup script, right above the definition of the graphics device:

```
-device ioh3420,bus=pci,addr=1c.0,multifunction=on,port=1,chassis=1,id=root.1
```

Should the above configuration give you a “Bus ‘pci’ not found” error, change the line as follows:

```
-device ioh3420,bus=pci.0,addr=1c.0,multifunction=on,port=1,chassis=1,id=root.1
```

Then you change the graphics card passthrough options as follows:

```
-device vfio-pci,host=02:00.0,bus=root.1,addr=00.0,multifunction=on \
-device vfio-pci,host=02:00.1,bus=root.1,addr=00.1 \
```

Identical graphics cards for host and guest

If you use two identical graphics cards for both the Linux host and the Windows guest, follow these instructions:

Modify the /etc/modprobe.d/local.conf file as follows:

```
install vfio-pci /sbin/vfio-pci-override-vga.sh
```

Create a /sbin/vfio-pci-override-vga.sh file with the following content:

```
#!/bin/sh

DEVS="0000:02:00.0 0000:02:00.1"

for DEV in $DEVS; do
echo "vfio-pci" > /sys/bus/pci/devices/$DEV/driver_override
done

modprobe -i vfio-pci
```

Make the vfio-pci-override-vga.sh file executable:

```
chmod u+x /sbin/vfio-pci-override-vga.sh
```

Windows ISO won't boot – 1

If you can't start the Windows ISO, it may be necessary to run a more recent version of Qemu to get features or work-arounds that solve problems. If you require a more updated version of Qemu (version 2.6.1 as of this writing), add the following PPA (warning: this is not an official repository – use at your own risk). At the terminal prompt, enter:

```
sudo add-apt-repository ppa:jacob/virtualisation
```

Windows ISO won't boot – 2

Sometimes the OVMF BIOS files from the official Ubuntu repository don't work with your hardware and the VM won't boot. In that case you can download alternative OVMF files from here: http://www.ubuntuupdates.org/package/core/wily/multiverse/base/ovmf, or get the most updated version from here:
https://www.kraxel.org/repos/jenkins/edk2/
Download the latest **edk2.git-ovmf-x64** file, in my case it was "edk2.git-ovmf-x64-0-20160914.b2137.gf8db652.noarch.rpm" for a 64bit installation. Open the downloaded .rpm file with root privileges and **unpack to /**.
Copy the following files:

```
sudo cp /usr/share/edk2.git/ovmf-x64/OVMF-pure-efi.fd /usr/share/ovmf/OVMF.fd
```

```
sudo cp /usr/share/edk2.git/ovmf-x64/OVMF_CODE-pure-efi.fd
/usr/share/OVMF/OVMF_CODE.fd
```

sudo cp /usr/share/edk2.git/ovmf-x64/OVMF_VARS-pure-efi.fd /usr/share/OVMF/OVMF_VARS.fd

Check to see if the /usr/share/ovmf/OVMF.fd link exists, if not, create it:

```
sudo ln -s '/usr/share/ovmf/OVMF.fd' '/usr/share/qemu/OVMF.fd'
```

Windows ISO won't boot – 3

Sometimes the Windows ISO image is corrupted or simply an old version that doesn't work with passthrough. Go to https://www.microsoft.com/en-us/software-download/windows10ISO and download the ISO you need (see your software license). Then try again.

Motherboard BIOS bugs

Some motherboard BIOSes have bugs and prevent passthrough. Use "dmesg" and look for entries like these:

```
[ 0.297481] [Firmware Bug]: AMD-Vi: IOAPIC[7] not in IVRS table
[ 0.297485] [Firmware Bug]: AMD-Vi: IOAPIC[8] not in IVRS table
[ 0.297487] [Firmware Bug]: AMD-Vi: No southbridge IOAPIC found in IVRS table
[ 0.297490] AMD-Vi: Disabling interrupt remapping due to BIOS Bug(s)
```

If you find entries that point to a faulty BIOS or problems with interrupt remapping, go to Easy solution to get IOMMU working on mobos with broken BIOSes. (All credits go to leonmaxx on the Ubuntu forum!)

Intel IGD and arbitration bug

For users of Intel CPUs with IGD (Intel graphics device): The Intel i915 driver has a bug, which has necessitated a kernel patch named i915 vga arbiter patch. According to developer Alex Williamson, this patch is needed any time you have host Intel graphics and make use of the x-vga=on option. This tutorial, however, does NOT use the x-vga option; the tutorial is based on UEFI boot and doesn't use VGA. That means you do NOT need the i915 vga arbiter patch! See http://vfio.blogspot.com/2014/08/primary-graphics-assignment-without-vga.html.

IOMMU group contains additional devices

When checking the IOMMU groups, your graphics card' video and audio part should be the only 2 entries under the respective IOMMU group. The same goes for any other PCI device you want to pass through, as you must pass through all devices within an IOMMU group, or nothing. If there are other devices within the same IOMMU group that you do not wish to pass through, or devices that are used by the host, you have the following options:

a) If the only device beside the PCI device you intend to pass through is a host controller, you may be able to disregard this device. Do NOT try to pass through a host controller! Continue with the instructions in the tutorial and cross your fingers – it might work.

b) Update your motherboard BIOS. Some motherboard manufacturers shipped faulty BIOS releases. With a little luck upgrading to the latest BIOS release can solve your problem.

c) If two graphics cards or add-on PCI cards appear in the same IOMMU, try to move one of your graphics card/PCI card to another slot (which may be bound to a different IOMMU group). If that is not possible or doesn't produce the expected results, see the following options.

d) Upgrade your kernel to kernel 4.15 or later. First, backup your entire Linux installation! Open Update Manager, then open View –> Linux kernels. Read the warning and choose Continue. Select a 4.15 or later kernel. Reboot after installation and check the IOMMU groups again.

e) If none of the above steps work, you will need to compile the kernel using the **ACS override patch**. Linux Mint forum member odtech has posted detailed instructions on how to compile and use the ACS override patch here. Make sure you are running the correct kernel, the one specified with the ACS patch! For further information, see http://vfio.blogspot.com/2014/08/vfiovga-faq.html and http://vfio.blogspot.com/2014/08/iommu-groups-inside-and-out.html.

Dual-graphics laptops (e.g. Optimus technology)

A quote from developer Alex Williamson:

> *OVMF is the way to go if you want to avoid patching your kernel, … if your GPU and guest OS support UEFI.*

> *Dual-graphics laptops are tricky. There are no guarantees that any of this will work, but especially custom graphics cards on laptops. The discrete GPU may not be directly connected to any of the outputs, so "remoting" the graphics internally may be the only way to get to the guest desktop. It's possible that the GPU does not have a discrete ROM, instead hiding it in ACPI or elsewhere to be extracted. Some hybrid graphics laptops require custom drivers from the vendor. The more integration it has into the system, probably the less likely that it will behave like a discrete desktop GPU.*

That said, there is light on the horizon. Misairu_G (username on forum) published a Guide to VGA passthrough on Optimus laptops. You may want to consult that guide if you use a laptop with Nvidia graphics.

Issues with Skylake CPUs

Another issue has come up with Intel Skylake CPUs. This problem is likely solved by now. Update to a recent kernel (e.g. 4.10), as described above.

In case the kernel upgrade doesn't solve the issue, see https://lkml.org/lkml/2016/3/31/1112 for an available patch. Another possible solution can be found here: https://teksyndicate.com/2015/09/13/wendells-skylake-pc-build-i7-6700k/.

If you haven't found a solution to your problem, check the references in part 15. You are also welcome to leave a comment and I or someone else may come to the rescue.

Part 13 – Run Windows VM in user mode (non-root)

The following tutorial explains how to run the Windows VM in unprivileged mode: https://www.evonide.com/non-root-gpu-passthrough-setup/. I haven't tried it, though.

Part 14 – Passing more PCI devices to guest

If you wish to pass additional PCI devices through to your Windows guest, you must make sure that you pass through all PCI devices residing under the same IOMMU group. Moreover, DO NOT PASS root devices to your guest. To check which PCI devices resider under the same group, use the following command:

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

The output on my system is:

```
/sys/kernel/iommu_groups/0/devices/0000:00:00.0

/sys/kernel/iommu_groups/1/devices/0000:00:01.0

/sys/kernel/iommu_groups/2/devices/0000:00:02.0

/sys/kernel/iommu_groups/3/devices/0000:00:03.0

/sys/kernel/iommu_groups/4/devices/0000:00:05.0

/sys/kernel/iommu_groups/4/devices/0000:00:05.2

/sys/kernel/iommu_groups/4/devices/0000:00:05.4

/sys/kernel/iommu_groups/5/devices/0000:00:11.0

/sys/kernel/iommu_groups/6/devices/0000:00:16.0

/sys/kernel/iommu_groups/7/devices/0000:00:19.0

/sys/kernel/iommu_groups/8/devices/0000:00:1a.0

/sys/kernel/iommu_groups/9/devices/0000:00:1c.0

/sys/kernel/iommu_groups/10/devices/0000:00:1c.1

/sys/kernel/iommu_groups/11/devices/0000:00:1c.2

/sys/kernel/iommu_groups/12/devices/0000:00:1c.3

/sys/kernel/iommu_groups/13/devices/0000:00:1c.4

/sys/kernel/iommu_groups/14/devices/0000:00:1c.7

/sys/kernel/iommu_groups/15/devices/0000:00:1d.0

/sys/kernel/iommu_groups/16/devices/0000:00:1e.0

/sys/kernel/iommu_groups/17/devices/0000:00:1f.0

/sys/kernel/iommu_groups/17/devices/0000:00:1f.2

/sys/kernel/iommu_groups/17/devices/0000:00:1f.3

/sys/kernel/iommu_groups/18/devices/0000:01:00.0

/sys/kernel/iommu_groups/18/devices/0000:01:00.1

/sys/kernel/iommu_groups/19/devices/0000:02:00.0

/sys/kernel/iommu_groups/19/devices/0000:02:00.1

/sys/kernel/iommu_groups/20/devices/0000:05:00.0

/sys/kernel/iommu_groups/20/devices/0000:06:04.0

…
```

As you can see in the above list, some IOMMU groups contain multiple devices on the PCI bus. I wanted to see which devices are in IOMMU group 17 and used the PCI bus ID:

```
lspci -nn | grep 00:1f.
```

Here is what I got:

```
00:1f.0 ISA bridge [0601]: Intel Corporation C600/X79 series chipset LPC
Controller [8086:1d41] (rev 05)
00:1f.2 SATA controller [0106]: Intel Corporation C600/X79 series chipset 6-Port
SATA AHCI Controller [8086:1d02] (rev 05)
00:1f.3 SMBus [0c05]: Intel Corporation C600/X79 series chipset SMBus Host
Controller [8086:1d22] (rev 05)
```

All of the listed devices are used by my Linux host:
– The ISA bridge is a standard device used by the host. You do not pass it through to a guest!
– All my drives are controlled by the host, so passing through a SATA controller would be a very bad idea!
– Do NOT pass through a host controller, such as the C600/X79 series chipset SMBus Host Controller!

In order to pass through individual PCI devices, edit the VM startup script and insert the following code underneath the vmname=… line:

```
configfile=/etc/vfio-pci.cfg

vfiobind() {
dev="$1"
vendor=$(cat /sys/bus/pci/devices/$dev/vendor)
device=$(cat /sys/bus/pci/devices/$dev/device)
if [ -e /sys/bus/pci/devices/$dev/driver ]; then
echo $dev > /sys/bus/pci/devices/$dev/driver/unbind
fi
echo $vendor $device > /sys/bus/pci/drivers/vfio-pci/new_id

}
```

Underneath the line containing "else", insert:

```
cat $configfile | while read line;do
echo $line | grep ^# >/dev/null 2>&1 && continue
vfiobind $line
done
```

You need to create a vfio-pci.cfg file in /etc containing the PCI bus numbers as follows:

```
0000:00:1a.0
0000:08:00.0
```

Make sure the file does NOT contain any blank line(s). **Replace the PCI IDs with the ones you found!**

Part 15 – References

For documentation on qemu/kvm, see the following directory on your Linux machine: `/usr/share/doc/qemu-system-common`

https://passthroughpo.st/ – a new "online news publication with a razor focus on virtualization and linux gaming, as well as developments in open source technology"

https://davidyat.es/2016/09/08/gpu-passthrough/ – a well written tutorial offering qemu script and virt-manager as options.

http://mathiashueber.com/amd-ryzen-based-passthrough-setup-between-xubuntu-16-04-and-windows-10/ – nice and detailed tutorial for a Ryzen based system using the Virtual Machine Manager GUI

https://ycnrg.org/vga-passthrough-with-ovmf-vfio/ – a new Ubuntu 16.04 tutorial using virt-manager.

https://qemu.weilnetz.de/doc/qemu-doc.html – QEMU user manual

https://bbs.archlinux.org/viewtopic.php?id=162768 – this gave me the inspiration – the best thread on kvm!

http://ubuntuforums.org/showthread.php?t=2266916 – Ubuntu tutorial.

https://wiki.archlinux.org/index.php/QEMU – Arch Linux documentation on QEMU – by far the best.

https://wiki.archlinux.org/index.php/PCI_passthrough_via_OVMF – PCI passthrough via OVMF tutorial for Arch Linux – provides excellent information.

https://aur.archlinux.org/cgit/aur.git/tree/?h=linux-vfio – a source for the ACS and i915 arbiter patches.

http://vfio.blogspot.com/2014/08/vfiovga-faq.html – one of the developers, Alex provides invaluable information and advice.

http://vfio.blogspot.com/2014/08/primary-graphics-assignment-without-vga.html

http://www.linux-kvm.org/page/Tuning_KVM – Redhat is the key developer of kvm, their website has lots of information.

https://wiki.archlinux.org/index.php/KVM – Arch Linux KVM page.

https://www.suse.com/documentation/sles11/book_kvm/data/part_2_book_book_kvm.html – Suse Linux documentation on KVM – good reference.

https://www.evonide.com/non-root-gpu-passthrough-setup/ – haven't tried it, but looks like a good tutorial.

https://forum.level1techs.com/t/gta-v-on-linux-skylake-build-hardware-vm-passthrough/87440 – tutorial with Youtube video to go along, very useful and up-to-date, including how to apply ACS override patch.

Below is the VM startup script I use, for reference only.
Note: **The script is specific for my hardware. Don't use it without modifying it!**

```
#!/bin/bash
```

```
configfile=/etc/vfio-pci.cfg
vmname="win10vm"

vfiobind() {
dev="$1"
vendor=$(cat /sys/bus/pci/devices/$dev/vendor)
device=$(cat /sys/bus/pci/devices/$dev/device)
if [ -e /sys/bus/pci/devices/$dev/driver ]; then
echo $dev > /sys/bus/pci/devices/$dev/driver/unbind
fi
echo $vendor $device > /sys/bus/pci/drivers/vfio-pci/new_id

}

if ps -A | grep -q $vmname; then
zenity --info --window-icon=info --timeout=15 --text="$vmname is already running." &
exit 1

else

cat $configfile | while read line;do
echo $line | grep ^# >/dev/null 2>&1 && continue
vfiobind $line
done

# use pulseaudio
#export QEMU_AUDIO_DRV=pa
#export QEMU_PA_SAMPLES=8192
#export QEMU_AUDIO_TIMER_PERIOD=99
#export QEMU_PA_SERVER=/run/user/1000/pulse/native

#use ALSA
export QEMU_AUDIO_DRV=alsa
export QEMU_ALSA_ADC_BUFFER_SIZE=1024 QEMU_ALSA_ADC_PERIOD_SIZE=256
export QEMU_ALSA_DAC_BUFFER_SIZE=1024 QEMU_ALSA_DAC_PERIOD_SIZE=256
export QEMU_AUDIO_DAC_FIXED_SETTINGS=1
export QEMU_AUDIO_DAC_FIXED_FREQ=44100 QEMU_AUDIO_DAC_FIXED_FMT=S16
QEMU_AUDIO_ADC_FIXED_FREQ=44100 QEMU_AUDIO_ADC_FIXED_FMT=S16
export QEMU_AUDIO_DAC_TRY_POLL=1 QEMU_AUDIO_ADC_TRY_POLL=1
export QEMU_AUDIO_TIMER_PERIOD=50

cp /usr/share/OVMF/OVMF_VARS.fd /tmp/my_vars.fd

taskset -c 0-9 qemu-system-x86_64 \
-monitor stdio \
-serial none \
-parallel none \
-nodefaults \
-nodefconfig \
-name $vmname,process=$vmname \
-machine q35,accel=kvm,kernel_irqchip=on,mem-merge=off \
-cpu
host,kvm=off,hv_vendor_id=1234567890ab,hv_vapic,hv_time,hv_relaxed,hv_spinlocks=
0x1fff \
-smp 12,sockets=1,cores=6,threads=2 \
-m 20G \
-mem-path /run/hugepages/kvm \
-mem-prealloc \
-balloon none \
-rtc base=localtime,clock=host \
-vga none \
-nographic \
-soundhw hda \
-device vfio-pci,host=02:00.0,multifunction=on \
-device vfio-pci,host=02:00.1 \
-device vfio-pci,host=00:1a.0 \
-device vfio-pci,host=08:00.0 \
-drive if=pflash,format=raw,readonly,file=/usr/share/OVMF/OVMF_CODE.fd \
-drive if=pflash,format=raw,file=/tmp/my_vars.fd \
-boot order=c \
-drive
```

```
id=disk0,if=virtio,cache=none,format=raw,aio=native,file=/dev/mapper/lm13-win10
\
 -drive
id=disk1,if=virtio,cache=none,format=raw,aio=native,file=/dev/mapper/photos-
photo_stripe \
 -drive
id=disk2,if=virtio,cache=none,format=raw,aio=native,file=/dev/mapper/media-
photo_raw \
 -drive file=/media/user/tmp_stripe/OS-backup/ISOs/virtio-win-
0.1.140.iso,index=3,media=cdrom \
 -drive file=/media/user/tmp_stripe/OS-
backup/ISOs/Win10_1511_English_x64.iso,index=4,media=cdrom \
 -netdev type=tap,id=net0,ifname=vmtap0,vhost=on \
 -device virtio-net-pci,netdev=net0,mac=00:16:3e:00:01:01

 #EOF

 #-drive file=/media/user/tmp_stripe/OS-
backup/ISOs/Win10_1511_English_x64.iso,index=1,media=cdrom \
 #-drive file=/media/user/tmp_stripe/OS-backup/ISOs/virtio-win-
0.1.140.iso,index=2,media=cdrom \

 exit 0
 fi
```

The command

```
 taskset -c 0-9 qemu-system-x86_64...
```

pins the vCPUs of the guest to processor threads 0-9 (I have a 6-core CPU with 2 threads per core=12 threads). I assign 10 out of 12 threads to the guest. While the guest is running, the host will have to make due with only 1 core (2 threads). This CPU pinning may improve performance of the guest.

Note: I am currently passing through all cores and threads, without CPU pinning. This seems to give me the best results in the benchmarks, as well as real-life performance.

Part 16 – Related Posts

Here a list of related posts:

Developments in Virtualization

Virtual Machines on Userbenchmark

qemu-system-x86_64 Drive Options

Part 17 – Benchmarks

Here the UserBenchmark results for my configuration:

UserBenchmarks: Game 60%, Desk 71%, Work 64%
CPU: Intel Core i7-3930K – 79.7%
GPU: Nvidia GTX 970 – 60.4%
SSD: Red Hat VirtIO 140GB – 74.6%
HDD: Red Hat VirtIO 2TB – 64.6%
HDD: Red Hat VirtIO 2TB – 66.1%
RAM: QEMU 20GB – 98.2%
MBD: QEMU Standard PC (Q35 + ICH9, 2009)

Part 18 – Performance Tuning

This chapter is in the making, so expect more tips to be added here in the future.

Turn on MSI Message Signaled Interrupts in your VM

Developer Alex Williamson argues that MSI Message Signaled Interrupts may provide a more efficient way to handle interrupts. A detailed description on how to turn on MSI in a Windows VM can be found here: Line-Based vs. Message Signaled-Based Interrupts.

Make sure to backup your entire Windows installation, or at least define a restore point for Windows.

In my case it improved sound quality (no more crackle), others have reported similar results – see these comments.

Share this:

2

Like this:

☆ Like

Be the first to like this.

Related

Developments in Virtualization

January 8, 2018
In "Virtualization"

Why run Windows on Linux?

April 8, 2018
In "Linux"

Glossary of Virtualization Terms

July 20, 2017
In "Linux"

Heiko Sieger / July 20, 2017 / Linux, Virtualization / kvm, Linux, linux mint, ovmf, qemu, ubuntu, uefi, vga passthrough, virtualization, windows

4 thoughts on "Running Windows 10 on Linux using KVM with VGA Passthrough"

Pingback: IOMMU Groups – What You Need to Consider – Heiko's Blog

mathias

December 17, 2017 at 16:50

Moin Heiko,
thank you for the detailed write up.
well done!
You mind sharing your virsh windows10.xml ?

Heiko Sieger

January 6, 2018 at 03:12

Hello Mathias,
I have been traveling a lot lately, so forgive my late reply. In answer to your question: I don't use a virsh windows10.xml file, just the script and qemu. Most other tutorials use libvirt and virt-manager to create and manage the VM, but I had not much luck with it.

Heiko

Heiko Sieger

April 9, 2018 at 12:53

Just posted a Passmark 9 CPU benchmark over at the Tom's Hardware forum: http://www.tomshardware.com/answers/id-3610702/tom-hardware-passmark-cpu-benchmark-thread/page-3.html#20869750

It shows that it compares well with other builds using the same CPU (i7-3930k in my case).