

# A streamlined VFIO+LookingGlass setup guide

Note: Guide is written primarily for Ubuntu 18.04 and newer

<b>Preamble (pre-ramble)</b>	<b>2</b>
<b>Why and what is this for? Why bother?</b>	<b>3</b>
<b>Setting everything up.</b>	<b>4</b>
Installing QEMU and Virt-Manager	4
Enable IOMMU on your system	4
Make a ls-iommu script just to make your life easier	5
Isolating/stubbing your GPU for passthrough	6
Tell KVM to ignore MSRS to avoid a BSOD on newer Windows 10 builds (1803+)	7
Edit apparmor permissions to give libvirt device access	8
<b>Making the virtual machine</b>	<b>9</b>
Using a physical disk for the virtual machine	10
Installing the VirtIO drivers and enabling the VirtIO bus	11
Enabling TRIM support for SSD passthrough	12
Passing through the GPU	13
Configure the virtual machine so the Nvidia driver will work	13
<b>Quality of life tweaks</b>	<b>14</b>
Building LookingGlass	14
Local “host-only” connection between Host and Guest	15
Software based KVM switch	16
Sound through headphones/speakers or with LookingGlass	17
Sound through virt-viewer (easy)	17
Stream audio through Screamer over the host-only network (easy, multiple users)	18
Sound through PulseAudio (single user: easy, multiple users: insecure)	19
Fixing audio crackle (moderate, takes time)	20
Game controllers and USB controllers	23
Using Moonlight Streaming to “pass through” a game controller	24
Enabling OpenGL support in flatpak for Nvidia Proprietary driver	24
Installing & configuring the Moonlight Streaming Client	25
Launch specific programs on specific cores in Windows	28
<b>Performance tuning &amp; exotics</b>	<b>29</b>
Pinning CPU cores to the VM for more performance	30
NUMA tuning for systems with multiple NUMA nodes	31
Isolate specific CPU cores for use only by the Virtual Machine	33
Increase gaming performance by changing clocksource to tsc	34
Newer mainline kernels and kernel drivers on Ubuntu	35

# Preamble (pre-ramble)

Hello, my name is Ove, also known as HikariKnight.

I am a mostly unemployed IT worker and have been so the last 6 years, I have a passion for scripting, automation, virtualization, Linux and cross platform development.

Overall I have about 15 years of experience in this through just studying in my personal time since I was 13.

I also have dyslexia, so if something is not understandable for you, feel free to @ me on twitter

<https://twitter.com/TheHikariKnight>

I decided to write this guide to make it easier for people to get VFIO GPU passthrough set up and running and properly configured, as I noticed all the guides out there stop after you have the virtual machine running with GPU passthrough and do not show how to improve it or maintain it if you are planning on using it permanently.

Like in some cases related to SSD passthrough, they do not notify users that they need to configure their SSD as a SCSI device and make the virtual machine use the virtio-scsi controller in order to enable TRIM support.

I also noticed all the guides I have seen, lack general quality of life tips and tricks as GPU passthrough is not super seamless by default.

During this guide you will encounter 3 types of tables.

The terminal

Text appearing like this, means it should be run in a terminal

File edit

Text in these boxes are meant to show what you should type in or copy+paste into files

The table below is used when there are different configurations to do based on hardware, you only need to focus on what the box below the configuration that matches your setup tells you to do, the boxes below the configuration header can be either a terminal box(black) or a file edit box(light grey).

Configuration 1	Configuration 2
Things to do for config 1	Things to do for config 2

Support me on patreon if you feel like it, sadly my release schedule is spotty at best due to real life.

<https://www.patreon.com/HikariKnight>

# Why and what is this for? Why bother?

Dual booting is annoying, constantly having to reboot (and possibly wait through Windows Update deciding it is time to force you to install an update) to the operating system you need, depending on what you want or need to do.

And then there is just so much that Wine, Lutris and Proton/SteamPlay can do, not everything will run through those.

Finally, virtualization tech and driver support has finally gotten to a point that it makes sense to do this now and to teach people how to do this without forcing them to learn how everything works internally.

Running programs inside a virtual machine will give you full compatibility, except for programs that rely on PCI-e hardware which is not supported in Linux, or maybe you are wanting to play a Windows game that does not work in Wine or Photon.

Passing that device through to the virtual machine using VFIO and PCI passthrough will then give the virtual machine full access to that hardware.

Doing this to a graphic card will then give you the ability to run Linux as your main operating system, while using Windows running inside Linux for playing windows games or running Windows only software with full hardware accelerated graphics and in most cases with less than 1% performance loss on modern hardware, on older hardware you may get a slight performance loss.

Sadly there may be more costs associated with it on some systems if you lack storage space or a 2nd graphics card, or lack an Internal GPU or APU, but it is a worthy price to pay if you rather want to run Linux as your main operating system but can't due to that one program or because all your games do not work in Linux.

So if you are like me and want to or prefer to use Linux, but still rely on Windows for some things, I have written this guide specifically for people like us!

It covers everything from setting up the system for VFIO, to fine tuning the system for your use case and specifications!

I really hope you will get good use of this guide and document.

And I hope this will make your life easier when it comes to using both systems at once.

Some information for this guide comes from other sources while most is from personal experience and is publicly available in several locations in bits and pieces or as incomplete to somewhat full guides.

Sources for some of this guides information:

Level1Techs - <https://forum.level1techs.com/>

LinuxUprising - <https://www.linuxuprising.com>

Rokups Tech Notebook - <https://rokups.github.io/>

My own personal experience

## Setting everything up.

1. Make sure virtualization is enabled in your UEFI/BIOS, it might be named VT-d or AMD-v or SVM, this is needed to literally even get started with anything here.
2. Your OS must be installed in UEFI mode too.
3. Only desktop computers can use VFIO easily, laptops can not use GPU Passthrough with VFIO reliably (really depends on how the laptop is assembled) and should not be attempted unless you know what you are doing.

## Installing QEMU and Virt-Manager

1. Open your terminal and install all the needed packages  
Note: Keep the terminal open for the remainder of the guide.



```
sudo apt install virt-manager qemu-kvm ovmf
```

## Enable IOMMU on your system

1. Edit “/etc/default/grub” and tell the kernel to enable IOMMU at boot  
Note: there are different parameters for intel and amd!

```
sudo nano /etc/default/grub
```

2. Add the below parameters based on your CPU manufacturer at the end of the GRUB\_CMDLINE\_LINUX\_DEFAULT= but make sure they are inside the quotes so it looks something similar to  
GRUB\_CMDLINE\_LINUX\_DEFAULT="quiet splash iommu=1 intel\_iommu=on"

AMD 	INTEL 
iommu=1 amd_iommu=on	iommu=1 intel_iommu=on

3. Press CTRL+X then Enter/Return to save.
4. Update the grub config with the below command.

```
sudo update-grub
```

5. Reboot your system.

## Make a ls-iommu script just to make your life easier

1. Create the file /usr/local/bin/ls-iommu with.

```
sudo nano /usr/local/bin/ls-iommu
```

2. Paste the below bash script into the file and save with CTRL+X

Note: Code gotten from [here](#) originally by Wendell from Level1Techs.

```
#!/bin/bash
for d in /sys/kernel/iommu_groups/*/devices/*; do
    n=${d#/iommu_groups/*}; n=${n%%/*}
    printf 'IOMMU Group %s ' "$n"
    lspci -nns "${d##*/}"
done
```

3. Now make the script executable with

```
sudo chmod +x /usr/local/bin/ls-iommu
```

4. The script is now executable and you will be able to run it with.

```
ls-iommu
```

5. The output should look something like this.

```
IOMMU Group 32 41:00.1 Audio device [0403]: Advanced Micro Devices, Inc. [AMD/ATI] Ellesmere [Radeon RX 580] [1002:aaf0]
IOMMU Group 33 42:00.0 Non-Essential Instrumentation [1300]: Advanced Micro Devices, Inc. [AMD] Device [1022:145a]
IOMMU Group 34 42:00.2 Encryption controller [1080]: Advanced Micro Devices, Inc. [AMD] Family 17h (Models 00h-0fh) Platform Security Processor [1022:1456]
IOMMU Group 35 42:00.3 USB controller [0c03]: Advanced Micro Devices, Inc. [AMD] Family 17h (Models 00h-0fh) USB 3.0 Host Controller [1022:145c]
IOMMU Group 36 43:00.0 Non-Essential Instrumentation [1300]: Advanced Micro Devices, Inc. [AMD] Device [1022:1455]
IOMMU Group 37 43:00.2 SATA controller [0106]: Advanced Micro Devices, Inc. [AMD] FCH SATA Controller [AHCI mode] [1022:7901] (rev 51)
IOMMU Group 3 00:02.0 Host bridge [0600]: Advanced Micro Devices, Inc. [AMD] Family 17h (Models 00h-0fh) PCIe Dummy Host Bridge [1022:1452]
IOMMU Group 4 00:03.0 Host bridge [0600]: Advanced Micro Devices, Inc. [AMD] Family 17h (Models 00h-0fh) PCIe Dummy Host Bridge [1022:1452]
IOMMU Group 5 00:03.1 PCI bridge [0604]: Advanced Micro Devices, Inc. [AMD] Family 17h (Models 00h-0fh) PCIe GPP Bridge [1022:1453]
IOMMU Group 6 00:04.0 Host bridge [0600]: Advanced Micro Devices, Inc. [AMD] Family 17h (Models 00h-0fh) PCIe Dummy Host Bridge [1022:1452]
IOMMU Group 7 00:07.0 Host bridge [0600]: Advanced Micro Devices, Inc. [AMD] Family 17h (Models 00h-0fh) PCIe Dummy Host Bridge [1022:1452]
IOMMU Group 8 00:07.1 PCI bridge [0604]: Advanced Micro Devices, Inc. [AMD] Family 17h (Models 00h-0fh) Internal PCIe GPP Bridge 0 to Bus B [1022:1454]
IOMMU Group 9 00:08.0 Host bridge [0600]: Advanced Micro Devices, Inc. [AMD] Family 17h (Models 00h-0fh) PCIe Dummy Host Bridge [1022:1452]
hk@hk-TR1920X ~$
```

6. Make sure that the VGA device you want to passthrough is in its own group along with its audio device, like in this example for my GTX 1070 ti.

```
IOMMU Group 15 09:00.0 USB controller [0c03]: VIA Technologies, Inc. VL805 USB 3.0 Host Controller [1106:3483] (rev 01)
IOMMU Group 16 0a:00.0 VGA compatible controller [0300]: NVIDIA Corporation GP104 [GeForce GTX 1070 Ti] [10de:1b82] (rev a1)
IOMMU Group 16 0a:00.1 Audio device [0403]: NVIDIA Corporation GP104 High Definition Audio Controller [10de:10f0] (rev a1)
IOMMU Group 17 0b:00.0 Non-Essential Instrumentation [1300]: Advanced Micro Devices, Inc. [AMD] Device [1022:145a]
```

If the VGA controller and accompanying audio device are not in their own group together with nothing else in the group, you will have to apply the ACS override patch to the kernel, which is outside the scope of this guide as I have not personally needed it yet.

If anyone knows how to apply it then message me on twitter and I will gladly include it!



<https://twitter.com/TheHikariKnight>

## Isolating/stubbing your GPU for passthrough



Warning: Before continuing with this, make sure you have a second, different model of GPU available, this can be an internal GPU like an Intel GPU on your CPU or a second dedicated Graphics Card (that is a different model! Or you might run into issues from my experience), make sure this GPU is used as your current Linux Graphics output! As the instructions below will make the stubbed GPU not usable in your Linux system until you undo this!

For simplicity and to avoid unnecessary files, this guide will only use the modprobe approach to stub the GPU, other distributions may or may not use a different approach.

1. Depending on the card you want to pass through being Nvidia or AMD please create the modprobe config file for that gpu.

Nvidia GPU Passthrough 	AMD GPU Passthrough 
<code>sudo nano /etc/modprobe.d/nvidia.conf</code>	<code>sudo nano /etc/modprobe.d/amdgpu.conf</code>

2. Paste in the below content based on your card to make sure vfio is loaded before your graphic card driver, then save the file with CTRL+X.

Nvidia GPU Passthrough 	AMD GPU Passthrough 
<code>softdep nvidia pre: vfio vfio_pci</code> <code>softdep nouveau pre: vfio vfio_pci</code>	<code>softdep amdgpu pre: vfio vfio_pci</code>

3. Now we need to find the PCI-e ID for the graphic card (and its audio controller) that we will be passing through to the virtual machine. To find it we will use the ls-iommu script we made earlier.

```
ls-iommu | grep -iP "VGA|audio"
```

4. This should list up any VGA and audio devices, for my example I will be passing through my Nvidia GTX 1070 ti, I have highlighted the PCI-e ID in the output so you know what you are looking for (make sure you grab the ID for both the VGA device and the Audio device that belongs to the graphic card, you will need both!!).

```
rk@rk:~$ ls-iommu | grep -iP "VGA|audio"
IOMMU Group 16 0a:00:0 VGA compatible controller [0300]: NVIDIA Corporation GP104 [GeForce GTX 1070 Ti] [10de:1b82] (rev a1)
IOMMU Group 16 0a:00:1 Audio device [0403]: NVIDIA Corporation GP104 High Definition Audio Controller [10de:10f0] (rev a1)
IOMMU Group 22 0c:00:3 Audio device [0403]: Advanced Micro Devices, Inc. [AMD] Family 17h (Models 00h-0fh) HD Audio Controller [1022:1457]
IOMMU Group 32 41:00:0 VGA compatible controller [0300]: Advanced Micro Devices, Inc. [AMD/ATI] Ellesmere [Radeon RX 470/480/570/570X/580/580X] [1002:67df] (rev ef)
IOMMU Group 32 41:00:1 Audio device [0403]: Advanced Micro Devices, Inc. [AMD/ATI] Ellesmere [Radeon RX 580] [1002:aaf0]
```

If it is not noticable in the image, I am interested in the IDs which are inside the brackets at the end of the lines with the Nvidia device, these being **10de:1b82** and **10de:10f0** for me.

5. Make a modprobe config file for vfio.

```
sudo nano /etc/modprobe.d/vfio_pci.conf
```

6. Paste the following into the file (replace my PCI-e IDs with your own!)

Then save the file with CTRL+X.

```
options vfio_pci ids=10de:1b82,10de:10f0
```

7. Reboot your computer then verify that the graphic card you stubbed is being used by the vfio-pci kernel driver using the below command.

You are looking for the line “Kernel driver in use: vfio-pci” on the graphic card and its audio device that you stubbed.

```
lspci -vnn | grep -iP "vga|amdgpu|nvidia|nouveau|vfio-pci"
```

## Tell KVM to ignore MSRS to avoid a BSOD on newer Windows 10 builds (1803+)

1. Make a new modprobe file for kvm.

```
sudo nano /etc/modprobe.d/kvm.conf
```

2. Paste in the content from the box below.

The first line tells kvm to ignore MSRS while the second line stops the kvm from spamming notifications into the tty that it has ignored MSRS.

```
options kvm ignore_msrs=1  
options kvm report_ignored_msrs=0
```

3. Save the file by pressing CTRL+X.

## Edit apparmor permissions to give libvirt device access

1. Edit the abstractions file for libvirt-qemu.

```
sudo nano /etc/apparmor.d/abstractions/libvirt-qemu
```

2. Press CTRL+W and search for “usb access” without the quotes and make sure that section looks like the box below to enable full USB access.

```
# for usb access
/dev/bus/usb/** rw,
/etc/udev/udev.conf r,
/sys/bus/ r,
/sys/class/ r,
/run/udev/data/* rw,
```

3. Now go to the bottom of the file and add the permissions for looking-glass (which we will install later!).

```
# Looking Glass
/dev/shm/looking-glass rw,
```

4. If you are planning to pass through a RAW block device (like an SSD) you can add them to the bottom of the file too, however instead of using /dev/sdX I will highly recommend you use the persistent names for the block devices instead, to avoid accidentally passing the wrong device if the letter changes after a reboot or change to the physical system (which can happen!).

I personally use the WWID symbolic links for the block devices inside /dev/disk/by-id/ as this is guaranteed to be persistent and unique per storage device!

The below box is an example based on me passing through my SSD.

I would then use the same path I gave apparmor when I pass through that SSD, and it will guarantee that only that disk gets used for the virtual machine.

```
# Raw Disk Access
/dev/disk/by-id/ata-SAMSUNG_MZ7TD128HAFV-000L1_S14TNSAD626520 rw,
```

5. Once done, save the file with CTRL+X and restart apparmor with the below command.

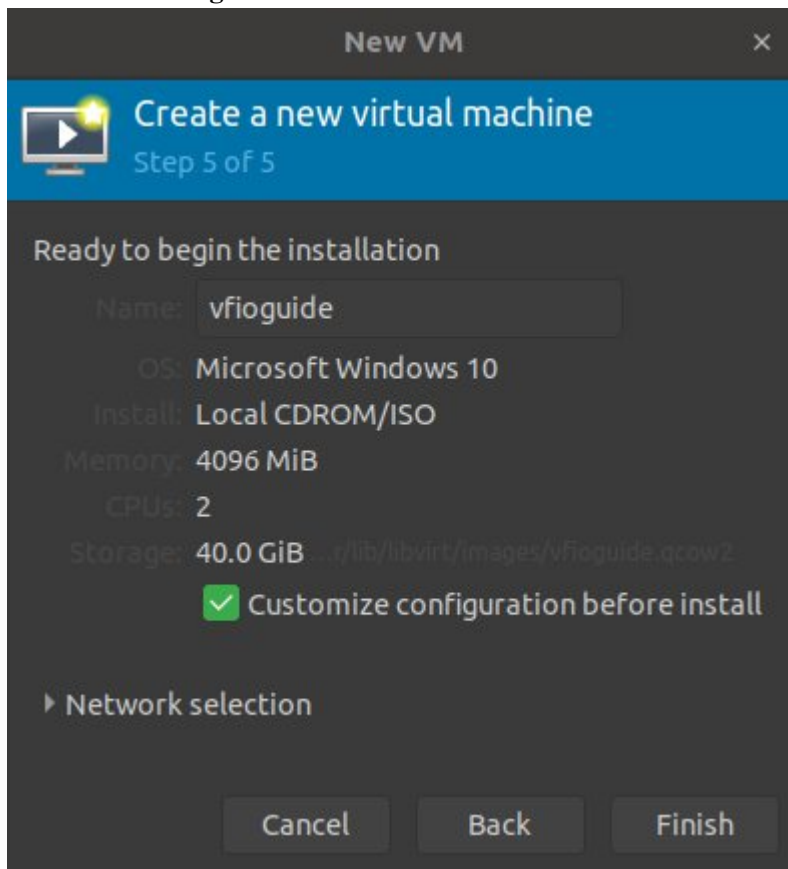
```
sudo service apparmor restart
```



# Making the virtual machine

Note: If you have been following this guide until now, reboot your computer to make sure all your changes get applied and libvirt and QEmu gets properly started.

1. Open the Virtual Machine Manager and make a new virtual machine.  
Note: if you are planning to use a physical disk device, make a small qcow2 disk as you will need it later.
2. At the end of the configuration wizard, make sure you check the box for **Customize configuration before install**.



3. Switch the firmware from BIOS to UEFI (this cannot be changed through the gui once the virtual machine is created!)
4. Make sure the chipset is set to i440FX (q35 can work but is a lot harder to setup)
5. Click on CPUs and click Topology and check the checkbox for Manually set CPU topology.
6. The best topology varies from system to system and use case to use case, but on my personal system (Threadripper 1920x) I use 1 socket with 6 cores and 1 thread per core.  
On most systems it might be preferable to use 1 socket with 2 or 4 cores and 1 thread per core.
7. Click on IDE Disk 1 and change the disk bus to SATA for now.
8. Click Begin Installation and install windows as usual, then turn off UAC(User Account Control)if you are planning to use LookingGlass.

Note: Force off the virtual machine if you are planning to use a physical disk then read the next section before actually installing windows.

## Using a physical disk for the virtual machine

Warning: I have gotten extremely mixed results with usb (both 2.0 and 3.x) attached disks (including those in powered docking stations!) and would not recommend it due to the potential of data corruption, only use disks attached directly to the motherboard through sata, m.2 or pci-e!

1. Open the configuration file for the virtual machine.

This guide will assume you are using nano as your editor, if you are comfortable using any of the other editors then use those, nano is just the easiest for beginners to use.

Note: replace yourvirtualmachinename with the name of your virtual machine.

```
virsh edit yourvirtualmachinename
```

2. Locate this part of the file that is talking about disk and qcow2, this is the disk you made in the configuration wizard.

```
<disk type='file' device='disk'>
  <driver name='qemu' type='qcow2' />
  <source file='/var/lib/libvirt/images/vfloguide.qcow2' />
  <target dev='sda' bus='sata' />
  <address type='drive' controller='0' bus='0' target='0' unit='0' />
</disk>
```

3. To make the virtual machine use the raw disk of your choice you want to replace it with this, and remember to use the same file you added to the apparmor configuration as the source device.

```
<disk type='block' device='disk'>
  <driver name='qemu' type='raw' />
  <source dev='/dev/disk/by-id/ata-SAMSUNG_MZ7TD128HAFV-000L1_S14TNSAD626520' />
  <target dev='sda' bus='sata' />
</disk>
```

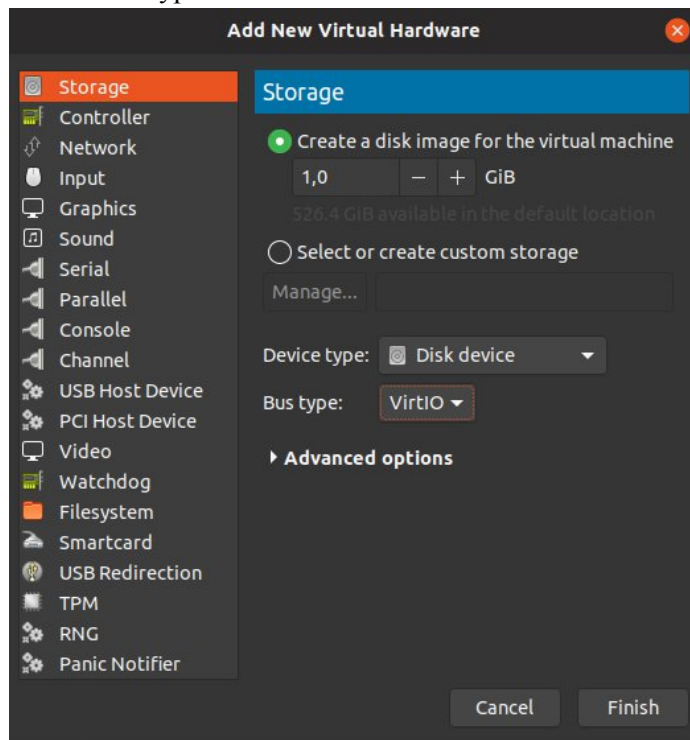
4. **WARNING:** If you are using an SSD as the physical device, it is very important you head to the **Enabling TRIM support for SSD passthrough** section of this guide once Windows and the VirtIO drivers are installed. Otherwise your SSD will be detected as a HDD by Windows 10 and will slowly get destroyed by unnecessary read/writes by the windows defragmentation tool schedule, plus your SSD will never run TRIM which is a bad thing!

## Installing the VirtIO drivers and enabling the VirtIO bus

1. After Windows (preferably Windows 10 if you plan to use LookingGlass) is installed, install the VirtIO drivers from <https://docs.fedoraproject.org/en-US/quick-docs/creating-windows-virtual-machines-using-virtio-drivers/index.html>

If you mount the iso through the virtual machine or directly in windows is up to you.

2. Install all the drivers, especially the VIO drivers  
Note: Drivers without an exe or msi file for install come as .inf files, right click on the .inf file and choose install in order to install it on your system.
3. Shut down (not reboot) your virtual machine.
4. Open your Virtual Machine's hardware details in Virtual Machine Manager and click Add Hardware.
5. Add a new storage disk image (or reuse the one from the configuration wizard if you are using a physical disk instead).
6. Set the Bus type to VirtIO and click finish.



7. Start your Virtual Machine, once Windows has finished loading, shut the machine down again and change the Bus type for Disk 1 to VirtIO.
8. Start your Virtual Machine again, if Windows loads without issues, then VirtIO is working fine, if not then set the Disk 1 Bus type back to SATA and re-install the VirtIO drivers for storage while Disk 2 set to VirtIO is present on the machine.
9. Once everything is working, set your virtual network card to use virtio as the device model.

## Enabling TRIM support for SSD passthrough

Note: Make sure you have installed the VirtIO drivers and configured your Disk 1 to use the VirtIO bus

1. Shut down the Virtual Machine.
2. Open the Virtual Machine configuration.

```
virsh edit yourvirtualmachinename
```

3. Locate your physical disk configuration, it should look something like this now.

```
<disk type='block' device='disk'>  
  <driver name='qemu' type='raw' cache='none' io='native'/>  
  <source dev='/dev/disk/by-id/ata-SAMSUNG_MZ7TD128HAFV-000L1_S14TNSAD626520'/>  
  <target dev='sda' bus='virtio'/>  
  <address type='drive' controller='0' bus='0' target='0' unit='0'/>  
</disk>
```

4. Add discard='unmap' to the end of the driver tag and edit bus='virtio' to bus='scsi'  
The end result should look something like this.

```
<disk type='block' device='disk'>  
  <driver name='qemu' type='raw' cache='none' io='native' discard='unmap'/>  
  <source dev='/dev/disk/by-id/ata-SAMSUNG_MZ7TD128HAFV-000L1_S14TNSAD626520'/>  
  <target dev='sda' bus='scsi'/>  
  <address type='drive' controller='0' bus='0' target='0' unit='0'/>  
</disk>
```

5. Now we need to add a SCSI controller to our configuration, add this below the last </controller> tag.

```
<controller type='scsi' index='0' model='virtio-scsi'>  
</controller>
```

6. Press CTRL+X to save when done.
7. What this does is mark the disk as a Thin Provisioned Storage, which has TRIM capability (through the SCSI interface) and the discard='unmap' part from the driver tag for our disk tells the physical disk to unmap data marked for deletion when TRIM is run, this is currently the only way I know of to mark a physical disk as an “SSD” with libvirt successfully.

## Passing through the GPU

Once you have installed Windows, installed the virtIO drivers and verified everything is working. It is finally time to pass the GPU to the virtual machine.

1. If the virtual machine is on, shut it down
2. Open the Virtual Machine Manager and go to your virtual machines hardware details
3. Click Add Hardware and choose PCI Host Device
4. Locate the Graphic card you have stubbed earlier and click it, then click Finish
5. Do the same (step 3 and 4) for the Graphic Cards High Definition Audio Controller.
6. Turn on the computer to verify that the device has been passed to the Virtual Machine  
Note: You can check this in the device manager in Windows
7. If you have passed through an AMD Graphics card, you are done and can install the graphic card driver, if you passed through an Nvidia card you will have to read the next section first before installing the graphics card drivers.

## Configure the virtual machine so the Nvidia driver will work

The Nvidia driver will check to see if it is running in a hypervisor and will disable the GPU driver if it detects it, so we will need to hide the hypervisor from it

1. If the virtual machine is on, shut down the virtual machine.
2. If you are passing through an Nvidia graphic card, you will need to make the Nvidia driver believe it is not running on a hypervisor, we can do this by doing a slight edit to our virtual machine configuration.

```
virsh edit yourvirtualmachinename
```

3. Locate the tag ending `</hyperv>` and add this line **above** it.

```
<vendor_id state='on' value='anything'/>
```

4. Then add these lines **below** the `</hyperv>` line.

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

5. Save the file with CTRL+X when you are done.

# Quality of life tweaks

## Building LookingGlass

Note: This guide was originally intended to show how to compile the client for LookingGlass A12, as the instructions were not on the homepage last time I checked, they are now however so follow those!

What is LookingGlass?

The simple answer is that it lets you view the framebuffer from the GPU you are using in the virtual machine, so you get full hardware acceleration, however the Graphic Card still needs to be connected to a display or a “ghost display emulator” so that the graphic card powers on with the virtual machine. This way you do not need to constantly switch monitor output or have a dedicated monitor for the virtual machine.

LookingGlass has a built in SPICE client for seamless keyboard and mouse integration, however it is spotty in its reliability at best when it comes to the cursor. Personally I have found that using a software based KVM switch is a lot more reliable, or just passing through a dedicated keyboard and mouse for the virtual machine.

You can however press Scroll Lock and lock and unlock your cursor to the LookingGlass window, which in most cases solves a lot of the unreliability it has with the cursor, however it does not solve it for every application.

Please note that LookingGlass, as of the time of writing this is very early Alpha to Beta software, however it is far enough in development to be usable and is really good at what it does, with maybe some extra help in certain places!

1. Turn off UAC in Windows as LookingGlass cannot display the Secure Desktop (the UAC prompt) currently (not sure if this is only for passed through NVidia cards as i lack an extra AMD card to test with).
2. Follow the developers guide here: <https://looking-glass.hostfission.com/quickstart>

Note: It would also be nice if you can, to support or donate to the project, more details are on the homepage <https://looking-glass.hostfission.com/>!

## Local “host-only” connection between Host and Guest

1. Make a template file for the isolated network.

```
nano /tmp/isolated.xml
```

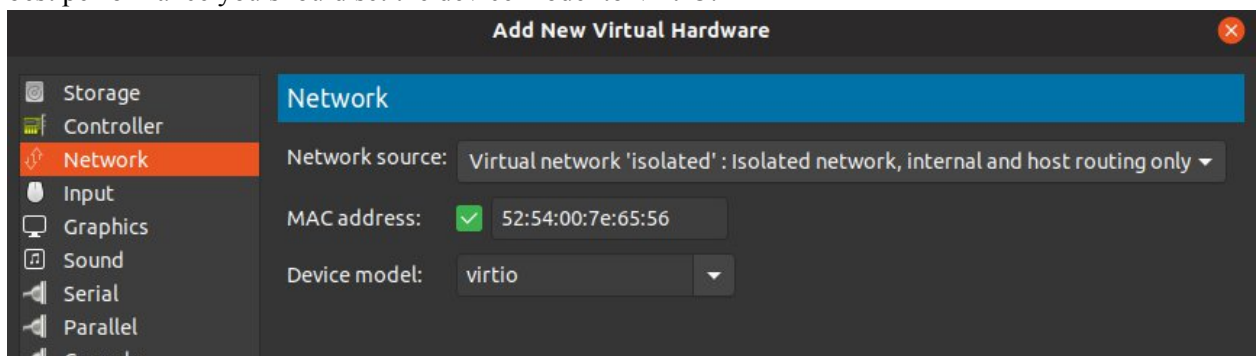
2. Paste in the configuration below (edit it to avoid using the same address range as your network! The configuration provided should not clash with any home network from my experience).

```
<network>
  <name>isolated</name>
  <ip address='10.10.10.1' netmask='255.255.255.0'>
    <dhcp>
      <range start='10.10.10.2' end='10.10.10.254' />
    </dhcp>
  </ip>
</network>
```

3. Save the file with CTRL+X and then run the below commands to import the network config and enable it for use in the Virtual Machine Manager.

```
virsh net-define /tmp/isolated.xml
virsh net-autostart isolated
virsh net-start isolated
```

4. You will now be able to add the isolated virtual network to any new virtual network card, for the best performance you should set the device model to VirtIO.



5. You will now be able to set up a file sharing to share files between the Host and the Guest should you wish to do that, you will also be able to set up the software based kvm switch mentioned later in this document. Host can be reached at 10.10.10.1 if you used the default config.
6. Should you need to edit the isolated network settings in the future, use the below command.

```
virsh net-edit isolated
```

## Software based KVM switch

This is a good alternative to LookingGlass alone or to use in along with LookingGlass by commenting out the DDC related lines of the scripts.

1. In order to get this to work, you need more USB ports on your virtual machine.  
Open the Virtual Machine Manager and go to your virtual machines hardware details.
2. Click on Controller USB 0 and change the model from USB 2 to USB 3 and apply the changes.
3. Open the virtual machine configuration.

```
virsh edit yourvirtualmachinename
```

4. Look for this line.

```
<controller type='usb' index='0' model='nec-xhci'>
```

5. Add ports='10' to the end of the tag, the max amount of ports you can make are 10 from what I remember. The line should in the end look like this.

```
<controller type='usb' index='0' model='nec-xhci' ports='10'>
```

6. Press CTRL+X to save and then continue forward.
7. Follow the guide at <https://rokups.github.io/#!/pages/full-software-kvm-switch.md>  
My personal preference is to use the hotkey CTRL+ALT+SPACEBAR.  
Here is my autohotkey script to replicate that.

```
SendMode Input
#SingleInstance Force
#NoEnv
Process, Priority,, High

^<!Space::
    Run, putty.exe -load "vm-detach",,hide
    return
```



8. If you are using this with LookingGlass client, you would want to disable the spice client and be able to see the host cursor above the LookingGlass window by using the -s and -M parameters!

Note: This solution also works great with [Synergy](#) or [Barrier](#) if you have a license for it, personally I use Synergy1/Barrier as a server on the Windows VM and as a client on the Linux host so that it is only active when the keyboard and mouse are connected directly to the VM, this is due to keyboard layout issues when running Synergy as a server on Linux and due to it not working great in some games on the client side.

## Sound through headphones/speakers or with LookingGlass

### Sound through virt-viewer (easy)

The easiest solution I have found for this is to disable the spice client in the LookingGlass client using the -s parameter along with using the software KVM switch mentioned above and then use a script to open the virt-viewer for the virtual-machine and just minimize it or move it to a different workspace (or both!). For routing audio to PulseAudio instead (but may not work on some systems), check the [ArchLinux Wiki](#).

1. Make a new script in /usr/local/bin.

```
sudo nano /usr/local/bin/vm-sound
```

2. Paste in the following script and replace yourvirtualmachinename with the name of your virtual machine.

```
#!/bin/bash  
virt-viewer -w "yourvirtualmachinename"
```

3. You can also use something like xdotool to move the window out of the way once it is launched, my full script looks something like this.

```
#!/bin/bash  
virt-viewer -w "win10" &  
sleep 10  
xdotool windowminimize $(xdotool search --name "win10 \\\(1\\) \\\- Virt Viewer")
```

4. Once you are done, save the file with CTRL+X and then run the below command to make it executable.

```
sudo chmod +x /usr/local/bin/vm-sound
```

5. Now you can just run “vm-sound” whenever you want to get sound from the virtual machine, it is not perfect but it works.

## Stream audio through Scream over the host-only network (easy, multiple users)

Note: The benefit with this over routing PulseAudio is that you can get sound from a VM that boots with the system, and you can also get better audio quality without crackles and low latency, this also works the same on qemu2.11 and qemu3.0, which means you do not need to build qemu3.0.

1. First in the windows VM you will need to download the scream server and install it from: <https://github.com/duncanthrax/scream/releases>
2. To enable unicast. Open notepad and paste this in and save it as a .reg file and run it, change the IPv4 to point to your Linux host on your host-only network (default port is 4011).

```
Windows Registry Editor Version 5.00

[HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\Scream\Options]
"UnicastIPv4"="10.10.10.1"
"UnicastPort"=dword:00000fab
```

3. Set the windows default playback device to Scream (WDM) and reboot the VM.
4. On the Linux host we will need to build the receiver, the commands below will install the build-essential tools, git and libpulse headers so we can compile the pulseaudio version of scream, it will then download the scream sources and compile scream-pulse.

```
sudo apt install build-essential libpulse-dev git
mkdir devel && cd devel
git clone https://github.com/duncanthrax/scream.git
cd ./scream/Receivers/pulseaudio
make
```

5. Copy the built binary to /usr/local/bin

```
sudo cp -v ./scream-pulse /usr/local/bin/
```

6. Now we just need to start the client (you can add this to your startup applications)  
for me the host-only interface for my host-only network is virbr1, yours might be different (it's the one with the ip you wrote into the reg file earlier)

```
scream-pulse -u -p 4011 -i virbr1
```

## Sound through PulseAudio (single user: easy, multiple users: insecure)

Another more seamless solution is to route audio through PulseAudio directly, removing virt-viewer completely from the equation. This method is a lot nicer and you do not need to keep a virt-viewer window open in the background.

I will write how to do this on a single user, as enabling this for multiple users requires adding more lines to apparmor for security (change 1000 to match other users IDs) and the simple way to make it work for multiple users is to make an anonymous socket which is insecure, as it will let other users listen into your audio while you are logged in, I will not show how to make the anonymous socket for that reason.

1. Edit the qemu config for libvirt

```
sudo nano /etc/libvirt/qemu.conf
```

2. Locate and uncomment the that starts with “user =” and replace the “yourusername” with your username.

```
user = “yourusername”
```

3. Add our socket to apparmor

```
sudo nano /etc/apparmor.d/abstractions/libvirt-qemu
```

4. Add the below part to the bottom of the file (replace 1000 with your user id number, you can find it by typing “id” into the terminal).

Then press CTRL+X to save.

```
# Pulseaudio access
/run/user/1000/pulse/native rw,
```

5. Now we need to edit our virtual machine to add the pulseaudio socket to the virtual machine.

```
virsh edit yourvirtualmachinename
```

6. Replace <domain type='kvm'> with the below line if your virtual machine file starts with that.

```
<domain type='kvm' xmlns:qemu='http://libvirt.org/schemas/domain/qemu/1.0'>
```

7. Now scroll to the bottom and add the below lines between </devices> and </domain> and change 1000 to match the user ID of the user that will be using the virtual machine, if you already have a

<qemu:commandline> section, only add the “<qemu:env” lines inside your existing <qemu:commandline> section.

```
<qemu:commandline>
<qemu:env name='QEMU_AUDIO_DRV' value='pa'/>
<qemu:env name='QEMU_PA_SERVER' value='/run/user/1000/pulse/native'/>
</qemu:commandline>
```

8. Press CTRL+X to save the file and reboot your system (or restart apparmor and pulseaudio, however I prefer a reboot).

## Fixing audio crackle (moderate, takes time)

Qemu 2.xx has audio crackling whenever you run anything CPU intensive, the severeness varies from task to task and in some cases can become unbearable. This is however fixed with a new timing system for the pulseaudio driver in Qemu3.0 and newer. The problem is that it is not available for Ubuntu yet, so we will have to compile Qemu3.1 from source.

1. Enable source repositories by ticking the “source code” checkbox in software-properties-gtk or by removing the # from the deb-src lines for the ubuntu repositories inside your sources.list then press CTRL+X to save the file.

```
sudo nano /etc/apt/sources.list
```

2. Get the source dependencies to compile qemu.

```
sudo apt-get update && sudo apt-get build-dep qemu
```

3. Get the source files for Qemu3.1.

```
mkdir ~/devel
cd ~/devel
wget https://download.qemu.org/qemu-3.1.0.tar.xz
tar -xf qemu-3.1.0.tar.xz
cd qemu-3.1.0
./configure --target-list=x86_64-softmmu --audio-driv-list=alsa,pa
```

4. Next we will have to compile Qemu, however we do not want to run make install when we are done as we do not want to replace the system installed qemu as this version we are compiling would get replaced by a system update.  
Instead we will copy the binary and bios files to /usr/local so we can keep both the system qemu and our own qemu3.0+

```
make
```

```
sudo cp x86_64-sofmmu/qemu-system-x86_64 /usr/local/bin/qemu3.1-system-x86_64
sudo cp -r pc-bios /usr/local/share/qemu
```

5. Verify your qemu3.1 is working by typing in.

```
qemu3.1-system-x86_64 --version
```

6. Now we need to add some new rules to apparmor to allow libvirt to access the binary.

```
sudo nano /etc/apparmor.d/abstractions/libvirt-qemu
```

7. Add the lines below to the bottom of the file and press CTRL+X to save it.

```
# Allow access to qemu3.1
/usr/local/bin/qemu3.1-system-x86_64 rmix,
/usr/local/share/qemu/** r,
```

8. Now we need to edit another apparmor file to allow libvirtd to execute qemu3.1

```
sudo nano /etc/apparmor.d/usr.sbin.libvirtd
```

9. Locate where the rules for /bin/ and /sbin/ executables are and add the line below into that section and save with CTRL+X.

```
# Access to execute custom qemu3.1 binary
/usr/local/bin/qemu3.1-system-x86_64 PUX,
```

10. Reload the apparmor rules.

```
sudo service apparmor reload
```

11. Finally we need to edit our virtual machine to use one of the qemu3.0 specific chipset and tell libvirt to use qemu3.1 for this specific virtual machine.

```
virsh edit yourvirtualmachinename
```

12. Locate the line that starts with “<type arch='x86\_64' machine=” it should be somewhere inside the <os> section. Once you find it you will need to change your chipset to the 3.0 version (this example assumes youre using the default and most common i440fx chipset, but it also works on the q35 chipset) machine='pc-i440fx-XXXXXX' into machine='pc-i440fx-3.0' so it would look like something like this.

```
<type arch='x86_64' machine='pc-i440fx-3.0'>hvm</type>
```

13. Now locate the start of the <devices> section and edit the <emulator> tag to point to our qemu3.1 binary.

The emulator tag should look like this when you are done.

```
<emulator>/usr/local/bin/qemu3.1-system-x86_64</emulator>
```

14. Press CTRL+X to save and we will be done and you should now be able to enjoy almost completely crackle-free audio!

## Game controllers and USB controllers

Several game controllers will not work with a normal USB passthrough, in these cases you will need to pass an entire USB controller, yikes.

There are a few solutions though.

- Pass through a PCI-e USB controller (no stubbing required! Just make sure it is in its own IOMMU group).
- Pass through a free USB controller from your motherboard if it has one (high end boards like the Asus Prime X399-A has 3 USB controllers, 2 of which are in their own IOMMU groups!).
- If you passed through an Nvidia card to your VM you can use the [Moonlight Streaming Client](#) to “bridge” your game controller to the VM, also you will be able to get sound from the moonlight client instead of spice.
- Steam In-Home streaming is an alternative but I have not had any reliable success with it when it comes to stable contact with the VM.

For option 1 and 2 you use the below 2 commands to figure out which groups the usb controllers are in then use the second command (but replace X with the group number that the usb controllers are in) to check if there are any other devices in that group, if the controllers are in their own group before passing them to the VM the same way you would with a graphics card after it has been stubbed.

```
ls-iommu | grep -i "usb controller"  
ls-iommu | grep -i "group X"
```

## Using Moonlight Streaming to “pass through” a game controller

Using Moonlight Streaming to pass through your game controller will add some overhead and CPU heavy games might get some stutter, further testing on more hardware is needed to verify. Also you will have to use the Moonlight Streaming window when you plan to use the game controller, you can probably make a script to keep focus on the Moonlight Streaming window but set LookingGlass to be “always on top” for better image quality.

Lastly, latency should be very low as you will be using the Host-Only isolated network with the paravirtualized VirtIO driver.

First we need to install flatpak, as the Moonlight Streaming client is only available as a flatpak, unless you compile it from source.

Head over to <https://flatpak.org/> and follow the install instructions.

Newer AMD GPUs that use the amdgpu kernel driver have no issues with OpenGL and flatpak, everything works out of the box!

Note: From my experience, Moonlight Streaming does not support the nouveau driver.

## Enabling OpenGL support in flatpak for Nvidia Proprietary driver

If you are using an Nvidia Graphics Card on your Linux Host, You will need to do some extra work to enable OpenGL for flatpak.

Linux Uprising has a nice guide that you can read [here](#), but a simplified version will be written here just in case the article vanishes in the future.

1. Find out the exact version of your Nvidia driver (if you are unsure, try both commands).

Using newer nvidia drivers from 2018+	Nvidia driver is older than 390 (2017 and earlier)
<code>apt-cache policy "nvidia-driver-*"</code>	<code>apt-cache policy "nvidia-3*"</code>

2. Note down the numbers before the “-” and then check if your driver is supported by flatpak and available.

```
flatpak remote-ls flathub | grep nvidia
```

3. Install the appropriate 32bit and 64bit runtime for your driver (replace MAJOR and MINOR with the numbers that you wrote down, 390.48 would be 390-48 in the commands).

```
flatpak install flathub org.freedesktop.Platform.GL.nvidia-MAJOR-MINOR
flatpak install flathub org.freedesktop.Platform.GL32.nvidia-MAJOR-MINOR
```



4. The example for installing the runtime for Nvidia driver 390.48 would then be

```
flatpak install flathub org.freedesktop.Platform.GL.nvidia-390-48  
flatpak install flathub org.freedesktop.Platform.GL32.nvidia-390-48
```

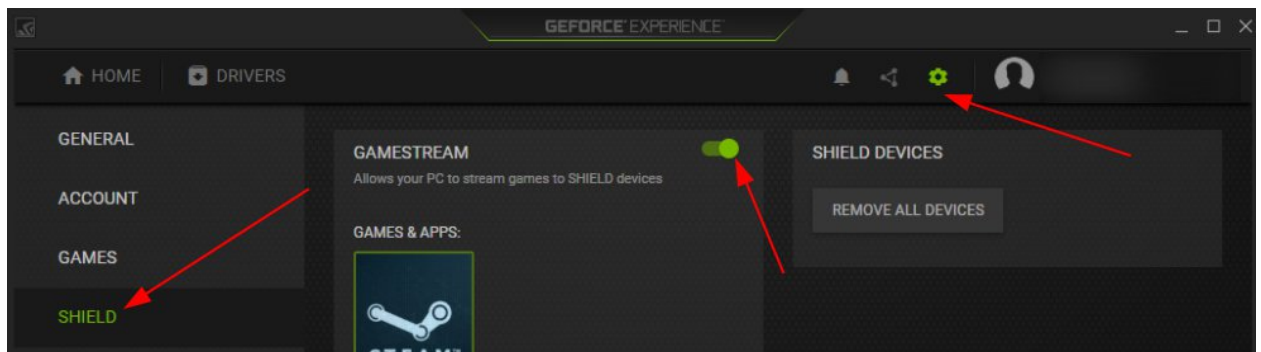
5. You are now done with installing flatpak with OpenGL support

### Installing & configuring the Moonlight Streaming Client

1. If you followed the flatpak install guide, you can install the Moonlight Streaming Client from here: [https://flathub.org/apps/details/com.moonlight\\_stream.Moonlight](https://flathub.org/apps/details/com.moonlight_stream.Moonlight) or by using the command below.

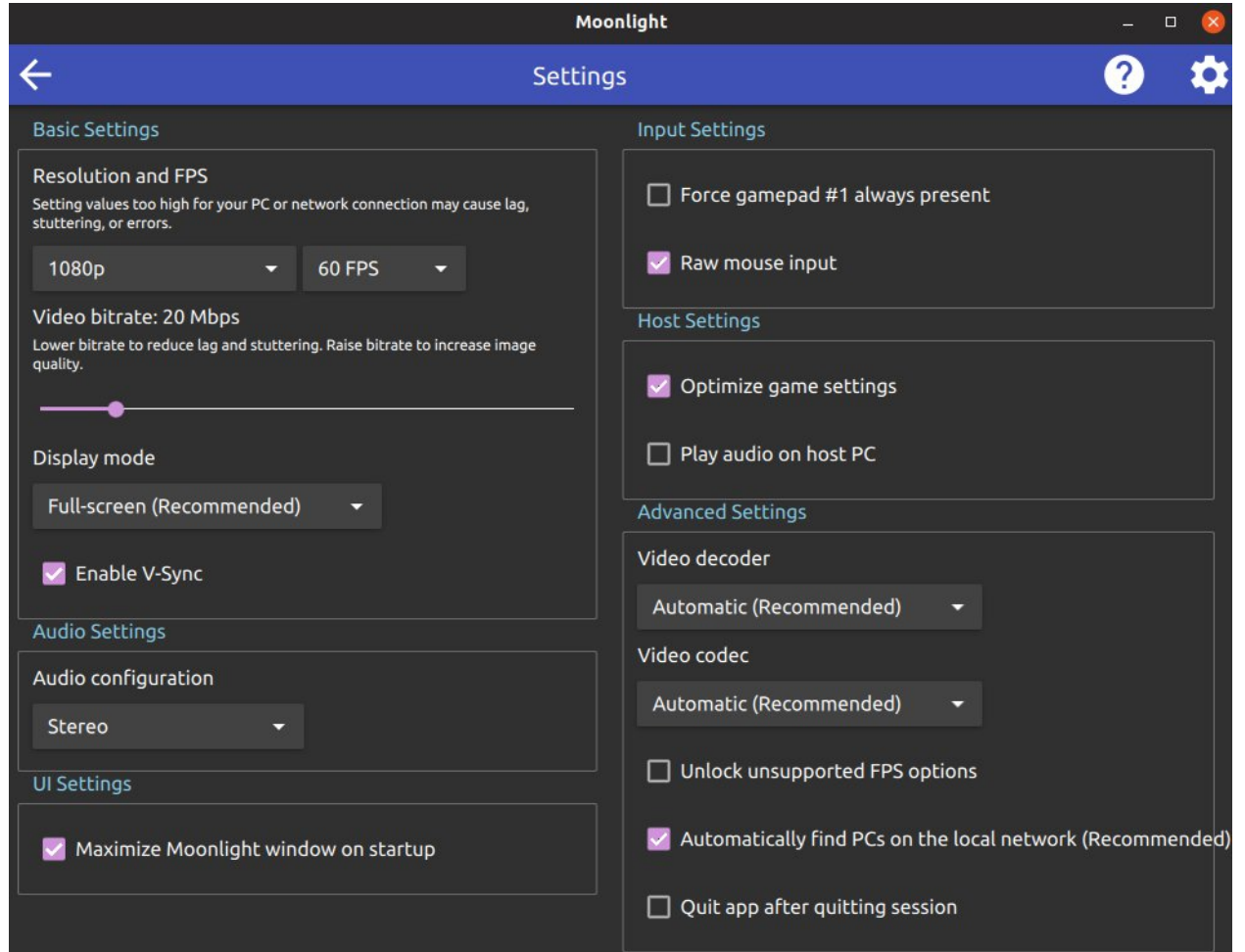
```
flatpak install flathub com.moonlight_stream.Moonlight
```

2. Open Geforce Experience on your virtual machine, click on the cogwheel and go to the Shield section and enable Gamestream.

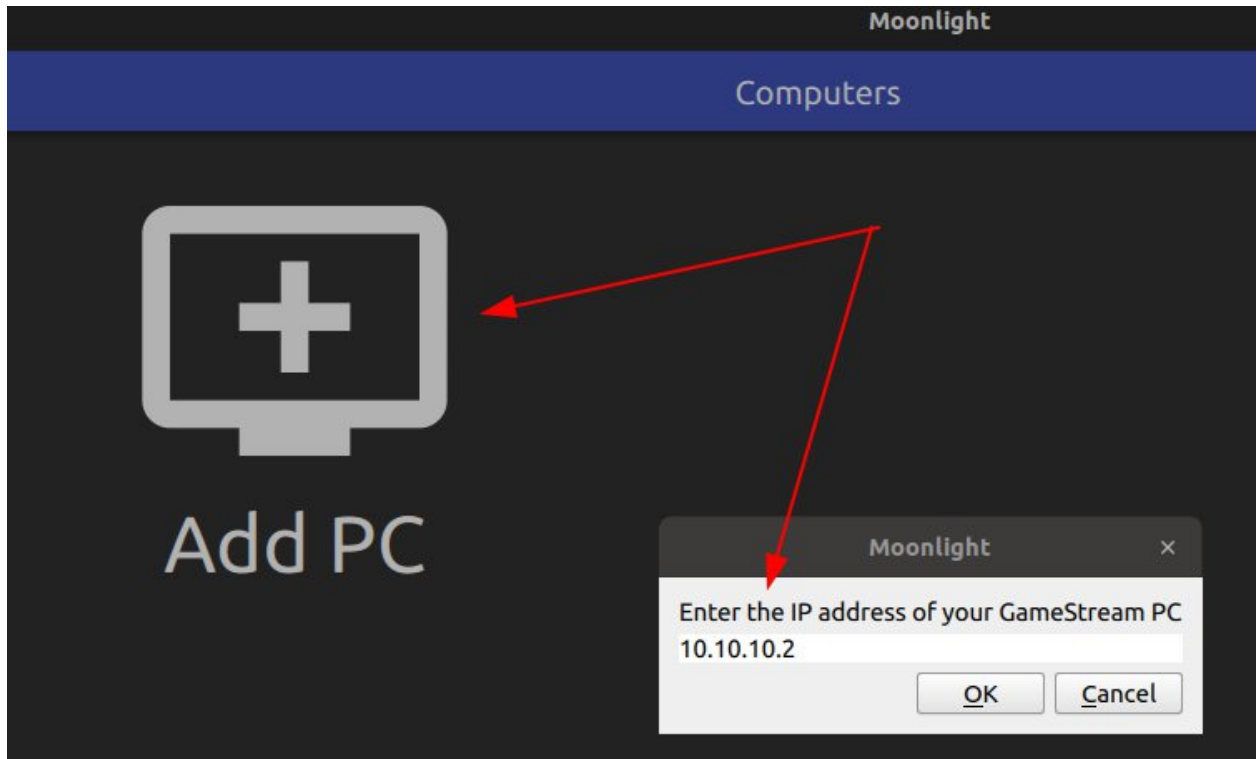


3. Open up Moonlight Streaming and click the cogwheel in the top right and change any settings you would like.

Note: You can play around with the video bitrate for better quality overall, I have not personally noticed any issues related to latency or stutter when using the Host-Only network over the VirtIO network device when I decided to max the bitrate for testing once.

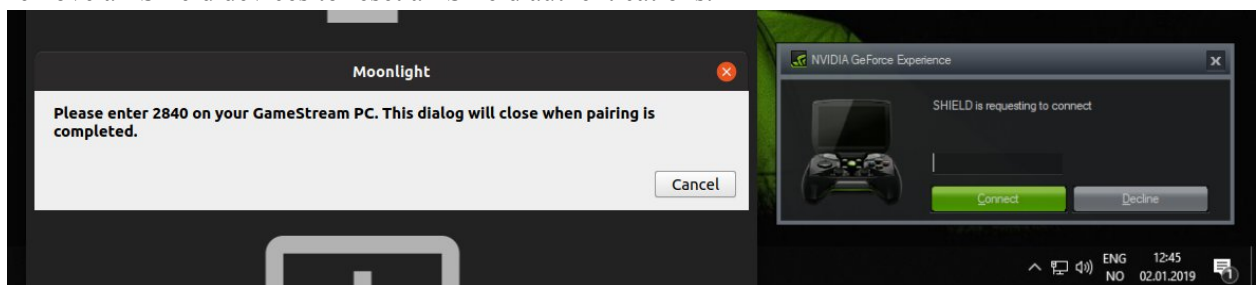


- Once done, click the back button and click Add PC and input the Host Only network IP address for the virtual machine.



- The machine should be detected and added to the list with a lock on it, once you click on it you will be asked to input a one time pin given to you by the Moonlight Streaming Client that you will have to type into your virtual machine.

Note: If the pin input notification does not show up for you, try go into GeForce Experience and remove all Shield devices to reset all Shield authentications.



- If successful the lock will be removed from the icon and you now just launch Steam or your games through Moonlight whenever you need Gamepad support!

Read about any hotkeys that Moonlight Streaming has here: <https://github.com/moonlight-stream/moonlight-docs/wiki/Setup-Guide#keyboardmousegamepad-input-options>

## Launch specific programs on specific cores in Windows

This is something you in most cases will never need but I will add it here in case you do.

Make a new text file and name it what you want but save it with the extension .ps1 and then paste in the below contents and change the filepath for what you want to launch, this example shows how to launch Guild Wars 2 and tell it to only use cores 3,4,5 and 6.

The ProcessorAffinity value is essentially the value of all the cores you want the program to run on, added together.

```
# Start the exe
$app = Start-Process -FilePath "C:\Games\Guild Wars 2\Gw2-64.exe" -ArgumentList "$args"

# Get the process ID
$proc = Get-Process Gw2-64
# Set the priority and affinity
$proc.ProcessorAffinity = 60
$proc.PriorityClass = "High"

# Uncomment the below line if you want the script to pause at the end
#cmd /c pause | out-null

# Core list
# core# = Value
# 1 = 1
# 2 = 2
# 3 = 4
# 4 = 8
# 5 = 16
# 6 = 32
# 7 = 64
# 8 = 128
```

You can then run the script through the command (or make a shortcut using it)

```
powershell -nologo -executionpolicy bypass -File "C:\path\to\ps1\file" arg1 arg2 etc
```

For a shortcut the process may have to start inside, this depends on how the script is launched.

```
C:\Windows\System32\WindowsPowerShell\v1.0\
```

# Performance tuning & exotics

This section is here for performance tweaking to get the most out of your virtual machine, however this is also where things can go wrong or because most systems will not need to do this.

You are expected to be familiar with how to work with grub and the root rescue shell in case anything goes wrong so you can repair your system in case you changed something you should not have changed.

You are also expected to know how a computer works and how the software works with the hardware.

In short, you need to know or understand what you are doing and you need to be capable of fixing your errors yourself if you do something wrong.

You have been warned.

## Pinning CPU cores to the VM for more performance

Pinning the cpus or setting CPU affinity helps in many cases to give stable performance in the virtual machine. Usually I pin 2-6 cores to the virtual machine depending on how many host CPU cores I have, if I have 4 cores I pin 2 cores, if I have 6 cores I pin 4 cores, etc.

I also only give the virtual machine access to the same amount of cores that I pin.

Note: This guide does not cover pinning CPU threads

1. Shut down the virtual machine and open the virtual machine config, as this cannot be done through the GUI

```
virsh edit yourvirtualmachinename
```

2. Locate the “<vcpu” tag, if you have chosen a simple topology (1 socket, x cores, 1 thread) it should be very simple to pin the CPU cores, the line should look something like this.

```
<vcpu placement='static'>2</vcpu>
```

3. What we want to do is add cpuset='0-1' after 'static', this will pin core 0 to 1 to the virtual machine. You can also pin separate cores by instead provide a comma separated list of cores, like lets say I want to pin core 0 and 2 then I would write cpuset='0,2' instead.

The line should look something like this now.

```
<vcpu placement='static' cpuset='0-1'>2</vcpu>
```

4. Now we need to fine tune our pinned cores by assigning specific cores to specific virtual cores. This can be done by adding this below the <vcpu line.

```
<cputune>  
  <vcpupin vcpu='0' cpuset='0'/>  
  <vcpupin vcpu='1' cpuset='1'/>  
  <emulatorpin cpuset='0-1'/>  
</cputune>
```

5. This tells vcpu 0 to only use CPU core 0 and vcpu 1 to only use CPU core 1, while the emulatorpin tag will pin the CPU cores we provided in the vcpu tag to the QEmu process from my understanding as the [documentation](#) does not cover it much.  
For more information about vcpu and thread pinning check the full [documentation](#).

## NUMA tuning for systems with multiple NUMA nodes

In most cases you do not need to delve into this, but if you have a Threadripper system or any other CPU with multiple NUMA nodes, this will be of great help to you to get even more performance out of the virtual environment!

Make sure your Memory Interleaving setting in the UEFI is set to NUMA mode instead of UMA mode, or set to “Channel” instead of “Die”.

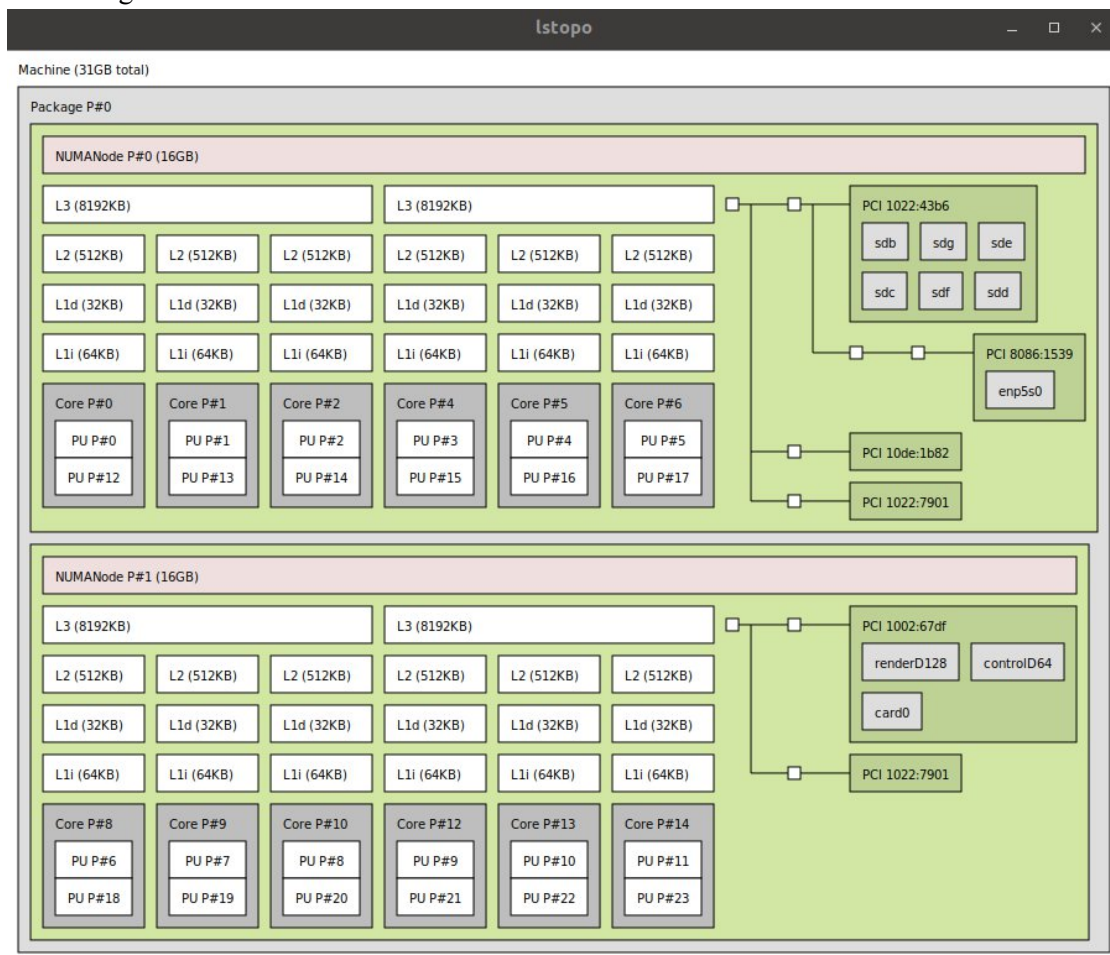
1. First we need to install hwloc so we can use lstopo, this will be extremely useful to understand your CPU topology.

```
sudo apt install hwloc
```

2. Next we launch lstopo

```
lstopo
```

3. You will get a window similar to this



4. If you look carefully, the graphics card I stubbed earlier is on NUMA node 0 (remember I stubbed the PCI-e ids **10de:1b82** and **10de:10f0**) so preferably I want to pin only cpus from

NUMA node 0 to my virtual machine (see previous section) and only allocate memory from NUMA node 0 to the virtual machine, so let us do exactly that!

5. Shut down your virtual machine and open the virtual machine config.

```
virsh edit yourvirtualmachinename
```

6. Write down the amount of memory you have given the machine (this is written in KiB in the memory tag almost at the very top of the file) as you will need this soon.
7. Now locate the “</cputune>” end tag and paste the contents of the box below it (set nodeset to 1 if the graphic card you gave the virtual machine is on NUMA node 1)

Note: If the graphic card you gave the virtual machine is located on NUMA node 1 then this is a good time to edit your CPU pins to only use CPU cores from NUMA node 1.

```
<numatune>  
<memory mode='strict' nodeset='0' />  
</numatune>
```

8. This will make the machine fail to start if it cannot allocate all the memory on the specified NUMA node, there are more mode policies you can use as specified by the [documentation](#). However strict will give you the most consistent performance as you avoid having to cross communicate between the NUMA nodes.
9. Now locate the “<cpu>” tag and add this inside it, edit the memory to match the amount of memory you have allocated to the virtual machine.

The below configuration uses CPU core 0 to 1 and 8GB memory located on NUMA node 0.

```
<numa>  
<cell id='0' cpus='0-1' memory='8388608' unit='KiB' />  
</numa>
```

10. The final configuration should look something like this if you have used a manual topology in the CPU configuration in the Virtual Machine Manager.

```
<cpu mode='custom' match='exact' check='partial'>  
<model fallback='allow'>EPYC-IBPB</model>  
<topology sockets='1' cores='2' threads='1' />  
<numa>  
<cell id='0' cpus='0-1' memory='8388608' unit='KiB' />  
</numa>  
</cpu>
```

11. Once you are done you save the file with CTRL+X.



## Isolate specific CPU cores for use only by the Virtual Machine

The Linux kernel will still be doing system interrupts on the isolated cores but the rest of the system will not touch them unless specifically told to do so, why would you do this?

To avoid the host and guest from competing for the cores. However I do not recommend doing this on any system with less than 6 cores!

1. Edit the grub defaults file.

```
sudo nano /etc/default/grub
```

2. Add the isolcpus parameter to GRUB\_CMDLINE\_LINUX\_DEFAULT, edit the parameter to contain the core numbers you do not want the system to touch.

The example below isolates CPU cores 0,1 and 3, you may have more parameters listed than this example.

```
GRUB_CMDLINE_LINUX_DEFAULT="quiet splash isolcpus=0,1,3"
```

3. Save the file with CTRL+X then run the update-grub command.

```
sudo update-grub
```

4. Reboot your system.

## Increase gaming performance by changing clocksource to tsc

This is a solution I have found to work on high core count systems.

If you rely on super accurate clock timings then you should not change this and just accept the stutter until another solution is found!

1. Edit the grub defaults file.

```
sudo nano /etc/default/grub
```

2. Add clock=tsc to the GRUB\_CMDLINE\_LINUX\_DEFAULT line. the line should look something like this and you may have more parameters than listed in this example.

```
GRUB_CMDLINE_LINUX_DEFAULT="quiet splash clock=tsc"
```

3. Save the file with CTRL+X and then update grub.

```
sudo update-grub
```

## Newer mainline kernels and kernel drivers on Ubuntu

The Level1Techs forum made me aware of this amazing tool to get easier access to mainline kernels for Ubuntu called ukuu, it lets you install and remove newer and older kernels at the click of a button, super handy if you use only hardware that is supported directly in the kernel!

If you are using an NVidia GPU as your host graphics and the proprietary driver, do not use ukuu as you will be at the mercy of NVidia about which kernels you can use, and those are usually the ones released through traditional Ubuntu updates.

But if you use an AMD GPU, using a newer kernel lets you get easy access to newer amdgpu drivers and other fixes for things you might have encountered, if any.

You can install ukuu with the commands below

```
sudo apt-add-repository -y ppa:teejee2008/ppa
sudo apt-get update
sudo apt-get install ukuu
```

When you open ukuu it should look like this once it has checked for any new kernels.  
Kernels with the Ubuntu icon are kernels scheduled or already released for Ubuntu.

