PCI(e) Passthrough

PCI(e) passthrough is a mechanism to give a virtual machine control over a PCI device from the host. This can have some advantages over using virtualized hardware, for example lower latency, higher performance, or more features (e.g., offloading).

But, if you pass through a device to a virtual machine, you cannot use that device anymore on the host or in any other VM.

Note that, while PCI passthrough is available for i440fx and q35 machines, PCIe passthrough is only available on q35 machines. This does not mean that PCIe capable devices that are passed through as PCI devices will only run at PCI speeds. Passing through devices as PCIe just sets a flag for the guest to tell it that the device is a PCIe device instead of a "really fast legacy PCI device". Some guest applications benefit from this.

General Requirements

Since passthrough is performed on real hardware, it needs to fulfill some requirements. A brief overview of these requirements is given below, for more information on specific devices, see PCI Passthrough Examples.

Hardware

Your hardware needs to support `IOMMU` (**I**/**O** **M**emory **M**anagement **U**nit) interrupt remapping, this includes the CPU and the motherboard.

Generally, Intel systems with VT-d and AMD systems with AMD-Vi support this. But it is not guaranteed that everything will work out of the box, due to bad hardware implementation and missing or low quality drivers.

Further, server grade hardware has often better support than consumer grade hardware, but even then, many modern system can support this.

Please refer to your hardware vendor to check if they support this feature under Linux for your specific setup.

Determining PCI Card Address

The easiest way is to use the GUI to add a device of type "Host PCI" in the VM's hardware tab. Alternatively, you can use the command line.

You can locate your card using

```
lspci
```

Once you ensured that your hardware supports passthrough, you will need to do some configuration to enable PCI(e) passthrough.

IOMMU

You will have to enable IOMMU support in your BIOS/UEFI. Usually the corresponding setting is called `IOMMU` or `VT-d`, but you should find the exact option name in the manual of your motherboard.

With AMD CPUs IOMMU is enabled by default. With recent kernels (6.8 or newer), this is also true for Intel CPUs. On older kernels, it is necessary to enable it on Intel CPUs via the kernel command line by adding:

```
intel_iommu=on
```

IOMMU Passthrough Mode

If your hardware supports IOMMU passthrough mode, enabling this mode might increase performance. This is because VMs then bypass the (default) DMA translation normally performed by the hyper-visor and instead pass DMA requests directly to the hardware IOMMU. To enable these options, add:

```
iommu=pt
```

to the kernel commandline.

Kernel Modules

You have to make sure the following modules are loaded. This can be achieved by adding them to '*/etc/modules*'.

Mediated devices passthrough

If passing through mediated devices (e.g. vGPUs), the following is not needed. In these cases, the device will be owned by the appropriate host-driver directly.

```
vfio
vfio_iommu_type1
vfio_pci
```

After changing anything modules related, you need to refresh your `initramfs`. On Proxmox VE this can be done by executing:

```
# update-initramfs -u -k all
```

To check if the modules are being loaded, the output of

```
# lsmod | grep vfio
```

should include the four modules from above.

Finish Configuration

Finally reboot to bring the changes into effect and check that it is indeed enabled.

```
# dmesg | grep -e DMAR -e IOMMU -e AMD-Vi
```

should display that `IOMMU`, `Directed I/O` or `Interrupt Remapping` is enabled, depending on hardware and kernel the exact message can vary.

For notes on how to troubleshoot or verify if IOMMU is working as intended, please see the Verifying IOMMU Parameters section in our wiki.

It is also important that the device(s) you want to pass through are in a **separate** `IOMMU` group. This can be checked with a call to the Proxmox VE API:

```
# pvesh get /nodes/{nodename}/hardware/pci --pci-class-blacklist ""
```

It is okay if the device is in an `IOMMU` group together with its functions (e.g. a GPU with the HDMI Audio device) or with its root port or PCI(e) bridge.

PCI(e) slots

Some platforms handle their physical PCI(e) slots differently. So, sometimes it can help to put the card in a another PCI(e) slot, if you do not get the desired `IOMMU` group separation.

Unsafe interrupts

For some platforms, it may be necessary to allow unsafe interrupts. For this add the following line in a file ending with ‘.conf’ file in **/etc/modprobe.d/**:

```
options vfio_iommu_type1 allow_unsafe_interrupts=1
```

Please be aware that this option can make your system unstable.

GPU Passthrough Notes

It is not possible to display the frame buffer of the GPU via NoVNC or SPICE on the Proxmox VE web interface.

When passing through a whole GPU or a vGPU and graphic output is wanted, one has to either physically connect a monitor to the card, or configure a remote desktop software (for example, VNC or RDP) inside the guest.

If you want to use the GPU as a hardware accelerator, for example, for programs using OpenCL or CUDA, this [illegible]

Host Device Passthrough

The most used variant of PCI(e) passthrough is to pass through a whole PCI(e) card, for example a GPU or a network card.

Host Configuration

Proxmox VE tries to automatically make the PCI(e) device unavailable for the host. However, if this doesn't work, there are two things that can be done:

- pass the device IDs to the options of the *vfio-pci* modules by adding

  ```
   options vfio-pci ids=1234:5678,4321:8765
  ```

  to a .conf file in **/etc/modprobe.d/** where `1234:5678` and `4321:8765` are the vendor and device IDs obtained by:

  ```
  # lspci -nn
  ```

- blacklist the driver on the host completely, ensuring that it is free to bind for passthrough, with

  ```
   blacklist DRIVERNAME
  ```

  in a .conf file in **/etc/modprobe.d/**.

  To find the drivername, execute

  ```
  # lspci -k
  ```

  for example:

  ```
  # lspci -k | grep -A 3 "VGA"
  ```

  will output something similar to

  ```
  01:00.0 VGA compatible controller: NVIDIA Corporation GP108 [GeForce GT 1030] (rev a1)
          Subsystem: Micro-Star International Co., Ltd. [MSI] GP108 [GeForce GT 1030]
          Kernel driver in use: <some-module>
          Kernel modules: <some-module>
  ```

  Now we can blacklist the drivers by writing them into a .conf file:

  ```
  echo "blacklist <some-module>" >> /etc/modprobe.d/blacklist.conf
  ```

For both methods you need to update the `initramfs` again and reboot after that.

Should this not work, you might need to set a soft dependency to load the gpu modules before loading *vfio-pci*. This can be done with the *softdep* flag, see also the manpages on *modprobe.d* for more information.

For example, if you are using drivers named <some-module>:

```
# echo "softdep <some-module> pre: vfio-pci" >> /etc/modprobe.d/<some-module>.conf
```

Verify Configuration

To check if your changes were successful, you can use

```
# lspci -nnk
```

and check your device entry. If it says

```
Kernel driver in use: vfio-pci
```

or the *in use* line is missing entirely, the device is ready to be used for passthrough.

Mediated devices

For mediated devices this line will differ as the device will be owned as the host driver directly, not *vfio-pci*.

VM Configuration

When passing through a GPU, the best compatibility is reached when using *q35* as machine type, *OVMF* (*UEFI* for VMs) instead of SeaBIOS and PCIe instead of PCI. Note that if you want to use *OVMF* for GPU passthrough, the GPU needs to have an UEFI capable ROM, otherwise use SeaBIOS instead. To check if the ROM is UEFI capable, see the PCI Passthrough Examples wiki.

Furthermore, using OVMF, disabling vga arbitration may be possible, reducing the amount of legacy code needed to be run during boot. To disable vga arbitration:

```
 echo "options vfio-pci ids=<vendor-id>,<device-id> disable_vga=1" > /etc/modprobe.d/vfio.conf
```

replacing the <vendor-id> and <device-id> with the ones obtained from:

```
# lspci -nn
```

PCI devices can be added in the web interface in the hardware section of the VM. Alternatively, you can use the command line; set the **hostpciX** option in the VM configuration, for example by executing:

```
# qm set VMID -hostpci0 00:02.0
```

or by adding a line to the VM configuration file:

```
 hostpci0: 00:02.0
```

If your device has multiple functions (e.g., ‘`00:02.0`’ and ‘`00:02.1`’), you can pass them through all together with the shortened syntax ``00:02`. *This is equivalent with checking the ``All Functions`* checkbox in the web interface.

There are some options to which may be necessary, depending on the device and guest OS:

- **x-vga=on|off** marks the PCI(e) device as the primary GPU of the VM. With this enabled the **vga** configuration option will be ignored.
- **pcie=on|off** tells Proxmox VE to use a PCIe or PCI port. Some guests/device combination require PCIe rather than PCI. PCIe is only available for *q35* machine types.
- **rombar=on|off** makes the firmware ROM visible for the guest. Default is on. Some PCI(e) devices need this disabled.
- **romfile=<path>**, is an optional path to a ROM file for the device to use. This is a relative path under **/usr/share/kvm/**.

Example

An example of PCIe passthrough with a GPU set to primary:

```
# qm set VMID -hostpci0 02:00,pcie=on,x-vga=on
```

PCI ID overrides

You can override the PCI vendor ID, device ID, and subsystem IDs that will be seen by the guest. This is useful if your device is a variant with an ID that your guest’s drivers don’t recognize, but you want to force those drivers to be loaded anyway (e.g. if you know your device shares the same chipset as a supported variant).

The available options are `vendor-id`, `device-id`, `sub-vendor-id`, and `sub-device-id`. You can set any or all of these to override your device’s default IDs.

For example:

```
# qm set VMID -hostpci0 02:00,device-id=0x10f6,sub-vendor-id=0x0000
```

SR-IOV

Another variant for passing through PCI(e) devices is to use the hardware virtualization features of your devices, if available.

Enabling SR-IOV

To use SR-IOV, platform support is especially important. It may be necessary to enable this feature in the BIOS/UEFI first, or to use a specific PCI(e) port for it to work. In doubt, consult the manual of the platform or contact its vendor.

SR-IOV (**S**ingle-**R**oot **I**nput/**O**utput **V**irtualization) enables a single device to provide multiple *VF* (**V**irtual **F**unctions) to the system. Each of those *VF* can be used in a different VM, with full hardware features and also better performance and lower latency than software virtualized devices.

Currently, the most common use case for this are NICs (**N**etwork **I**nterface **C**ard) with SR-IOV support, which can provide multiple VFs per physical port. This allows using features such as checksum offloading, etc. to be used inside a VM, reducing the (host) CPU overhead.

Host Configuration

Generally, there are two methods for enabling virtual functions on a device.

- sometimes there is an option for the driver module e.g. for some Intel drivers

  ```
  max_vfs=4
  ```

  which could be put file with *.conf* ending under **/etc/modprobe.d/**. (Do not forget to update your initramfs after that)

  Please refer to your driver module documentation for the exact parameters and options.

- The second, more generic, approach is using the `sysfs`. If a device and driver supports this you can change the number of VFs on the fly. For example, to setup 4 VFs on device 0000:01:00.0 execute:

  ```
  # echo 4 > /sys/bus/pci/devices/0000:01:00.0/sriov_numvfs
  ```

  To make this change persistent you can use the ‘sysfsutils` Debian package. After installation configure it via **/etc/sysfs.conf** or a `FILE.conf’ in **/etc/sysfs.d/**.

VM Configuration

After creating VFs, you should see them as separate PCI(e) devices when outputting them with `lspci`. Get their ID and pass them through like a normal PCI(e) device.

Mediated Devices (vGPU, GVT-g)

Mediated devices are another method to reuse features and performance from physical hardware for virtualized hardware. These are found most common in virtualized GPU setups such as Intel's GVT-g and NVIDIA's vGPUs used in their GRID technology.

With this, a physical Card is able to create virtual cards, similar to SR-IOV. The difference is that mediated devices do not appear as PCI(e) devices in the host, and are such only suited for using in virtual machines.

Host Configuration

In general your card's driver must support that feature, otherwise it will not work. So please refer to your vendor for compatible drivers and how to configure them.

Intel's drivers for GVT-g are integrated in the Kernel and should work with 5th, 6th and 7th generation Intel Core Processors, as well as E3 v4, E3 v5 and E3 v6 Xeon Processors.

To enable it for Intel Graphics, you have to make sure to load the module *kvmgt* (for example via `/etc/modules`) and to enable it on the Kernel commandline and add the following parameter:

```
 i915.enable_gvt=1
```

After that remember to update the `initramfs`, and reboot your host.

VM Configuration

To use a mediated device, simply specify the `mdev` property on a `hostpciX` VM configuration option.

You can get the supported devices via the *sysfs*. For example, to list the supported types for the device *0000:00:02.0* you would simply execute:

```
# ls /sys/bus/pci/devices/0000:00:02.0/mdev_supported_types
```

Each entry is a directory which contains the following important files:

- *available_instances* contains the amount of still available instances of this type, each *mdev* use in a VM reduces this.

- *description* contains a short description about the capabilities of the type

- *create* is the endpoint to create such a device, Proxmox VE does this automatically for you, if a *hostpci*X option with mdev is configured.

Example configuration with an `Intel GVT-g vGPU` (`Intel Skylake 6700k`):

```
# qm set VMID -hostpci0 00:02.0,mdev=i915-GVTg_V5_4
```

With this set, Proxmox VE automatically creates such a device on VM start, and cleans it up again when the VM stops.

Use in Clusters

It is also possible to map devices on a cluster level, so that they can be properly used with HA and hardware changes are detected and non root users can configure them. See Resource Mapping for details on that.

vIOMMU (emulated IOMMU)

vIOMMU is the emulation of a hardware IOMMU within a virtual machine, providing improved memory access control and security for virtualized I/O devices. Using the vIOMMU option also allows you to pass through PCI(e) devices to level-2 VMs in level-1 VMs via Nested Virtualization. To pass through physical PCI(e) devices from the host to nested VMs, follow the PCI(e) passthrough instructions.

There are currently two vIOMMU implementations available: Intel and VirtIO.

Intel vIOMMU

Intel vIOMMU specific VM requirements:

- Whether you are using an Intel or AMD CPU on your host, it is important to set `intel_iommu=on` in the VMs kernel parameters.

- To use Intel vIOMMU you need to set **q35** as the machine type.

If all requirements are met, you can add `viommu=intel` to the machine parameter in the configuration of the VM that should be able to pass through PCI devices.

```
# qm set VMID -machine q35,viommu=intel
```

QEMU documentation for VT-d

VirtIO vIOMMU

This vIOMMU implementation is more recent and does not have as many limitations as Intel vIOMMU but is currently less used in production and less documentated.

With VirtIO vIOMMU there is **no** need to set any kernel parameters. It is also **not** necessary to use q35 as the machine type, but it is advisable if you want to use PCIe.

```
# qm set VMID -machine q35,viommu=virtio
```

Blog-Post by Michael Zhao explaining virtio-iommu

See Also

- PCI Passthrough Examples