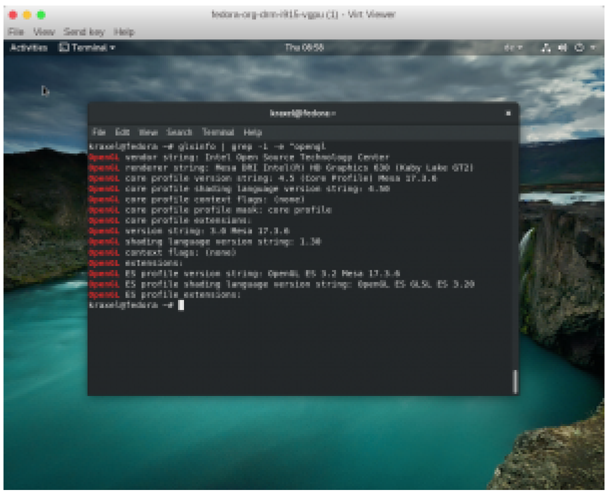
kraxel's news

TAG ARCHIVES: VGPU

vgpu display support finally merged upstream


fedora-org-drm-i915-vgpu (1) - Virt Viewer
File View Send key Help
Activities Terminal
Thu 08:58
kraxel@fedora:~
File Edit View Search Terminal Help

```
kraxel@fedora ~# glxinfo | grep -i -e "opengl
OpenGL vendor string: Intel Open Source Technology Center
OpenGL renderer string: Mesa DRI Intel(R) HD Graphics 630 (Kaby Lake GT2)
OpenGL core profile version string: 4.5 (Core Profile) Mesa 17.3.6
OpenGL core profile shading language version string: 4.50
OpenGL core profile context flags: (none)
OpenGL core profile profile mask: core profile
OpenGL core profile extensions:
OpenGL version string: 3.0 Mesa 17.3.6
OpenGL shading language version string: 1.30
OpenGL context flags: (none)
OpenGL extensions:
OpenGL ES profile version string: OpenGL ES 3.2 Mesa 17.3.6
OpenGL ES profile shading language version string: OpenGL ES GLSL ES 3.20
OpenGL ES profile extensions:
kraxel@fedora ~#
```

It took more than a year from the first working patches to the upstream merge. But now it's finally done. The linux kernel 4.16 (released on easter weekend) has the kernel-side code needed. The qemu code has been merged too (for gtk and spice user interfaces) and will be in the upcoming 2.12 release which is in code freeze right now. The 2.12 release candidates already have the code, so you can grab one if you don't want wait for the final release to play with this.

The vgpu code in the intel driver is off by default and must be enabled via module option. And, while being at it, also suggest to load the kvmgt module. So I've dropped a config file with these lines …

```
options i915 enable_gvt=1
softdep i915 pre: kvmgt
```

… into `/etc/modprobe.d/`. For some reason dracut didn't pick the changes up even after regenerating the initrd. Because of that I've blacklisted the intel driver (`rd.driver.blacklist=i915` on the kernel command line) so the driver gets loaded later, after mounting the root filesystem, and modprobe actually sets the parameter.

With that in place you should have a `/sys/class/mdev_bus` directory with the intel gpu in there. You can create vgpu devices now. Check the mediated device documentation for details.

One final thing to take care of: Currently using gvt mlocks all guest memory. For that work the mlock limit (`ulimit -l`) must be high enough, otherwise the vgpu will not work correctly and you'll see a scrambled display. Limit can be configured in `/etc/security/limits.conf`.

Now lets use our new vgpu with qemu:

```
qemu-system-x86_64 \
     -enable-kvm \
     -m 1G \
     -nodefaults \
     -M graphics=off \
     -serial stdio \
     -display gtk,gl=on \
     -device vfio-pci,sysfsdev=/sys/bus/mdev/devices/UUID,display=on \
     -cdrom /vmdisk/iso/Fedora-Workstation-Live-x86_64-27-1.6.iso
```

Details on the non-obvious qemu switches:

`-nodefaults`
Do not create default devices (such as vga and nic).

`-M graphics=off`
Hint for the firmware that the guest runs without a graphical display. This enables serial console support in seabios. We use this here because the vgpu has no firmware support (i.e. no vgabios), therefore nothing is visible on the display until the i915 kernel module loads.

`-display gtk,gl=on`
Use gtk display, enable opengl.

`-device vfio-pci,sysfsdev=/sys/bus/mdev/devices/UUID,display=on`
Add the vgpu to the guest, enable the display. Of course you have to replace *UUID* with your device.

libvirt support is still being worked on. Most bits are there, but some little details are missing. For example there is no way (yet) to tell libvirt the guest doesn't need an emulated vga device, so you'll end up with two spice windows, one for the emulated vga and one for the vgpu. Other than that things are working pretty straight forward. You need spice with opengl support enabled:

```
<graphics type='spice'>
  <listen type='none'/>
  <gl enable='yes'/>
</graphics>
```

And the vgpu must be added of course:

```
<hostdev mode='subsystem' type='mdev' managed='no' model='vfio-pci'>
  <source>
    <address uuid='UUID'/>
  </source>
  <address type='pci' domain='0x0000' bus='0x00' slot='0x05' function='0x0'/>
</hostdev>
```

Then you can start the domain. Use "virt-viewer –attach *guest*" to connect to the guest. Note that guest and virt-viewer must run on the same machine, sending the vgpu display to a remote machine does not yet work.

This entry was posted in Tech Tips and tagged qemu, vgpu on April 9, 2018 [https://www.kraxel.org/blog/2018/04/vgpu-display-support-finally-merged-upstream/] .

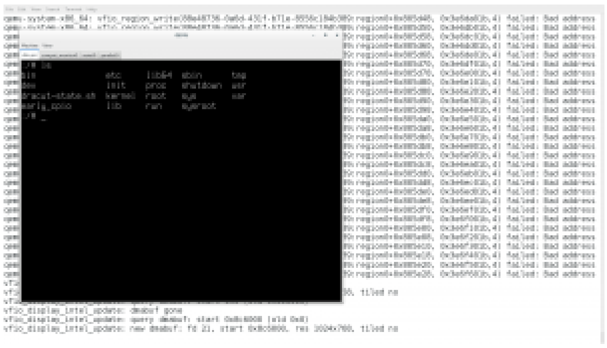
local vgpu display – status update

After a looong time things are finally landing upstream. Both vfio API update for vgpu local display support and support for that in the intel drm driver have been merged during the 4.16 merge window. Should not take too long to get the qemu changes merged upstream now, if all goes well the next release (2.12) will have it.

Tina Zhang wrote a nice blog article with both technical background and instructions for those who want play with it.

This entry was posted in Tech Tips and tagged qemu, vgpu on February 13, 2018 [https://www.kraxel.org/blog/2018/02/local-vgpu-display-status-update/] .

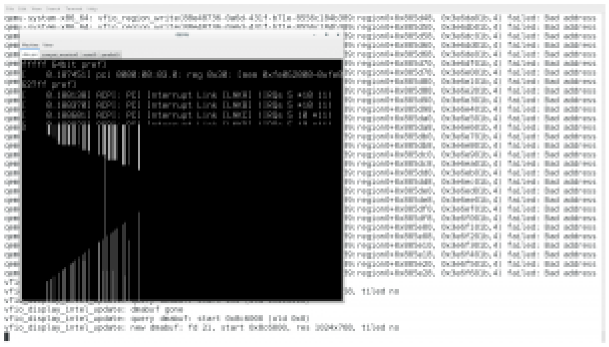
local display for intel vgpu starts working



```
/# ls
bin               etc     lib64  sbin      tmp
dev               init    proc   shutdown  usr
dracut-state.sh   kernel  root   sys       var
early_cpio        lib     run    sysroot
/# _
```

```
vfio_display_intel_update: dmabuf gone
vfio_display_intel_update: query dmabuf: start 0x[illegible] ([illegible])
vfio_display_intel_update: new dmabuf: fd 21, start 0x[illegible], res 1024x768, tiled no
```

Intel vgpu guest display shows up in the qemu gtk window. This is a linux kernel booted to the dracut emergency shell prompt, with the framebuffer console running @ inteldrmfb. Booting a full linux guest with X11 and/or wayland not tested yet. There are rendering glitches too, running "`dmesg`" looks like this:



So, a bunch of issues to solve before this is ready for users, but it's a nice start.

For the brave:

host kernel: https://www.kraxel.org/cgit/linux/log/?h=intel-vgpu
qemu: https://www.kraxel.org/cgit/qemu/log/?h=work/intel-vgpu

Take care, both branches are moving targets (aka: rebasing at times).

This entry was posted in Uncategorized and tagged qemu, vgpu on January 24, 2017 [https://www.kraxel.org/blog/2017/01/local-display-for-intel-vgpu-starts-working/] .

virtual gpu support landing upstream

The upstreaming process of virtual gpu support (vgpu) made a big step forward with the 4.10 merge window. Two important pieces have been merged:

First, the mediated device framework (mdev). Basically this allows kernel drivers to present virtual pci devices, using the vfio framework and interfaces. Both nvidia and intel will use mdev to partition the physical gpu of the host into multiple virtual devices which then can be assigned to virtual machines.

Second, intel landed initial mdev support for the i915 driver too. There is quite some work left to do in future kernel releases though. Accessing to the guest display is not supported yet, so you must run x11vnc or simliar tools in the guest to see the screen. Also there are some stability issues to find and fix.

If you want play with this nevertheless, here is how to do it. But be prepared for crashes and better don't try this on a production machine.

On the host: create virtual devices

On the host machine you obviously need a 4.10 kernel. Also the intel graphics device (igd) must be broadwell or newer. In the kernel configuration enable vfio and mdev (all `CONFIG_VFIO_*` options). Enable

`CONFIG_DRM_I915_GVT` and `CONFIG_DRM_I915_GVT_KVMGT` for intel vgpu support. Building the mtty sample driver (`CONFIG_SAMPLE_VFIO_MDEV_MTTY`, a virtual serial port) can be useful too, for testing.

Boot the new kernel. Load all modules: `vfio-pci`, `vfio-mdev`, optionally `mtty`. Also `i915` and `kvmgt` of course, but that probably happened during boot already.

Go to the `/sys/class/mdev_bus` directory. This should look like this:

```
kraxel@broadwell ~# cd /sys/class/mdev_bus
kraxel@broadwell .../class/mdev_bus# ls -l
total 0
lrwxrwxrwx. 1 root root 0 17. Jan 10:51 0000:00:02.0 -> ../../devices/pci0000:00/0000:00:
lrwxrwxrwx. 1 root root 0 17. Jan 11:57 mtty -> ../../devices/virtual/mtty/mtty
```

Each driver with mdev support has a directory there. Go to *$device*`/mdev_supported_types` to check what kind of virtual devices you can create.

```
kraxel@broadwell .../class/mdev_bus# cd 0000:00:02.0/mdev_supported_types
kraxel@broadwell .../0000:00:02.0/mdev_supported_types# ls -l
total 0
drwxr-xr-x. 3 root root 0 17. Jan 11:59 i915-GVTg_V4_1
drwxr-xr-x. 3 root root 0 17. Jan 11:57 i915-GVTg_V4_2
drwxr-xr-x. 3 root root 0 17. Jan 11:59 i915-GVTg_V4_4
```

As you can see intel supports three different configurations on my machine. The configuration (basically the amount of video memory) differs, and the number of instances you can create. Check the `description` and `available_instance` files in the directories:

```
kraxel@broadwell .../0000:00:02.0/mdev_supported_types# cd i915-GVTg_V4_2
kraxel@broadwell .../mdev_supported_types/i915-GVTg_V4_2# cat description
low_gm_size: 64MB
high_gm_size: 192MB
fence: 4
kraxel@broadwell .../mdev_supported_types/i915-GVTg_V4_2# cat available_instance
2
```

Now it is possible to create virtual devices by writing a UUID into the `create` file:

```
kraxel@broadwell .../mdev_supported_types/i915-GVTg_V4_2# uuid=$(uuidgen)
kraxel@broadwell .../mdev_supported_types/i915-GVTg_V4_2# echo $uuid
f321853c-c584-4a6b-b99a-3eee22a3919c
kraxel@broadwell .../mdev_supported_types/i915-GVTg_V4_2# sudo sh -c "echo $uuid > create
```

The new vgpu device will show up as subdirectory of the host gpu:

```
kraxel@broadwell .../mdev_supported_types/i915-GVTg_V4_2# cd ../../$uuid
kraxel@broadwell .../0000:00:02.0/f321853c-c584-4a6b-b99a-3eee22a3919c# ls -l
total 0
lrwxrwxrwx. 1 root root    0 17. Jan 12:31 driver -> ../../../../bus/mdev/drivers/vfio_md
lrwxrwxrwx. 1 root root    0 17. Jan 12:35 iommu_group -> ../../../../kernel/iommu_groups
lrwxrwxrwx. 1 root root    0 17. Jan 12:35 mdev_type -> ../mdev_supported_types/i915-GVTg
drwxr-xr-x. 2 root root    0 17. Jan 12:35 power
--w-------. 1 root root 4096 17. Jan 12:35 remove
lrwxrwxrwx. 1 root root    0 17. Jan 12:31 subsystem -> ../../../../bus/mdev
-rw-r--r--. 1 root root 4096 17. Jan 12:35 uevent
```

You can see the device landed in iommu group 10. We'll need that in a moment.

On the host: configure guests

Ideally this would be as simple as adding `<hostdev>` to your guests libvirt xml config. The mdev devices don't have a pci address on the host though, and because of that they must be passed to qemu using the sysfs device path instead of the pci address. libvirt doesn't (yet) support sysfs paths though, so it is a bit more complicated for now. Alot of the setup libvirt does for hostdevs automatically must be done manually instead.

First, we must allow qemu access /dev. By default libvirt uses control groups to restrict access. That must be turned off. Edit `/etc/libvirt/qemu.conf`. Uncomment the `cgroup_controllers` line. Remove `"devices"` from the list. Restart libvirtd.

Second, we must allow qemu access the iommu group (10 in my case). A simple chmod will do:

```
kraxel@broadwell ~# chmod 666 /dev/vfio/10
```

Third, we must update the guest configuration:

```
<domain type='kvm' xmlns:qemu='http://libvirt.org/schemas/domain/qemu/1.0'>
  [ ... ]
  <currentMemory unit='KiB'>1048576</currentMemory>
  <memoryBacking>
    <locked/>
  </memoryBacking>
  [ ... ]
  <qemu:commandline>
    <qemu:arg value='-device'/>
    <qemu:arg value='vfio-pci,addr=05.0,sysfsdev=/sys/class/mdev_bus/0000:00:02.0/f3
  </qemu:commandline>
</domain>
```

There is special qemu namespace which can be used to pass extra command line arguments to qemu. We do this here to use a qemu feature not yet supported by libvirt (use sysfs paths for vfio-pci). Also we must explicitly allow to lock down guest memory.

Now we are ready to go:

```
kraxel@broadwell ~# virsh start --console $guest
```

In the guest

It is a good idea to prepare the guest a bit before adding the vgpu to the guest configuration. Setup a serial console, so you can talk to it even in case graphics are broken. Blacklist the `i915` module and load it manually, at least until you have a known-working configuration. Also booting to runlevel 3 (aka multi-user.target) instead of 5 (aka graphical.target) and starting the xorg server manually is better for now.

For the guest machine intel recommends the 4.8 kernel. In theory newer kernels should work too, in practice they didn't last time I tested (4.10-rc2). Also make sure the xorg server uses the modesetting driver, the intel driver didn't work in my testing. This config file will do:

```
root@guest ~# cat /etc/X11/xorg.conf.d/intel.conf
Section "Device"
        Identifier  "Card0"
#       Driver      "intel"
        Driver      "modesetting"
        BusID       "PCI:0:5:0"
EndSection
```

I'm starting the xorg server with x11vnc, xterm and mwm (motif window manager) using this little script:

```
#!/bin/sh

# debug
echo "# $0: DISPLAY=$DISPLAY"

# start server
if test "$DISPLAY" != ":4"; then
        echo "# $0: starting Xorg server"
        exec startx $0 -- /usr/bin/Xorg :4
        exit 1
fi
echo "# $0: starting session"

# configure session
xrdb $HOME/.Xdefaults

# start clients
x11vnc -rfbport 5904 &
xterm &
exec mwm
```

The session runs on display 4, so you should be able to connect from the host this way:

```
kraxel@broadwell ~# vncviewer $guest_ip:4
```

Have fun!

This entry was posted in Tech Tips and tagged qemu, vgpu on January 17, 2017 [https://www.kraxel.org/blog/2017/01/virtual-gpu-support-landing-upstream/] .