James' Personal Site

Menu

GPU Virtualization with Hyper-V

August 25, 2020 by james

For a short guide on how to enable GPU-PV for Hyper-V, click here.

Nowadays, Hyper-V has evolved from a server virtualization software to something very integral to contemporary Windows operating systems. The ability to run Linux applications (WSL 2), Virtualization-based Security, and application containerization (the WDAG and Windows Sandbox) are powered Hyper-V. Still, Hyper-V is one of the best virtualization platform available in Windows platform, if your guest is either a modern Linux or modern Windows. Being a Type-1 Hypervisor with tight integration to the host operating system means that Hyper-V offers unmatched performance when virtualizing said guests.

In this post, I am going to guide how to enable GPU acceleration in Hyper-V on Windows 10 guests. This is different from the previous RemoteFX approach (which is already discontinued, by the way), in which it creates a virtual GPU on guest and the guest application's call on graphics API upon that virtual GPU is interpreted, translated, forwarded to host, and back again. Besides inefficient, this approach means limited graphics API support as different hardware has different characteristics and supports a different set of APIs, where a common denominator has to be selected. Further, every time there is a new graphics API such as Vulkan or DirectX 12, the whole system has to be engineered. This approach is similar to those used in VMware Workstation and VirtualBox, and they share the same limitations as well.

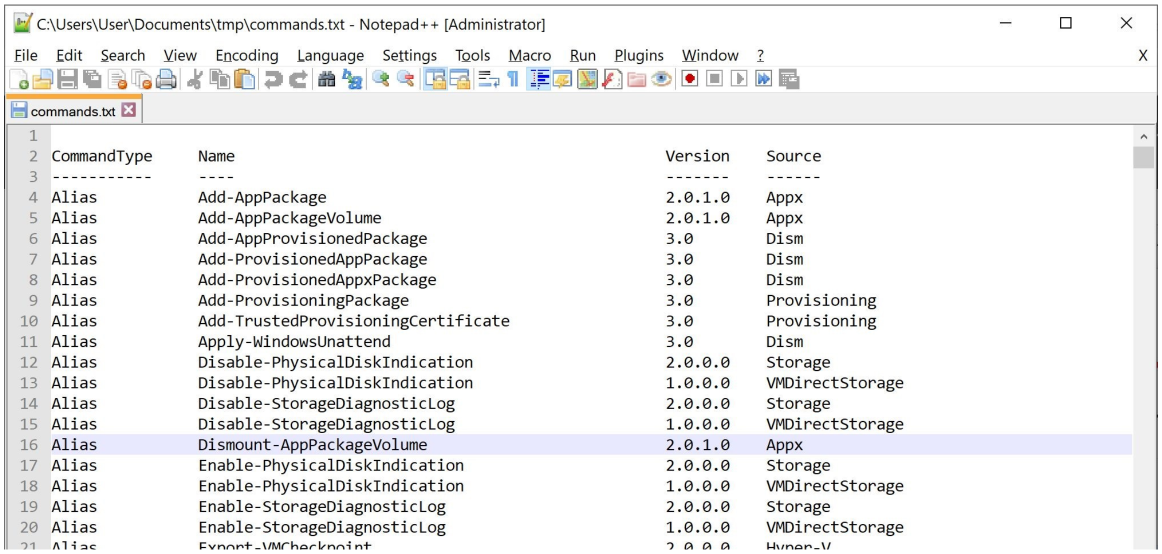
WDDM GPU Paravirtualization

Enter the WDDM GPU Paravirtualization (or GPU-PV for short). GPU-PV leverages the new graphics driver model in WDDM 2.5 (first introduced in Windows 10 version 1809) to provide support for virtualizing GPUs right from the graphics driver itself. The GPU is virtualized similarly to CPU (for example, multiple virtual machines and the host can share a single GPU, and a single virtual machine can fully utilize the GPU when the GPU is unused). Guest virtual machines use the *same GPU driver as the host*, which means that the full capabilities of the GPU are supported on the guest as well.

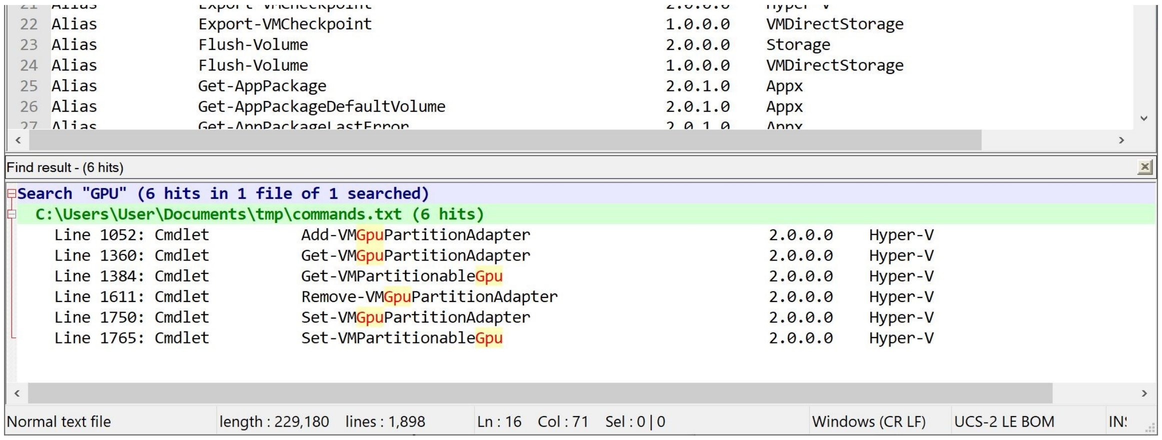
In this DirectX support for WSL 2 announcement, it is said that GPU-PV is already supported for WDAG (which was introduced way back in Windows 10 version 1809) and Windows Sandbox (introduced in Windows 10 version 1903). The announcement goes further that “GPU-PV is now a foundational part of Windows” and “Today this technology is limited to Windows guests, i.e. Windows running inside of a VM or container”. Indeed, as can be seen in this Windows Sandbox configuration reference, GPU support is explicitly mentioned there. Since Windows Sandbox is pretty limited, it would be nice to have this feature on a regular Windows 10 Hyper-V virtual machines. Unfortunately, documentation to enable this feature is scarce and practically nonexistent even on the internet.

Figuring It Out

Now cut to the chase, let’s start from PowerShell. Hyper-V virtual machines are typically managed with PowerShell, and perhaps we could find something there. Let’s begin with `Get-Command`, which will return all commands available in the current PowerShell session.


C:\Users\User\Documents\tmp\commands.txt - Notepad++ [Administrator]
File Edit Search View Encoding Language Settings Tools Macro Run Plugins Window ?
commands.txt

```

CommandType     Name                                               Version    Source
-----------     ----                                               -------    ------
Alias           Add-AppPackage                                     2.0.1.0    Appx
Alias           Add-AppPackageVolume                               2.0.1.0    Appx
Alias           Add-AppProvisionedPackage                          3.0        Dism
Alias           Add-ProvisionedAppPackage                          3.0        Dism
Alias           Add-ProvisionedAppxPackage                         3.0        Dism
Alias           Add-ProvisioningPackage                            3.0        Provisioning
Alias           Add-TrustedProvisioningCertificate                 3.0        Provisioning
Alias           Apply-WindowsUnattend                              3.0        Dism
Alias           Disable-PhysicalDiskIndication                     2.0.0.0    Storage
Alias           Disable-PhysicalDiskIndication                     1.0.0.0    VMDirectStorage
Alias           Disable-StorageDiagnosticLog                       2.0.0.0    Storage
Alias           Disable-StorageDiagnosticLog                       1.0.0.0    VMDirectStorage
Alias           Dismount-AppPackageVolume                          2.0.1.0    Appx
Alias           Enable-PhysicalDiskIndication                      2.0.0.0    Storage
Alias           Enable-PhysicalDiskIndication                      1.0.0.0    VMDirectStorage
Alias           Enable-StorageDiagnosticLog                        2.0.0.0    Storage
Alias           Enable-StorageDiagnosticLog                        1.0.0.0    VMDirectStorage
Alias           Export-VMCheckpoint                                2.0.0.0    Hyper-V
```


```
Alias          Export-VMCheckpoint                                1.0.0.0    VMDirectStorage
Alias          Flush-Volume                                       2.0.0.0    Storage
Alias          Flush-Volume                                       1.0.0.0    VMDirectStorage
Alias          Get-AppPackage                                     2.0.1.0    Appx
Alias          Get-AppPackageDefaultVolume                        2.0.1.0    Appx
```

Find result - (6 hits)

```
Search "GPU" (6 hits in 1 file of 1 searched)
  C:\Users\User\Documents\tmp\commands.txt (6 hits)
    Line 1052: Cmdlet          Add-VMGpuPartitionAdapter                          2.0.0.0    Hyper-V
    Line 1360: Cmdlet          Get-VMGpuPartitionAdapter                          2.0.0.0    Hyper-V
    Line 1384: Cmdlet          Get-VMPartitionableGpu                             2.0.0.0    Hyper-V
    Line 1611: Cmdlet          Remove-VMGpuPartitionAdapter                       2.0.0.0    Hyper-V
    Line 1750: Cmdlet          Set-VMGpuPartitionAdapter                          2.0.0.0    Hyper-V
    Line 1765: Cmdlet          Set-VMPartitionableGpu                             2.0.0.0    Hyper-V
```

Normal text file | length : 229,180 lines : 1,898 | Ln : 16 Col : 71 Sel : 0 | 0 | Windows (CR LF) | UCS-2 LE BOM | IN

Searching the output for the word “GPU” yields these results! Seems like we are on the right path! Let’s try to run those commands. First, let’s begin with the `Get-VMPartitionableGpu`, which yields the following:

```
Name                     : \\?\PCI#VEN_10DE&DEV_1C8D&SUBSYS_07BE1028&REV_A1#4&1
                           c845897dd59}\GPUPARAV
ValidPartitionCounts     : {32}
PartitionCount           : 32
TotalVRAM                : 1000000000
AvailableVRAM            : 1000000000
MinPartitionVRAM         : 0
MaxPartitionVRAM         : 1000000000
OptimalPartitionVRAM     : 1000000000
TotalEncode              : 18446744073709551615
AvailableEncode          : 18446744073709551615
MinPartitionEncode       : 0
MaxPartitionEncode       : 18446744073709551615
OptimalPartitionEncode   : 18446744073709551615
TotalDecode              : 1000000000
AvailableDecode          : 1000000000
MinPartitionDecode       : 0
MaxPartitionDecode       : 1000000000
OptimalPartitionDecode   : 1000000000
TotalCompute             : 1000000000
AvailableCompute         : 1000000000
MinPartitionCompute      : 0
MaxPartitionCompute      : 1000000000
OptimalPartitionCompute  : 1000000000
CimSession               : CimSession: .
ComputerName             : DESKTOP-61MEFBD
```

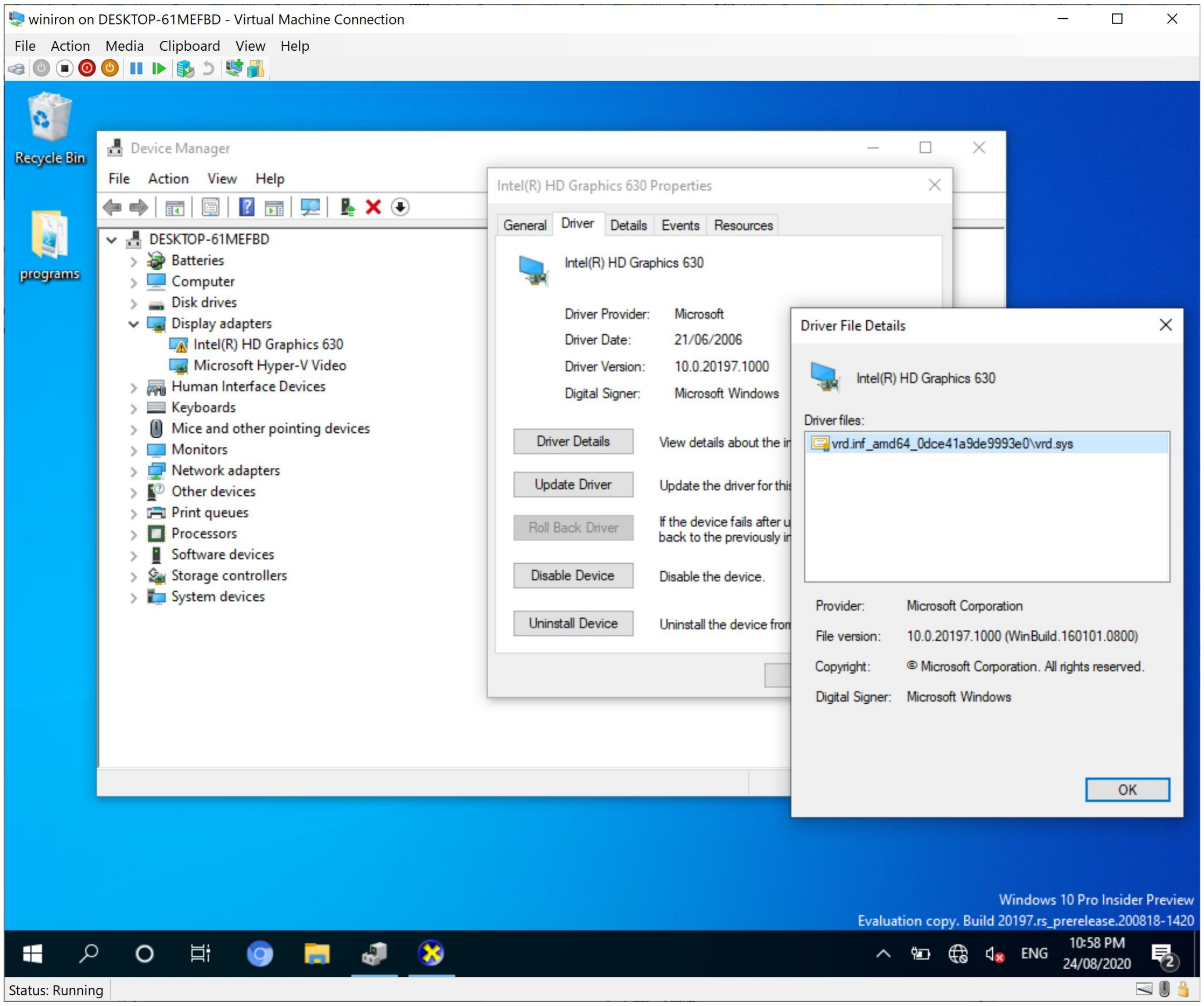
```
IsDeleted               : False

Name                    : \\?\PCI#VEN_8086&DEV_591B&SUBSYS_07BE1028&REV_04#3&1
                          45897dd59}\GPUPARAV
ValidPartitionCounts    : {32}
PartitionCount          : 32
TotalVRAM               : 1000000000
AvailableVRAM           : 1000000000
MinPartitionVRAM        : 0
MaxPartitionVRAM        : 1000000000
OptimalPartitionVRAM    : 1000000000
TotalEncode             : 18446744073709551615
AvailableEncode         : 18446744073709551615
MinPartitionEncode      : 0
MaxPartitionEncode      : 18446744073709551615
OptimalPartitionEncode  : 18446744073709551615
TotalDecode             : 1000000000
AvailableDecode         : 1000000000
MinPartitionDecode      : 0
MaxPartitionDecode      : 1000000000
OptimalPartitionDecode  : 1000000000
TotalCompute            : 1000000000
AvailableCompute        : 1000000000
MinPartitionCompute     : 0
MaxPartitionCompute     : 1000000000
OptimalPartitionCompute : 1000000000
CimSession              : CimSession: .
ComputerName            : DESKTOP-61MEFBD
IsDeleted               : False
```

It lists two GPUs here, which is what my system has (the Dell XPS 9560 with the NVIDIA Optimus system). Based on the PCI vendor IDs, the first one must correspond to the NVIDIA GeForce GTX 1050, and the second one to the Intel HD Graphics 630. The opposite of this command, `Set-VMPartitionableGpu`, seems to be used only for limiting the partition count.

Now, let's try to set up a Hyper-V Windows 10 virtual machine, and try to run the `Set-VMGpuPartitionAdapter` on that virtual machine. This command accepts a plethora of options, which includes many things listed from the `Get-VMPartitionableGpu` above. However, we will skip those options and stick with the default settings.

```
Set-VMGpuPartitionAdapter -VMName $vmname
```

Now, let's start the VM. And here we go! We got this:



Intel(R) HD Graphics 630

Driver Provider:	Microsoft
Driver Date:	21/06/2006
Driver Version:	10.0.20197.1000
Digital Signer:	Microsoft Windows

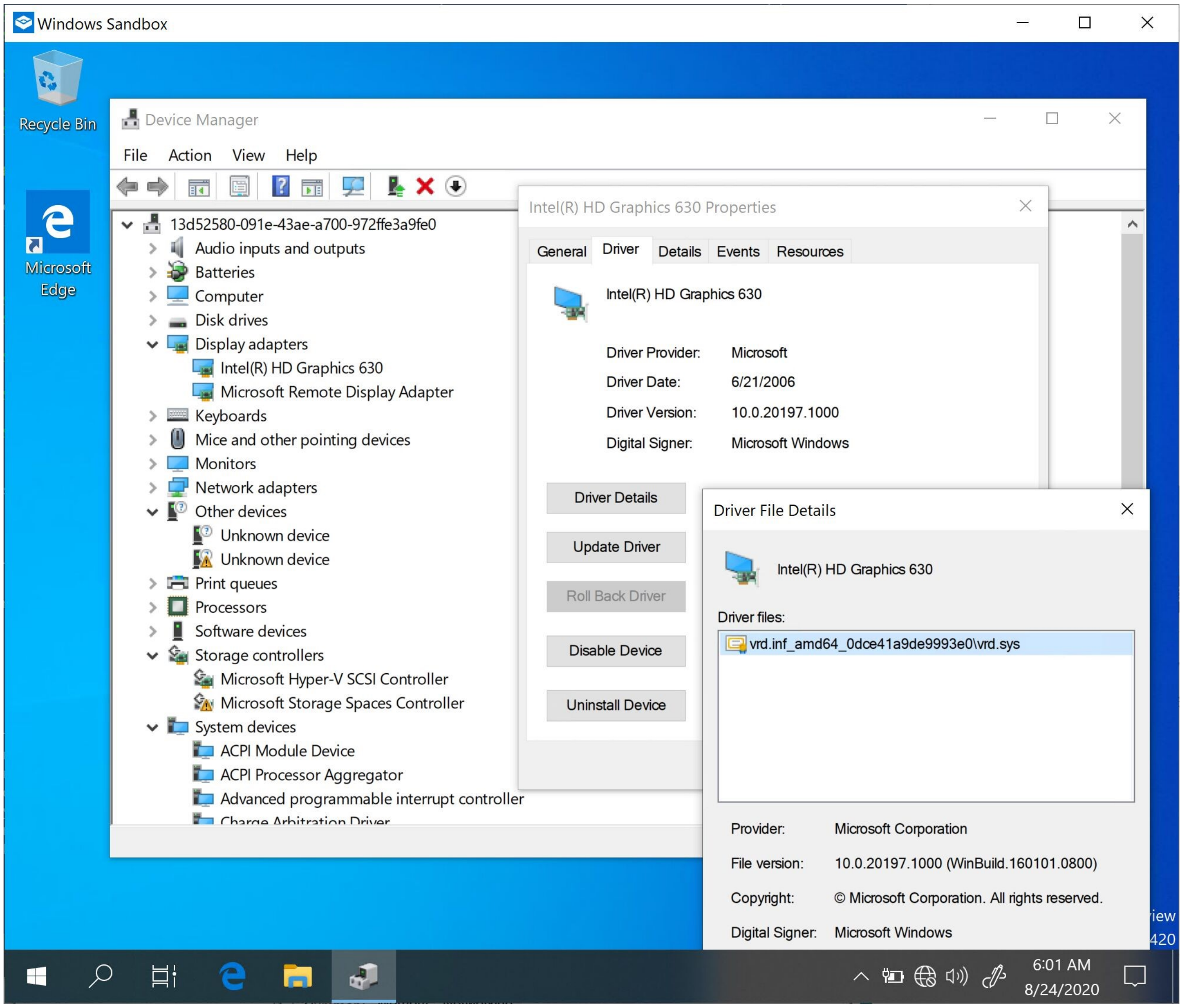
Intel(R) HD Graphics 630

Driver files:

vrd.inf_amd64_0dce41a9de9993e0\vrd.sys

Provider:	Microsoft Corporation
File version:	10.0.20197.1000 (WinBuild.160101.0800)
Copyright:	© Microsoft Corporation. All rights reserved.
Digital Signer:	Microsoft Windows

Neat! Now GPU is showing up with the same name as the host's GPU. However, the driver being used is `vrd.sys` and there is error code 43 as well. GPU acceleration still does not work obviously. Let's go back to Windows Sandbox and see what it is doing there.



```
13d52580-091e-43ae-a700-972ffe3a9fe0
  Audio inputs and outputs
  Batteries
  Computer
  Disk drives
  Display adapters
    Intel(R) HD Graphics 630
    Microsoft Remote Display Adapter
  Keyboards
  Mice and other pointing devices
  Monitors
  Network adapters
  Other devices
    Unknown device
    Unknown device
  Print queues
  Processors
  Software devices
  Storage controllers
    Microsoft Hyper-V SCSI Controller
    Microsoft Storage Spaces Controller
  System devices
    ACPI Module Device
    ACPI Processor Aggregator
    Advanced programmable interrupt controller
    Charge Arbitration Driver
```

Intel(R) HD Graphics 630 Properties

General | Driver | Details | Events | Resources

Intel(R) HD Graphics 630

Driver Provider:	Microsoft
Driver Date:	6/21/2006
Driver Version:	10.0.20197.1000
Digital Signer:	Microsoft Windows

Driver File Details

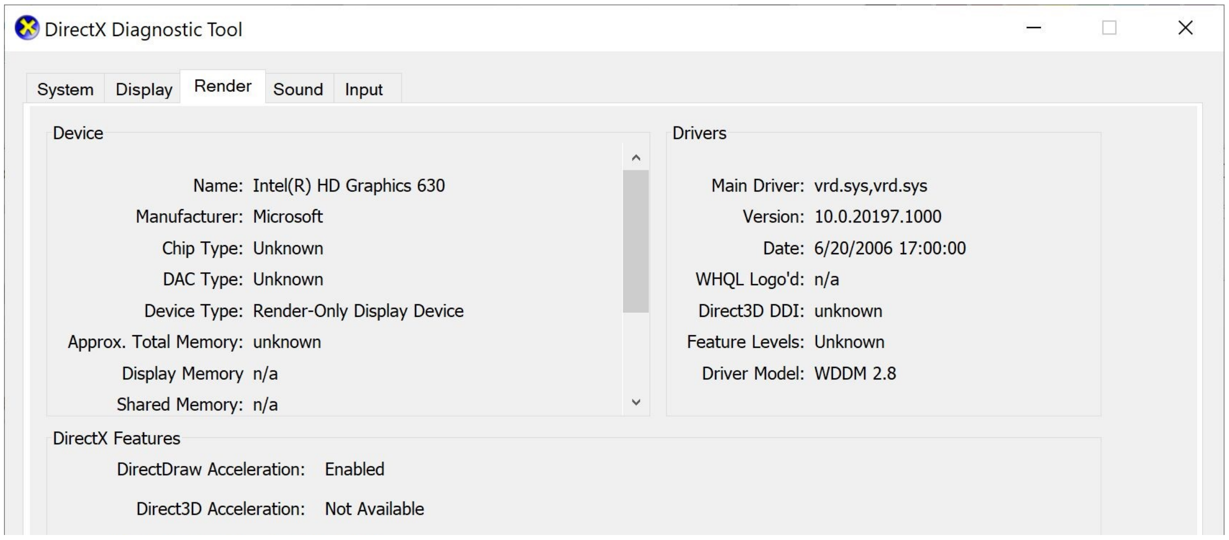
Intel(R) HD Graphics 630

Driver files:

vrd.inf_amd64_0dce41a9de9993e0\vrd.sys

Provider:	Microsoft Corporation
File version:	10.0.20197.1000 (WinBuild.160101.0800)
Copyright:	© Microsoft Corporation. All rights reserved.
Digital Signer:	Microsoft Windows

The Windows Sandbox has the same behavior as our VM! The GPU shows up as Intel HD Graphics and is using `vrd.sys` as well. However, GPU acceleration is working fine here. What gives? Let's continue to `dxdiag` inside Windows Sandbox.



Device

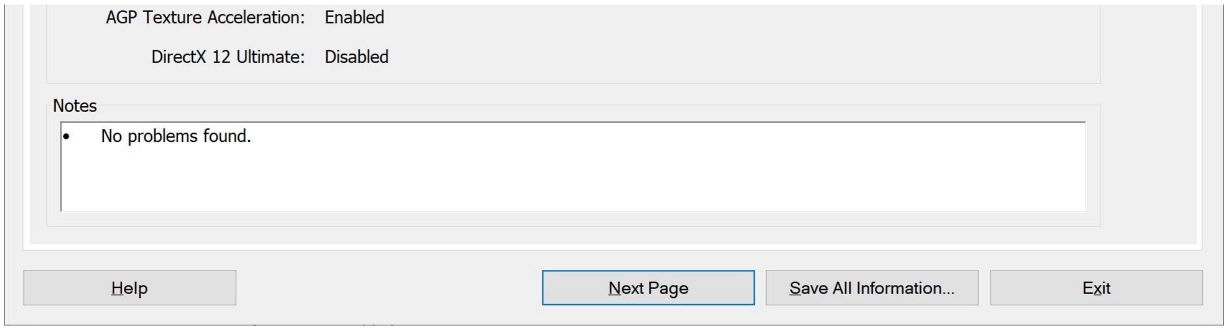
Name:	Intel(R) HD Graphics 630
Manufacturer:	Microsoft
Chip Type:	Unknown
DAC Type:	Unknown
Device Type:	Render-Only Display Device
Approx. Total Memory:	unknown
Display Memory	n/a
Shared Memory:	n/a

Drivers

Main Driver:	vrd.sys,vrd.sys
Version:	10.0.20197.1000
Date:	6/20/2006 17:00:00
WHQL Logo'd:	n/a
Direct3D DDI:	unknown
Feature Levels:	Unknown
Driver Model:	WDDM 2.8

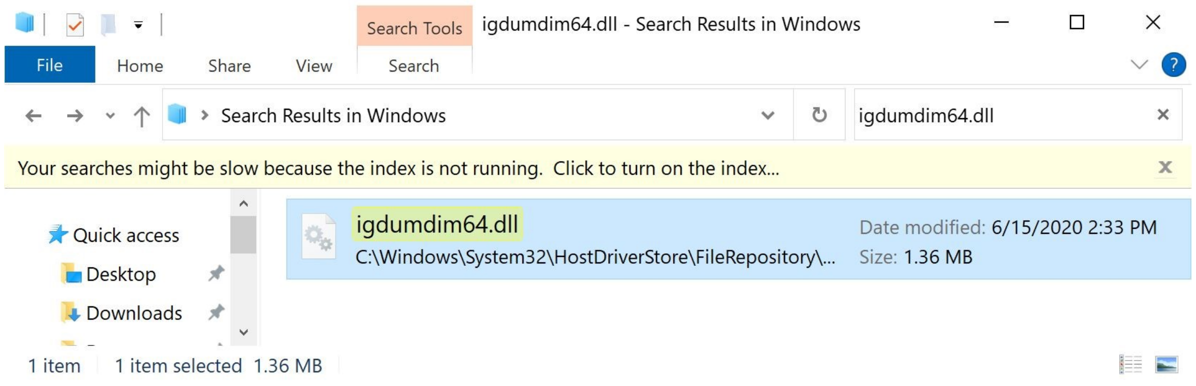
DirectX Features

DirectDraw Acceleration:	Enabled
Direct3D Acceleration:	Not Available


AGP Texture Acceleration: Enabled
DirectX 12 Ultimate: Disabled

Notes
- No problems found.

Here, we can see that we got a render-only device. Both name and Driver Model (WDDM version) correspond to host drivers. However, other elements do not seem to match up. Perhaps `vrd.sys` might internally load Intel's driver somewhere? One thing for sure is that Windows Sandbox mounts certain components from the host Windows operating system, so the host's drivers might be mounted somewhere as well. In the host's `dxdiag`, one of the driver file listed is `igdumdim64.dll`.


igdumdim64.dll - Search Results in Windows

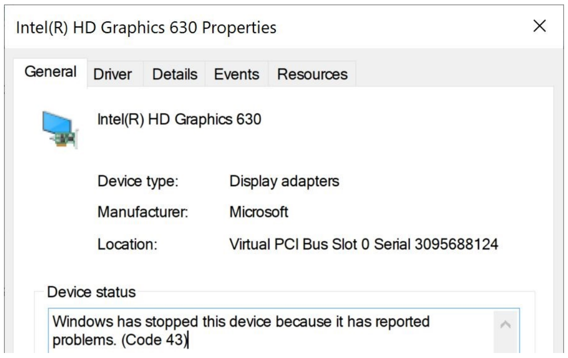
Search Results in Windows
igdumdim64.dll

Your searches might be slow because the index is not running. Click to turn on the index...

igdumdim64.dll
C:\Windows\System32\HostDriverStore\FileRepository\...
Date modified: 6/15/2020 2:33 PM
Size: 1.36 MB

1 item 1 item selected 1.36 MB

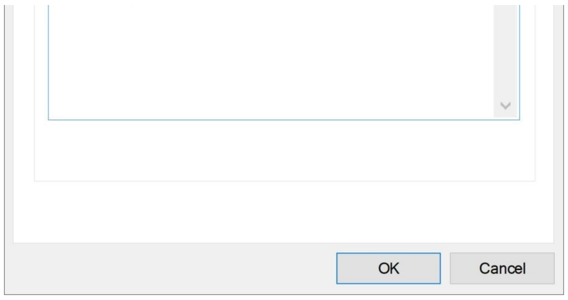
Lo and behold, searching for this file in `C:\Windows` folder results in a new discovery! There is a folder in Windows Sandbox called `C:\Windows\System32\HostDriverStore`, where the host's `C:\Windows\System32\DriverStore` of hosts get mounted here. This folder is not available in either host or our custom Windows 10 VM. So, let's copy the driver inside host's `DriverStore` that corresponds to the Intel HD Graphics (this information could be found in either `dxdiag` or Windows Device Manager by opening "Driver Details").


Intel(R) HD Graphics 630 Properties

Intel(R) HD Graphics 630

Device type: Display adapters
Manufacturer: Microsoft
Location: Virtual PCI Bus Slot 0 Serial 3095688124

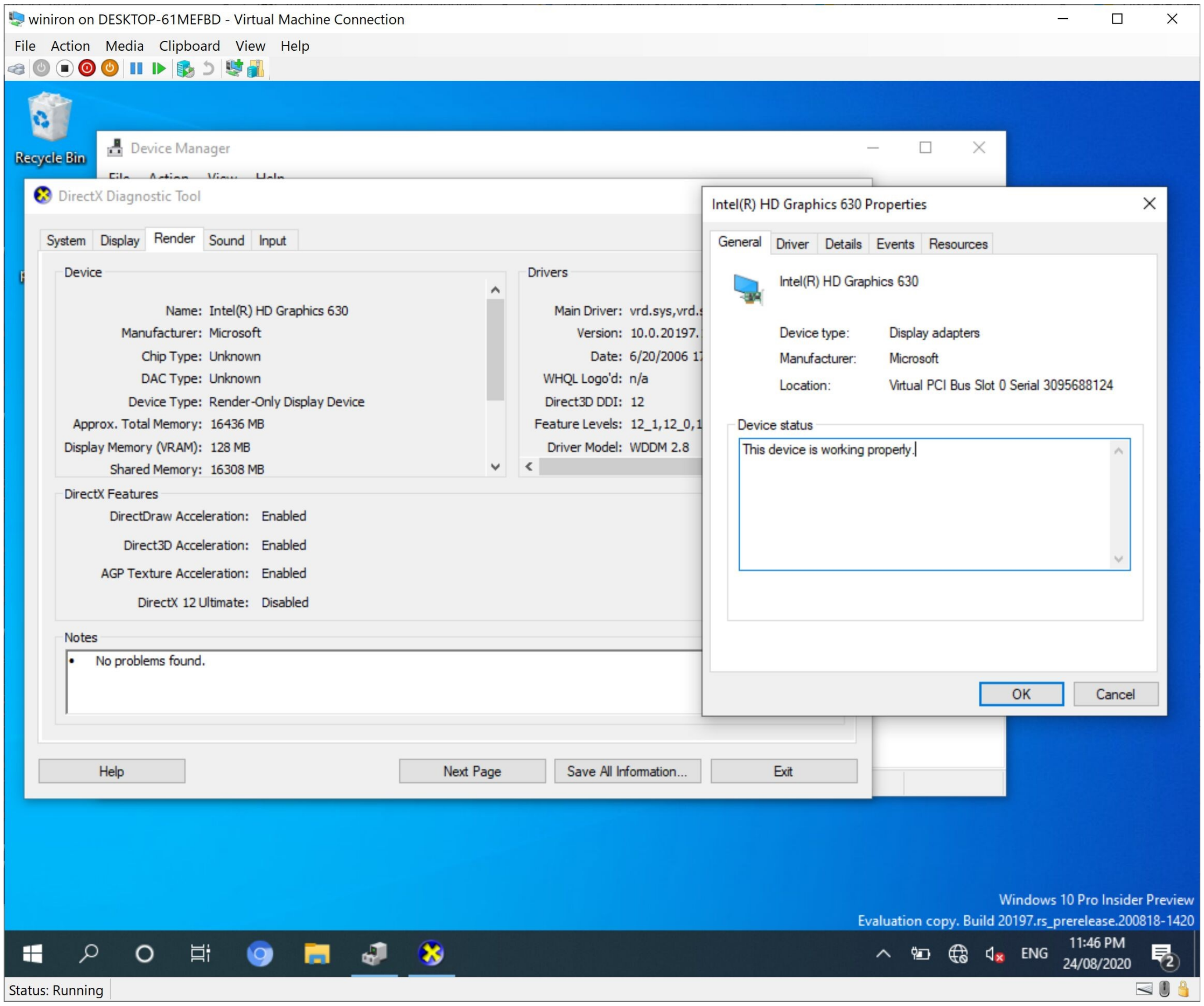
Device status
Windows has stopped this device because it has reported problems. (Code 43)



After copying the files, starting the virtual machine resulted in the virtual machine stuck at the Hyper-V boot logo for a few seconds until finally, we got to the login screen and … no GPU acceleration 😥 . Opening the device manager still shows error 43. Time to give up? Not yet! Perhaps the partitioned GPU is lacking a memory-mapped IO (MMIO) virtual address space?

Hyper-V supports another kind of GPU for the virtual machines, which is Discrete Device Assignment (DDA). DDA fully dedicates a GPU to a guest virtual machine. This blog post explains how to enlarge the MMIO space for both under 4 GB mark and above it for use with DDA. The post also suggests that “write-combining” be enabled. So let’s follow the post’s suggestions:

```
Set-VM -VMName $vmname -GuestControlledCacheTypes $true -LowMemoryMappedIoSpac
```


winiron on DESKTOP-61MEFBD - Virtual Machine Connection

DirectX Diagnostic Tool — Render

Device

Name: Intel(R) HD Graphics 630
Manufacturer: Microsoft
Chip Type: Unknown
DAC Type: Unknown
Device Type: Render-Only Display Device
Approx. Total Memory: 16436 MB
Display Memory (VRAM): 128 MB
Shared Memory: 16308 MB

Drivers

Main Driver: vrd.sys,vrd.s
Version: 10.0.20197.
Date: 6/20/2006 1
WHQL Logo'd: n/a
Direct3D DDI: 12
Feature Levels: 12_1,12_0,1
Driver Model: WDDM 2.8

DirectX Features

DirectDraw Acceleration: Enabled
Direct3D Acceleration: Enabled
AGP Texture Acceleration: Enabled
DirectX 12 Ultimate: Disabled

Notes

- No problems found.

Intel(R) HD Graphics 630 Properties

Intel(R) HD Graphics 630
Device type: Display adapters
Manufacturer: Microsoft
Location: Virtual PCI Bus Slot 0 Serial 3095688124

Device status

This device is working properly.

Windows 10 Pro Insider Preview
Evaluation copy. Build 20197.rs_prerelease.200818-1420

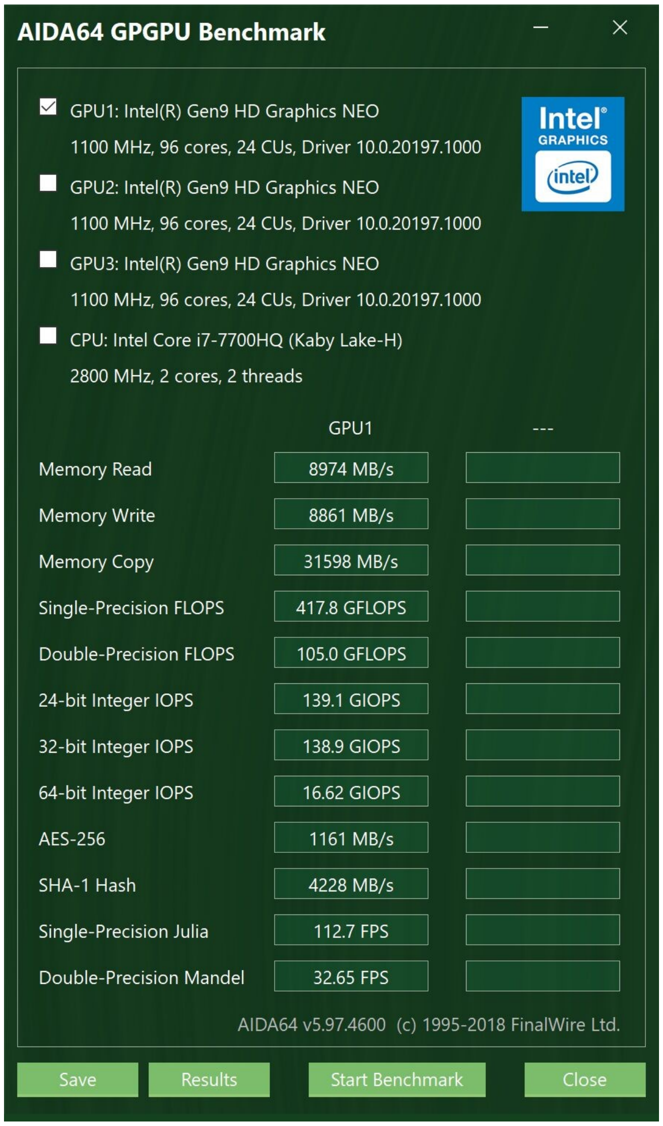
11:46 PM 24/08/2020

Status: Running

And here we go! GPU acceleration works the same way as it is in Windows Sandbox! Now time to do some benchmarks!

Benchmarking

Let’s start with the AIDA64’s compute benchmark. This benchmark performs various relatively simple tasks using OpenCL API. The first image (the one that says “HD Graphics NEO”) is the result from the guest, and the other is from the host. The results are pretty good! Well within the margin of error of the host’s GPU performance!



AIDA64 GPGPU Benchmark

☑ GPU1: Intel(R) Gen9 HD Graphics NEO
1100 MHz, 96 cores, 24 CUs, Driver 10.0.20197.1000

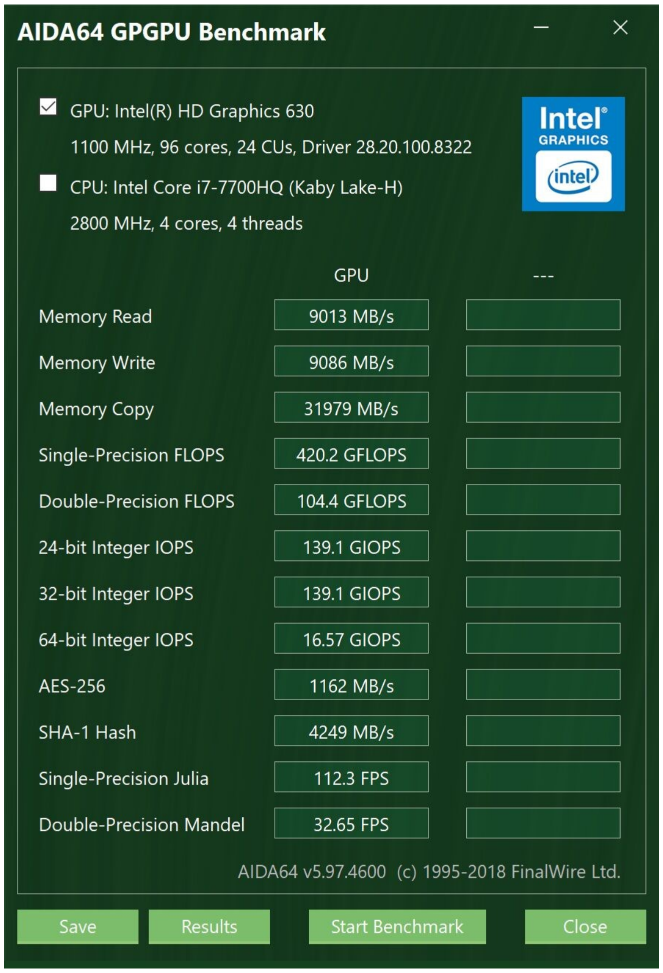
☐ GPU2: Intel(R) Gen9 HD Graphics NEO
1100 MHz, 96 cores, 24 CUs, Driver 10.0.20197.1000

☐ GPU3: Intel(R) Gen9 HD Graphics NEO
1100 MHz, 96 cores, 24 CUs, Driver 10.0.20197.1000

☐ CPU: Intel Core i7-7700HQ (Kaby Lake-H)
2800 MHz, 2 cores, 2 threads

	GPU1	---
Memory Read	8974 MB/s	
Memory Write	8861 MB/s	
Memory Copy	31598 MB/s	
Single-Precision FLOPS	417.8 GFLOPS	
Double-Precision FLOPS	105.0 GFLOPS	
24-bit Integer IOPS	139.1 GIOPS	
32-bit Integer IOPS	138.9 GIOPS	
64-bit Integer IOPS	16.62 GIOPS	
AES-256	1161 MB/s	
SHA-1 Hash	4228 MB/s	
Single-Precision Julia	112.7 FPS	
Double-Precision Mandel	32.65 FPS	

AIDA64 v5.97.4600 (c) 1995-2018 FinalWire Ltd.



AIDA64 GPGPU Benchmark

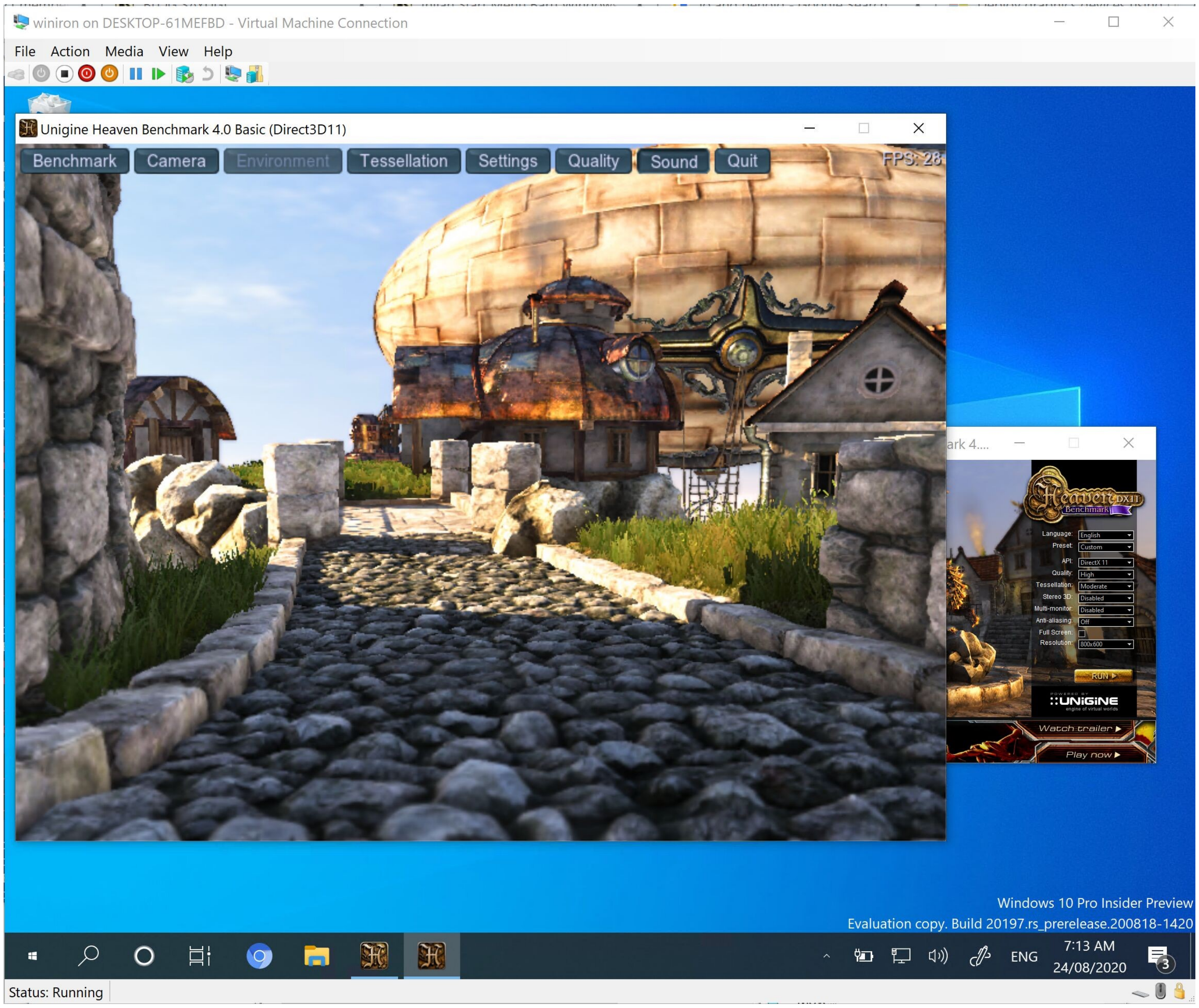
☑ GPU: Intel(R) HD Graphics 630
1100 MHz, 96 cores, 24 CUs, Driver 28.20.100.8322

☐ CPU: Intel Core i7-7700HQ (Kaby Lake-H)
2800 MHz, 4 cores, 4 threads

	GPU	---
Memory Read	9013 MB/s	
Memory Write	9086 MB/s	
Memory Copy	31979 MB/s	
Single-Precision FLOPS	420.2 GFLOPS	
Double-Precision FLOPS	104.4 GFLOPS	
24-bit Integer IOPS	139.1 GIOPS	
32-bit Integer IOPS	139.1 GIOPS	
64-bit Integer IOPS	16.57 GIOPS	
AES-256	1162 MB/s	
SHA-1 Hash	4249 MB/s	
Single-Precision Julia	112.3 FPS	
Double-Precision Mandel	32.65 FPS	

AIDA64 v5.97.4600 (c) 1995-2018 FinalWire Ltd.

Now for the Unigine Heaven benchmark, set at 800×600 windowed, high quality, moderate tessellation, and with DirectX 11 API.


winiron on DESKTOP-61MEFBD - Virtual Machine Connection
File Action Media View Help
Unigine Heaven Benchmark 4.0 Basic (Direct3D11)
Benchmark Camera Environment Tessellation Settings Quality Sound Quit
FPS: 28
ark 4....
Heaven DX11 Benchmark
RUN
POWERED BY UNIGINE engine of virtual worlds
Watch trailer
Play now
ENG
7:13 AM
24/08/2020
Status: Running

Language: English
Preset: Custom
API: DirectX 11
Quality: High
Tessellation: Moderate
Stereo 3D: Disabled
Multi-monitor: Disabled
Anti-aliasing: Off
Full Screen: ☐
Resolution: 800x600

Windows 10 Pro Insider Preview
Evaluation copy. Build 20197.rs_prerelease.200818-1420

The guest yielded an average FPS of 24.5 and a score of 617, whereas the host yielded an average FPS of 41.6 and a score of 1047. A pretty significant amount of performance loss this time, but anyways, nothing else even came close in the performance and API parity in GPU virtualization!

Now the only thing remaining is the ability to select which GPU should be used for the virtual machine. Unfortunately, I could not figure it out yet. The GPU-PV related commands shown at the beginning of this post does not seem to indicate that we can choose GPUs manually, without disabling the GPUs that we don't need. Please comment if you know the way!

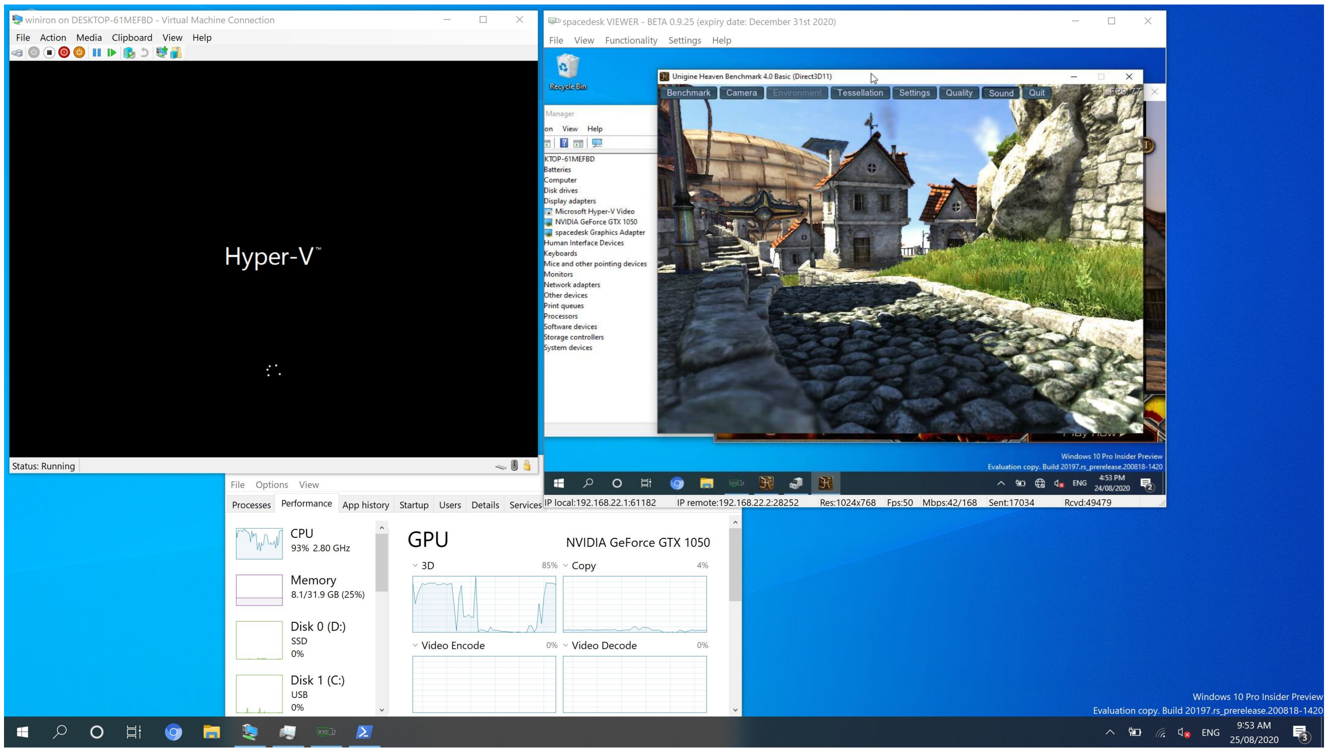
System Requirements

Since there is no official documentation regarding this feature, I don't know the exact system requirements. However, after several testing with several computer systems, I can summarize the system requirements as follows:

- Windows 10 version 1903 or newer for both host and guest I also tested GPU-PV on Windows 10 version 1809 as well. It kind of works but after booting the guest, the guest stuck.
- GPU driver with WDDM 2.5 or higher It doesn't matter if the driver is DCH or not. GPU drivers supporting WDDM 2.4 or lower are just outright not listed on `Get-VMPartitionableGpu`. Tip: you can check the WDDM version easily by looking at the first number of the driver version stated in the INF file (this version number could be different from the "marketed" version number, especially on NVIDIA GPUs). Typically, but not always all big three GPU vendors made this number correspond to the WDDM version is supported. For instance, number 27 corresponds to WDDM 2.7 support.
- GPU must drive the display directly I only tested this on NVIDIA GPUs with Optimus configuration, but this limitation might apply to other "headless" GPUs as well. Such GPUs simply don't work (guest's graphics got stuck on boot). See the next section for more details.

Partitioning a Render Only GPU

Render only GPU drivers do not have display outputs attached to them. This is commonly the case with "compute-only" GPUs, such as NVIDIA Tesla and AMD Radeon Instinct. However, these GPUs are also common on laptops with NVIDIA Optimus configuration, where the NVIDIA is the render only device. Testing GPU-PV on such GPUs (all tested are NVIDIA Optimus GPUs) on the released version of Windows 10 (version 2004 as of this writing) resulted in guest's graphics stuck. Guest's CPU still responds normally, and the guest responds to remote desktop requests as well (remote desktop has to have graphics acceleration disabled). Connecting to a remote desktop with graphics acceleration doesn't work, and neither does attaching an indirect display (IddCx) to the partitioned GPU. Windows Sandbox on Windows 10 version 2004 also crashes when used with such GPUs.


winiron on DESKTOP-61MEFBD - Virtual Machine Connection
Hyper-V
Status: Running
spacedesk VIEWER - BETA 0.9.25 (expiry date: December 31st 2020)
Unigine Heaven Benchmark 4.0 Basic (Direct3D11)
Microsoft Hyper-V Video
NVIDIA GeForce GTX 1050
spacedesk Graphics Adapter
GPU
NVIDIA GeForce GTX 1050
CPU 93% 2.80 GHz
Memory 8.1/31.9 GB (25%)
Windows 10 Pro Insider Preview
Evaluation copy. Build 20197.rs_prerelease.200818-1420

However, in the latest version of Windows 10 “Iron” insider preview (build 20197 as of this writing), GPU-PV on such GPUs works. The only caveat is that the display output cannot be seen from the built-in Hyper-V video driver: we have to use either a graphic accelerated RDP (which is always the case when we use the Hyper-V’s enhanced session mode) or IddCx drivers. More things regarding IddCx will warrant a separate post :), however, my favorite IddCx driver right now is SpaceDesk, which will stream whatever displayed on the driver over IP.

In fact, the guest performs better than the host in this case! Guest scored 2424 at 96.2 average FPS, whereas host only scored at 2246 at 89.2 average FPS. This disparity might come down to the Intel’s SpeedShift on the i7 7700HQ mispredicted the workload and throttles it up and down occasionally. Locking host’s CPU to base clock should improve host’s performance, but not guest’s, as the CPU was pretty busy as well in guest workloads: it has to encode the stream from IddCx display, trasmit the data over local TCP/IP (which has their own overheads), and decode it back. The system used is my personal laptop, which is also used to take the above screenshot.

TL; DR;

To enable GPU virtualization (GPU-PV) on arbitrary Hyper-V Windows 10 virtual machines, do the following:

1. Ensure that both the host and the guest meet the system requirements, and use the generation 2 virtual machines.
2. The following commands are to be done in PowerShell. Check that GPU-PV is available by executing the `Get-VMPartitionableGpu`. If more than one GPUs are available, the first one listed will be used. Currently, the only way to "select" GPU in use is by disabling other GPUs until the wanted GPU resides at the top of the output of the said command.
3. Copy the appropriate GPU driver from host to guest. Copying has to be done manually. Installing the GPU driver from an installation package will not work.
   1. From the host, find out the correct driver package path. Open Windows Device Manager, open the GPU to be used for GPU-PV, go to the "Driver" tab, click "Driver Details", and scroll down until you find a file with the following pattern: `(filename.inf)_amd64_(random_hex_number)\(somefiles.dll)`.
   2. On host, go to `C:\Windows\System32\DriverStore\FileRepository`. There should be a folder with the same name as above.
   3. Copy that folder to guest under `C:\Windows\System32\HostDriverStore\FileRepository`. If the folder is not there, create it.
4. Enable write-combining and enlarge MMIO address spaces on the guest (see here for details). The following example sets the 32-bit MMIO to 3 GB and above 32-bit MMIO to 30 GB.

```
Set-VM -VMName $vmname -GuestControlledCacheTypes $true -LowMemoryMappe
```

5. Mark the virtual machine to use GPU-PV.

```
Set-VMGpuPartitionAdapter -VMName $vmname
```

6. If everything works, the virtual machine should start with GPU acceleration. Enjoy! However, if it does not work, it is likely that you are partitioning a render only GPU. GPU-PV on such GPUs seems broken under Windows 10 version 2004 but seems to be fixed as well for the next version of Windows

10. See this section for details.

Hyper-V

Solution to Visual Studio Code “Reconnecting” Problem on WSL 2

5 thoughts on “GPU Virtualization with Hyper-V”

HugehardDoors
December 1, 2020 at 6:51 am

Thank you for your guide.
After some testing ,I find the GPU partitioning feature exists since windows client insider 17074 (1803))with wddm2.4 graphic driver model.

on wddm 2.4 system(before windows insider 17723),the host and guest must have the same main kernel version to make guest gpu work.

on wddm 2.5 and later system(windows insider 17723 and later) ,the host and guest can be different os version, for example ,you can run 20h2 in the guest and the host can be 1809.
to prevent host bsod on older system(before wddm2.7),you can make a wrapper dll to prevent apps enter full screen mode.

Reply

james
January 21, 2021 at 10:18 am

What an interesting insight. Thank you!

That’s why I heard some early adopters of GPU-PV complains that guest can crash the host. Since I am using this exclusively on Windows 10 version 2004 (which came with WDDM 2.7), I never experienced such issues.

Reply

yunfeng
January 22, 2021 at 12:37 pm

Thanks a lot of your guide,
I find a mistake in guide, the “write-combining” command is `Set-VM pickyourvmname -GuestControlledCacheTypes $true`.

Reply

james
February 1, 2021 at 1:18 pm

That’s spot on! Thank you for pointing out my mistake! I have fixed it.

Reply

Adam Reynolds
April 24, 2021 at 5:26 am

To select a specific adapter, you can use the command Get-VMHostPartitionableGpu to get the adapter's ID, then add it explicitly with:

Add-VMGpuPartitionAdapter -InstancePath [adapter ID from above]

E.g., on my system I have the onboard Radeon GPU + an RTX 2060, so I get the following for Get-VMHostPartitionableGpu (output truncated for readability):

```
PS C:\Users\a> Get-VMHostPartitionableGpu

Name : \\?
\PCI#VEN_1002&DEV_1636&SUBSYS_1F111043&REV_C5#4&12c9051d&0&0
041#{064092b3-625e-43bf-9eb5-dc845897dd59}\gpuparav
ValidPartitionCounts : {32}
PartitionCount : 32

Name : \\?
\PCI#VEN_10DE&DEV_1F12&SUBSYS_1F111043&REV_A1#4&21cf790e&0&0
009#{064092b3-625e-43bf-9eb5-dc845897dd59}\GPUPARAV
ValidPartitionCounts : {32}
PartitionCount : 32
```

The first entry is the Radeon onboard graphics, so this gets added if you just run Add-VMGpuPartitionAdapter without any arguments. `Name` in the output above is the `InstanceId` for the Add-VMGpuPartitionAdapter command.

Tested this on my dual-GPU machine and I was able to get my choice of GPU assigned to the VM and working.

Reply

Leave a Comment

Name *

Email *

Website

☐ Save my name, email, and website in this browser for the next time I comment.

Post Comment

Categories

Hyper-V

Linux