Confidential Computing Deployment Guide - (Intel TDX & KVM)

DU-11462-001_3.0 | February 2024

Documentation History

DU-11462-001

Version	Date	Authors	Description of Change
1.0	7/25/2023	Rob Nertney	Initial Version for Early Access
2.0	8/30/2023	Rob Nertney	Minor fixes. EA2 Updates for Kata/CoCo and TDX installs
3.0	2/22/2024	Rob Nertney	GA Version Release
4.0	7/09/2024	Rob Nertney	Updating instructions from MVP Intel stack to more upstreamable flows.

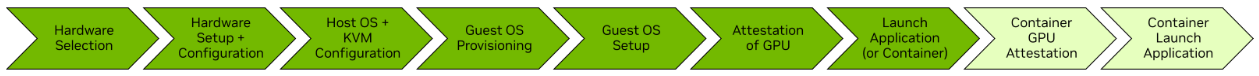
Table of Contents

Using This Guide

This guide is the most distilled set of instructions required to configure a system for Confidential Computing (CC) with the NVIDIA®Hopper™ H100 GPU. Explanations as to the value of a particular step, or the details of what is going on behind the scenes are covered in several of our other collateral, such as our whitepaper, GTC talks, and YouTube videos.

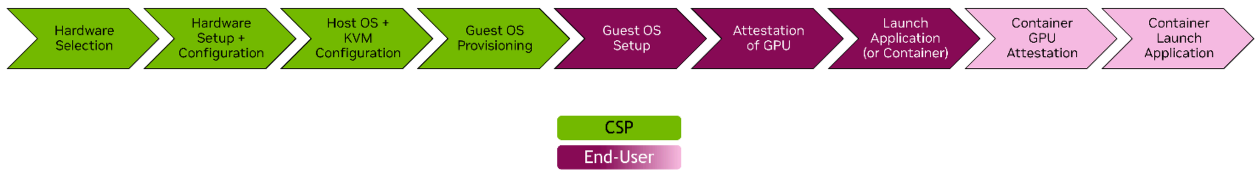
Here, you will find instructions that are targeted to various personas who want to use Hopper Confidential Compute (HCC). These personas are rough definitions of individuals who might have different responsibilities in the overall confidential system. The overall flow of using one is illustrated in Figure 1.

Figure 1. Overall Workflow


Hardware Selection
Hardware Setup + Configuration
Host OS + KVM Configuration
Guest OS Provisioning
Guest OS Setup
Attestation of GPU
Launch Application (or Container)
Container GPU Attestation
Container Launch Application

You can see that not every person involved in enabling and using CC will be required at every step. For example, a CSP might only provision a VM, and the user then takes over.

Figure 2. Workflow Example


Hardware Selection
Hardware Setup + Configuration
Host OS + KVM Configuration
Guest OS Provisioning
Guest OS Setup
Attestation of GPU
Launch Application (or Container)
Container GPU Attestation
Container Launch Application
CSP
End-User

In this example, the CSP does not require a policy for how often the GPU must be checked for integrity/validity, nor does it need to consider the infrastructure requirements for Confidential Containers. The user/tenant of the CSP does not need to consider the steps required to configure the GPU for confidential or non-confidential modes. Depending on who you are, and what your goals are, you might require all, or only a fraction, of the steps.

The following personas have been defined:

- **Hardware IT Administrator**
- **Host OS Administrator**

- **Virtual Machine Administrator**
- **Virtual Machine User**
- **Container User**

You can read the entire documentation or jump directly to the section that most accurately describes your persona use case. This guide is organized in a linear manner, so reading all sections in order will make logical sense to a developer who considers themselves all the above personas.

Document Structure

In this document, for code, if there is no prefix that is an output from a command.

```
$ shell-command to execute

# (optional) NVIDIA-commentary

sample output 1st row

sample output 2nd row

...
```

There might be times where, for the sake of simplicity, output will be omitted when not required to be noted. The below example shows `shell-command-A` and `shell-command-B`:

```
$ shell-command-A


$ shell-command-B
```

Output might occur after either of these commands, however, the output is not important (unless there are errors) and will not be included.

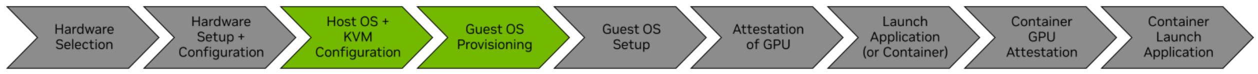
Supported Combinations

Due to the nascency of the CC market, many of the vendors, both hardware and software alike, are currently split in their tested-and-supported environments. As such, there (currently) is a very specific set of supported software/hardware combinations, outlined in the table below:

CPU Vendor	Operating System	Host Kernel	Hypervisor	Guest Kernel	Notes
Intel Emerald Rapids	Ubuntu 24.04	6.9+	KVM	6.8+	The only validated Intel TDX branch for nvTrust solutions is described in this guide

Host OS Administrator (Part-1)

Figure 3. The Host OS Administrator Persona


Hardware Selection
Hardware Setup + Configuration
Host OS + KVM Configuration
Guest OS Provisioning
Guest OS Setup
Attestation of GPU
Launch Application (or Container)
Container GPU Attestation
Container Launch Application

The Host OS Administrator is the persona that has received a system with its BIOS/UEFI configured so that it is *racked and stacked* with the CC modes enabled. This persona is responsible for selecting the Operating System (OS) that is installed on the host so that the OS can provision virtual machines (VMs). The roles are **System Architects**, **Cloud Administrators**, or **Advanced On-Premise Users**.

Setting Up the Host OS

This section provides information about setting up the Host OS.

Installing the Required Host Prerequisites

Install a supported Host OS by following their standard installation instructions. It is not important if you were using a different Linux kernel other than what is listed above. After completing these steps you will have the correct kernel installed.

Before building the Linux, some prerequisite software packages must be installed.

Preparing the Host

To install the prerequisites, run the following commands.:

```
# Packages to support the build
$ sudo apt update
$ sudo apt install build-essential libncurses-dev bison flex libssl-dev libelf-dev
debhelper-compat=12 meson ninja-build libglib2.0-dev python3-pip nasm iasl
```

We will now build a 6.9-based kernel with TDX support. This kernel will be installed on the host and will be the basis for guests that are created.

The following commands build the Host and Guest Linux kernel, Qemu, and the Ovmf BIOS that was used to launch the TDX guests.

Note: Here is some important information:

- The Intel TDX tree is continually evolving in sync with the kernel version.
- The only supported Intel TDX branch for HCC use with KVM is the device-passthrough.

For ease of use, we will be operating in the `/shared` directory and load all supporting items in this folder. You can modify the scripts to point to locations more reasonable to your system

```
# Ensure /shared has read/write permissions for the user via chmod
$ sudo mkdir /shared
$ cd /shared/
$ sudo chmod -R 777 /shared

# To apply git patches, you must set some settings:
$ git config --global user.email "youremail@yourdomain.com"
$ git config --global user.name "Your Name"
```

Downloading the GitHub Packages

```
# Clone the required GitHub Repositories:

## Mainline nvtrust
$ git clone https://github.com/NVIDIA/nvtrust.git

## Intel's patches to upstream
$ git clone https://github.com/intel/tdx-linux.git
$ cd tdx-linux
$ git checkout -b device-passthrough 1323f7b1ddf81076e3fcda6385c0c0dcf506258c

## Specific branch of Linux Kernel
```

```
$ git clone -b kvm-coco-queue-20240512
https://git.kernel.org/pub/scm/linux/kernel/git/vishal/kvm.git

## QEMU mainline
$ git clone https://gitlab.com/qemu-project/qemu

### Checkout the QEMU commit that supports H100 CC Passthrough
$ cd qemu
$ git checkout -b hcc-h100 ff6d8490e33acf44ed8afd549e203a42d6f813b5
$ cd ..

## OVMF
$ git clone -b edk2-stable202405 https://github.com/tianocore/edk2
```

Patch the GitHub Packages

```
# Patch the kernel:
$ cd /shared/tdx-linux/kvm
$ cp ../tdx-kvm/tdx_kvm_baseline_698ca1e40357.mbox .
$ git am --empty=drop tdx_kvm_baseline_698ca1e40357.mbox

# Patch QEMU
$ cd /shared/tdx-linux/qemu
$ cp ../tdx-qemu/tdx_qemu_baseline_900536d3e9.mbox .
$ git am --empty=drop tdx_qemu_baseline_900536d3e9.mbox
```

Note: If you receive errors about `error: unknown option –empty=drop` not being a valid flag, then you are likely running 22.04 as a host OS.

This solution ONLY works with Ubuntu 24.04 on the host.

Build the Kernel

Rebuild and package that are required by the Linux files.

```
$ cd /shared/tdx-linux/kvm
$ cp /boot/config-$(uname -r) .config
$ scripts/config -d KEXEC \
-d KEXEC_FILE \
-d SYSTEM_TRUSTED_KEYS \
-d SYSTEM_REVOCATION_KEYS

$ scripts/config -e KVM \
-e KVM_INTEL \
-e KVM_TDX_GUEST_DRIVER \
-e HYPERV \
-e INTEL_TDX_HOST \
-e CRYPTO_ECC \
-e CRYPTO_ECDH \
-e CRYPTO_ECDSA \
-e CRYPTO_ECRDSA


$ make oldconfig
# Press and hold “enter” when prompted for new features to add to the config file.

$ make -j$(nproc)
$ make modules -j$(nproc)
```

Install the & Configure the Host OS

```
$ sudo make modules_install
$ sudo make install

$ sudo sh -c “echo options kvm_intel tdx=on > /etc/modprobe.d/tdx.conf”

# Edit /etc/default/grub
# Change GRUB_CMDLINE_LINUX_DEFAULT to the following:

 GRUB_CMDLINE_LINUX_DEFAULT="nohibernate kvm_intel.tdx=on"

# save and close your editor
$ sudo update-grub
```

Build QEMU

```
$ cd /shared/tdx-linux/qemu

# Obtain the network package so the CVM can have internet access.
$ git clone  https://gitlab.freedesktop.org/slirp/libslirp.git
$ cd libslirp
$ meson build
$ sudo ninja -C build install
$ cd ..
```

```
# Use your preference to ensure that libslirp.so is in the ldconfig path
# Here is one option:
$ sudo ln -s /usr/local/lib/x86_64-linux-gnu/libslirp.so.0 /lib/x86_64-linux-gnu/

# Build & Install QEMU
$ ./configure --enable-slirp --enable-kvm --target-list=x86_64-softmmu
$ make -j$(nproc)
$ sudo make install
```

Build OVMF

```
$ cd /shared/tdx-linux/edk2

# initialize the submodules
$ git submodule update --init
```

Create a new file titled `build_ovmf.sh` and copy this code:

```
#!/bin/bash

rm -rf Build
make -C BaseTools
. edksetup.sh
cat <<-EOF > Conf/target.txt
	ACTIVE_PLATFORM = OvmfPkg/OvmfPkgX64.dsc
	TARGET = DEBUG
	TARGET_ARCH = X64
	TOOL_CHAIN_CONF = Conf/tools_def.txt
	TOOL_CHAIN_TAG = GCC5
	BUILD_RULE_CONF = Conf/build_rule.txt
	MAX_CONCURRENT_THREAD_NUMBER = $(nproc)
EOF
build clean
build

if [ ! -f Build/OvmfX64/DEBUG_GCC5/FV/OVMF.fd ]; then
	echo "Build failed, OVMF.fd not found"
	exit 1
fi

cp Build/OvmfX64/DEBUG_GCC5/FV/OVMF.fd ./OVMF.fd
```

Save and close the file above, and set it to executable via

```
$ chmod +x ./build_ovmf.sh
```

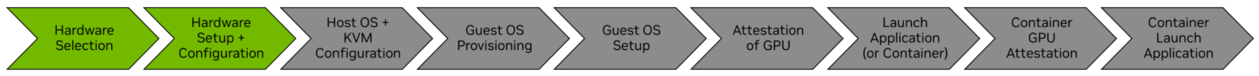
Execute the script:

```
$ ./build_ovmf.sh

# Reboot the host
$ sudo reboot
```

Ubuntu 24.04's kernel may not boot when TDX is pre-enabled in the BIOS/UEFI. As such, your Hardware IT Administrator should be involved in the next steps: configuring the System BIOS to enable TDX.

Hardware IT Administrator

Figure 4. The Hardware IT Administrator Persona


Hardware Selection
Hardware Setup + Configuration
Host OS + KVM Configuration
Guest OS Provisioning
Guest OS Setup
Attestation of GPU
Launch Application (or Container)
Container GPU Attestation
Container Launch Application

The Hardware IT Administrator persona is nedar the beginning of the CC chain and attention needs to be paid to selecting your CPU and GPU. This persona should contain **System Architects** and **IT Administrators**, selects the correct part numbers, and configures the BIOS/UEFI for the subsequent steps.

Selecting Hardware

CC requires CPUs and GPUs with specific functionality that enable the security outlined by the CC Consortium.

- CPU Requirements
  - Intel with TDX support
- GPU Requirements
  - NVIDIA Hopper H100 GPU
- Other Recommendations
  - Your motherboard vendor can be configured with Secure Boot and TDX enabled

To set up your system, you need to configure the motherboard's BIOS to enable the CC mode options. NVIDIA has tested the following motherboard vendors with Hopper CC and provided the BIOS menu-flows so that you can easily set them.

Setting Up the Hardware Setup and Configuring Your System

Supermicro System: BIOS Firmware Version 2.1

```
CPU Configuration -->
      Processor Configuration -->
            Limit CPU PA to 46 Bits -> Disable
            Intel TME, Intel TME-MT, Intel TDX -->
                  Total Memory Encryption (Intel TME) -> Enable
                  Total Memory Encryption (Intel TME) Bypass -> Auto
                  Total Memory Encryption Multi-Tenant (Intel TME-MT) -> Enable
                  Memory Integrity -> Disable
                  Intel TDX -> Enable
                  TDX Secure Arbitration Mode Loader (SEAM) -> Enabled
                  Disable excluding Mem below 1MB in CMR -> Auto
                  Intel TDX Key Split -> <Non-zero value>

            Software Guard Extension -> Enable
```

With the above System BIOS configured for Intel TDX, you are now ready to begin configuring the Host Operating System and the Hypervisor.

Host OS Administrator (Part-2)

After ensuring that you have built and installed Linux kernel enabling TDX, and configuring your BIOS/UEFI for the feature, you may continue onward:

Validating the Host Detects TDX

After the host reboots, to check that our kernel is the new TDX-aware version, and that our configuration options were correctly applied, run the following commands.

```
$ uname -a
```

```
Linux hcc-host 6.9.0-rc7+ #1 SMP PREEMPT_DYNAMIC Tue Jul  9 20:29:57 UTC 2024 x86_64
x86_64 x86_64 GNU/Linux


$ sudo dmesg | grep -i tdx

[sudo] password for user:

[    0.000000] Command line: BOOT_IMAGE=/vmlinuz-6.9.0-rc7+
root=/dev/mapper/ubuntu--vg-ubuntu--lv ro clearcpuid=mtrr,avx,avx2 nohibernate
kvm_intel.tdx=on
[    4.967868] Kernel command line: BOOT_IMAGE=/vmlinuz-6.9.0-rc7+
root=/dev/mapper/ubuntu--vg-ubuntu--lv ro clearcpuid=mtrr,avx,avx2 nohibernate
kvm_intel.tdx=on
[   10.580190] virt/tdx: BIOS enabled: private KeyID range [64, 128)
[   10.580193] virt/tdx: Disable ACPI S3. Turn off TDX in the BIOS to use ACPI S3.
[   21.364881] virt/tdx: 8405028 KB allocated for PAMT
[   21.364890] virt/tdx: module initialized
[   21.364909] virt/tdx: SEAMCALL (0x0000000000000022) failed: 0xc0000c0000000000
[   21.364913] virt/tdx: RCX 0x0000000000000000 RDX 0xffffffffffffffff R08
0x0000000000000000
[   21.364916] virt/tdx: R09 0x0000000000000000 R10 0x0000000000000000 R11
0x0000000000000000
```

Note: Errors of SEAMCALL (0x0000000000000022) failed: 0xc0000c0000000000 may be ignored for this release. This error occurs when you do not have the latest TDX-Module installed. Please contact your OEM for an updated BIOS.

Workaround instructions that do not modify the BIOS may be documented in a future release of this document.

Preparing to Launch a Guest Virtual Machine with KVM

This section covers how the Host Administrator can use KVM/QEMU to launch a Confidential VM (CVM) for a guest. These instructions can be followed by new developers who want to start from scratch , but you can modify the steps at your discretion.

Note: While the hypervisor set up and VM launch steps might be redundant for a developer who has an existing orchestration flow, there are steps that must be taken to enable the NVIDIA H100 in confidential modes.

(**Required**) Configuring the GPU for Confidential Compute Mode

The NVIDIA H100 can only be toggled into and out of CC modes with a privileged call from in the Host.
Here are the main flags:

- --query-cc-settings
  - Prints the current mode that the GPU is operating in
- --set-cc-mode <MODE>
  - Where MODE is
    - on
    - off
    - devtools

Refer to our whitepaper for more information about what the modes represent. NVIDIA has provided the following script to help facilitate this call.:

```
$ cd /shared/nvtrust
$ git submodule update --init
$ cd host_tools/python
```

Query the state to see how your H100's are configured

```
$ sudo python3 ./nvidia_gpu_tools.py --gpu-name=H100  --query-cc-mode


NVIDIA GPU Tools version v2024.02.14o
Command line arguments: ['./nvidia_gpu_tools.py', '--gpu-name=H100',
'--query-cc-mode']
Topo:
  Intel root port 0000:a7:01.0
   PCI 0000:a8:00.0 0x1000:0xc030
    PCI 0000:a9:03.0 0x1000:0xc030
```

```
     GPU 0000:ad:00.0 H100-PCIE 0x2321 BAR0 0x10e042000000

2024-07-09,22:35:42.530 INFO     Selected GPU 0000:ad:00.0 H100-PCIE 0x2321 BAR0
0x10e042000000

2024-07-09,22:35:42.530 INFO     GPU 0000:ad:00.0 H100-PCIE 0x2321 BAR0 0x10e042000000
CC mode is off
```

Change the state to enable CC mode:

```
$ sudo python3 ./nvidia_gpu_tools.py --gpu-name=H100 --set-cc-mode=on
--reset-after-cc-mode-switch

NVIDIA GPU Tools version v2024.02.14o

Command line arguments: ['./nvidia_gpu_tools.py', '--gpu-name=H100',
'--set-cc-mode=on', '--reset-after-cc-mode-switch']

Topo:

  Intel root port 0000:a7:01.0

   PCI 0000:a8:00.0 0x1000:0xc030

    PCI 0000:a9:03.0 0x1000:0xc030

     GPU 0000:ad:00.0 H100-PCIE 0x2321 BAR0 0x10e042000000

2024-07-09,22:37:21.028 INFO     Selected GPU 0000:ad:00.0 H100-PCIE 0x2321 BAR0
0x10e042000000

2024-07-09,22:37:21.173 INFO     GPU 0000:ad:00.0 H100-PCIE 0x2321 BAR0 0x10e042000000
CC mode set to on. It will be active after GPU reset.

2024-07-09,22:37:23.029 INFO     GPU 0000:ad:00.0 H100-PCIE 0x2321 BAR0 0x10e042000000
was reset to apply the new CC mode.
```

The GPU is now configured and ready to be directly assigned to your CVM. If you already have an orchestration flow for building, configuring, and so onVMs with KVM, you can skip the next section.

(**Optional**) Setting up the Guest VM

This section provides information about how to set up the guest VM.

Create the VM Base Image

1. Run the following commands to obtain an ISO of a supported operating system.
   In this example, we are using Ubuntu 24.04.
   ```
   $ cd /shared/nvtrust/host_tools/sample_kvm_scripts/isos
   ```

```
#download an ISO of a supported OS here, for example:

$ wget https://releases.ubuntu.com/24.04/ubuntu-24.04-live-server-amd64.iso
```

2. Create a blank VM drive, which is one file that acts as the VM's storage drive.

```
# Create the empty Virtual Disk Drive. Change 500G to your desired size
$ qemu-img create -f qcow2
/shared/nvtrust/host_tools/sample_kvm_scripts/images/ubuntu.qcow2 500G
```

Installing the Guest OS

Create a file: `/shared/launch_vm.sh` and put this info in it.

```
PREFIX="/shared/nvtrust/host_tools/sample_kvm_scripts/"



qemu-system-x86_64                  \
    -enable-kvm                     \
    -drive file=$PREFIX/images/ubuntu.qcow2,if=virtio \
    -smp cores=50,threads=2,sockets=2   \
    -m 512G                           \
    -cpu host                       \
    -netdev user,id=n1,ipv6=off,hostfwd=tcp::2022-:22      \
    -device virtio-net-pci,netdev=n1,mac=7c:c2:55:9f:64:58 \
    -nographic                      \
    -object tdx-guest,id=tdx        \
    -object memory-backend-ram,id=mem0,size=512G  \
    -machine
q35,kernel-irqchip=split,confidential-guest-support=tdx,memory-backend=mem0 \
    -bios /shared/tdx-tools/edk2/OVMF.fd                   \
    -vga none                       \
    -nodefaults                     \
    -serial stdio              \
    -object iommufd,id=iommufd0 \
    -device pcie-root-port,id=pci.1,bus=pcie.0 \
    -device vfio-pci,host=ad:00.0,bus=pci.1,iommufd=iommufd0 -fw_cfg
name=opt/ovmf/X-PciMmio64,string=262144 \
    -cdrom $PREFIX/isos/ubuntu-24.04-live-server-amd64.iso
```

Please ensure that the the following flags are appropriate for your system:

- -drive file=ubuntu.qcow2,if=virtio
  - Check the name and location of when you created it above
- -smp
  - Total cores, threads, and socket topology desired in your VM

- -m
  - Memory allocated to the VM
- -device virtio-net-pci,netdev=n1,mac=7c:c2:55:9f:64:58
  - Change the mac address; for the virtIO device
- -object memory-backend-ram,id=mem0,size=512G
  - Change this memory size to match your -m flag above
- -device vfio-pci,host=ad:00.0 (...)
  - Align this to the found H100 when you previously ran `lspci -d 10de: -nn`

Ensure that `launch_vm.sh` is executable:

```
$ chmod +x /shared/launch_vm.sh
```

Identifying the GPUs to be Passed Through to the Guest

In the Host OS, to identify an H100 to pass to our new Guest VM.

1. Identify the NVIDIA devices in the system.

   ```
   $ lspci -d 10de: -nn
   ad:00.0 3D controller [0302]: NVIDIA Corporation GH100 [H100L 94GB] [10de:2321]
   (rev a1)
   ```

   The value above tells us about an H100 that is found in the system:

   - The slot ID: ad`:00.0`
   - The device ID of the specific H100 in slot ad:00.0: **2321**

   KVM a Virtual Function I/O (VFIO), which is a Linux kernel feature that allows a VM to access and control physical hardware devices for improved performance as if they were directly connected to it.

2. Tell the host kernel that these device IDs should be allocated for VMs.

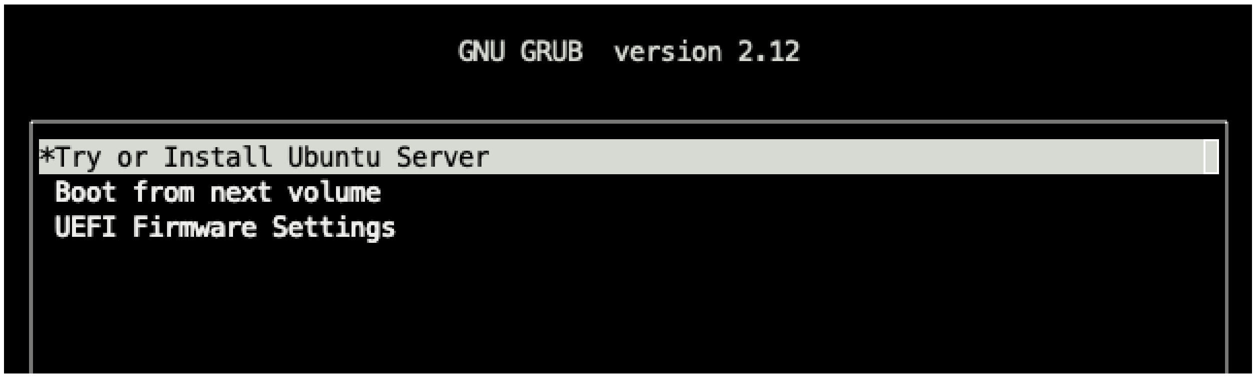
   ```
   # Insert the VFIO drivers
   $ sudo modprobe vfio
   $ sudo modprobe vfio_pci

   # Tag the H100s in your system as being ready for passthrough:
   $ sudo sh -c "echo 10de 2321 > /sys/bus/pci/drivers/vfio-pci/new_id"
   ```

Note: This assignment must be done each time the Host reboots. You can restart Guests multiple times or reassign the GPU(s) without repeating the steps.

Modifying GRUB to Print the Kernel Launch to the TTY

1. Launch `sudo /shared/launch_vm.sh`
2. Select install option "Try or Install Ubuntu Server"

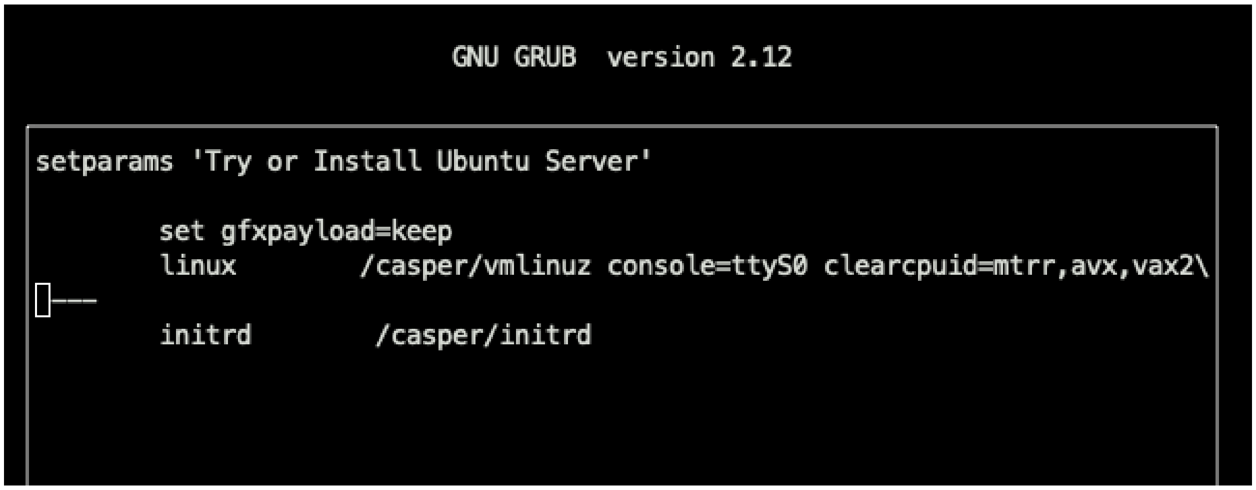
Figure 5. Selecting an Installation Option



```
GNU GRUB  version 2.12

*Try or Install Ubuntu Server
 Boot from next volume
 UEFI Firmware Settings
```

3. To edit the command, press **e**.

Figure 6. Editing the Command



```
GNU GRUB  version 2.12

setparams 'Try or Install Ubuntu Server'

        set gfxpayload=keep
        linux        /casper/vmlinuz console=ttyS0 clearcpuid=mtrr,avx,vax2\
---
        initrd        /casper/initrd
```

4. To modify the Linux launch and print to the local console, edit the following command.

```
linux        /casper/vmlinuz  console=ttyS0 clearcpuid=mtrr ---
```

Note: If you do not have the latest TDX-Module installed–generally delivered via the system-OEM's BIOS–AVX may not work, and you may encounter errors during boot, or instability during testing. If you notice this, please attempt to modify the code to:

```
linux        /casper/vmlinuz  console=ttyS0 clearcpuid=mtrr,avx,avx2 ---
```

5. To continue the launch, press CTRL+X.

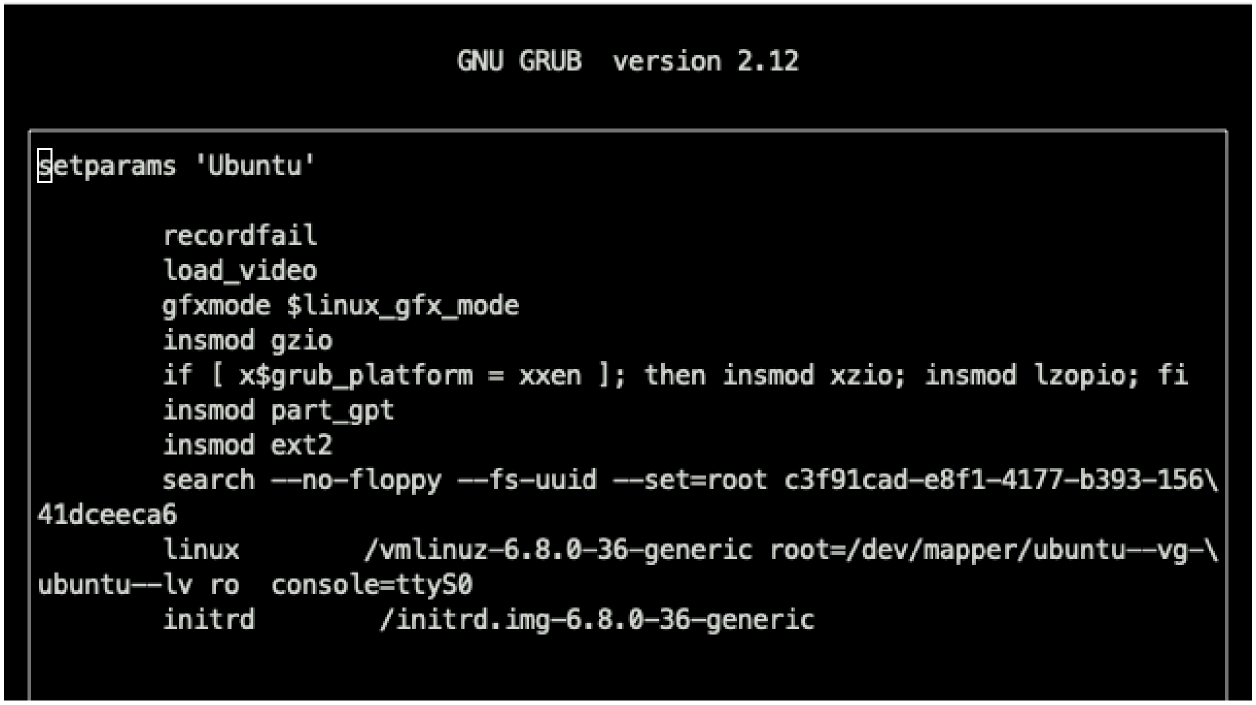
You can now configure the Guest OS install parameters. No specific options during this install are required. After the Guest OS is installed, Ubuntu will prompt you to reboot, and the VM will terminate, which returns you to the host.

Note: Due to CC limitations, a reboot command terminates the VM. This is expected for all subsequent reboots.

6. After you complete the install, and the reboot of the Guest VM is requested, QEMU will terminate. Modify `launch_vm.sh` afterwards and remove the `-cdrom` line, also removing the `\` on the line above:

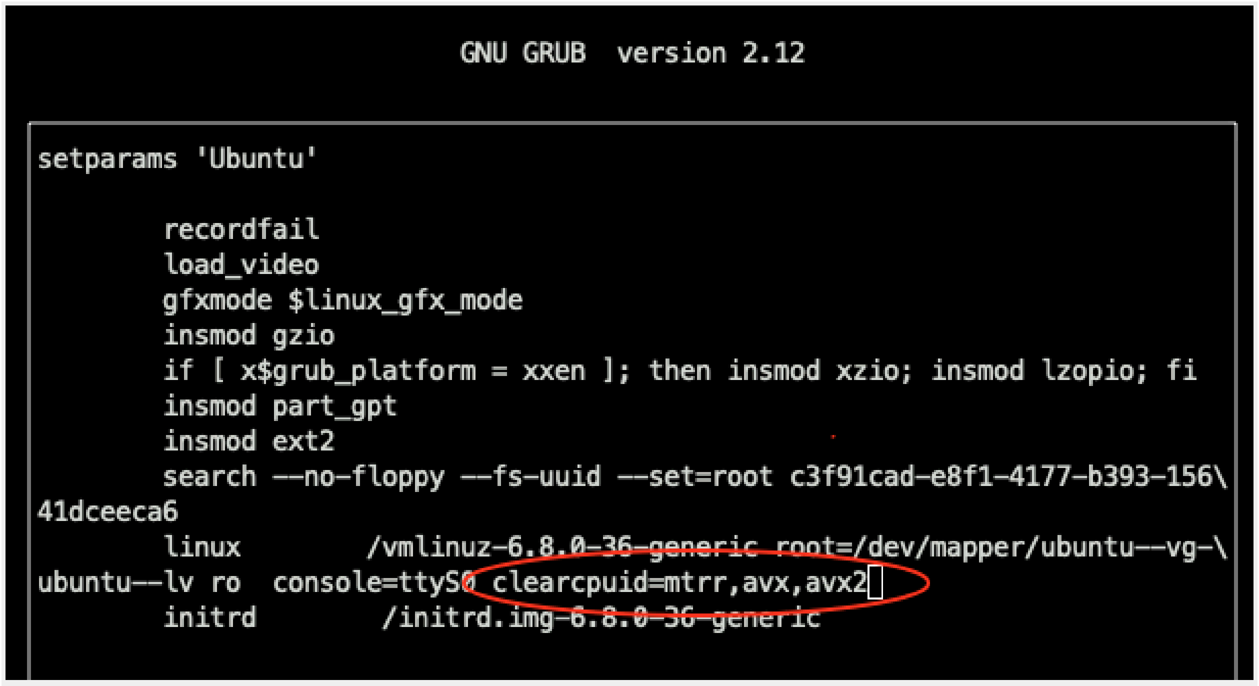
```
PREFIX="/shared/nvtrust/host_tools/sample_kvm_scripts/"


qemu-system-x86_64                    \
    -enable-kvm                       \
    -drive file=$PREFIX/images/ubuntu.qcow2,if=virtio \
    -smp cores=50,threads=2,sockets=2    \
    -m 512G                             \
    -cpu host                         \
    -netdev user,id=n1,ipv6=off,hostfwd=tcp::2022-:22      \
    -device virtio-net-pci,netdev=n1,mac=7c:c2:55:9f:64:58 \
    -nographic                        \
    -object tdx-guest,id=tdx          \
    -object memory-backend-ram,id=mem0,size=512G  \
    -machine
q35,kernel-irqchip=split,confidential-guest-support=tdx,memory-backend=mem0 \
    -bios /shared/edk2/OVMF.fd                     \
    -vga none                         \
    -nodefaults                       \
    -serial stdio               \
    -object iommufd,id=iommufd0 \
    -device pcie-root-port,id=pci.1,bus=pcie.0 \
    -device vfio-pci,host=ad:00.0,bus=pci.1,iommufd=iommufd0 -fw_cfg
name=opt/ovmf/X-PciMmio64,string=262144 ~~\~~
~~    -cdrom $PREFIX/isos/ubuntu-24.04-live-server-amd64.iso~~
```

7. Relaunch `launch_vm.sh`, stopping once more to hit `e` on the “Ubuntu” option to edit the launch parameters:



```
                              GNU GRUB  version 2.12

setparams 'Ubuntu'

        recordfail
        load_video
        gfxmode $linux_gfx_mode
        insmod gzio
        if [ x$grub_platform = xxen ]; then insmod xzio; insmod lzopio; fi
        insmod part_gpt
        insmod ext2
        search --no-floppy --fs-uuid --set=root c3f91cad-e8f1-4177-b393-156\
41dceeca6
        linux        /vmlinuz-6.8.0-36-generic root=/dev/mapper/ubuntu--vg-\
ubuntu--lv ro  console=ttyS0
        initrd        /initrd.img-6.8.0-36-generic
```

8. After `console=ttyS0` add once again `clearcpuid=mtrr,avx,avx2`



```
                              GNU GRUB  version 2.12

setparams 'Ubuntu'

        recordfail
        load_video
        gfxmode $linux_gfx_mode
        insmod gzio
        if [ x$grub_platform = xxen ]; then insmod xzio; insmod lzopio; fi
        insmod part_gpt
        insmod ext2
        search --no-floppy --fs-uuid --set=root c3f91cad-e8f1-4177-b393-156\
41dceeca6
        linux        /vmlinuz-6.8.0-36-generic root=/dev/mapper/ubuntu--vg-\
ubuntu--lv ro  console=ttyS0 clearcpuid=mtrr,avx,avx2
        initrd        /initrd.img-6.8.0-36-generic
```

9. Again, to continue the launch, press CTRL+X. We will make this option persist in a later step.

Validating the TDX Guest:

Once the CVM guest is launched, log in via SSH and check the dmesg log to validate the TDX hooks are detected. Change the username to match what you configured in previous steps.

Log In:

```
$ ssh -p2022 user@localhost
```

Check the kernel logs for TDX support:

```
nvidia@tdx-guest:~$ sudo dmesg | grep -i tdx
[sudo] password for user:
[    0.000000] tdx: Guest detected
[   29.669467] process: using TDX aware idle routine
[   29.669467] Memory Encryption Features active: Intel TDX
[   46.864001] systemd[1]: Detected confidential virtualization tdx.
```

Modify GRUB configuration file to persistently disable mtrr, avx, and avx2

```
Update the grub file

$ nano /etc/default/grub

GRUB_CMDLINE_LINUX_DEFAULT="console=ttyS0 clearcpuid=mtrr"

#save and close the file, and run:

$ update-grub

$ reboot
```

Note: If you do not have the latest TDX-Module installed–generally delivered via the system-OEM's BIOS–AVX may not work, and you may encounter errors during boot, or instability during testing. If you notice this, please attempt to modify the code to:

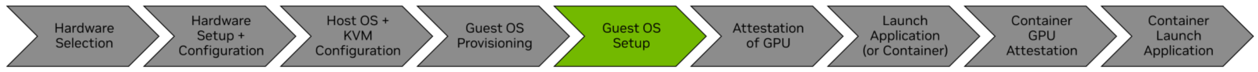
```
linux        /casper/vmlinuz  console=ttyS0 clearcpuid=mtrr,avx,avx2 ---
```

Note: Due to TDX limitations, a reboot command terminates the VM. This is expected for all subsequent reboots.

At this point, the Host OS Administrator persona has completed the required work to enable a Confidential VM with a Confidential H100 attached to it. The next steps will be from the persona of a user who has received access to a VM and is ready to develop or deploy a confidential application.

Virtual Machine Administrator

Figure 8. Virtual Machine Administrator


Hardware Selection
Hardware Setup + Configuration
Host OS + KVM Configuration
Guest OS Provisioning
Guest OS Setup
Attestation of GPU
Launch Application (or Container)
Container GPU Attestation
Container Launch Application

The Virtual Machine Administrator persona assumes that the hardware is correctly configured and expects to receive a CVM that can be attested to, with a GPU attached to it by the hypervisor. This persona might (or might not) have awareness about the lower-level details of the system, such as the BIOS or Host OS configuration. ***Most users will begin their journey here.***

Note: The sample code snippets in this section will be presented as a continuation from the previous steps of this document, which means a clean Ubuntu 22.04 install. If you have been provided a CVM from your System Administrators, you might have a slightly different output, but the overall flow and instructions should not differ greatly.

Log into your CVM.

```
hostUser@host:~$ ssh -p2022 user@localhost
```

Enabling LKCA on the Guest VM

LKCA is required for Hopper CC all operation modes, so we recommend that you enable it in the guest VM.

1. Create a `/etc/modprobe.d/nvidia-lkca.conf` file and add this line to it:

```
install nvidia /sbin/modprobe ecdsa_generic; /sbin/modprobe ecdh; /sbin/modprobe
--ignore-install nvidia
```

2. Update the `initramfs`.

```
sudo update-initramfs -u
sudo reboot
```

Installing the NVIDIA Driver and CUDA Toolkit

We recommend you use the Package Manager method of installing the NVIDIA drivers. OpenRM is the open-source version of our Kernel drivers, and the source can be found on our GitHub.

Hopper CC is enabled starting with CUDA 12.5 and is paired with driver r550, which can be downloaded as described below:

```
# In the Guest:

# Obtain the NVIDIA keys to download the CUDA Toolkit
$ wget
https://developer.download.nvidia.com/compute/cuda/repos/ubuntu2404/x86_64/cuda-keyrin
g_1.1-1_all.deb
$ sudo dpkg -i cuda-keyring_1.1-1_all.deb
$ sudo apt-get update

# Install the toolkit
$ sudo apt-get -y install cuda-toolkit-12-5

# Install the Driver
$ sudo apt install nvidia-driver-550-server-open
```

Setting up the NVIDIA Driver to be in Persistence Mode

When the NVIDIA driver loads, we will automatically establish a secured SPDM session with the H100. As part of this session, secure ephemeral encryption keys are exchanged between the host and the device.

In a typical operation, when the NVIDIA device resources are no longer being used, the NVIDIA kernel driver will tear down the device state. However, in the CC mode, this leads to destroying the shared-secret and the shared keys that were established during the setup SPDM phase of the driver. Due to security concerns, the GPU will not allow the restart of an SPDM session establishment without an FLR which resets and scrubs the GPU.

To avoid this situation, `nvidia-persistenced` provides a configuration option called persistence mode that can be set by NVIDIA management software, such as nvidia-smi. When the persistence mode is enabled, the NVIDIA kernel driver is prevented from exiting.

`nvidia-persistenced` does not use any device resources. It simply sleeps while maintaining a reference to the NVIDIA device state.

1. Determine whether `nvidia-persistenced` is already running.

```
$ ps -aux | grep nvidia-persistenced
nvidia-+     797  0.0  0.0   5472  1852 ?        Ss   17:23   0:00
/usr/bin/nvidia-persistenced --user nvidia-persistenced --no-persistence-mode
--verbose
```

2. If you see the above with `--no-persistence-mode` or the only output is the grep command:

```
$ ps -aux | grep nvidia-persistenced
user       25944  0.0  0.0   4032  2180 pts/0    S+   18:52   0:00 grep
--color=auto nvidia-persistenced
```

   then we must make changes for Confidential Computing modes.

   a. Modify the service that automatically launches `nvidia-persistenced`:

```
# On the Guest:
# Edit /usr/lib/systemd/system/nvidia-persistenced.service

# Change:
    ExecStart=/usr/bin/nvidia-persistenced --user nvidia-persistenced
--no-persistence-mode --verbose

# to this:
    ExecStart=/usr/bin/nvidia-persistenced --user nvidia-persistenced
--uvm-persistence-mode --verbose
```

   b. Tell systemd to reload its modules, and reboot the guest

```
# On the Guest:

$ sudo systemctl daemon-reload
$ sudo reboot
```

Validating State and Versions

1. With the driver in persistent mode, you can check the status of the GPU to ensure that it is configured in a CC mode.

```
$ nvidia-smi conf-compute -f
CC status: ON
```

2. Ensure that the firmware on the H100 is at a minimum version of **96.00.5E.00.00**

```
$ nvidia-smi -q | grep VBIOS
```

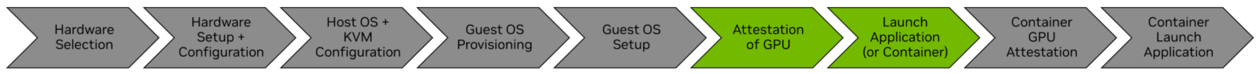
```
    VBIOS Version                              : 96.00.5E.00.25
```

If you have an earlier version of the VBIOS, contact NVIDIA for instructions on how to upgrade to version 96.00.5E.00.00.

You have successfully configured the Guest CVM to operate in the CC mode with a secured H100 accelerator! The next section is the persona for CVM users. However, before this persona can use the GPU, we strongly recommend that you complete the attestation of the GPU.

Virtual Machine User

Figure 9. Virtual Machine Administrator


Hardware Selection
Hardware Setup + Configuration
Host OS + KVM Configuration
Guest OS Provisioning
Guest OS Setup
Attestation of GPU
Launch Application (or Container)
Container GPU Attestation
Container Launch Application

The Virtual Machine user might (or might not) be the administrator of the system (refer to Virtual Machine Administrator for more information). This Persona assumes that the system is correctly configured for CC.

Note: We recommend that you complete your work in the `/shared` folder in the guest VM.

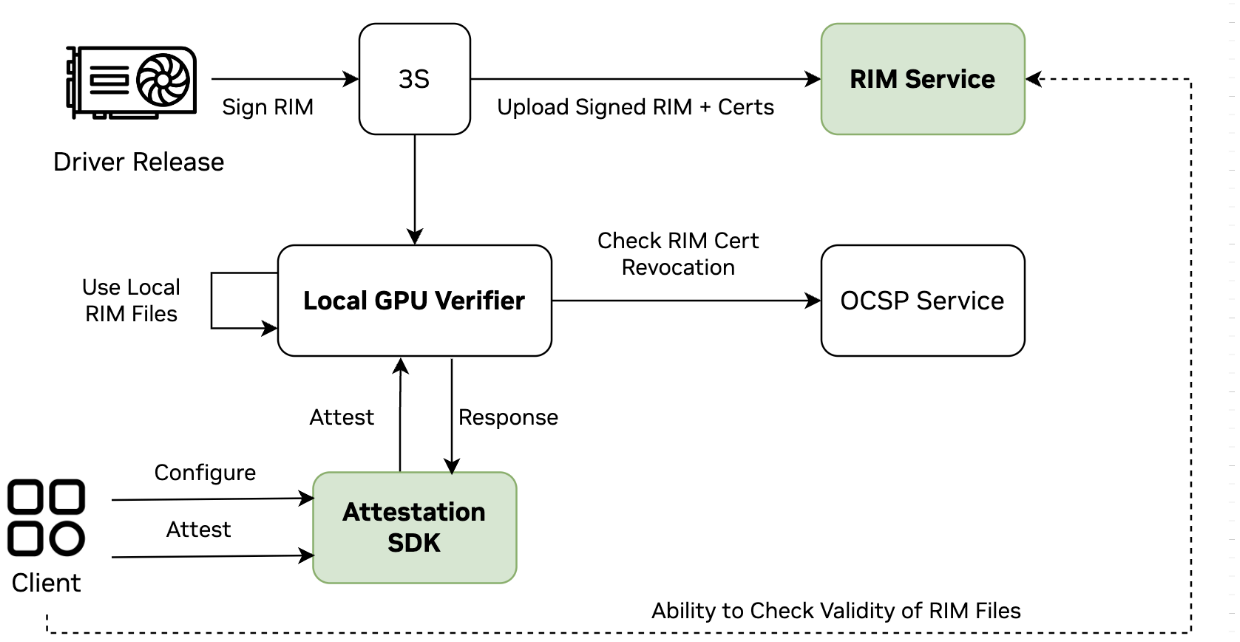
At this point, users must begin the attestation workflow to ensure the system is authentic and valid. Attestation is the process of challenging the GPU where measurements are collected and signed by the GPU, and these measurements are compared to known-good, golden measurements. After the verification passes, you might want to enable the GPU by setting its `ReadyState`.

Note: The GPU will not accept any work until an enlightened CVM user sets the `ReadyState`. This is to prevent accidental usage before the confirmation of the GPU is complete.

Successfully passing attestation as root (see below) or running the command below will set the ready state.

```
nvidia-smi conf-compute -srs 1
```

Figure 10. Attesting the GPU



The recommended flow for attestation is to directly use the Attestation SDK and its APIs . However, you can directly call the Local GPU Verifier. This flow to learn more about the Local GPU Verifier, refer to the *NVIDIA Attestation SDK* guide at https://docs.nvidia.com/nvtrust

Validating Your Configuration

After the driver is successfully installed, and you can query the device, the next step is to attest to the GPU.

1. If you are coming directly to this persona section, ensure that `nvidia-persistenced` is already running. If you started in the previous persona, you skip this verification step.

```
$ ps -aux | grep nvidia-persistenced
 root        2327 20.1  0.0   5312  1788 ?        Ss   08:57   0:05
nvidia-persistenced
```

2. If nothing is returned, run the following command to start it.
   ```
   $ sudo nvidia-persistenced
   ```

3. Check the status of the GPU to ensure that it is configured in a CC mode.
   ```
   $ nvidia-smi conf-compute -f
   CC status: ON
   ```

Installing the Attestation SDK

Before you begin, install the Local GPU Verifier, which is also in the nvTrust repository. To keep it simple, perform another clone of the repo:

```
$ sudo mkdir /shared
$ sudo chmod 777 /shared

$ cd /shared
$ git clone https://github.com/nvidia/nvtrust

$ cd nvtrust/guest_tools/
```

Installation Prerequisites

The Attestation SDK and the Local GPU Verifier require Python3. We also recommend that you also install the Virtual Environment module, which can keep your primary system Python environment clean.

```
$ sudo apt install python3-pip

# Install:
$ sudo apt install python3-venv
```

(Optional) Configuring a Python Virtual Environment

```
# Create a new virtual env named nvAttest
python3 -m venv /shared/nvAttest

# Configure the shell to use nvAttest
$ source /shared/nvAttest/bin/activate
(nvAttest) user@guestVM:/shared/$
```

Your Python virtual environment will now always be prefixed with `(nvAttest)`. If you do not see this string on your terminal (for example, after changing terminal windows, logging out, and so on), run the following command again.

```
$ source /shared/nvAttest/bin/activate
```

Installing the Local GPU Verifier

You must install the plugins **before** you install the Attestation SDK, otherwise you will get errors.

```
(nvAttest) $ cd /shared/nvtrust/guest_tools/gpu_verifiers/local_gpu_verifier
(nvAttest) $ pip3 install .
```

Installing the Attestation SDK

Note: Ensure you are running in the same python environment (either the optional virtual environment nvAttest created above or your default one).

```
(nvAttest) $ cd /shared/nvtrust/guest_tools/attestation_sdk/
(nvAttest) $ pip3 install .
```

Executing an Attestation of the GPU

After the components have been installed, you are ready to perform an attestation using the SDK. The sample code can be found on the nvTrust GitHub under `nvtrust/guest_tools/attestation_sdk/tests/SmallGPUTest.py`
However, here is the Python code, and you can run this code on the `python3` command line.

```
from nv_attestation_sdk import attestation

# Create a Attestation object
client = attestation.Attestation("test_node")

# Add the type of verifier that you would like to use
client.add_verifier(attestation.Devices.GPU, attestation.Environment.LOCAL, "", "")

# Set the Attestation Policy that you want to validate your token against.
attestation_results_policy =
'{"version":"1.0","authorization-rules":{"x-nv-gpu-available":true,' \
                         '"x-nv-gpu-attestation-report-available":true}}'

# Run Attest
print(client.attest())

# Call validate_token to validate the results against the Appraisal policy for
Attestation Results
print(client.validate_token(attestation_results_policy))
```

The primary focus of the attestation should be the yellow highlighted variable. As the developer, you can decide which claims constitute a pass or a fail result from the Attestation SDK. In the example above, the code will return `TRUE` as long as there is an NVIDIA GPU detected in the CVM and the process to obtain the GPU measurements was returned properly.

We provide a full list of all the possible claims that returned during an attestation query. It is listed below for your reference.

```
# /shared/nvtrust/guest_tools/attestation_sdk/tests/NVGPUPolicyExample.json

{

    "version":"1.0",

    "authorization-rules":{

      "x-nv-gpu-available":true,

      "x-nv-gpu-attestation-report-available":true,

      "x-nv-gpu-info-fetched":true,

      "x-nv-gpu-arch-check":true,

      "x-nv-gpu-root-cert-available":true,

      "x-nv-gpu-cert-chain-verified":true,

      "x-nv-gpu-ocsp-cert-chain-verified":true,

      "x-nv-gpu-ocsp-signature-verified":true,

      "x-nv-gpu-cert-ocsp-nonce-match":true,

      "x-nv-gpu-cert-check-complete":true,

      "x-nv-gpu-measurement-available":true,

      "x-nv-gpu-attestation-report-parsed":true,

      "x-nv-gpu-nonce-match":true,

      "x-nv-gpu-attestation-report-driver-version-match":true,

      "x-nv-gpu-attestation-report-vbios-version-match":true,

      "x-nv-gpu-attestation-report-verified":true,

      "x-nv-gpu-driver-rim-schema-fetched":true,

      "x-nv-gpu-driver-rim-schema-validated":true,

      "x-nv-gpu-driver-rim-cert-extracted":true,

      "x-nv-gpu-driver-rim-signature-verified":true,

      "x-nv-gpu-driver-rim-driver-measurements-available":true,

      "x-nv-gpu-driver-vbios-rim-fetched":true,

      "x-nv-gpu-vbios-rim-schema-validated":true,

      "x-nv-gpu-vbios-rim-cert-extracted":true,
      "x-nv-gpu-vbios-rim-signature-verified":true,
      "x-nv-gpu-vbios-rim-driver-measurements-available":true,
      "x-nv-gpu-vbios-index-conflict":true,
      "x-nv-gpu-measurements-match":true
    }
  }
```

Successful Attestation Result

When the Attestation SDK has successfully returned a valid result, you should see something like below (varies slightly based on your specific system):

```
python3
Python 3.10.6 (main, May 29 2023, 11:10:38) [GCC 11.3.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> from nv_attestation_sdk import attestation
>>> client = attestation.Attestation("test_node")
>>> client.add_verifier(attestation.Devices.GPU, attestation.Environment.LOCAL, "",
"")
>>> attestation_results_policy =
'{"version":"1.0","authorization-rules":{"x-nv-gpu-available":true,' \
...
'"x-nv-gpu-attestation-report-available":true,"x-nv-gpu-info-fetched":true,' \
...
'"x-nv-gpu-arch-check":true,"x-nv-gpu-root-cert-available":true,' \
...
'"x-nv-gpu-cert-chain-verified":true,"x-nv-gpu-ocsp-cert-chain-verified":true,' \
...
'"x-nv-gpu-ocsp-signature-verified":true,"x-nv-gpu-cert-ocsp-nonce-match":true,' \
...
'"x-nv-gpu-cert-check-complete":true,"x-nv-gpu-measurement-available":true,' \
...
'"x-nv-gpu-attestation-report-parsed":true,"x-nv-gpu-nonce-match":true,' \
...
'"x-nv-gpu-attestation-report-driver-version-match":true,' \
...
'"x-nv-gpu-attestation-report-vbios-version-match":true,' \
...
'"x-nv-gpu-attestation-report-verified":true,"x-nv-gpu-driver-rim-schema-fetched":true
,' \
...
'"x-nv-gpu-driver-rim-schema-validated":true,"x-nv-gpu-driver-rim-cert-extracted":true
,' \
...                              '"x-nv-gpu-driver-rim-signature-verified":true,' \
...
'"x-nv-gpu-driver-rim-driver-measurements-available":true,' \
...
'"x-nv-gpu-driver-vbios-rim-fetched":true,"x-nv-gpu-vbios-rim-schema-validated":true,'
\
...
'"x-nv-gpu-vbios-rim-cert-extracted":true,"x-nv-gpu-vbios-rim-signature-verified":true
,' \
...
'"x-nv-gpu-vbios-rim-driver-measurements-available":true,' \
...
'"x-nv-gpu-vbios-index-conflict":true,"x-nv-gpu-measurements-match":true}}'
>>> client.attest()
Number of GPUs available : 1
-----------------------------------
Fetching GPU 0 information from GPU driver.
VERIFYING GPU : 0
    Driver version fetched : 535.86.05
    VBIOS version fetched : 96.00.5e.00.01
    Validating GPU certificate chains.
        GPU attestation report certificate chain validation successful.
            The certificate chain revocation status verification successful.
    Authenticating attestation report
        The nonce in the SPDM GET MEASUREMENT request message is matching with the
generated nonce.
        Driver version fetched from the attestation report : 535.86.05
        VBIOS version fetched from the attestation report : 96.00.5e.00.01
        Attestation report signature verification successful.
        Attestation report verification successful.
    Authenticating the RIMs.
        Authenticating Driver RIM
            RIM Schema validation passed.
            driver RIM certificate chain verification successful.
            The certificate chain revocation status verification successful.
            driver RIM signature verification successful.
            Driver RIM verification successful
        Authenticating VBIOS RIM.
            RIM Schema validation passed.
            vbios RIM certificate chain verification successful.
            The certificate chain revocation status verification successful.
            vbios RIM signature verification successful.
            VBIOS RIM verification successful
    Comparing measurements (runtime vs golden)
            The runtime measurements are matching with the golden measurements.
        GPU is in expected state.
    GPU 0 verified successfully.
    GPU Attested Successfully
True
>>> client.validate_token(attestation_results_policy)
```

Conclusion

With this guide, we have provided information about the process from when the machine is racked and stacked to configuring the host and guest operating systems, and finally to attaching an H100 in a CVM. This flow can be modified to suit your specific needs, and we encourage you to provide feedback, comments, or questions.. You can reach out to us on our forum page:
https://forums.developer.nvidia.com/c/accelerated-computing/confidential-computing/663
.
Stay tuned to our GitHub for the latest updates, news, and solutions in the meantime.
Happy coding!