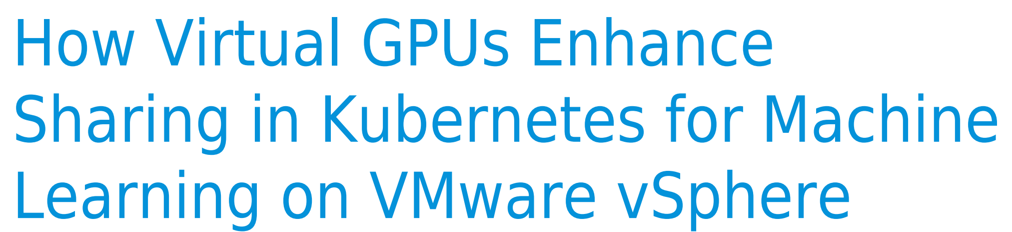


How Virtual GPUs Enhance Sharing in Kubernetes for Machine Learning on VMware vSphere

VMware AI/ML

vmware by Broadcom © VMware LLC.

How Virtual GPUs Enhance Sharing in Kubernetes for Machine Learning on VMware vSphere

Outline - Why is this Important?

This article explores how the Kubernetes cluster scheduler becomes aware of GPUs and virtual GPUs on its nodes. We discuss this particularly in the context of VMware vSphere with Tanzu. This GPU acceleration area is very important for data scientists so that they can rapidly execute their machine learning model training and inference phases and iterate more quickly in the development/deployment cycle. The material here is of interest also to data center and Kubernetes platform administrators as they need to assess the impact of GPUs on their clusters - and understand the utilization of GPUs across different clusters. Here is a summary of the core ideas we will look at:

- A Kubernetes pod running on a bare metal server that needs a GPU, has exclusive access to that GPU
- This means that on physical machines, the entire GPU is dedicated to that one pod – this can be wasteful of GPU power
- vGPUs solve this issue by allowing a virtual GPU (vGPU) to represent part of a physical GPU to the Kubernetes pod scheduler. This optimizes the use of the GPU hardware and it can serve more than one user, reducing costs.

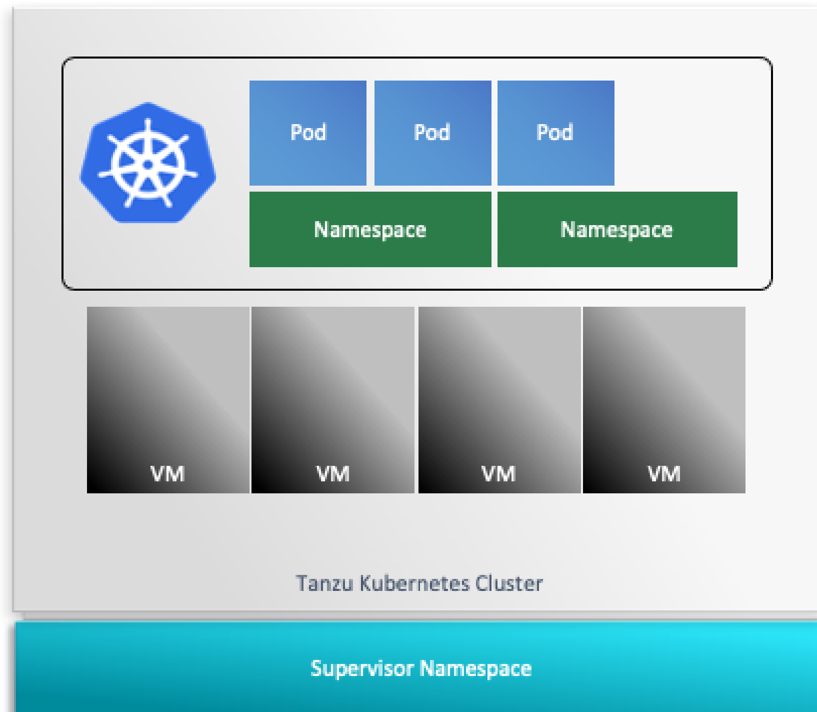
A basic level of familiarity with the core concepts in Kubernetes and in GPU Acceleration will be useful to the reader of this article. We first look more closely at pods in Kubernetes and how they relate to a GPU.

Pods in Kubernetes

A pod is the unit of deployment, at the lowest level, in Kubernetes. A pod can have one or more containers within it. The lifetime of the containers within a pod tend to be about the same, although one container may start before the others, as the “init” container. You can deploy higher-level objects like Kubernetes services and deployments that have many pods in them. We focus on pods and their use of GPUs in this article. Given access rights to a Tanzu Kubernetes cluster (TKC) running on the VMware vSphere with Tanzu environment (i.e. a set of host servers running the ESXi hypervisor, managed by VMware vCenter), a user can issue the command:

```
kubectl apply -f <pod-specification-file-name>.yaml
```

to deploy a new pod for an ML application, for example, into that cluster. We will see an example of such a pod shortly. Here is a set of Kubernetes pods running on VMs that implement Kubernetes nodes on VMware vSphere with Tanzu.



A pod is scheduled to run on a node by the Kubernetes scheduler. We use the terms "node" and "VM" to mean the same thing in vSphere with Tanzu. The Kubernetes scheduler for each TKC runs in a TKC Control Plane node. If it is successfully scheduled, the pod then runs, by starting its containers, in one of the suitable TKC worker nodes, i.e., one of the VMs that support that Tanzu Kubernetes Cluster on vSphere.

In a Kubernetes cluster, it is common to see many pods running at once. Even small clusters can have 60 or more pods running on them. Pods can come and go and some live on for long periods - for many months in the case of some infrastructure pods, as we will see. Here is a subset of the pods running in one of our TKCs in the labs.

```
$ k get pods -n gpu-operator
NAME                                                          READY   STATUS      RESTARTS   AGE
gpu-feature-discovery-f2tm4                                   1/1     Running     0          6d
gpu-operator-f8bc4c8f8-774fc                                  1/1     Running     0          10d
gpu-operator-node-feature-discovery-master-58d884d5cc-zzszm   1/1     Running     0          10d
gpu-operator-node-feature-discovery-worker-49vfv              1/1     Running     0          10d
gpu-operator-node-feature-discovery-worker-5qgfd              1/1     Running     0          6d
gpu-operator-node-feature-discovery-worker-qgs24              1/1     Running     0          10d
gpu-operator-node-feature-discovery-worker-rcwb7              1/1     Running     0          10d
gpu-operator-node-feature-discovery-worker-tdhbn              1/1     Running     0          10d
nvidia-container-toolkit-daemonset-tzkt6                      1/1     Running     0          6d
nvidia-cuda-validator-9jcb2                                   0/1     Completed   0          6d
nvidia-dcgm-exporter-xm7lb                                    1/1     Running     0          6d
nvidia-device-plugin-daemonset-t6h9q                          1/1     Running     0          6d
nvidia-device-plugin-validator-7tcln                          0/1     Completed   0          6d
nvidia-driver-daemonset-d7nvd                                 1/1     Running     0          6d
nvidia-mig-manager-d6xdf                                      1/1     Running     0          6d
nvidia-operator-validator-gl2pt                               1/1     Running     0          6d
```

The command

“kubectl get pods -A”

gives a view of how many pods are running in your Tanzu Kubernetes cluster.

The example above shows the pods running in just one TKC namespace – in this case, the "gpu-operator" namespace - within the

 © VMware LLC.

cluster. There can be many more namespaces and pods within a TKC, and there can also be many TKCs running concurrently on your collection of vSphere servers. Of the potentially large number of running pods across your TKCs, only a subset of them will need to access a GPU for machine learning application acceleration. In the discussion below, we delve into how these pods may use and share the available physical GPUs on your servers.

There can be several data science users/ML developers in your organization, each executing in their own individual TKC cluster, or they can manage separate deployments in the same cluster. These users are very likely to need a GPU at the same time. In many enterprises, more than one machine learning application will be under development, testing or deployment at one time.

When a Tanzu Kubernetes Cluster (TKC) is running on vSphere, a pod may be using a vGPU (a virtualized GPU), where this vGPU represents a part of, or all of, a physical GPU. The questions we want to answer here are:

- how are pods scheduled onto those nodes that have a GPU/vGPU associated with them?

- can two pods in different nodes on a server share one physical GPU?

Scheduling Pods in Kubernetes that Need a GPU

Firstly, when creating a pod using the kubectl command, we declare that the pod requires a GPU by means of a special section in the pod specification in YAML. An example is given below.

```
apiVersion: v1

kind: Pod

metadata:

  name: tf-notebook

  labels:

    app: tf-notebook

spec:

  containers:

  - name: tf-notebook

    image: tensorflow/tensorflow:latest-gpu-jupyter

    resources:

      limits:

        nvidia.com/gpu: 1
```

Notice the section titled "limits" above. It specifies that this pod (really a container within it) needs a single GPU to work with. It may require more than one GPU, so that number would be higher in the YAML spec. This pod will start successfully only if there is an available node in the cluster with a GPU or virtual GPU associated with it – and that node is not already fulfilling a GPU request for another pod, i.e. the GPU is available for use at this time.

The identification of which nodes have GPUs is done using specific labels. But how do Kubernetes nodes get the GPU-specific labels assigned to them in the first place? These labels are not there when the VM representing the node starts up. That labeling is the job of two kinds of pods that are part of the NVIDIA GPU Operator.

The Role of the NVIDIA GPU Operator

On vSphere with Tanzu, you deploy the NVIDIA GPU Operator into a TKC (using Helm) to take care of the NVIDIA vGPU guest driver installation, the device plugin installation and several other functions, along with the lifecycle management of all the pods in its care. The GPU Operator installs the vGPU driver into a container that is run as part of one or more NVIDIA "driver daemonset" pods. In order to create a driver daemonset pod on the correct GPU-capable node, the GPU Operator must first discover the GPU capabilities of each node (i.e. whether there even exists a GPU on each node). To do this, when the GPU Operator is installing, it starts a Node Feature Discovery (NFD) worker pod on every node in your Tanzu Kubernetes Cluster.

The NVIDIA GPU Operator also starts one or more GPU Feature Discovery (GFD) pods. The NFD Worker pod probes the node it is

running on and if it finds a GPU device on there, then it applies the label to the node as follows:

```
feature.node.kubernetes.io/pci-10de.present=true
```

The code "pci-10de" is the NVIDIA vendor ID in the PCI standard. A GPU Feature Discovery pod is started on any node that obeys the following condition:

```
nodeSelector:

        feature.node.kubernetes.io/pci-10de.present: "true"
```

The NFD and GFD pods therefore cooperate together so that a set of labels is applied to the TKC node, as follows:

```
nvidia.com/gpu.present=true

nvidia.com/gpu.count=1

nvidia.com/vgpu.present=true
```

After the GPU Operator has been deployed, you can see those labels on a TKC node which has a vGPU associated with it by using:

kubectl describe node <nodename>

For each VM or Kubernetes node in a TKC, we also see the fields referring to the GPU allocation by using the above command. Two areas of the output from that command are shown here:

```
Allocatable:

  nvidia.com/gpu:     1


Allocated resources:

  Resource            Requests      Limits
  --------            --------      ------
  nvidia.com/gpu      0             0
```

Once a pod is scheduled on to a VM that has a vGPU, the Requests and Limits fields for “nvidia.com/gpu” will be incremented in the “Allocated Resources” section above. Kubectl passes the above information to the Kubernetes scheduler so that the scheduler knows exactly the inventory state of the cluster for further GPU to pod allocation decisions.

You can read more about the Node Feature Discovery and GPU Feature Discovery pods that are used by the NVIDIA GPU Operator here

If the single vGPU on our TKC node is already taken by a previously running GPU-aware pod, then another new pod that requires a vGPU will not start.

How vGPU-based Sharing Can Help with Pods

Each vGPU may represent a share of a physical GPU, i.e., part of its cores and framebuffer memory, to a VM. Alternatively, a vGPU may also represent ALL of a physical GPU if you want to dedicate the full physical GPU to that VM. These options are chosen by the user, either in the vSphere Client (for non-TKC VMs) or at VM Class creation time (for use in TKCs), using pre-ordained vGPU profiles. Those profiles express how much of the physical GPU is to be assigned. Often, framebuffer memory sizing is the main concern here. For more on that subject, look at this article. You can see the vGPU profile associated with a VM by highlighting the VM in the vSphere Client and using “Edit Settings” on it. We will see examples of doing this below.

To address the pod-to-GPU mapping question, a vGPU profile that represents **part of the shared physical GPU** to one or more Kubernetes nodes can be used to fulfill the need of a pod. **This is one of the core values of virtualizing your GPU in the Kubernetes world.**

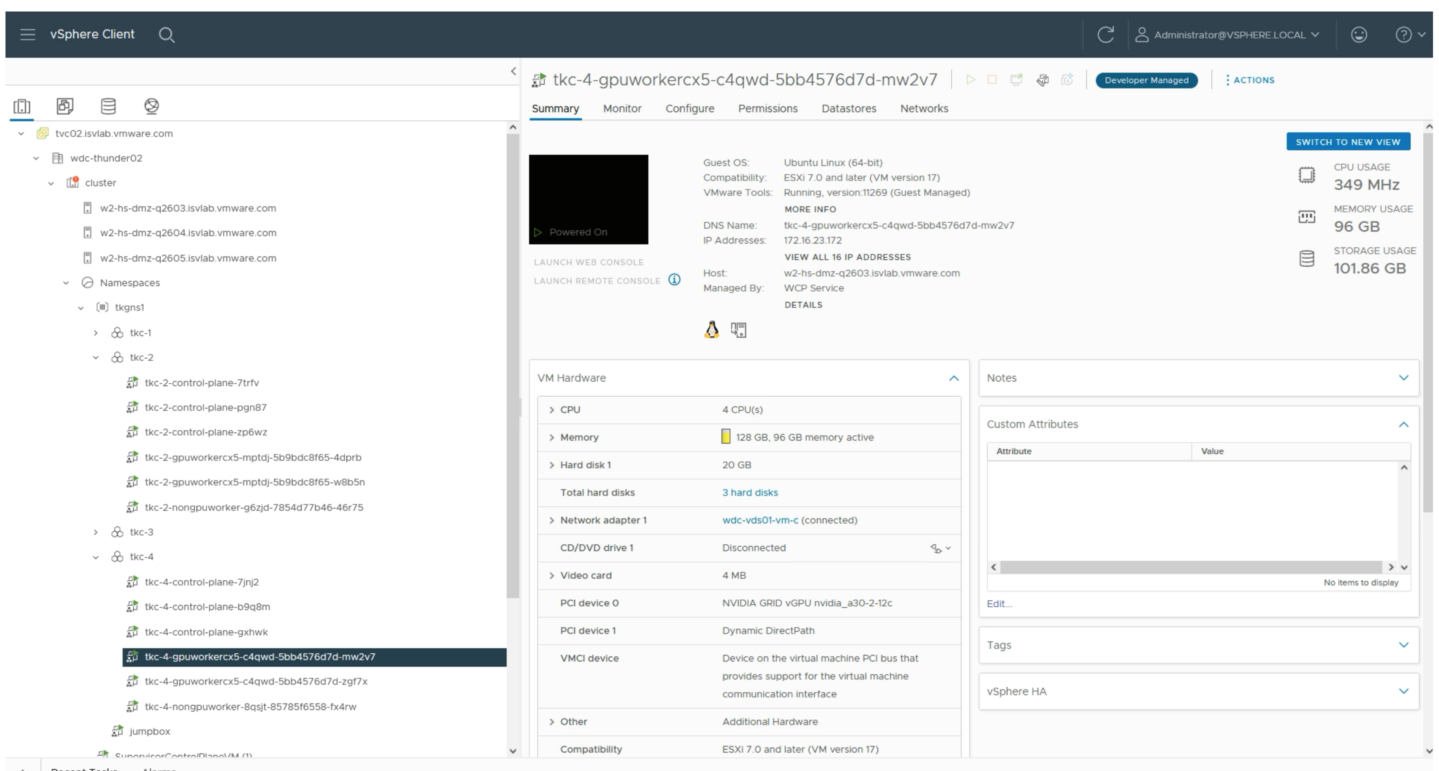
A vGPU on a node (a VM) is enough to satify the “nvidia.com/gpu: 1” requirement of the pod specification above. So now a pod on

a second, separate VM (even if it is part of a separate TKC) can share one physical GPU with the pod belonging to your first VM. The two pods may be in separate application workloads living in different VMs that run on the same server. The pods can now execute against the same physical GPU, concurrently. The device plugin pod decreases by 1 the number of vGPUs available on the node where the pod is scheduled to.

Below, we see an example deployment with two VMs in two separate TKCs, named tkc-2 and tkc-4, serving different data science needs. There are other TKCs running here too, tkc-1 and tkc-3, that share the same hardware cluster. We focus in on tkc-2 and tkc-4 here.

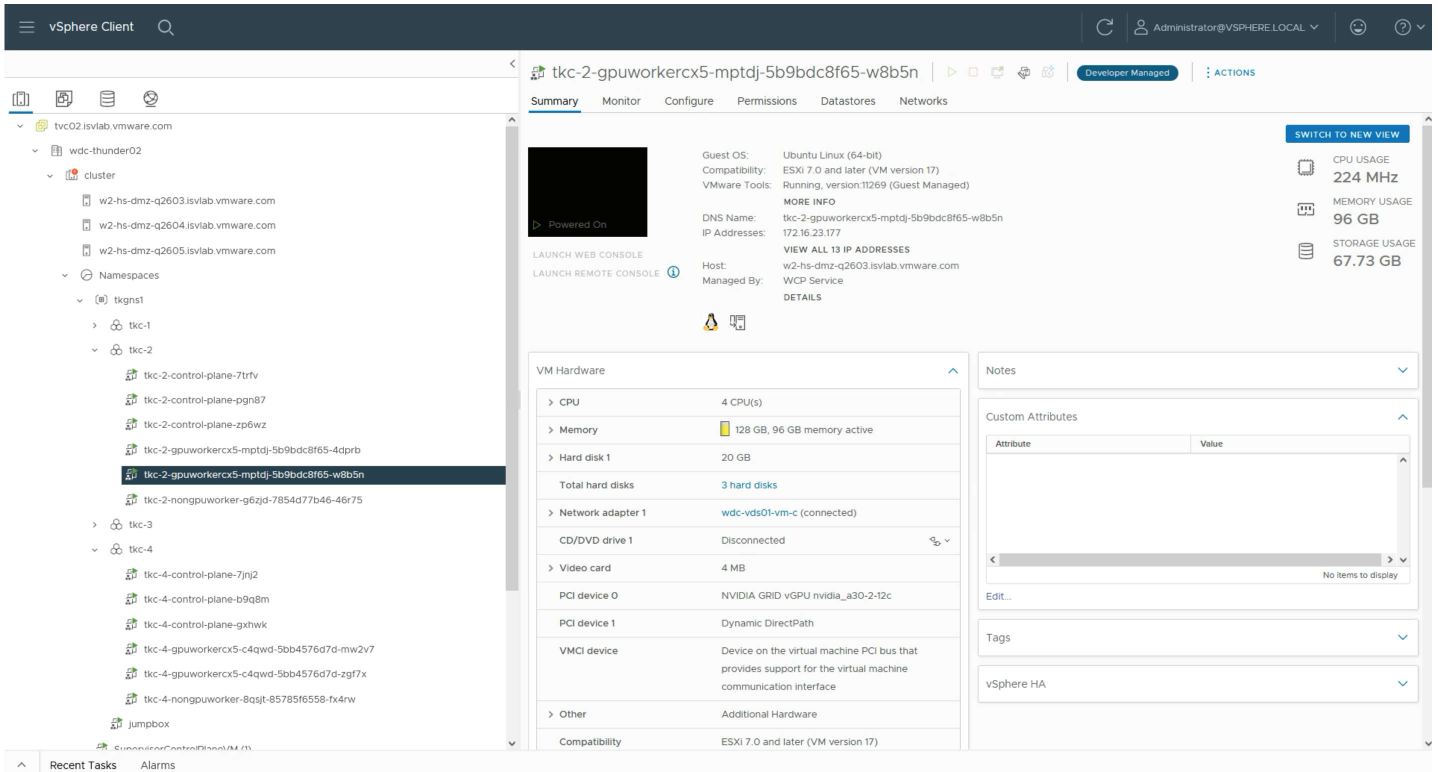
The VMs that make up each TKC below, each a Kubernetes node, have green arrow icons, indicating that they are running. Each of the the gpuworker VMs (one is highlighted) hosts a pod that requires a GPU in the manner described in the limits section of the YAML specification above.

Each of the two VMs has a vGPU profile named “nvidia_a30_2_12c”, giving it half the cores and half the framebuffer memory of an A30 GPU. The A30 GPU has 24 GB of framebuffer memory. You can see the vGPU profile at the “PCI Device 0” entry below. Our first VM, whose name ends in “mw2v7” is running on host server “w2-hs-dmz-q2603.isvlab.vmware.com”, as we see in the “Host” section below.



In the next image from the vSphere Client below, we see the same vGPU profile on our second example VM whose name ends in “w8b5n”. This VM is also running on host “w2-hs-dmz-q2603” – the same host server as the first VM. This tells us that we can have nodes from separate TKC clusters sharing the same server hardware.

 © VMware LLC.



The two VMs of interest here - both of them Kubernetes nodes - are sharing a physical GPU. We confirm that they share the same physical GPU by looking at the last two lines of the **nvidia-smi** command output that is executed on the physical ESXi host server, w2-hs-dmz-q2603, below. To see this view across all VMs on this one host server, we logged into that ESXi server to issue the nvidia-smi command. There are two A30 GPUs on that server and the first one, GPU zero is fully occupied by the process whose name ends in "cwsdp". Let's set that GPU to one side for now - it belongs to another cluster that we are not examining. We can see from the Processes section here that our two VMs from the separate TKCs are running on GPU #1 on this host.

© VMware LLC.

```
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 470.82       Driver Version: 470.82       CUDA Version: N/A      |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|                               |                      |               MIG M. |
|===============================+======================+======================|
|   0  NVIDIA A30          On   | 00000000:81:00.0 Off |                   On |
| N/A   33C    P0    67W / 165W |  24064MiB / 24258MiB |     N/A      Default |
|                               |                      |              Enabled |
+-------------------------------+----------------------+----------------------+
|   1  NVIDIA A30          On   | 00000000:E2:00.0 Off |                   On |
| N/A   33C    P0    67W / 165W |  23808MiB / 24258MiB |     N/A      Default |
|                               |                      |              Enabled |
+-------------------------------+----------------------+----------------------+

+-----------------------------------------------------------------------------+
| MIG devices:                                                                |
+------------------+----------------------+-----------+-----------------------+
| GPU  GI  CI  MIG |         Memory-Usage |        Vol|         Shared        |
|      ID  ID  Dev |           BAR1-Usage | SM     Unc| CE  ENC  DEC  OFA  JPG|
|                  |                      |        ECC|                       |
|==================+======================+===========+=======================|
|  0    0   0   0  |  24064MiB / 24258MiB | 56      0 |  4   0    4    1    1 |
|                  |      1MiB / 32768MiB |           |                       |
+------------------+----------------------+-----------+-----------------------+
|  1    1   0   0  |  11904MiB / 12096MiB | 28      0 |  2   0    2    0    0 |
|                  |      0MiB / 16383MiB |           |                       |
+------------------+----------------------+-----------+-----------------------+
|  1    2   0   1  |  11904MiB / 12096MiB | 28      0 |  2   0    2    0    0 |
|                  |      0MiB / 16383MiB |           |                       |
+------------------+----------------------+-----------+-----------------------+

+-----------------------------------------------------------------------------+
| Processes:                                                                  |
|  GPU   GI   CI        PID   Type   Process name                  GPU Memory |
|        ID   ID                                                   Usage      |
|=============================================================================|
|    0    0    0   13021334    C+G   ...er-qfjd4-6fdbb9cb97-cwsdp    24064MiB |
|    1    2    0    8908935    C+G   ...x5-c4qwd-5bb4576d7d-mw2v7    11904MiB |
|    1    1    0   11841538    C+G   ...x5-mptdj-5b9bdc8f65-w8b5n    11904MiB |
+-----------------------------------------------------------------------------+
```

The test described above shows that the GPU required by a pod in its spec is fulfilled by a vGPU profile on a node. ***Using vGPUs, therefore, we do not have to dedicate a full physical GPU to one pod.***

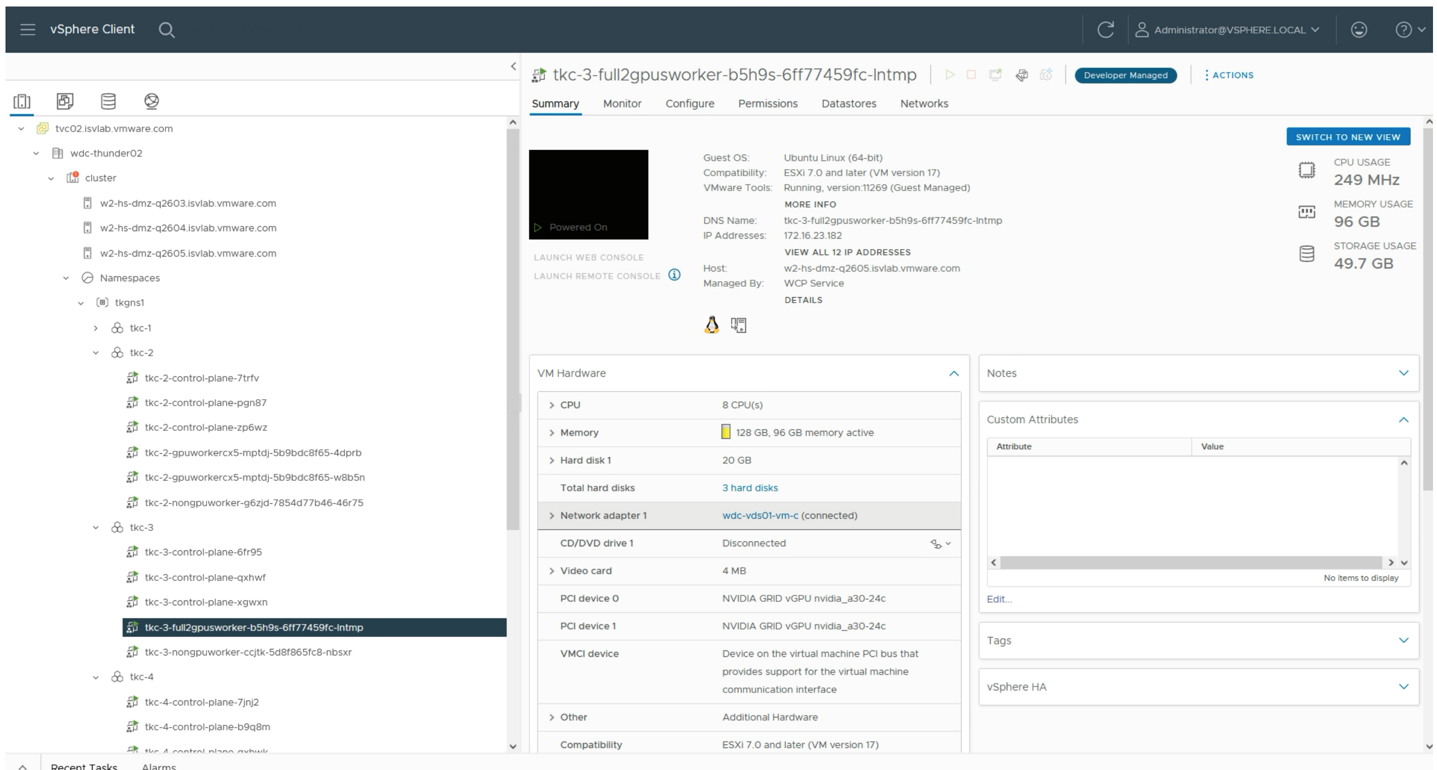
These VMs have a vGPU profile that gives them half of the physical GPU each. We can use smaller-sized vGPU profiles, and run more VMs on this host in the same way. This is the case where the vGPU profile is a subset of the physical GPU's cores (in the MIG case) and framebuffer memory.

On the other hand, a vGPU profile may be set up to represent ALL of one physical GPU to a VM. One full GPU may be required by the application, if the ML model fully occupies the framebuffer memory of that GPU, for example. In that case, one pod in that application will consume the VM's full vGPU profile and with it the full physical GPU. For those VMs that can access more than one vGPU, and therefore more than one full physical GPU, we follow a similar approach.

 © VMware LLC.

Multiple vGPUs on One Kubernetes Node (VM)

If you have two physical GPUs on one ESXi host server, then you can use two vGPU profiles to dedicate those two physical GPUs fully to one VM, provided the GPUs are in time-sliced mode and are fully allocated (i.e. no subsets of memory/cores are allowed). You may do this to accommodate a machine learning model that is too large to fit into one physical GPUs memory, for example, and you want all of the GPU power assigned to one training exercise.



With two or more such full-profile, time-sliced vGPUs associated with a VM, as shown above at the PCI device 0 and 1 lines above, then different applications can run within the same VM and use the individual vGPUs for pods independently of each other. You can do this by using a full profile vGPU like "nvidia-a30-24c" on an A30 GPU or "a100-40c" on an A100 GPU.

Those two vGPU profiles indicate that all of the memory on the GPU hardware is assigned to the VM. These profiles can be assigned to VMs of a custom VM Class in vSphere with Tanzu as discussed in a previous article. This design appears to the host server as one VM consuming both full GPUs as seen below. Note that the names of the two running processes on the GPU map to one name, the VM name ending in “lntmp” in this case, as we saw in the vSphere Client above.

```
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 470.82       Driver Version: 470.82       CUDA Version: N/A      |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|                               |                      |               MIG M. |
|===============================+======================+======================|
|   0  NVIDIA A30          On   | 00000000:81:00.0 Off |                    0 |
| N/A   25C    P0    28W / 165W |  23936MiB / 24258MiB |      0%      Default |
|                               |                      |             Disabled |
+-------------------------------+----------------------+----------------------+
|   1  NVIDIA A30          On   | 00000000:E2:00.0 Off |                    0 |
| N/A   26C    P0    31W / 165W |  23936MiB / 24258MiB |      0%      Default |
|                               |                      |             Disabled |
+-------------------------------+----------------------+----------------------+

+-----------------------------------------------------------------------------+
| Processes:                                                                  |
|  GPU   GI   CI        PID   Type   Process name                  GPU Memory |
|        ID   ID                                                   Usage      |
|=============================================================================|
|    0   N/A  N/A   6986246    C+G   ...er-b5h9s-6ff77459fc-lntmp    23936MiB |
|    1   N/A  N/A   6986246    C+G   ...er-b5h9s-6ff77459fc-lntmp    23936MiB |
+-----------------------------------------------------------------------------+
```

Two Full Profile vGPU profiles for two A30 GPUs are in use here by one VM (lntmp) on one host server

We can have up to four full physical GPUs represented on one VM using the correct vGPU profiles in this way. That number of concurrent full GPUs on a single VM may increase in future versions of vSphere. This multi-GPU approach on one VM is used for handling very large models that need the space, in terms of the GPU framebuffer memory, from more than one physical GPU.

Implications for Node/VM Design

Examine how your ML application pods will use a GPU. If the pod can execute well on a share of a GPU, such as one half, one seventh or another fraction of a physical GPU, then that pod would run in a VM alongside other non-GPU consuming pods with the appropriate vGPU profile assigned to it.

Pods from separate TKC clusters and VMs can share a single physical GPU if they are set up to use only a subset of that GPU. The particular subset is assigned via a vGPU profile that is assigned to the node in which that pod is running.

If your pod requires access to a full GPU and there is only one physical GPU available to the pod's VM on the host, then that pod will have exclusive use of that GPU, via its full memory vGPU profile.

If two application pods are required to be run together on the same node (i.e. VM) , and both need access to a GPU simultaneously, then you will need two physical GPUs assigned to that VM. This can be done using full-memory, time-sliced vGPU profiles. This is the only case in which a VM can have multiple vGPUs with the same profile - at the time of this writing. A VM can have up to four full memory profile vGPUs in this way and can therefore host four GPU-consuming pods at once.

Conclusions

- Many application and infrastructure pods can run concurrently in a Kubernetes cluster. Some of those pods require access to a GPU for ML model training and inferencing acceleration.
- A K8S pod running on bare metal that needs a GPU has exclusive access to that GPU This means that on physical machines, the entire GPU is dedicated to that one pod – which can be wasteful
- Virtual GPUs on vSphere solve this issue by allowing a virtual GPU (vGPU) to represent part of a physical GPU to the Kubernetes pod scheduler and thereby fullfilling the need of more than one pod accessing a physical GPU in parallel. This can clearly save significant costs for these devices in the modern AI-driven data center.

© VMware LLC.

References

Accelerating Workloads on vSphere 7 with Tanzu - a Technical Preview of Kubernetes Clusters with GPUs

Determining GPU Memory for Machine Learning Application on VMware vSphere with Tanzu

NVIDIA AI Enterprise Documentation Site

NVIDIA AI Enterprise - Deploy the GPU Operator

Deploy an AI-Ready Enterprise Platform on VMware vSphere 7 with VMware Tanzu Kubernetes Grid Service

vSphere 7 Update 2 vGPU Operations Guide

Sizing Guidance for AI/ML in VMware Environments

Using GPUs with Virtual Machines on vSphere - Part 3: Installing the NVIDIA Virtual GPU Technology

vSphere 7 with Multi-Instance GPUs (MIG) on the NVIDIA A100 for Machine Learning Applications

 © VMware LLC.

vmware by Broadcom © VMware LLC.

Copyright © 2005-2024 Broadcom. All Rights Reserved. The term “Broadcom” refers to Broadcom Inc. and/or its subsidiaries.