Services for Machine Learning applications (part 2 of 3)

Diana Gaponcic, IT-CD-PI

GPU clusters at CERN

How to create a GPU cluster

```
$ openstack coe cluster create digaponc-gpu-004 --merge-labels --labels nvidia_gpu_enabled=true
```

How to create a GPU cluster

1. By default 2 nodes are deployed: the master and the default worker node

How to create a GPU cluster

1. By default 2 nodes are deployed: the master and the default worker node
2. **No GPU yet**
   a. the cluster is configured to manage GPUs, but we don't get a GPU by default

GPU flavors

Flavor Name	GPU	RAM	vCPUs	Disk	Ephemeral	Comments
g1.xlarge	V100	16 GB	4	56 GB	96 GB	[^1], deprecated
g1.4xlarge	V100 (4x)	64 GB	16	80 GB	528 GB	[^1]
g2.xlarge	T4	16 GB	4	64 GB	192 GB	[^1], deprecated
g2.5xlarge	T4	168 GB	28	160 GB	1200 GB	[^1]
g3.xlarge	V100S	16 GB	4	64 GB	192 GB	[^1]
g3.4xlarge	V100S (4x)	64 GB	16	128 GB	896 GB	[^1]
g4.p1.40g	A100 (1x)	120 GB	16	600 GB	-	[^1], AMD CPUs
g4.p2.40g	A100 (2x)	240 GB	32	1200 GB	-	[^1], AMD CPUs
g4.p4.40g	A100 (4x)	480 GB	64	2400 GB	-	[^1], AMD CPUs

Consult https://clouddocs.web.cern.ch/gpu_overview.html for an up-to-date list of GPU flavors

Add a GPU node

```
$ openstack coe nodegroup create digaponc-gpu-004 gpu-t4 --flavor g2.5xlarge --node-count 1

...

$ kubectl get no
NAME                                          STATUS   ROLES    AGE   VERSION
digaponc-gpu-004-6zombv4qhhxi-master-0        Ready    master   17d   v1.30.2
digaponc-gpu-004-6zombv4qhhxi-node-0          Ready    <none>   17d   v1.30.2
digaponc-gpu-004-gpu-t4-rr5badjdpuyc-node-0   Ready    <none>   17d   v1.30.2
```

NVIDIA GPU operator

```
$ kubectl get pod -n kube-system | grep nvidia
nvidia-container-toolkit-daemonset-8hfwn                  1/1     Running     0          14d
nvidia-cuda-validator-dlpmt                               0/1     Completed   0          14d
nvidia-dcgm-exporter-lm4kn                                1/1     Running     0          14d
nvidia-device-plugin-daemonset-9w9xk                      2/2     Running     0          14d
nvidia-driver-daemonset-sqs5c                             1/1     Running     0          14d
nvidia-operator-validator-7scl5                           1/1     Running     0          14d
```

nvidia-driver-daemonset

Loads the drivers on the node

nvidia-container-toolkit-ctr

The toolkit includes a container runtime library and utilities to automatically configure containers to leverage NVIDIA GPUs.

nvidia-dcgm-exporter + nvidia-operator-validator

NVIDIA Data Center GPU Manager (DCGM) is a suite of tools for managing and monitoring NVIDIA datacenter GPUs. It exposes GPU metrics exporter for Prometheus leveraging NVIDIA DCGM.

nvidia-device-plugin-daemonset

Allows to automatically:

1. Expose the number of GPUs on each nodes of your cluster
2. Keep track of the health of your GPUs
3. Run GPU enabled containers in your Kubernetes cluster.

This is what allows NVIDIA GPUs to be requested by a container using the **nvidia.com/gpu** resource type.

nvidia-cuda-validator

Validates that the stack installation worked

Node feature discovery

```
$ kubectl get pod -n kube-system | grep node-feature-discovery
cern-magnum-node-feature-discovery-gc-7985cbd94b-q499t       1/1     Running     0          17d
cern-magnum-node-feature-discovery-master-7bbccf9b68-fjpp8   1/1     Running     0          17d
cern-magnum-node-feature-discovery-worker-5qjzq              1/1     Running     0          17d
cern-magnum-node-feature-discovery-worker-qhbrc              1/1     Running     0          17d
```

Node feature discovery

```
$ kubectl get pod -n kub
cern-magnum-node-feature                                    ning     0          17d
cern-magnum-node-feature                                    ning     0          17d
cern-magnum-node-feature                                    ning     0          17d
cern-magnum-node-feature                                    ning     0          17d
```

```
    local.feature:
      elements:
        nvidia.com/cuda.driver-version.full: 550.54.15
        nvidia.com/cuda.driver-version.major: "550"
        nvidia.com/cuda.driver-version.minor: "54"
        nvidia.com/cuda.driver-version.revision: "15"
        nvidia.com/cuda.driver.major: "550"
        nvidia.com/cuda.driver.minor: "54"
        nvidia.com/cuda.driver.rev: "15"
        nvidia.com/cuda.runtime-version.full: "12.4"
        nvidia.com/cuda.runtime-version.major: "12"
        nvidia.com/cuda.runtime-version.minor: "4"
        nvidia.com/cuda.runtime.major: "12"
        nvidia.com/cuda.runtime.minor: "4"
        nvidia.com/gfd.timestamp: "1728992460"
        nvidia.com/gpu.compute.major: "7"
        nvidia.com/gpu.compute.minor: "5"
        nvidia.com/gpu.count: "1"
        nvidia.com/gpu.family: turing
        nvidia.com/gpu.machine: OpenStack-Compute
        nvidia.com/gpu.memory: "15360"
        nvidia.com/gpu.mode: compute
        nvidia.com/gpu.product: Tesla-T4
        nvidia.com/gpu.replicas: "1"
        nvidia.com/gpu.sharing-strategy: none
        nvidia.com/mig.capable: "false"
        nvidia.com/mig.strategy: mixed
        nvidia.com/mps.capable: "false"
        nvidia.com/vgpu.present: "false"
```

Allocatable:

...

nvidia.com/gpu: 1

```yaml
apiVersion: v1
kind: Pod
metadata:
 name: tf-gpu
spec:
 containers:
 - name: tf
   image: tensorflow/tensorflow:latest-gpu
   command: ["sleep", "inf"]
   resources:
     limits:
       nvidia.com/gpu: 1
```

```
$ kubectl exec -it tf-gpu -- bash

WARNING: You are running this container as root, which can cause new files in
mounted volumes to be created as the root user on your host machine.

To avoid this, run the container by specifying your user's userid:

$ docker run -u $(id -u):$(id -g) args...

root@tf-gpu:/# nvidia-smi
Tue Oct 29 14:09:00 2024
+-----------------------------------------------------------------------------------------+
| NVIDIA-SMI 550.54.15              Driver Version: 550.54.15      CUDA Version: 12.4     |
|-----------------------------------------+------------------------+----------------------+
| 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  Tesla T4                       On  |   00000000:00:07.0 Off |                    0 |
| N/A   49C    P8             16W /   70W |       0MiB /  15360MiB |      0%      Default |
|                                         |                        |                  N/A |
+-----------------------------------------+------------------------+----------------------+

+-----------------------------------------------------------------------------------------+
| Processes:                                                                              |
|  GPU   GI   CI        PID   Type   Process name                              GPU Memory |
|        ID   ID                                                               Usage      |
|=========================================================================================|
|  No running processes found                                                             |
+-----------------------------------------------------------------------------------------+
```

Tainting

Taint Nodes

With kubernetes templates 1.24+, the gpu-operator helm chart does not taint GPU nodes which will allow all workloads to run in this nodes. We suggest to taint the nodes explicitly by adding the following taint to the GPU nodegroups:

```
node-role.kubernetes.io/gpu=true:NoSchedule
```

Disclaimer:
We will have automatic tainting in the next release

Let’s run some workloads

Example Use Cases
(very different GPU consumption behaviour)

Badly coded simulation job:

- Low average GPU usage (CPU dependant workload)
- Needs 10 GiB VRAM (8 + 2 dynamic)
- Long running process

Never know what to expect from a notebook user:

- Potential memory leaks
- Poorly considered batch size
- GPU memory locked by an idle notebook

An inference service which is occasionally triggered by outside events:

- Spiky and unpredictable execution
- Mostly sits idle
- Saturates the GPU cores
- Max 10 GiB VRAM (2 + 8 dynamic)

* All use cases were run on a CERN Kubernetes cluster with 1 NVIDIA A100 40GB GPU

Onboard **Only** Use Case 1 = Dedicated GPU

Badly coded simulation job:

- Low average GPU usage (CPU dependant workload)
- Needs 10 GiB VRAM (8 + 2 dynamic)
- Long running process

- GPU underutilized
- Steady memory utilization ~ 20%

Dedicated GPU drawbacks

- Dedicated GPUs => small/limited GPU offering
- Some use cases cannot fully utilize a GPU => idle time

Dedicated GPU drawbacks

- Dedicated GPUs => small/limited GPU offering
- Some use cases cannot fully utilize a GPU => idle time

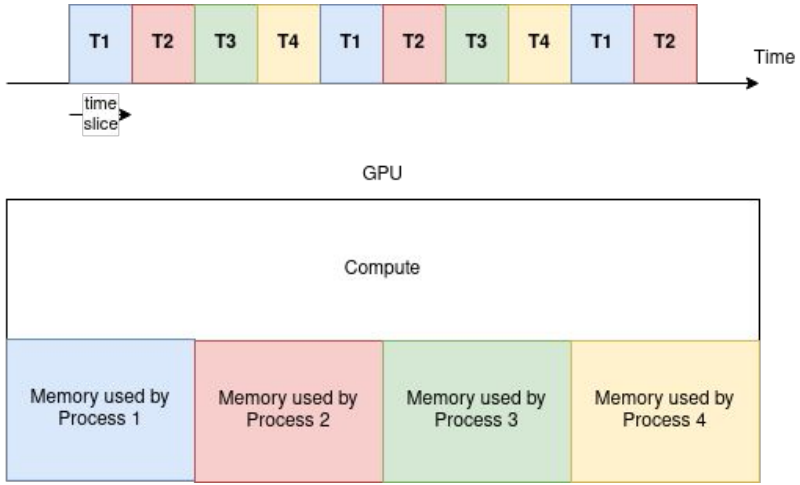
How to improve?

GPU Sharing

1. Time-slicing

Time-slicing

- The scheduler gives an equal share of time to all GPU processes and alternates them in a round-robin fashion.
- The memory is shared between the processes
- The compute resources are assigned to one process at a time



Allocatable:
...
nvidia.com/gpu: 1

```
# values.yaml in NVIDIA gpu operator Helm chart
...
devicePlugin:
 config:
   name: nvidia-time-slicing-config
```

```
# $ cat nvidia-time-slicing-config.yaml
apiVersion: v1
kind: ConfigMap
metadata:
 name: nvidia-time-slicing-config
 namespace: kube-system
data:
 slice-4: |-
   version: v1
   sharing:
     timeSlicing:
       renameByDefault: true
       failRequestsGreaterThanOne: true
       resources:
       - name: nvidia.com/gpu
         replicas: 4
```

```
apiVersion: v1
kind: Pod
metadata:
 name: tf-gpu
spec:
 containers:
   - name: tf
     image:
tensorflow/tensorflow:latest-gpu
     command: ["sleep", "inf"]
     resources:
       limits:
         nvidia.com/gpu.shared: 1
```

Allocatable:
...
nvidia.com/gpu: 0
nvidia.com/gpu.shared: 4

```
kubectl label node <node-name> nvidia.com/device-plugin.config=slice-4
```

Use case 1

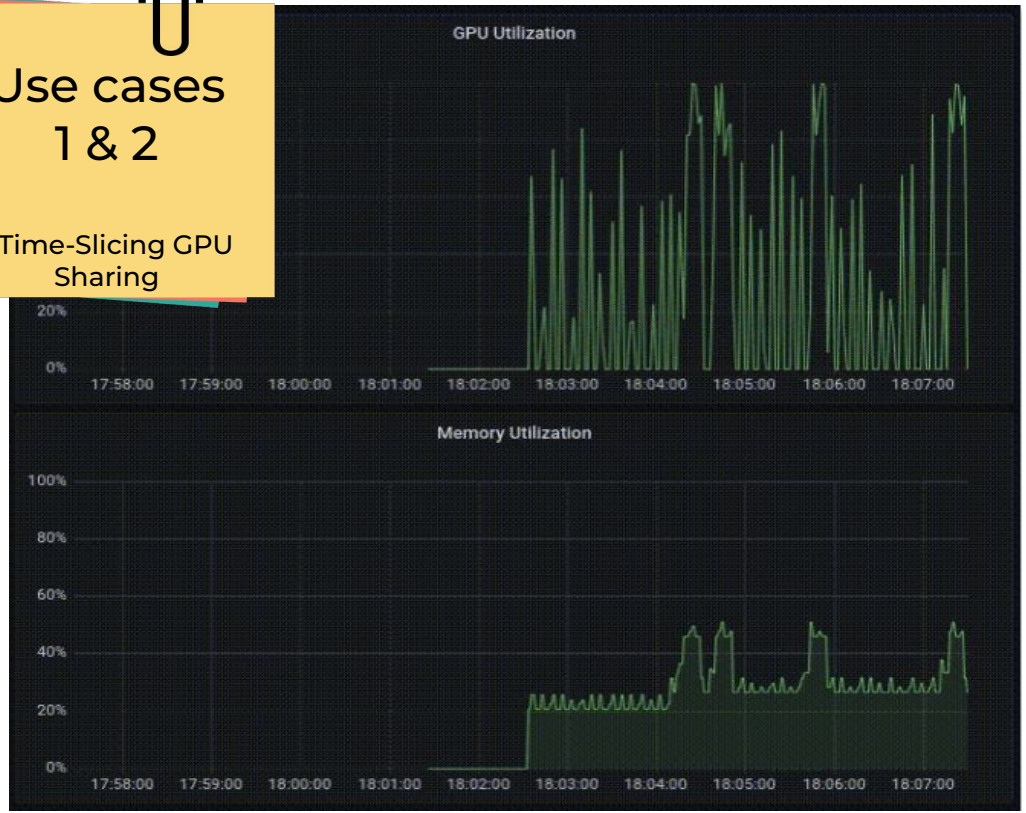
- GPU underutilized
- Steady memory utilization ~ 20%

Use case 1

- GPU underutilized
- Steady memory utilization ~ 20%

Use cases 1 & 2

* Time-Slicing GPU Sharing



- Improved GPU utilization
- **Better memory consumption (~ 50 %)**

Use cases 1 & 2 & 3

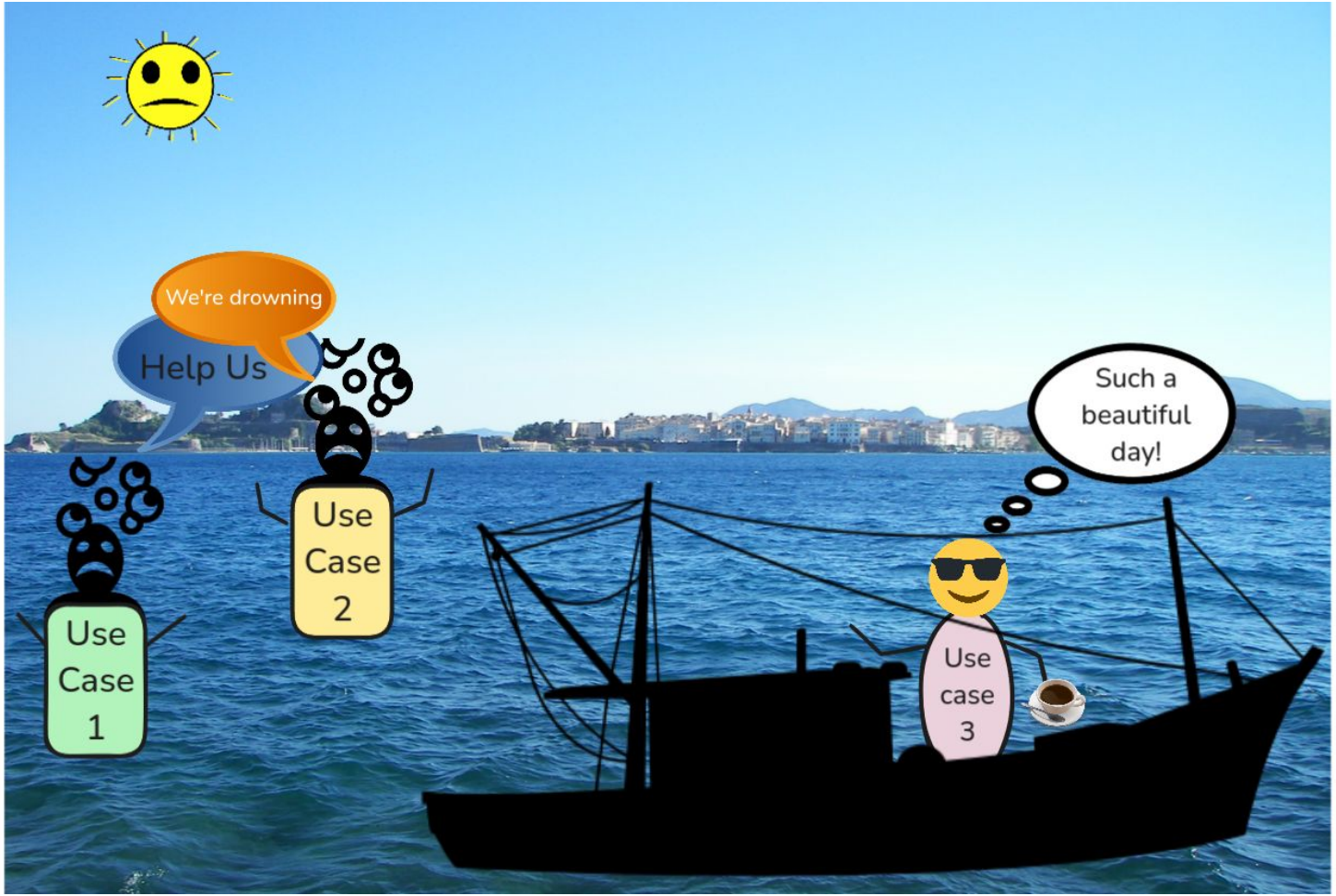
* Time-Slicing GPU Sharing

GPU utilization 100%

… Perfect, right?

No.

Use case 3 used all the memory, and **starved** the other 2 processes.


We're drowning
Help Us
Use Case 2
Use Case 1
Such a beautiful day!
Use case 3

Time-Slicing

Advantages	Disadvantages
Works on a wide range of NVIDIA architectures	No process/memory isolation
An easy way to set up GPU concurrency	No ability to set priorities
An unlimited number of partitions	Inappropriate for latency-sensitive applications (ex: desktop rendering for CAD workloads)

GPU Sharing

2. Multi Instance GPU

Multi Instance GPU

Multi Instance GPU (MIG) can partition the GPU into up to seven instances, each fully isolated with its own high-bandwidth memory, cache, and compute cores.

MIG Profiles on A100

NVIDIA MIG provides multiple strategies for allowing users to reference the graphic card resources:

- **mixed**: Different resource types are enumerated for every MIG device available. Ex: nvidia.com/mig-3g.20gb
- **single**: MIG devices are enumerated as nvidia.com/gpu, and map to the MIG devices available on that node, instead of the full GPUs.
- **none**: No distinction between GPUs with MIG or without. The available devices are listed as nvidia.com/gpu.

```
Allocatable:
  ...
  nvidia.com/gpu:  1
```

```
# values.yaml in NVIDIA gpu operator
Helm chart
...
mig:
 strategy: mixed
migManager:
 config:
   name: nvidia-mig-config

apiVersion: v1
kind: ConfigMap
metadata:
 name: nvidia-mig-config
data:
 config.yaml: |
   version: v1
   mig-configs:
     # A100-40GB
     3g.20gb-2x2g.10gb:
     - devices: all
       mig-enabled: true
       mig-devices:
         "2g.10gb": 2
         "3g.20gb": 1
```

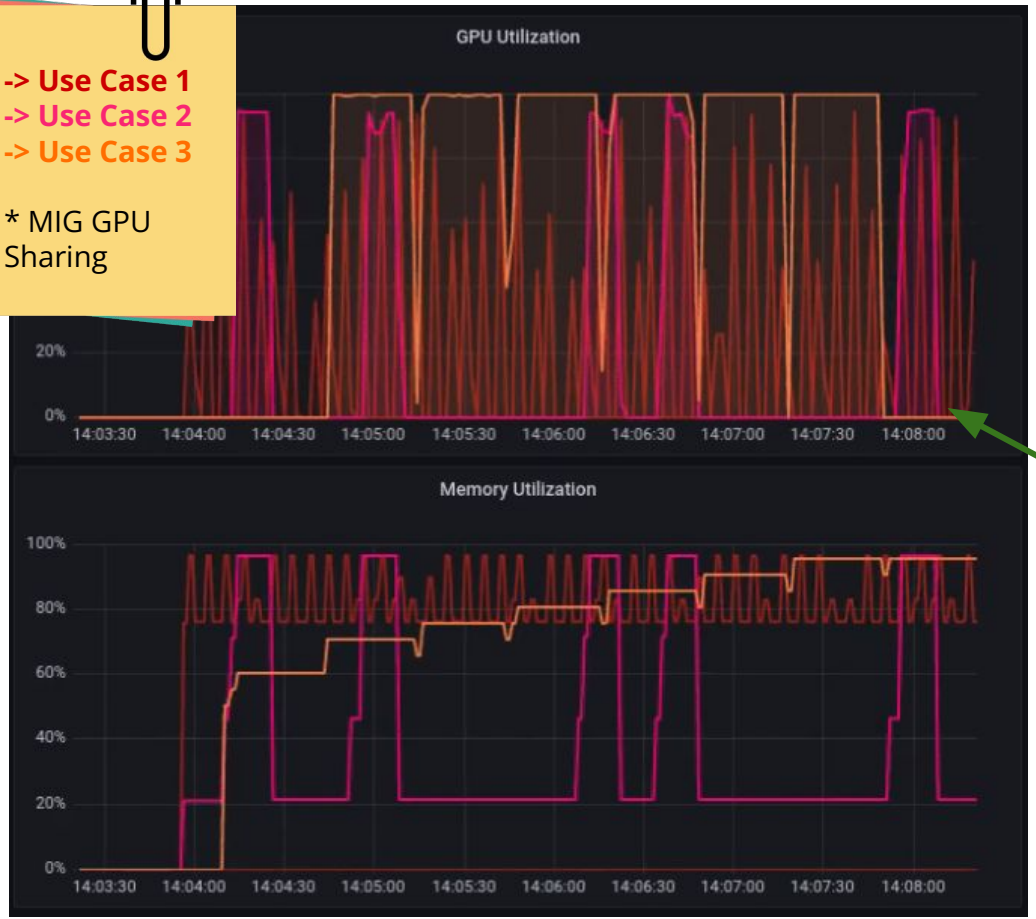
```
apiVersion: v1
kind: Pod
metadata:
 name: tf-gpu
spec:
 containers:
   - name: tf
     image:
tensorflow/tensorflow:latest-gpu
     command: ["sleep", "inf"]
     resources:
       limits:
         nvidia.com/mig-3g.20gb: 1
```

```
Allocatable:
  ...
  nvidia.com/gpu:          0
  nvidia.com/mig-2g.10gb:  2
  nvidia.com/mig-3g.20gb:  1
```

```
kubectl label nodes <node-name> nvidia.com/mig.config=3g.20gb-2x2g.10gb
```

-> Use Case 1
-> Use Case 2
-> Use Case 3

* MIG GPU Sharing



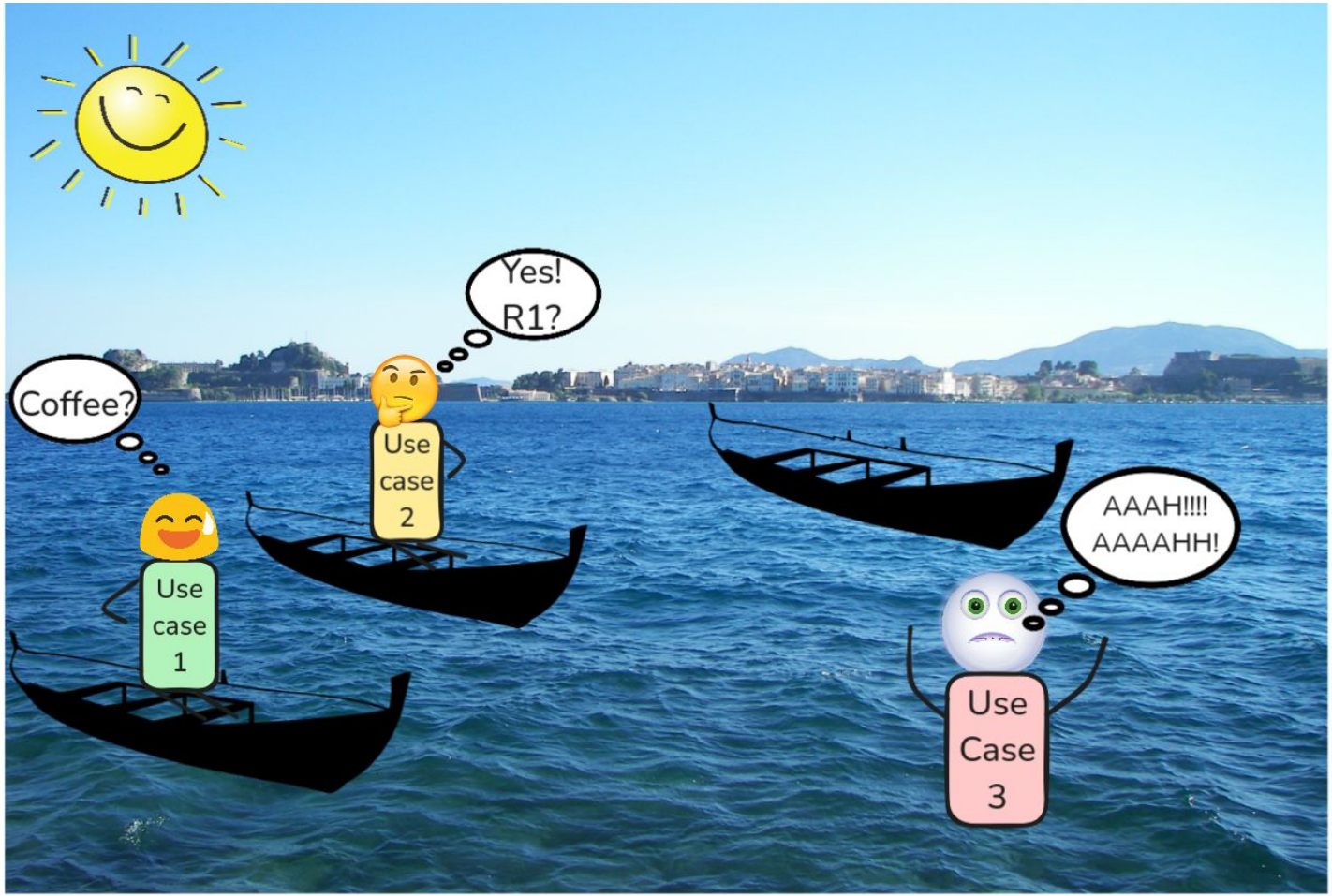
Every process:

- Is isolated
- Saturates own resources
- Cannot influence other processes

... Perfect, right?

Yes.

Use case 3 **starved itself**, use cases 1 & 2 continued running without issues!


Coffee?
Use case 1
Yes! R1?
Use case 2
AAAH!!!! AAAAHH!
Use Case 3

Hardware level sharing - MIG

Advantages	Disadvantages
Hardware isolation allows processes to run securely in parallel and not influence each other	Only available for Ampere, Hopper, and Blackwell architecture
Monitoring and telemetry data available at partition level	Reconfiguring the partition layout requires all running processes to be evicted
Allows partitioning based on use cases, making the solution flexible	* Potential loss of available memory depending on chosen profile layout

* Not a risk if the partitioning layout is chosen in an informed way after careful consideration.

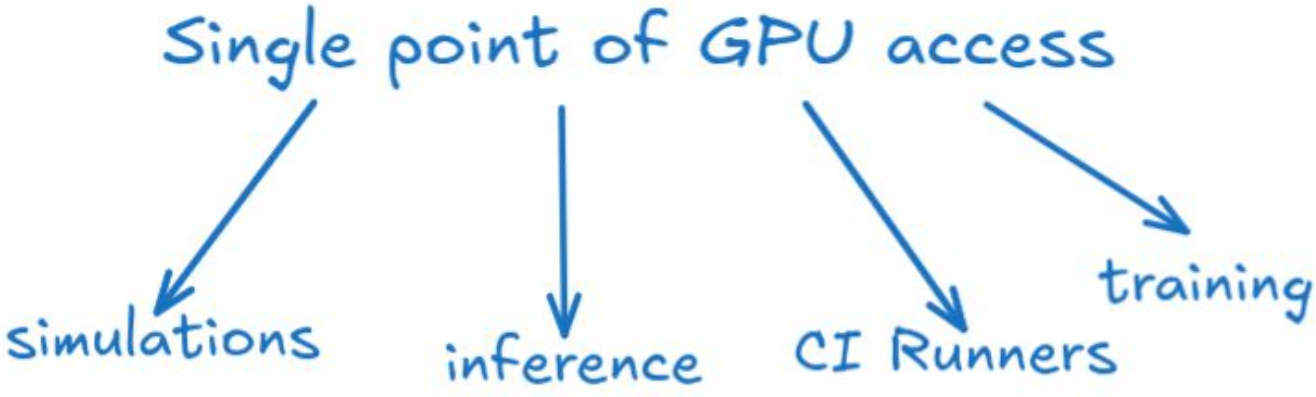
We established that GPU sharing increases overall usage.

But how do we share in the best way?

But how do we share in the best way?

1. Summarize **in your team** all workloads that need GPUs. Run them on one cluster and collocate them using time-slicing and MIG.

2. Single point of GPU Access across multiple teams



1. GPUs are always in-use
   a. As soon as a GPU is released by an user, it is reassigned to another one requesting a GPU
2. People can get access to multiple types of GPUs, or even other accelerators (TPUs, IPUs) through public cloud.

GPU sharing tradeoffs

Benchmarked script:

- Simulation script that generates collision events. Find more
- Built with Xsuite (Suite of python packages for multiparticle simulations for particle accelerators)
- Very heavy on GPU usage
- Low on memory accesses
- Low on CPU-GPU communication

Environment:

- NVIDIA A100 40GB PCIe GPU
- Kubernetes version 1.22
- Cuda version utilized: 11.6
- Driver Version: 470.129.06

Time-slicing Performance Analysis

Number of particles	Shared x1 [seconds]	Expected Shared x2 = Shared x1 * 2 [seconds]	Actual Shared x2 [seconds]	Loss [%]
15 000 000	77.12	154.24	212.71	37.90
20 000 000	99.91	199.82	276.23	38.23
30 000 000	152.61	305.22	423.08	38.61

The GPU **context switching** caused a ~38% performance loss

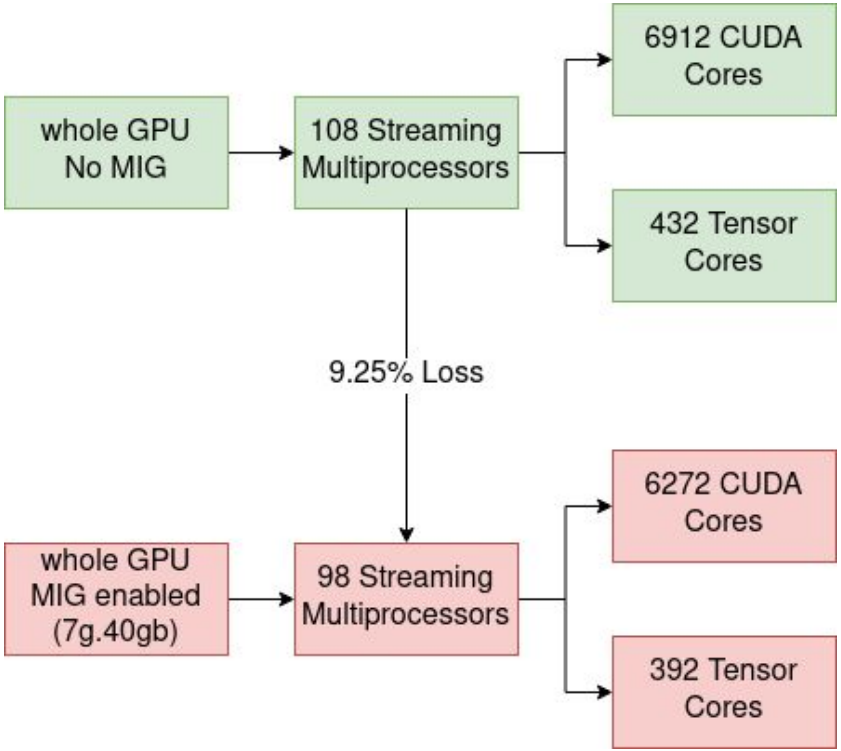
Time-slicing Performance Analysis

Number of particles	Shared x2 [seconds]	Shared x4 [seconds]	Loss [%]
15 000 000	212.71	421.55	0
20 000 000	276.23	546.19	0
30 000 000	423.08	838.55	0

Number of particles	Shared x4 [seconds]	Shared x8 [seconds]	Loss [%]
15 000 000	421.55	838.22	0
20 000 000	546.19	1087.99	0
30 000 000	838.55	1672.95	0

There is no additional performance loss when sharing the GPU between more processes (4, 8, and even more).

MIG Performance Analysis



MIG Performance Analysis

Number of particles	Whole GPU, no MIG [seconds]	Whole GPU, with MIG (7g.40gb) [seconds]	Loss [%]
5 000 000	26.365	28.732	**8.97 %**
10 000 000	51.135	55.930	**9.37 %**
15 000 000	76.374	83.184	**8.91 %**

The theoretical loss of **9.25%** is seen experimentally.

MIG Performance Analysis

Number of particles	7g.40gb [s]	3g.20gb [s]	2g.10gb [s]	1g.5gb [s]
5 000 000	28.732	62.268	92.394	182.32
10 000 000	55.930	122.864	183.01	362.10
15 000 000	83.184	183.688	273.7	542.3

Number of particles	3g.20gb / 7g.40gb	2g.10gb / 3g.20gb	1g.5gb / 2g.10gb
5 000 000	2.16	1.48	1.97
10 000 000	2.19	1.48	1.97
15 000 000	2.20	1.48	1.98
ideal scale	7/3 = 2.33	3/2 = 1.5	2/1 = 2

The scaling between partitions converges to ideal values.

GPU Sharing Use Cases

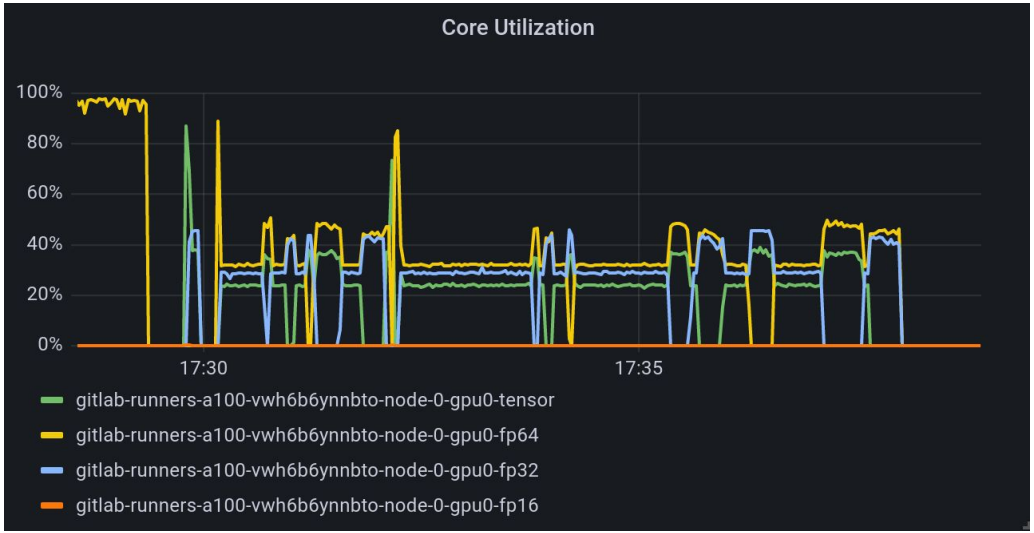
Category	Examples	Time slicing	MIG
Latency sensitive	CAD, Engineering Applications	✗	✓
Interactive	Notebooks	✓[1]	✓
Performance intensive	Simulation	✗	✓
Low priority	CI Runners	✓	✓

[1] Independent workloads can trigger OOM errors between each other. Needs an external mechanism to control memory usage (similar to kubelet CPU memory checks)

Monitoring

https://grafana.com/grafana/dashboards/18288-nvidia-gpu/

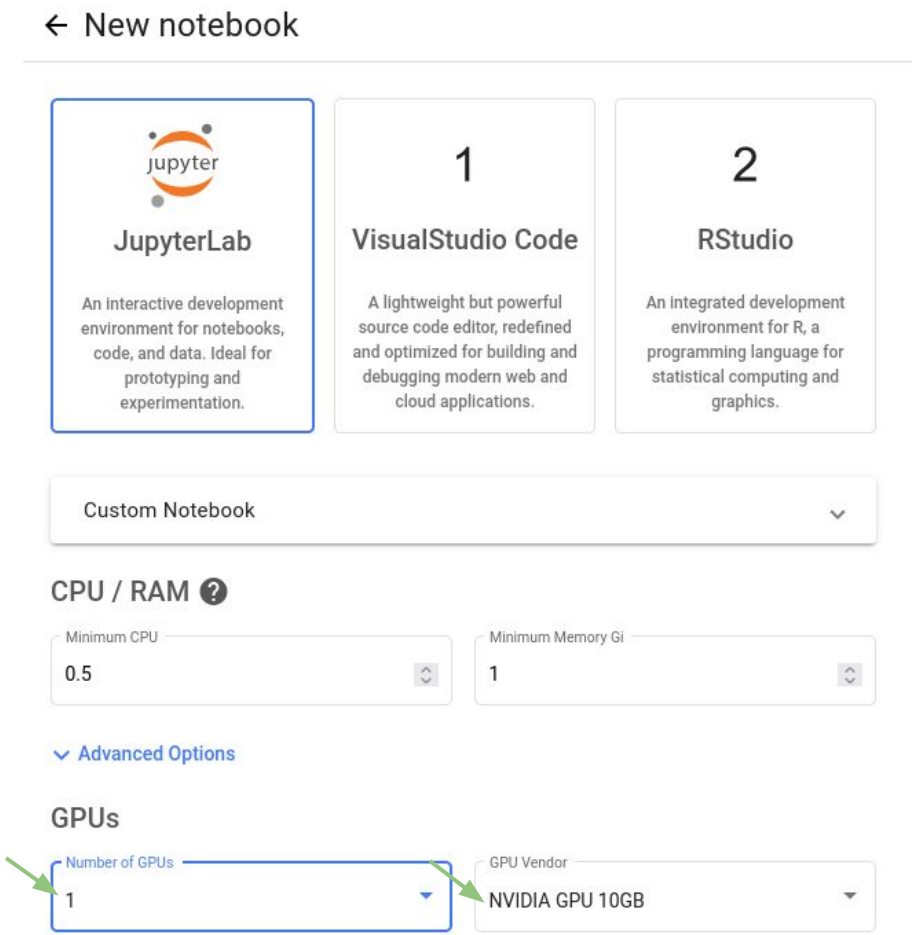
```
# dcgm-metrics.csv
...
    DCGM_FI_PROF_PIPE_TENSOR_ACTIVE, gauge, Ratio of cycles the tensor (HMMA) pipe is active (in %).
    DCGM_FI_PROF_PIPE_FP64_ACTIVE,   gauge, Ratio of cycles the fp64 pipes are active (in %).
    DCGM_FI_PROF_PIPE_FP32_ACTIVE,   gauge, Ratio of cycles the fp32 pipes are active (in %).
    DCGM_FI_PROF_PIPE_FP16_ACTIVE,   gauge, Ratio of cycles the fp16 pipes are active (in %).
```



Profiling the A100 compute pipeline utilization

https://docs.nvidia.com/datacenter/dcgm/2.4/dcgm-api/dcgm-api-field-ids.html

GPU access using Kubeflow



Find more:

- https://ml.docs.cern.ch/
- https://ml.cern.ch/

... and in the next ML session :)

Conclusions

1. It is easy to create a cluster with GPU nodes
   a. The user is abstracted away from having to set any drivers

2. GPU sharing is useful to improve the overall GPU utilization, but it comes with performance tradeoffs
   a. Sharing helps us to offer GPUs to more users
   b. For use cases that can fully utilize the GPU, we need to consider allocating dedicated GPUs

3. Monitoring is very important
   a. The current infrastructure is flexible enough to cater for various use cases

4. For ML workloads consider using Kubeflow

Thank you!

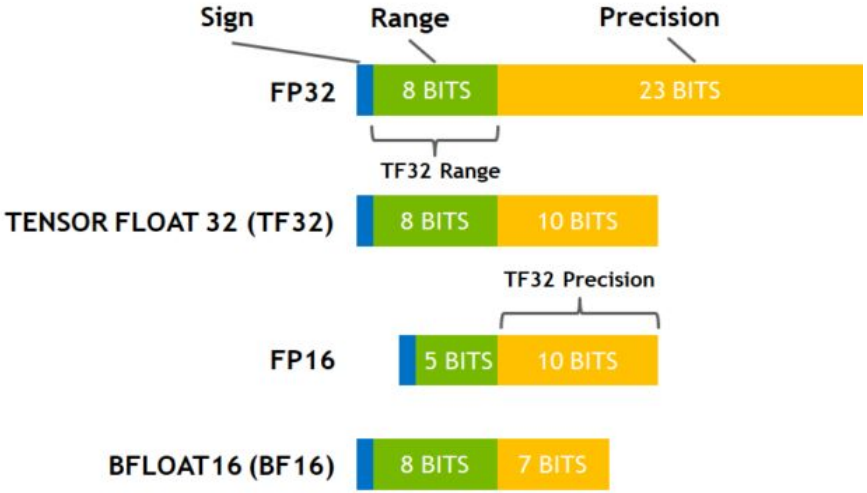
Bonus slides

Not a service, but very good to know

Data Formats

We’ve heard about fp32 and fp64, but what about ML?

Data formats



Usually ML doesn't need much precision, so people starting using half precision (fp16).

Since fp16 has small range (5 bits), Trainings with fp16 only, very often results in numerical errors (underflow, overflow).

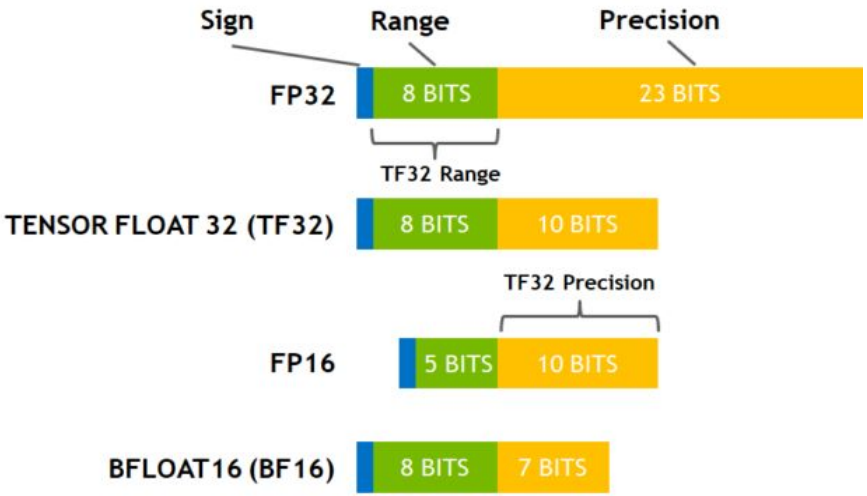
But ML needs range!

This is how mixed precision was invented:

- What is numerical stable -> done in fp16
- For the rest -> there is a copy stored in fp32

You get the speed, while avoiding numerical errors, but memory consumption is bigger (V100, T4).

Data formats



* ML needs range

This is how BFloat16 was created (available for Ampere and newer)

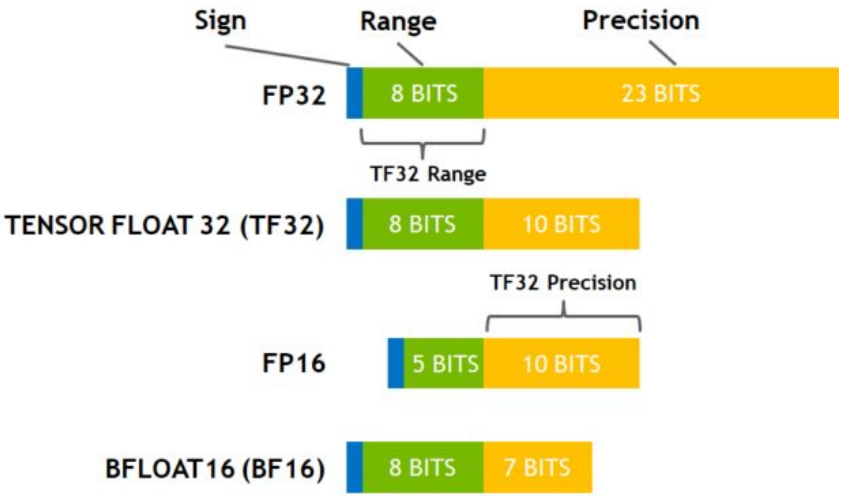
The range of fp32, the precision less than fp16

* ML doesn't need a lot of precision!

Using low precision works because ML tries to minimize the error. As long as it can represent the number and move towards a smaller error - low precision will work and bring speedup.

Low precision can be a problem for regression tasks (when you want to get a real number as output) - then think about using another format.

Data formats



tf32 is a solution to use when you still want the speedup of bf16, but with a bigger precision.

Things to keep in mind

- This is an NVIDIA solution (amd still not very clear what is the current state)
- not existing on older GPUs

Exercises Time

- Access the exercises:
- https://indico.cern.ch/event/1441237/contributions/6073469/attachments/2952019/5211317/exercises

- Group 1
  - Shahzaib Aamir
  - Berk Balci
  - Nayana Bangaru
  - Gábor Bíró
  - Abhishek Bohare
  - Marco Buonsante
- Group 2
  - Gianluca De Bonis
  - Elena de la Fuente Garcia
  - Jesse Geens
  - Ediz Genc
  - Panagiotis Georgopoulos
  - Daniel Goncalves Portovedo
- Group 3
  - Panagiotis Gkonis
  - Dmytro Gruzdo
  - Hannes Jakob Hansen
  - Musa Kaymaz
  - Idriss Larbi
  - Manuel Ramirez Garcia
- Group 4
  - Rimsky Alejandro Rojas Caballero
  - Jonathan Samuel
  - Nikolaos Smyrnioudis
  - Lorenzo Valentini
  - Jan Hauke Voss
  - Julian Weick
- Group 5
  - Joao Ramiro
  - Lorenzo Ventura Vagliano
  - Pascal Egner
  - Jakub Jelinek
  - Luis Pelaez Bover
  - Franciska-Leonora Toeroek