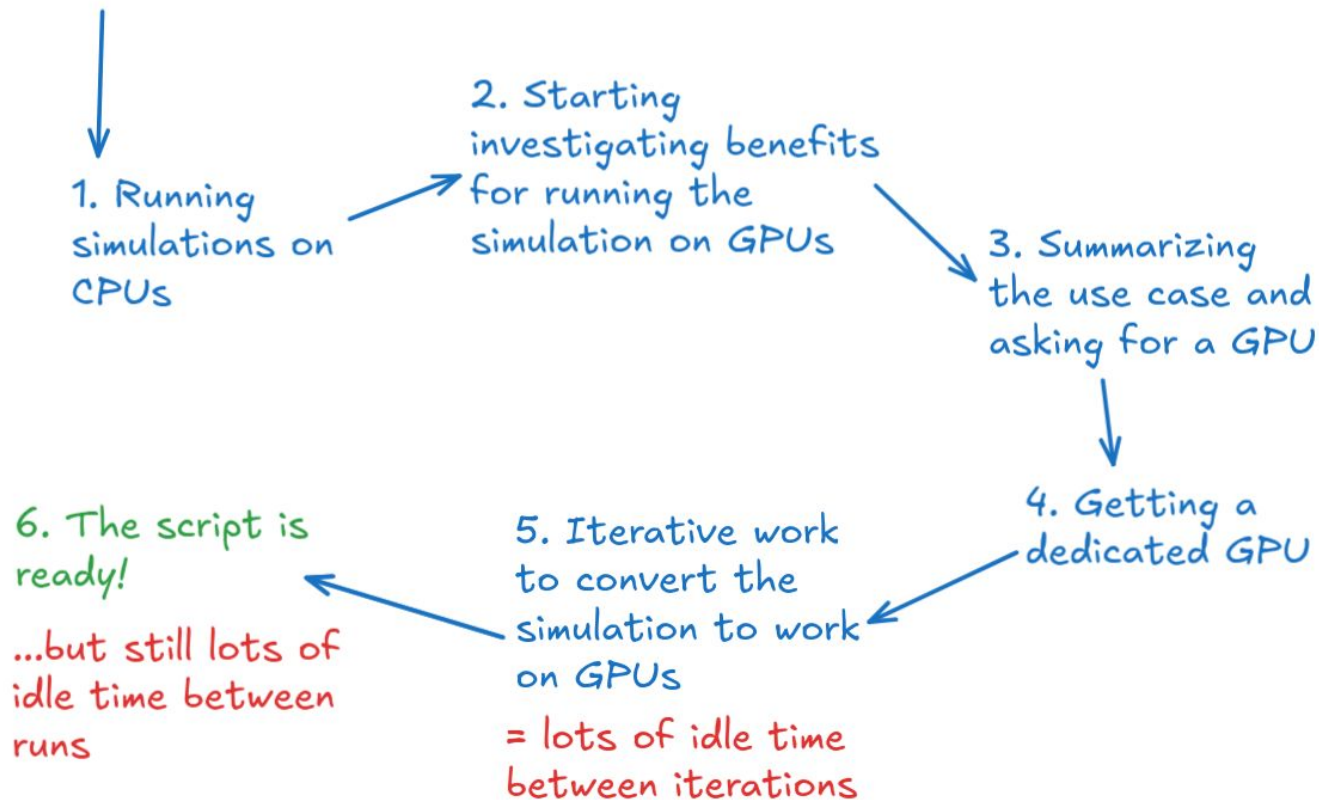


Improving overall GPU sharing and usage efficiency with Kubernetes

Diana Gaponcic, Ricardo Rocha, Diogo Filipe Tomas Guerra, Dejan Golubovic

CHEP 2024

What often happens...



What can we do?

What can we do?

GPU Sharing is Caring

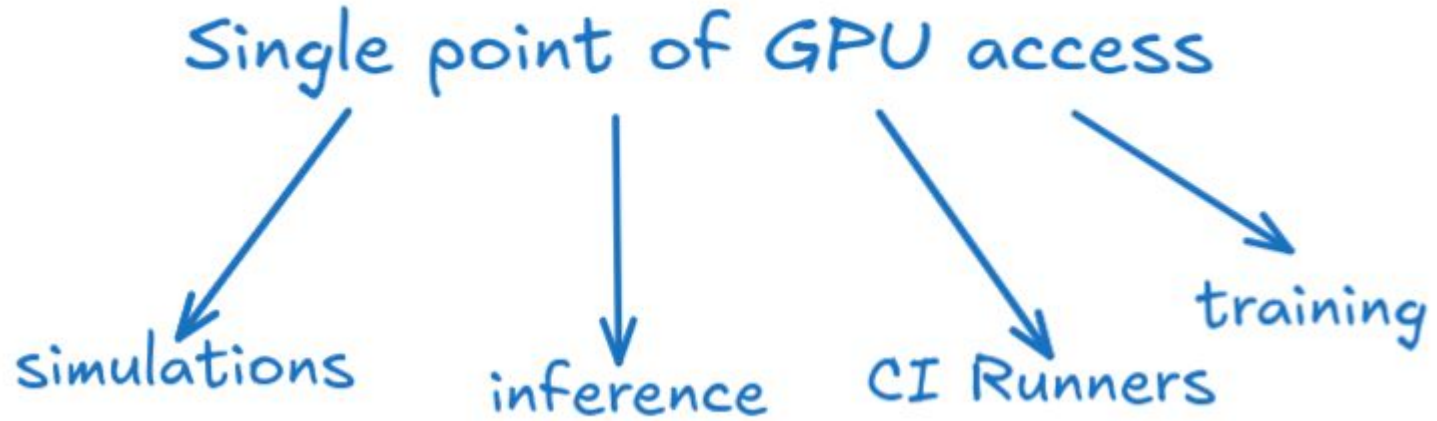
What can we do?

GPU Sharing is Caring

How do we share?

How do we share?

1. Infrastructure level
2. GPU level
 - a. logical level
 - b. hardware level



Kubernetes  does a great job at this

GPU Sharing at the Infrastructure level

Single point of access to a platform with GPU access:

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.

Let's see this in practice

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

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

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)

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 Use Case 1

Badly coded simulation job:

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

Onboard Use Case 1

Badly coded simulation job:

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

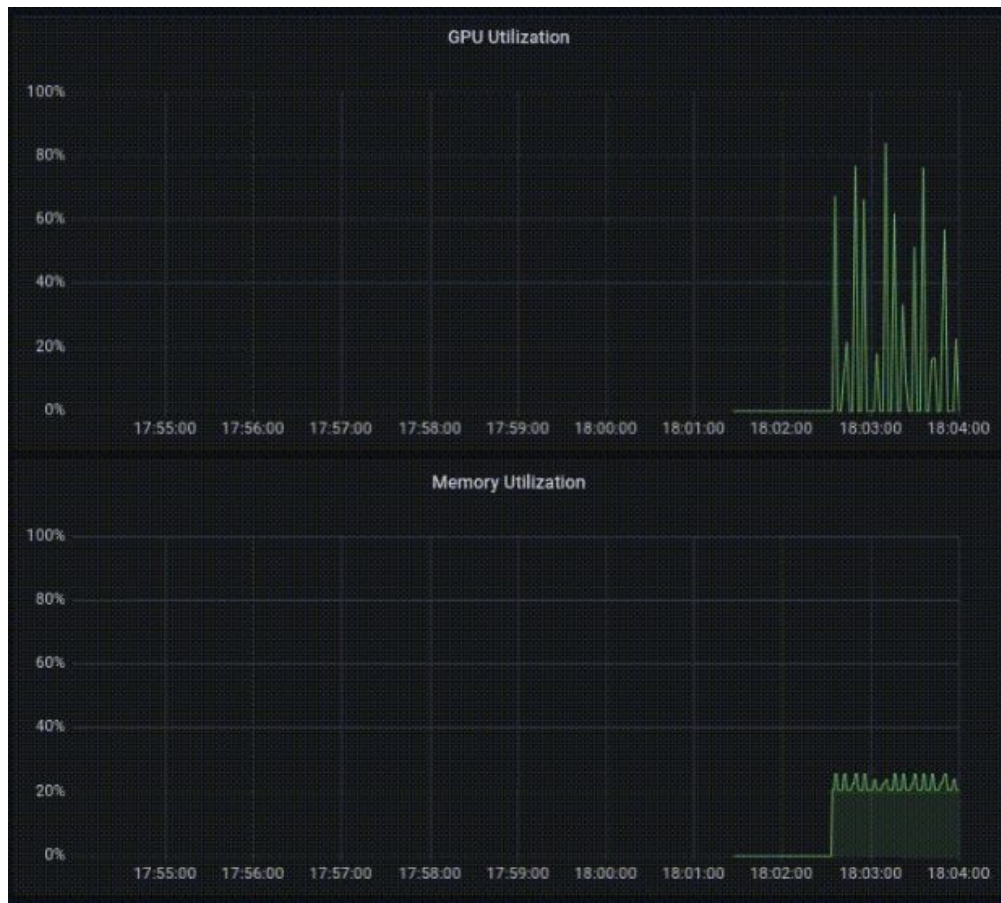


Onboard Use Case 1

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%



Conclusion

Unused GPUs can be re-assigned to other users requesting them => This ensures GPUs are always in use

Many use cases will not fully utilize the GPUs => A lot of wasted idle resources

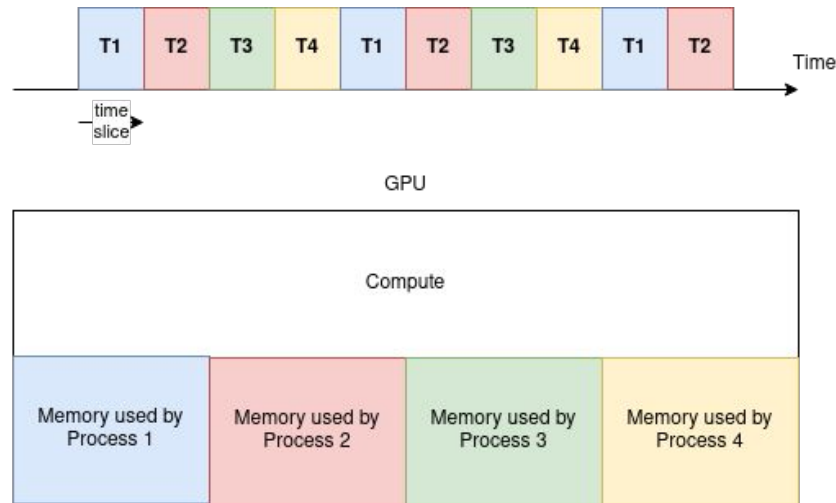
Can we do more?

How do we share

1. Infrastructure level
2. GPU level
 - a. logical level
 - b. hardware level

Logical level sharing: 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



How to setup Time-slicing on Kubernetes

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

Allocatable:

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



```
# $ 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
```



Allocatable:

```
...
nvidia.com/gpu: 0
nvidia.com/gpu.shared: 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?

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.

Logical level sharing: Time-Slicing

Advantages

Works on a wide range of NVIDIA architectures

Easy way to set up GPU concurrency

An unlimited number of partitions

Disadvantages

No process/memory isolation

No ability to set priorities

Inappropriate for latency-sensitive applications (ex: desktop rendering for CAD workloads)

Can we do even more?

How do we share

1. Infrastructure level
2. GPU level
 - a. logical level
 - b. hardware level

Hardware level sharing - MIG

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.



- 1 x 7g.40gb
or
- 2 x 3g.20gb
or
- 3 x 2g.10gb
or
- 7 x 1g.5gb

[MIG Profiles on A100](#)

How to setup MIG on Kubernetes

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

Allocatable:

...
nvidia.com/gpu: 1



```
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
```



Allocatable:

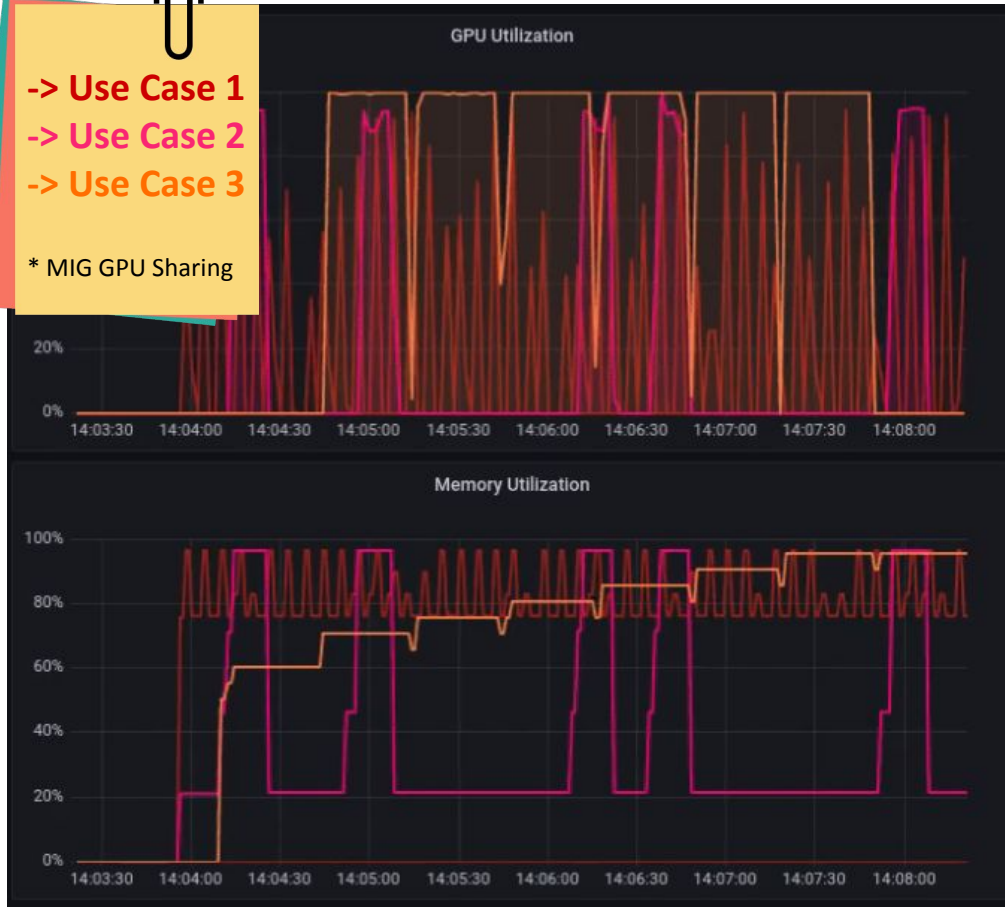
...
nvidia.com/gpu: 0
nvidia.com/mig-2g.10gb: 2
nvidia.com/mig-3g.20gb: 1

-> Use Case 1

-> Use Case 2

-> Use Case 3


* MIG GPU Sharing



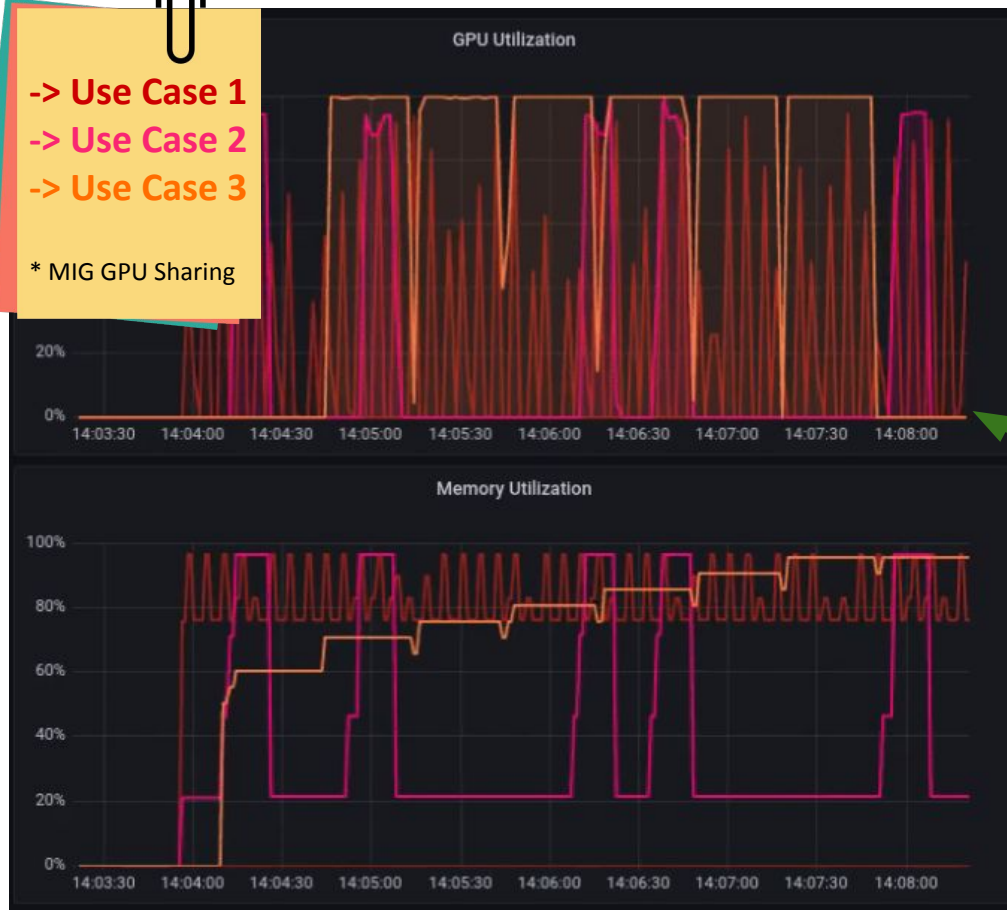
Every process:

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

... Perfect, right?

- 
- > 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!

Hardware level sharing - MIG

Advantages

Hardware isolation allows processes to run securely in parallel and not influence each other

Monitoring and telemetry data available at partition level

Allows partitioning based on use cases, making the solution flexible

Disadvantages

Only available for Ampere, Hopper, and Blackwell architecture

Reconfiguring the partition layout requires all running processes to be evicted

* 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.

Performance tradeoffs

What is the price of sharing?

Benchmarked script:

- Simulation script that generates collision events
- Built with Xsuite
- Very heavy on GPU usage
- Low on memory accesses
- Low on CPU-GPU communication

Find more:

- [Xsuite](#)
- [Benchmarked script](#)

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 (going from shared x1 to shared x2) leads to a **performance loss of ~38%**.

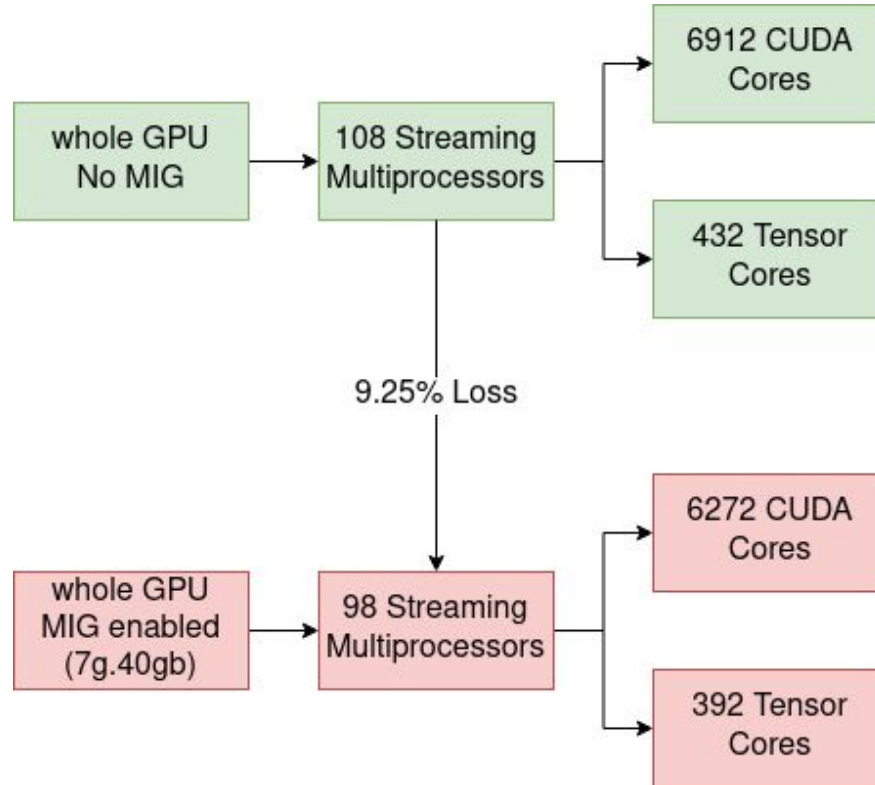
Time-slicing Performance Analysis

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

Sharing the GPU between more processes (4, 8), doesn't introduce additional performance loss.

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% can be 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.

A note on monitoring

- Never underestimate the importance of the monitoring infrastructure.
- Kubernetes makes monitoring easy (kube-prometheus-stack + gpu-operator)

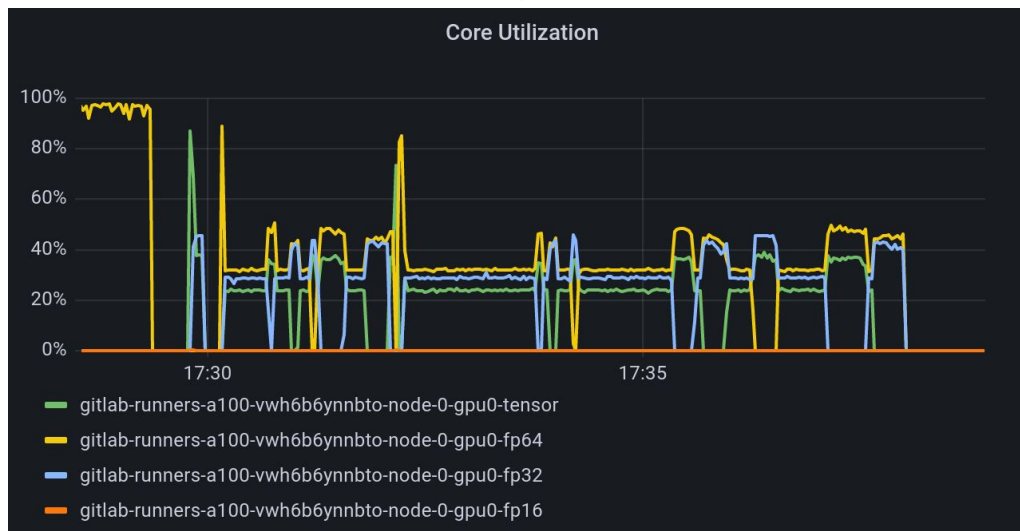
Find more:

- [GPU Grafana Dashboard](#)
- [NVIDIA DCGM](#)
- [DCGM Field Identifiers](#)


```
# 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

Conclusions

1. Single Point of Access for GPUs is the way
 - a. This is still not enough if the use cases are not fully utilising the GPU
2. There are multiple solutions to share a single GPU between multiple users - but this comes with tradeoffs
 - a. Provide dedicated GPUs to use cases that fully utilize the GPUs, to avoid performance losses
3. **A combination of sharing at different infrastructure levels is needed to gain optimal GPU usage.**

Thank you!