EPJ Web of Conferences **337**, 01079 (2025) https://doi.org/10.1051/epjconf/202533701079
CHEP 2024

Improving overall GPU sharing and usage efficiency with Kubernetes

Gaponcic Diana[1,*], *Ricardo* Rocha[1,**], *Diogo Filipe* Tomas Guerra[1], and *Dejan* Golubovic[1]

[1]CERN

Abstract. GPUs and accelerators are enabling High Energy Physics (HEP) to keep pace with the growing data volume and computational complexity. The challenge remains to improve overall efficiency and sharing opportunities of what are currently expensive and scarce resources.

In this paper, we describe the common patterns of GPU usage in HEP, including spiky requirements with low overall usage for interactive access, as well as more predictable but potentially bursty workloads. We then explore the multiple mechanisms to share and partition GPUs, covering time-slicing, and physical partitioning (MIG) for NVIDIA devices.

We conclude with the results of an extensive set of benchmarks for representative HEP use cases. We highlight the limitations of each option and the use cases where they fit best. Finally, we cover the deployment aspects and the different options available targeting a centralized GPU pool that can significantly push the overall GPU usage efficiency.

1 Introduction

GPUs are shaping the way organizations access and use their data, and CERN is not an exception. High Energy Physics (HEP) analysis and deployments are being rethought due to the growing complexity and volume of data, which traditional computational methods struggle to process efficiently. In this context, accelerators remain the key to enabling efficient Machine Learning (ML). [1]

The problem arises when we realize that in reality, many use cases cannot fully utilize the available hardware. This can be caused by many reasons such as workloads designed with CPU in mind, badly considered batch sizes, wrong assumptions about hardware requirements, etc.

Even when the experts understand the software and the hardware well, and are invested in getting maximum performance, some resources will stay idle due to the iterative nature of the development process. To solve this problem, it is important to come up with ways of sharing the available resources. This can be done either at the infrastructure level or at the GPU level itself.

*e-mail: diana.gaponcic@cern.ch
**e-mail: ricardo.rocha@cern.ch

© The Authors, published by EDP Sciences. This is an open access article distributed under the terms of the Creative Commons Attribution License 4.0 (https://creativecommons.org/licenses/by/4.0/).

2 Sharing Resources at the Infrastructure Level

Sharing at the infrastructure level means having a single point of GPU access. As you can see in figure 1, this needs to be a platform that acts as the entry point for workloads requiring a GPU, regardless of what is being run on it: simulations, inference, CI jobs, training, etc.

Users access the platform, choose what GPU is preferred or required, how much memory is needed, and what is the usage pattern, and then get access to accelerators from a common pool of resources. This way, GPUs are always in use. As soon as a GPU is released, it can be reassigned to someone else. This also creates an opportunity to access GPUs, or even other accelerators (TPUs, IPUs) through the public clouds (especially for specialized hardware that we cannot get on-premise).

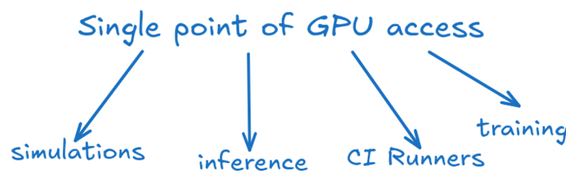

Figure 1. Single point of GPU access for various use cases

Kubernetes does a great job at being the common infrastructure for different use cases and utilization patterns. Many of them can be covered with Kubeflow [2] (a foundation of tools for AI platforms on Kubernetes), e.g. distributed training for machine learning. For the others, dedicated services can be deployed in the same cluster or across multiple clusters. Having a common pool of resources:

- increases the GPU offering at CERN (since many of the dedicated GPUs can be added to the common pool)
- increases the overall GPU usage (since GPUs stay idle for less time).

Even though this is a huge improvement, one issue still remains: some resources will be wasted if the user case cannot fully utilize the obtained GPU. To solve this, it is important to think of sharing at a more granular level - sharing at the GPU level itself.

3 Sharing Resources at the GPU Level

There are many benefits to sharing GPUs, but sharing also comes with added complexity. One big benefit is the cost optimization, since accelerators are expensive and organizations can benefit from giving access to more users, and getting increased overall hardware utilization. The same applies to energy and sustainability awareness.

At the same time, in a multi-tenant setup we introduce new problems that didn't previously exist. Some of them are the noisy neighbor issue, the lack of data isolation, the loss of efficienty due to the complexity to manage multiple users concurrently, etc. Those issues are key points that need to be addressed to ensure multiple workloads can run safely at the same time on the same chip.

3.1 Time-slicing

Time-slicing 2 is a sharing mechanism, where the scheduler gives an equal share of time to all GPU processes and alternates them in a round-robin fashion. Figure 2 shows how process are sharing the memory, while the compute resources are assigned to one process at a time.

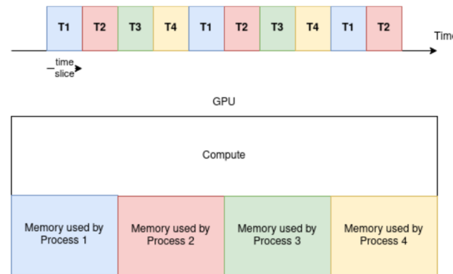

Figure 2. High level overview how time-slicing works

To provision GPUs in Kubernetes, we need to treat them as special resources and have a way to identify them, allocate them to workloads, and monitor their health. To help with this task, we can use the NVIDIA gpu-operator [3]. The operator automates the management of NVIDIA software needed to use GPUs, those include the NVIDIA drivers, the device plugin, the container toolkit, etc.

To enable time-slicing on Kubernetes, some extra configuration needs to be provided to the gpu-operator. First we need to let the device plugin know that it needs to use the configuration available in a ConfigMap (see figure 3), in this case, we called the ConfigMap nvidia-time-slicing-config. Then the description of the desired sharing options needs to be added to the referenced ConfigMap. For example, in slice-4, the GPU will be shared into 4 replicas with time-slicing, and the GPU resource will be renamed.

```
# 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
data:
 slice-4: |-
   version: v1
   sharing:
     timeSlicing:
       renameByDefault: true
       failRequestsGreaterThanOne: true
       resources:
       - name: nvidia.com/gpu
         replicas: 4
```

Figure 3. The configuration needed to enable time-slicing

When the configuration works properly, the node will start advertising nvidia.com/gpu.shared resources, instead of the default nvidia.com/gpu:

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

Time-slicing works on a wide range of NVIDIA architecture, it is an easy way to set up GPU concurrency, and offers an unlimited number of partitions. On the other hand, there is no process/memory isolation, and no ability to set priorities.

3.2 Multi Instance GPU

Multi Instance GPU (MIG) is a mechanism that allows the partitioning of a GPU into up to seven instances, each fully isolated with its own high-bandwidth memory, cache, and compute cores. The partitions on an A100 40GB are shown in figure 4.

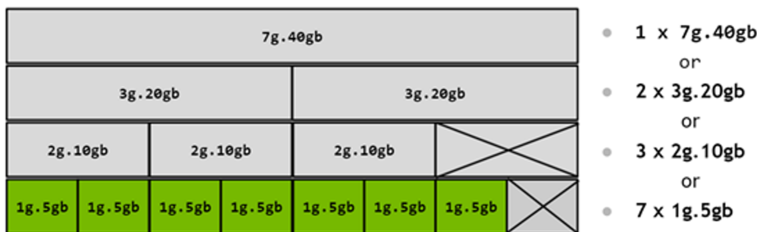

Figure 4. MIG Profiles on A100 40GB [4]

To enable MIG on Kubernetes, some extra configuration needs to be provided to the GPU operator:

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

# $ cat nvidia-mig-config.yaml
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
```

First we need to decide on a MIG strategy (mixed/single/none), then let the MIG manager know that it needs to use the configuration available in the nvidia-mig-config. Lastly, the description of the mig options has to be added to the referenced ConfigMap. For example, with 3g.20gb-2x2g.10gb, the GPU will be shared into 3 instances - 2 instances 2g.10gb, and 1 instance 3g.20gb. If we sum up the instances, we get 7g.40gb, which is the equivalent of the full GPU.

If the configuration works properly, the node will start advertising the new resources, which in this case are named based on the instance they represent:

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

A very important MIG addition is the possibility to have telemetry data per instance. As shown in the figure 5, every instance can be monitored independently, The telemetry allows for detailed insights and analysis of each instance's performance, resource utilization, and health metrics.

Figure 5. Monitoring a MIG partitioned GPU per instance

MIG offers numerous advantages in various computing environments. One key benefit is the hardware isolation that allows processes to run securely in parallel and not influence each other. Additionally, MIG provides monitoring and telemetry data at partition level. This makes MIG a very flexible solution, that can be tailored to diverse workload requirements.

Howether, MIG also comes with certain disadvantages that need to be considered. First of all, it is only available for Ampere, Hopper, and Blackwell architecture. Another challenge is that reconfiguring the partition layout requires all running processes to be evicted, which can disrupt ongoing tasks. Furthermore, there is a potential loss of available memory depending on the chosen profile layout, although this risk can be mitigated if the partitioning layout is selected thoughtfully after careful consideration.

4 Benchmarking GPU Sharing Mechanisms

The benchmarking was done with a simulation of LHC turning particles. We used an OpenCL-oriented Simpletrack benchmarking built for a selection of GPU and CPU platforms. In this case, we only used the NVIDIA one. More details on the benchmarked script:

- built with Xsuite [5]
- heavy on GPU usage
- low on CPU-to-GPU communication and memory accesses
- run on an NVIDIA A100 40GB PCIE GPU
- run on a Kubernetes 1.22 cluster (Cuda version utilized: 11.6, Driver Version: 470.129.06)

The used script [6] allows changing the number of particles and turns. We used the default value for turns, which is 15, but were changing the number of particles to increase and decrease the amount of computation to be done on the NVIDIA device.

4.1 Time-slicing

First we schedule only one process on the GPU, then slowly double the number of processes. Initially we compare the times to execute on a full GPU and a GPU time-sliced into 2, the results are shown in table 1. We consider as the reference value the execution time when only one process is running multiplied by the number of processes. Afterwards, we compare the execution time when we time-sliced into 2 vs 4 (see table 2) and 4 vs 8 (see table 3)

Number of particles	Shared x1 [s]	Shared x1 * 2 [s]	Shared x2 [s]	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

Table 1. Comparing full GPU vs sharing between two processes with time-slicing

Number of particles	Shared x2 [s]	Shared x4 [s]	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

Table 2. Benchmarking time-slicing with two and four processes

Number of particles	Shared x4 [s]	Shared x8 [s]	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

Table 3. Comparing sharing between four and eight processes with time-slicing

Conclusions:

1. If we run 1 process on a dedicated GPU, and then 2 identical processes, we would expect the execution time to double. In practice, since the GPU needs to perform context switching (going from shared x1 to shared x2), there is a performance loss of 38%.

2. Sharing a GPU with time-slicing between 2 processes introduces a big performance penalty. However increasing the number of processes on the GPU (4, 8) doesn't introduce additional performance loss.

4.2 Multi Instance GPU

A full A100 GPU has 108 Streaming Multiprocessors, or SMs. When MIG is enabled on the GPU (7g.40gb), 10 SMs are lost - which represents 9.25% of the available compute. Figure 6 shows how much those lost SMs influence the amount of CUDA cores and tensor cores available to perform the actual work.

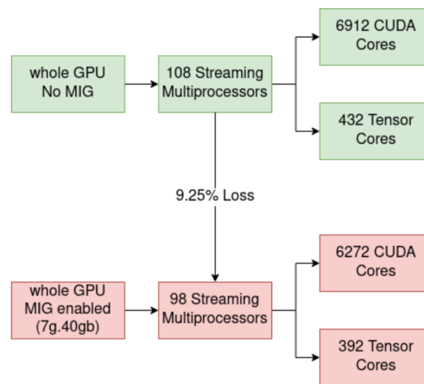

Figure 6. Performance loss for an A100 GPU when MIG is enabled

First we compare the benchmarks on the full GPU, then on a full GPU but with MIG enabled (see table 4). Afterwards we partition into smaller instances (table 5) instances and compare the performance (table 6).

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

Table 4. Benchmarking on a full GPU and a GPU with MIG enabled

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

Table 5. Benchmarking MIG partitioning

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

Table 6. Comparing the scaling between MIG partitions

Conclusions:

1. When we run the benchmarking script on a full GPU without MIG and with MIG enabled (7g.40gb), we conclude that the theoretical loss of 9.25% is also seen experimentally.

2. We should never enable MIG on a GPU, if we will not make use of it by partitioning, as it comes with huge performance loss.

3. The scaling between partitions converges to ideal values. In a linear manner, when the available resources increase, the execution time becomes smaller.

5 Monitoring GPUs

Monitoring is an important part of any infrastructure. It allows tracking and identifying issues, predicting usage or behaviour, setting up alerting, etc. Even though it can be complicated to set up, Kubernetes makes it quite easy to get insights about the GPU usage on a cluster.

We can start by using kube-prometheus-stack [7], which is a set of manifests, tools, dashboards, that make cluster monitoring easy. On the other side we have the gpu-operator (in particular NVIDIA DCGM [8]), that collects metrics from the GPUs in the cluster, and exposes them to an endpoint that needs to be scraped. In this way, by using kube-prometheus-stack + gpu-operator, the GPU metrics are already generated and available to the Prometheus/Grafana stack for visualization.

Usually, the default GPU metrics are enough, but when more granular/specific metrics are needed, one can make use of DCGM Field Identifiers [9], which allow enabling and disabling a very wide range of metrics.

An example of dashboard can be seen in figure 7. It shows the GPU utilization across all available GPUs, lists the MIG devices, has extensive information about the memory utilization per device or partition, the temperature, and can be adapted and extended as needed.

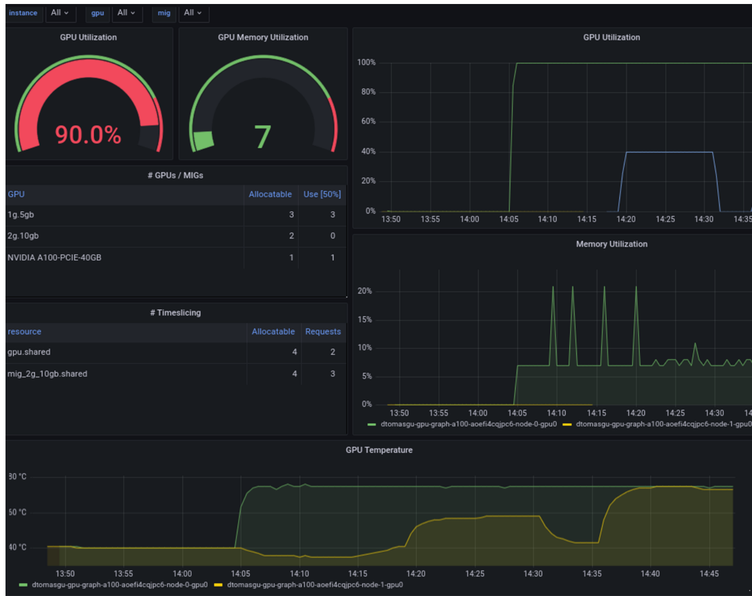

Figure 7. Example dashboard for monitoring GPUs

6 Conclusions

To conclude, having a single point of access for GPUs is a solution that can greatly decrease the idle time of resources, while also increasing the overall GPU offering at CERN. Still, this cannot be a solution when the use cases cannot fully utilize the available GPUs. To improve this, we need to start thinking of sharing at the GPU level (either logical or hardware). Still, since sharing comes with performance tradeoffs, there must be a mechanism to provide dedicated GPUs to use cases that will fully utilize them, to avoid performance losses. Therefore a combination of sharing at different infrastructure levels is needed to gain optimal GPU usage.

References

[1] Efficient Access to Shared GPU Resources https://kubernetes.web.cern.ch/tags/gpu/. Accesses on: December 12 2024
[2] Kubeflow https://www.kubeflow.org/. Accessed on: June 18 2025
[3] . Accesses on: December 12 2024 NVIDIA gpu-operator https://github.com/NVIDIA/gpu-operator
[4] . Accesses on: December 12 2024 MIG User Guide https://docs.nvidia.com/datacenter/tesla/mig-user-guide/. Accesses on: December 12 2024
[5] Xsuite https://github.com/xsuite/xsuite Accessed on: June 20 2025
[6] Benchmarked script https://gitlab.cern.ch/hep-benchmarks/hep-workloads-gpu/-/blob/master/lhc/simpletrack/lhc-simpletrack.sh?ref_type=heads. Accessed on: June 20 2025
[7] kube-prometheus-stack https://github.com/prometheus-community/helm-charts/tree/main/charts/kube-prometheus-stack. Accesses on: December 23 2024
[8] NVIDIA DCGM https://developer.nvidia.com/dcgm. Accesses on: December 23 2024
[9] NVIDIA Field Identifiers https://docs.nvidia.com/datacenter/dcgm/2.4/dcgm-api/dcgm-api-field-ids.html. Accesses on: December 23 2024