Pinpoint resource allocation for GPU batch applications

Tim Voigtländer, Manuel Giffels, Günter Quast, Matthias Schnepf, Roger Wolf
tim.voigtlaender@kit.edu

Karlsruhe Institute of Technology (KIT) - Institute of Experimental Particle Physics (ETP)

Big GPUs and small applications

Modern HEP workflows often require GPU resources

- The range of how much they require is vast
- Many of them will not fully occupy a datacenter GPU on their own

➔ **Hardware has to be shared to reach maximum utilization**

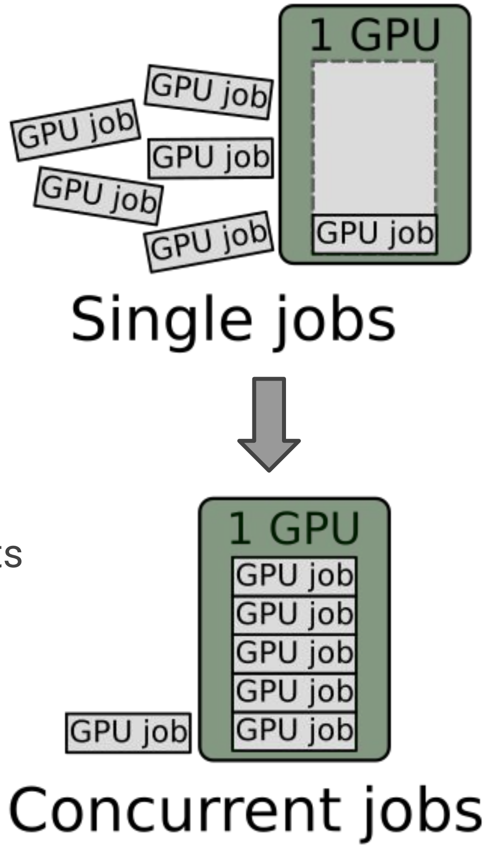
Complex topic for batch systems with GPU resources

- Ideally, one GPU could be shared by multiple users

But...

- No guarantee that resource requirements match resource requests
- Little control over quality of executing software
- Shared resources might influence each other

➔ **Jobs have to be as isolated as possible**



What are we working with?

The TOpAS cluster near the T1 GridKa

- KIT provides GPU resources to CMS, ATLAS and Belle II opportunistically
- There is a mix of V100 and A100 GPUs available
- One full node was used for the tests
  - 8 A100 GPUs and 255 CPU threads

Medium scale machine learning as benchmark

- Training of an event classifier
  - Example from a real HEP workflow
- Up to 12 can fit in GPU memory at once
- Only actual training performance is considered
- Same workload as in the 2022 contribution [1]

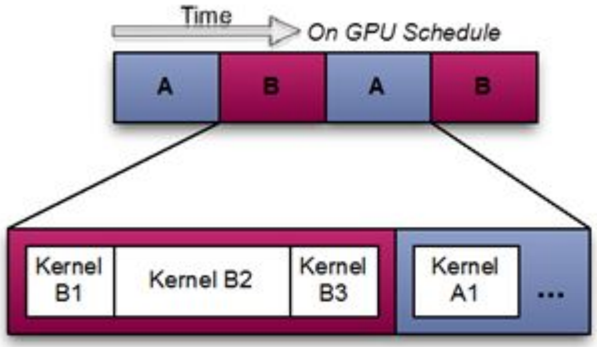
Exact specifications and setup in backup

[1] https://indico.cern.ch/event/1106990/contributions/4991345/

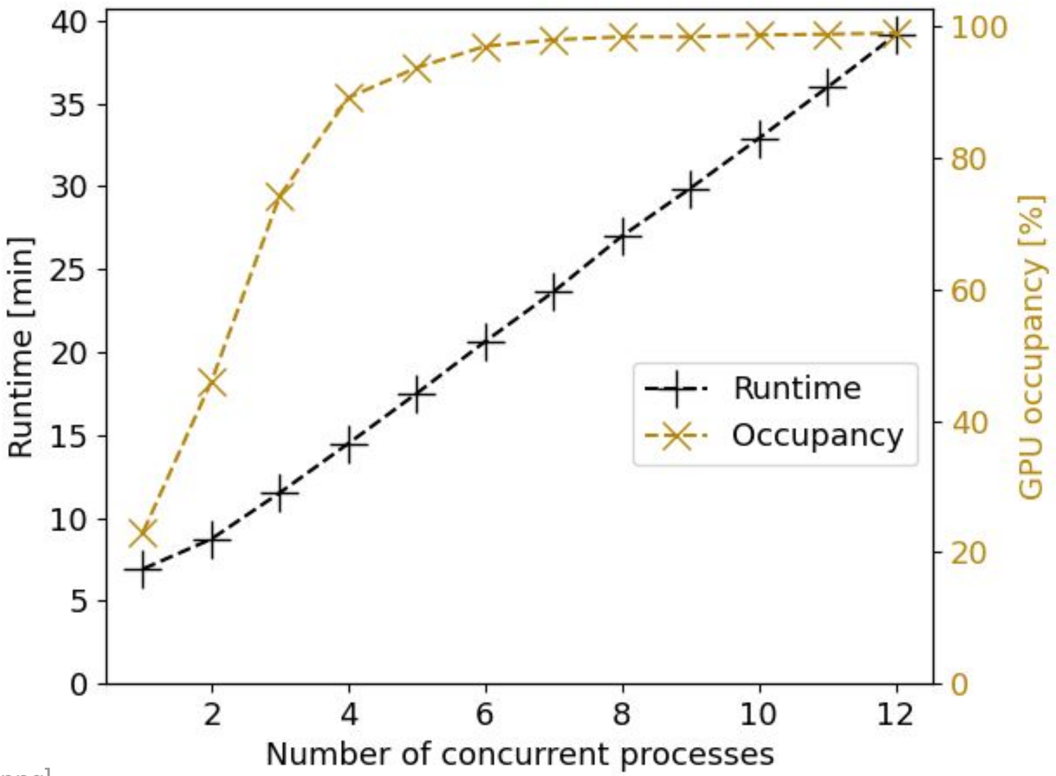
Bad performance with concurrent processes

- Runtime increases close to linearly with the number of processes
- GPU occupation increases sharply when shared by processes

Most GPUs handle concurrent GPU calls sequentially via time slicing



[https://docs.nvidia.com/deploy/mps/topics/media/image3.png]

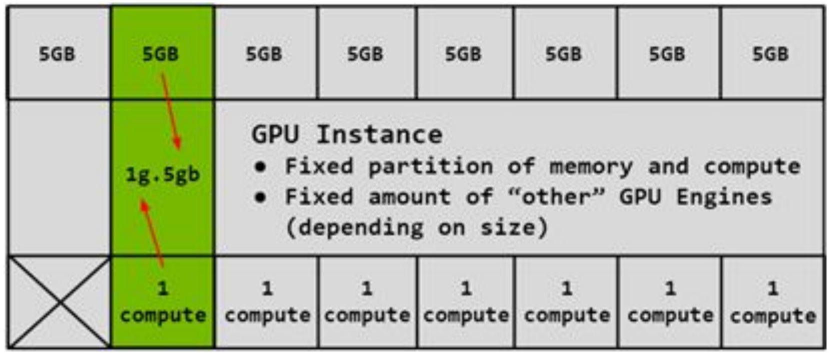


MIG

Multi instance GPU (MIG)

Device setup to split one large GPU into pieces

- Up to seven pieces per GPU
- Each piece acts as its own GPU
- 1/8th of the GPU is reserved for overhead



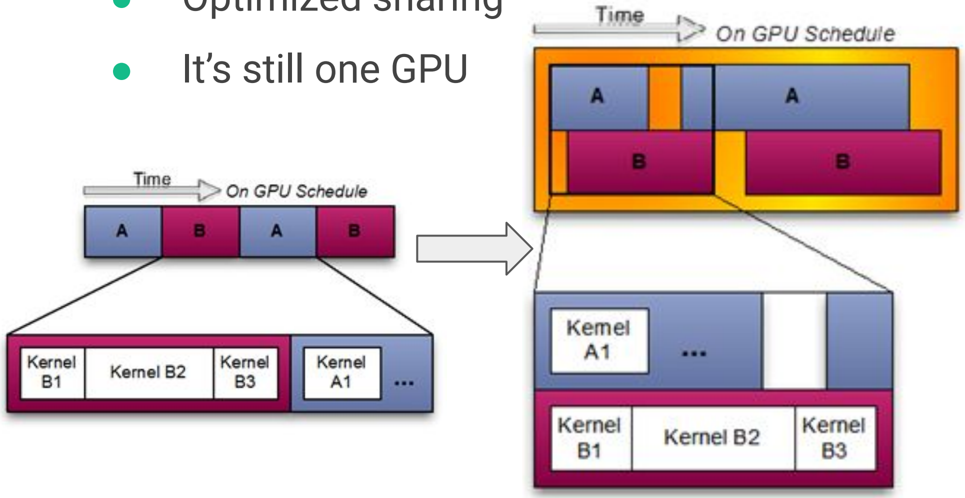
NVIDIA MIG sketch

MPS

Multi-process service (MPS)

Service to optimize concurrent contexts

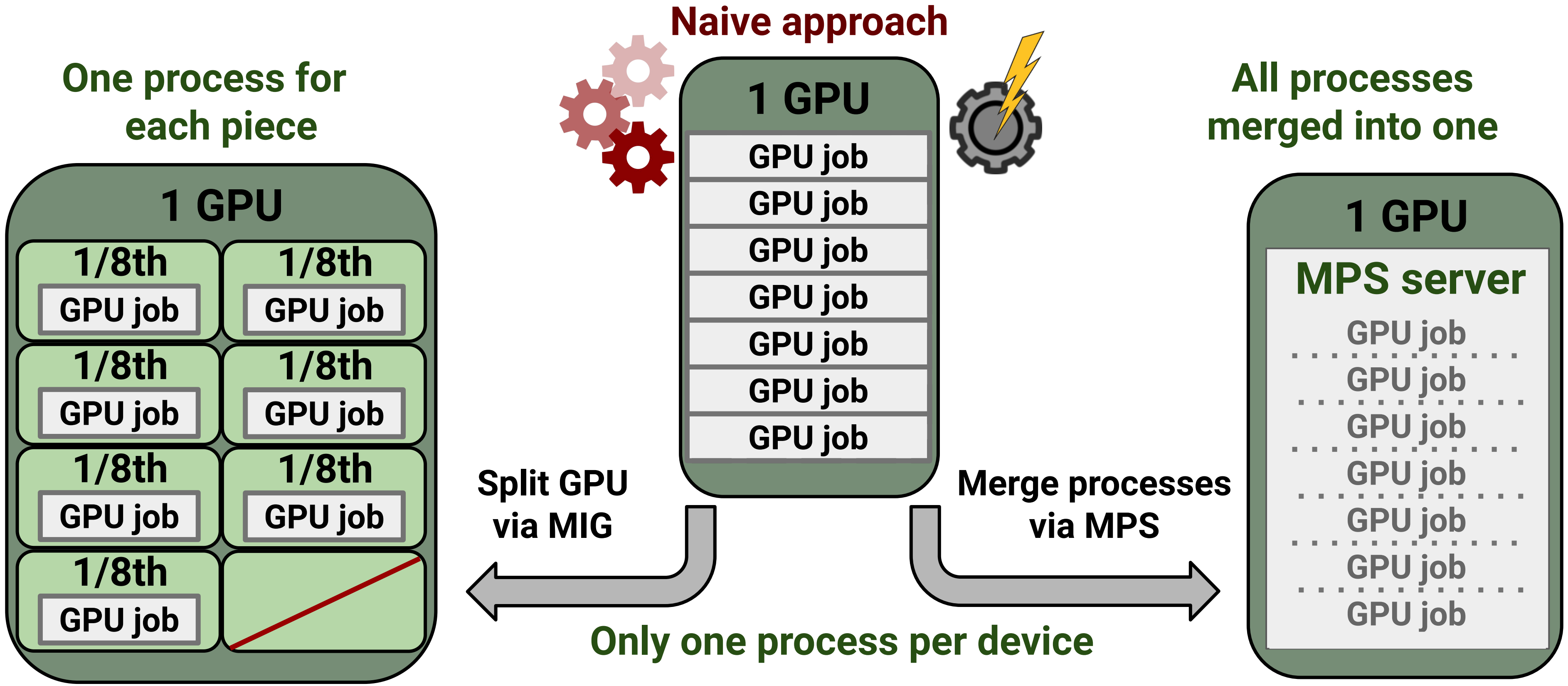
- Summarizes multiple contexts into one
- Optimized sharing
- It's still one GPU



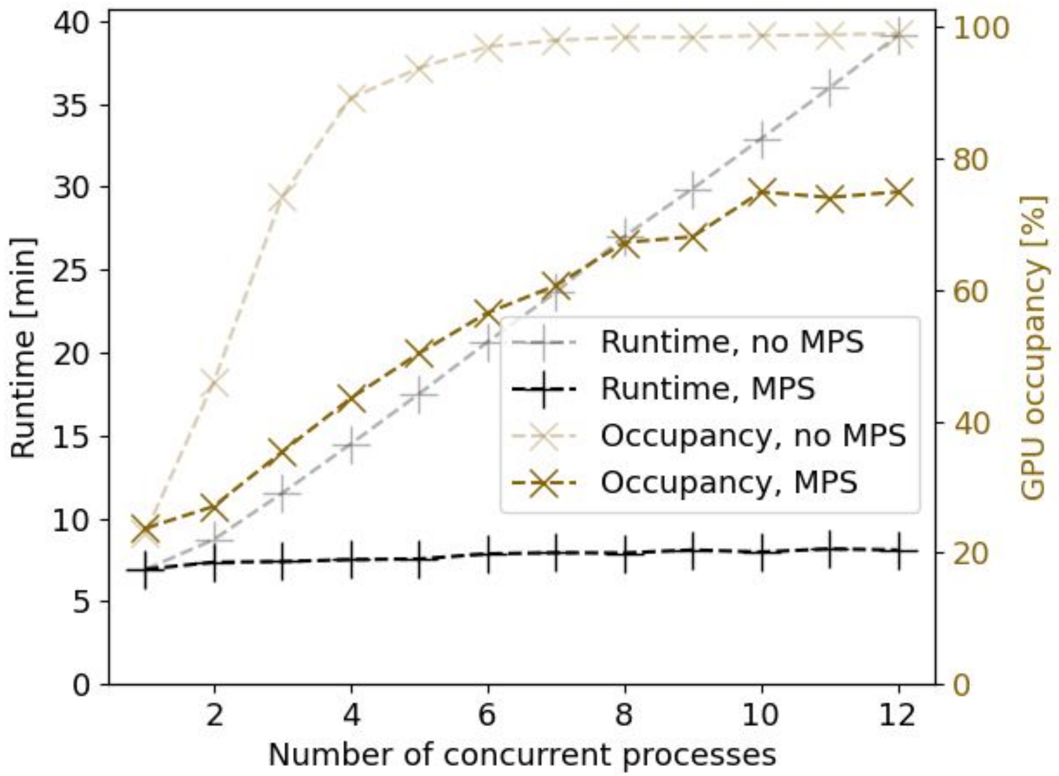
NVIDIA MPS sketch

MIG

MPS


Naive approach
One process for each piece
1 GPU
1/8th
GPU job
1/8th
GPU job
1/8th
GPU job
1/8th
GPU job
1/8th
GPU job
1/8th
GPU job
1/8th
GPU job
1 GPU
GPU job
GPU job
GPU job
GPU job
GPU job
GPU job
GPU job
Split GPU via MIG
Merge processes via MPS
Only one process per device
All processes merged into one
1 GPU
MPS server
GPU job
GPU job
GPU job
GPU job
GPU job
GPU job
GPU job

Better performance with MPS



Runtime stays nearly constant, even with concurrent processes

- Better utilization of GPU capacity
- Only limited by device memory

MPS can have a great impact on the overall throughput

- GPU occupation in line with expected performance

➔ Reported GPU occupation without MPS can be misleading

MPS is beneficial in most cases where a GPU is shared between processes

Performance comparison

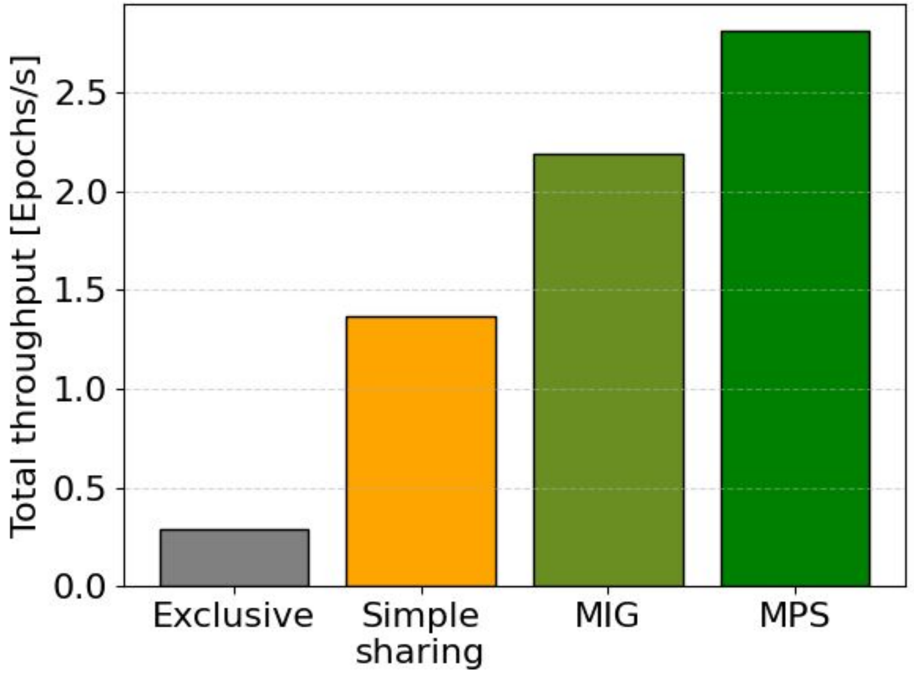
All variants can be used with a batch system

- Small scale tasks occupying a whole GPU is inefficient
- Sharing the GPU improves throughput significantly
- Splitting the GPU with MIG has even higher throughput
- Depending on the task, MPS can be even faster than MIG due to limit of 7 pieces

MIG and MPS generally show best throughput

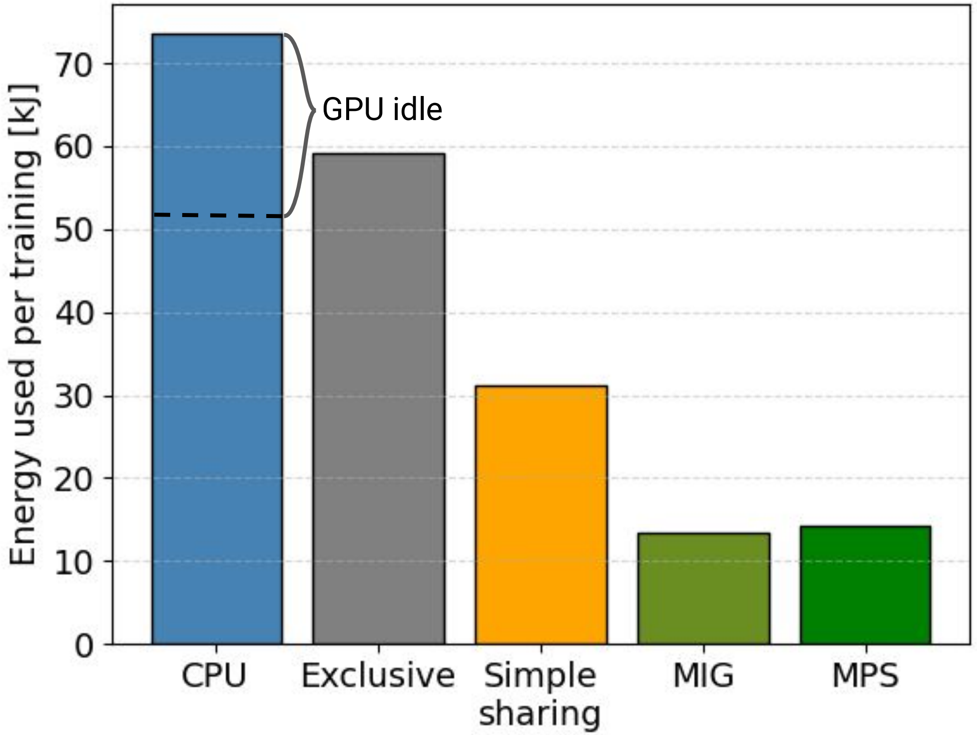
- Exact performance is workload dependent

Throughput of jobs managed by HTC batch system



Energy efficiency comparison

Energy efficiency of various hardware setups



- Power usage of full machine measured over runtime of benchmark

Throughput is related to energy efficiency

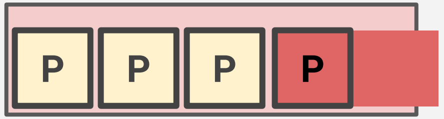
- Slow setups also waste energy
- A GPU with only one small task can have worse energy efficiency than CPU
- Sharing the GPU allows for higher efficiency
- MIG and MPS show best efficiency
  - Around four times better than CPU

Maximum energy efficiency can only be reached by relying on MPS or MIG

Job isolation

MPS provides additional options to isolate GPU-processes compared to simple sharing

- Only fatal GPU failure will influence concurrent processes
- A hard limit for the amount of assignable GPU memory can be set
  - Prevents common issue of one process taking more memory than intended

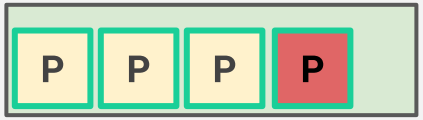


Without limit set by MPS

A bad process can kill everything on GPU

Not safe enough for GPU sharing

With limit set by MPS



A bad process will be stopped by the set limit

Safe with sensible limits

There is no guarantee that users will apply these options on their own

- Some limitations can be set by the batch system itself
  - No perfect way to set a limit on the whole job

Perfect isolation is not possible, even with MPS, but the risks can be reduced

Comparison of setup options

Metric	Exclusive GPU usage	MIG	MPS
Availability	Default use for most GPUs	Available for selected Nvidia Ampere GPUs and newer	Available for Nvidia Volta GPUs and newer
Concurrency	Only one process allowed	Up to seven processes per GPU with prior configuration	No hard limit
Flexibility	Exactly one slot	Reconfiguration is difficult during active work	No predefined slots or limits Custom limits can be set
Isolation	Perfectly isolated	Great isolation between pieces	Not fully isolated Most issues handled on a per-process basis
Monitoring	All monitoring can be performed on a per-job basis	Utilization and memory can be monitored separately	Only total GPU metrics can be monitored

Summary and outlook

An HTCondor setup using MPS to share GPUs was successfully tested

- All requirements of MPS proved to be feasible in a batch environment
- Performance is as good as expected
- Jobs are mostly isolated, but fatal GPU errors will influence other jobs
    - Over-assignment of GPU memory can be caught before fatal error

Some limitations still remain

- Currently no easy way to allow for multi-GPU and split-GPU jobs at the same time
    - HTCondor signaled that there will be limited support in the future
- GPU memory limitation not yet totally fail-safe
- No straightforward way to monitor resource usage of shared GPU

Backup

The HTCondor MPS setup

- One MPS daemon exists for each GPU
  - assigned via **CUDA_MPS_PIPE_DIRECTORY**
- Jobs can request GPU memory as a custom resource via the HTCondor job submission file
- The memory resource is managed by using partitionable slots
  - One partitionable slot for each GPU
  - Does not work with multi-GPU jobs, but there are plans from HTCondor for this
- Job wrapper script detects this request and adds necessary env variables to exec env
  - **CUDA_MPS_PIPE_DIRECTORY** for demon assignment
  - **CUDA_VISIBLE_DEVICES** for GPU assignment
  - **CUDA_MPS_PINNED_DEVICE_MEM_LIMIT** for process specific memory limit
    - User can overwrite this if they want to use multiple contexts in the same job
- External monitoring script tracks total GPU memory limit (sum of all limits spawned by job)
  - Removes jobs that break requested GPU memory limit
  - Not 100% safe, but very short dead time (~1 s)

Useful MPS commands

- How to start MPS software
  - **nvidia-cuda-mps-control -d**
- Set alternative MPS pipe/socket directory (also has to be set for processes on running GPU)
  - **CUDA_MPS_PIPE_DIRECTORY=<GPU-uuid> nvidia-cuda-mps-control -d** or **<Process>**
- Assign only specific GPUs to MPS software (will reorder ids of GPUs, e.g. 1,3,6 → 1,2,3)
  - **CUDA_VISIBLE_DEVICES=<GPU-uuid> nvidia-cuda-mps-control -d**
- How to stop MPS software
  - **echo “quit” | nvidia-cuda-mps-control**
- How to limit available GPU memory
  - **CUDA_MPS_PINNED_DEVICE_MEM_LIMIT="<GPU-id>=<Memory-limit>" <Process>**
- More information: https://man.archlinux.org/man/extra/nvidia-utils/nvidia-cuda-mps-control.1.en
  https://docs.nvidia.com/deploy/mps/index.html

Notes on MPS

- Only one user on a system may have an active MPS server.
  - The MPS control daemon will queue MPS server activation requests from separate users, leading to serialized exclusive access of the GPU between users regardless of GPU exclusivity settings.
  - In a batch environment, all jobs can be executed as the same user (e.g. “nobody”)
- All MPS client behavior will be attributed to the MPS server process by system monitoring and accounting tools (e.g., nvidia-smi, NVML API).
- One MPS daemon can only manage around 40 contexts before performance degrades
  - Can be alleviated by using multiple daemons (one per GPU)
- **CUDA_MPS_PINNED_DEVICE_MEM_LIMIT** only limits the available memory for one context
  - It’s still possible to over-allocate memory with multiple improper contexts
- MPS will reorder indices of assigned GPUs (0,3,4 -> 0,1,2), UUIDs stay the same
- More information: https://man.archlinux.org/man/extra/nvidia-utils/nvidia-cuda-mps-control.1.en
  https://docs.nvidia.com/deploy/mps/index.html

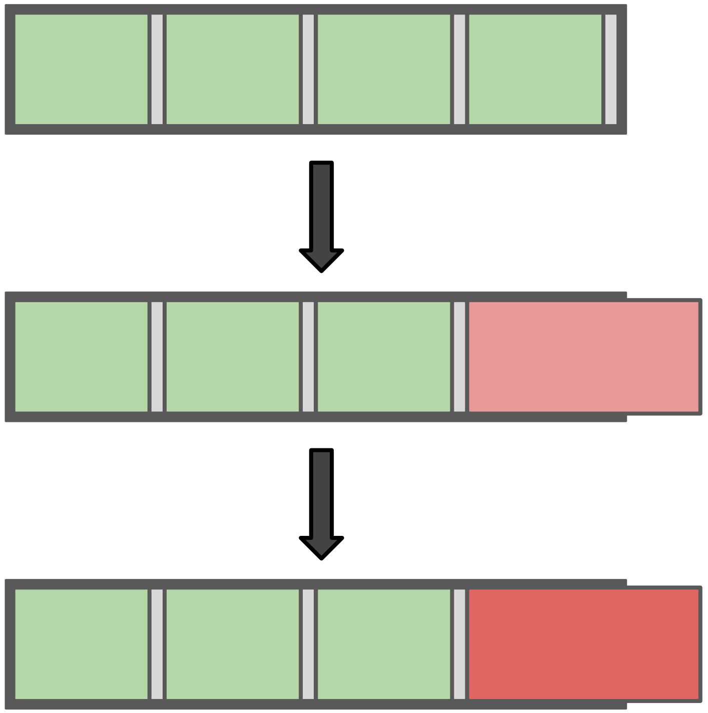
GPU “Out of memory” (OOM) crash

Default GPU behaviour

1. Memory is allocated from the entire scope
2. One process tries to allocate beyond the scope of total available memory
3. All processes on the GPU die due to OOM

With MPS memory limit

1. Memory is allocated from the assigned scope
2. One process tries to allocate beyond the scope of available memory
3. Only the specific processes that tried to over allocate dies due to OOM



Benchmark information

The machine:

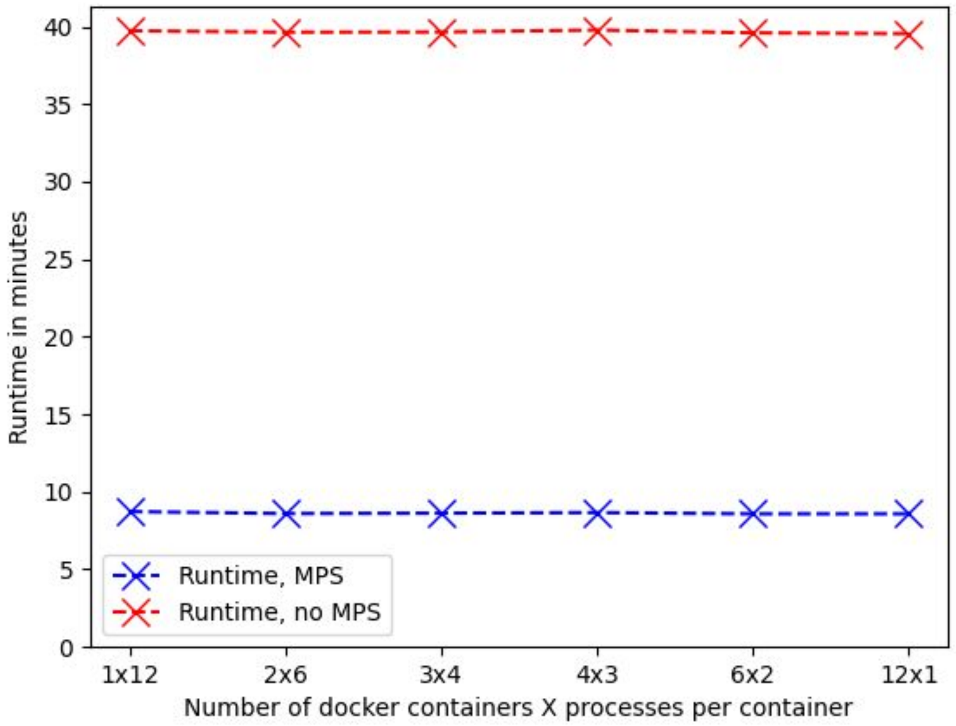
- One node of the TOpAS cluster
- 255 CPU-threads of two AMD EPYC 7662 CPUs
  (2 CPU threads per training used)
- 8 NVIDIA A100 GPUs (One GPU was used for all trainings)

The workload:

- Training of fully connected feed forward neural network
- 14 input variables
- ~2 Million input samples
- 3 hidden layers with 512 nodes each
- 6 output classes
- 600 samples per balanced batch
- ¾:¼ split between training and validation data
- Ran for 100 epochs each

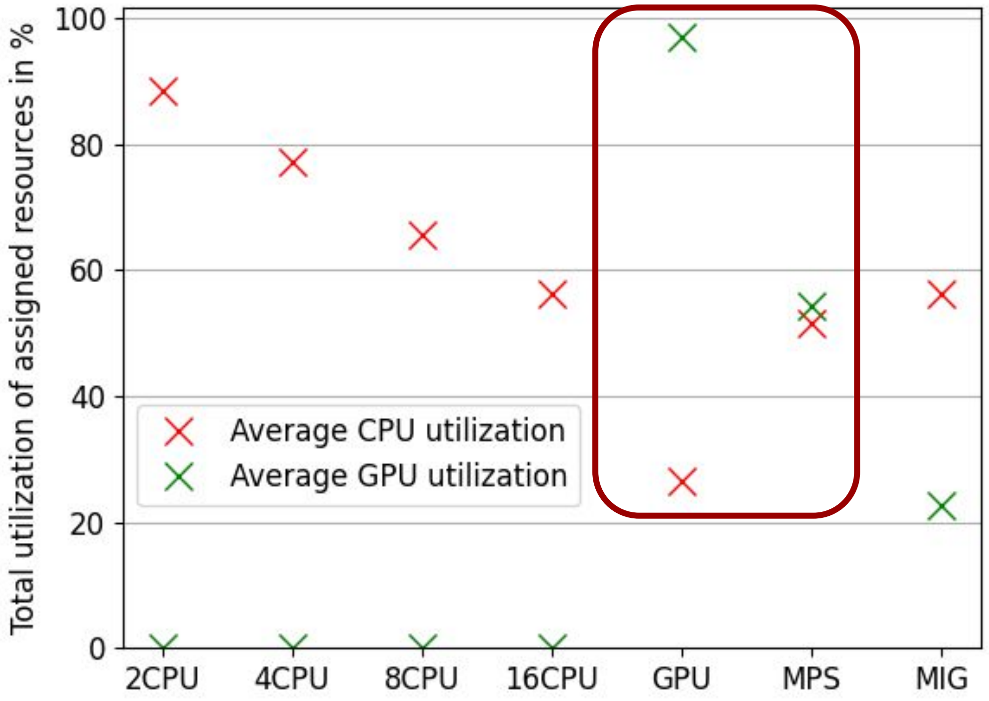
MPS with docker

- 12 concurrent processes distributed among a number of docker containers with and without MPS
- No difference between the different distributions
- MPS is able to function through docker containers
- **--icp="host"** has to be set for the docker containers



Utilization

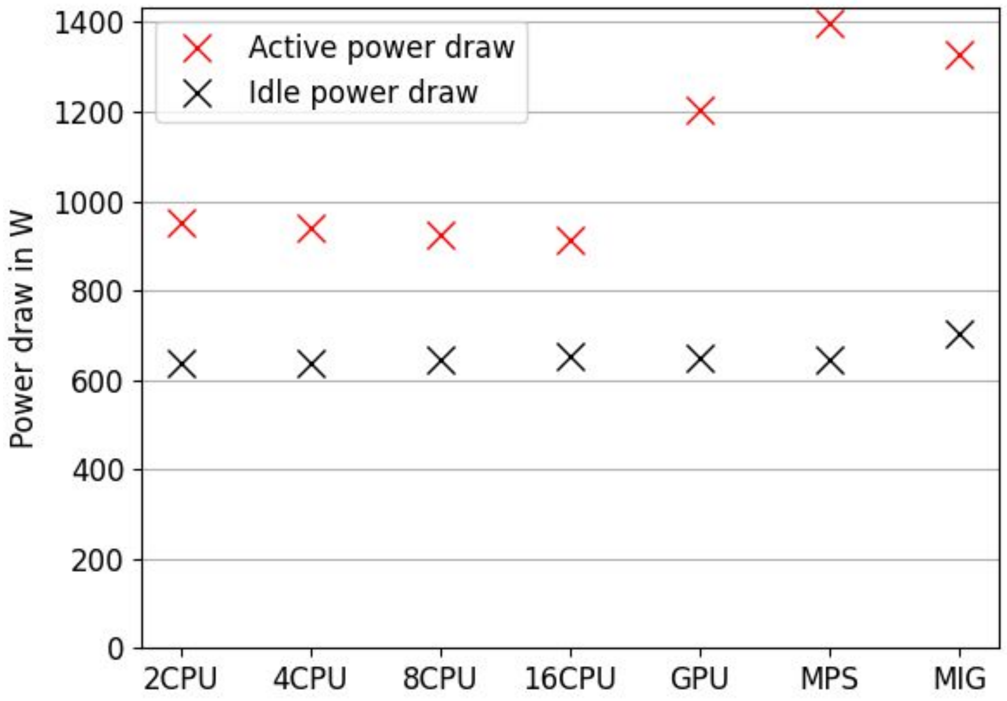
- High CPU utilization for low degree of parallelism
- Utilization of CPU decreases with increasing parallelism
- Pure GPU variant is limited due to GPU occupation
- MPS and MIG variant are not limited in this way
- Less GPU utilization as there are fewer trainings on the same hardware as MPS



Power draw

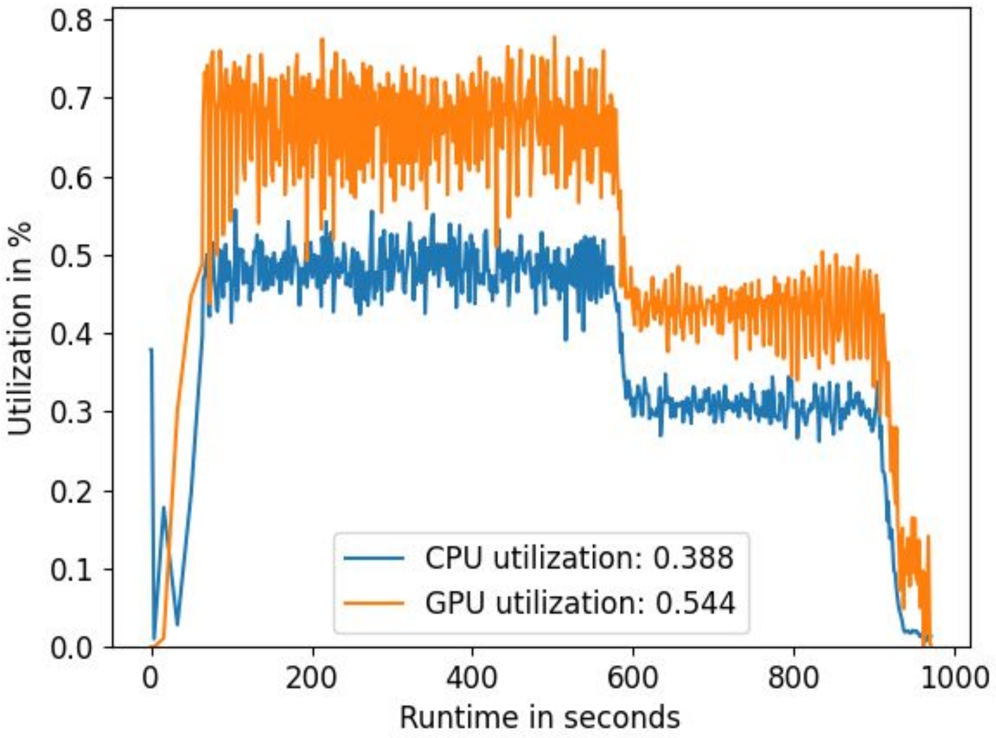
All power measurements have a systematic error of 5%

- The idle power draw is the same for every variant except MIG
- The active power draw for the CPU variants differs only slightly
- The GPU variants draw more power in accordance with their performed work



Limits of MPS

- High number (>40) of contexts lead to declining performance
- Multiple Daemons improve things
- Leads to questionable performance patterns
  - Not clear what causes this



Second Benchmark: Simulation

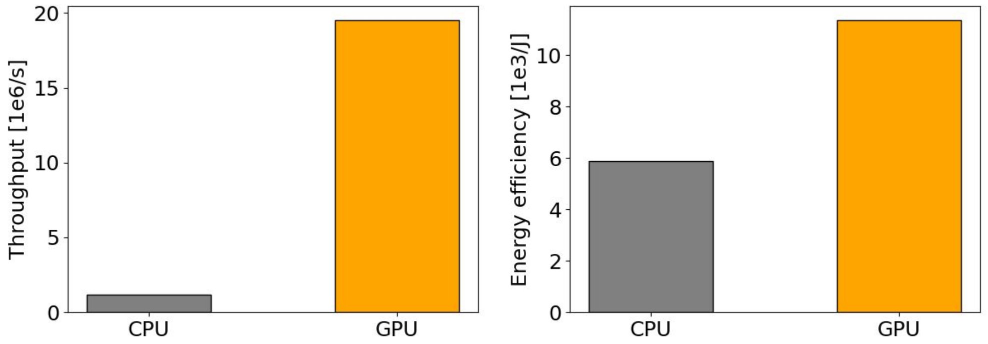
Madgraph based benchmark for simulation on CPU and GPU

- Candidate for future GPU HEPScore benchmark [2]
- One of the main tasks performed in HEP computing

Same machine as before, run at maximum capacity

- Only achievable with MPS

[https://github.com/madgraph5]



[mg5amc-madgraph4gpu-2022-bmk:v0.7]

- Compared to vectorized CPU setup
- Throughput increased by a factor of 17
- Energy efficiency increased by ~2

Significant improvement over best CPU result

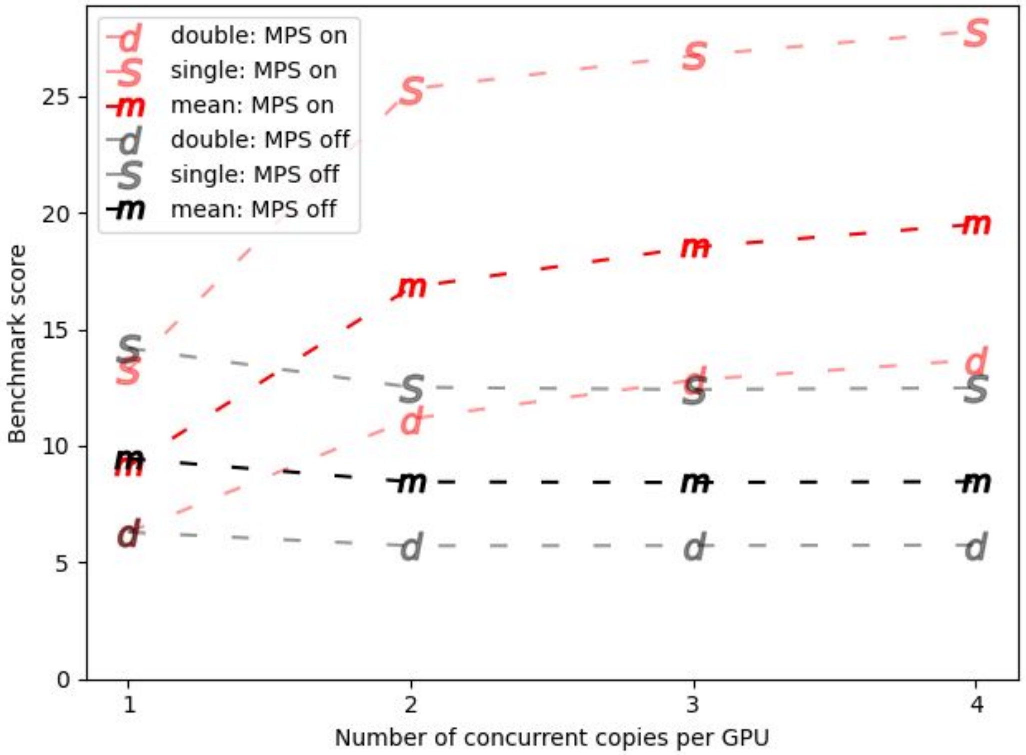
[2] https://indico4.twgrid.org/event/14/contributions/325/

Second Benchmark: Throughput

The benchmark recommends only one concurrent copy when using a GPU

- Multiple copies per GPU are less performant when not using MPS
- They are more performant when using MPS
- The change is workload dependent

Variants with >1 copies per GPU were considered in the benchmark


double: MPS on
single: MPS on
mean: MPS on
double: MPS off
single: MPS off
mean: MPS off
Benchmark score
Number of concurrent copies per GPU