Experiences with concurrent use of GPU at KIT

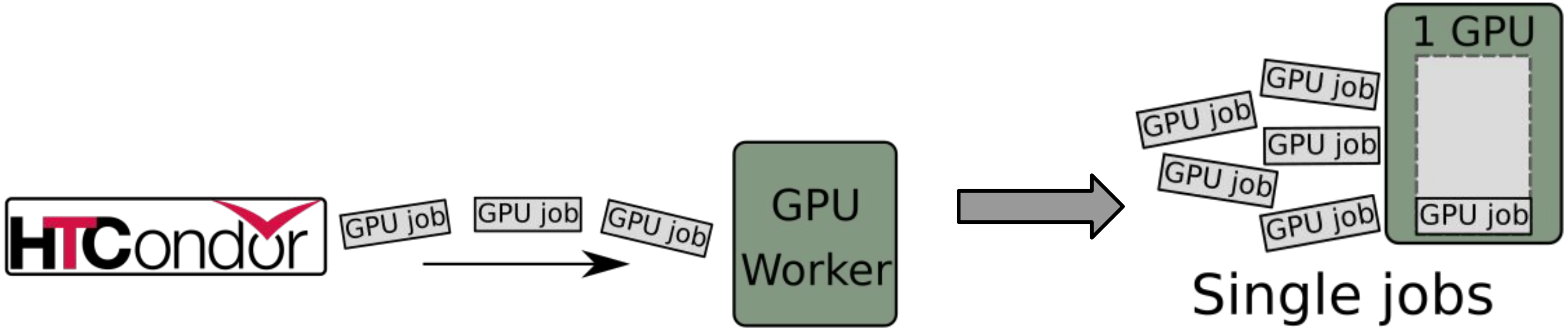
Tim Voigtländer, Manuel Giffels
tim.voigtlaender@kit.edu

Karlsruher Institut für Technologie (KIT) - Institut für experimentelle Teilchenphysik (ETP)

Motivation

Use of GPU is becoming more widespread in high energy physics

- Provision of GPU resources through batch systems is an important topic
- The range of necessary resources per job is large
  - Some Applications need more than one GPU to run in a reasonable amount of time
  - Some applications do not fill a GPU on their own

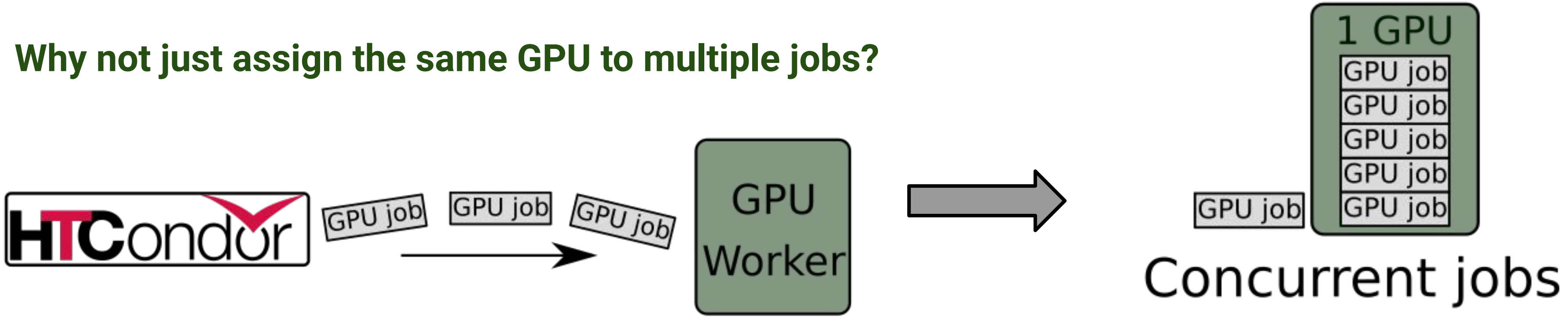


Motivation

Use of GPU is becoming more widespread in high energy physics

- Provision of GPU resources through batch systems is an important topic
- The range of necessary resources per job is large
  - Some Applications need more than one GPU to run in a reasonable amount of time
  - Some applications do not fill a GPU on their own

Why not just assign the same GPU to multiple jobs?

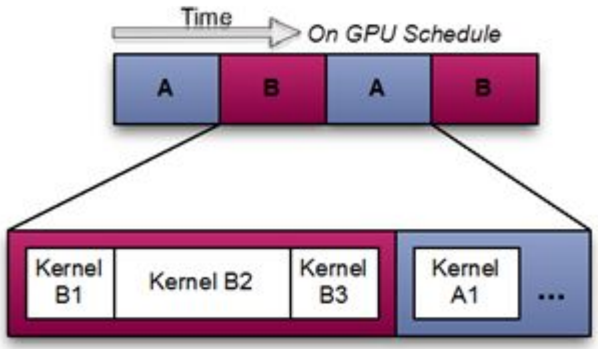


Concurrent jobs on one GPU

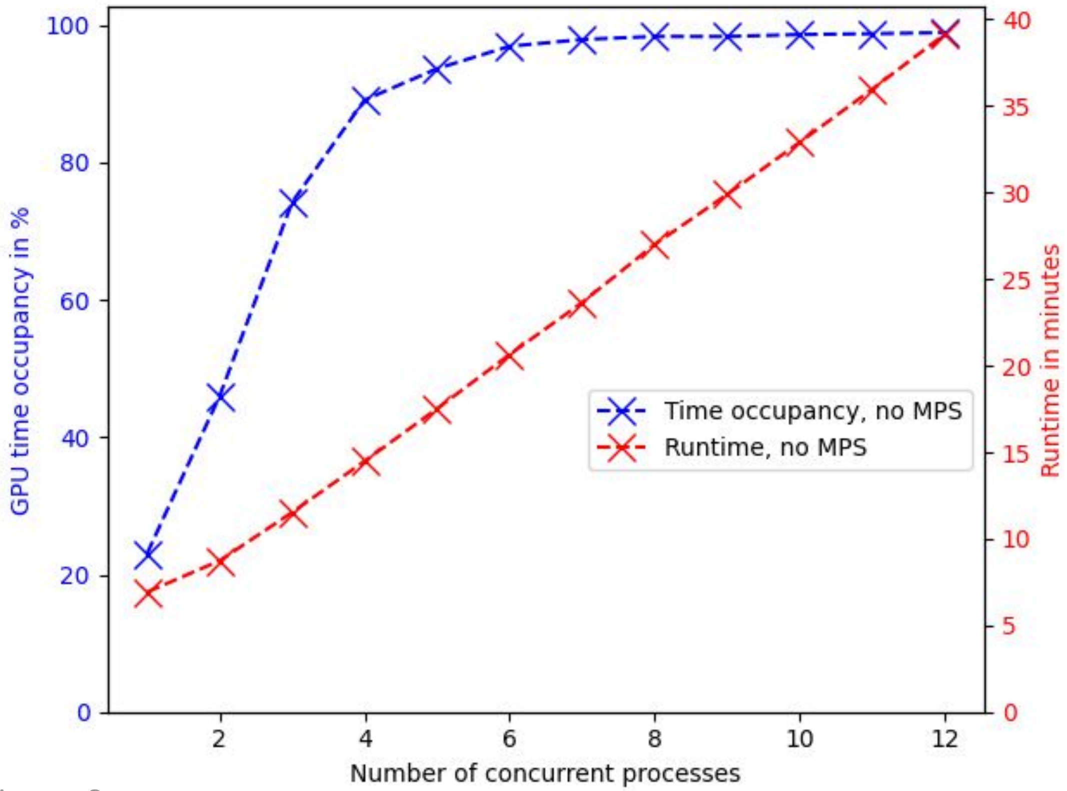
There are two issues with placing multiple jobs on the same GPU:

Bad performance with concurrent processes

- GPU time occupation increases sharply with multiple processes
- Runtime also increases linearly with the number of processes
- **Concurrent GPU calls are normally ran sequentially**



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



Concurrent jobs on one GPU

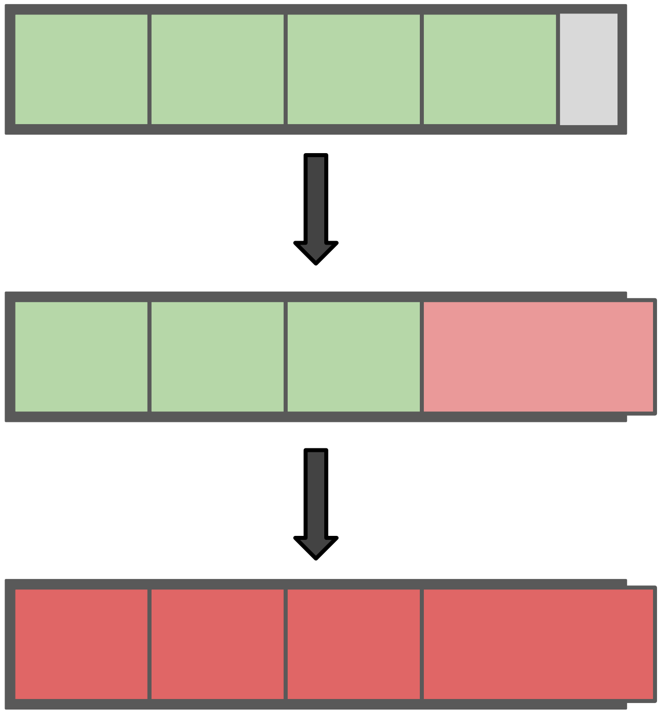
There are two issues with placing multiple jobs on the same GPU:

1. Default implementation for concurrent GPU applications has bad performance
   - Processes are basically run in sequence instead of concurrent

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



Concurrent jobs on one GPU

There are two issues with placing multiple jobs on the same GPU:

1. Default implementation for concurrent GPU applications has bad performance
   - Processes are basically run in sequence instead of concurrent

2. Issues with “Out of memory” crashes when GPU device memory is exceeded
   - Processes on shared GPU will collectively crash if memory limit is exceeded
   - One bad job will cause others to fail as well

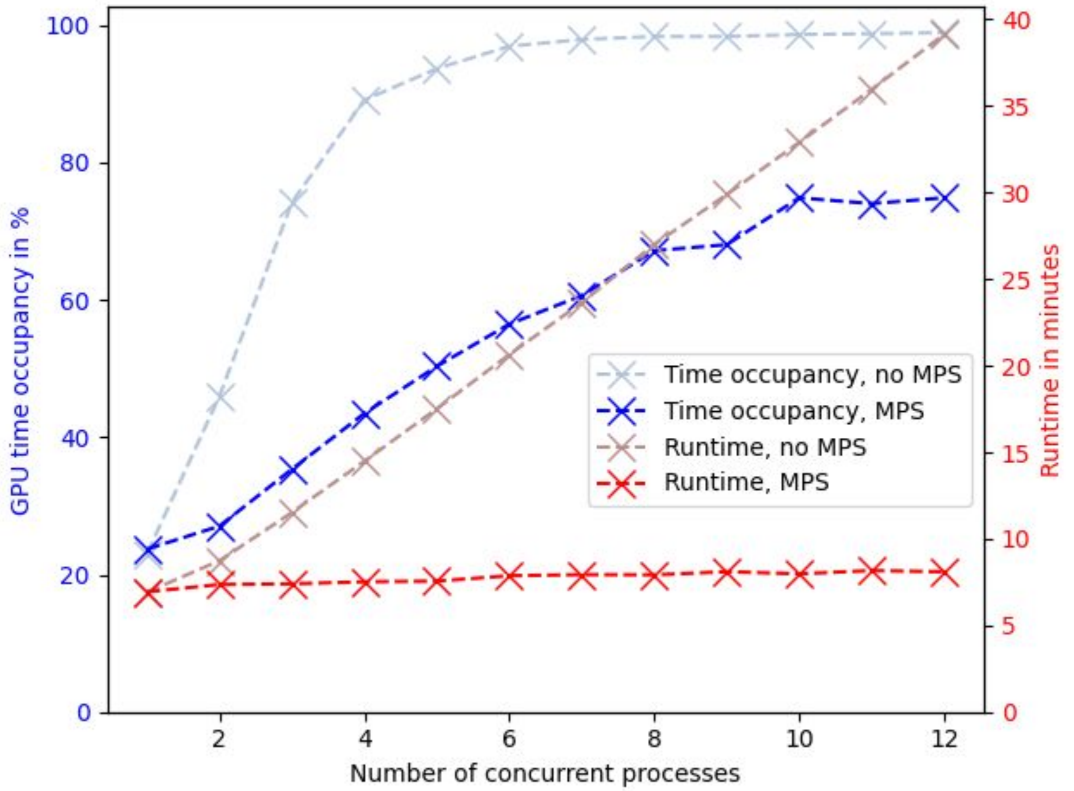
Options for concurrent jobs

What can be done to fix these issues?

- **Buy smaller hardware**
  - Leads to issues with large GPU jobs and already bought hardware
- **Build something ourselves**
  - Very costly and probably impossible in general
- **Use existing solutions made by hardware producers**
  - NVIDIA Multi-instance GPU (MIG) — Split one GPU into multiple
    - Very limited in which models can use it
  - NVIDIA Multi-process service (MPS) — Optimize concurrent execution
    - Less limited than MIG

Better performance with concurrent processes and MPS

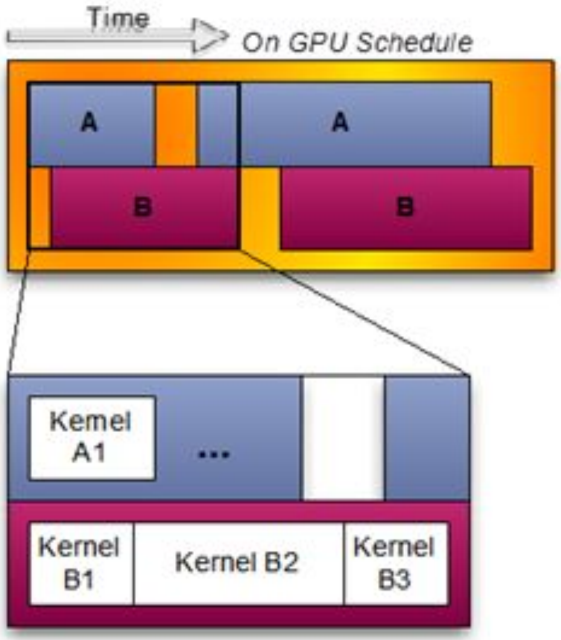
- GPU time occupation rises significantly slower
- Runtime stays nearly the same
  - ➔ **GPU occupation observed without MPS is inflated**
- Other metrics like power consumption also indicate this



Better performance with concurrent processes and MPS

- GPU time occupation rises significantly slower
- Runtime stays nearly the same
  - ➔ **GPU occupation observed without MPS is inflated**
- Other metrics like power consumption also indicate this

MPS achieves true concurrency between processes on a GPU



https://docs.nvidia.com/deploy/mps/topics/media/image4.png

What does this mean?

Without MPS

- Adding more concurrent processes increases runtime linearly
- The GPU is underutilized regardless of visible utilization metrics

With MPS

- All concurrent MPS processes are treated as one big process by the GPU
- Adding more concurrent processes doesn’t increase runtime
  as long as the GPU is not full occupied
- Utilization metric shows true values

Depending on the number of concurrent processes, speedup of over >5 is possible

Concurrent jobs on one GPU

There are two issues with placing multiple jobs on the same GPU:

1. Default implementation for concurrent GPU applications has bad performance
    - Processes are basically run in sequence instead of concurrent

    MPS optimizes processes to execute concurrently, leading to the expected performance

2. Issues with “Out of memory” crashes when GPU device memory is exceeded
    - Processes on shared GPU will collectively crash if memory limit is exceeded
    - One bad job will cause others to fail as well

Memory limitation with MPS

It can be challenging to prevent processes from over allocating memory on GPU

- It has to be actively checked which process is using how much memory
  - Over allocation can cause errors before it is detected and resolved
  - Short bursts of over allocation might not be detected at all
- Keeping a buffer reduces overall efficiency

MPS can be used to assign maximum GPU memory limits to each MPS-managed process

- Hard limit to the amount of allocatable GPU memory
  - Over allocation leads to OOM error only for that specific process
  - Limit can be set individually for each process
- Total memory can still be over allocated if processes are run with improper limits

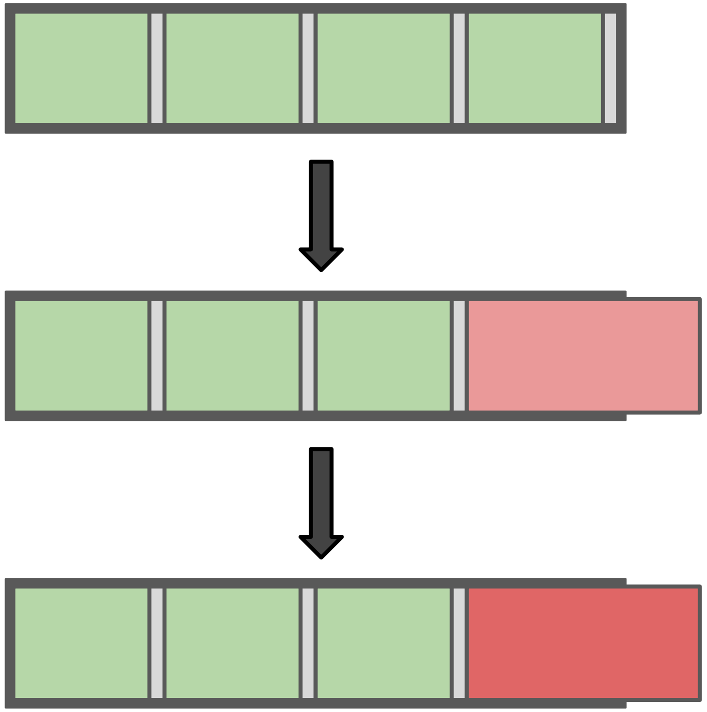
GPU “Out of memory” (OOM) crash

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



Concurrent jobs on one GPU

There are two issues with placing multiple jobs on the same GPU:

1. Default implementation for concurrent GPU applications has bad performance
    - Processes are basically run in sequence instead of concurrent

    MPS optimizes processes to execute concurrently, leading to the expected performance

2. Issues with "Out of memory" crashes when GPU device memory is exceeded
    - Processes on shared GPU will collectively crash if memory limit is exceeded
    - One bad job will cause others to fail as well

    MPS allows hard limits to GPU memory allocation that is isolated from other processes

Summary

- GPUs in batch systems can only be fully utilized by small jobs if they are shared between jobs
- Sharing GPUs has two issues
  1. Bad performance of concurrent processes on GPU
  2. Possible GPU memory over-allocation, leading to the failure of multiple jobs
- There are existing solutions for both problems, like MPS
- Basic MPS setup is very simple
  - More information about the setup in the backup slides

MPS is a promising approach to solve the issues that exist with sharing GPUs in batch systems

Outlook

- **Prototype of HTCondor setup with MPS**
  - MPS process run by host
  - Memory limitations set through class ad
  - GPU memory management by host
- **Check for possible issues**
  - Are some of the requirement detrimental
  - Does it work for all relevant hardware and software
  - Will memory limitation per process be enough
  - etc.
- **Compare to other solutions like MIG or completely custom ones**

FAQ

- **Does this work with docker?**
  - Yes, if **--icp="host"** is used
- **How well does it scale?**
  - Around 50 processes the performance deteriorates
  - Can be alleviated by running multiple MPS servers
- **How well are the processes isolated?**
  - They can still crash if the total GPU memory is over-allocated
  - The memory limitation only ensures that individual processes are kept under control
- **For which GPUs does MPS work?**
  - Documentation claims it to work for all NVIDIA GPUs of Volta architecture and later
- **Are there any other requirements?**
  - A GPU process will only attempt to use the server started by the same user id

Thank you for your attention

Backup

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

Benchmark information

The machine:

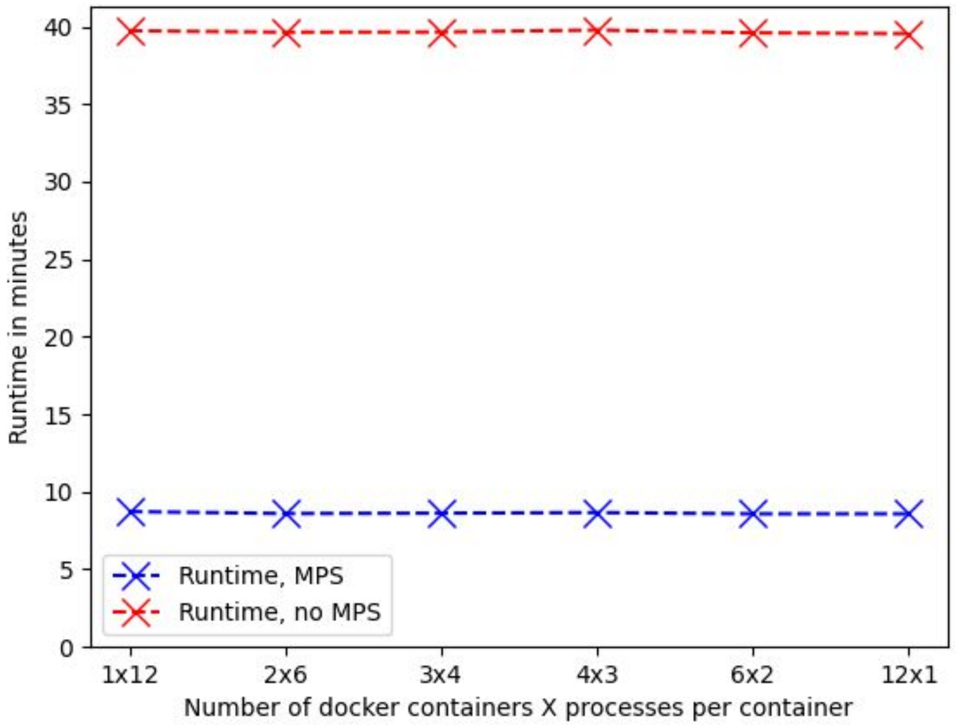
- One node of the TOpAS cluster
- 255 CPU-threads of two AMD EPYC 7662 CPUs
  (2 CPU threads per training used)
- 8 NVIDIA A100 GPUs (One GPU was used for all trainings)

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

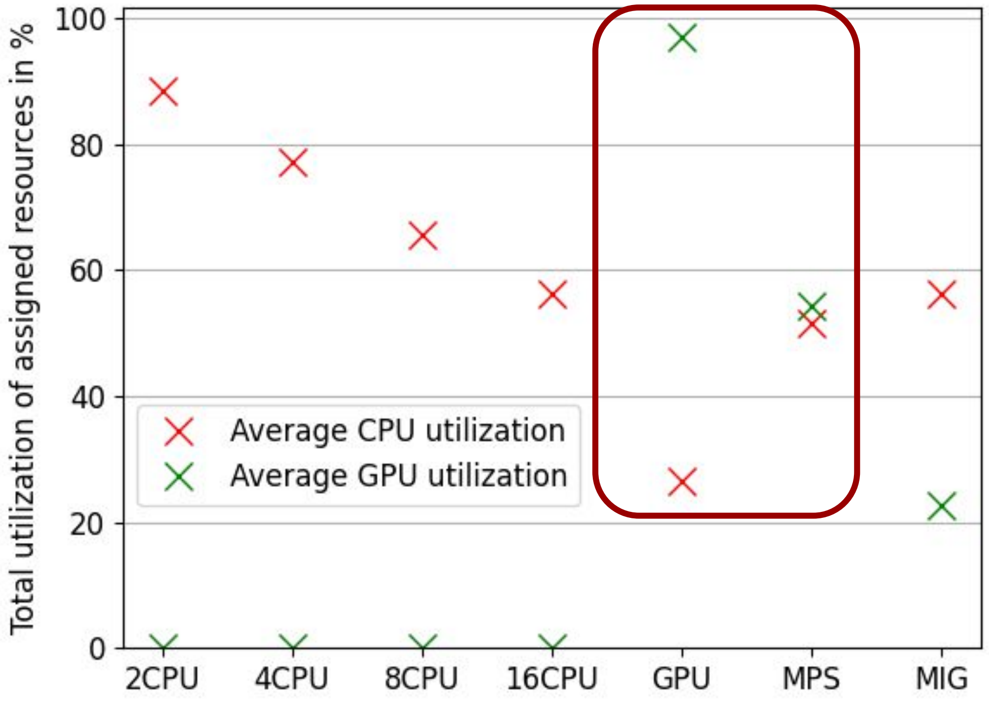
MPS with docker

- 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

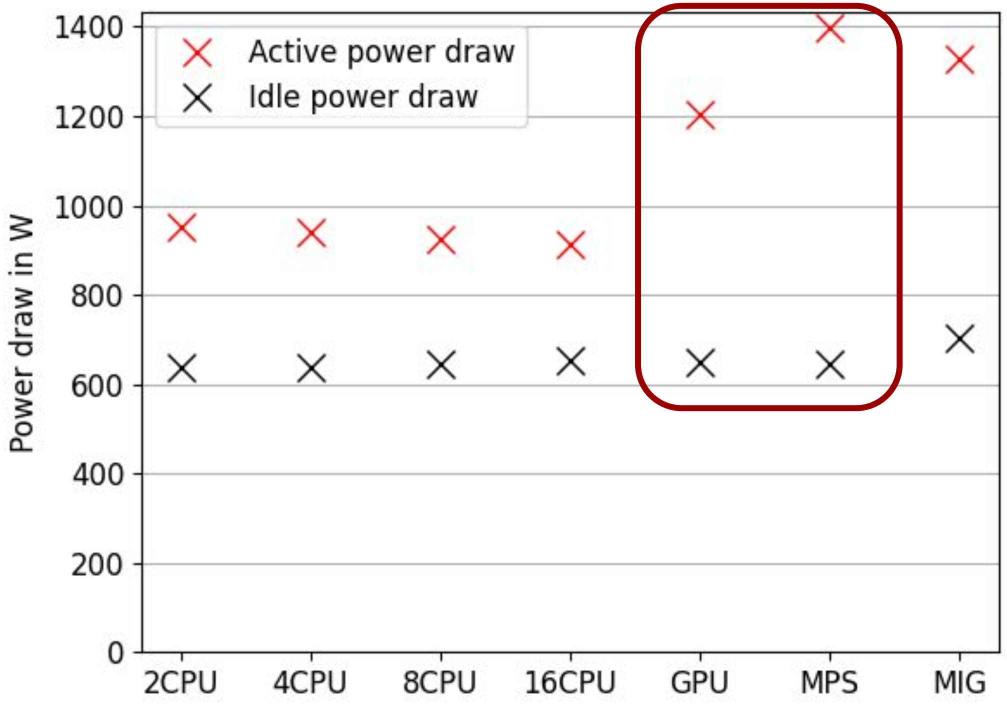
- High CPU utilization for low degree of parallelism
- Utilization of CPU decreases with increasing parallelism
- Pure GPU variant is limited due to GPU occupation
- MPS and MIG variant are not limited in this way
- Less GPU utilization as there are fewer trainings on the same hardware as MPS



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


Active power draw
Idle power draw
Power draw in W
2CPU
4CPU
8CPU
16CPU
GPU
MPS
MIG