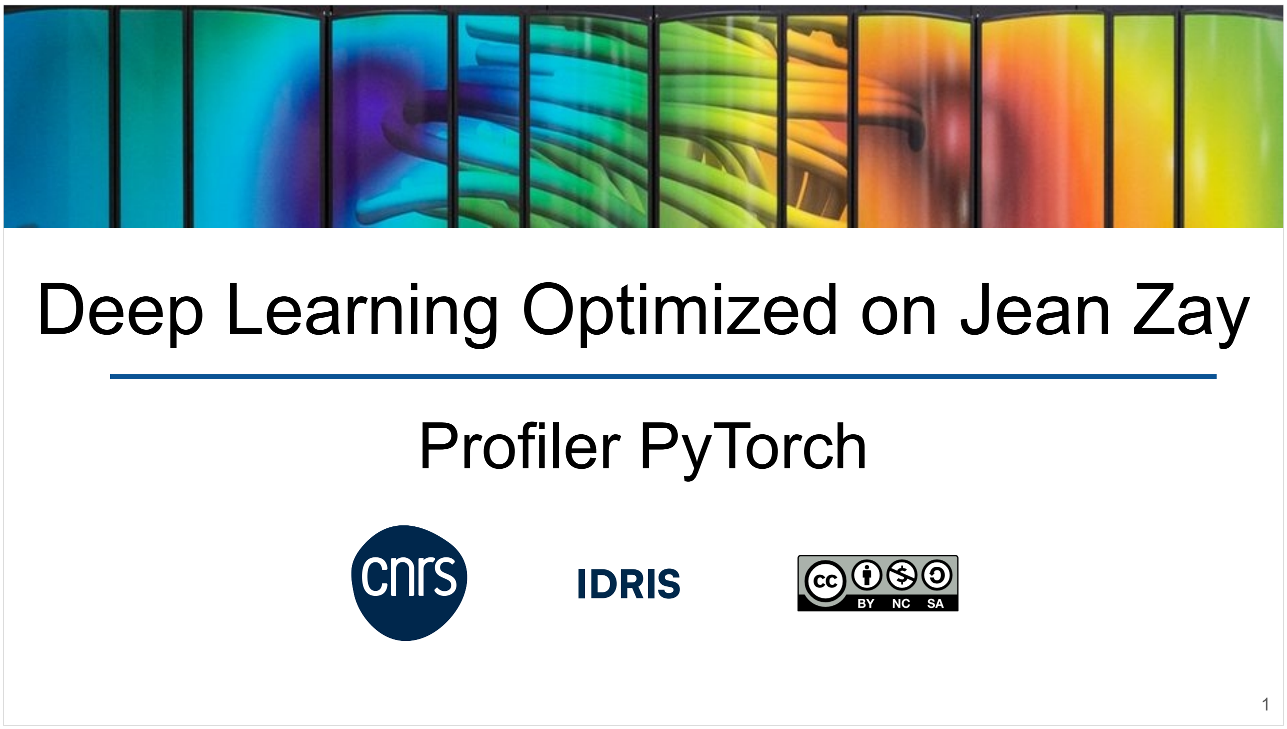


Deep Learning Optimized on Jean Zay

Profiler PyTorch

DLO-JZ course
Commented slides
Author: Myriam Peyrounette
Updated February 2024

PyTorch Profiler

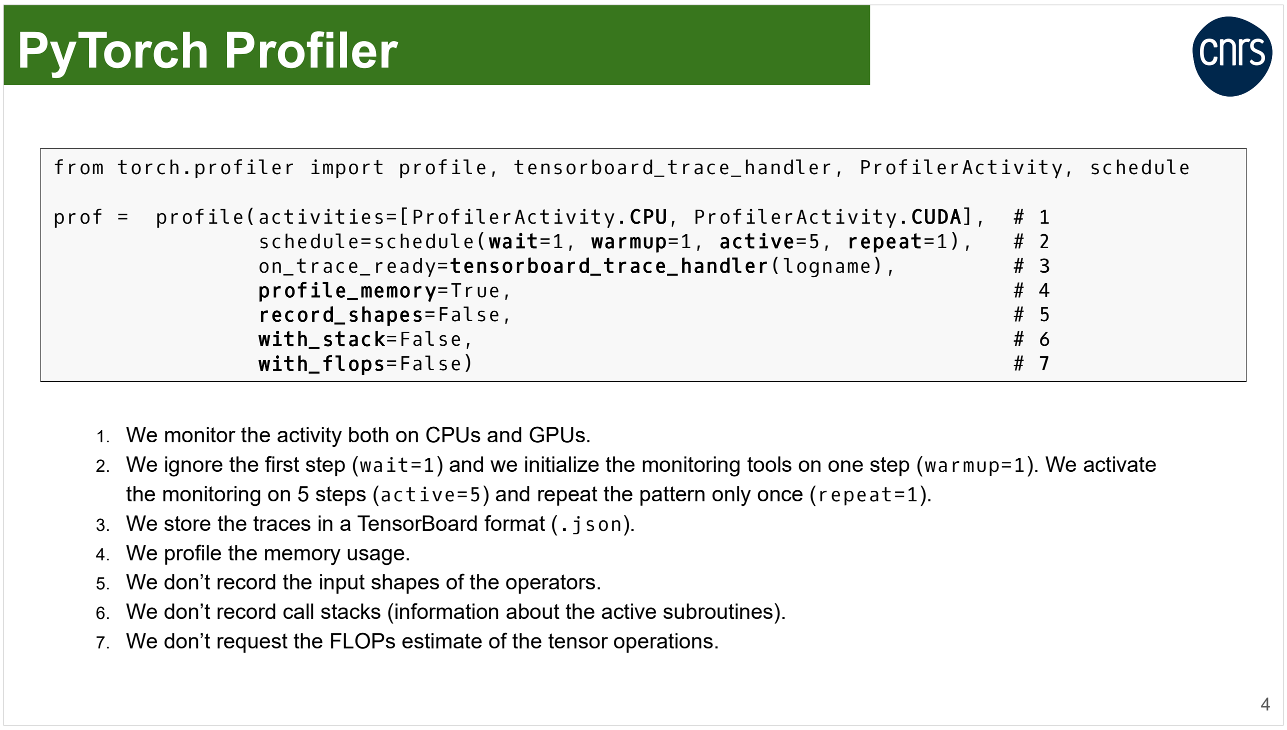
In this chapter, we will learn how to use the profiler implemented in PyTorch. We will visualize the traces generated with TensorBoard.

PyTorch Profiler

- We use a profiler to monitor an execution.
- It allows us to know the **time** and **memory** consumed by each part of the code.
- The results returned by the profiler point to the weaknesses of our code and tell us which parts we should **optimize** in priority.
- The profiler is a wrapper which records various information during the execution of the code.

⚠ This could be slowed down depending on the requested traces. We usually monitor only **a few training steps**.

```
with prof:
    for epoch in range(0,args.epochs):
        for i, (images, labels) in enumerate(train_loader):
            [...]
            prof.step()
```



PyTorch Profiler

```
from torch.profiler import profile, tensorboard_trace_handler, ProfilerActivity, schedule

prof =  profile(activities=[ProfilerActivity.CPU, ProfilerActivity.CUDA],  # 1
                schedule=schedule(wait=1, warmup=1, active=5, repeat=1),    # 2
                on_trace_ready=tensorboard_trace_handler(logname),          # 3
                profile_memory=True,                                        # 4
                record_shapes=False,                                        # 5
                with_stack=False,                                           # 6
                with_flops=False)                                           # 7
```

1. We monitor the activity both on CPUs and GPUs.
2. We ignore the first step (`wait=1`) and we initialize the monitoring tools on one step (`warmup=1`). We activate the monitoring on 5 steps (`active=5`) and repeat the pattern only once (`repeat=1`).
3. We store the traces in a TensorBoard format (`.json`).
4. We profile the memory usage.
5. We don't record the input shapes of the operators.
6. We don't record call stacks (information about the active subroutines).
7. We don't request the FLOPs estimate of the tensor operations.

The different input arguments of the profiler are detailed here.

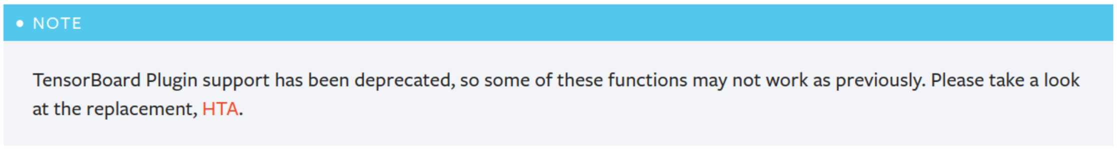
If **`record_shapes`** is activated, a new column containing the operators input shapes appears in the Operator View tab (select “Group By: Operator + Input Shape”).

By activating the **`with_stack`** option, call stacks are recorded and displayed in the Operator View tab (last column). This also improves the timeline readability. In return, it significantly increases the size of the traces (about x10) and more resources are required to keep the visualization fluent.

TP2_2 : Profiler PyTorch

- Implement the PyTorch profiler in `dlojz.py`.
- Visualize the trace with TensorBoard and draw conclusions about possible optimizations.

TP2_2 : Profiler PyTorch

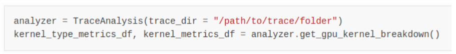


• NOTE

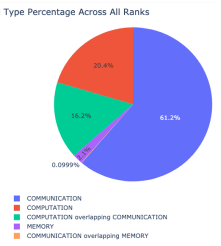
TensorBoard Plugin support has been deprecated, so some of these functions may not work as previously. Please take a look at the replacement, HTA.

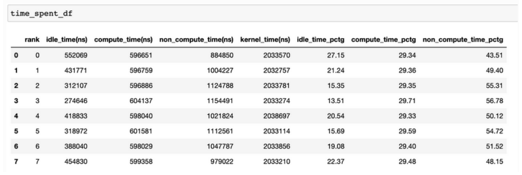
Holistic Trace Analysis: https://hta.readthedocs.io/en/latest/

- Analyses PyTorch Profiler traces.
- Less user-friendly than TensorBoard Plugin.
- More thorough?



```
analyzer = TraceAnalysis(trace_dir = "/path/to/trace/folder")
kernel_type_metrics_df, kernel_metrics_df = analyzer.get_gpu_kernel_breakdown()
```





```
time_spent_df
```

	rank	idle_time(ns)	compute_time(ns)	non_compute_time(ns)	kernel_time(ns)	idle_time_pctg	compute_time_pctg	non_compute_time_pctg
0	0	552069	596651	884850	2033570	27.15	29.34	43.51
1	1	431771	596759	1004227	2032757	21.24	29.36	49.40
2	2	312107	596886	1124788	2033781	15.35	29.35	55.31
3	3	274646	604137	1154491	2033274	13.51	29.71	56.78
4	4	418833	598040	1021824	2038697	20.54	29.33	50.12
5	5	318972	601581	1112561	2033114	15.69	29.59	54.72
6	6	388040	598029	1047787	2033856	19.08	29.40	51.52
7	7	454830	599358	979022	2033210	22.37	29.48	48.15

The TensorBoard plugin `torch_tb_profiler` is deprecated. The developers redirect people to the HTA (Holistic Trace Analysis) library.

This one is also based on the traces generated by the PyTorch profiler.

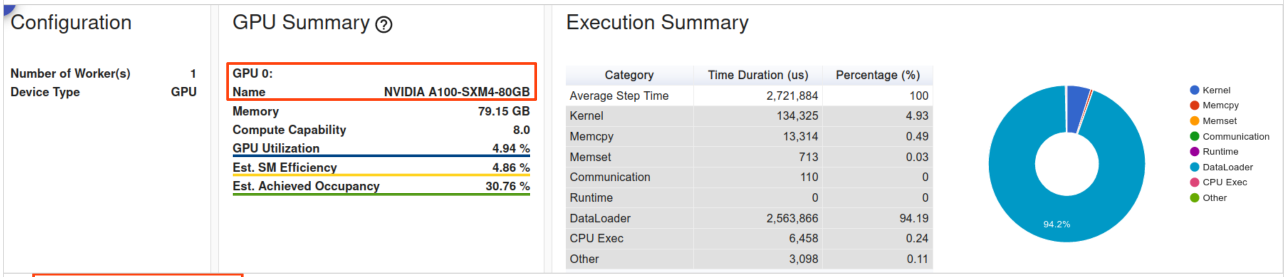
With HTA, traces can be analyzed using a series of tables (DataFrames) or plotting functions.

We think the HTA library is more difficult to grasp than the TensorBoard plugin.

We think the TensorBoard plugin is still operational for now despite some minor bugs.

TP2_2: Profiler Overview

Tutorial: https://pytorch.org/tutorials/intermediate/tensorboard_profiler_tutorial.html



Configuration

Number of Worker(s)	1
Device Type	GPU

GPU Summary

GPU 0:	
Name	NVIDIA A100-SXM4-80GB
Memory	79.15 GB
Compute Capability	8.0
GPU Utilization	4.94 %
Est. SM Efficiency	4.86 %
Est. Achieved Occupancy	30.76 %

Execution Summary

Category	Time Duration (us)	Percentage (%)
Average Step Time	2,721,884	100
Kernel	134,325	4.93
Memcpy	13,314	0.49
Memset	713	0.03
Communication	110	0
Runtime	0	0
DataLoader	2,563,866	94.19
CPU Exec	6,458	0.24
Other	3,098	0.11

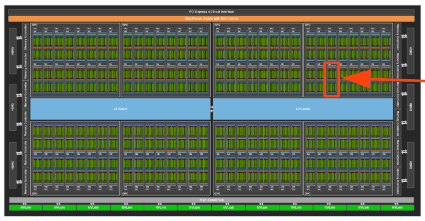
Type and memory capacity of the GPU

% of time spent with an active GPU

% of active SMs

% of active wraps on an SM

A100



Link to image

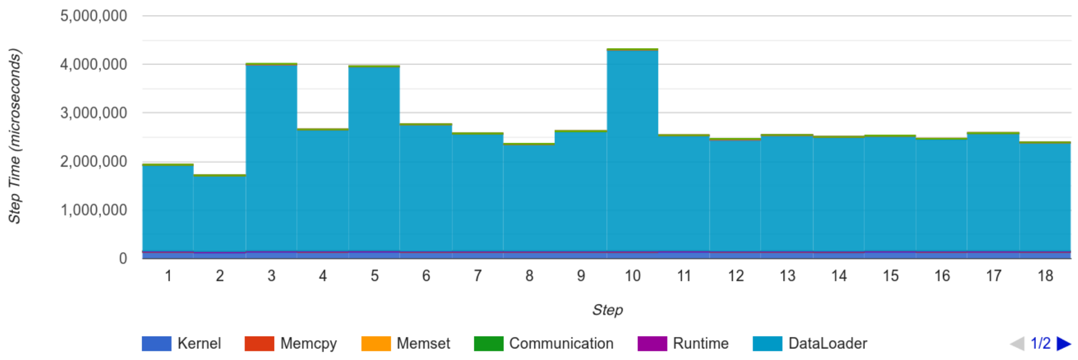
Streaming Multiprocessor

The `Overview` tab gives information on using the GPU up to a very fine granularity (at the scale of Streaming Multiprocessor wraps).

The functions called during the execution are sorted by categories and the percentage of time spent in each category is calculated.

Here, the `DataLoader` is the most time-consuming category, far ahead of the other categories.

TP2_2: Profiler Step Time



Performance Recommendation

- This run has high time cost on input data loading. 94.2% of the step time is in DataLoader. You could try to set num_workers on DataLoader's construction and enable multi-processes on data loading.
- GPU 0 has low utilization. You could try to increase batch size to improve. Note: Increasing batch size may affect the speed and stability of model convergence.

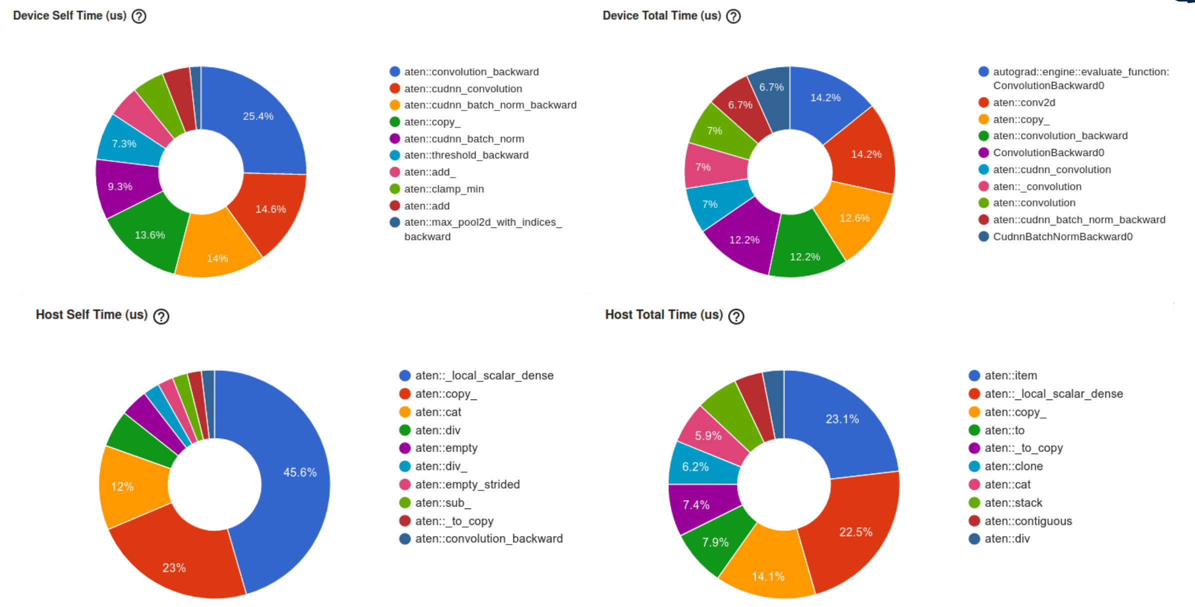
We are visualizing the time taken by each function category, by step.

The PyTorch profiler is capable of furnishing recommendations for possible optimizations of the code.

Here, the time is principally spent in the `DataLoader` at each iteration.

Comment: In general, we ignore the first iteration during which many functionalities are initialized because the time here is not representative.

TP2_2: Profiler Operator View

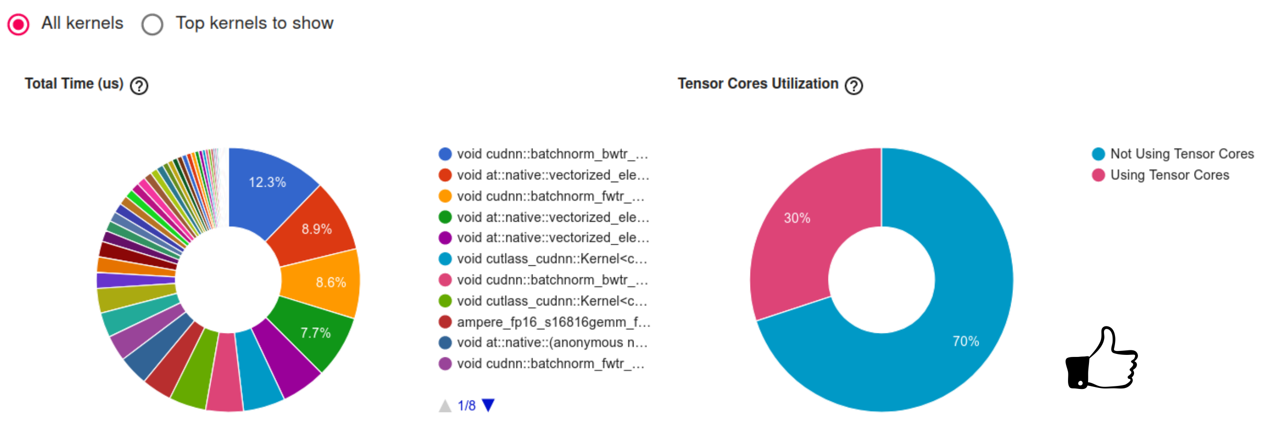


In the `Operator View` tab, we visualize the PyTorch operators which take the most time, on GPU (`Device`) and on CPU (`Host`).

Certain operators call other operators. During the `Self Time` calculation, we ignore the time passed in the child operators. We take this into account during the `Total Time` calculation.

At the time of this writing, a bug prevents the upper “Device” part of this view from being displayed.

TP2_2: Profiler Kernel View



[✓ All kernels] Top kernels to show

- ● void cudnn::batchnorm_bwtr_...
- ● void at::native::vectorized_ele...
- ● void cudnn::batchnorm_fwtr_...
- ● void at::native::vectorized_ele...
- ● void at::native::vectorized_ele...
- ● void cutlass_cudnn::Kernel<c...
- ● void cudnn::batchnorm_bwtr_...
- ● void cutlass_cudnn::Kernel<c...
- ● ampere_fp16_s16816gemm_f...
- ● void at::native::(anonymous n...
- ● void cudnn::batchnorm_fwtr_...

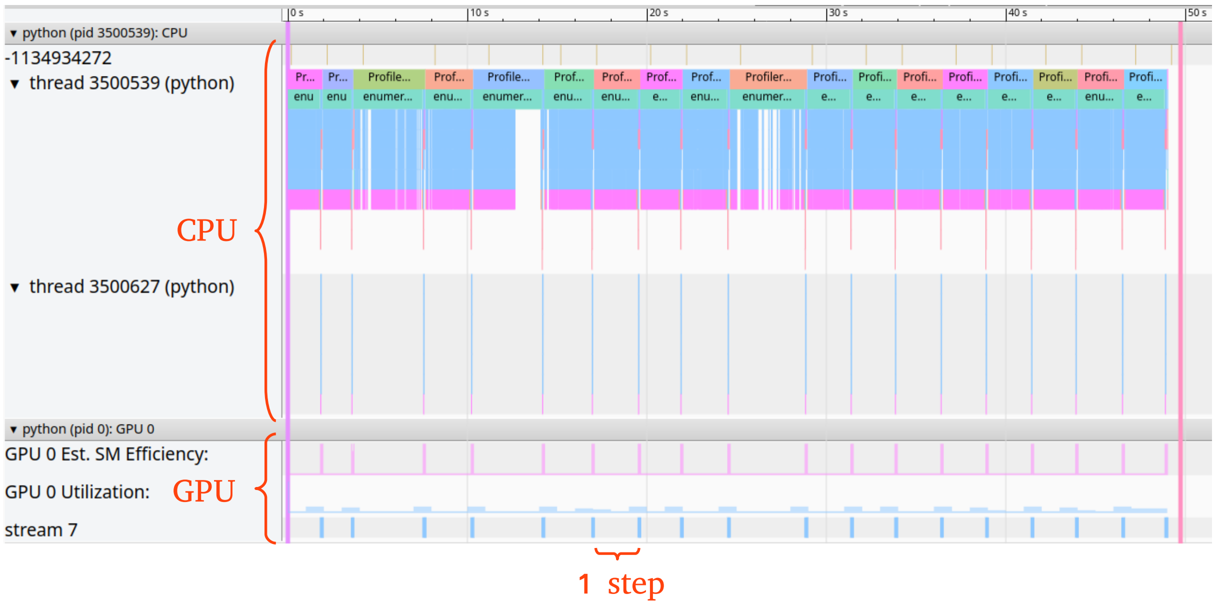
- ● Not Using Tensor Cores
- ● Using Tensor Cores

In the `Kernel View` tab, we visualize the information at the kernel level.

On the left, we visualize the CUDA functions which take the most time.

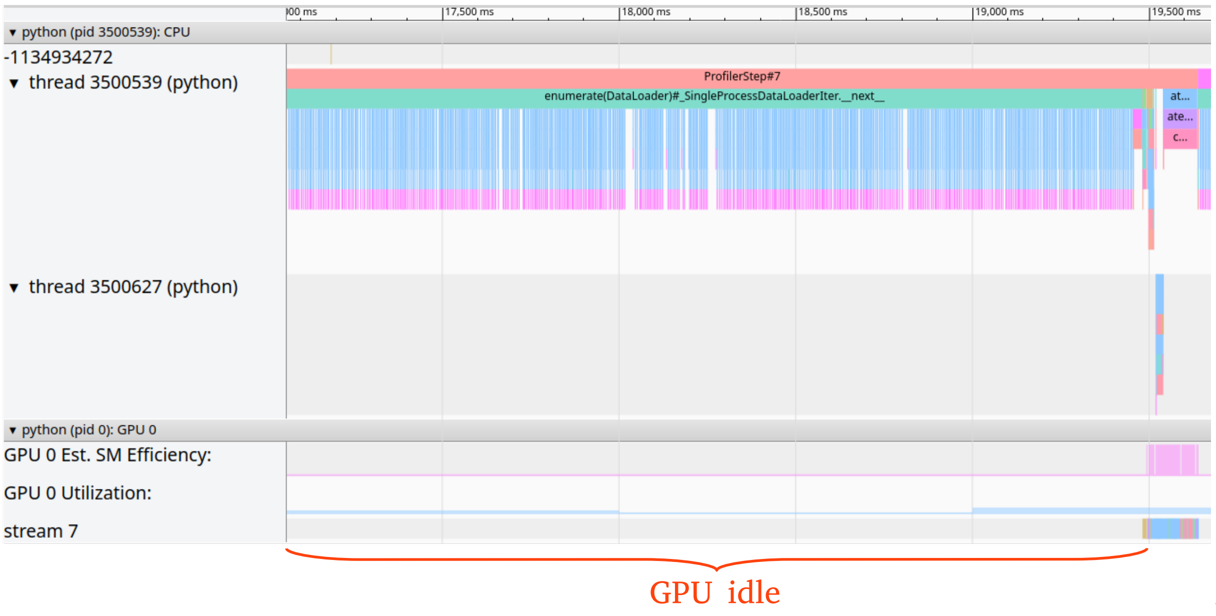
On the right, we visualize the percentage of time spent on the TensorCores. If the TensorCores are used very little or not at all, this could indicate that the mixed precision was not correctly implemented.

TP2_2: Profiler Trace



In the `Trace` tab, the profiler furnishes an execution timeline. We differentiate the CPU activity from the GPU activity.

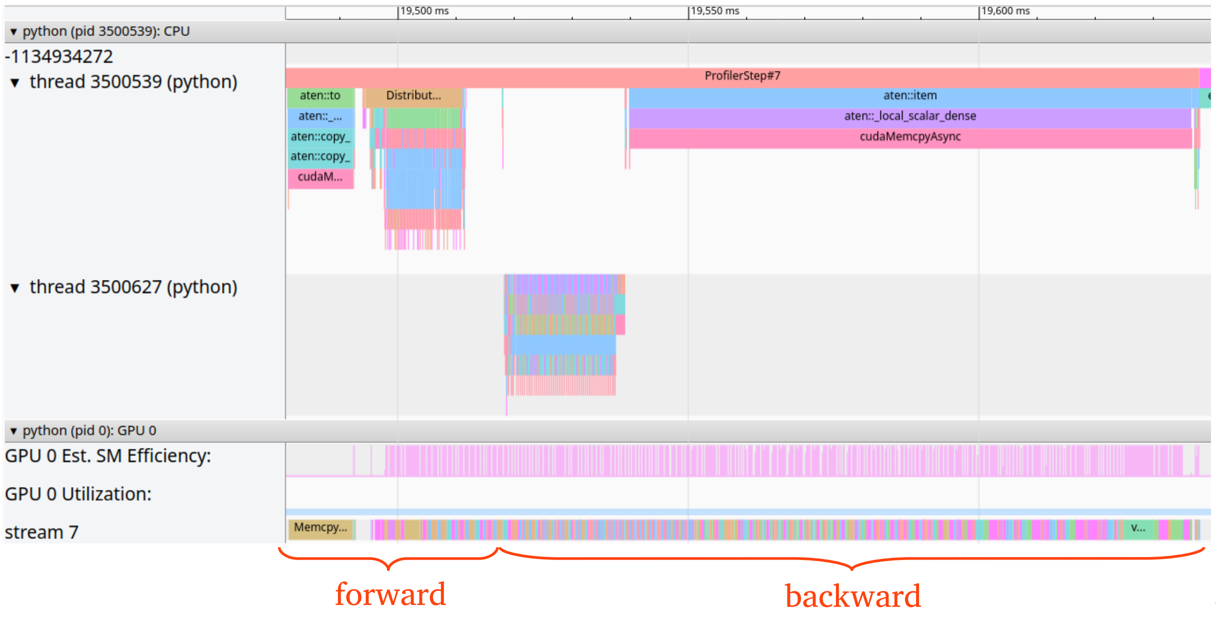
TP2_2: Profiler Trace (1 step)



00 ms
17,500 ms
18,000 ms
18,500 ms
19,000 ms
19,500 ms
▾ python (pid 3500539): CPU
-1134934272
▾ thread 3500539 (python)
ProfilerStep#7
enumerate(DataLoader)#_SingleProcessDataLoaderIter.__next__
at...
ate...
c...
▾ thread 3500627 (python)
▾ python (pid 0): GPU 0
GPU 0 Est. SM Efficiency:
GPU 0 Utilization:
stream 7
GPU idle

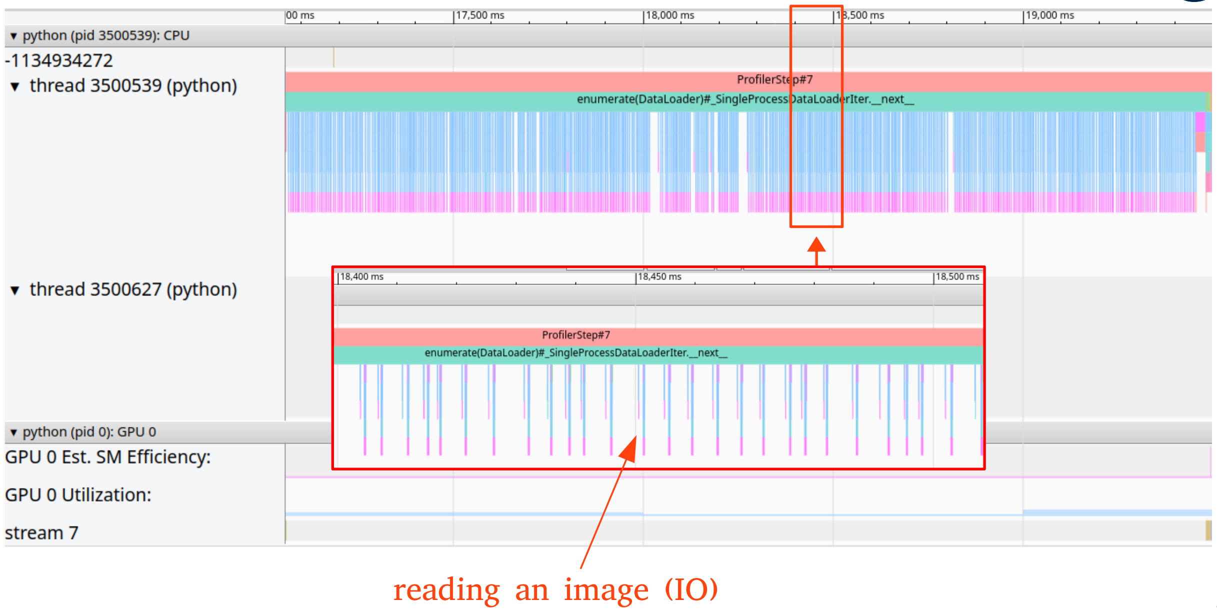
By zooming in on an iteration, we note that the GPU is inactive most of the time.

TP2_2: Profiler Trace (1 step - GPU)



By zooming in on the part of the iteration during which the GPU is active, we differentiate the forward and backward steps.

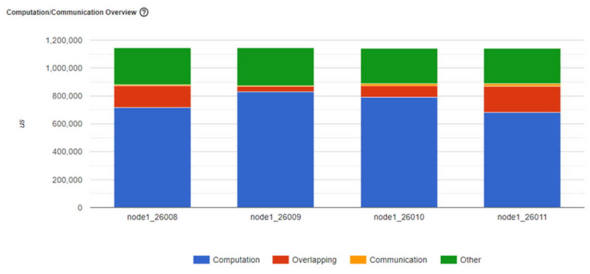
TP2_2: Profiler Trace (1 step - CPU)



By zooming in on the part of the iteration during which the CPU is active, we see the call to the `DataLoader`. It is at this moment that the reading of input images takes place.

These “IO calls” are not displayed when `num_workers>0`.

TP2_2: Profiler Distributed



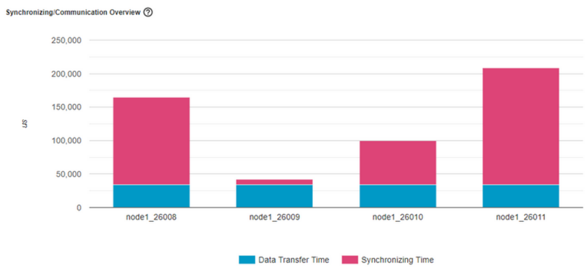


Image from the tutorial: https://pytorch.org/tutorials/intermediate/tensorboard_profiler_tutorial.html

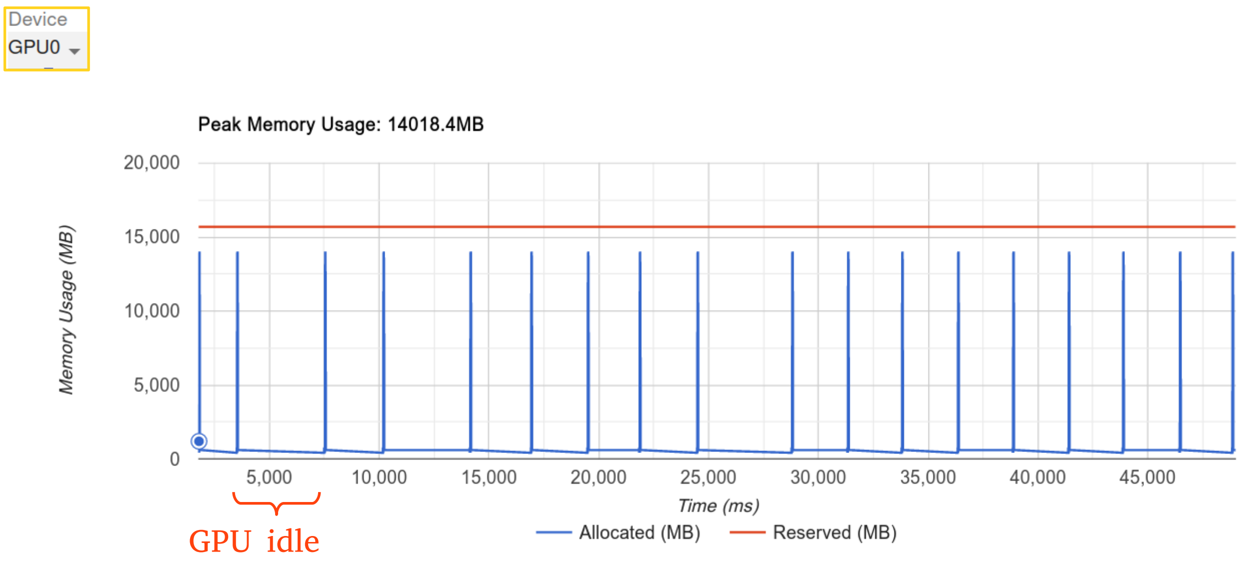
The `Distributed` tab is relevant when you work with multiple GPUs.

We see the computation/communication ratio on the left hand side and the communication efficiency on the right hand side.

These results allow us to determine if the workload is properly balanced between the processes.

At the time of this writing, a bug prevents the “Distributed” tab from being displayed when using multiple GPUs.

TP2_2: Profiler Memory View (GPU)

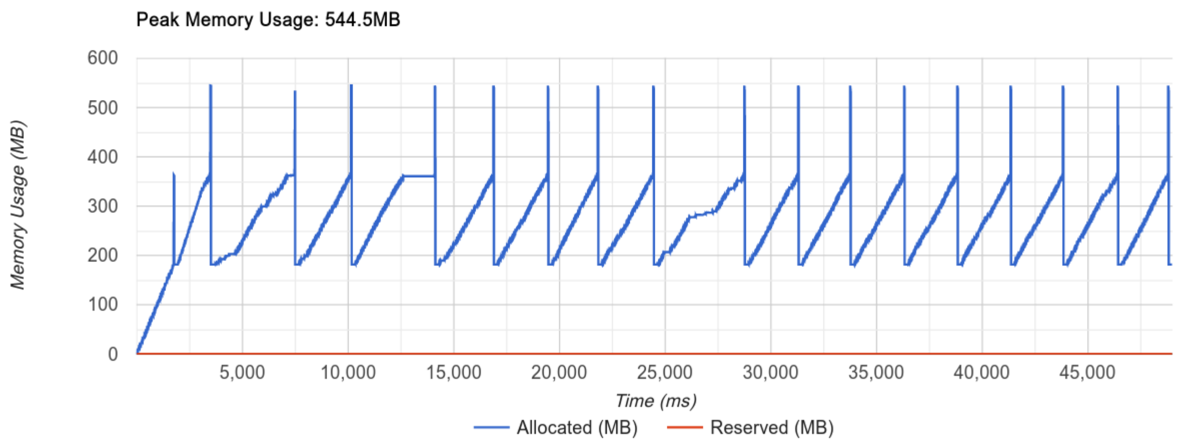


In the `Memory View` tab, we see the GPU memory usage over time.

We distinguish here also the periods during which the GPU is inactive.

TP2_2: Profiler Memory View (CPU)

Device
CPU

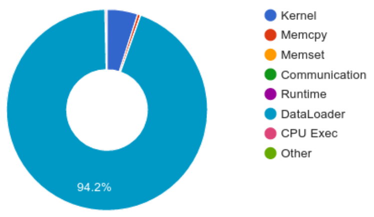


The CPU memory usage over time is also shown.

We distinguish a progressive loading of the memory at each iteration during the reading of images, a peak during the data transformation, then a significant deallocation when the batch is transferred from CPU memory to GPU memory.

Note: This view is not available anymore when `num_workers>0`.

TP2_2: Profiler PyTorch (conclusion)



After seeing the traces, it is obvious that the optimization efforts need to concentrate on the DataLoader.

Deep Learning Optimized on Jean Zay

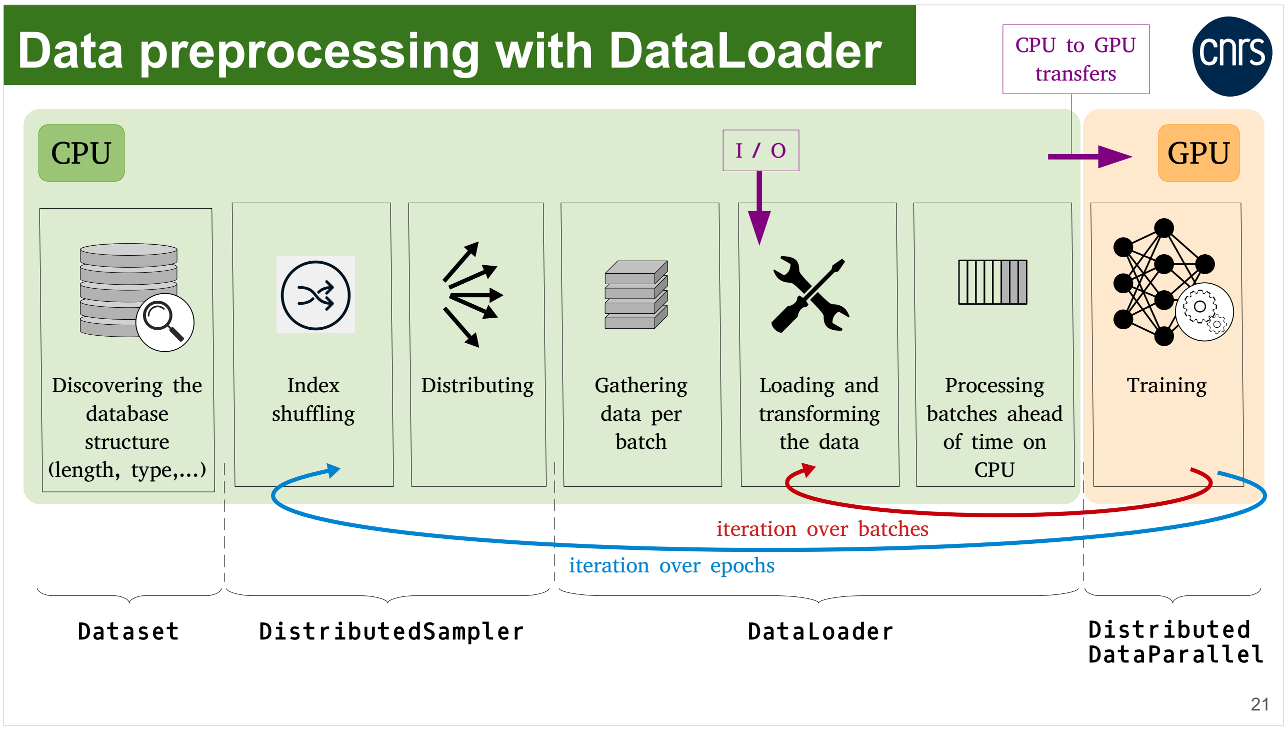
Optimization of the data preprocessing

Optimization of the data preprocessing

Data preprocessing with DataLoader ◀

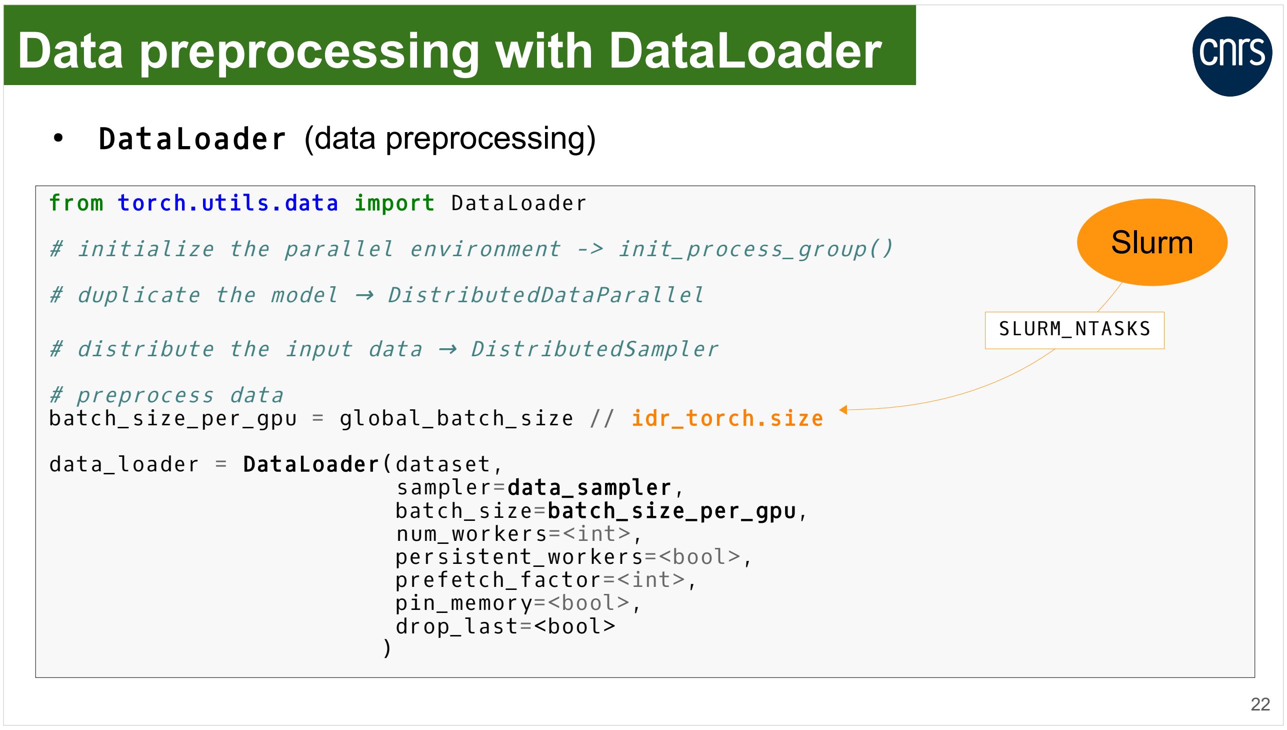
Optimization of the DataLoader ◀

Data preprocessing with DataLoader



Details of the data preprocessing workflow during a training.

In this section, we will focus on the steps managed by the `DataLoader`.



Data preprocessing with DataLoader

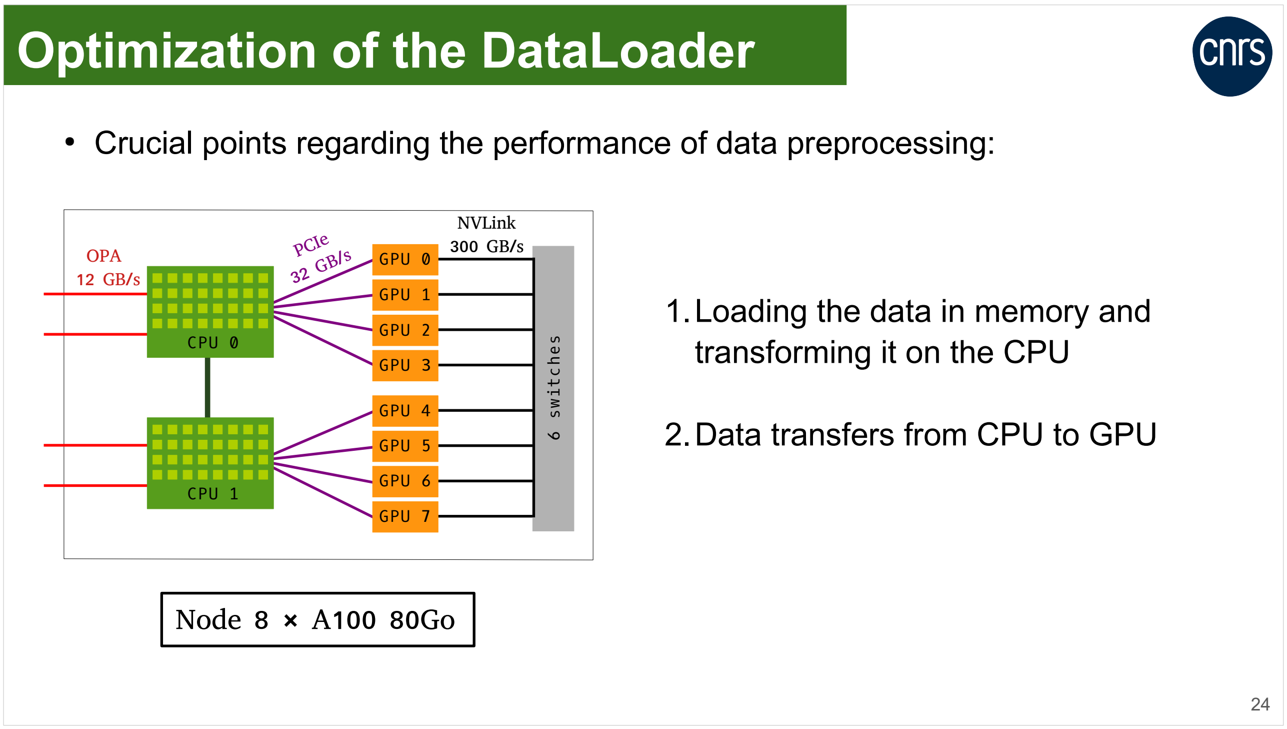
- `DataLoader` (data preprocessing)

```
from torch.utils.data import DataLoader

# initialize the parallel environment -> init_process_group()

# duplicate the model → DistributedDataParallel

# distribute the input data → DistributedSampler

# preprocess data
batch_size_per_gpu = global_batch_size // idr_torch.size

data_loader = DataLoader(dataset,
                         sampler=data_sampler,
                         batch_size=batch_size_per_gpu,
                         num_workers=<int>,
                         persistent_workers=<bool>,
                         prefetch_factor=<int>,
                         pin_memory=<bool>,
                         drop_last=<bool>
                        )
```

Example: A usual call to the `DataLoader`.

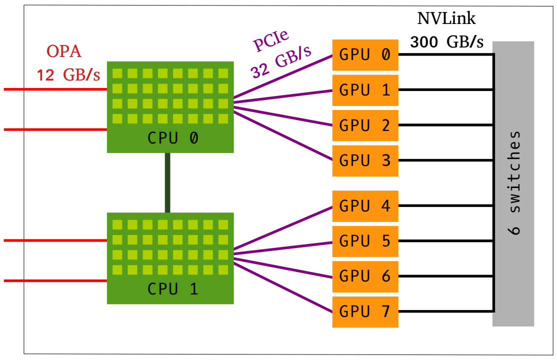
Important: The batch size indicated at the moment of the creation of the `DataLoader` is the batch size per GPU: `batch_size_per_gpu`.

Optimization of the data preprocessing



Optimization of the DataLoader

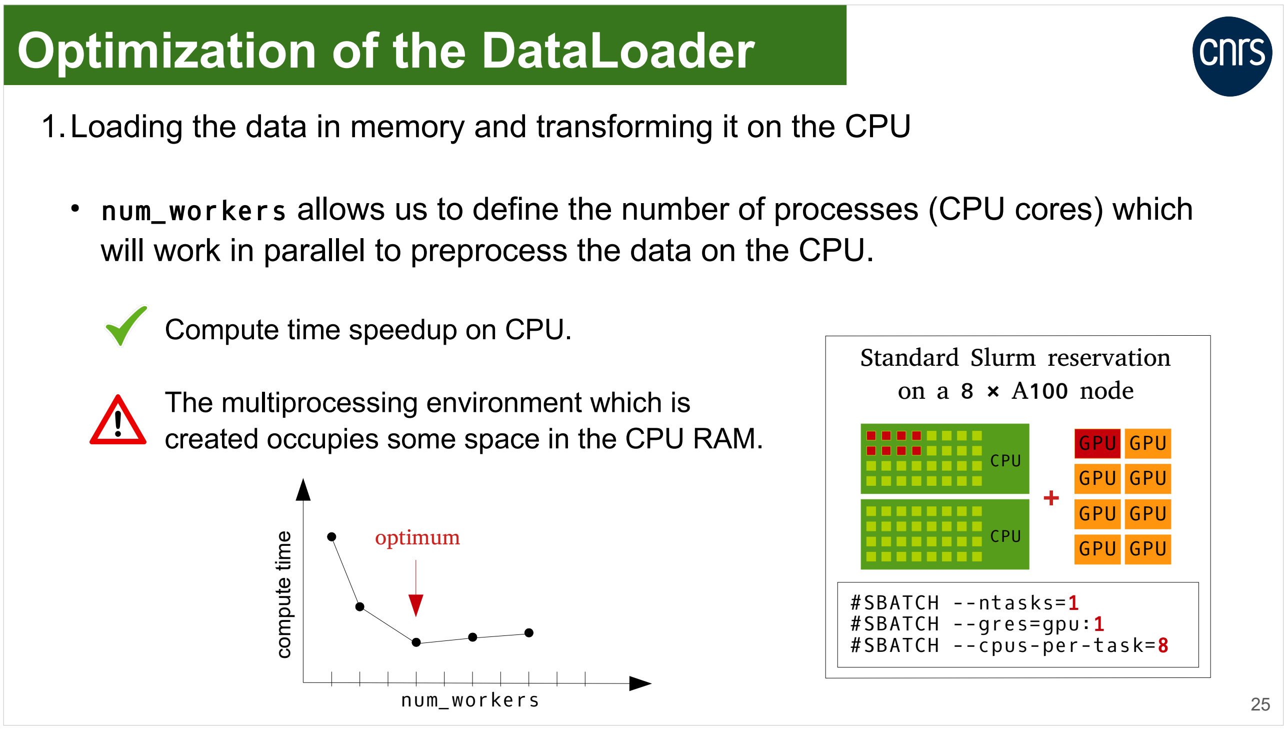
- Crucial points regarding the performance of data preprocessing:



Node 8 × A100 80Go

1. Loading the data in memory and transforming it on the CPU
2. Data transfers from CPU to GPU

The performance of the `DataLoader` is mainly driven by: the CPU performance for data loading and transformation, and the time spent in CPU to GPU transfers.



Optimization of the DataLoader

1. Loading the data in memory and transforming it on the CPU

- `num_workers` allows us to define the number of processes (CPU cores) which will work in parallel to preprocess the data on the CPU.

✓ Compute time speedup on CPU.

⚠ The multiprocessing environment which is created occupies some space in the CPU RAM.

compute time
optimum
num_workers
Standard Slurm reservation on a 8 × A100 node
CPU
CPU
GPU GPU GPU GPU GPU GPU GPU GPU

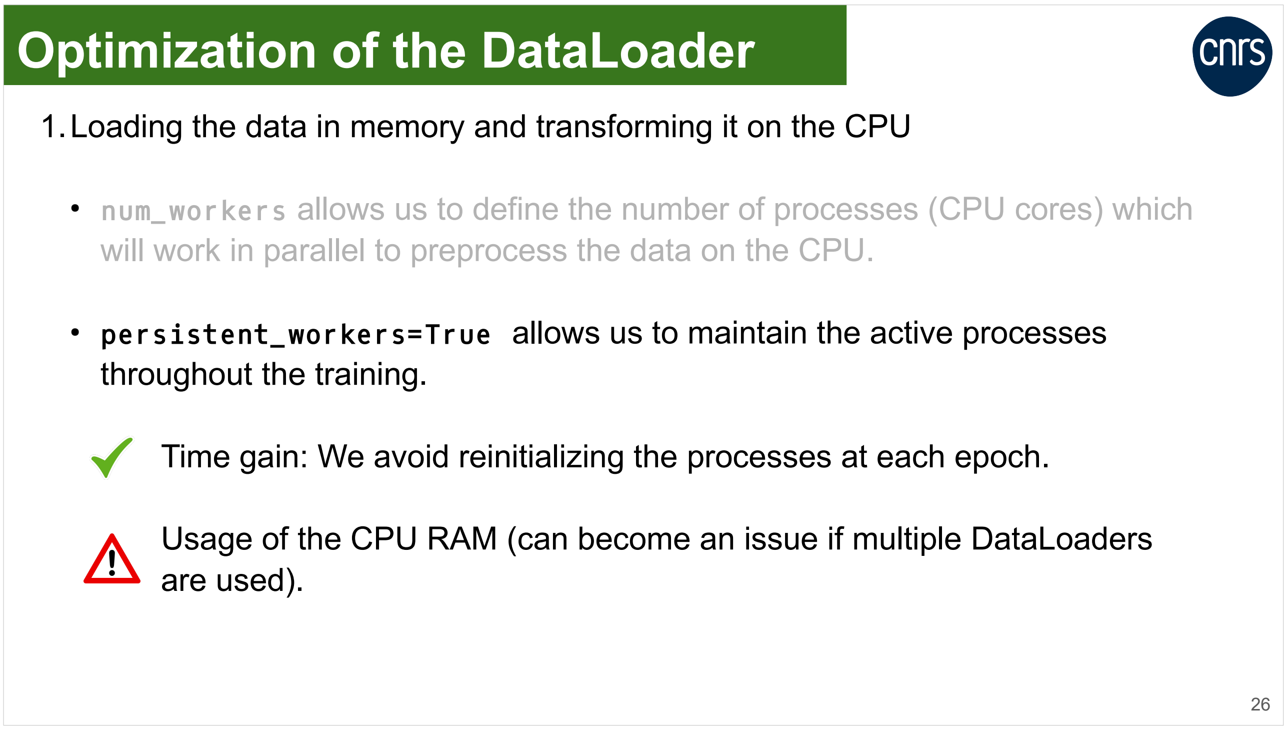
```
#SBATCH --ntasks=1
#SBATCH --gres=gpu:1
#SBATCH --cpus-per-task=8
```

The data loading and transformation operations can be effectuated in parallel on multiple CPU cores. The processes implicated are called “workers” here.

Instead of one batch, `num_workers` batches will be preprocessed at the same time.

This option offers an important speedup in preprocessing time. On the other hand, the creation of a parallel environment takes up space in the CPU RAM.

In a typical Slurm allocation on a Jean Zay octo-GPU, we reserve 8 CPU cores per task. Optimization tests should be conducted for each test case to define the optimal number of workers (lower than 8? multiple of 8?).



Optimization of the DataLoader

1. Loading the data in memory and transforming it on the CPU

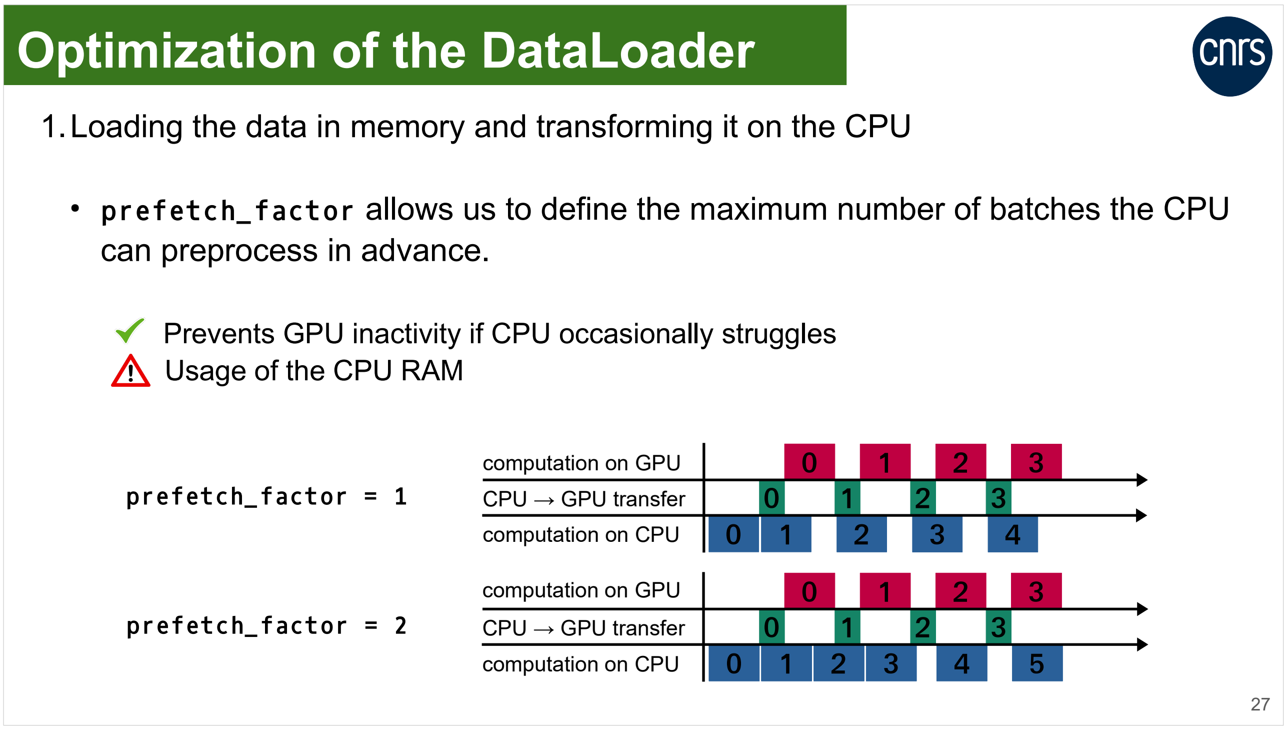
- `num_workers` allows us to define the number of processes (CPU cores) which will work in parallel to preprocess the data on the CPU.

- `persistent_workers=True` allows us to maintain the active processes throughout the training.

✓ Time gain: We avoid reinitializing the processes at each epoch.

⚠ Usage of the CPU RAM (can become an issue if multiple DataLoaders are used).

Initializing the workers takes time (proportionately to their number). It is advised to avoid reinitializing them at each epoch unless you need to make room in the CPU RAM.



Optimization of the DataLoader

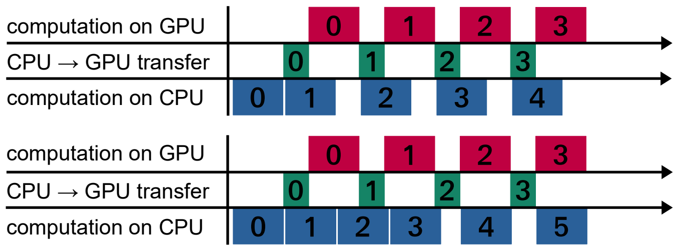
1. Loading the data in memory and transforming it on the CPU

   - `prefetch_factor` allows us to define the maximum number of batches the CPU can preprocess in advance.

     ✓ Prevents GPU inactivity if CPU occasionally struggles
     ⚠ Usage of the CPU RAM

`prefetch_factor = 1`

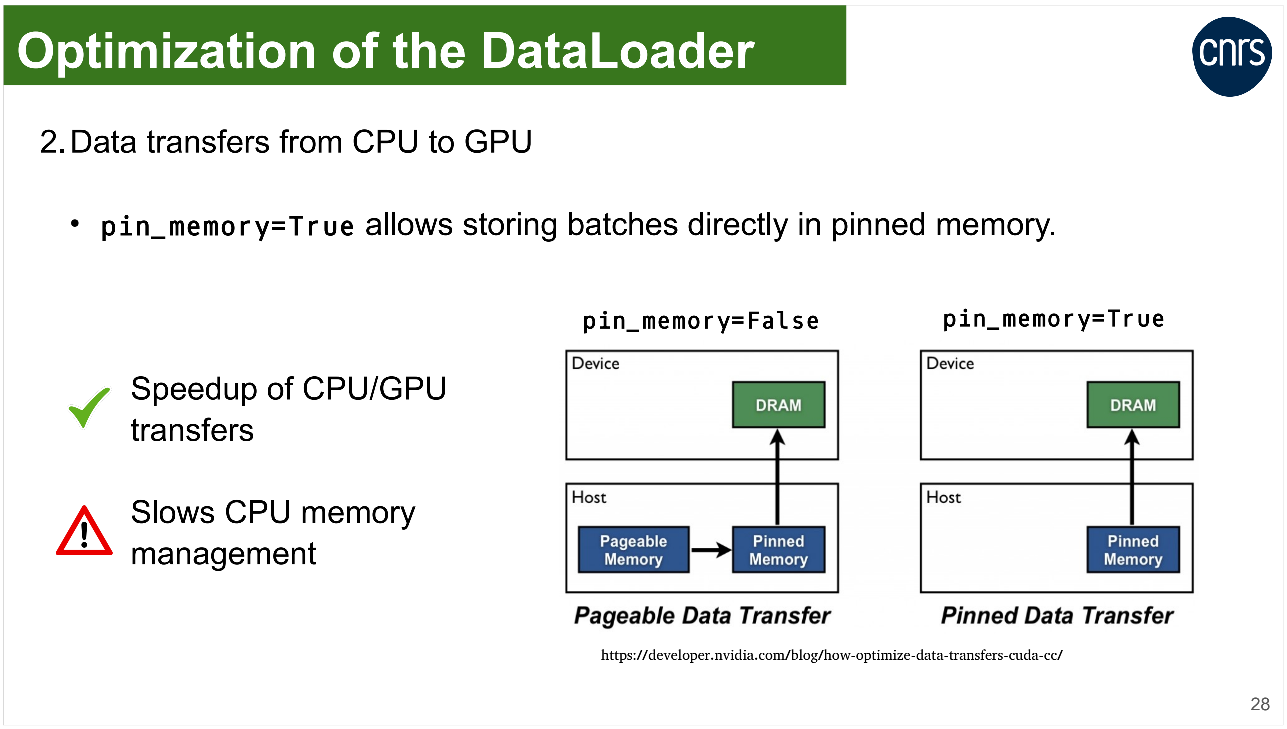
`prefetch_factor = 2`



Preprocessing batches in advance on CPU can prevent GPU inactivity in case of occasional CPU slowdowns.

The CPU will preprocess input data per pack of `num_workers×prefetch_factor` batches.

Important: The preprocessed batches occupy CPU memory and could saturate it if the prefetch factor is too large.



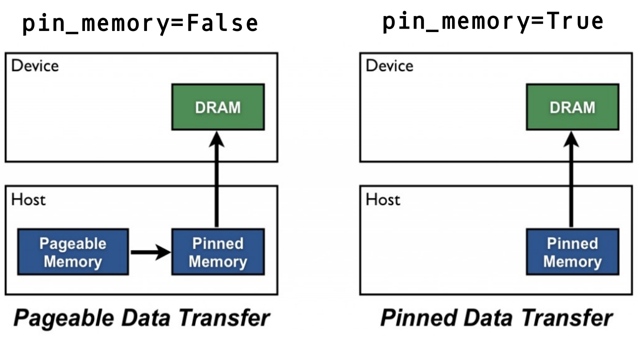
Optimization of the DataLoader

2. Data transfers from CPU to GPU

- `pin_memory=True` allows storing batches directly in pinned memory.

✓ Speedup of CPU/GPU transfers

⚠ Slows CPU memory management

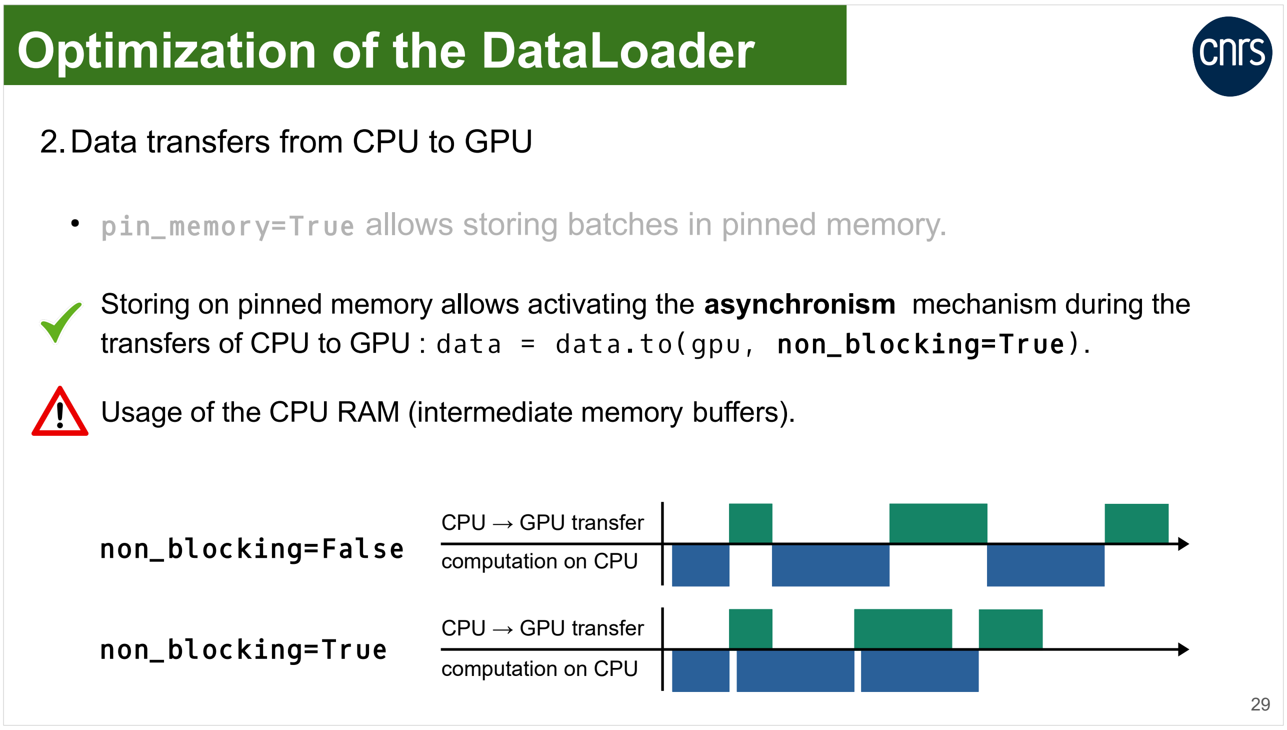


https://developer.nvidia.com/blog/how-optimize-data-transfers-cuda-cc/

To be transferred to the GPU, a message must first be copied from the pageable memory to the pinned memory on the CPU.

The `pin_memory=True` option enables storing the message directly in pinned memory to speed up the transfer.

It is advised to activate this option except if it slows down the CPU performance.



Optimization of the DataLoader

2. Data transfers from CPU to GPU

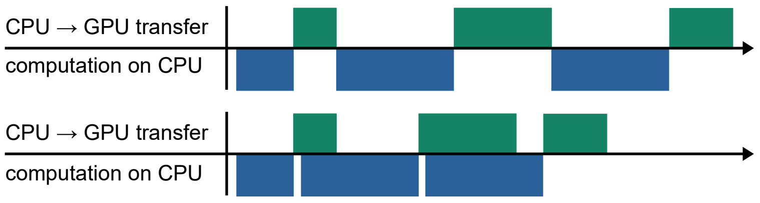
- `pin_memory=True` allows storing batches in pinned memory.

✓ Storing on pinned memory allows activating the **asynchronism** mechanism during the transfers of CPU to GPU : `data = data.to(gpu, non_blocking=True)`.

⚠ Usage of the CPU RAM (intermediate memory buffers).

`non_blocking=False`

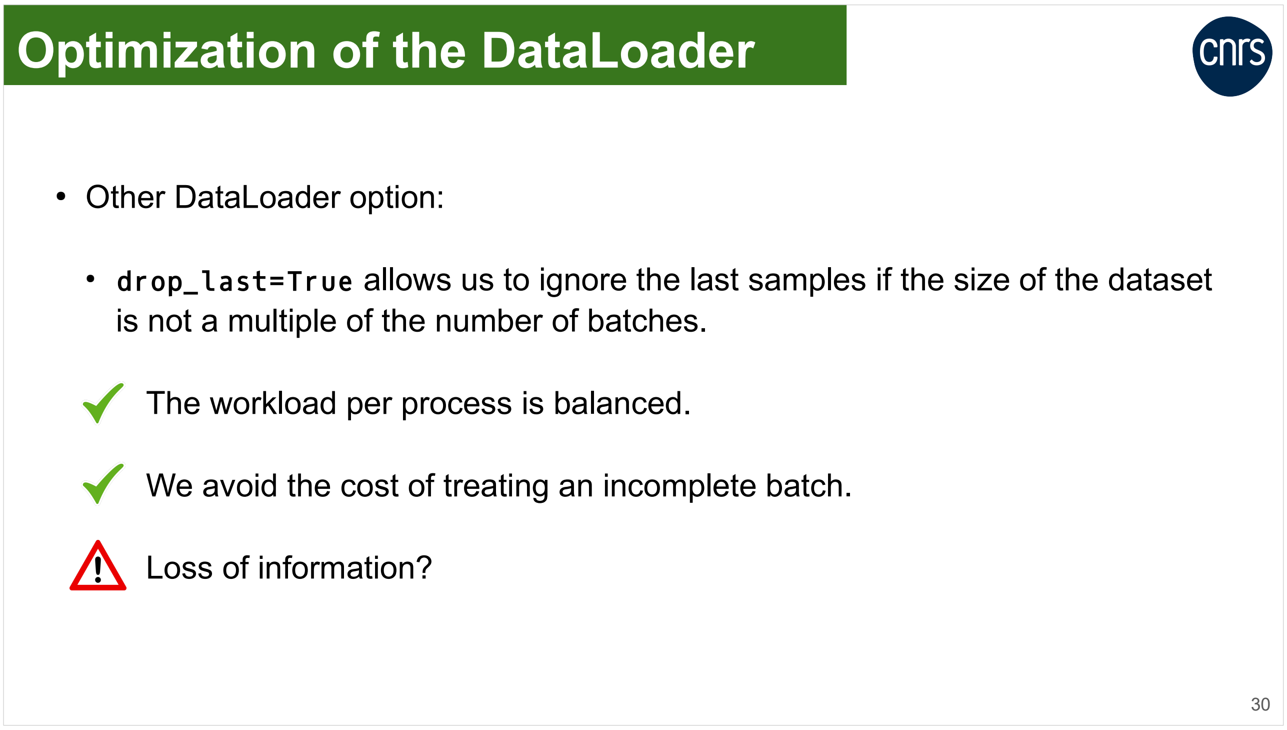
`non_blocking=True`



Activating the `pin_memory` option allows effectuating asynchronous sends during the transfers from CPU to GPU.

As a result, the CPU no longer needs to wait for the GPU to receive the message before continuing its computation.

The CPU computation and the CPU to GPU transfer overlap so we gain time.



Optimization of the DataLoader

- Other DataLoader option:
  - `drop_last=True` allows us to ignore the last samples if the size of the dataset is not a multiple of the number of batches.

✓ The workload per process is balanced.

✓ We avoid the cost of treating an incomplete batch.

⚠ Loss of information?

To avoid GPU inactivity, setting `drop_last=True` is is good practice. In return, you may lose some information.

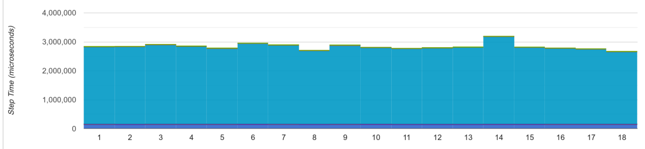
TP2_3 : Optimization of the DataLoader

- Modify the DataLoader options.
- Measure the time gain on a few steps.

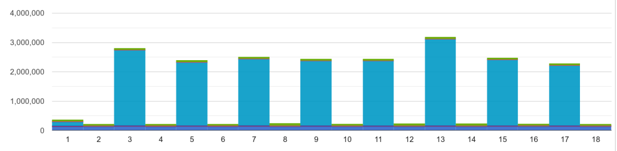
TP2_3 : Optimization of the DataLoader

- The most efficient optimization is the increase of `num_workers`.

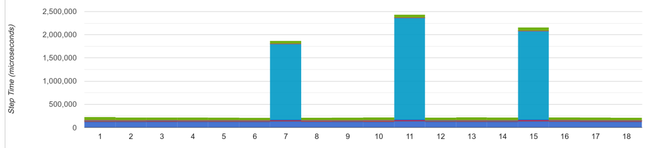
`num_workers=0`



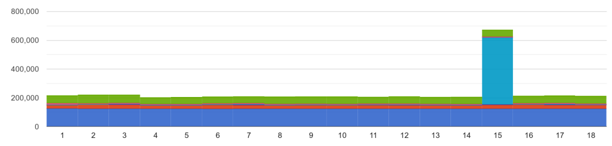
`num_workers=2`



`num_workers=4`



`num_workers=8`

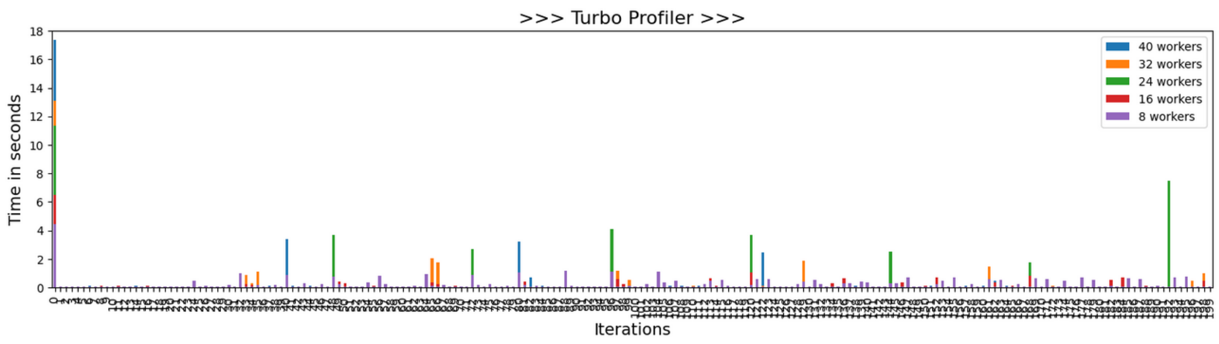


🟦 Kernel 🟥 Memcpy 🟧 Memset 🟩 Communication 🟪 Runtime 🟦 DataLoader 🟥 CPU Exec 🟩 Other

Since the CPU preprocesses packs of `num_workers` batches, we can see that `num_workers` steps can run in a row without waiting.

We gain a lot of time by increasing the number of workers .

TP2_3 : Optimization of the DataLoader



	jobid	num_workers	persistent_workers	pin_memory	non_blocking	prefetch_factor	drop_last	loading_time	training_time
1	830199	16	False	False	False	2	False	0.140631	81.492809s
3	830217	32	False	False	False	2	False	0.145662	146.490717s
4	830224	40	False	False	False	2	False	0.147003	150.194498s
2	830213	24	False	False	False	2	False	0.200591	151.584189s
0	830180	8	False	False	False	2	False	0.204219	87.450866s

During the first iteration, we can see that the initialization takes more time when using more workers.

Even when using a large number of workers, time peaks still appear due to the variability of IO performance on Jean Zay.

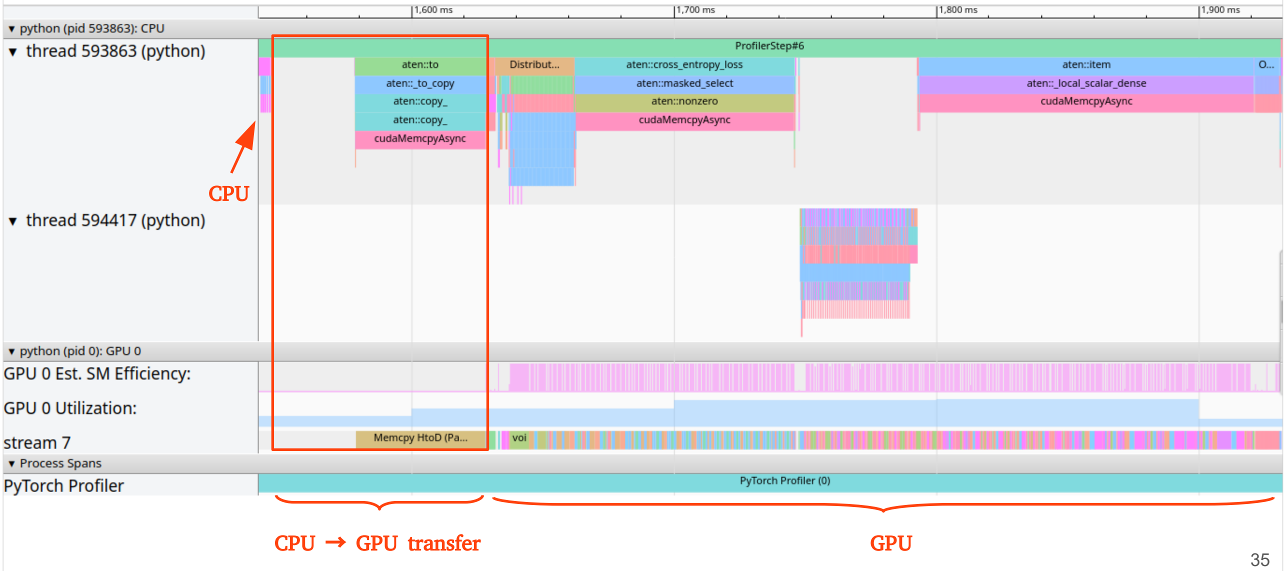
TP2_3 : Optimization of the DataLoader

Intermediate conclusion about `num_workers` setting:

- Increase `num_workers` progressively and observe if the DataLoader scales or not on a few steps.
- For low CPU workload, `num_workers` can be a multiple of `cpus-per-task`.
- Setting too many workers creates bottlenecks or Out Of Memory failures.
- Be aware that few steps are not completely representative.
- IOs on Jean Zay are erratic.

TP2_3 : Optimization of the DataLoader

`pin_memory=False, non_blocking=False`



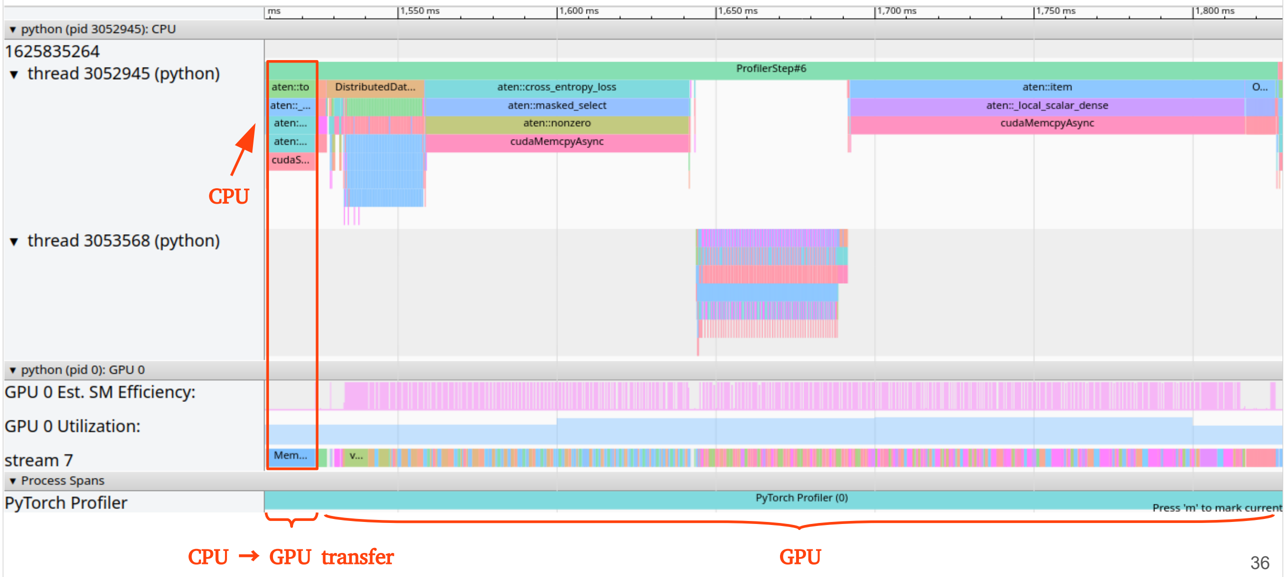
If we zoom on an iteration, we note that the time taken by the `DataLoader` on CPU becomes negligible when increasing the number of workers.

The GPU is now active three-quarters of the time.

We will try to optimize the part outlined in red which corresponds to the transfer of data from CPU to GPU.

TP2_3 : Optimization of the DataLoader

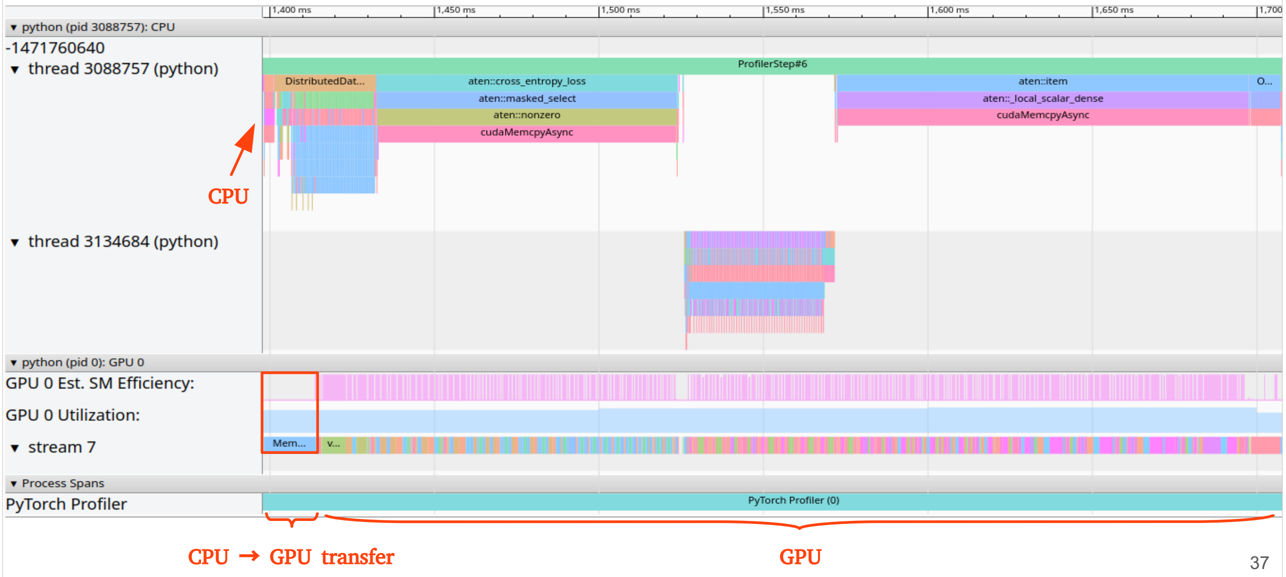
`pin_memory=True, non_blocking=False`



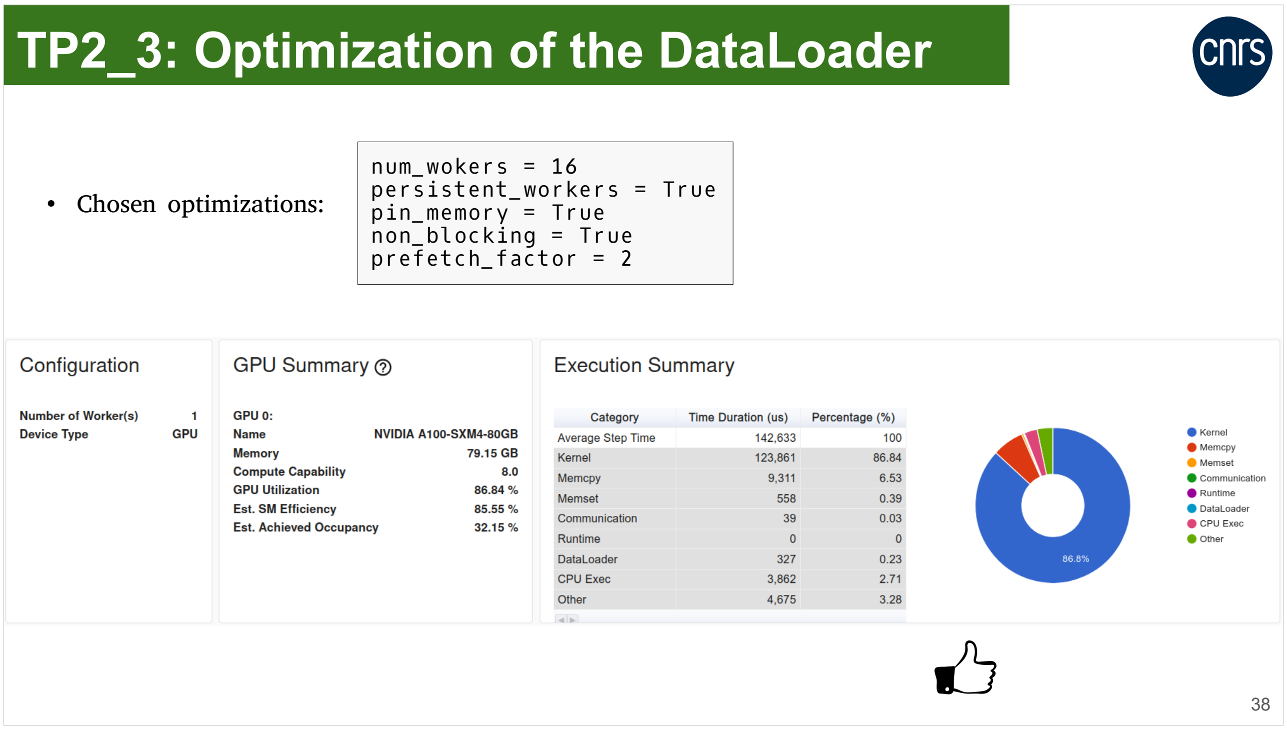
By storing the preprocessed batches directly in pinned memory, we reduce the transfer time.

TP2_3 : Optimization of the DataLoader

`pin_memory=True, non_blocking=True`



By activating the asynchronism mechanism, the CPU does not wait for the transfer from CPU to GPU to be terminated before continuing its instructions.



TP2_3: Optimization of the DataLoader

- Chosen optimizations:

```
num_wokers = 16
persistent_workers = True
pin_memory = True
non_blocking = True
prefetch_factor = 2
```

Configuration

Number of Worker(s)	1
Device Type	GPU

GPU Summary

GPU 0:	
Name	NVIDIA A100-SXM4-80GB
Memory	79.15 GB
Compute Capability	8.0
GPU Utilization	86.84 %
Est. SM Efficiency	85.55 %
Est. Achieved Occupancy	32.15 %

Execution Summary

Category	Time Duration (us)	Percentage (%)
Average Step Time	142,633	100
Kernel	123,861	86.84
Memcpy	9,311	6.53
Memset	558	0.39
Communication	39	0.03
Runtime	0	0
DataLoader	327	0.23
CPU Exec	3,862	2.71
Other	4,675	3.28

The major optimization of the DataLoader is the parallelization of the data loading and transformation on CPU with `num_workers>1`.

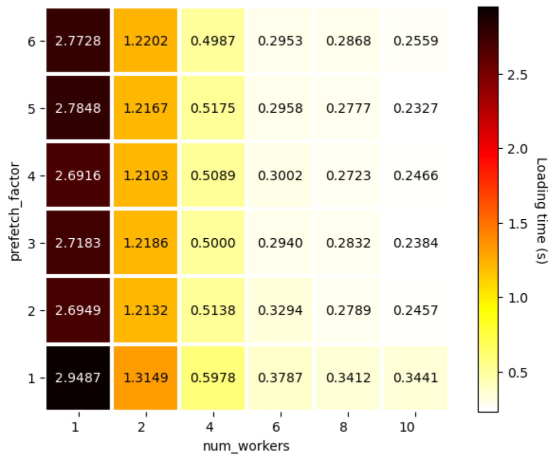
The optimal number of workers depends on your use case. You should run tests on a few steps to choose. To many workers can slow down the execution.

We advise you to:
- Avoid reinitializing the workers at each epoch by setting `persistent_workers=True`.
- Store batches directly on the pinned memory by setting `pin_memory=True` unless you observe some weird CPU slowdown.
- Activate the asynchronism of CPU to GPU tranfers by setting `non_blocking=True` when sending batches to GPU.
- Avoid incomplete batches by setting `drop_last=True` unless you think you might lose too much information.

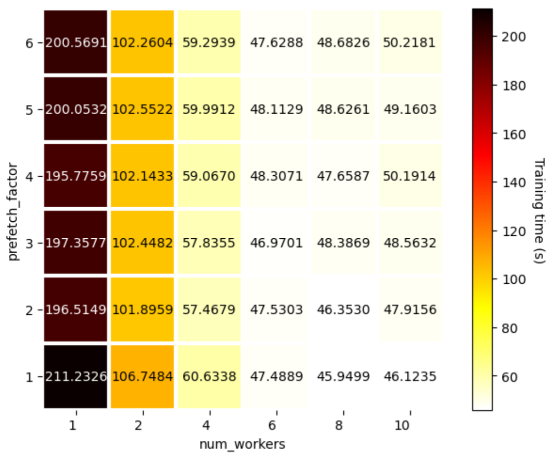
Setting `prefetch_factor=2` (default) is usually enough.

Appendix: Optimization of the DataLoader

- Impact of the `prefetch factor`
  dlojz.py - 50 iterations - test partition gpu_p4
  NB: These results don't correspond to our usage case but still illustrate the influence of the parameters.



prefetch_factor \ num_workers	1	2	4	6	8	10
6	2.7728	1.2202	0.4987	0.2953	0.2868	0.2559
5	2.7848	1.2167	0.5175	0.2958	0.2777	0.2327
4	2.6916	1.2103	0.5089	0.3002	0.2723	0.2466
3	2.7183	1.2186	0.5000	0.2940	0.2832	0.2384
2	2.6949	1.2132	0.5138	0.3294	0.2789	0.2457
1	2.9487	1.3149	0.5978	0.3787	0.3412	0.3441



prefetch_factor \ num_workers	1	2	4	6	8	10
6	200.5691	102.2604	59.2939	47.6288	48.6826	50.2181
5	200.0532	102.5522	59.9912	48.1129	48.6261	49.1603
4	195.7759	102.1433	59.0670	48.3071	47.6587	50.1914
3	197.3577	102.4482	57.8355	46.9701	48.3869	48.5632
2	196.5149	101.8959	57.4679	47.5303	46.3530	47.9156
1	211.2326	106.7484	60.6338	47.4889	45.9499	46.1235

We present here the results of a parametric study on the impact of the `num_workers` and `prefetch_factor` parameters on the data preprocessing time on CPU.

This study was conducted on the `dlojz.py` test case on 50 iterations, on the Jean Zay `gpu_p4` partition (octo-GPU A100 PCIe node). The numbers seen here are not completely the same as in our "hands-on" case but the same trends appear.

The principal acceleration factor is the number of workers. For a given number of workers, a `prefetch_factor` of 2 or 3 is generally sufficient.