Deep Learning Optimized on Jean Zay

Profiler PyTorch

IDRIS

PyTorch Profiler

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()
```

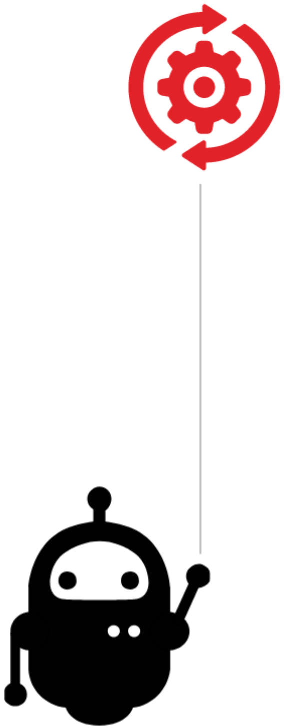
PyTorch Profiler

```
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.

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

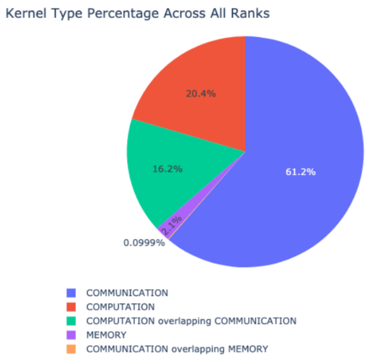
• NOTE

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

TP2_2: Profiler Overview

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

Configuration

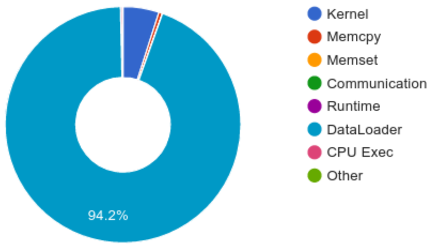
Number of Worker(s)	**1**
Device Type	**GPU**

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

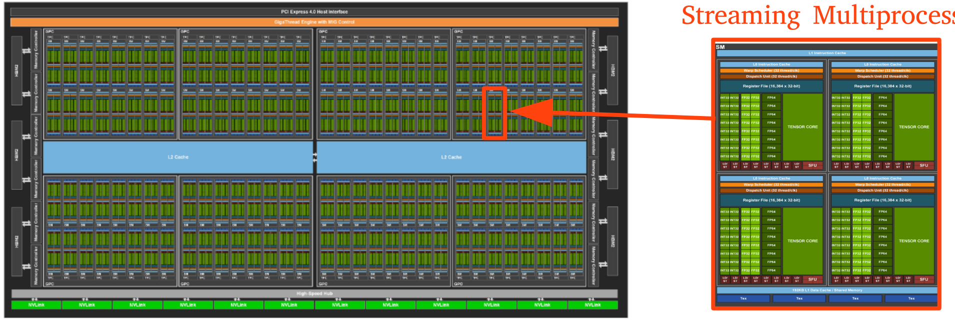
% of time spent with an active GPU

% of active SMs

% of active wraps on an SM

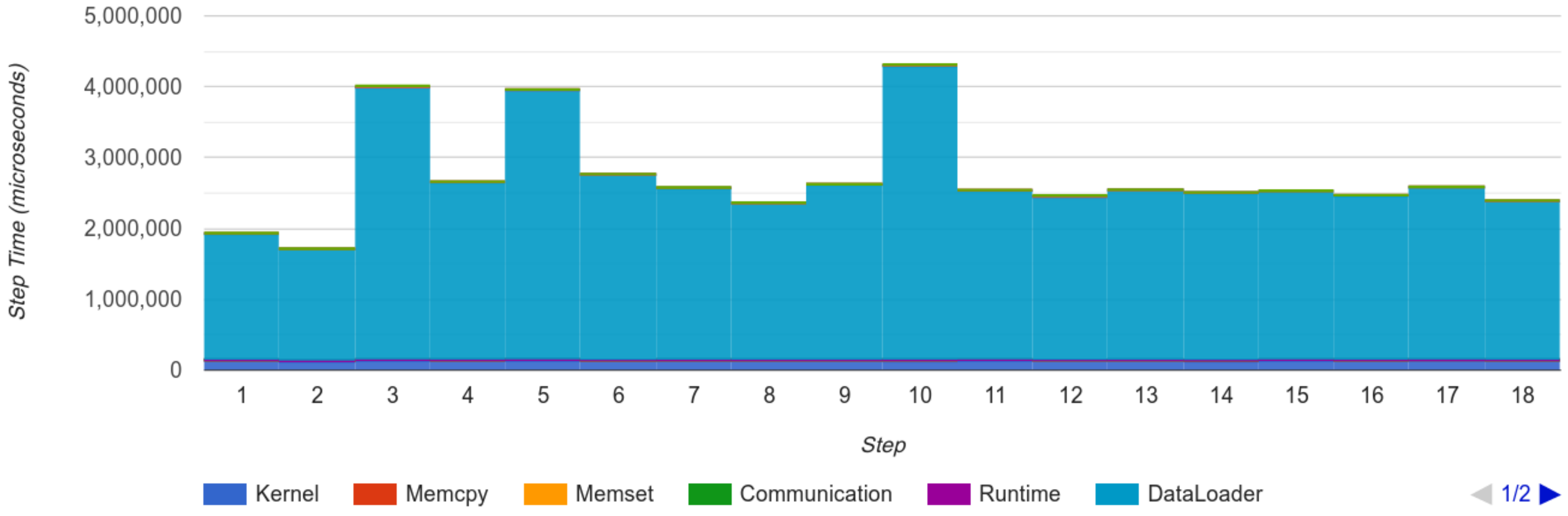
A100

Streaming Multiprocessor



Link to image

TP2_2: Profiler Step Time

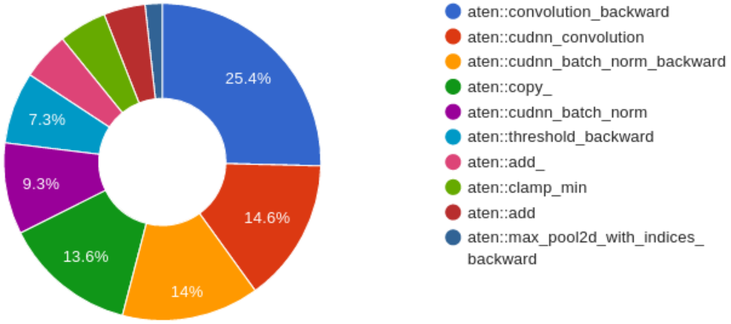


Performance Recommendation

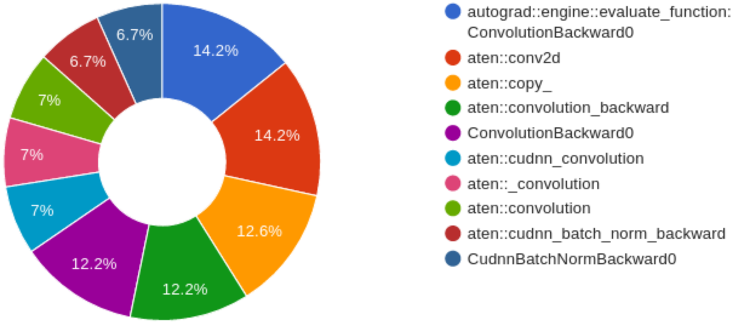
- This run has high time cost on input data loading. 94.2% of the step time is in DataLoader. You could try to set num_workers on DataLoader's construction and enable multi-processes on data loading.
- GPU 0 has low utilization. You could try to increase batch size to improve. Note: Increasing batch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

TP2_2: Profiler Operator View

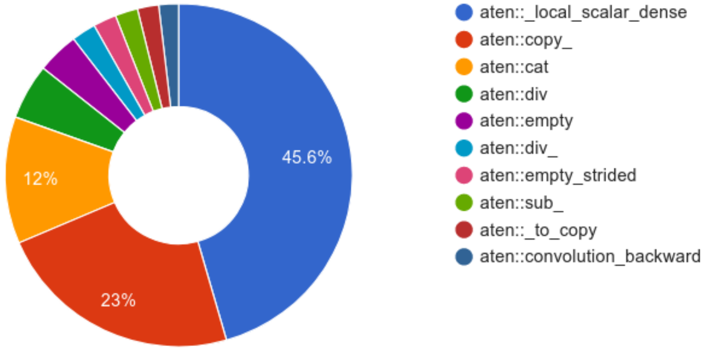
Device Self Time (us)



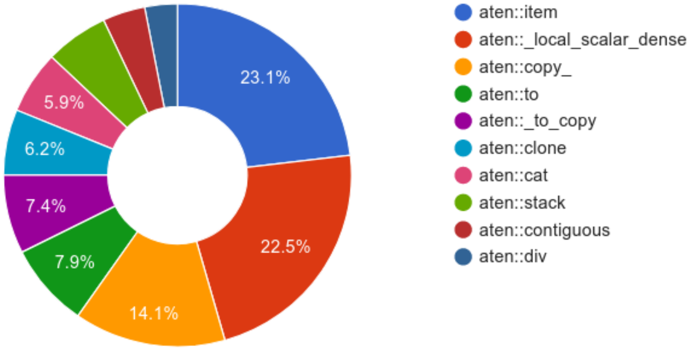
Device Total Time (us)



Host Self Time (us)



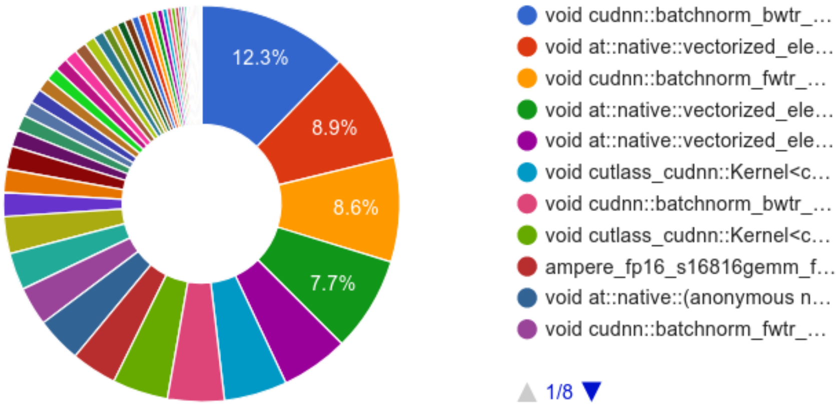
Host Total Time (us)



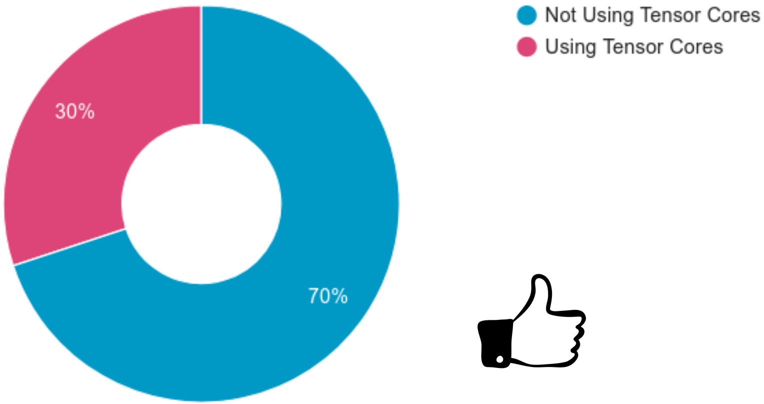
TP2_2: Profiler Kernel View

All kernels   Top kernels to show

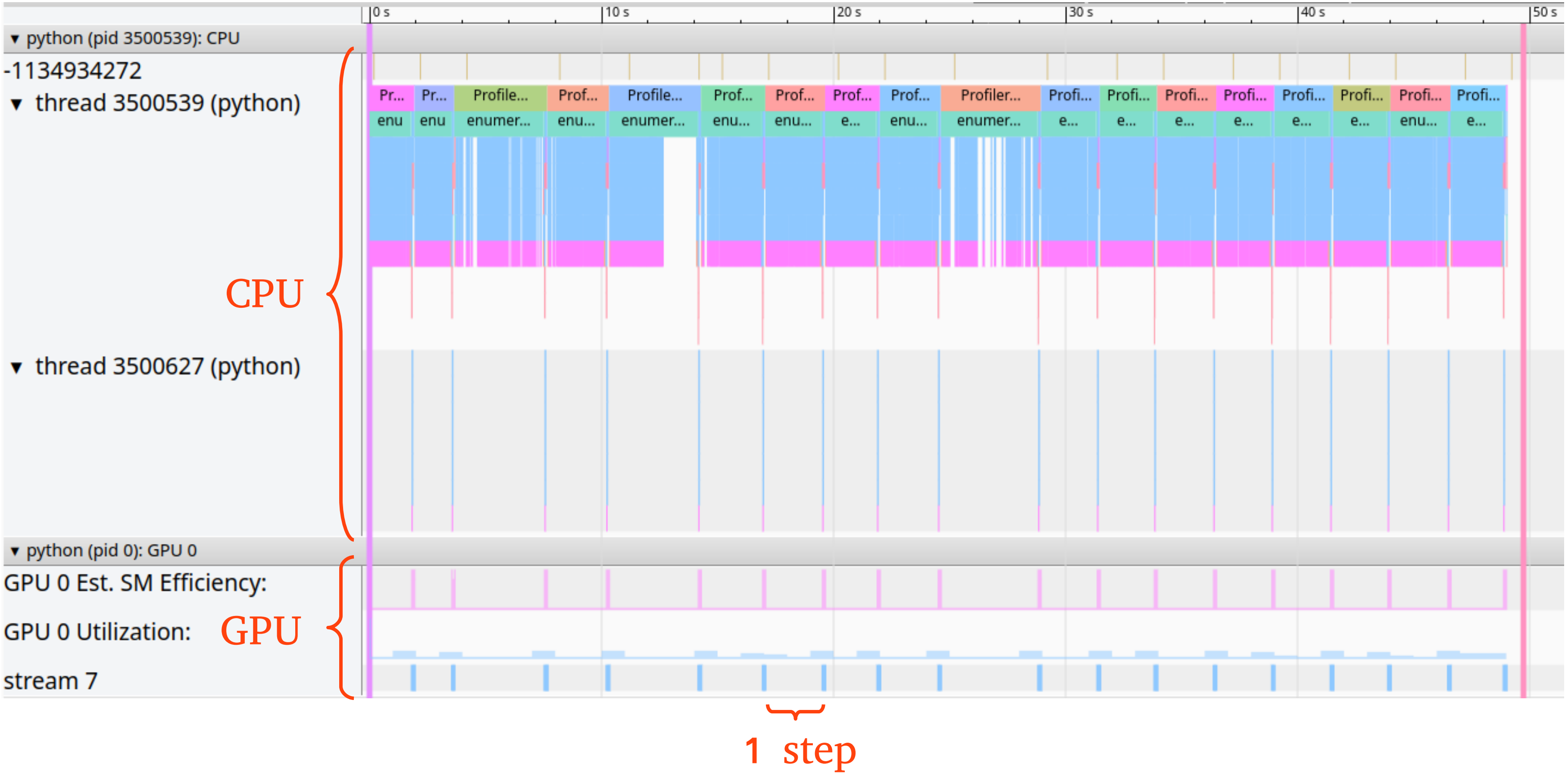
Total Time (us)



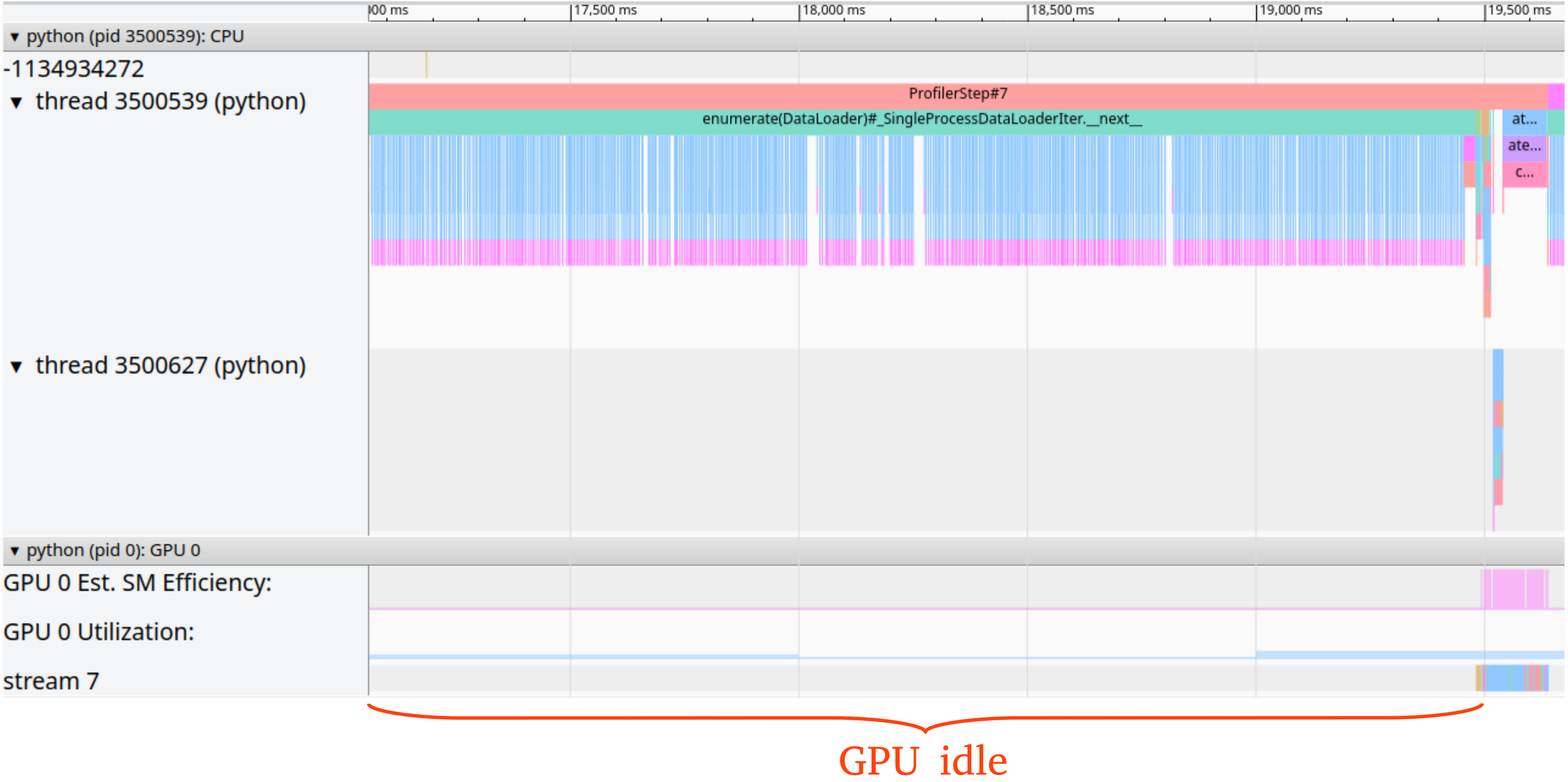
Tensor Cores Utilization



TP2_2: Profiler Trace

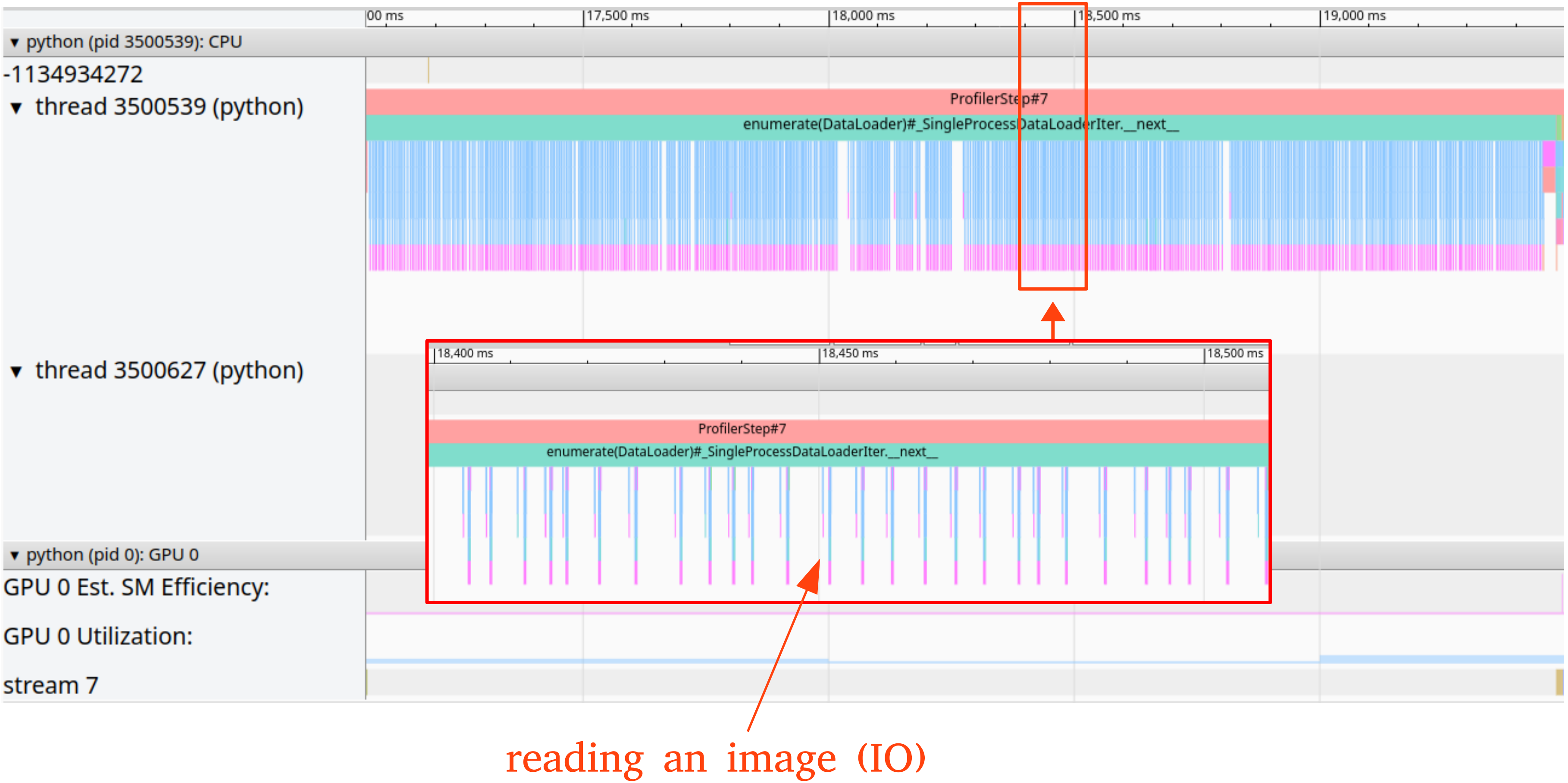


TP2_2: Profiler Trace (1 step)

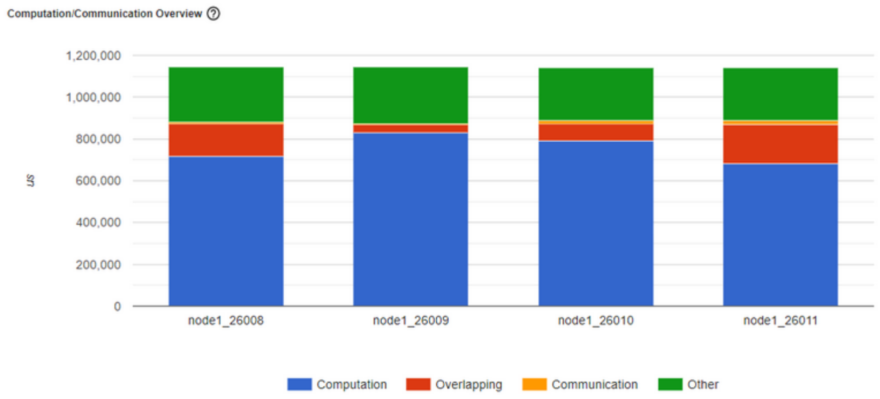


TP2_2: Profiler Trace (1 step - GPU)

TP2_2: Profiler Trace (1 step - CPU)



TP2_2: Profiler Distributed



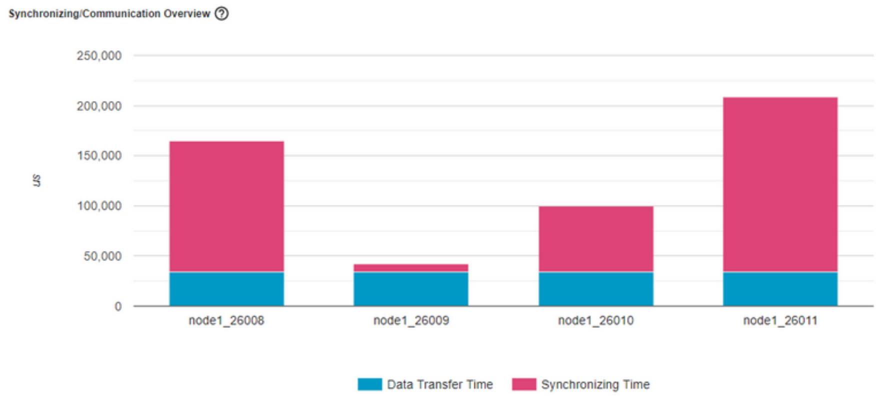
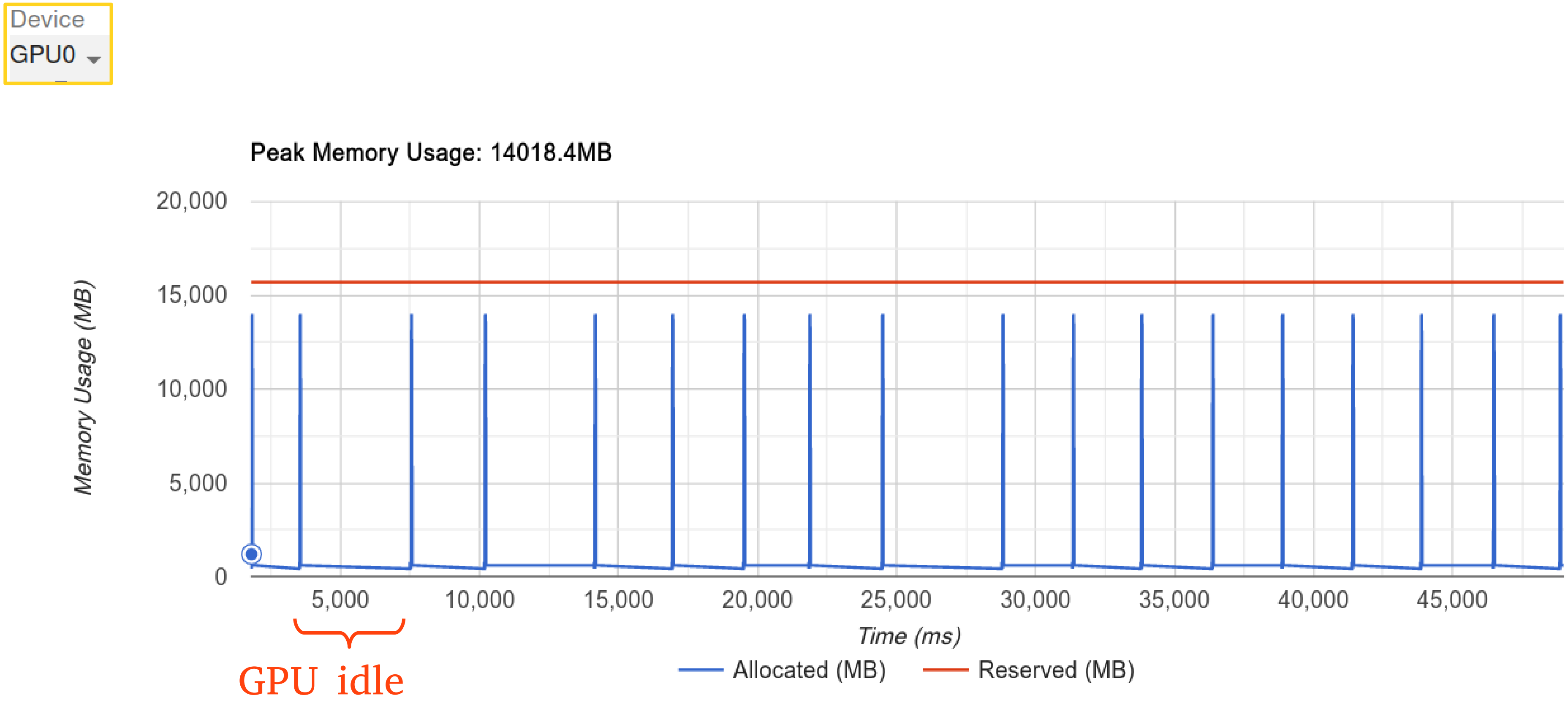


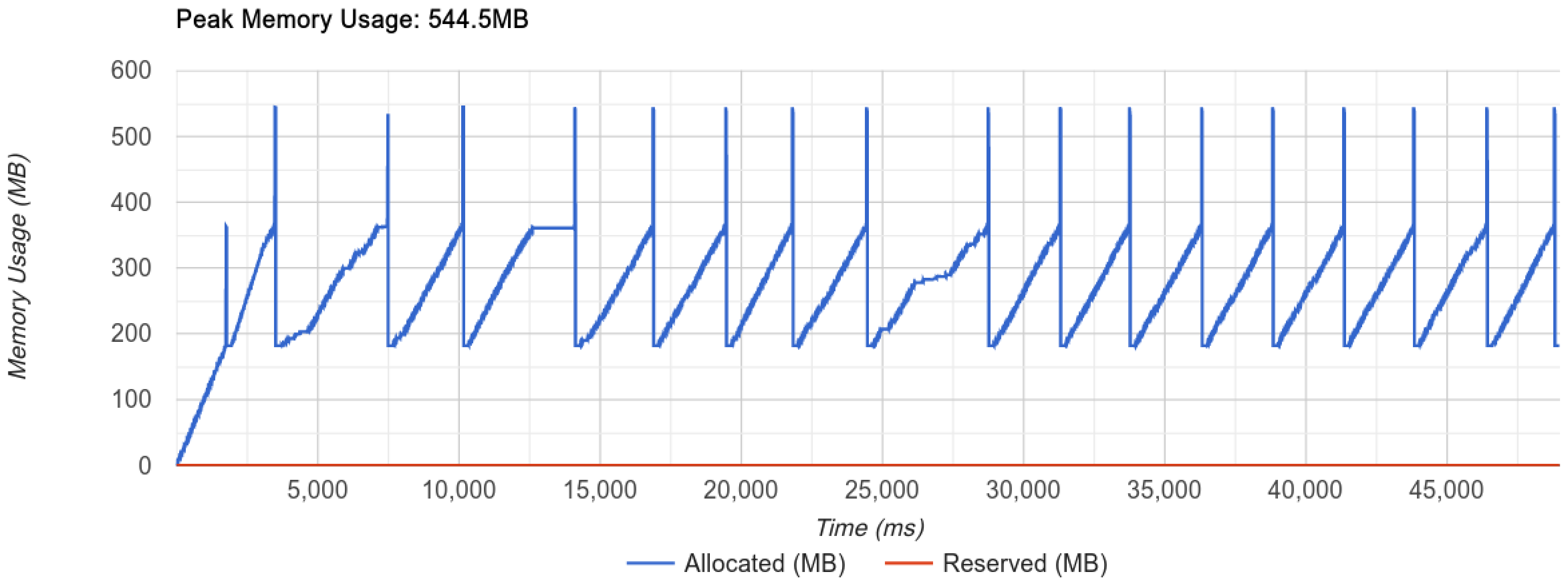
Image from the tutorial: https://pytorch.org/tutorials/intermediate/tensorboard_profiler_tutorial.html

TP2_2: Profiler Memory View (GPU)

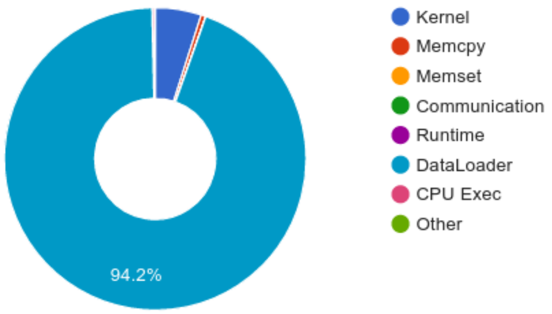


TP2_2: Profiler Memory View (CPU)

Device
CPU



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

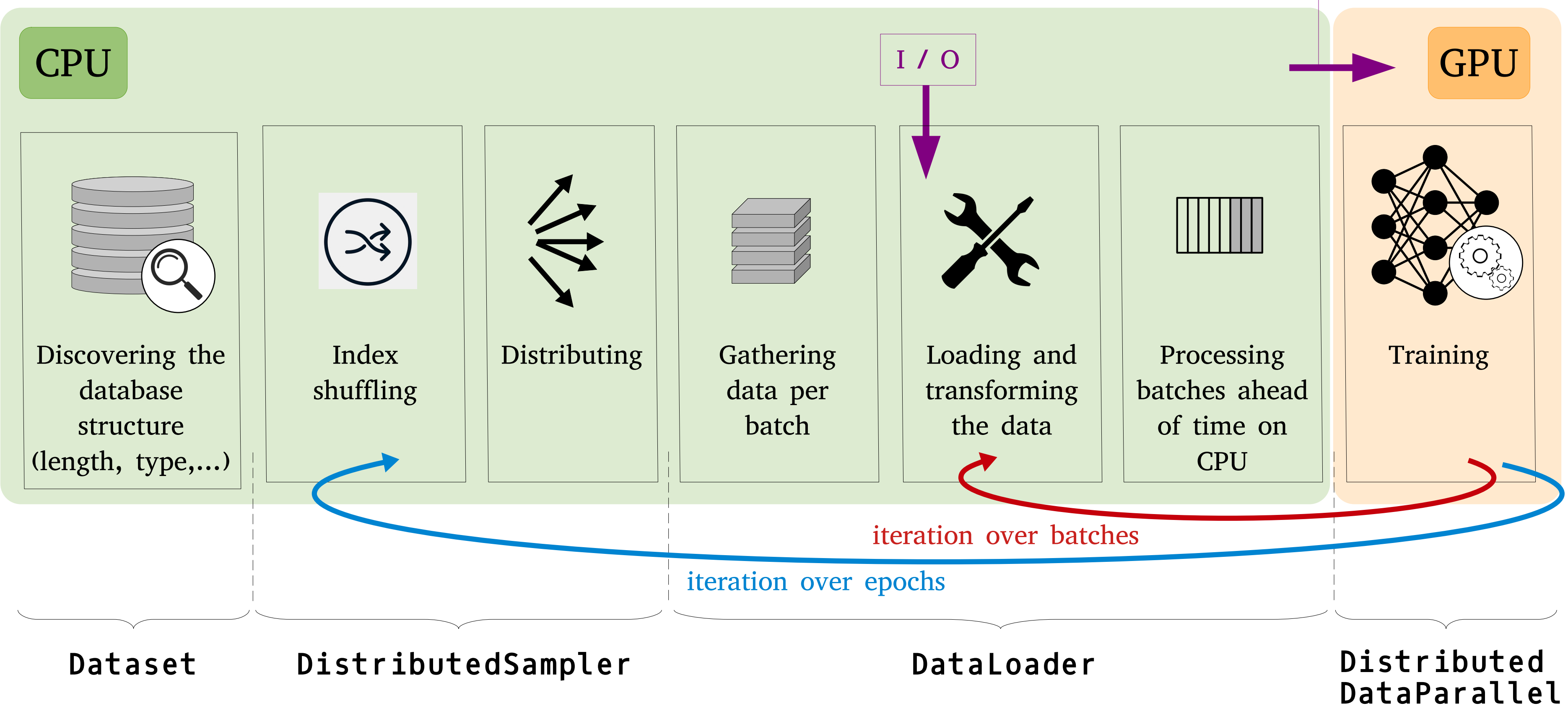
cnrs IDRIS

Optimization of the data preprocessing

Data preprocessing with DataLoader ◂

Optimization of the DataLoader ◂

Data preprocessing with 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>
                        )
```

Slurm

SLURM_NTASKS

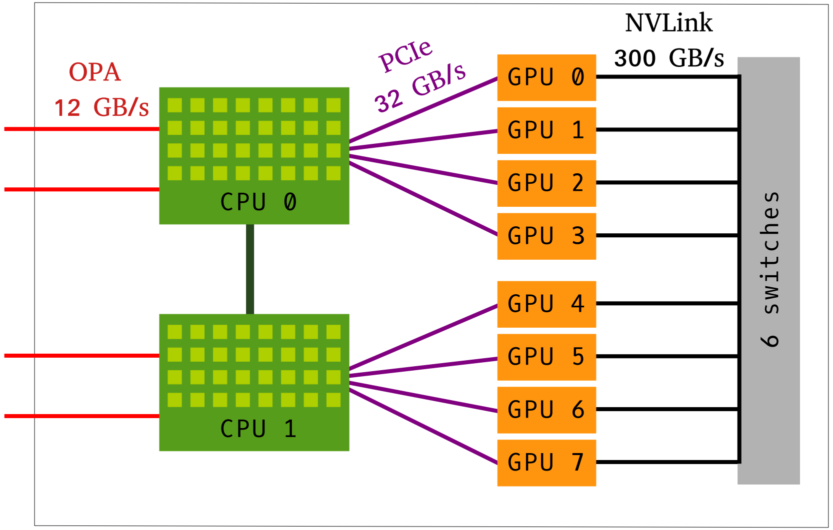
Optimization of the data preprocessing

Data preprocessing with DataLoader ◂

Optimization of the DataLoader ◂

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

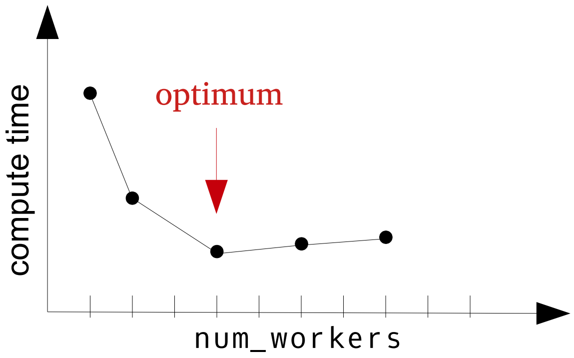
Optimization of the DataLoader

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

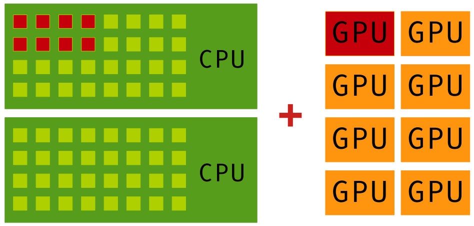
- `num_workers` allows us to define the number of processes (CPU cores) which will work in parallel to preprocess the data on the CPU.

✔ Compute time speedup on CPU.

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



Standard Slurm reservation
on a 8 × A100 node




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

Optimization of the DataLoader

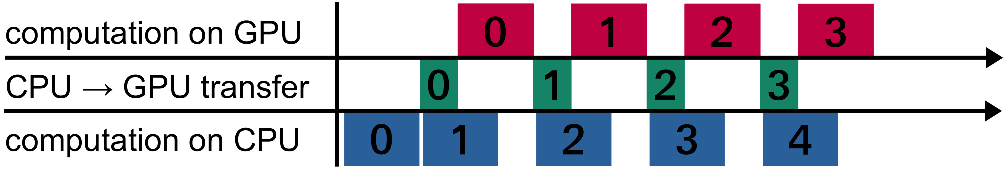
1. Loading the data in memory and transforming it on the CPU

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

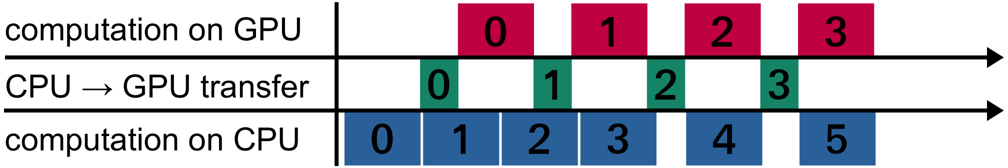
✓ Prevents GPU inactivity if CPU occasionally struggles

⚠ Usage of the CPU RAM

`prefetch_factor = 1`



`prefetch_factor = 2`



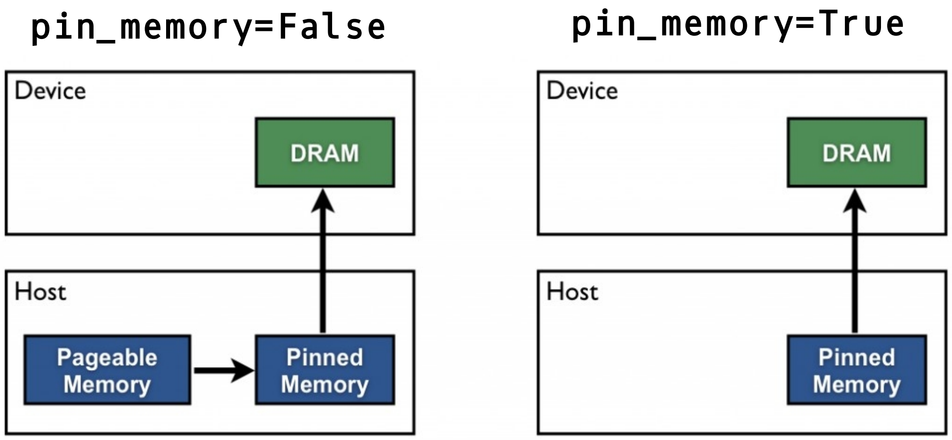
Optimization of the DataLoader

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



Pageable Data Transfer *Pinned Data Transfer*

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

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`

Optimization of the DataLoader

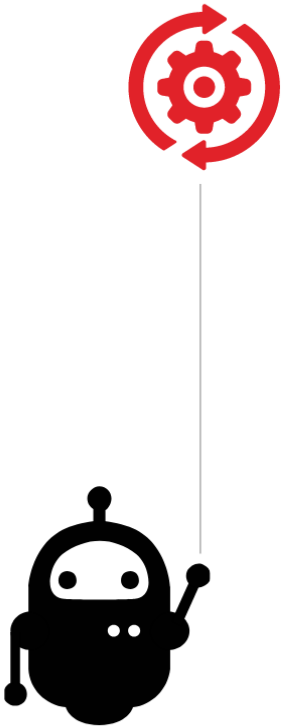
- Other DataLoader option:
  - `drop_last=True` allows us to ignore the last samples if the size of the dataset is not a multiple of the number of batches.

✓ The workload per process is balanced.

✓ We avoid the cost of treating an incomplete batch.

⚠ Loss of information?

TP2_3 : Optimization of the DataLoader

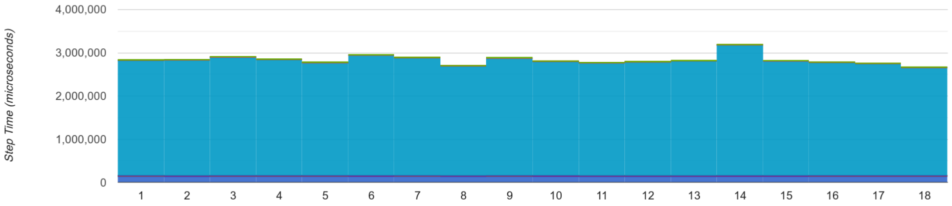


- Modify the DataLoader options.
- Measure the time gain on a few steps.

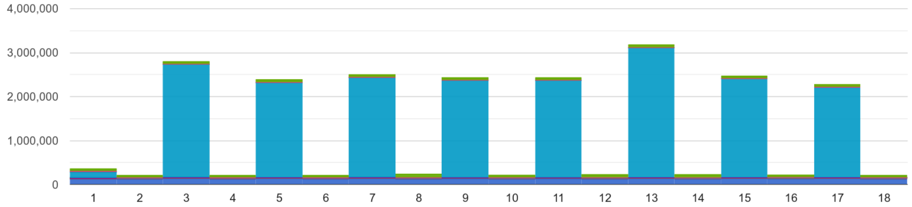
TP2_3 : Optimization of the DataLoader

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

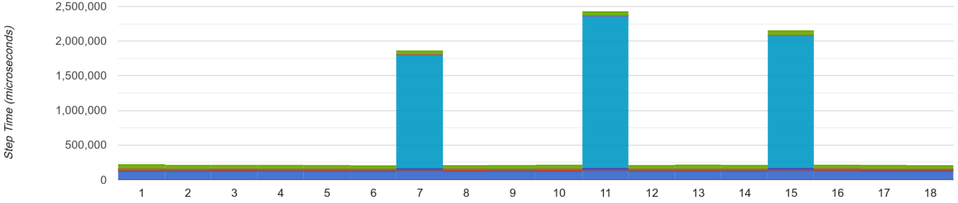
`num_workers=0`



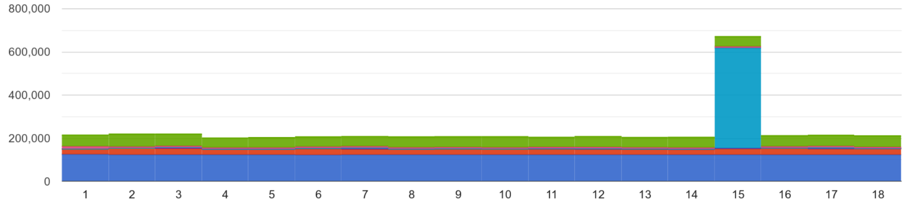
`num_workers=2`



`num_workers=4`

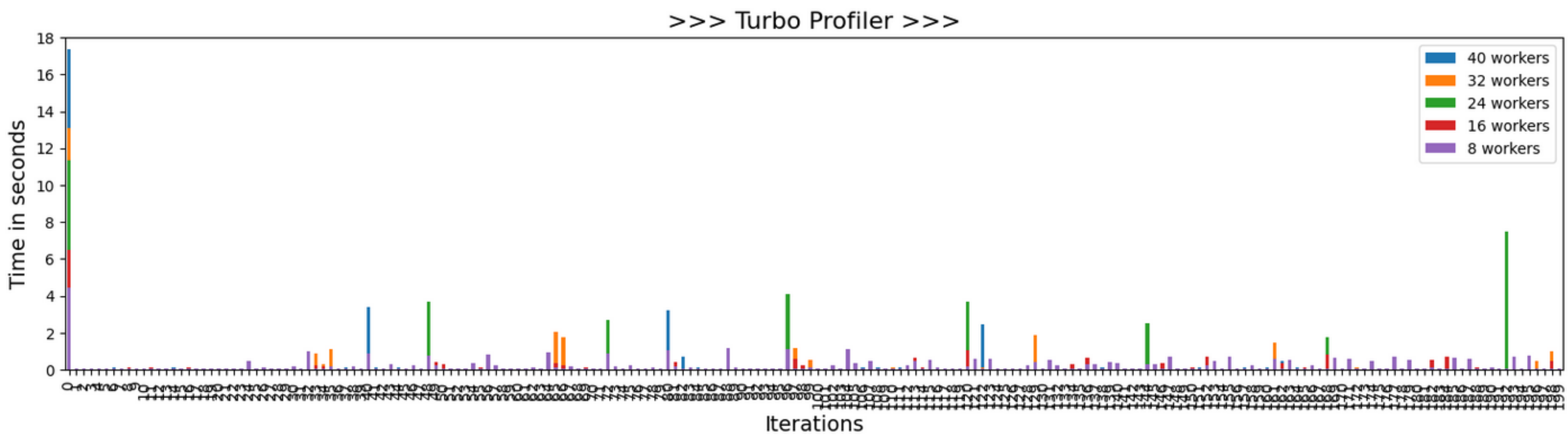


`num_workers=8`



Kernel Memcpy Memset Communication Runtime DataLoader CPU Exec Other

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

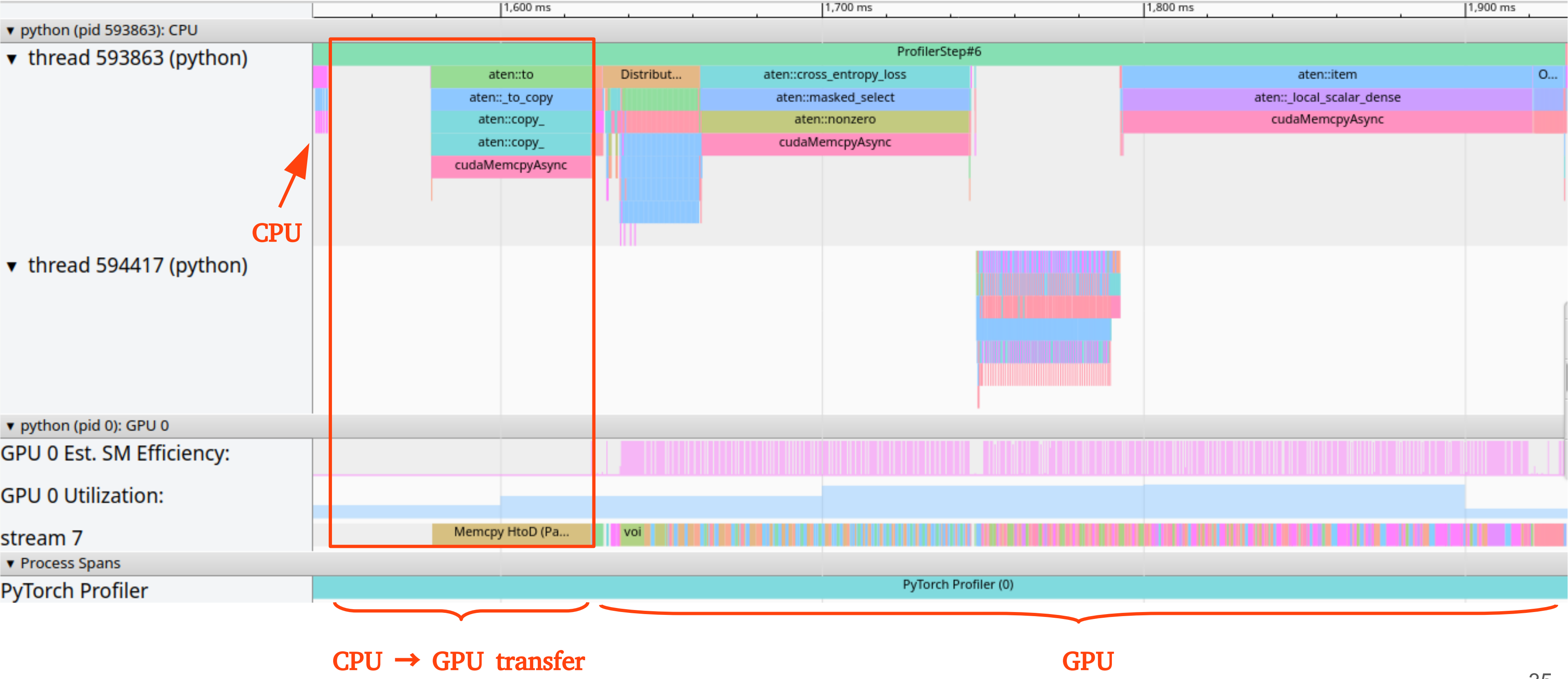
TP2_3 : Optimization of the DataLoader

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.

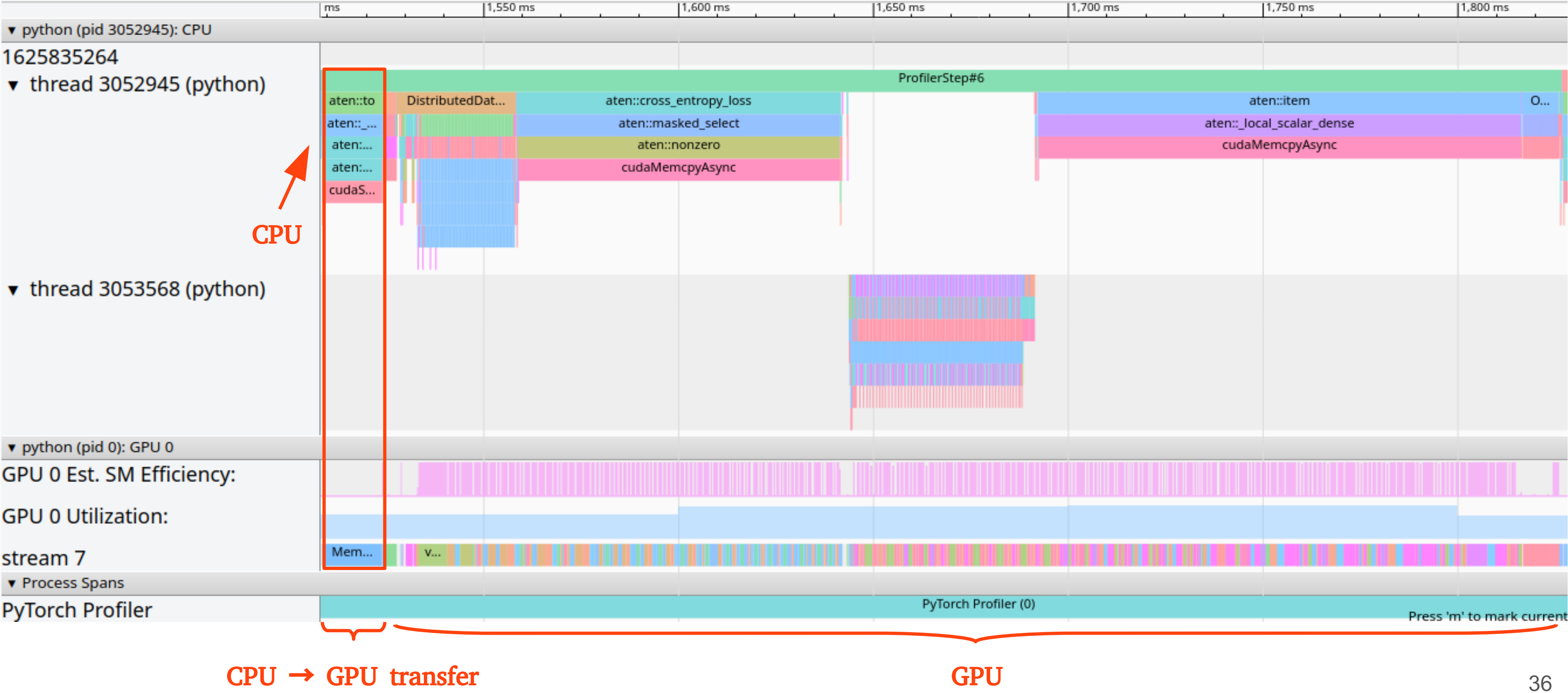
TP2_3 : Optimization of the DataLoader

`pin_memory=False, non_blocking=False`



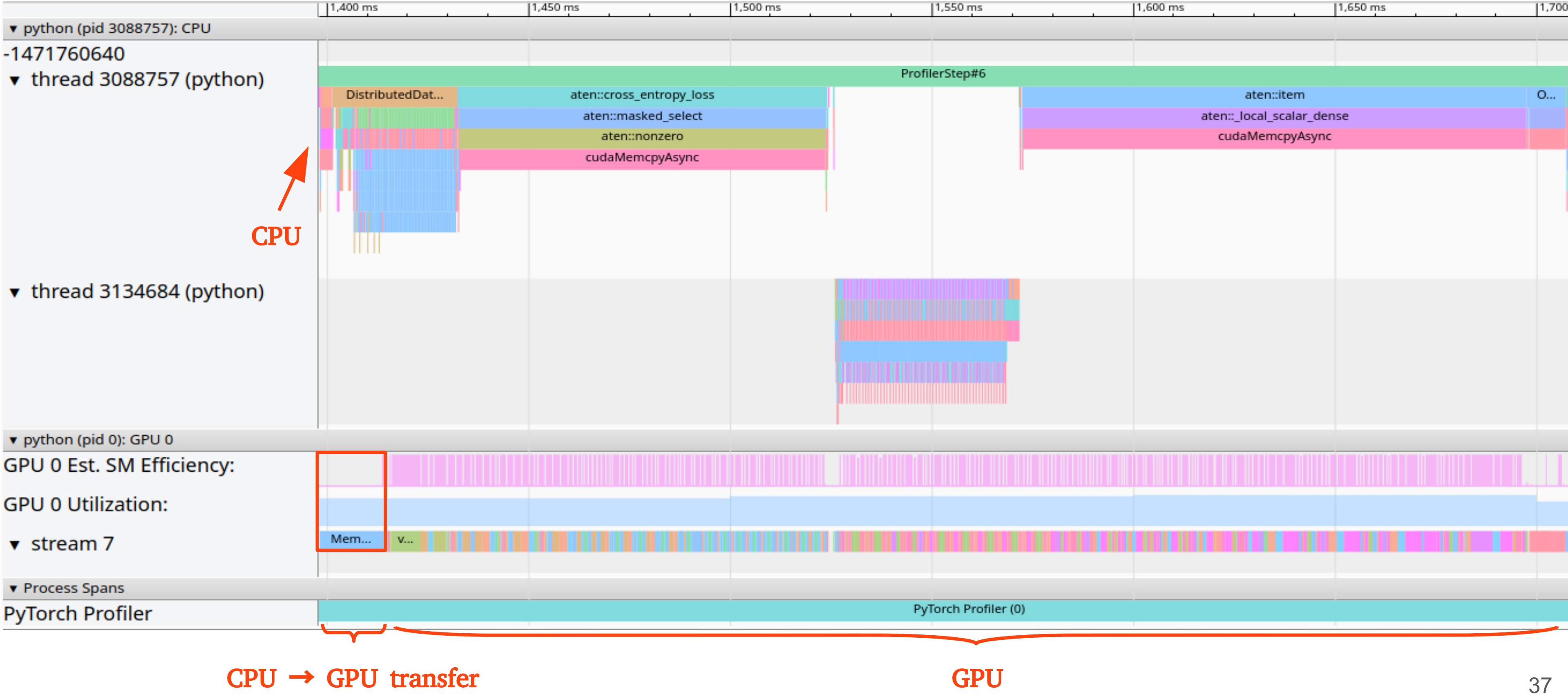
TP2_3 : Optimization of the DataLoader

`pin_memory=True, non_blocking=False`



TP2_3 : Optimization of the DataLoader

```
pin_memory=True, non_blocking=True
```



TP2_3: Optimization of the DataLoader

- Chosen optimizations:

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

Configuration

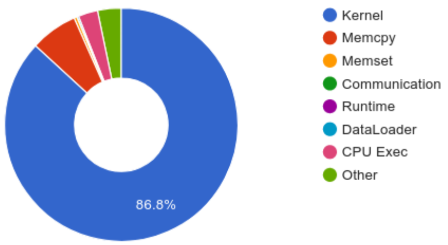
Number of Worker(s)	1
Device Type	GPU

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



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.