

Deep Learning Optimized on Jean Zay

Optimization of the data preprocessing

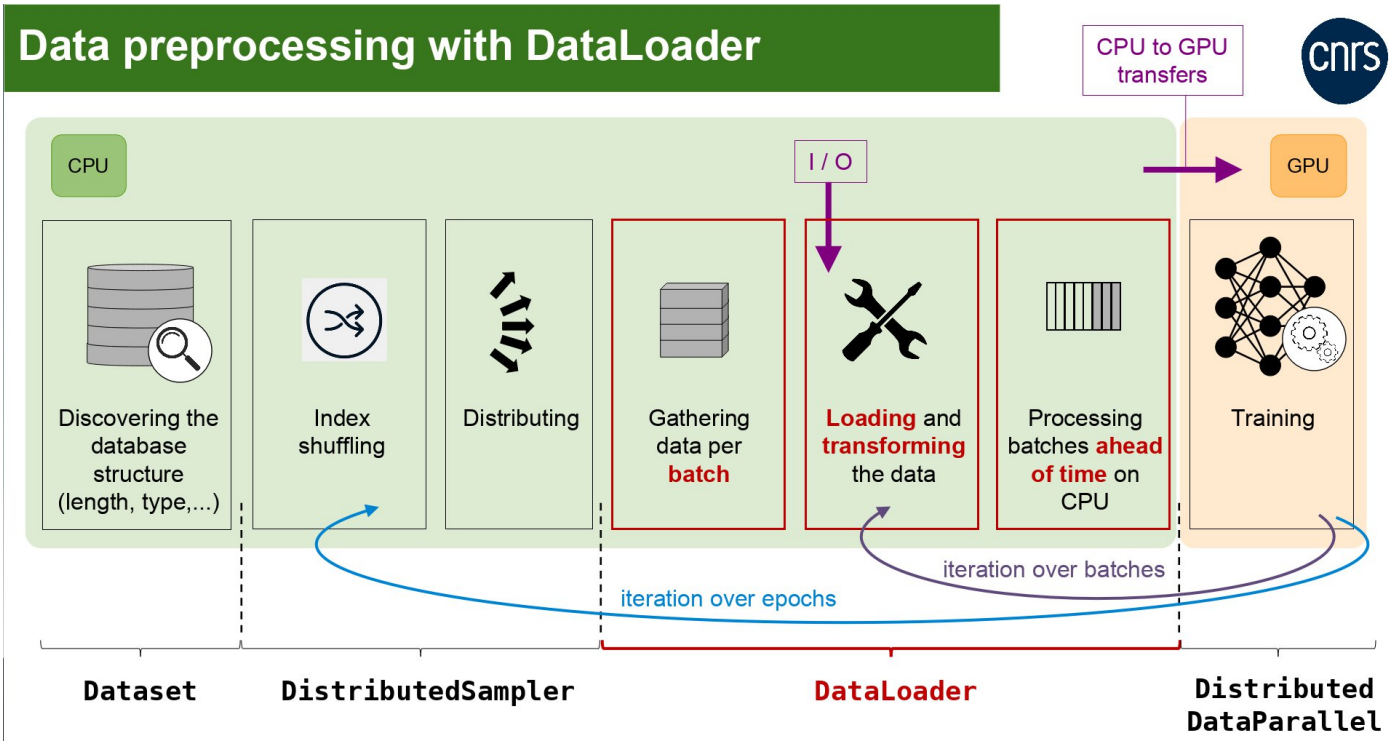
cnrs IDRIS

CC BY NC SA

Optimization of the data preprocessing

Data preprocessing with DataLoader ◂

Optimization of the DataLoader ◂



The complete training process

The Dataloader's role is to group the data indexed by the Sampler into batches, load it into memory from the disk (applying any necessary transformations) and then send it to the GPU for training.

Data preprocessing with DataLoader

- **DataLoader** (data preprocessing)

```
from torch.utils.data import DataLoader



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


The DataLoader class in PyTorch.

The dataset and **batch**_size parameters are hyperparameters of the training. They are decided in advance and do not change the dataloader's behaviour.

The rest of the parameters will be explained one by one in this course.

Optimization of the data preprocessing

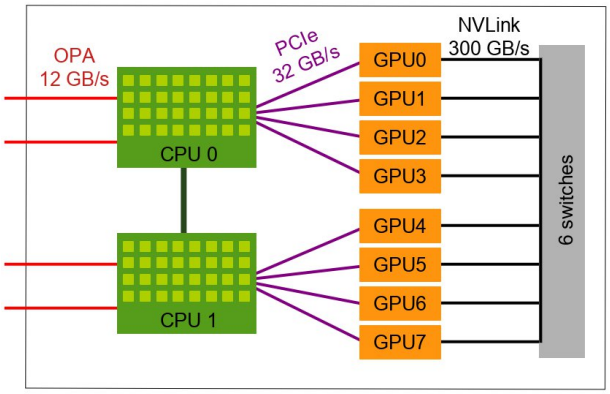
Data preprocessing with DataLoader ◂

Optimization of the DataLoader ◂

We will look at how the various Dataloader settings can improve the performance of input data loading and pre-processing by the CPU, and by extension, the performance of the training process as a whole.

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

DataLoader performance depends on:

i. CPU performance when loading and transforming data

ii. The time spent on CPU–GPU communication when transferring data

We will focus on these two aspects to optimise data loading

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.

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

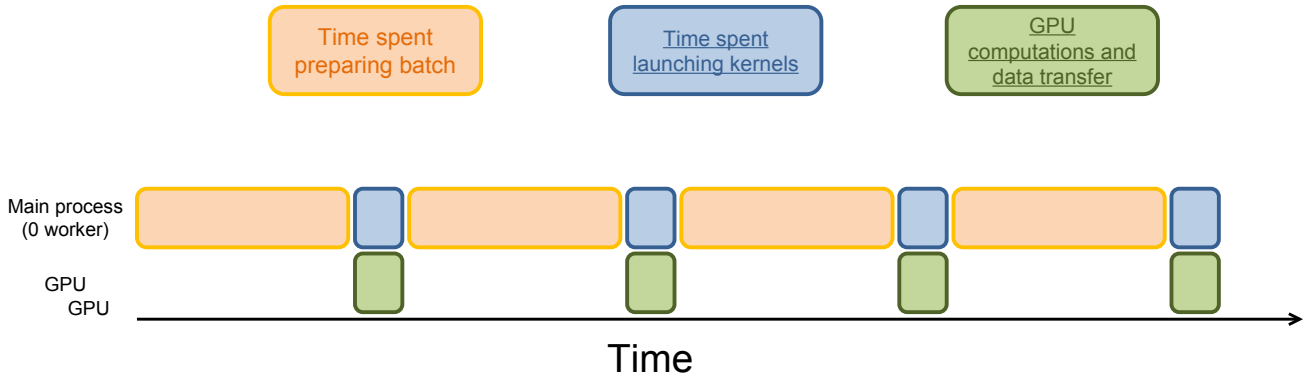

To speed up data loading, we can start by parallelising the work by increasing the number of active processes, known as ‘workers’.

Instead of pre-processing a single batch at a time, we will process num_worker batches simultaneously.

On the other hand, setting up a larger parallel processing environment consumes more RAM and CPU resources.

Optimization of the DataLoader

num_workers = 0

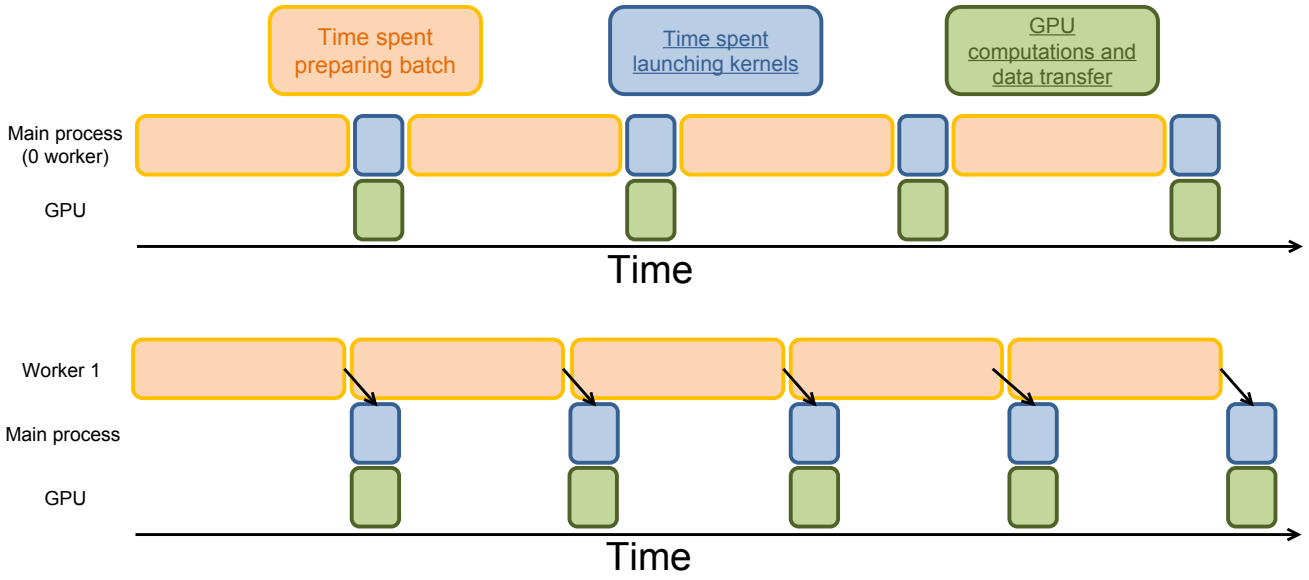


When num_workers=0, the main process handles both data loading and the orchestration of GPU operations, including the launch of CUDA kernels.

As these operations are sequential, the process cannot handle both them and the Dataloader at the same time.

Optimization of the DataLoader

num_workers = 1

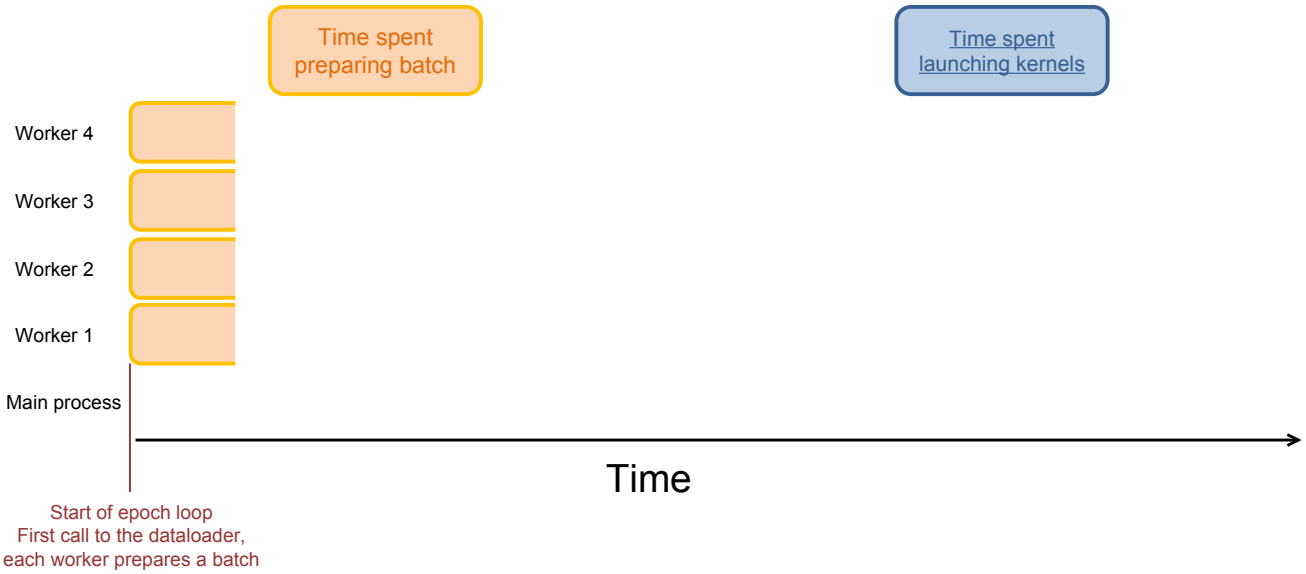


With `num_worker` set to 1, an additional process is created, the so-called 'worker', and the data-loading instructions are delegated to it. This worker can therefore run whilst the main process manages the orchestration of the GPU instructions.

Of course, in some cases the preparation time for the next batch is longer than the forward/backward run time – which prevents us from chaining them together as quickly as we would like – but parallelising this work is a first step towards optimising data loading.

Optimization of the DataLoader

num_workers = 4
n_batches = 12



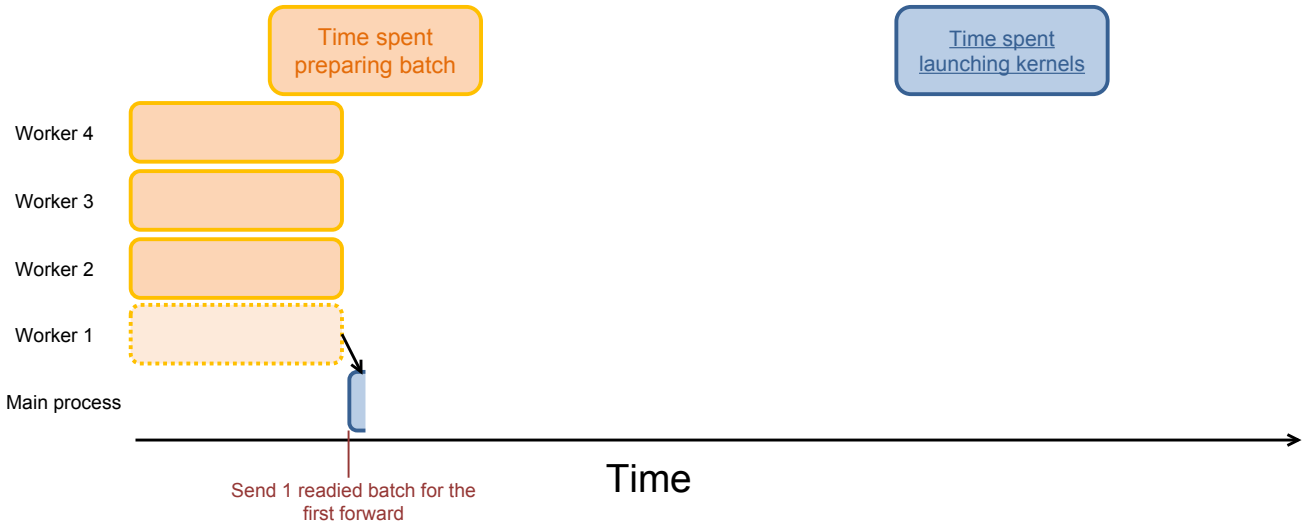
If loading the batches takes too long, the `num_workers` parameter can be increased further so that several batches can be prepared in parallel at the same time. The aim is to reduce the waiting time between two GPU computations by ensuring that a batch is available for a forward pass as soon as the previous one has finished.

On the first call to the Dataloader (typically at the start of the "*for **batch** in **dataloader***" loop within an epoch), all workers are created and each begins to prepare a batch.

In this example, each worker prepares a batch and waits for it to be "consumed" before preparing another one.

Optimization of the DataLoader

num_workers = 4
n_batches = 12

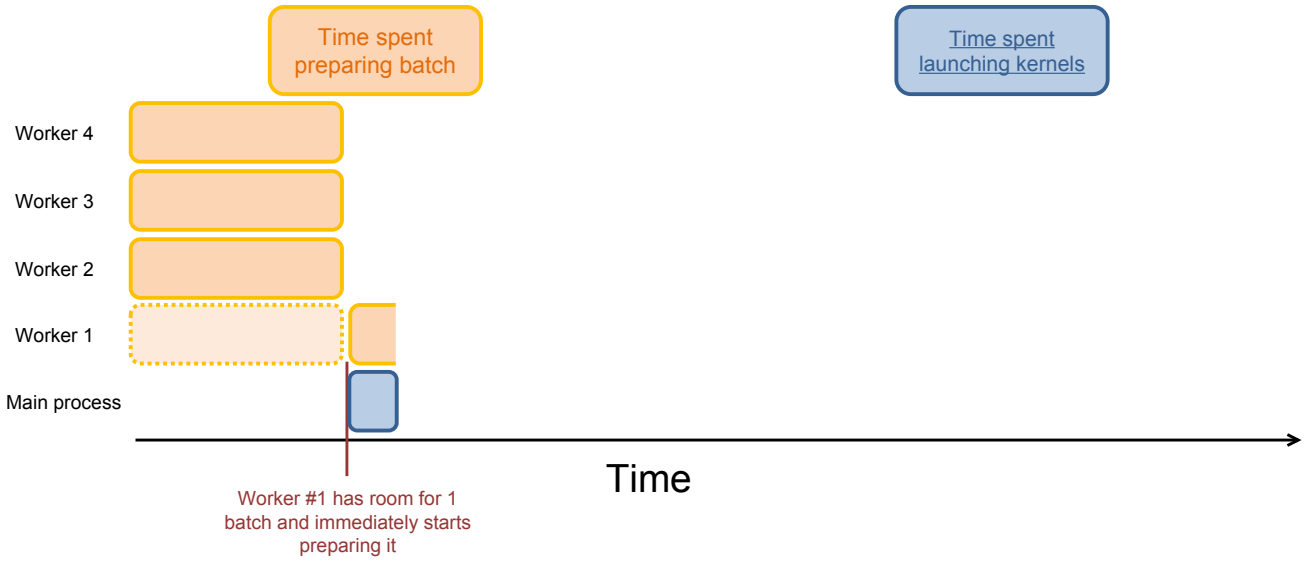


If loading the batches takes too long, the `num_workers` parameter can be increased further so that several batches can be prepared in parallel at the same time. The aim is to reduce the waiting time between two GPU computations by ensuring that a batch is available for a forward pass as soon as the previous one has finished.

As we have at least one batch ready, we can start the first forward.

Optimization of the DataLoader

num_workers = 4
n_batches = 12

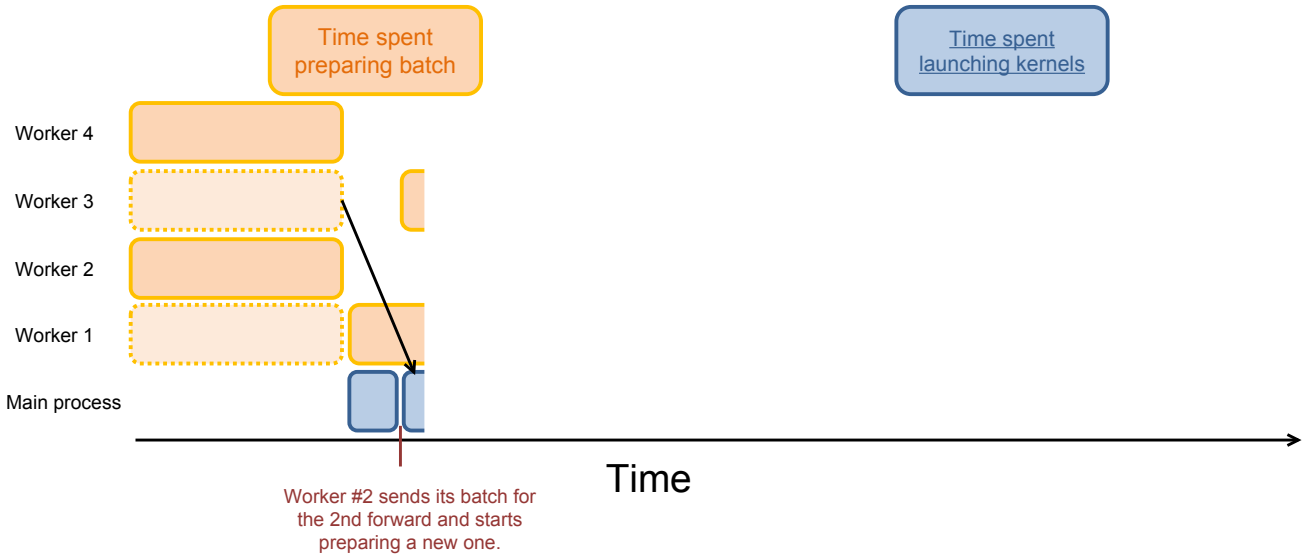


If loading the batches takes too long, the `num_workers` parameter can be increased further so that several batches can be prepared in parallel at the same time. The aim is to reduce the waiting time between two GPU computations by ensuring that a batch is available for a forward pass as soon as the previous one has finished.

As worker #1 has sent its first batch, it begins preparing the next one.

Optimization of the DataLoader

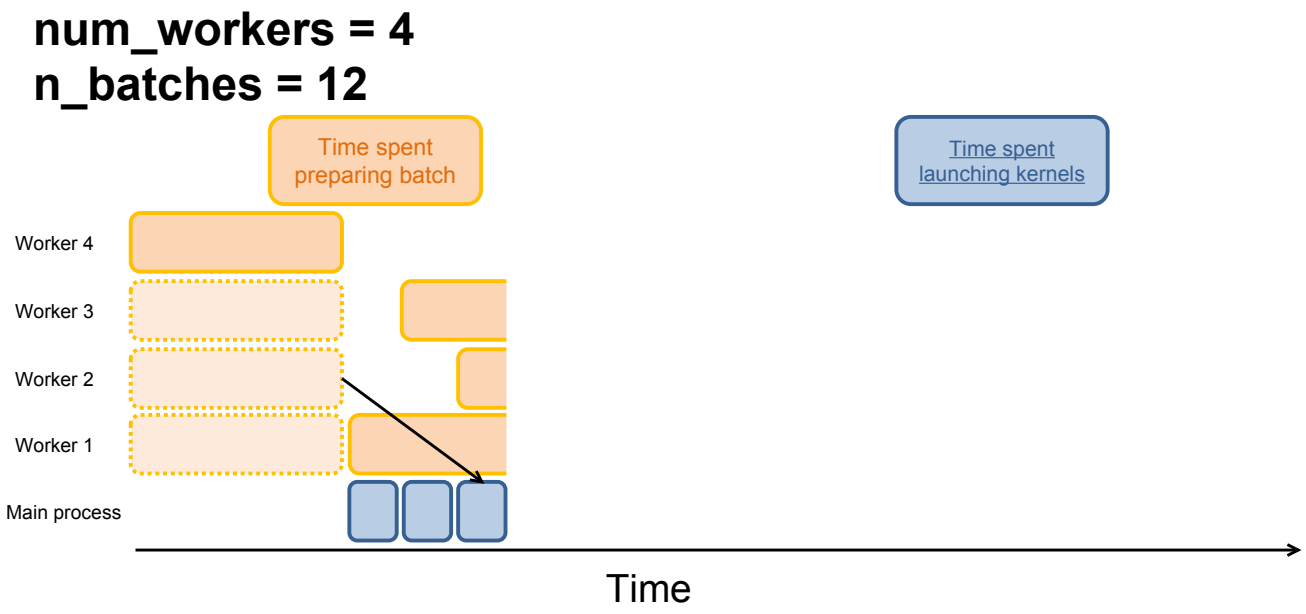
num_workers = 4
n_batches = 12



If loading the batches takes too long, the `num_workers` parameter can be increased further so that several batches can be prepared in parallel at the same time. The aim is to reduce the waiting time between two GPU computations by ensuring that a batch is available for a forward pass as soon as the previous one has finished.

At the start of the next forward, even if the first worker has not finished its task, we still have three batches in reserve to send.

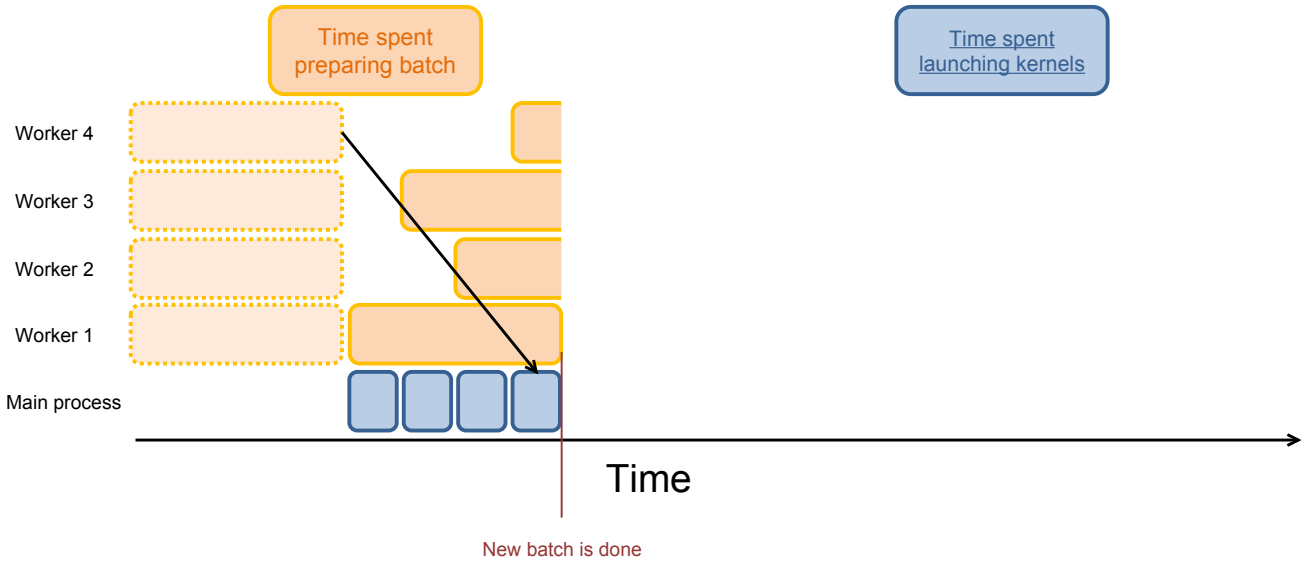
Optimization of the DataLoader



If loading the batches takes too long, the `num_workers` parameter can be increased further so that several batches can be prepared in parallel at the same time. The aim is to reduce the waiting time between two GPU computations by ensuring that a batch is available for a forward pass as soon as the previous one has finished.

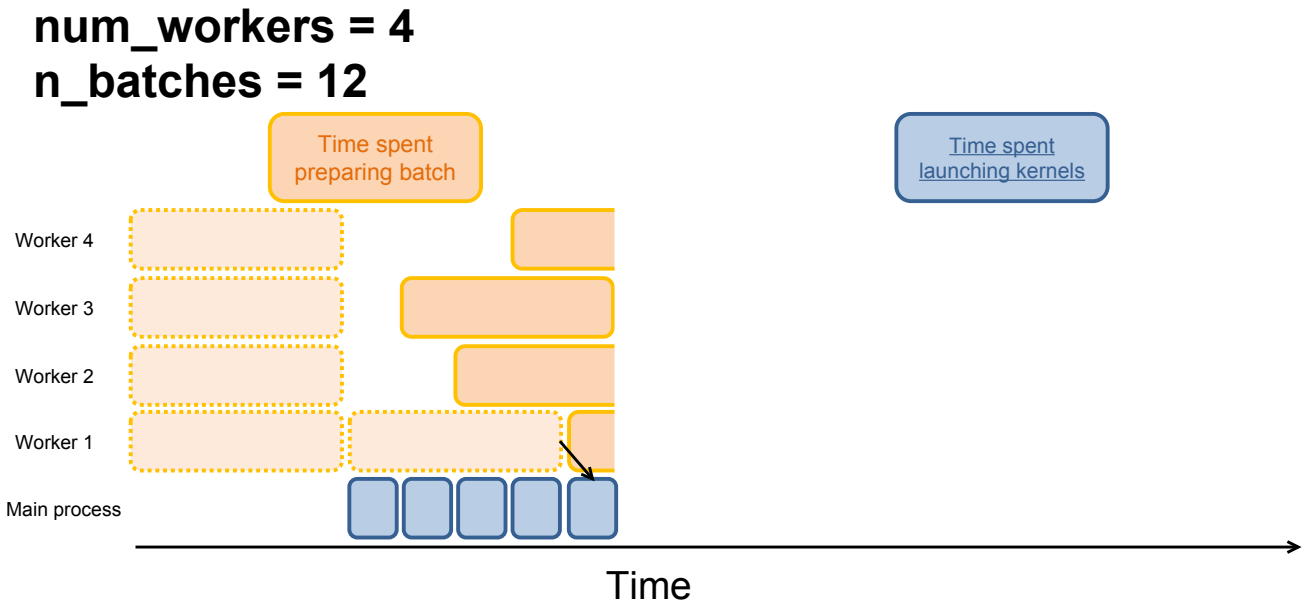
Optimization of the DataLoader

num_workers = 4
n_batches = 12



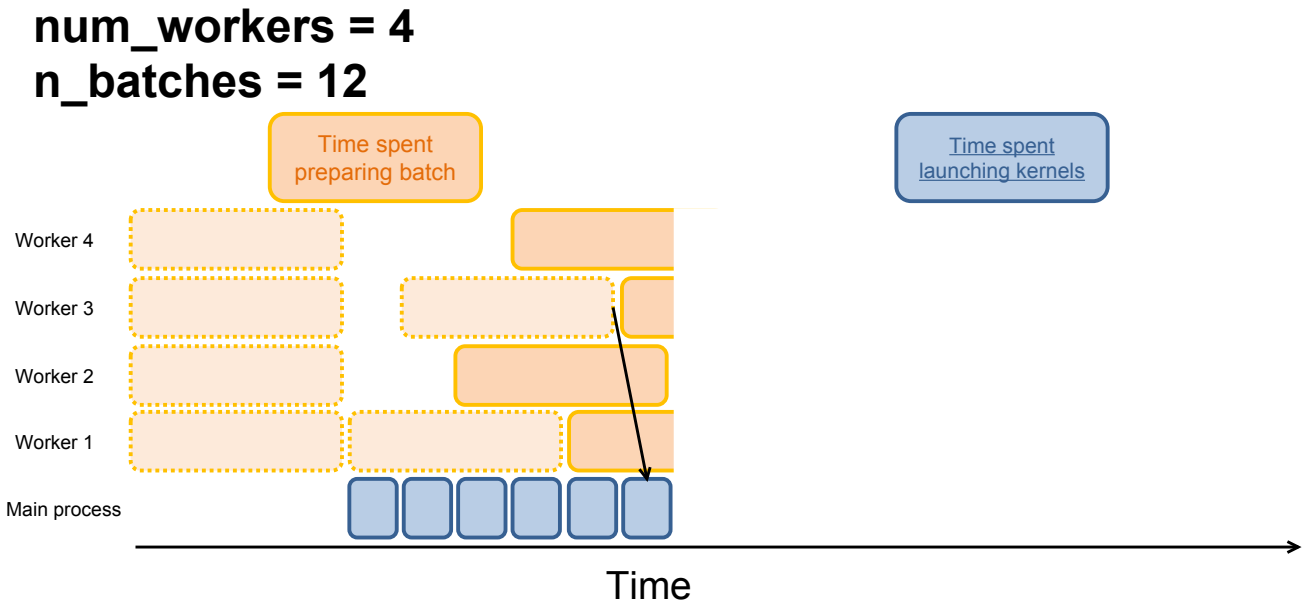
If loading the batches takes too long, the `num_workers` parameter can be increased further so that several batches can be prepared in parallel at the same time. The aim is to reduce the waiting time between two GPU computations by ensuring that a batch is available for a forward pass as soon as the previous one has finished.

Optimization of the DataLoader



If loading the batches takes too long, the `num_workers` parameter can be increased further so that several batches can be prepared in parallel at the same time. The aim is to reduce the waiting time between two GPU computations by ensuring that a batch is available for a forward pass as soon as the previous one has finished.

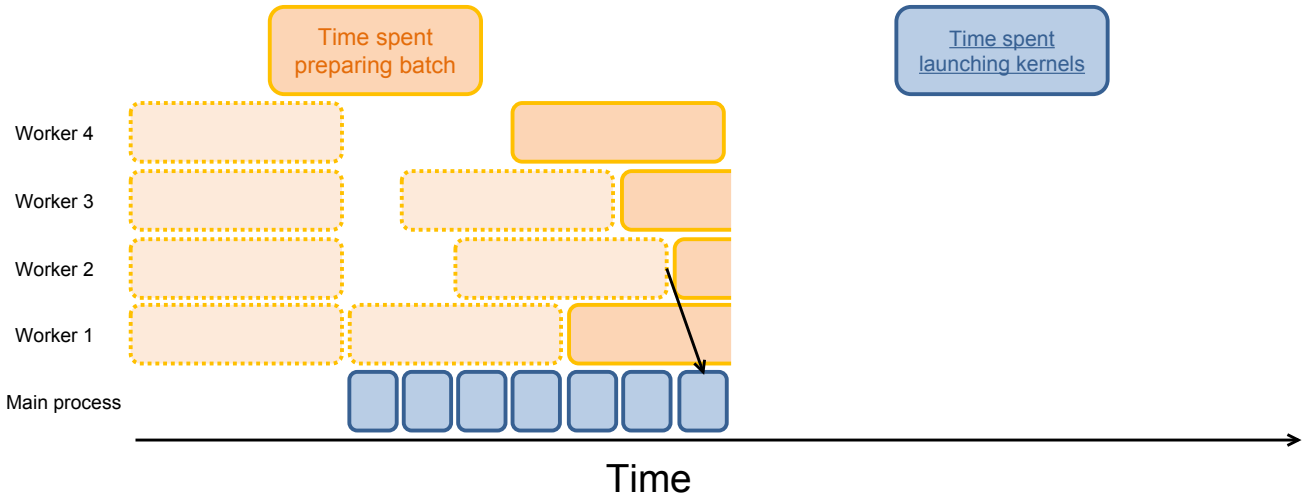
Optimization of the DataLoader



If loading the batches takes too long, the `num_workers` parameter can be increased further so that several batches can be prepared in parallel at the same time. The aim is to reduce the waiting time between two GPU computations by ensuring that a batch is available for a forward pass as soon as the previous one has finished.

Optimization of the DataLoader

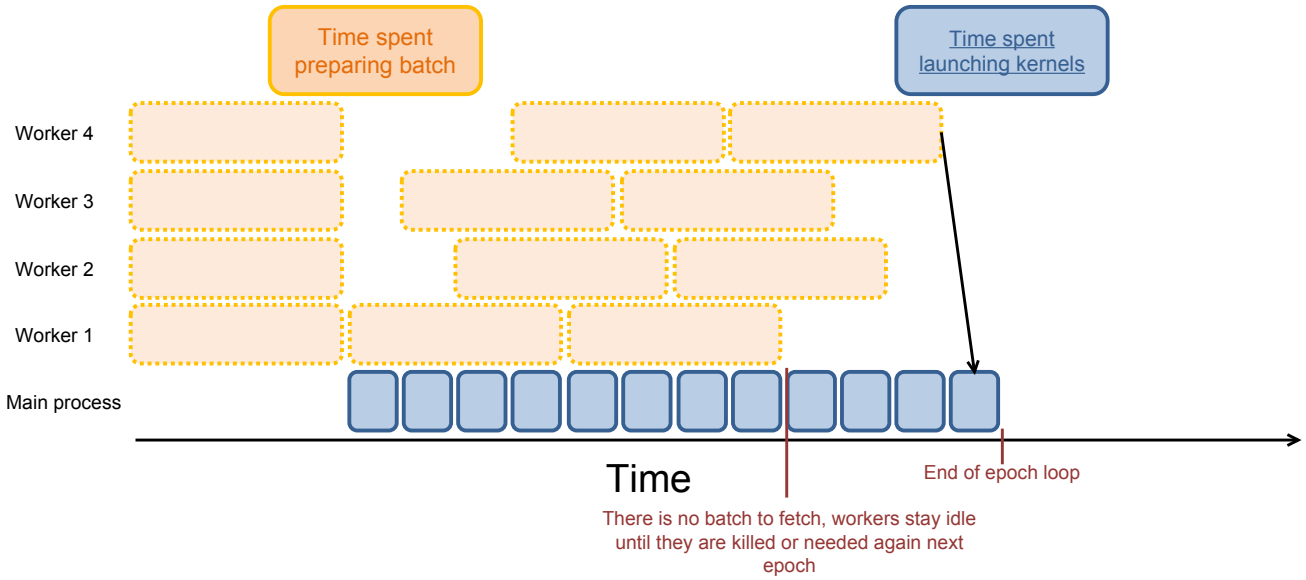
num_workers = 4
n_batches = 12



If loading the batches takes too long, the `num_workers` parameter can be increased further so that several batches can be prepared in parallel at the same time. The aim is to reduce the waiting time between two GPU computations by ensuring that a batch is available for a forward pass as soon as the previous one has finished.

Optimization of the DataLoader

num_workers = 4
n_batches = 12

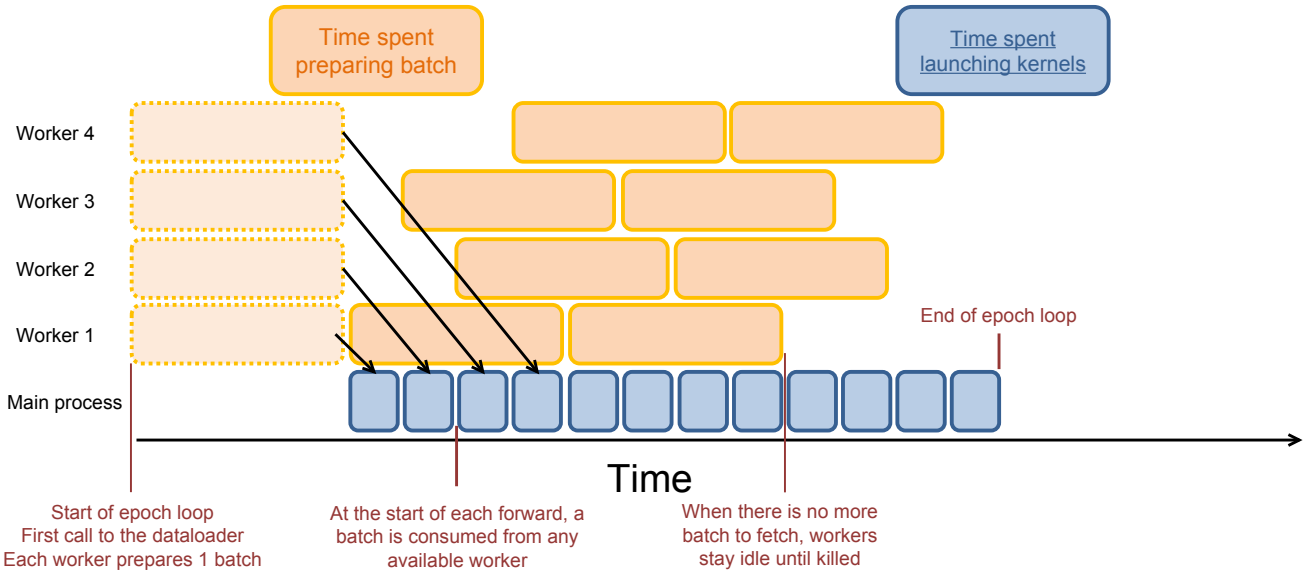


If loading the batches takes too long, the `num_workers` parameter can be increased further so that several batches can be prepared in parallel at the same time. The aim is to reduce the waiting time between two GPU computations by ensuring that a batch is available for a forward pass as soon as the previous one has finished.

Once all the batches have been prepared and sent, the workers remain idle until their process is terminated, or until the next epoch (depending on another parameter).

Optimization of the DataLoader

num_workers = 4
n_batches = 12



If loading the batches takes too long, the `num_workers` parameter can be increased further so that several batches can be prepared in parallel at the same time. The aim is to reduce the waiting time between two GPU computations by ensuring that a batch is available for a forward pass as soon as the previous one has finished.

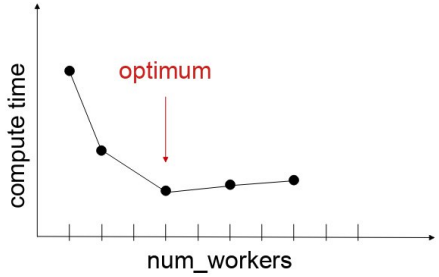
Optimization of the DataLoader

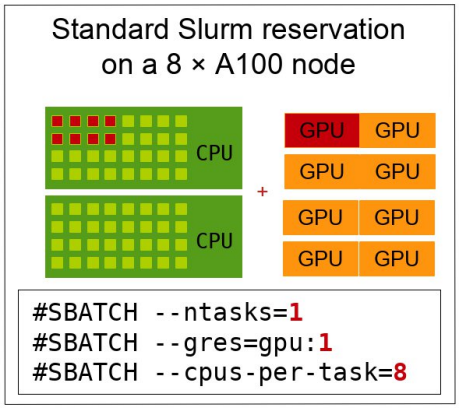
1. Loading the data in memory and transforming it on the CPU

- **`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.





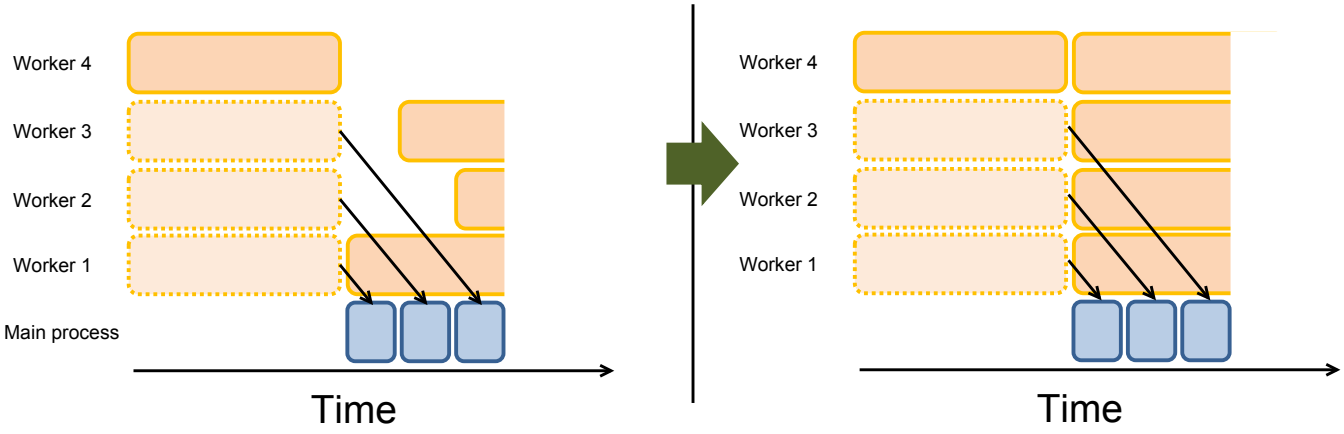
Of course, it's all a matter of compromise. Increasing the number of workers increases the memory footprint of the multiprocessing environment.

Furthermore, if the number of workers is too high, you run into problems of thread overhead (when the CPU's physical cores have to coordinate multiple workers) and concurrent read access to storage, which is limited by the storage's speed.

Typically, you would aim for one worker per physical core, but you should avoid exceeding this number.

Optimization of the DataLoader

Can we fetch a batch in advance ?



If the number of workers is increased, some may find themselves idle at times if no batches are requested. Is it therefore possible to pre-load one or more batches in advance?

Optimization of the DataLoader

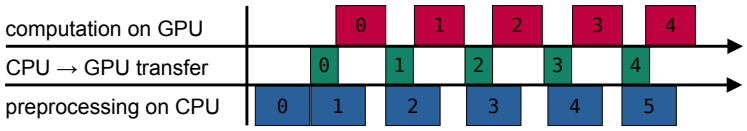
1. Loading the data in memory and transforming it on the CPU

- **`prefetch_factor`** allows us to define the maximum number of batches **each** worker can prepare in advance.

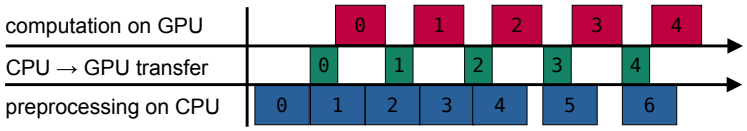
✓ Prevents GPU inactivity if CPU occasionally struggles

⚠ Usage of the CPU RAM

`prefetch_factor = 1`



`prefetch_factor = 2`



The `prefetch_factor` parameter is used to specify the number of batches to be kept ready in advance at all times, in order to minimise periods of inactivity.

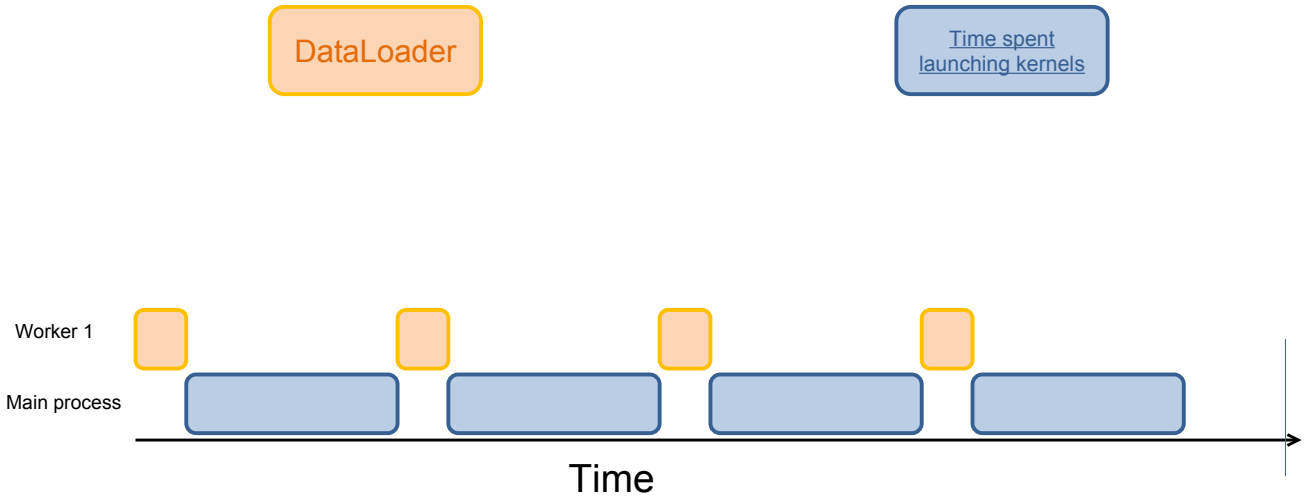
Each worker will prepare as many batches as specified by this parameter, for a total of `**num_workers**` × `**prefetch_factor**`.

The main advantage is that batches are prepared in advance whilst CPU processes are running smoothly, ensuring there are some in reserve should the CPU ever slow down.

Of course we have to ensure we have enough CPU RAM to hold the requested number of batches.

Optimization of the DataLoader

num_workers = 1, prefetch_factor = None

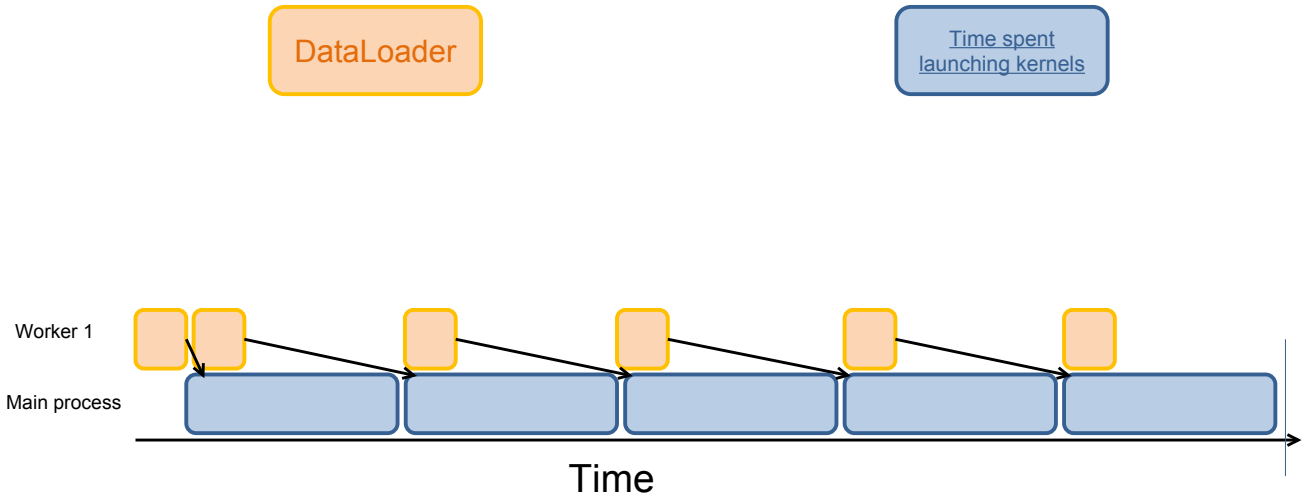


Example with `**prefetch=None**`

Batches are only prepared when they are needed. Nothing is done in parallel, and time is wasted.

Optimization of the DataLoader

num_workers = 1, prefetch_factor = 1



Example with `**prefetch=1**`

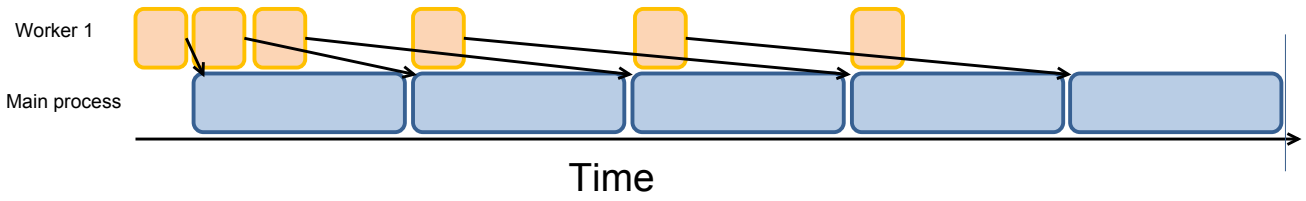
In this case, we want to be at least one batch ahead so as to avoid any downtime. As soon as the batch is processed, the worker immediately prepares a new one.

Optimization of the DataLoader

num_workers = 1, prefetch_factor = 2

DataLoader

Time spent launching kernels

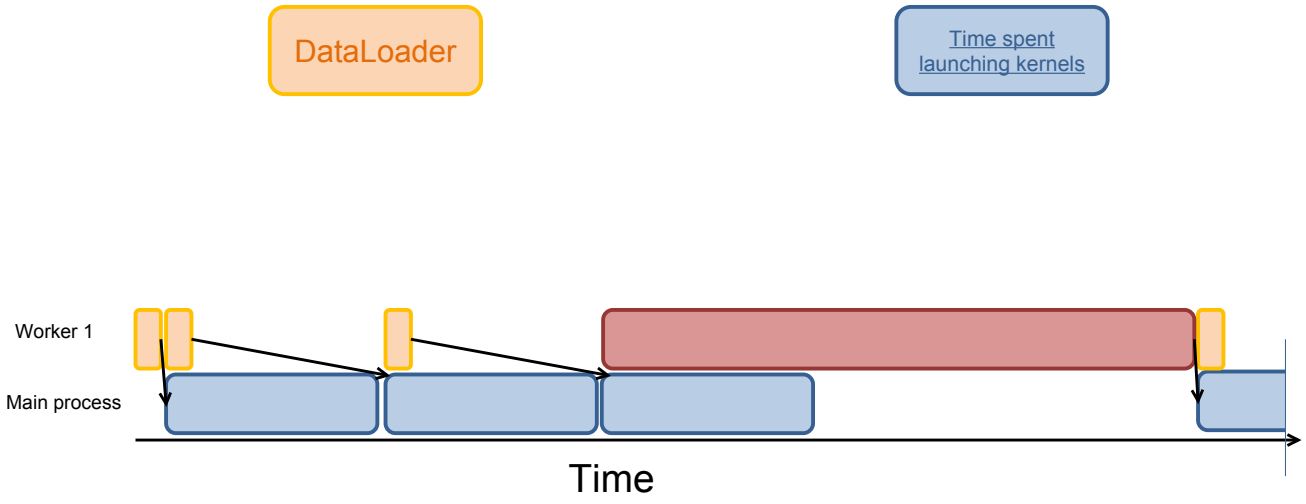


Example with `**prefetch=2**`

Similar to prefetch=1.

Optimization of the DataLoader

num_workers = 1, prefetch_factor = 1

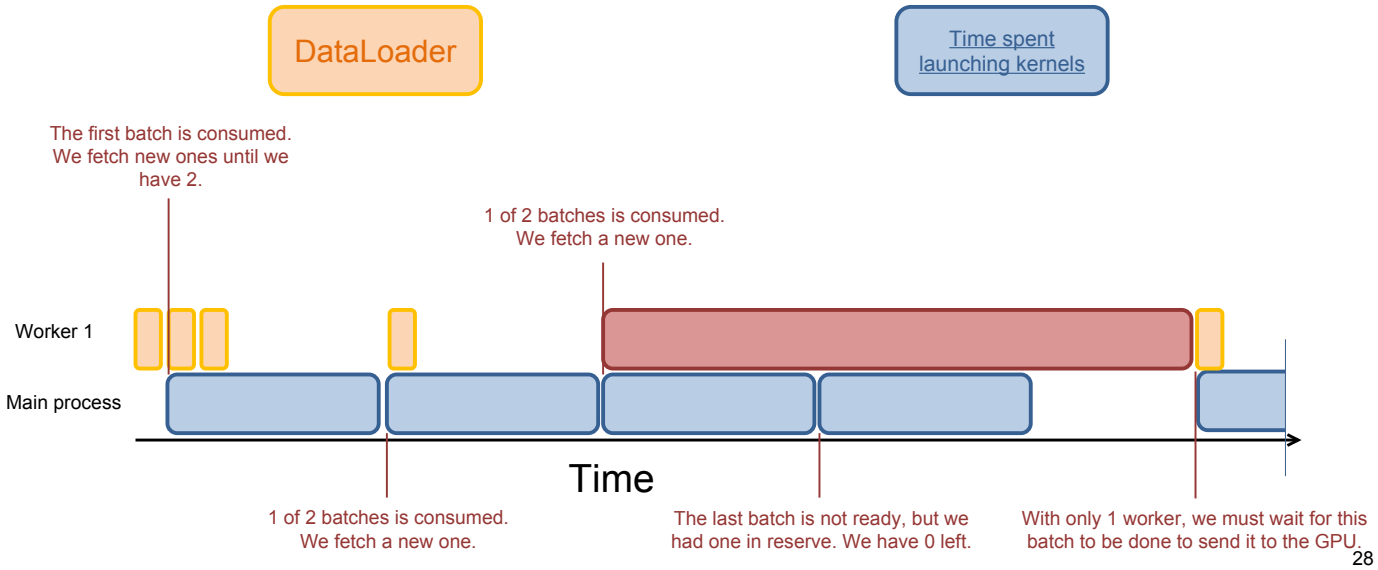


For example, when `**prefetch=1**` and the CPU is suddenly slowed down.

In this case, the CPU process becomes a bottleneck. The lack of a pre-prepared batch forces us to wait.

Optimization of the DataLoader

num_workers = 1, prefetch_factor = 2

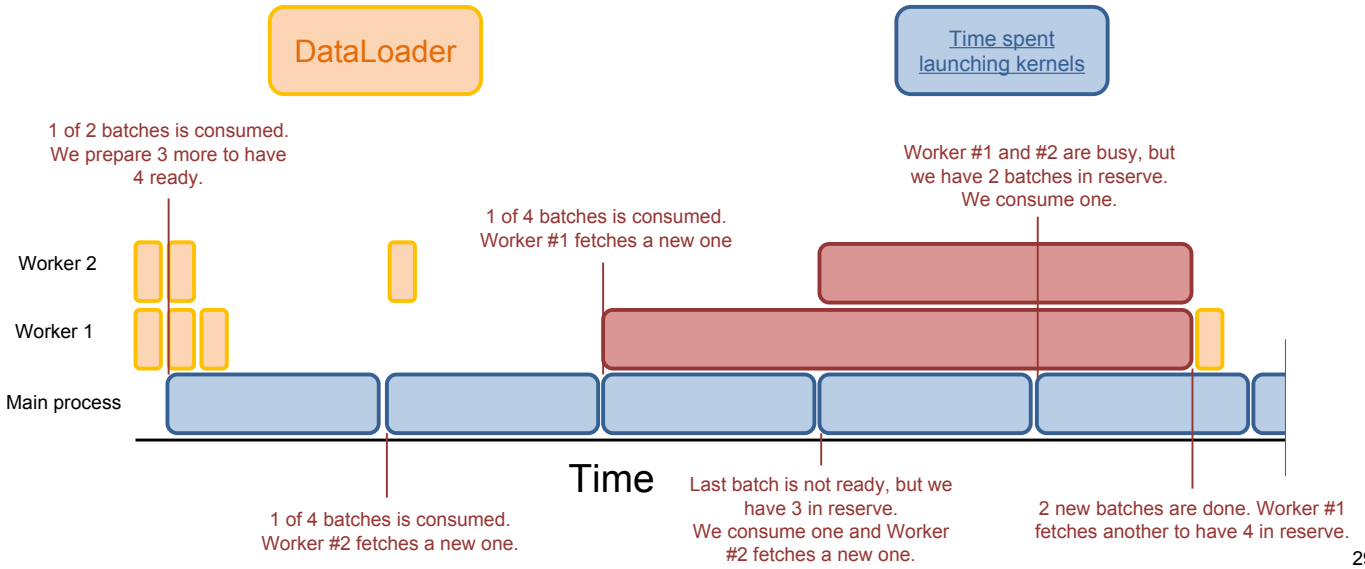


Example with `**prefetch=2**` if the CPU is suddenly slowed down

In this case, having a batch in the pipeline allows us to perform an additional forward pass whilst the CPU is working.

Optimization of the DataLoader

num_workers = 2, prefetch_factor = 2



Example with `**prefetch=2**` and `**num_workers=2**` if the CPU suddenly slows down

By doubling the number of workers, we double the number of batches prepared in advance, which fully compensates for the CPU bottleneck that occurred during training.

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.
- Workers are spawned when the dataloader iterable is called, and killed at the end of the loop : `for images, labels in data_loader`
- **`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).

NB: Workers can be killed whenever needed with **`dataloader._iterator. _shutdown_workers()`**

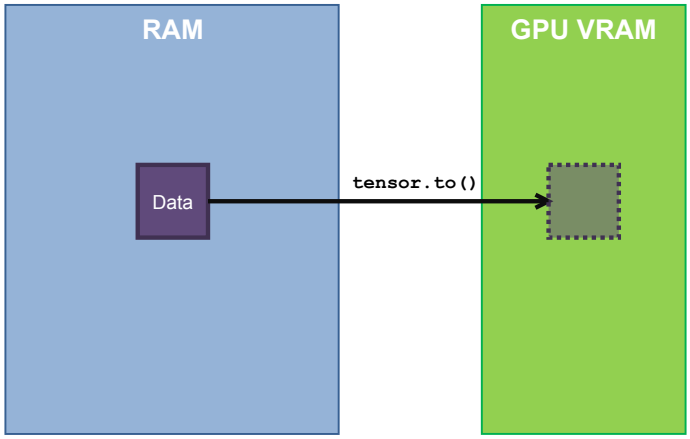
During training, workers are created when the Dataloader is called at the start of the loop, and destroyed at the end of the loop. This means that workers are recreated at every epoch.

This causes additional overhead, as initialising the workers and the parallel environment takes time.

Setting the `persistent_worker` option to `True` allows the same workers to be retained throughout. Of course, this comes at a cost in terms of CPU memory usage between epochs.

Optimization of the DataLoader

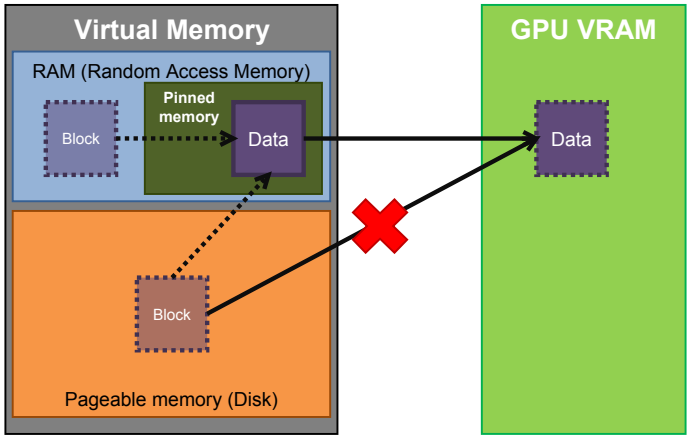
2. Data transfers from CPU to GPU
   - **`pin_memory=True`** allows storing batches directly in pinned memory.



Lorsque l'on veut copier des données préparée par le dataloader de la RAM vers la mémoire GPU, CUDA doit d'abord s'assurer d'avoir l'entièreté du tenseur accessible au moment de la copie.

Optimization of the DataLoader

2. Data transfers from CPU to GPU
   - **`pin_memory=True`** allows storing batches directly in pinned memory.



See https://docs.pytorch.org/tutorials/intermediate/pinmem_nonblock.html

In fact, a process's memory (known as virtual memory) is – to put it simply – divided into two parts:

Actual RAM, and what is known as 'pageable' memory, which may or may not be located on the storage device (hard disk).

A CPU tensor is divided into blocks, each of which may be located either in RAM or in pageablememory.

When attempting to access a block that is not in RAM, a 'page fault' occurs and the data is retrieved from the disk.

To transfer a tensor to GPU memory, CUDA first ensures that it is present in RAM before transferring it.

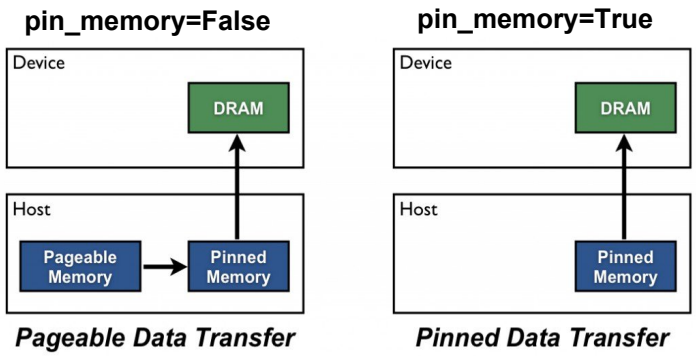
Pinned memory is a region of virtual memory that is guaranteed to be part of RAM and not subject to paging.

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/

Therefore, if a tensor is not already in pinned memory, the process of copying it to the GPU first involves pinning it, and then performing the transfer.

The “pin_memory” option forces the dataloader workers to create the tensors in pinned memory, which speeds up the CPU-to-GPU transfer but complicates the management of CPU RAM.

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

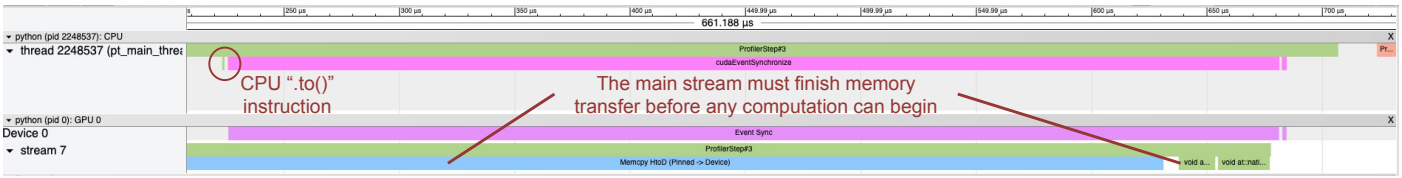


Again, see https://docs.pytorch.org/tutorials/intermediate/pinmem_nonblock.html

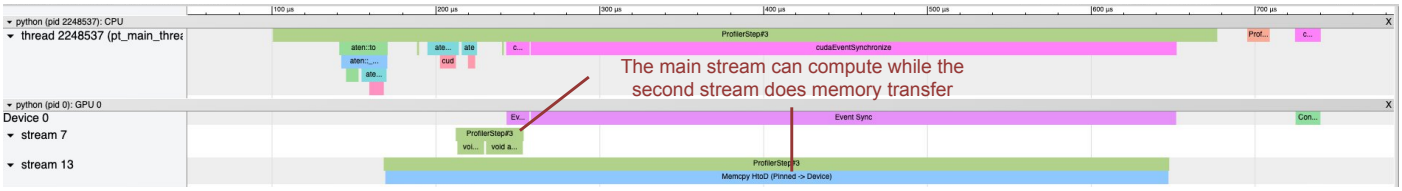
Using pinned memory also enables asynchronous memory transfers from the CPU to the GPU. By utilising the ability of modern GPUs to access pinned memory directly (DMA ==> Direct Memory Access), the CPU simply issues the transfer instruction and the GPU handles the rest, leaving the CPU free to execute other instructions.

Asynchronism side-tangent

Most of the time, to take advantage of the non-blocking option, data transfer must be offloaded to another cuda stream (GPU thread).



"Non-blocking" data transfer with only one cuda stream



"Non-blocking" data transfer offloaded to a second cuda stream

Of course this means the batch you are currently transferring must be different from the batch you are computing this step.

In most cases, to take advantage of asynchronous data transfer, you will need to delegate the data transfer to another CUDA stream. Otherwise, the main stream will also handle the transfer and will be unable to perform calculations in parallel.

Another scenario arises when you have lengthy CPU operations immediately following the transfer instruction. In this case, time is saved by executing them in parallel with the transfer.

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?

The “drop_last” option is used when the number of samples is not a multiple of the batch size. The data loader is then instructed to simply ignore the last incomplete batch. This avoids wasting loading and computation time on an incomplete batch (which therefore contains less information and is less stable).

Optimization of the DataLoader - Takeaway

`num_workers` (`int`; ≥0) : Spawn parallel CPU workers to offload and parallelize the dataloading. Should generally be equal to the number of requested CPU cores.

`prefetch_factor` (`int`; ≥0) : The number of batches each worker should prepare in advance. Helps avoiding wait periods between model passes and circumvent eventual CPU slowdowns.

`persistent_workers` (`bool`) : If **`True`**, worker processes will not be shutdown after a dataset has been consumed once. This maintains the workers' Dataset instances alive between epochs.

`pin_memory` (`bool`) : If **`True`**, workers will create batch tensors in a CUDA pinned memory, speeding up the transfer to GPU memory.

`drop_last` (`bool`) : If **`True`**, the last samples of the dataset will be dropped if the size of the dataset is not a multiple of the number of batches.

Summary of the DataLoader parameters:

num_workers (int; ≥0) : Spawn parallel CPU workers to offload and parallelize the dataloading. Should generally be equal to the number of requested CPU cores.

prefetch_factor (int; ≥0) : The number of batches each worker should prepare in advance. Helps avoiding wait periods between model passes and circumvent eventual CPU slowdowns.

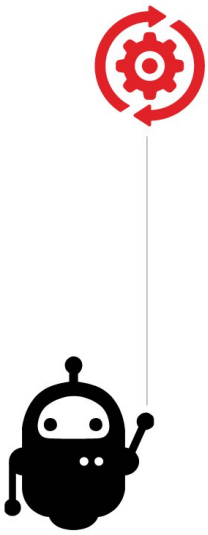
persistent_workers (bool) : If **True**, worker processes will not be shutdown after a dataset has been consumed once. This maintains the workers' Dataset instances alive between epochs.

pin_memory (bool) : If **True**, workers will create batch tensors in a CUDA pinned memory, speeding up the transfer to GPU memory.

drop_last (bool) : If **True**, the last samples of the dataset will be dropped if the size of the dataset is not a multiple of the number of batches.

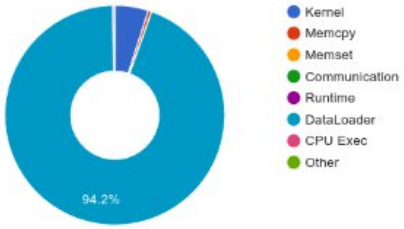
Remember that all these parameters (except **drop_last**) come at a cost in CPU memory usage.

TP2_3: Optimization of the DataLoader



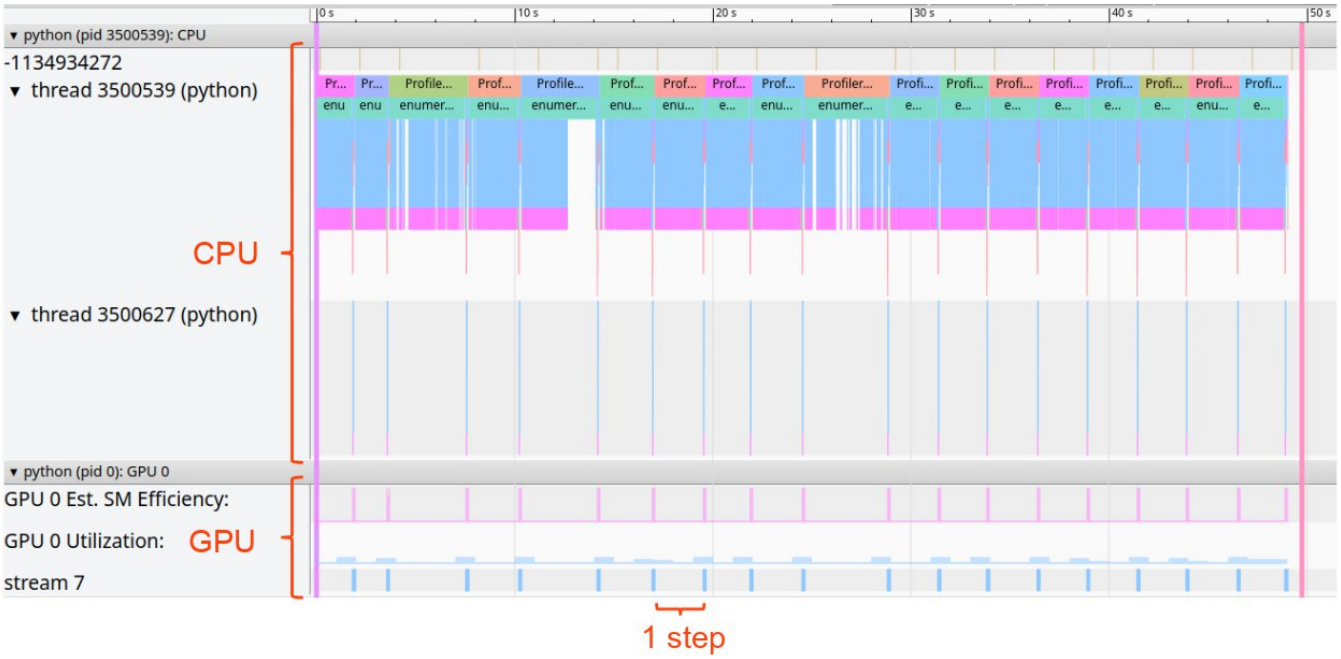
- Play with the values of Dataloader parameters
- Note their impact on training time and memory usage
- Find the most optimal set of parameters

TP2_1: Profiler PyTorch (conclusion)

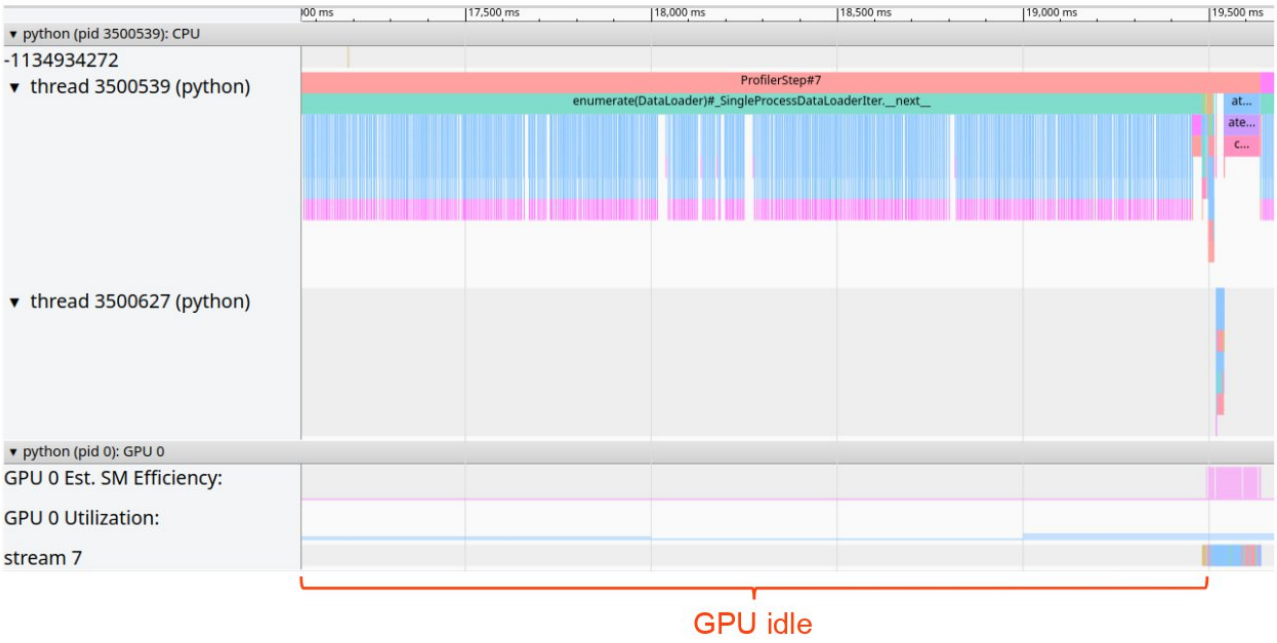


After seeing the traces, it is obvious that the optimization efforts need to concentrate on the DataLoader.

TP2_2: Profiler Trace



TP2_2: Profiler Trace (1 step)

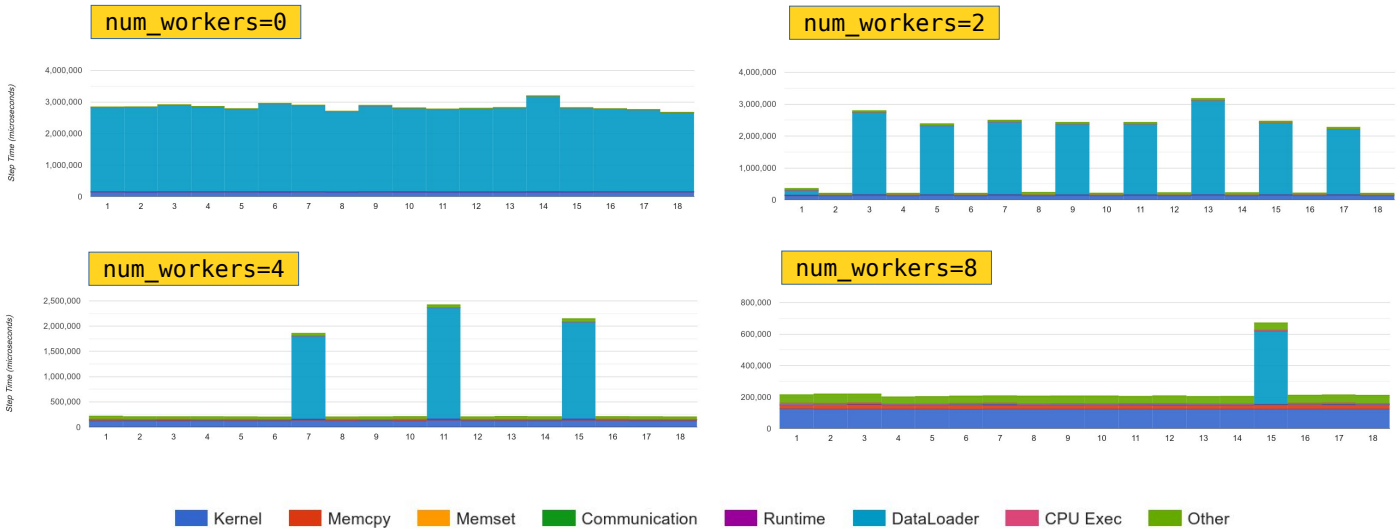


TP2_2: Profiler Trace (1 step - CPU)



TP2_1: Optimization of the DataLoader

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



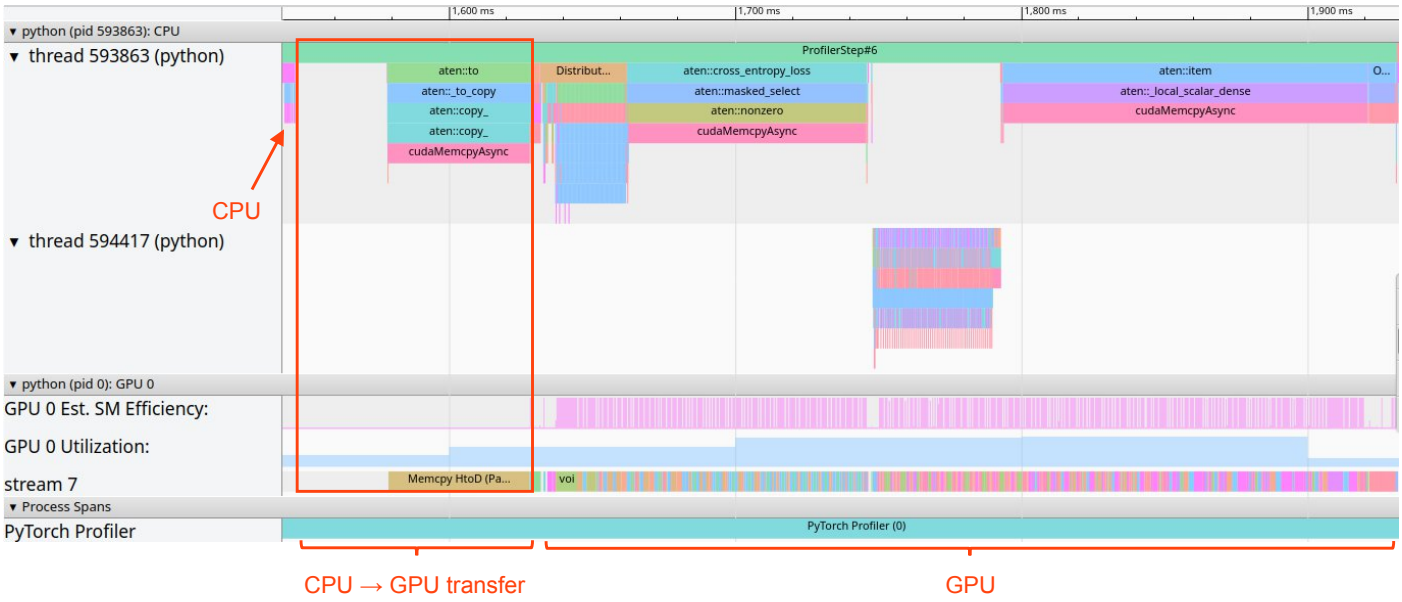
TP2_1: Optimization of the DataLoader

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_1: Optimization of the DataLoader

```
pin_memory=False, non_blocking=False
```



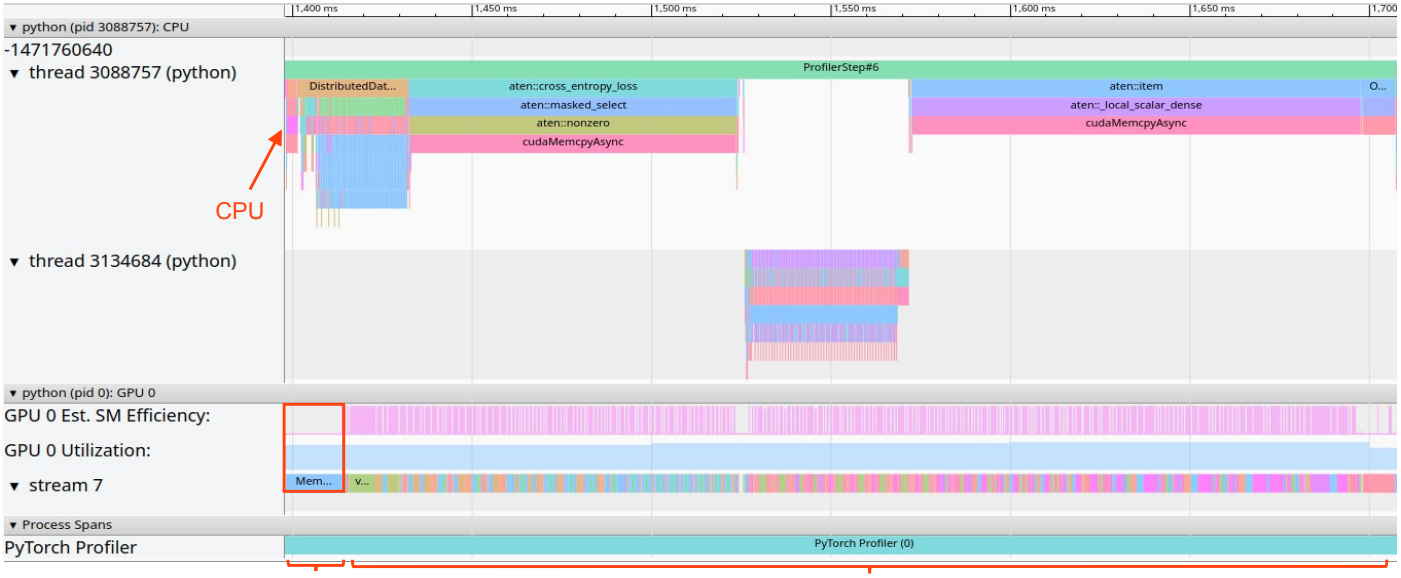
TP2_1: Optimization of the DataLoader

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



TP2_1: Optimization of the DataLoader

`pin_memory=True, non_blocking=True`



TP2_1: Optimization of the DataLoader

- Chosen optimizations:

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

Configuration

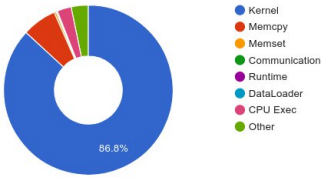
Number of Worker(s)	1
Device Type	GPU

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 %

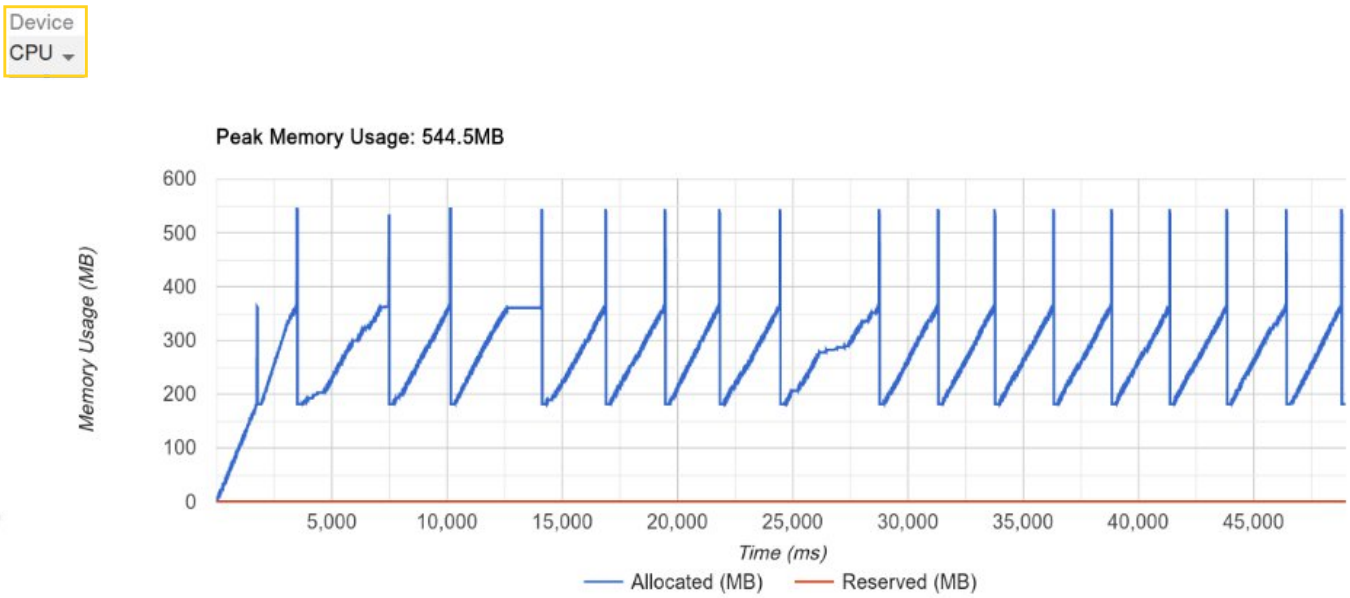
Execution Summary

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: Profiler Memory View (CPU)

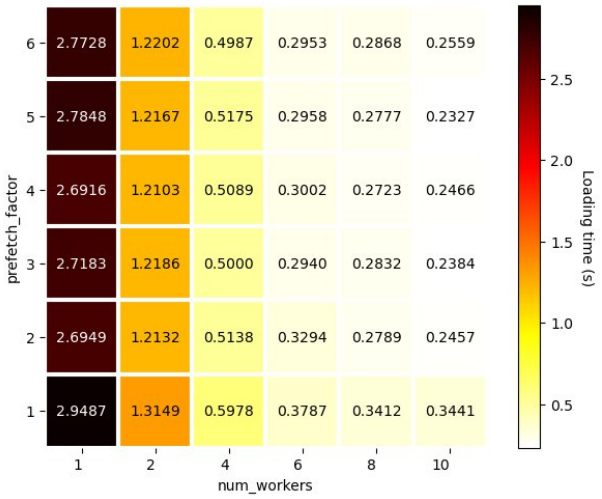


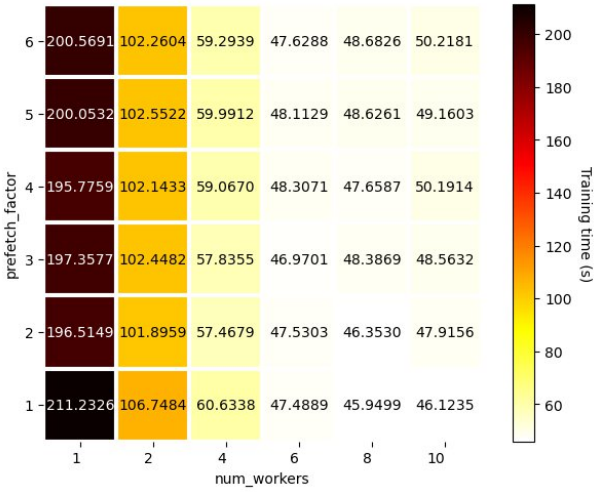
Appendix: Optimization of the DataLoader

- Impact of the **prefetch factor**

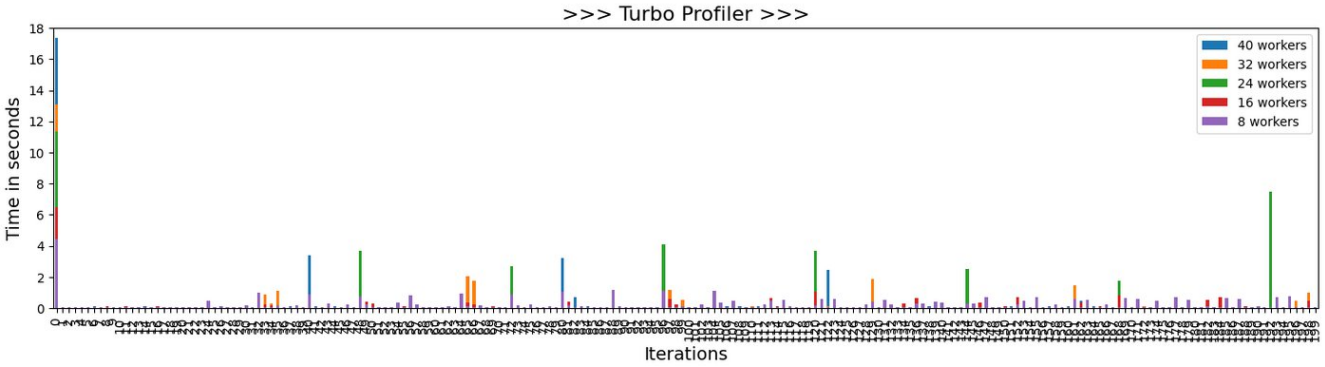
`dlojz.py` - 50 iterations - test partition `gpu_p4`

NB: These results don't correspond to our usage case but still illustrate the influence of the parameters.





Appendix: Optimization of the DataLoader (resnet 50)



	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