

# Data Loading

# Pourquoi ?

- ◆ Le data loading c'est la première étape de l'entraînement
- ◆ Data loading inefficace → bottleneck les GPUs
- ◆ Un Mauvais data loading sera amplifié par le nombre de GPU
- ◆ Dataset passe pas en RAM → adapter sa stratégie de chargement des données pour utiliser efficacement les ressources (pas forcément trivial)


# torch.utils.data.Dataset - Map Style dataset



```
class CustomDataset(torch.utils.data.Dataset):  
    def __init__(self, data):  
        self.data = data  
  
    def __len__(self):  
        return len(self.data)  
  
    def __getitem__(self, idx):  
        return self.data[idx]
```

- ◆ Dataset classique de Torch
- ◆ On accède aux données par un index.
- ◆ A favoriser si notre dataset est en RAM

# torch.utils.data.Dataloader



```
torch.utils.data.DataLoader(  
    dataset,  
    batch_size=1,  
    shuffle=False,  
    sampler=None,  
    batch_sampler=None,  
    num_workers=0,  
    collate_fn=None,  
    pin_memory=False,  
    drop_last=False,  
    timeout=0,  
    worker_init_fn=None,  
    *,  
    prefetch_factor=2,  
    persistent_workers=False  
)
```

- ◆ Wrap le dataset, récupère les samples et crée les batches.
- ◆ Possède de nombreux arguments, pas toujours facile à comprendre

# torch.utils.data.Dataloader - Batch Size



Batch size → nombre d'éléments dans un batch, influe sur la VRAM.

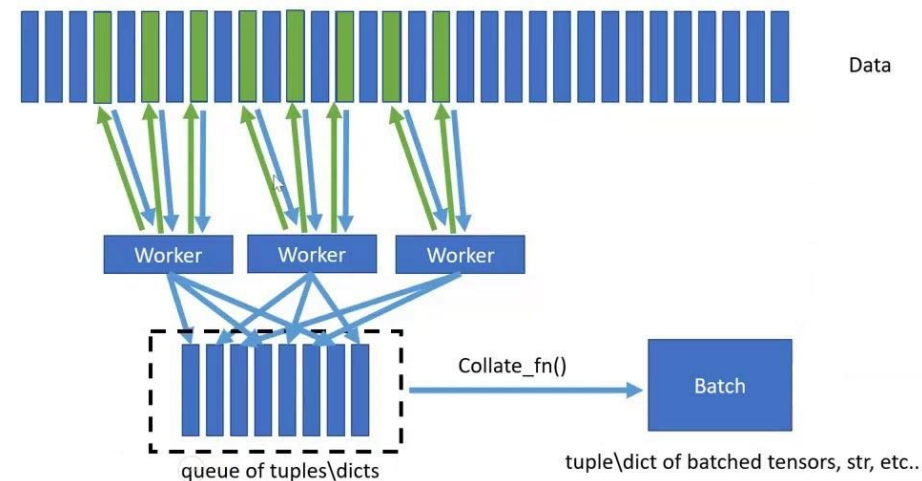
GPU  $\ni$  *warp* (groupe de threads)

Chaque thread dans un warp est exécuté en même temps.

Un warp contient 32 threads depuis les dernière générations → favoriser les puissances de 32 puis 16 puis 8 etc...

Batch size multiple de 32 = 😊

# torch.utils.data.Dataloader - Workers



Worker = processus qui charge les données

0 Workers → processus principal charge les données

N Workers → N processus chargent les données en parallèle (processus principal  $\nsubseteq$  N processus)

# torch.utils.data.Dataloader - Workers

⚠ Plusieurs subprocess = overhead → +workers ≠ +performances

⚠ Mémoire partagé entre subprocess → accès concurrentiels = possible

baisse de performance en lecture

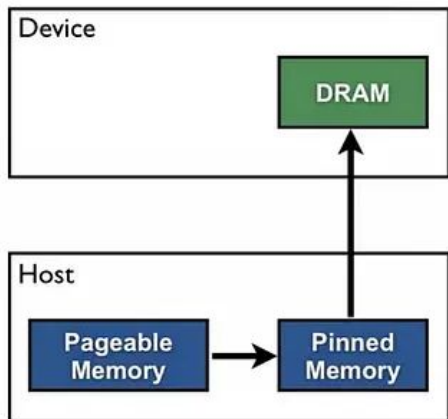
⚠ +workers = +RAM & +communications (IPC)

⚠ Trop de worker = Noisy Neighbor, 100% cpu = mauvais → baisse de perf de l'entraînement

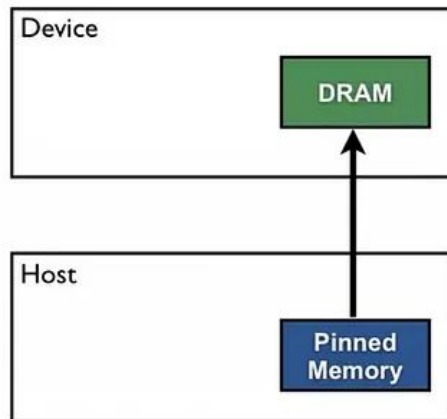
Entre 2 et 8 workers / GPU, faire un benchmark avant un entraînement.

# torch.utils.data.Dataloader - Memory Pinning

*Pageable Data Transfer*



*Pinned Data Transfer*



Pinned memory = Paged lock memory

Paged lock → seulement en RAM

Pageable → RAM et disque



# torch.utils.data.Dataloader - Memory Pinning


Transfert CPU → GPU = obligatoire que la donnée soit paged locked

`pin\_memory` utilise un thread pour faire le transfert pageable → pinned (donc non-bloquant)

A combiner avec `.to(device, non\_blocking=True)`, utilise un CUDA stream dédié pour le transfert CPU → GPU

💡 Toujours utiliser `non\_blocking=True` lors de vos transferts

# torch.utils.data.Dataloader - Prefetch



```
torch.utils.data.DataLoader(  
    dataset,  
    batch_size=1,  
    shuffle=False,  
    sampler=None,  
    batch_sampler=None,  
    num_workers=0,  
    collate_fn=None,  
    pin_memory=False,  
    drop_last=False,  
    timeout=0,  
    worker_init_fn=None,  
    *,  
    prefetch_factor=2,  
    persistent_workers=False  
)
```

Prefetch factor = batch par worker sauvegardé dans un buffer

Première itération,  $\text{workers} * \text{prefetch\_factor}$  batchs sauvegardés dans le buffer

Itération est finie → 1 worker sera assigné pour ajouter un nouveau batch au buffer

Permet d'avoir des batchs prêt plutôt que de faire le traitement à la volée

# torch.utils.data.Dataloader - A retenir

- ✓ Batch size multiple de 32 (multiple de 16 également possible)
- ✓ Workers entre 2 et 8 par GPUs, à benchmark
- ✓ Memory Pinning avec `non\_blocking=True`
- ✓ Prefetch factor par défaut suffisant

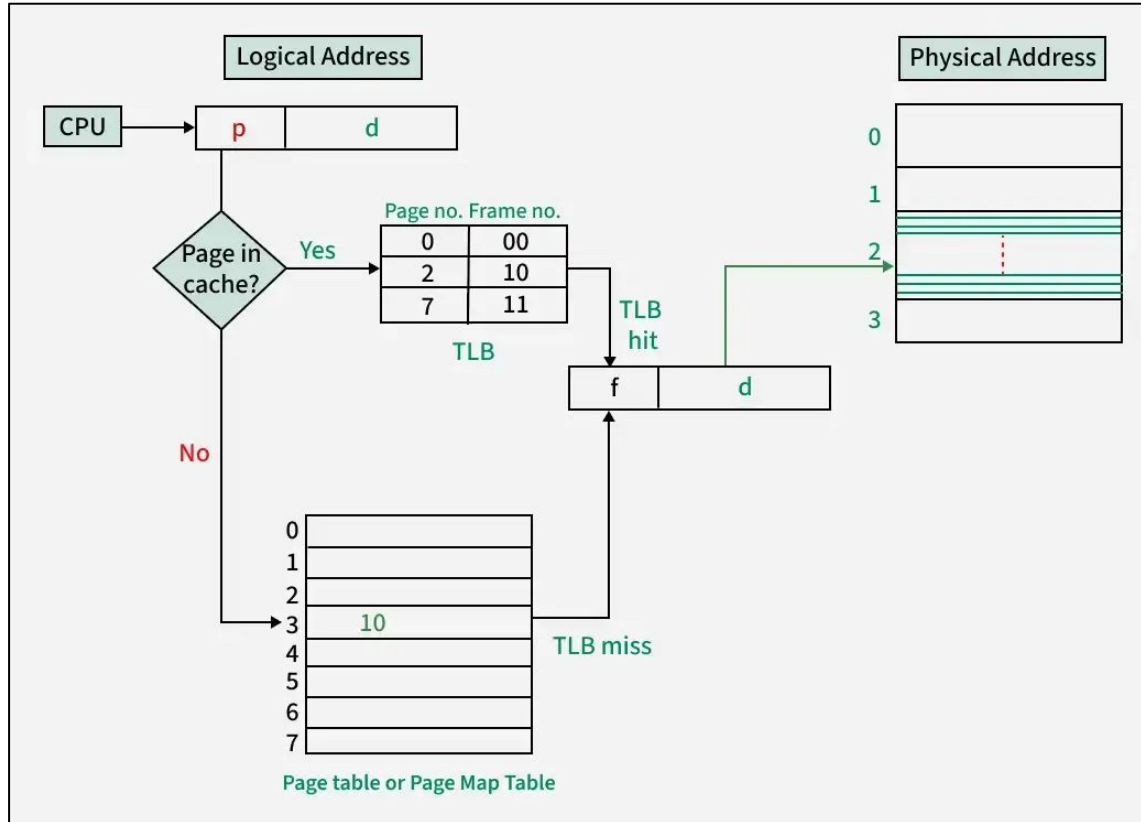
# Memory mapping

Utile pour les gros datasets qui ne passent pas en RAM. Permet de lier un fichier ou une partie de celui-ci à un espace de la mémoire virtuelle. Quelques termes utiles:

- **Paging (pagination)** : sépare la mémoire en sections → **pages** (pour la mémoire virtuelle) et **frames** (pour la mémoire physique). Les pages ont généralement une taille de 4KB à 8KB.
- **Page Table** : Lien entre les adresses virtuelles et les adresses physiques. Permet à un programme d'accéder à un espace d'adresses virtuelles continu, même si la mémoire physique n'est pas continue.
- **Translation Lookaside Buffer (TLB)** : accélère les traductions d'adresses de la mémoire virtuelle vers la mémoire physique.
- **Page fault** : erreur qui se produit lorsqu'un processus essaie d'accéder à une page qui n'est pas actuellement chargée dans la mémoire physique (RAM).

Avantages de la mémoire virtuelle → espace mémoire bien plus grand que la RAM physique. Le système d'exploitation ne charge en RAM que les pages dont un processus a besoin.

# Memory mapping



- ◆ Si la page existe en RAM, les autres accès à cette page seront très rapide
- ◆ Les processus se partagent les pages en RAM → super pour de la lecture parallèle 😊

# Memory mapping

- ◆ Important de lire ces fichiers de façon séquentielle pour avoir les meilleures performances → overhead dû au chargement d'une page
- ◆ Si plusieurs workers → charger la memmap dans chaque processus plutôt que l'instancier dans le processus principal

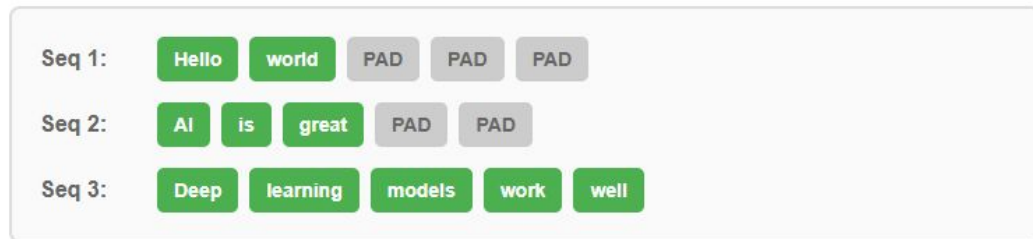
# torch.utils.data.IterDataset

- ◆ Iterable Dataset retourne un itérateur pour lire les données
- ◆ Faible utilisation mémoire → traite les données par petits morceaux, ne charge pas l'intégralité du jeu de données en RAM
- ◆ Cependant plus complexe d'utiliser du shuffling/sampling
- ◆ Adapté pour des applications avec de gros volumes de données séquentielles → texte/capteurs par exemple

# Sequence Packing

- ◆ Memory Efficient
- ◆ Compute Efficient
- ◆ Entrainement plus rapide
- ◆ Meilleure utilisation des GPUs
- ◆ Particulièrement bénéfique pour les LLMs

Before: Traditional Batching (with padding)



After: Sequence Packing (no padding)





# Pré-processing

- ◆ Pré-traiter les données en amont (Ex: pré-tokenization)
- ◆ Sauvegarder le résultat dans un format efficace (binaire, mmap, parquet/pyarrow)
- ◆ Pré-traitement trop volumineux/coûteux → déporter le pré-traitement sur un autre serveur

# A retenir



Dataset **pas**se en RAM → Map style dataset



Dataset ne **pas**se **pas** en RAM → Iterable dataset + memory mapping

Pré-traitement → faire à l'avance (si possible)

Eviter le padding pour accélérer l'entraînement (sequence packing)