Deep Learning Optimisé - Jean Zay

Introduction – Jean Zay – GPU

Institut du Développement et des Ressources en Informatique Scientifique

Présentation de DLO-JZ

Plan ◂

Imagenet / Resnet-50 ◂

Présentation des participants ◂

Présentation - Sujets traités

Jour 1

- Jean Zay
- Revue de code
- Les enjeux de la montée à l’échelle
- **GPU computing**
- **Tensor Cores**

Jour 2

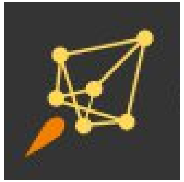
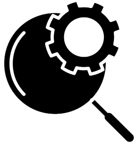
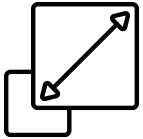
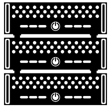
- **Distribution - Data Parallelism**
- Profiler PyTorch
- **Optimisation du DataLoader**

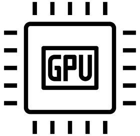
Jour 3

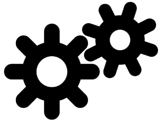
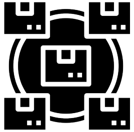
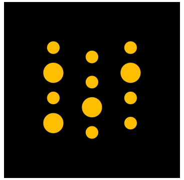
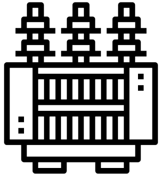
- **Stockage et format de données**
- Outil de visualisation
- **Entraînement et *large batches***
- *HyperParameter Optimization*

Jour 4

- Bonnes pratiques
- JIT
- **Les parallélismes de modèle**
- Les API pour les parallélismes de modèle

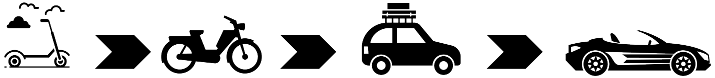





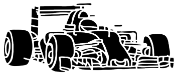


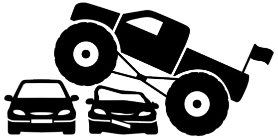
Déroulé des TP

- Les TP des jours 1, 2 et 3 :
  - Optimisations système : GPU, Mixed Precision, Data Parallelism
  - Profiler
  - DataLoader
  - *Optimizer*s et LR scheduler
  - *Hyper-Parameters Optimization (HPO)*

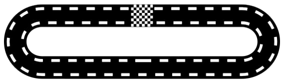




- Les TP du Jour 4 (à la carte) :
  - *Model parallelism* avec un gros modèle HuggingFace
  - Implémentation de *ddp*
  - Implémentation de *tensor parallelism* et *2D parallelism*
  - Data Augmentation
  - torch.compile & torchscript



- Mini Jean Zay réservé : 24 GPU A100 sur 3 noeuds



Données - Imagenet

But:

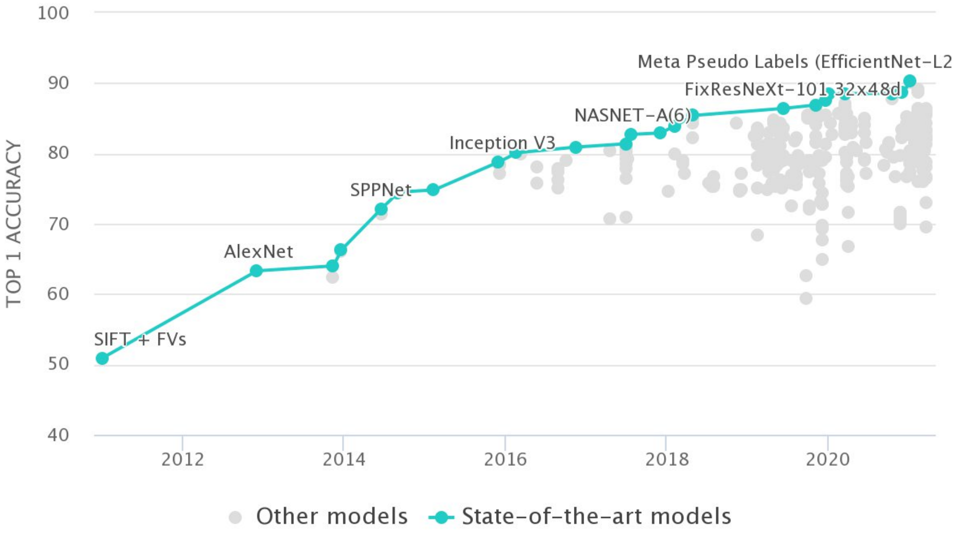
Classification (1000 classes)

Dataset:

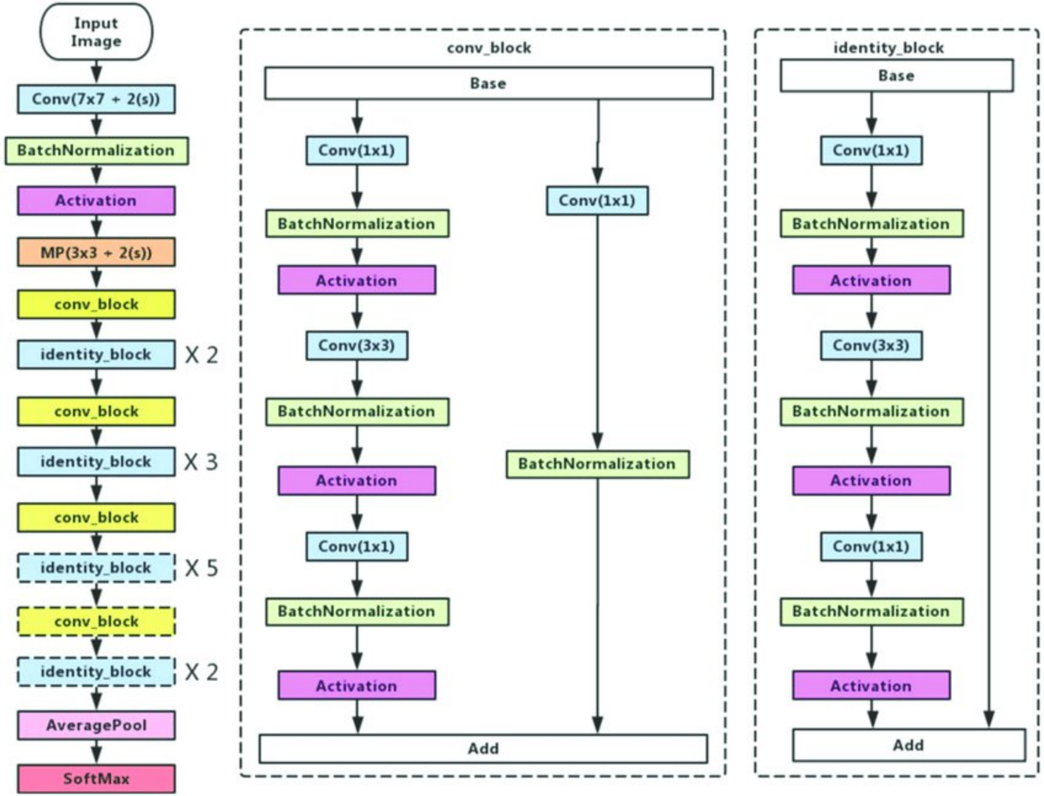
Train dataset: **1,2 Millions** d'images labellisées

Validation dataset: **50 000** images labellisées

http://www.image-net.org/



Imagenet - Resnet-50



Resnet :

- Residual Learning
- BatchNorm layer
  - Remplace les *dropouts*
- Average Pooling
  - Rend le modèle indépendant de la taille des images d’entrée

Imagenet - Resnet-50

How long does it take to train Resnet-50 on ImageNet?

Before 2017

14 days

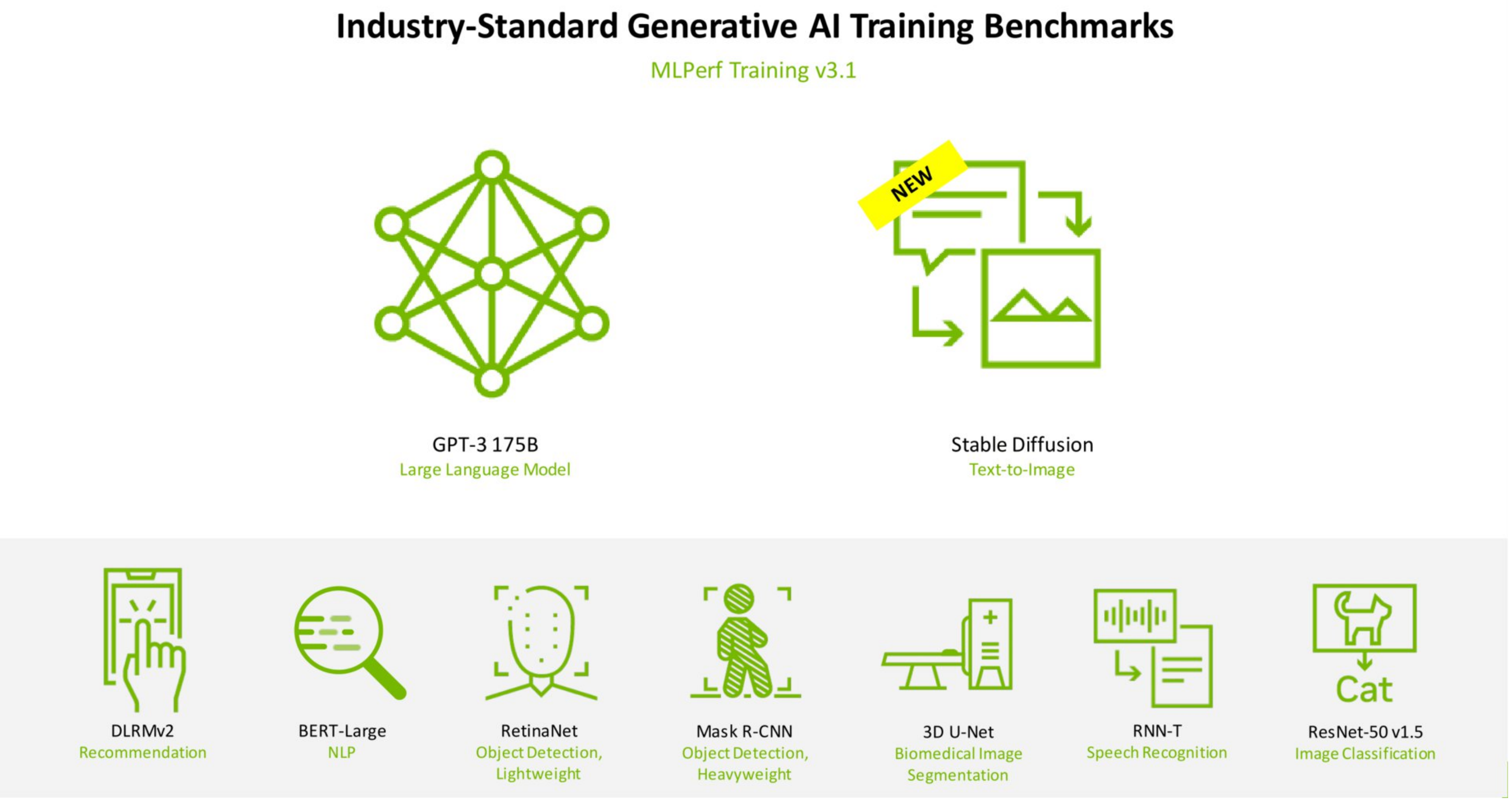
NVIDIA M40 GPU

Imagenet - Resnet-50

Training Resnet-50 on Imagenet

Facebook Caffe2	UC Berkeley, TACC, UC Davis Tensorflow	Preferred Network ChainerMN	Tencent TensorFlow	Sony Neural Network Library (NNL)	Fujitsu MXNet
1 hour	31 mins	15 mins	6.6 mins	2.0 mins	1.2 mins
Tesla P100 x 256	1,600 CPUs	Tesla P100 x 1,024	Tesla P40 x 2,048	Tesla V100 x 3,456	Tesla V100 x 2,048
Apr	Sept	Nov	July	Nov	Apr
2017				2018	2019

MLPerfv3.1 - Benchmark 2023



BLOOM on Jean Zay

Pre-training :
117 days (2022)
384 x 80GB A100 GPUs

DLO-JZ & SpeLLM

Being able to specialize an LLM to meet your specific needs.

Learn to make a prototype in 3 days → The training is hands-on centered

1st day

Transformers theory

Classic Fine-Tuning

Use case presentation

System improvement environment

Metrics and Evaluations (1st part)

2nd day

Data Cleaning

Prompt Engineering

Retrieval Augmentation Generation

Parameter Efficient Fine-Tuning

Hyper Parameter Optimization

3rd day

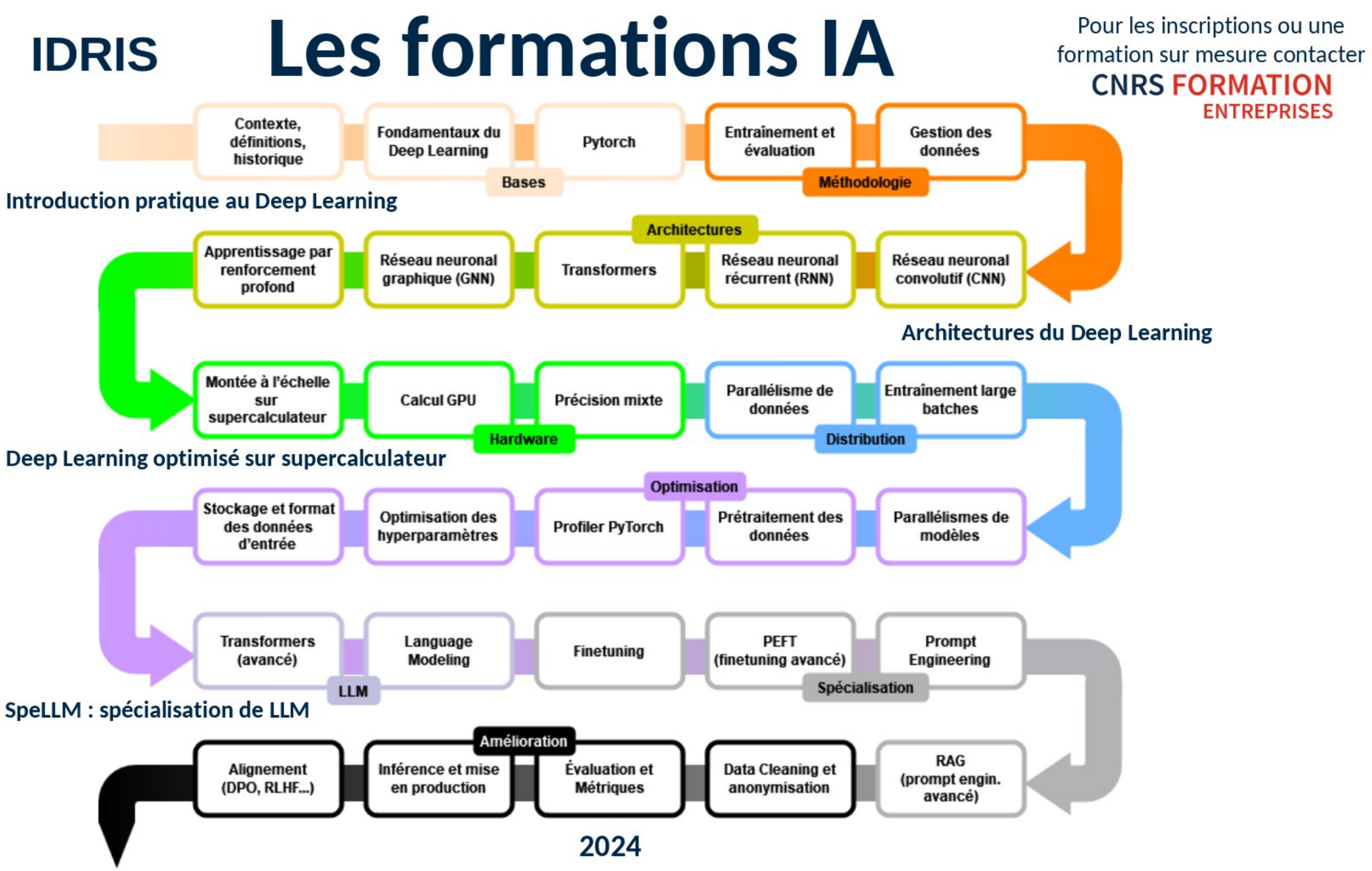
Metrics and Evaluations (2nd part)

Alignment

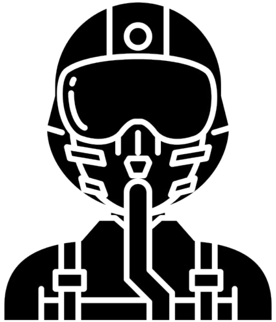
Inference

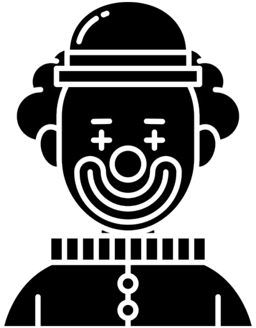
Discussions

Train modulaire de formations IDRIS



Présentation des participant·e·s





Jean Zay

Supercalculateur ◀

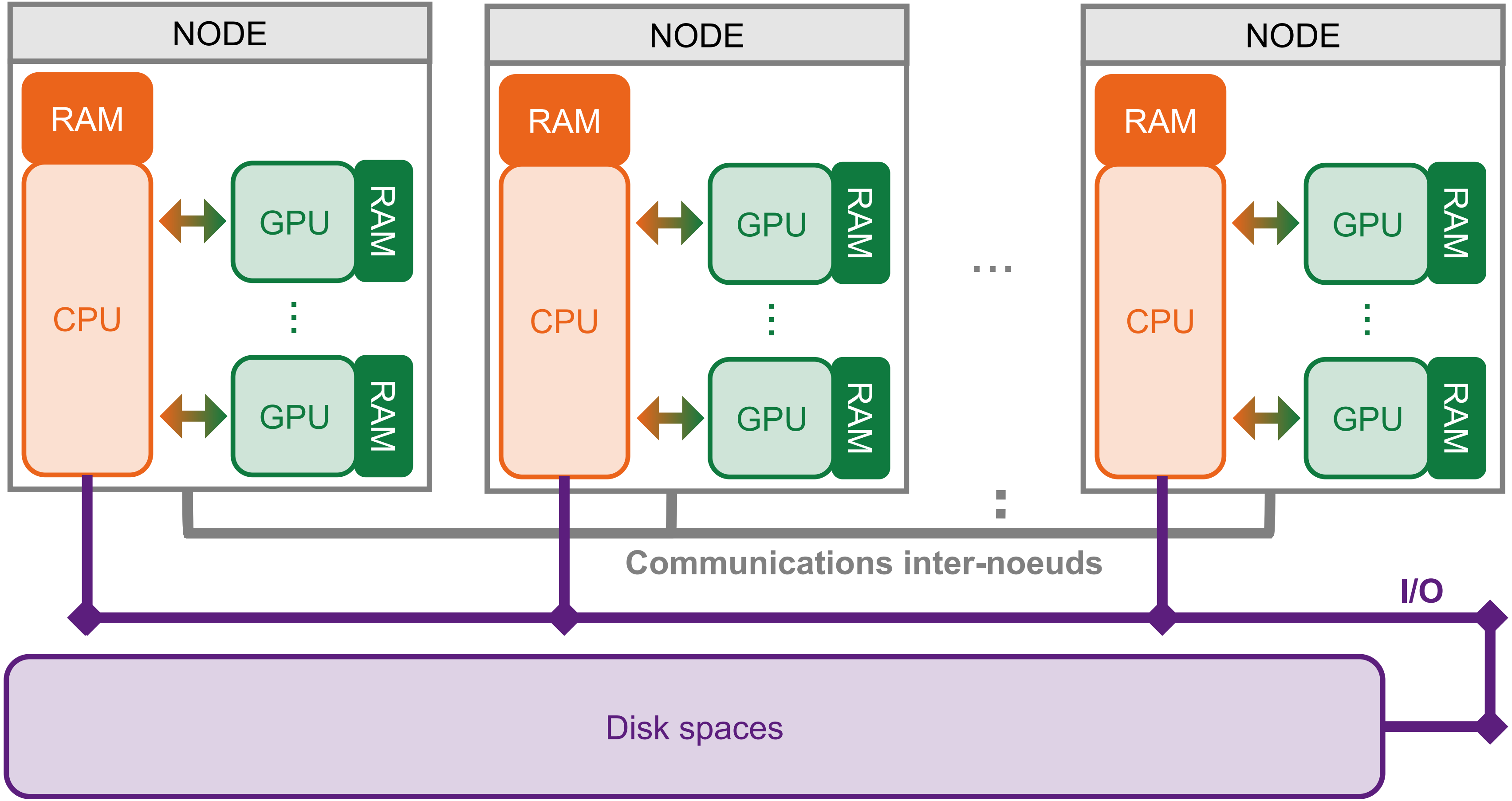
Jean Zay ◀

Soumission de jobs ◀

JupyterHub sur Jean Zay ◀

Outils Slurm pour notebook python ◀

C’est quoi un supercalculateur ?

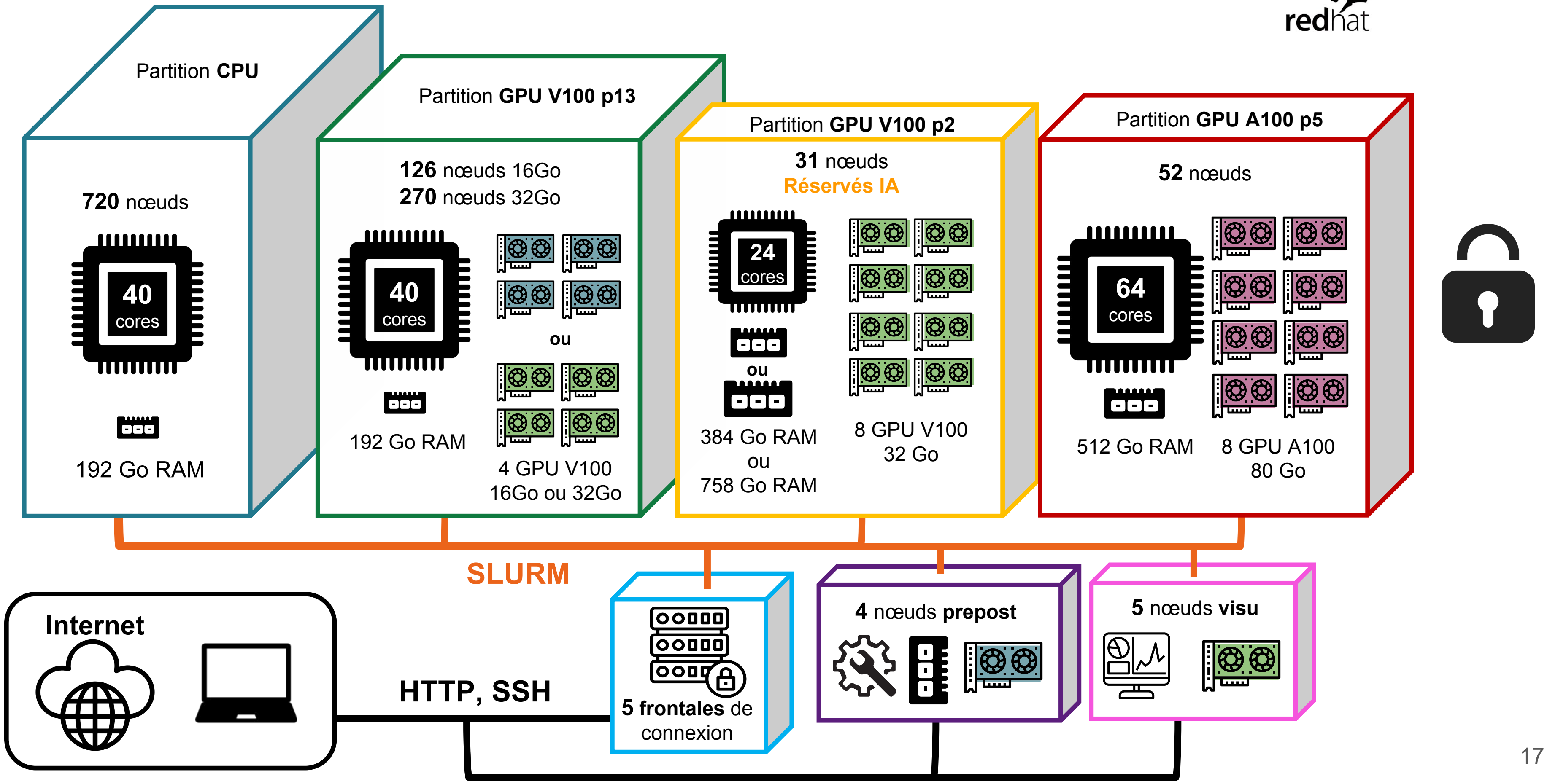


Jean Zay

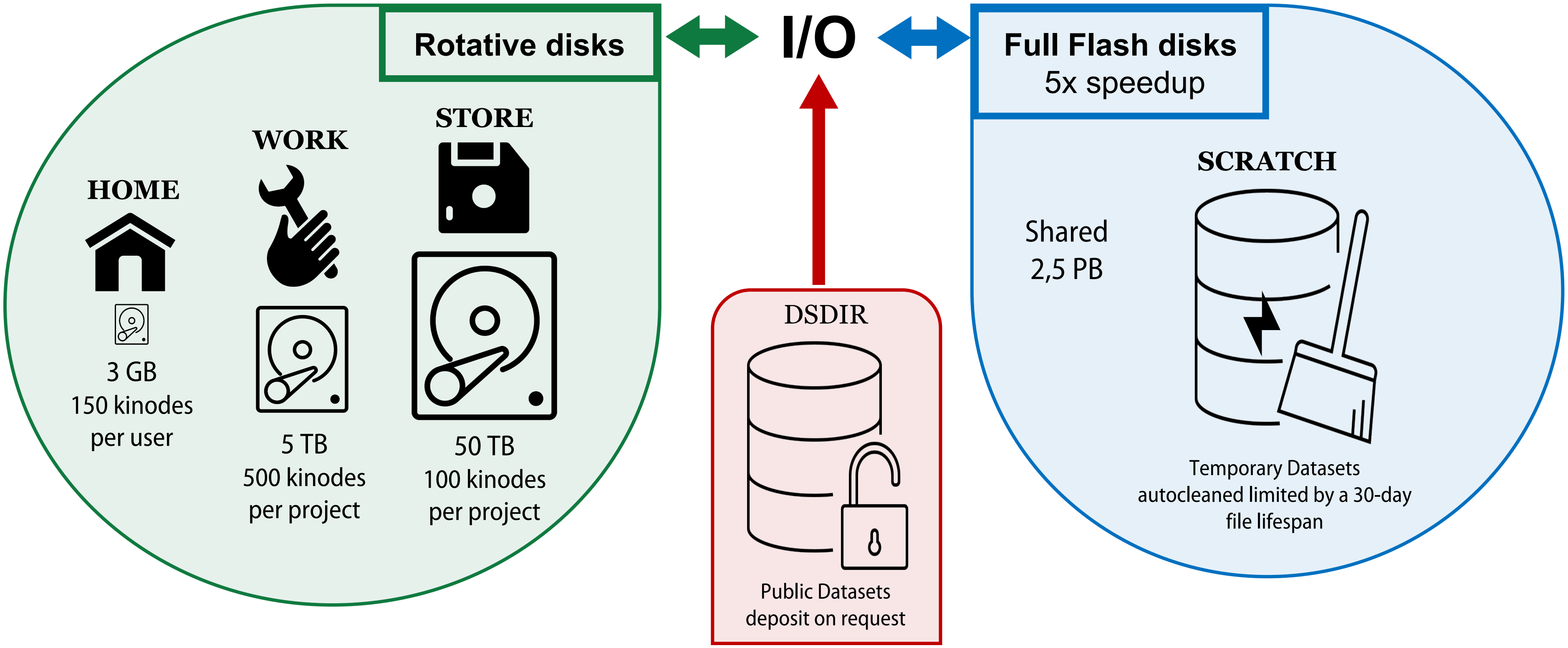
Tier1 - Premier supercalculateur convergé français pour l'Intelligence Artificielle (IA) et le Calcul Haute Performance (HPC)

- 50 M€
- 24h/24
- 320 m²
- 70 tonnes
- 2400 kW
- Refroidissement eau tiède 40-32°C
- 4000 MWh/an
- 36,8 Pflop/s
- 89832 CPU
- 3152 GPU
- RAM 454 To
- Full Flash 2,2 Po
- Rotative Disk 35 Po

Jean Zay : Ressources disponibles



Jean Zay : Espaces de stockage



Jean Zay : Environnement de travail

ENVIRONMENT MODULES

Catalogue de modules mutualisés (environnements conda)

- Installés par l'IDRIS
- Enrichis sur demande

```
login@jean-zay3:~$ module load pytorch-gpu/py3/1.11.0
  Loading requirement: ...
(pytorch-gpu-1.11.0+py3.9.12) login@jean-zay3:~$
```

- Personnalisables

```
~$ pip install --user --no-cache-dir <paquet>
```

Conflits entre les versions
Saturation de vos espaces disques

Environnements conda personnels

```
login@jean-zay3:~$ module load anaconda-py3/2023.03
(base) login@jean-zay3:~$ conda create -n myenv
```

Saturation de vos espaces disques ++

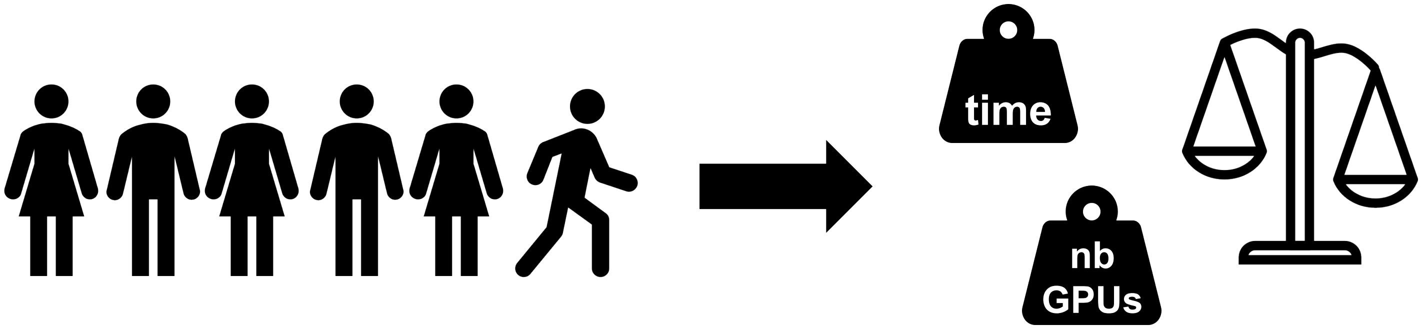
Conteneurs Singularity

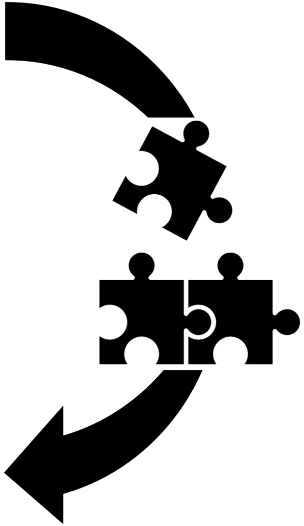
```
login@jean-zay3:~$ module load singularity
```

Images SIF à importer sur Jean Zay

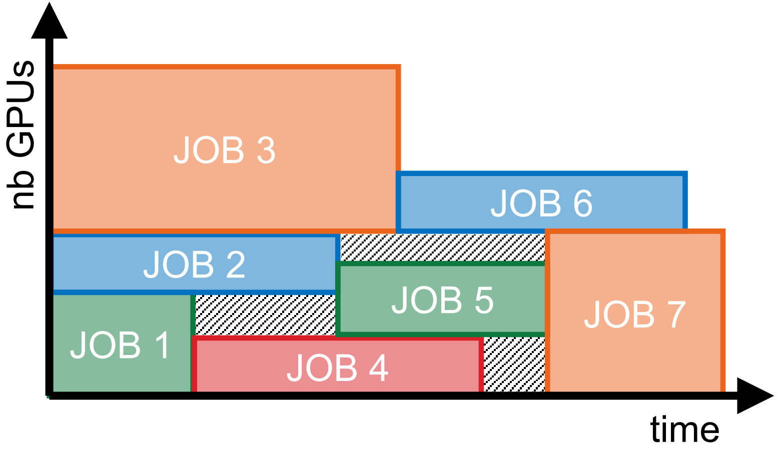
- Depuis votre PC personnel
- À partir de dépôts publics
- Possibilité de convertir une image docker

Soumission de jobs - Slurm





slurm
workload manager



Soumission de jobs - Slurm

script.slurm

```
#!/bin/bash

#SBATCH --job-name="dlojz"          # number of job
#SBATCH --output="dlojz%j.out"      # out file
#SBATCH --error="dlojz%j.err"       # error file
#SBATCH --nodes=2                   # nb of node
#SBATCH --gres=gpu:4                # nb of GPU per node
#SBATCH --ntasks-per-node=4         # nb of tasks per node
#SBATCH --cpus-per-task=10          # nb of cores
#SBATCH --hint=nomultithread        # no hyper threading
#SBATCH --time=03:00:00             # max execution time

module load pytorch-gpu/py3/2.1.1   # environment

srun python script.py               # run script
```

Soumission de jobs - Slurm

script.slurm

```
#!/bin/bash

#SBATCH --job-name="dlojz"
#SBATCH --output="dlojz%j.out"
#SBATCH --error="dlojz%j.err"
#SBATCH --nodes=2
#SBATCH --gres=gpu:4
#SBATCH --ntasks-per-node=4
#SBATCH --cpus-per-task=10
#SBATCH --hint=nomultithread
#SBATCH --time=03:00:00

module load pytorch-gpu/py3/2.1.1

srun python script.py
```

```
login@jean-zay3:~$ sbatch script.slurm
```

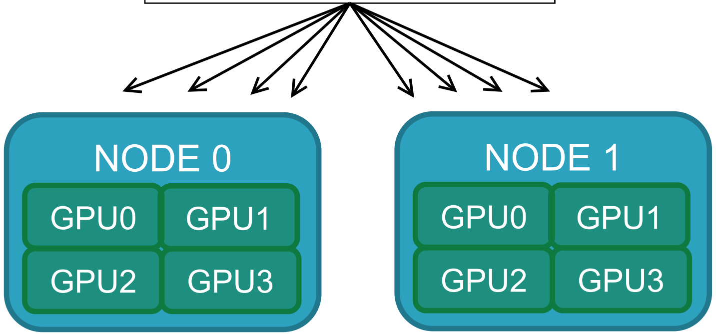
Soumission du job

Passage dans la file d’attente

```
login@jean-zay3:~$ squeue --me
  JOBID PARTITION     NAME     USER ST       TIME  NODES NODELIST(REASON)
 223225   gpu_p13    dlojz           PD       0:00      2 (Priority)
```

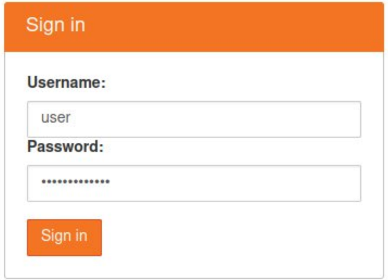
Lancement du job

srun python script.py



JupyterHub sur Jean Zay

1. Authentification sur https://jupyterhub.idris.fr

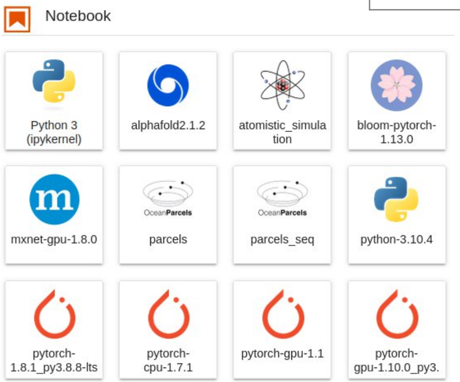


2. Choisir et configurer une instance

Lancer sur une frontale

Lancer sur un nœud de calcul

3. Choisir un kernel (pytorch-gpu-1.11.0)

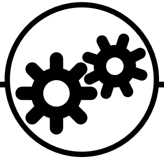


Jean Zay : Outils Slurm pour notebook python

```python
from idr_pytools import gpu_jobs_submitter
```

```python
command = ‘dlojz.py --batch-size 128 --image_size 176’
n_gpu = 8
MODULE = ‘pytorch-gpu/py3/2.1.1’
name = ‘dlojz’

jobid = gpu_jobs_submitter(command, n_gpu, MODULE, name=name, account='xyz@a100', time_max='05:00:00')
```

command
n_gpu
MODULE
name
account
time_max



script.slurm

```bash
#!/bin/bash

#SBATCH --job-name="dlojz"
#SBATCH --output="dlojz%j.out"
#SBATCH --error="dlojz%j.err"
#SBATCH --nodes=2
#SBATCH --gres=gpu:8
#SBATCH -C a100
#SBATCH --ntasks-per-node=8
#SBATCH --cpus-per-task=8
#SBATCH --hint=nomultithread
#SBATCH --time=05:00:00
#SBATCH --account=xyz@a100

module load pytorch-gpu/py3/2.1.1

srun python dlojz.py —batch-size 128 —image_size 176
```

```
$ sbatch script.slurm
```

jobid

Jean Zay : Outils Slurm pour notebook python

```
from idr_pytools import display_slurm_queue
```

```
name = 'dlojz'
display_slurm_queue(name)
```

→

```
$ squeue --me -n <name>
```

```
from idr_pytools import search_log
```

```
jobid = ['12345']

search_log(contains=jobid)[0]

search_log(contains=jobid, with_err=True)[0]
```

`search_log(contains=jobid)[0]` → nom du fichier *output*

`search_log(contains=jobid, with_err=True)[0]` → nom du fichier *error*

Revue du code

Revue générale ◂

Revue détaillée ◂

Code dlojz.py Review

Import

argparse : arguments

Instanciation

Model, distribution, optimizer, ...

Dataloader

Preprocessing, Optimisation, ...

Instanciation

Training

Mixed precision, distribution, ...

Validation

Mixed precision, distribution, ...

Checkpoint & report

Runner

dlojz.py – Import & run

```python
import os
import contextlib
import argparse
import torchvision
import torchvision.transforms as transforms
import torchvision.models as models
from torch.utils.checkpoint import checkpoint_sequential
import torch
import numpy as np
import apex

import idr_torch
from dlojz_chrono import Chronometer
from dlojz_torch import distributed_accuracy

import random
random.seed(123)
np.random.seed(123)
torch.manual_seed(123)
```

reproductibility

Import libraries

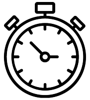
os
contextlib

argparse

PyTorch NumPy torchvision NVIDIA - APEX

Chronometer (DLO-JZ)

time log & home profiler



idr_torch (JZ users)

distribution utils for Jean Zay

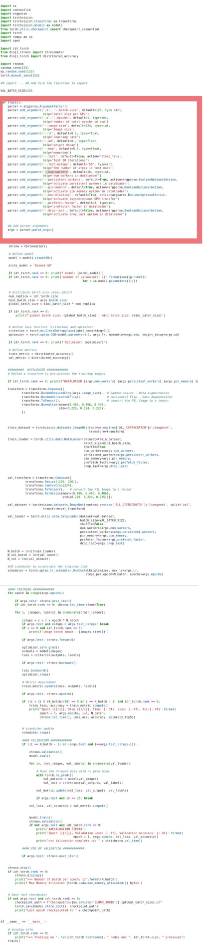
distributed_accuracy (DLO-JZ)

home metric utils (torchmetric-like)

```python
if __name__ == '__main__':

    # display info
    if idr_torch.rank == 0:
        print(">>> Training on ", len(idr_torch.hostnames), " nodes and ", idr_torch.size, " processes")
    train()
```

```python
#*******************************************
def train():
    parser = argparse.ArgumentParser()
    parser.add_argument('-b', '--batch-s
```

dlojz.py - arguments parser



```python
## import ... ## Add here the libraries to import

VAL_BATCH_SIZE=256

#*************************************************************************
def train():
    parser = argparse.ArgumentParser()
    parser.add_argument('-b', '--batch-size', default=128, type =int,
                        help='batch size per GPU')
    parser.add_argument('-e','--epochs', default=1, type=int,
                        help='number of total epochs to run')
    parser.add_argument('--image-size', default=224, type=int,
                        help='Image size')
    parser.add_argument('--lr', default=0.1, type=float,
                        help='learning rate')
    parser.add_argument('--wd', default=0., type=float,
                        help='weight decay')
    parser.add_argument('--mom', default=0.9, type=float,
                        help='momentum')
    parser.add_argument('--test', default=False, action='store_true',     ##    DON'T MODIFY    ########
                        help='Test 50 iterations')
    parser.add_argument('--test-nsteps', default='50', type=int,
                        help='the number of steps in test mode')
    parser.add_argument('--num-workers', default=10, type=int,
                        help='num workers in dataloader')
    parser.add_argument('--persistent-workers', default=True, action=argparse.BooleanOptionalAction,
                        help='activate persistent workers in dataloader')
    parser.add_argument('--pin-memory', default=True, action=argparse.BooleanOptionalAction,
                        help='activate pin memory option in dataloader')
    parser.add_argument('--non-blocking', default=True, action=argparse.BooleanOptionalAction,
                        help='activate asynchronous GPU transfer')
    parser.add_argument('--prefetch-factor', default=3, type=int,
                        help='prefectch factor in dataloader')
    parser.add_argument('--drop-last', default=False, action=argparse.BooleanOptionalAction,
                        help='activate drop_last option in dataloader')
#*************************************************************************

    ## Add parser arguments

    args = parser.parse_args()
```

Configurable Arguments :

—batch-size : batch size per GPU
—epochs : number of epochs
—image-size : image size

Optimizer :

—lr : learning rate
—wd : weight decay
—mom : momentum

Modes spéciaux :

—test : test mode
—test-nsteps : n steps for test mode

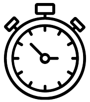
Optimisation du DataLoader :

—num-workers
—persistent-workers
—pin-memory
—non-blocking
—prefetch-factor
—drop-last

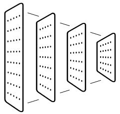
dlojz.py - instantiation

```python
## chronometer initialisation
chrono = Chronometer()

# define model
model = models.resnet50()

archi_model = 'Resnet-50'

if idr_torch.rank == 0: print(f'model: {archi_model}')
if idr_torch.rank == 0: print('number of parameters: {}'.format(sum([p.numel()
                                        for p in model.parameters()])))


# distribute batch size (mini-batch)
num_replica = idr_torch.size
mini_batch_size = args.batch_size
global_batch_size = mini_batch_size * num_replica

if idr_torch.rank == 0:
    print(f'global batch size: {global_batch_size} - mini batch size: {mini_batch_size}')


# define loss function (criterion) and optimizer
criterion = torch.nn.CrossEntropyLoss(label_smoothing=0.1)
optimizer = torch.optim.SGD(model.parameters(), args.lr, momentum=args.mom, weight_decay=args.wd)

if idr_torch.rank == 0: print(f'Optimizer: {optimizer}')

# define metrics
train_metric = distributed_accuracy()
val_metric = distributed_accuracy()
```

```python
#LR scheduler to accelerate the training time
scheduler = torch.optim.lr_scheduler.OneCycleLR(optimizer, max_lr=args.lr,
                                            steps_per_epoch=N_batch, epochs=args.epochs)
```



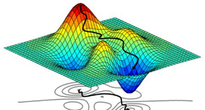
Chronometer



model : Resnet-50

mini batch size ⟺ global batch size

ε CrossEntropyLoss

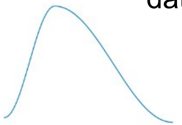


SGD Optimizer

Metric

?

need *N_batch*, given by dataloader



LR scheduler

dlojz.py - Dataloader

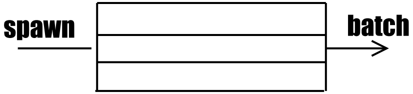
```python
#########  DATALOADER ############
# Define a transform to pre-process the training images.

if idr_torch.rank == 0: print(f"DATALOADER {args.num_workers} {args.persistent_workers} {args.pin_memory}

transform = transforms.Compose([
        transforms.RandomResizedCrop(args.image_size),  # Random resize - Data Augmentation
        transforms.RandomHorizontalFlip(),              # Horizontal Flip - Data Augmentation
        transforms.ToTensor(),                          # convert the PIL Image to a tensor
        transforms.Normalize(mean=(0.485, 0.456, 0.406),
                             std=(0.229, 0.224, 0.225))
        ])



train_dataset = torchvision.datasets.ImageNet(root=os.environ['ALL_CCFRSCRATCH']+'/imagenet',
                                              transform=transform)

train_loader = torch.utils.data.DataLoader(dataset=train_dataset,
                                           batch_size=mini_batch_size,
                                           shuffle=True,
                                           num_workers=args.num_workers,
                                           persistent_workers=args.persistent_workers,
                                           pin_memory=args.pin_memory,
                                           prefetch_factor=args.prefetch_factor,
                                           drop_last=args.drop_last)


val_transform = transforms.Compose([
          transforms.Resize((256, 256)),
          transforms.CenterCrop(224),
          transforms.ToTensor(),   # convert the PIL Image to a tensor
          transforms.Normalize(mean=(0.485, 0.456, 0.406),
                               std=(0.229, 0.224, 0.225))])

val_dataset = torchvision.datasets.ImageNet(root=os.environ['ALL_CCFRSCRATCH']+'/imagenet', split='val',
                            transform=val_transform)

val_loader = torch.utils.data.DataLoader(dataset=val_dataset,
                                         batch_size=VAL_BATCH_SIZE,
                                         shuffle=False,
                                         num_workers=args.num_workers,
                                         persistent_workers=args.persistent_workers,
                                         pin_memory=args.pin_memory,
                                         prefetch_factor=args.prefetch_factor,
                                         drop_last=args.drop_last)

N_batch = len(train_loader)
N_val_batch = len(val_loader)
N_val = len(val_dataset)
```

<u>train dataset :</u>

RandomResizedCrop
RandomHorizontalFlip
+ Normalize

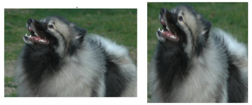
Shuffling

DataLoader optimization



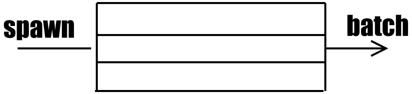
<u>validation dataset :</u>

Resize
CenterCrop
+ Normalize



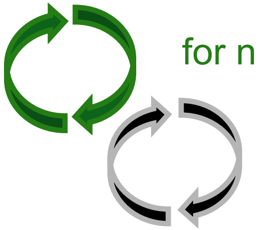
no shuffling

DataLoader optimization



dlojz.py - Training

```python
chrono.start()

#### TRAINING ############
for epoch in range(args.epochs):

    if args.test: chrono.next_iter()
    if idr_torch.rank == 0: chrono.tac_time(clear=True)

    for i, (images, labels) in enumerate(train_loader):

        csteps = i + 1 + epoch * N_batch
        if args.test and csteps > args.test_nsteps: break
        if i == 0 and idr_torch.rank == 0:
            print(f'image batch shape : {images.size()}')

        if args.test: chrono.forward()

        optimizer.zero_grad()
        outputs = model(images)
        loss = criterion(outputs, labels)

        if args.test: chrono.backward()

        loss.backward()
        optimizer.step()

        # Metric mesurement
        train_metric.update(loss, outputs, labels)

        if args.test: chrono.update()

        if ((i + 1) % (N_batch//10) == 0 or i == N_batch - 1) and idr_torch.rank == 0:
            train_loss, accuracy = train_metric.compute()
            print('Epoch [{}/{}], Step [{}/{}], Time: {:.3f}, Loss: {:.4f}, Acc:{:.4f}'.format(
                  epoch + 1, args.epochs, i+1, N_batch,
                  chrono.tac_time(), loss_acc, accuracy, accuracy_top5))


    # scheduler update
    scheduler.step()
```



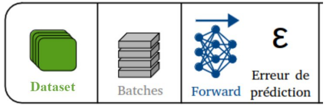
for n epochs

for each *batch*

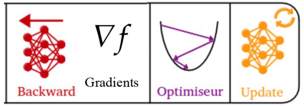
test mode : 50 steps

CPU compute by default !!

```python
optimizer.zero_grad()
outputs = model(images)
loss = criterion(outputs, labels)
```

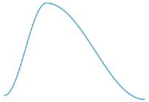



```python
loss.backward()
optimizer.step()
```



Aggregate the metrics (loss, accuracy)
10x per epoch, compute and print the metrics

Log 10x per epoch



Step up LR scheduler

dlojz.py - Validation

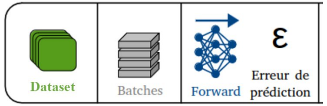
```python
#### VALIDATION ############
if ((i == N_batch - 1) or (args.test and i==args.test_nsteps-1)) :

    chrono.validation()
    model.eval()

    for iv, (val_images, val_labels) in enumerate(val_loader):

        # Runs the forward pass with no grad mode.
        with torch.no_grad():
            val_outputs = model(val_images)
            val_loss = criterion(val_outputs, val_labels)

        val_metric.update(val_loss, val_outputs, val_labels)

        if args.test and iv >= 20: break

    val_loss, val_accuracy = val_metric.compute()


    model.train()
    chrono.validation()
    if not args.test and idr_torch.rank == 0:
        print('##EVALUATION STEP##')
        print('Epoch [{}/{}], Validation Loss: {:.4f}, Validation Accuracy: {:.4f}'.format(
                            epoch + 1, args.epochs, val_loss, val_accuracy))
        print(">>> Validation complete in: " + str(chrono.val_time))

#### END OF VALIDATION ############
```

after each epoch
(or at the end of test mode)

for each *batch of validation*

(test mode : 20 steps)

```python
# Runs the forward pass with no grad mode.
with torch.no_grad():
    val_outputs = model(val_images)
    loss = criterion(val_outputs, val_labels)
```



Aggregate the metrics (loss, accuracy)

when it is over:

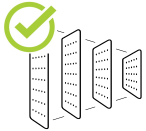
compute & **Log**

dlojz.py – Checkpoint & Report

```python
chrono.stop()
if idr_torch.rank == 0:
    chrono.display()
    print(">>> Number of batch per epoch: {}".format(N_batch))
    print(f'Max Memory Allocated {torch.cuda.max_memory_allocated()} Bytes')


# Save last checkpoint
if not args.test and idr_torch.rank == 0:
    checkpoint_path = f"checkpoints/{os.environ['SLURM_JOBID']}_{global_batch_size}.pt"
    torch.save(model.state_dict(), checkpoint_path)
    print("Last epoch checkpointed to " + checkpoint_path)
```

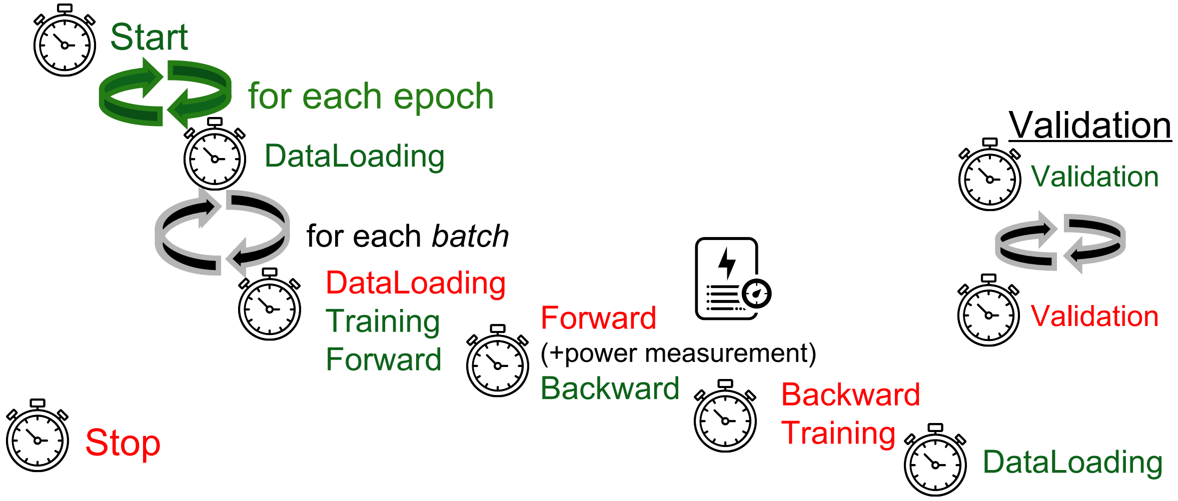
Log + Chronometer Display

Checkpoint at the end of training (=> not in test mode)



pre-trained model

dlojz.py - Chronometer



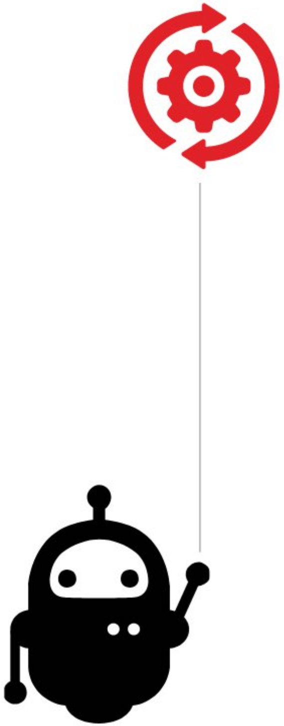
Display

```python
def display(self, val_steps):
    if self.rank == 0:
        print(">>> Training complete in: " + str(datetime.now() - self.start_proc))
        if self.test:
            print(">>> Training performance time: min {} avg {} seconds (+/- {})".format(np.min(self.time_perf_train[1:]), np.median(self.time_perf_train[1:]),
np.std(self.time_perf_train[1:])))
            print(">>> Loading performance time: min {} avg {} seconds (+/- {})".format(np.min(self.time_perf_load[1:]), np.mean(self.time_perf_load[1:]),
np.std(self.time_perf_load[1:])))
            print(">>> Forward performance time: {} seconds (+/- {})".format(np.mean(self.time_perf_forward[1:]), np.std(self.time_perf_forward[1:])))
            print(">>> Backward performance time: {} seconds (+/- {})".format(np.mean(self.time_perf_backward[1:]), np.std(self.time_perf_backward[1:])))
            if len(self.power)>0: print(">>> Peak Power during training: {} W)".format(np.max(self.power)))
            print(">>> Validation time estimation: {}".format(self.val_time/20 * val_steps))
            print(">>> Sortie trace #####################################" )
            print(">>>JSON", json.dumps({'GPU process - Forward/Backward':self.time_perf_train, 'CPU process - Dataloader':self.time_perf_load}))
```

dlojz.py – Distributed_accuracy

```python
class distributed_accuracy():
    def __init__(self):
        self.dist = dist.is_initialized()
        self.correct = torch.tensor(0)
        self.total = torch.tensor(0)
        self.loss = torch.tensor(0, dtype=torch.float)

    def update(self, losses, outputs, labels):
        _, predicted = torch.max(outputs.data, 1)
        ## for mixed data augmentation
        if len(labels.size()) > 1: labels = torch.argmax(labels, dim=1)
        self.correct += (predicted == labels).sum().item()
        self.total += labels.size(0)
        self.loss += losses.sum().item()

    def clear(self):
        self.correct = torch.tensor(0)
        self.total = torch.tensor(0)
        self.loss = torch.tensor(0, dtype=torch.float)

    def compute(self):
        if self.dist and idr_torch.size > 1:
            self.correct = self.correct.to('cuda')
            self.total = self.total.to('cuda')
            self.loss = self.loss.to('cuda')
            dist.all_reduce(self.correct, op=dist.ReduceOp.SUM)
            dist.all_reduce(self.total, op=dist.ReduceOp.SUM)
            dist.all_reduce(self.loss, op=dist.ReduceOp.SUM)
        accuracy = (self.correct / self.total).item()
        loss = (self.loss / self.total).item()
        self.clear()
        return loss, accuracy
```

TP0 : Préparation de l’environnement



- Lancer un terminal et faire les copies nécéssaires

```
local:~$ ssh jean-zay

jz:~$ cd $WORK
jz:~$ cp -r $ALL_CCFRWORK/DLO-JZ .
```

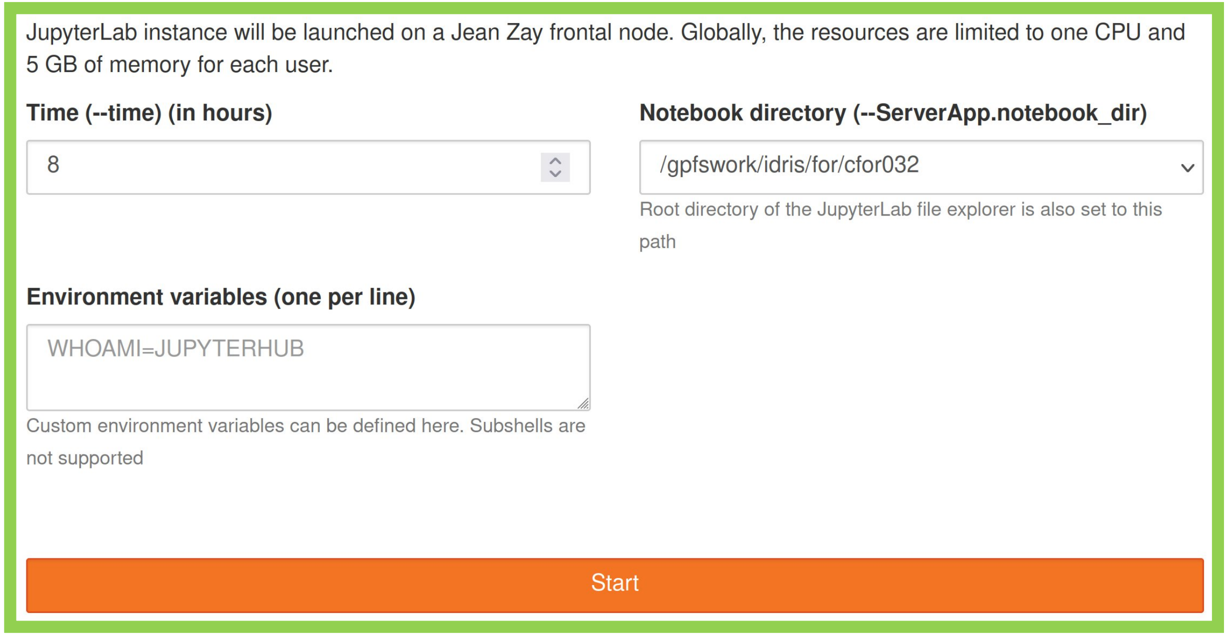
- Lancer firefox
- Accéder à jupyterhub.idris.fr

TP0 : Accès et prise en main de JupyterHub

- Se connecter avec vos identifiants de formation
- Lancer une instance

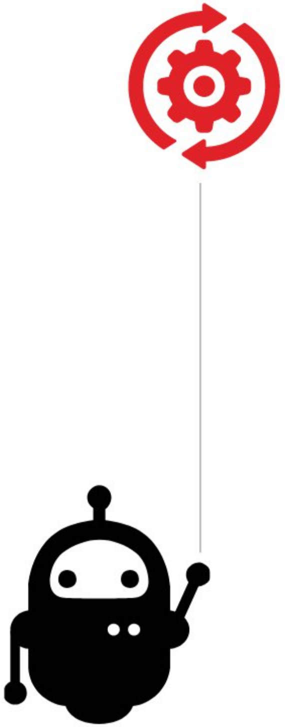
- Sélectionner le spawner ‘Interactive’

- Remplir la configuration



- Start

TP0 : Accès et prise en main du notebook



- Ouvrir le notebook DLO-JZ_Jour1.ipynb
- Choisir le kernel pytorch-gpu/py3/2.1.1 (en haut à droite) s’il n’est pas détecté automatiquement
- Choisir un pseudonyme
- Lancer un job
- Prendre en main le script de référence et les différentes fonctionnalités

Les enjeux de la montée à l'échelle

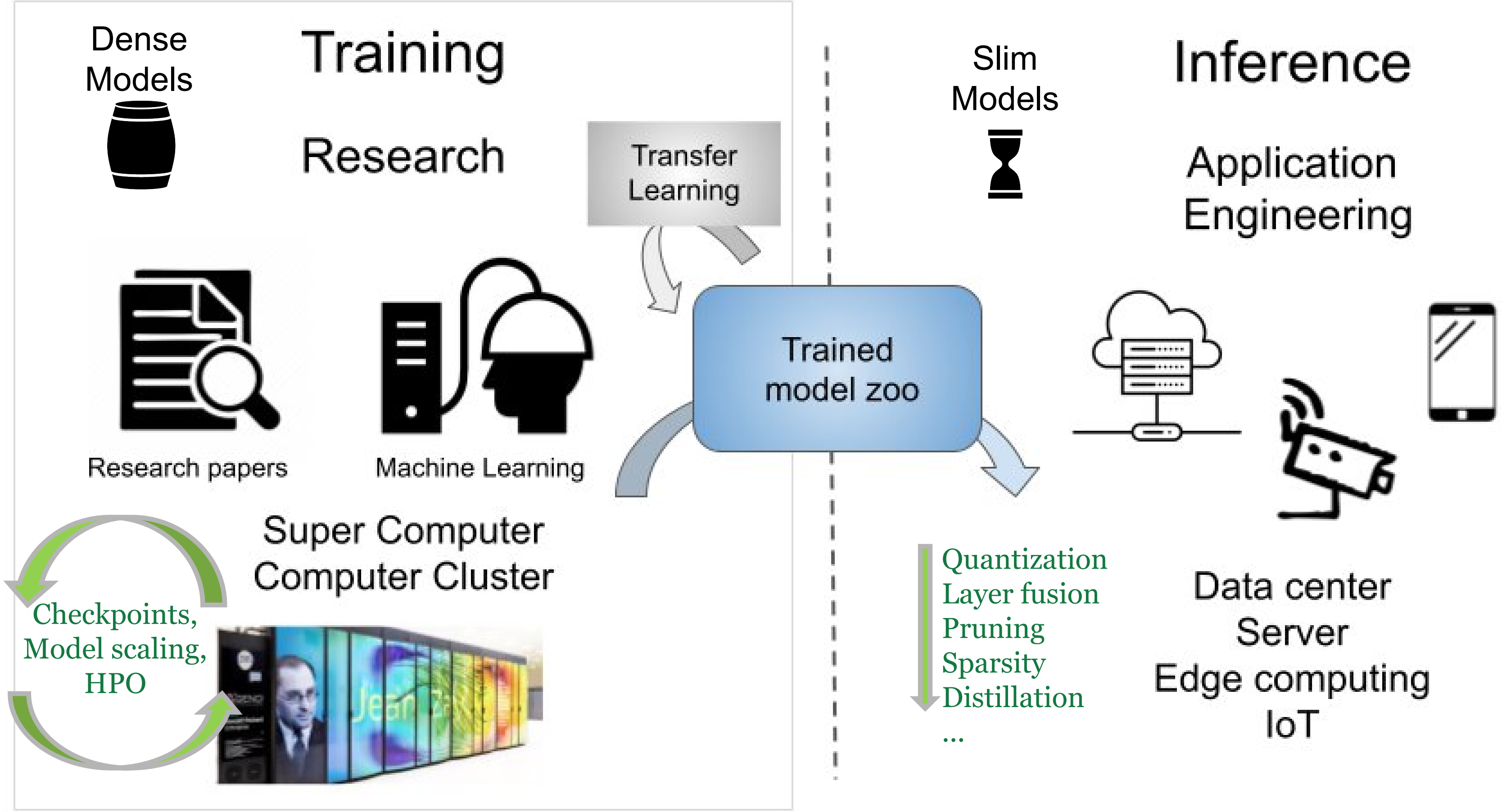
Temps d’apprentissage ◂

Empreinte mémoire ◂

Solutions ◂

Economie énergétique ◂

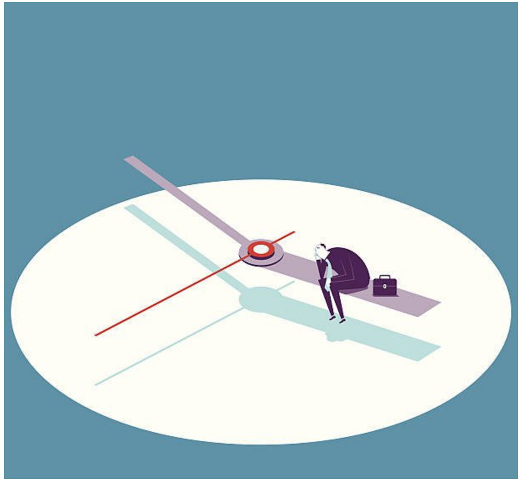
Apprentissage / Inférence



Contraintes du Deep Learning

2 problèmes à traiter:

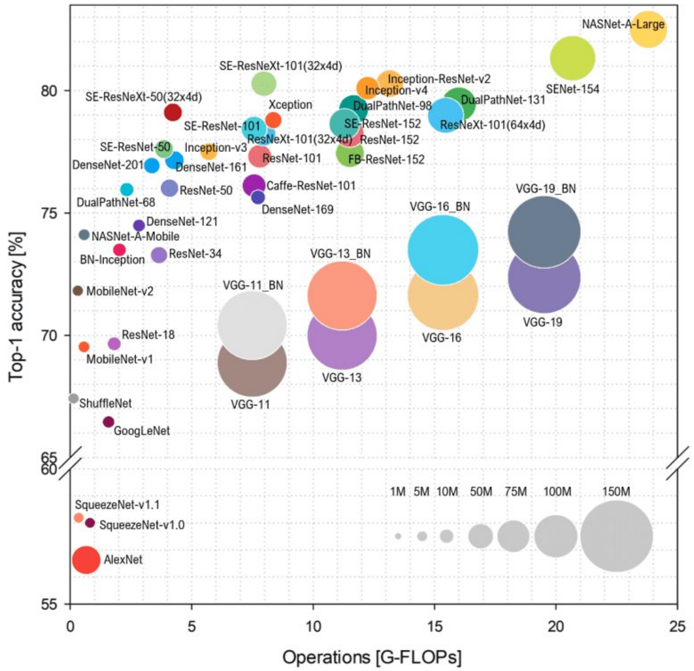
Temps d'apprentissage



Surconsommation mémoire (OOM)

Les gros modèles

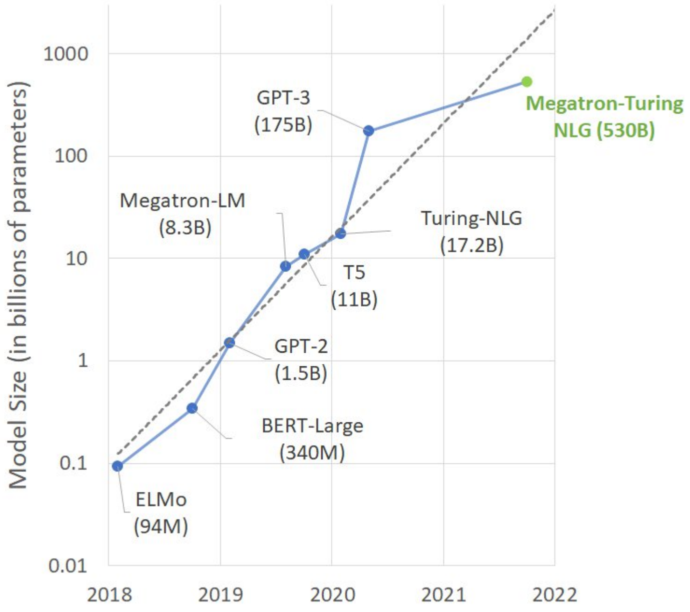
Convolutional Neural Network



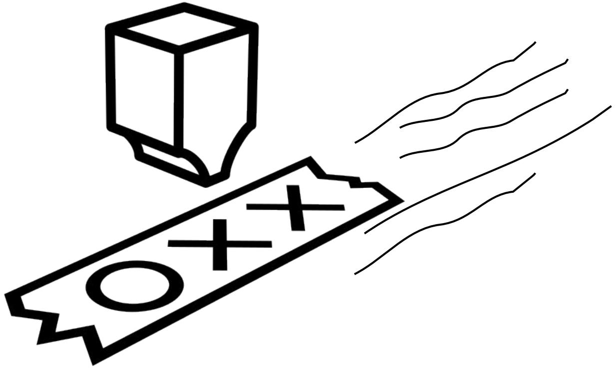
Les modèles gros, et profonds permettent d'obtenir de meilleurs métriques d'*accuracy*.

Les énormes modèles provoquent de très coûteux temps de calcul et de larges empreintes mémoire (4 Go pour un modèle d'1 milliard de paramètres).

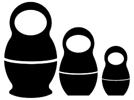
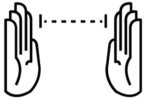
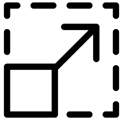
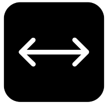
Transformers

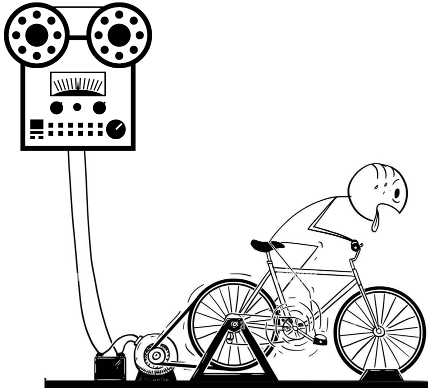


Le temps de calcul

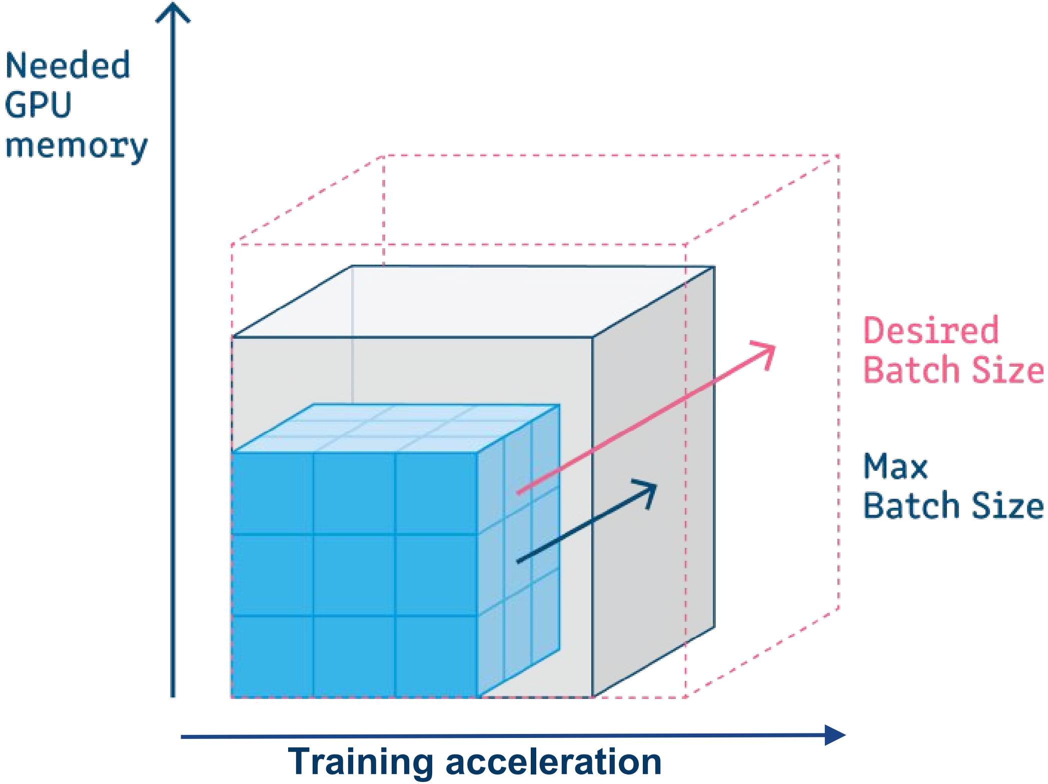


Le temps de calcul augmente avec le **nombre de FLOP nécessaire**, dépendant de :

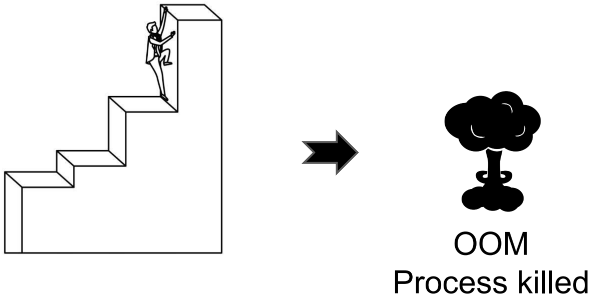
- La taille du modèle 
- La profondeur du modèle 
- La taille des données d’entrée (Résolution des images, longueur de la séquence, …*)* 
- La taille du *dataset* 
- Nombre d’*epochs* nécessaire



Taille de batch et mémoire



Augmenter la taille du batch et ainsi augmenter le pas d'itération permet d'accélérer l'apprentissage.



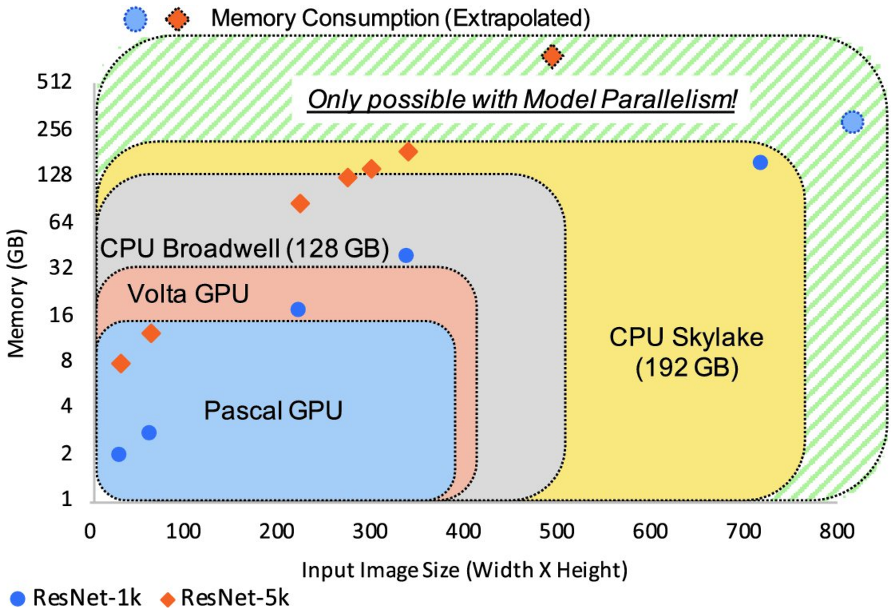
Cependant cela augmente d'autant l'empreinte mémoire risquant d'atteindre la limite du système.

Données à haute dimension

Les données à haute dimension provoquent de sérieux **problèmes d'occupation de mémoire** pendant l'apprentissage, accentués par **la profondeur du modèle**.

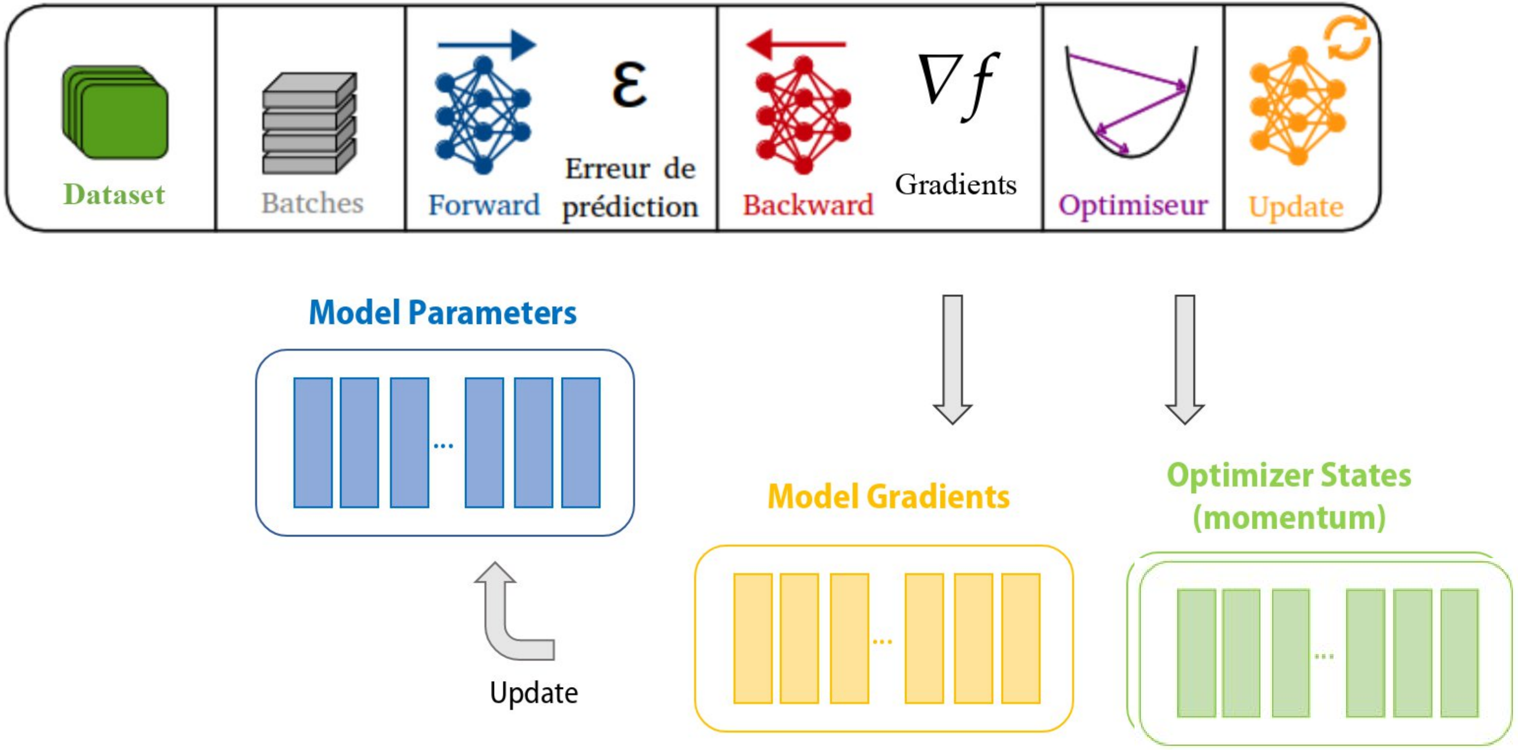
- ➔ Texte (N, 100, 500) ***~x1***
- ➔ Image 2D (N, 226, 226, 3) ***~x3***
- ➔ Image 3D (N, 226, 226, 100, 3) ***~x300***
- ➔ Video (N, 100, 226, 226, 3) ***~x300***

(GNN : Graph de petit à très très gros !!)

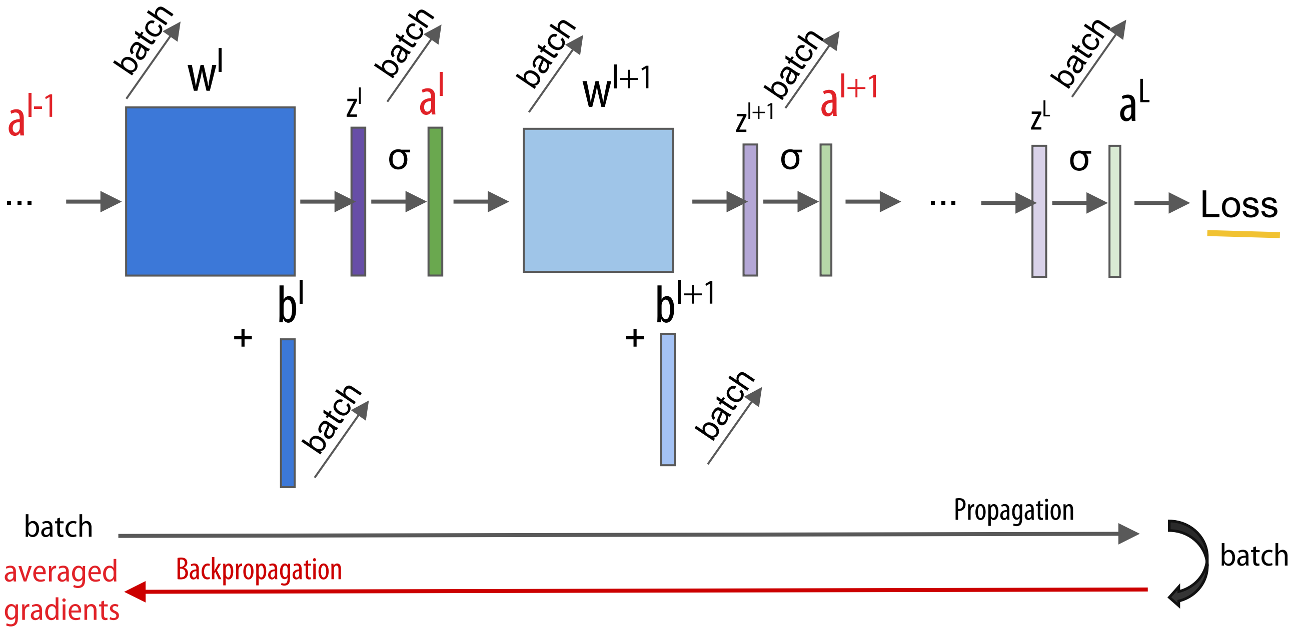


Source : HyPar-Flow

Forward / Backward – mémoire du modèle



Forward / Backward - problème des activations



Propagation

$$a^l = \sigma(w^l a^{l\text{-}1} + b^l) = \sigma z^l$$

Backpropagation

$$\delta^l = \frac{\partial C}{\partial z^l} \qquad w^l \rightarrow w^l - \frac{\eta}{m} \cdot \frac{\partial C}{\partial w^l} \qquad b^l \rightarrow b^l - \frac{\eta}{m} \cdot \frac{\partial C}{\partial b^l}$$

$$\delta^L = \nabla_a C \odot \sigma'(z^L)$$

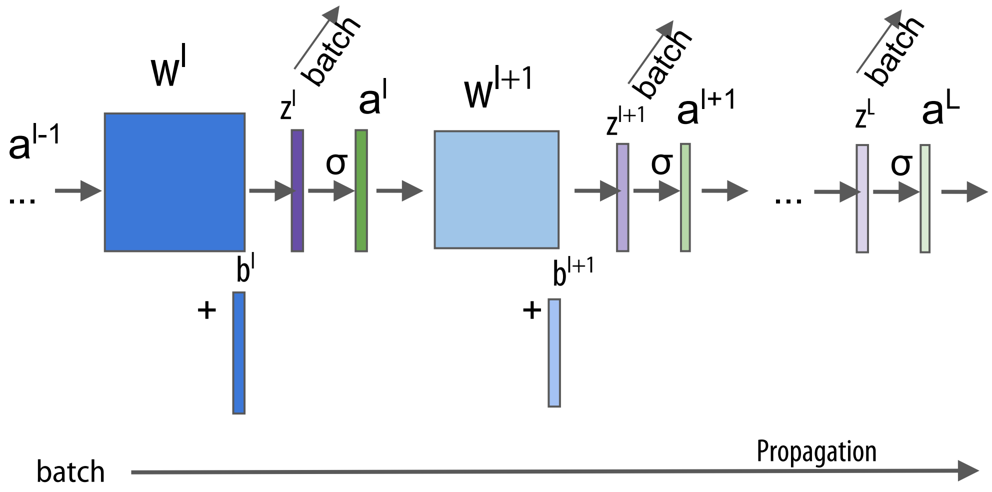
$$\delta^l = ((w^{l+1})^T \delta^{l+1}) \odot \sigma'(z^l)$$

$$\frac{\partial C}{\partial w^l} = \delta^l (a^{l-1})^T$$

$$\frac{\partial C}{\partial b^l} = \delta^l$$

Note: Pour la *backpropagation*, il est nécessaire de garder en mémoire les activations intermédiaires.

Inférence et évaluation



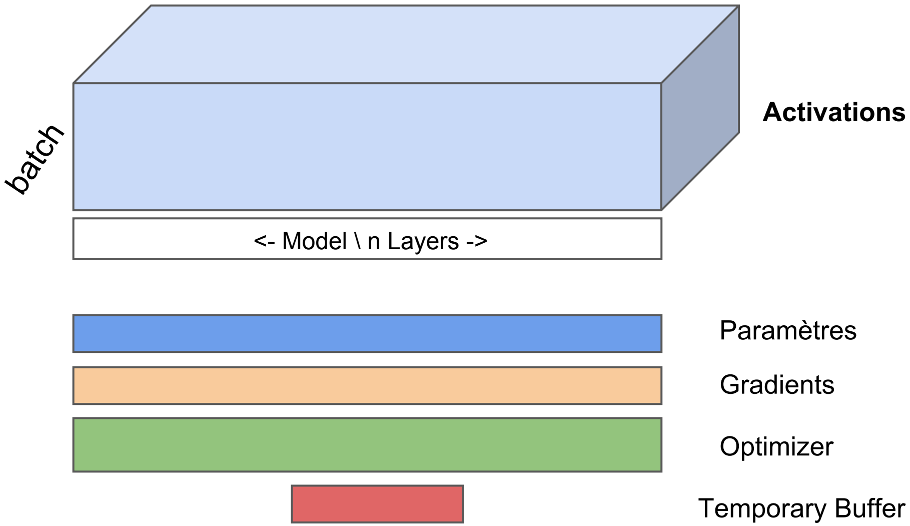

```
...
with torch.no_grad():
    val_outputs = model(val_images)
    loss = criterion(val_outputs, val_labels)
...
```

Propagation

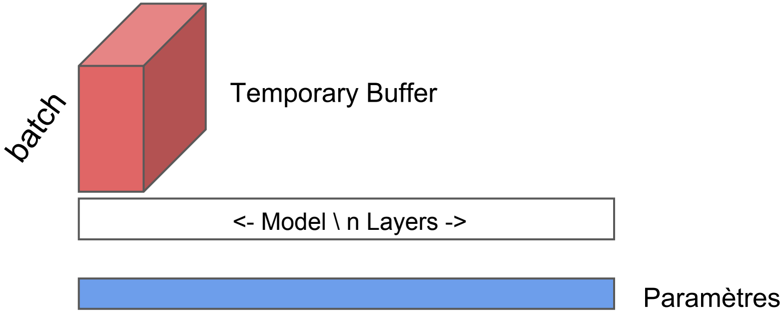
$$a^l = \sigma(w^l a^{l-1} + b^l) = \sigma z^l$$

Empreinte mémoire

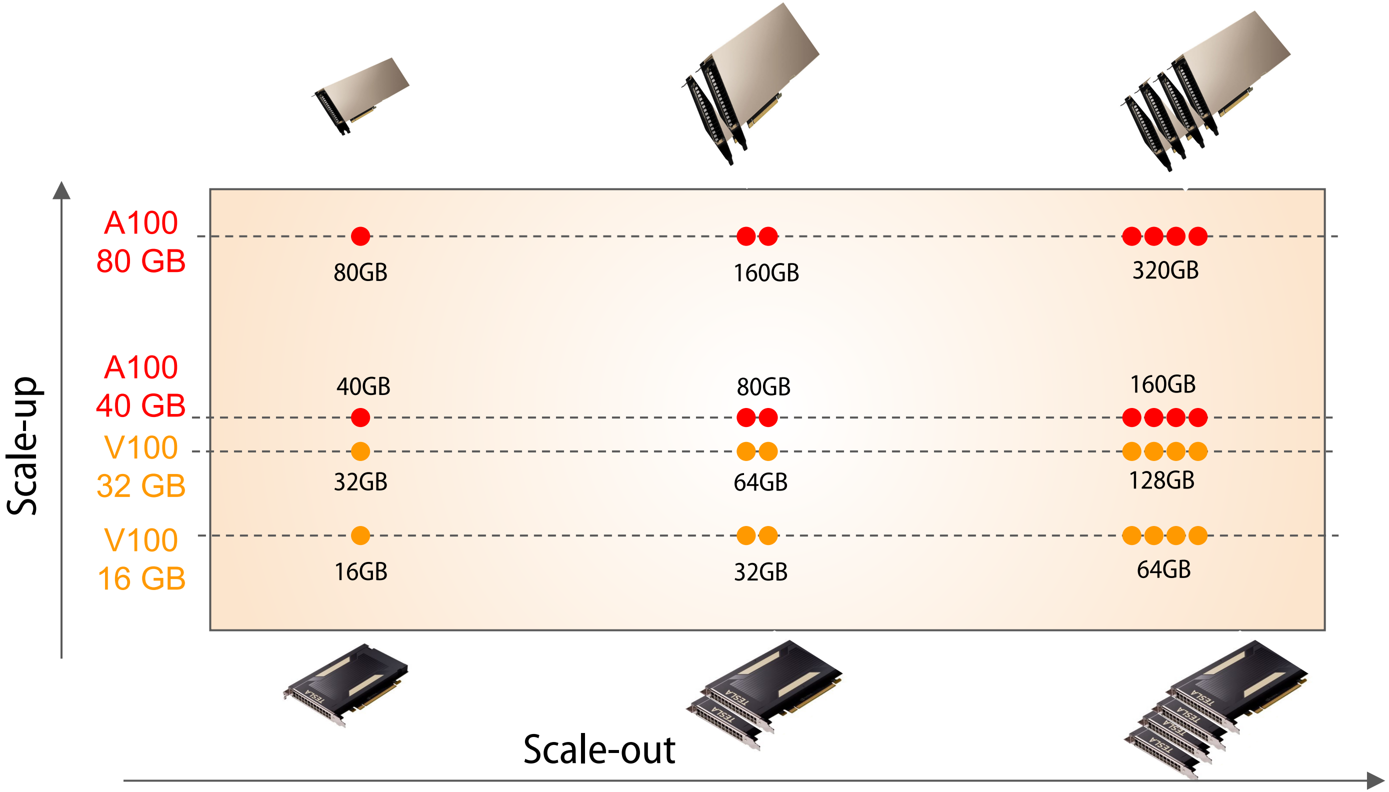
Training



Inference / Evaluation

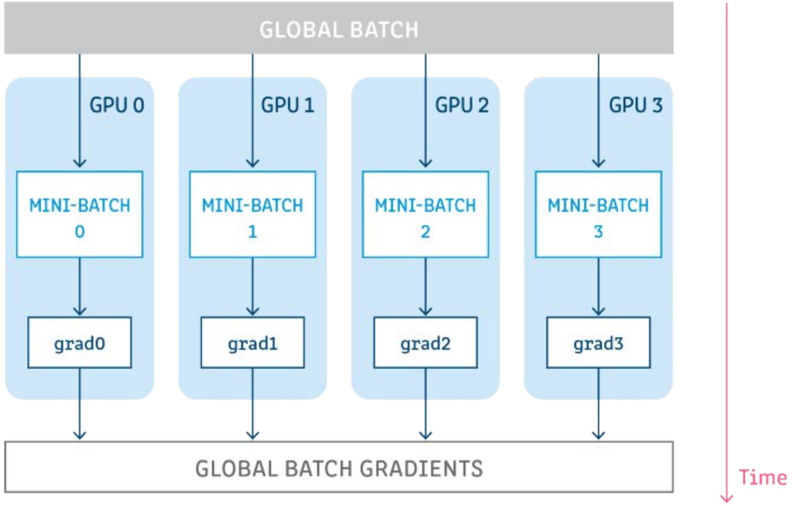


Solutions système

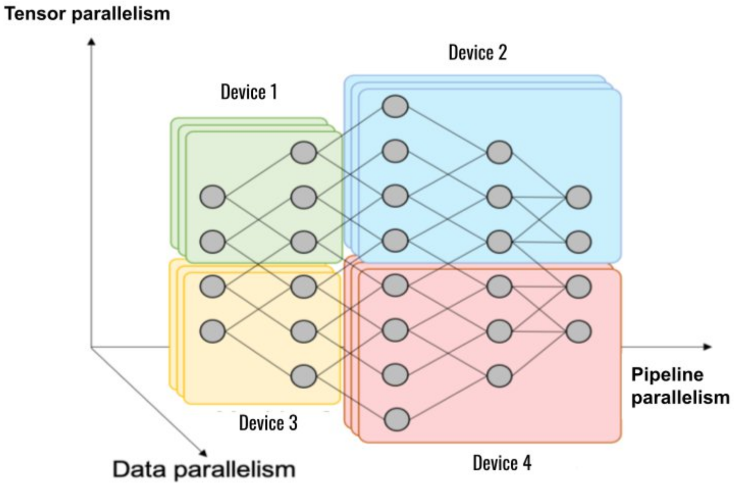


Solutions: Distribution – Scale-out

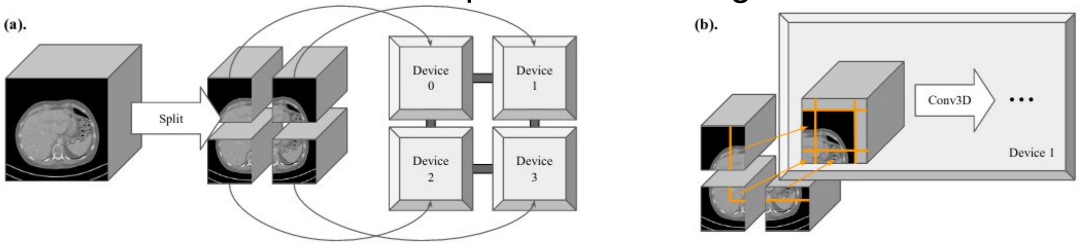
Data Parallelism



Model Parallelism

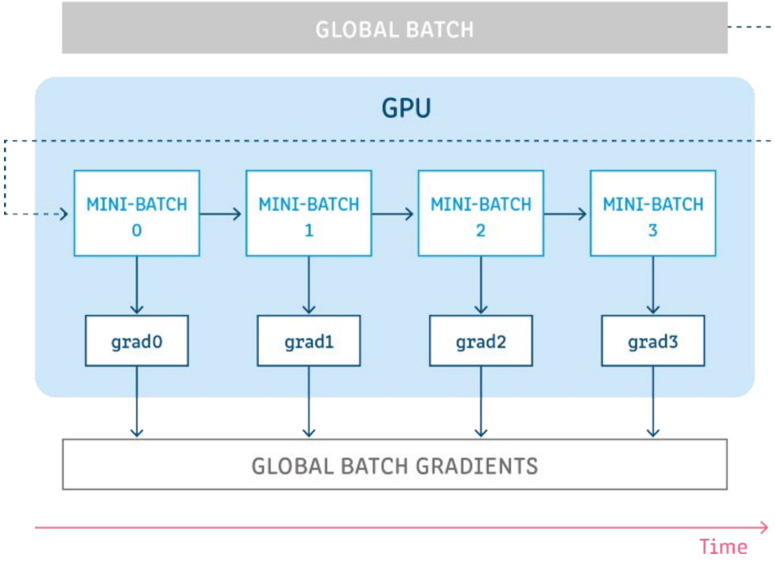


Spatial Partitioning

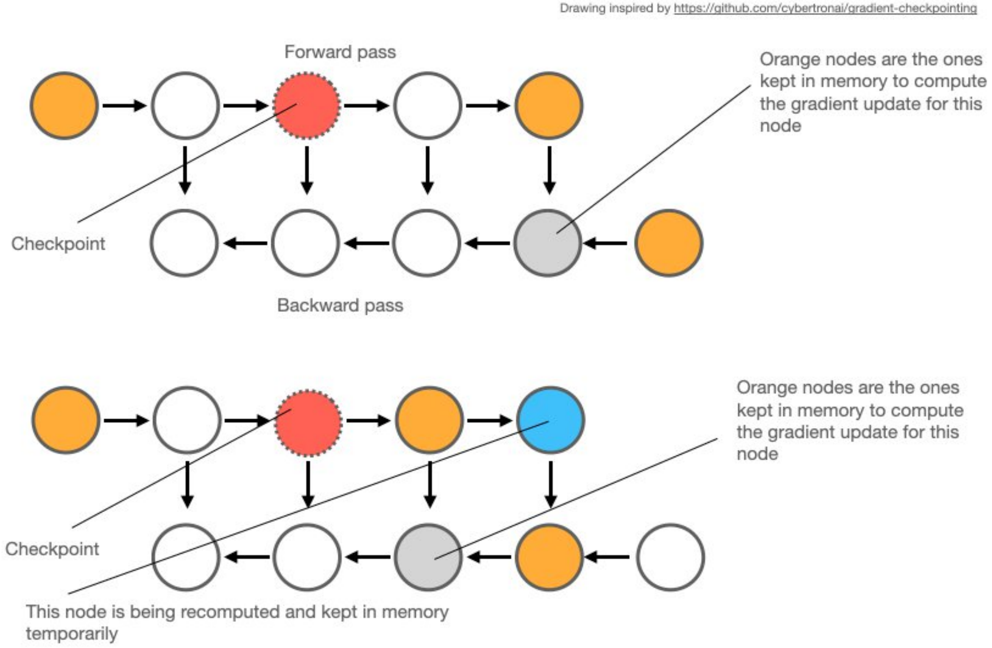


Solutions de contournement

Gradient aggregation



Gradient/activation checkpointing

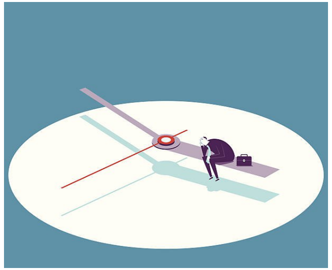


Un 3e problème à traiter ...

La consommation électrique !!

2 problèmes à traiter:

Temps d'apprentissage



Surconsommation mémoire (OOM)

Consommation énergétique

	A100 PCIe	A100 SXM2	V100 PCIe	V100 SXM2
Max Power	250W	400W	250W	300W
Idle Power	~30W	~60W	~40W	~45W
Performance	90%	100%	45%	50%

Pour un nœud : Le CPU (souvent 2 processeurs) consomme ce que consomme à peu près 1 GPU.

La consommation électrique varie selon l'utilisation partielle ou globale du GPU.

Cependant le rapport performance énergétique est en faveur d'une pleine utilisation du GPU.

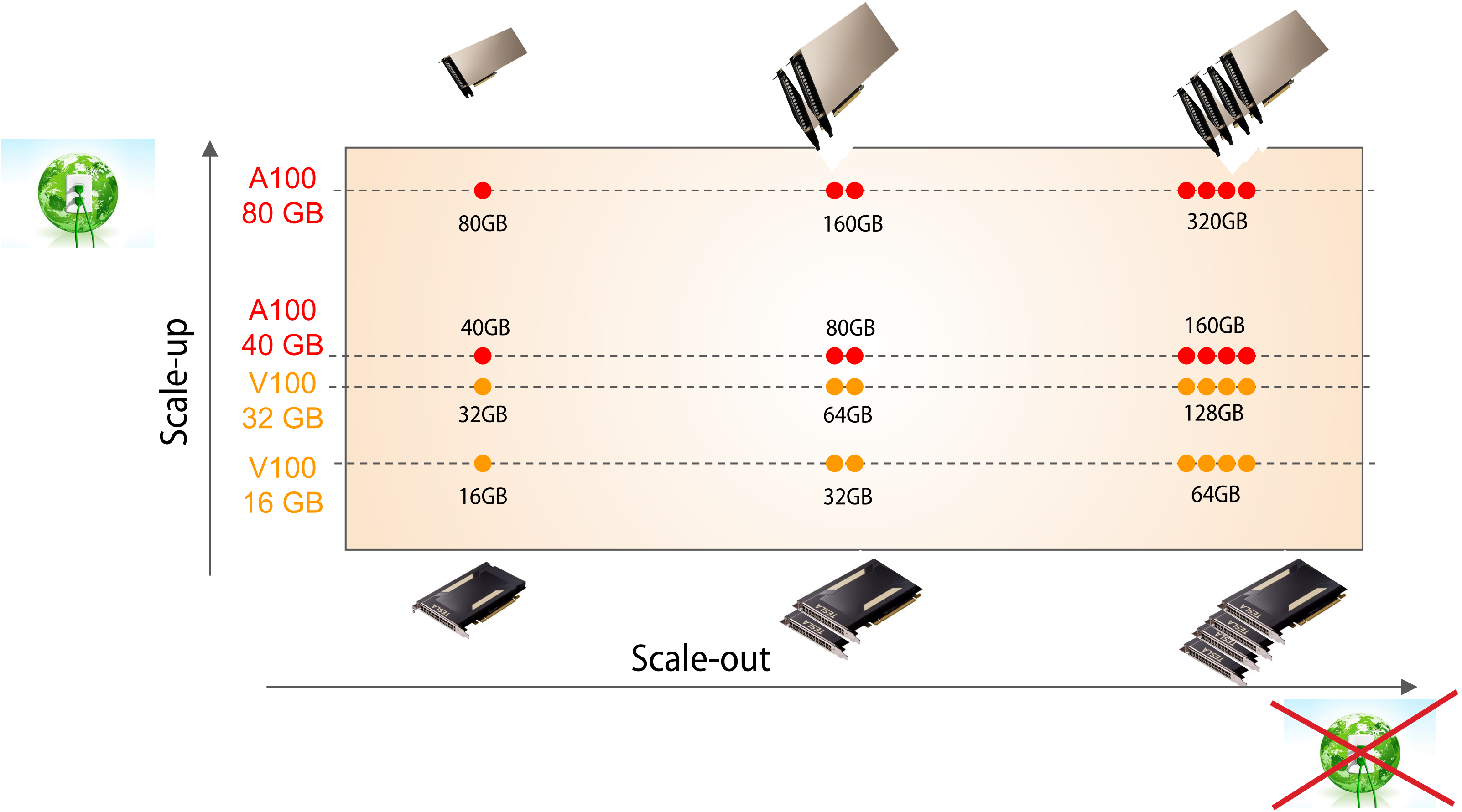
Économie énergétique / Heures GPU

Économie énergétique
≅
Économie d'heures GPU

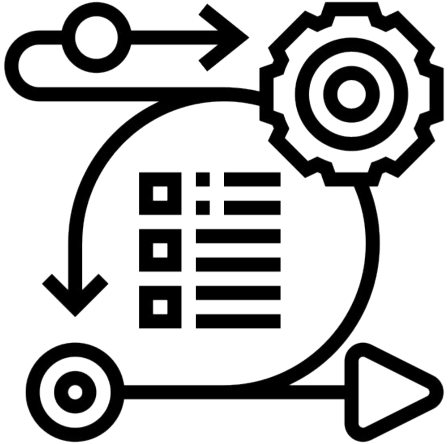
Optimisation du système (DLO-JZ)

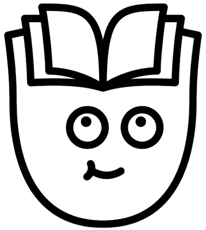
- Chercher le *throughput* le plus important
- Optimiser le chargement de données pour éliminer les temps vides du GPU
- Paralléliser l'apprentissage à la bonne mesure du modèle : ni trop, ni pas assez

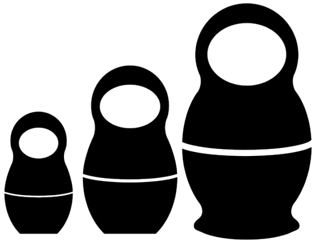
Économie énergétique / Heures GPU



Économie énergétique / Heures GPU







Méthodologie (économiser la recherche, ne pas répéter les apprentissages inutilement)

- Chercher les hypers paramètres dans les publications et reproduire l'état de l'art

- Chercher les bons hypers paramètres sur des plus petits modèles, puis appliquer à l'échelle

- Techniques d'*Hyper-Parameter Optimization* (HPO)

GPU computing

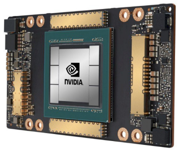
V100, A100 ◀

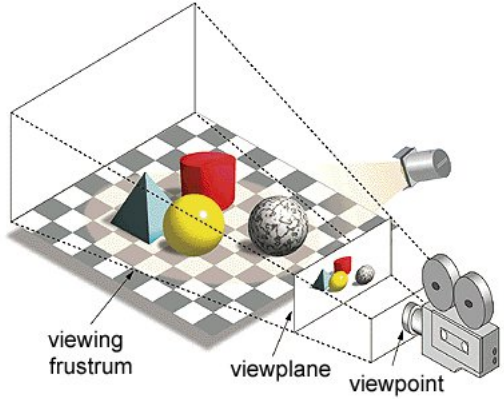
CUDA ◀

CuDNN ◀

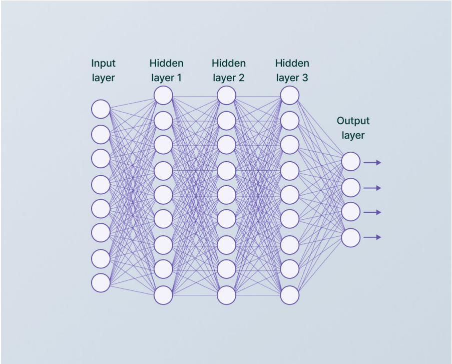
AMP ◀

GPU computing

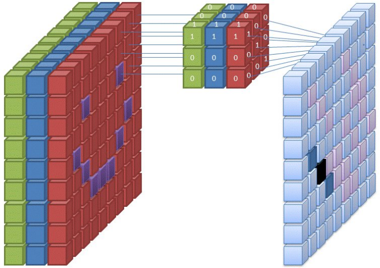




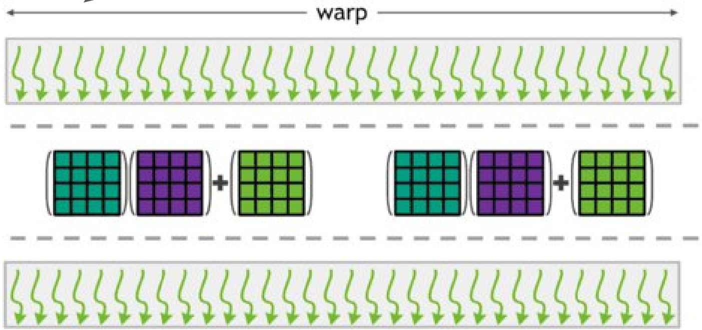
GPU Rendering & Game Graphics Pipeline



NN

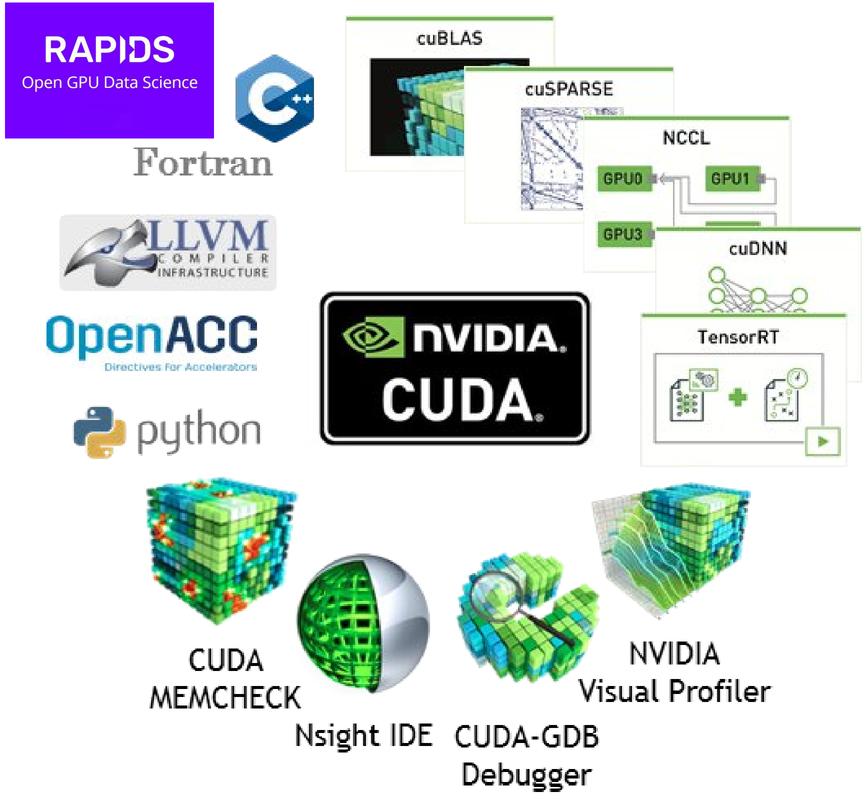


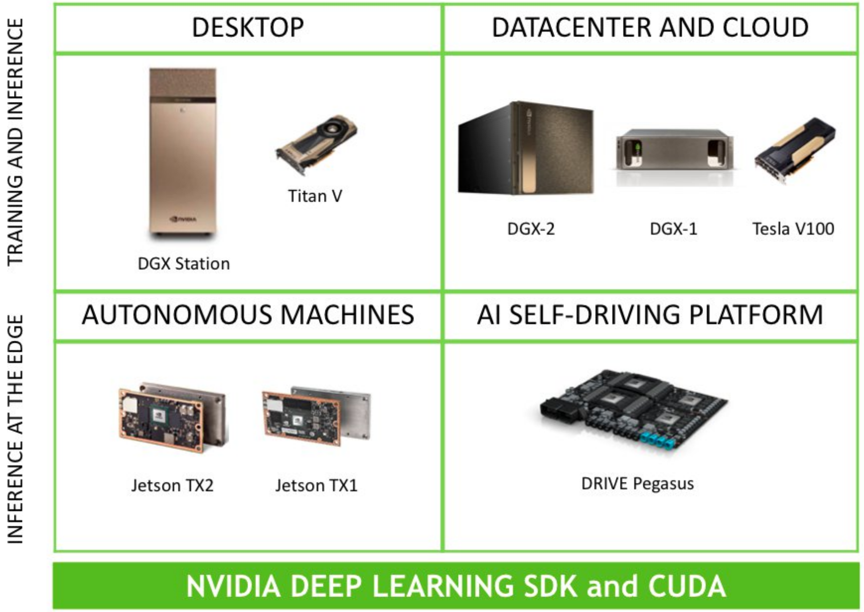
CNN



Matrix Multiply-accumulate operations

Galaxie NVIDIA

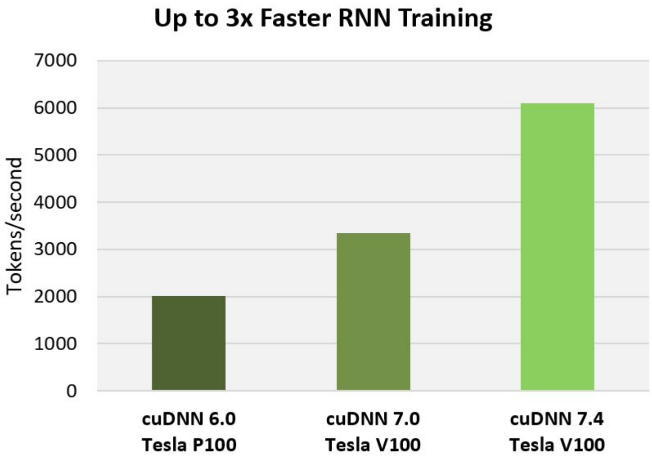




Source : NVidia

CuDNN

NVIDIA DEEP LEARNING SDK and CUDA

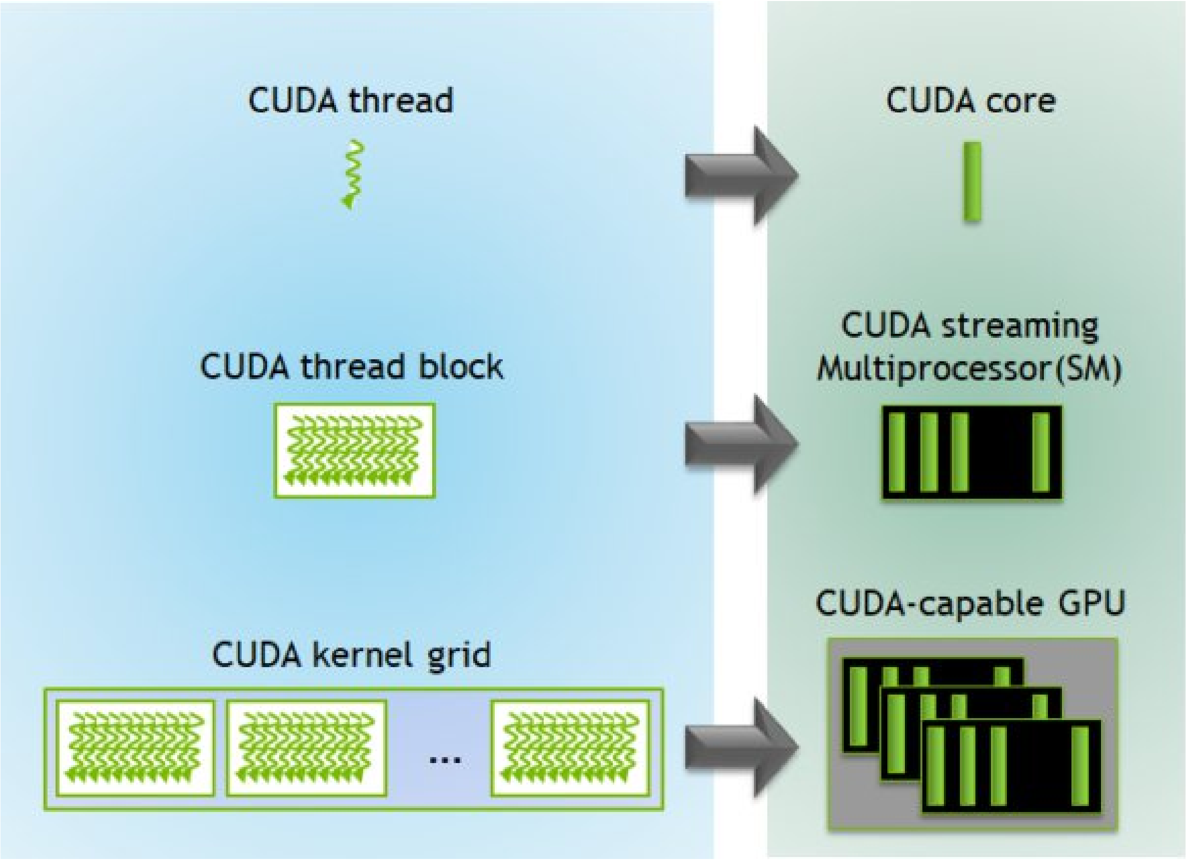


TensorFlow performance (tokens/sec), Tesla P100 + cuDNN 6 (FP32) on 17.12 NGC container, Tesla V100 + cuDNN 7.0 (Mixed) on 18.02 NGC container, Telsa V100 + cuDNN 7.4 (Mixed) on 18.10 NGC container, OpenSeq2Seq (GNMT), Batch Size: 64

L'ingénierie CUDA pour le deep learning sur GPU est gérée par cuDNN.
Merci cuDNN !!

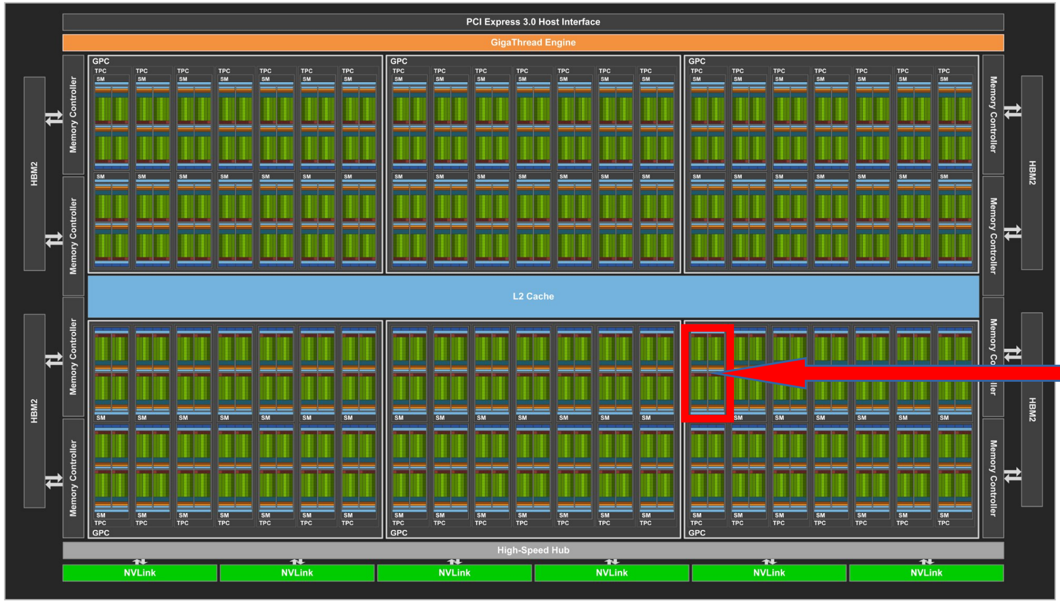
Recommandation: pour optimiser l'utilisation des *Tensor Cores* et des *Cuda Cores* : Utiliser des tenseurs aux dimensions (*batch size*, *sample size*, *channel*, layer dimension, etc …) **multiples de 8 !!**

GPU computing : CUDA



Source : NVidia

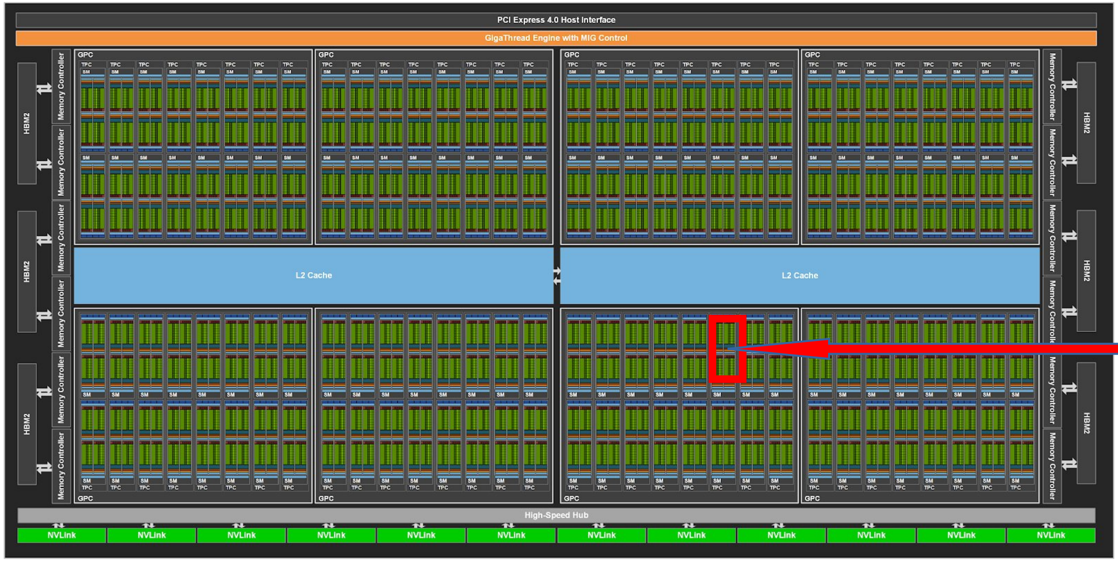
Architecture V100



- 6 GPC
- **84** Streaming Multiprocessors (SMs)
- 5376 CUDA Cores
- **672 2eGen** Tensor Cores per full GPU

Source : NVidia

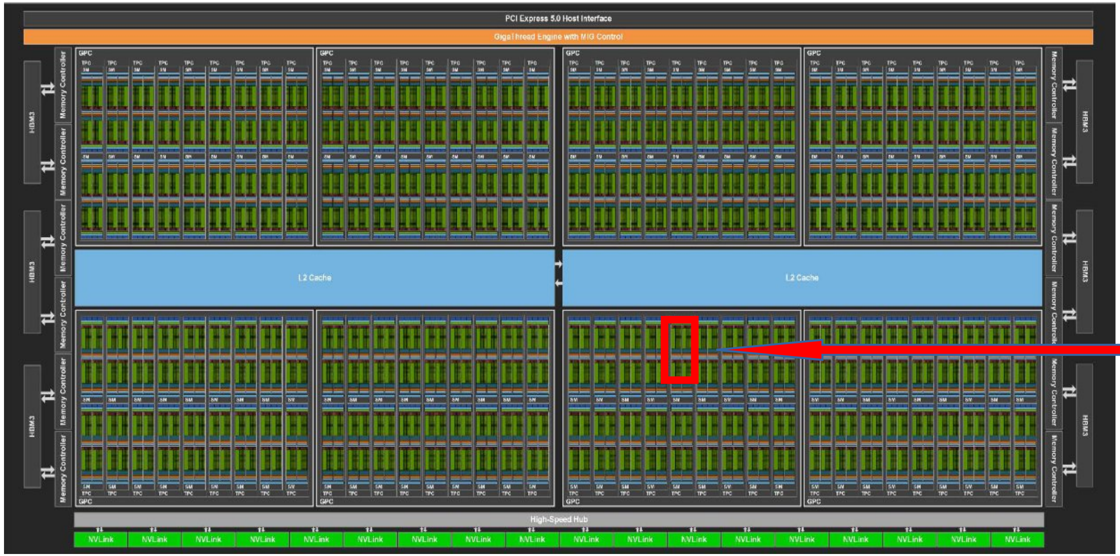
Architecture A100



- 8 GPC
- **128** Streaming Multiprocessors (SMs)
- 8192 CUDA Cores
- **512 3eGen** Tensor Cores per full GPU

Source : NVidia

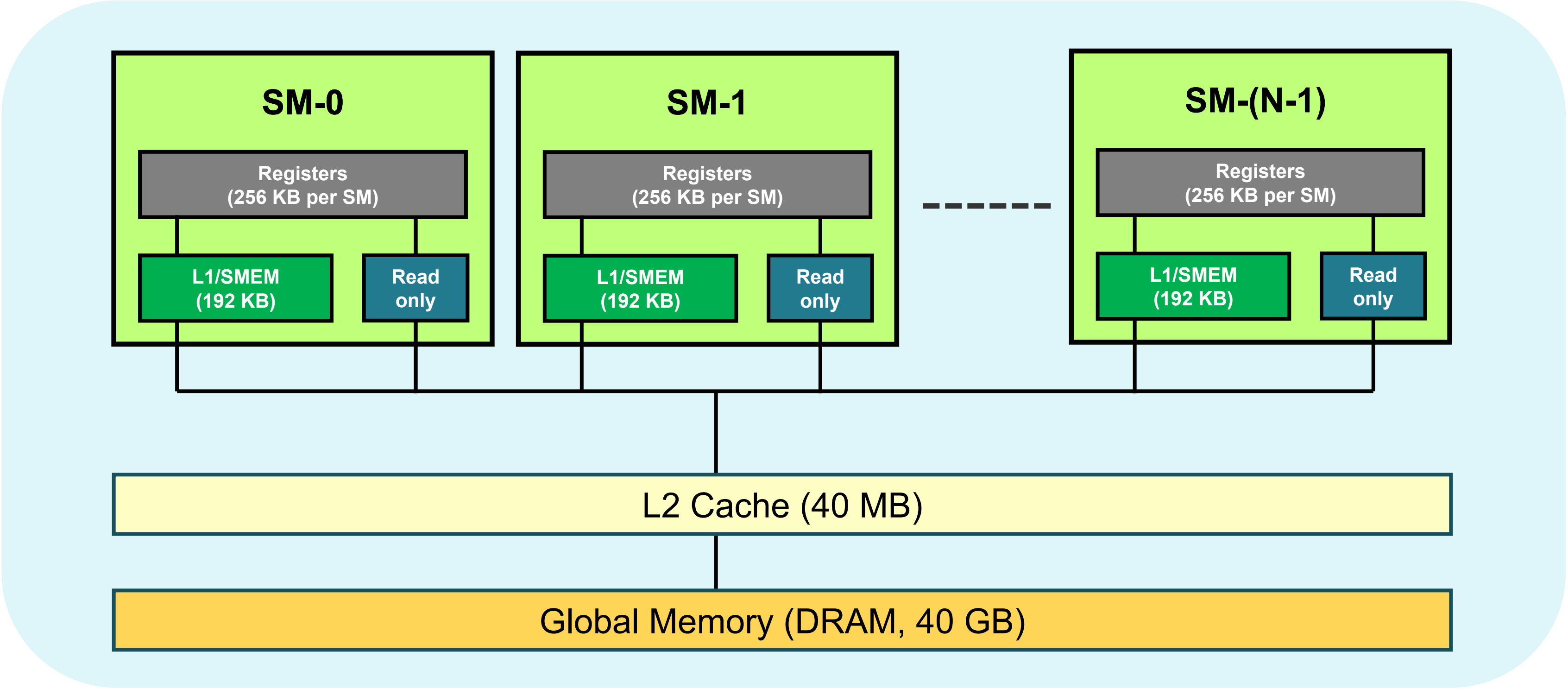
Architecture H100



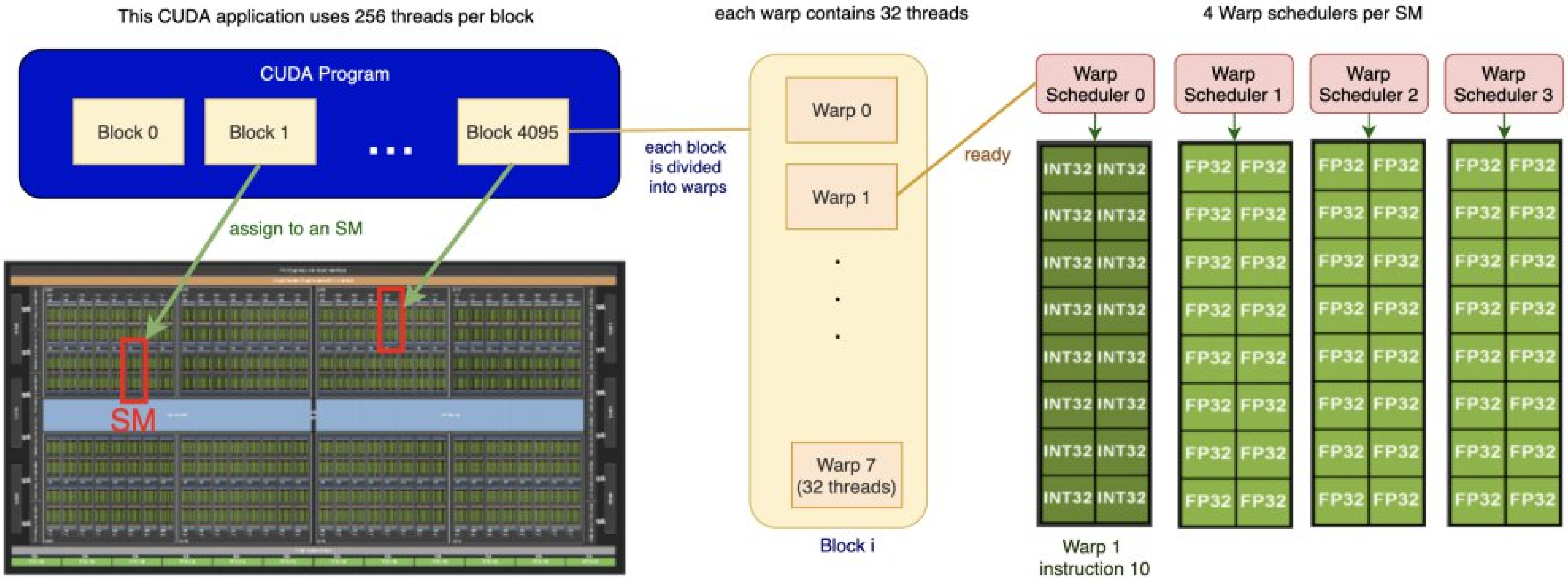
- 8 GPC
- **132** Streaming Multiprocessors (SMs)
- 16896 CUDA Cores
- **528 4eGen** Tensor Cores per full GPU

Source : NVidia

Gestion de la mémoire optimisée



Ingénierie CUDA



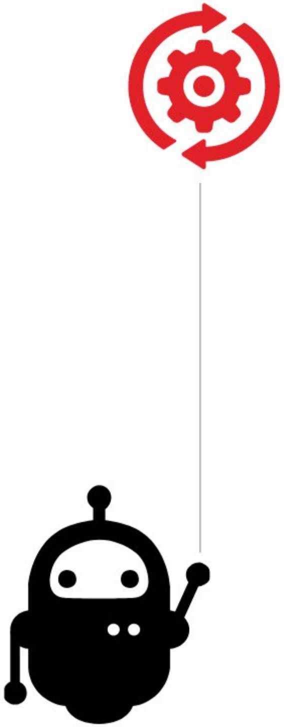
Optimisation :

- du remplissage d'un *block*
- de l'étalement sur le GPU

Optimisation avancée :

- Fusion des kernels pour économiser les temps d'initialisation

TP1 : Accélération GPU



- Envoyer le calcul sur le GPU
- Test Mémoire

Tensor Cores

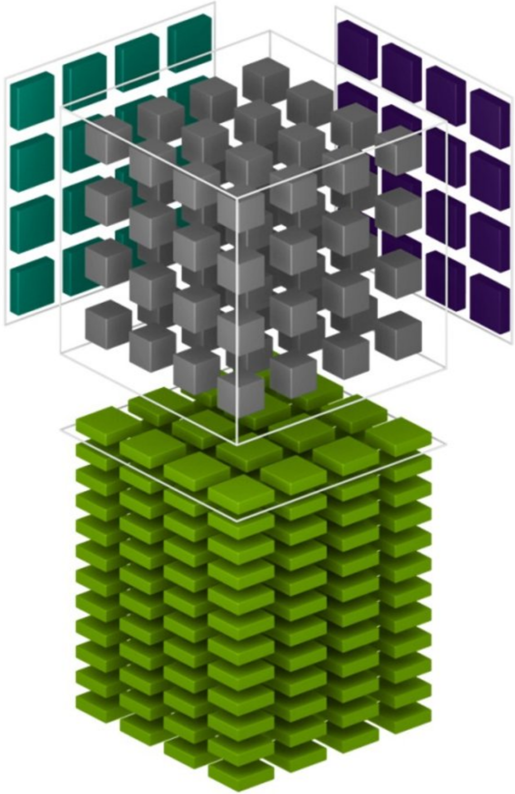
Tensor Cores ◀

Precisions ◀

AMP ◀

Channel last memory format ◀

Tensor Cores



Les *CUDA Core* sont spécialisés pour le calcul vectoriel.

Les *Tensor Core* sont spécialisés pour le **calcul matriciel.**

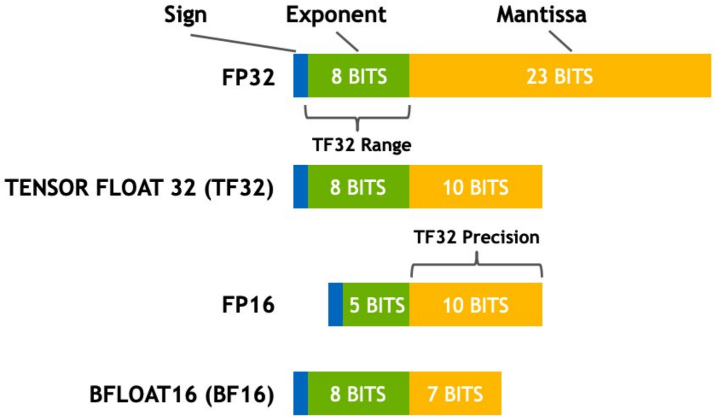
$$\underset{\text{FP16 or FP32}}{D} = \underset{\text{FP16}}{\begin{pmatrix} A_{0,0} & A_{0,1} & A_{0,2} & A_{0,3} \\ A_{1,0} & A_{1,1} & A_{1,2} & A_{1,3} \\ A_{2,0} & A_{2,1} & A_{2,2} & A_{2,3} \\ A_{3,0} & A_{3,1} & A_{3,2} & A_{3,3} \end{pmatrix}} \underset{\text{FP16}}{\begin{pmatrix} B_{0,0} & B_{0,1} & B_{0,2} & B_{0,3} \\ B_{1,0} & B_{1,1} & B_{1,2} & B_{1,3} \\ B_{2,0} & B_{2,1} & B_{2,2} & B_{2,3} \\ B_{3,0} & B_{3,1} & B_{3,2} & B_{3,3} \end{pmatrix}} + \underset{\text{FP16 or FP32}}{\begin{pmatrix} C_{0,0} & C_{0,1} & C_{0,2} & C_{0,3} \\ C_{1,0} & C_{1,1} & C_{1,2} & C_{1,3} \\ C_{2,0} & C_{2,1} & C_{2,2} & C_{2,3} \\ C_{3,0} & C_{3,1} & C_{3,2} & C_{3,3} \end{pmatrix}}$$

Chaque *Tensor Core* est capable de traiter 64 opérations en 1 temps d’horloge.

Source : [NVidia]

Précisions & Tensor Cores

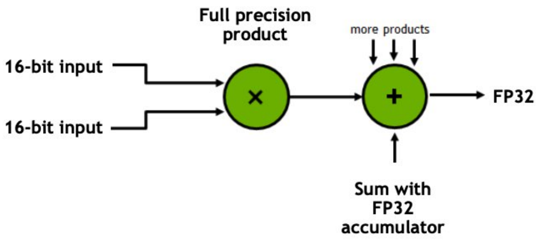
	NVIDIA H100	NVIDIA A100	NVIDIA Volta
Supported Tensor Core Precisions	**FP8**, FP64, TF32, bfloat16, FP16, ...	FP64, TF32, bfloat16, FP16, INT8, INT4, INT1	FP16
Supported CUDA® Core Precisions	FP64, FP32, FP16, bfloat16, INT8	FP64, FP32, FP16, bfloat16, INT8	FP64, FP32, FP16, INT8

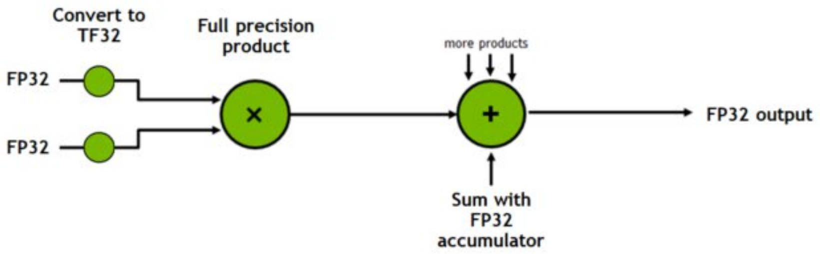


Source : NVidia

Précisions & Tensor Cores

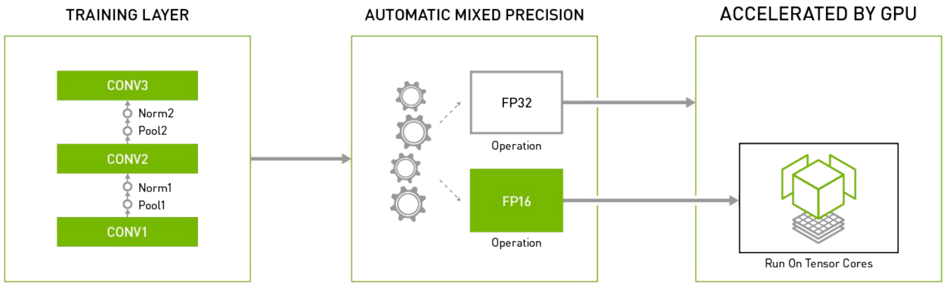
	INPUT OPERANDS	ACCUMULATOR	TOPS	X-factor vs. FFMA	SPARSE TOPS	SPARSE X-factor vs. FFMA
V100	FP32	FP32	15.7	1x	-	-
	FP16	FP32	125	8x	-	-
A100	FP32	FP32	19.5	1x	-	-
	TF32	FP32	156	8x	312	16x
	FP16	FP32	312	16x	624	32x
	BF16	FP32	312	16x	624	32x
	FP16	FP16	312	16x	624	32x
	INT8	INT32	624	32x	1248	64x
	INT4	INT32	1248	64x	2496	128x
	BINARY	INT32	4992	256x	-	-
	IEEE FP64		19.5	1x	-	-

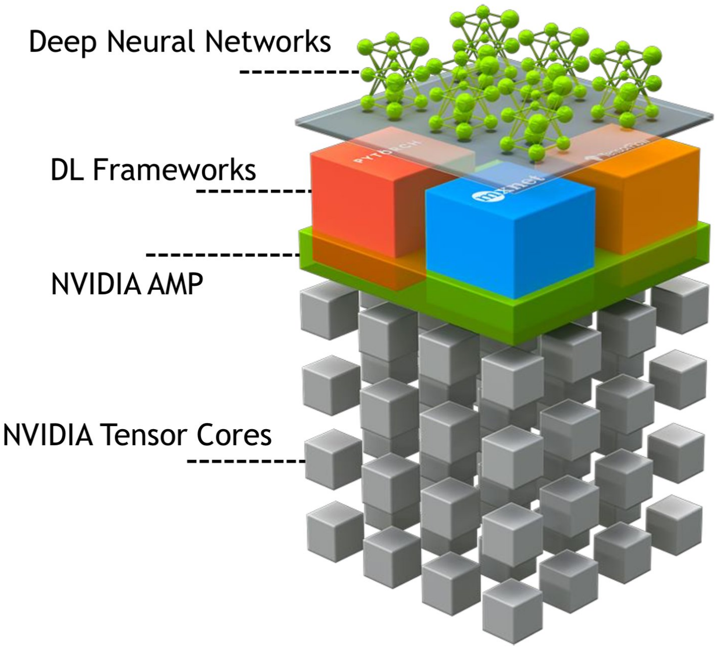




Automatic Mixed Precision

- Automatic Mixed Precision :
  - Nécessaire pour les V100 pour utiliser les *Tensor Core*
  - Les A100 utilisent les *Tensor Core* avec ou sans MP



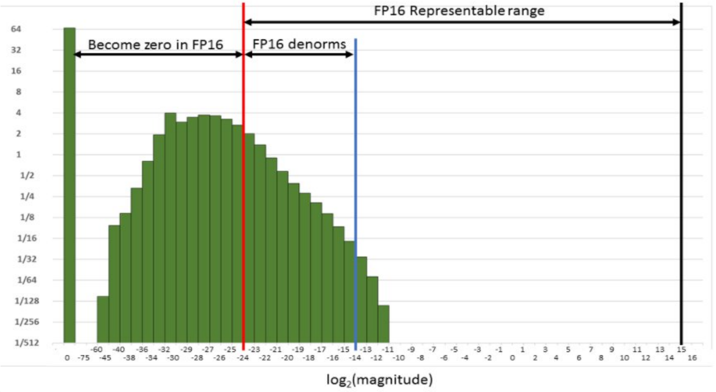


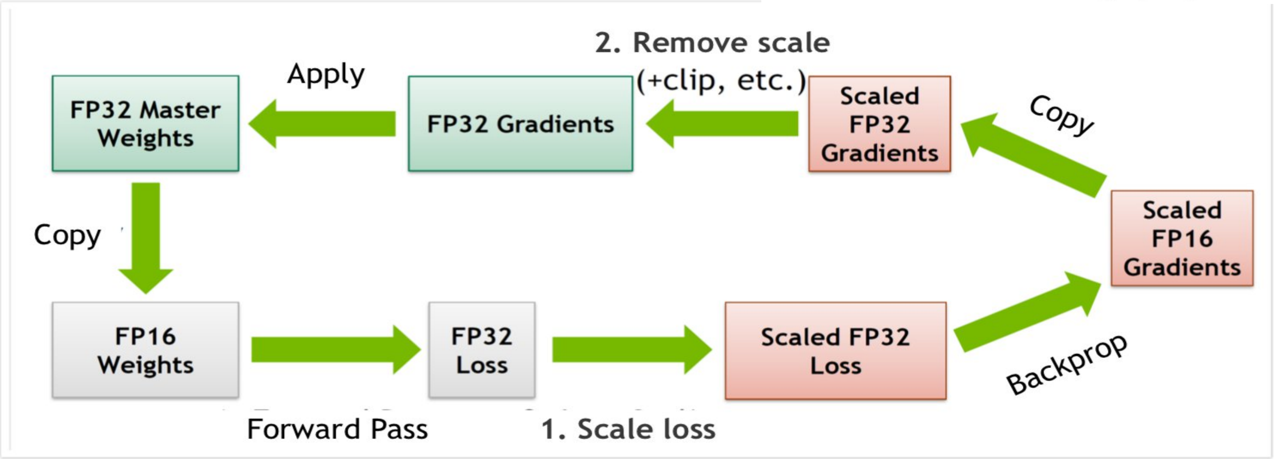
- Intérêts :
  - **Perte de précision non significative** pour l'apprentissage du modèle (gradient, loss, accuracy)
  - **Réduit** l'empreinte mémoire
  - **Accélère** les calculs
- 2 étapes à coder:
  - transformation des couches éligibles en FP16
  - Utilise un *scaling* pour le calcul des gradients

AMP Scaler

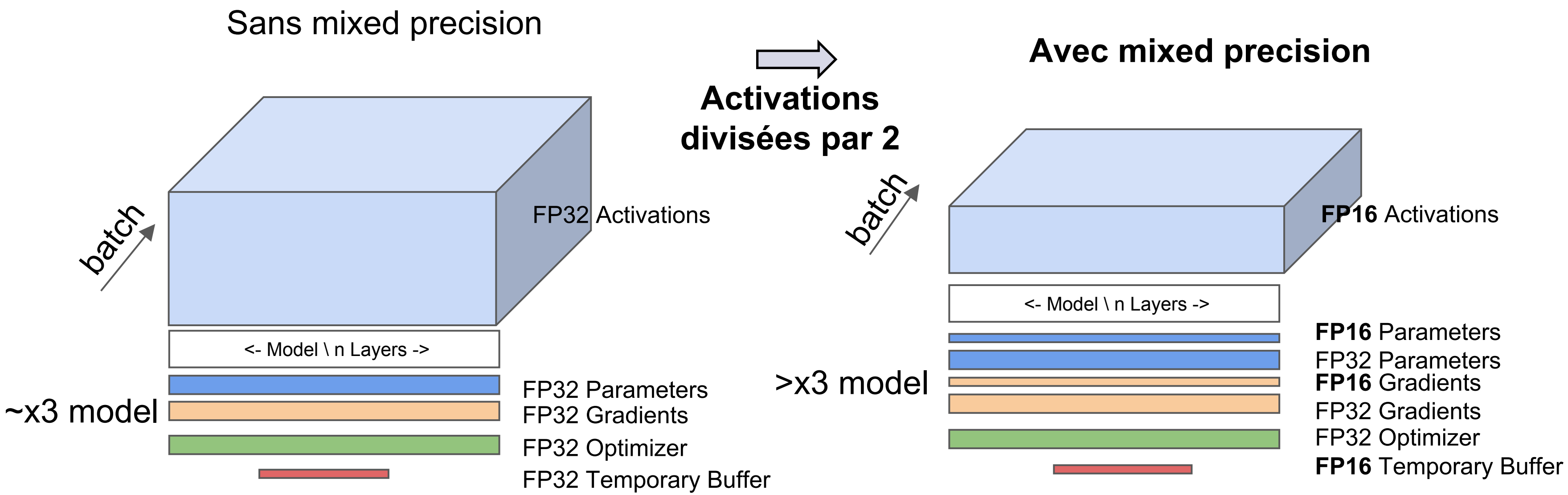
En FP16 les valeurs inférieures à 2^{-24} ($5.96e^{-8}$) sont considérées comme des 0.

Distribution des gradients





Empreinte mémoire avec la Mixed Precision



Channel last memory format

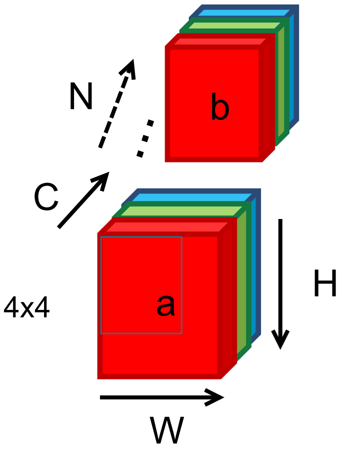
batch channel height width

NCHW

.shape()

memory contiguity by default

classic (contiguous) memory storage of NCHW tensor :



.stride()

b 0x: 0 1 2 3 4 5 6 7 8 9 a b c d e f 0 1 2 3 4 5 6 7 8 9 a b c d e f 0 1 2 3 4 5 6 7 8 9 a b c d e f

a 0x: 0 1 2 3 4 5 6 7 8 9 a b c d e f 0 1 2 3 4 5 6 7 8 9 a b c d e f 0 1 2 3 4 5 6 7 8 9 a b c d e f

Channels last memory format orders data differently:

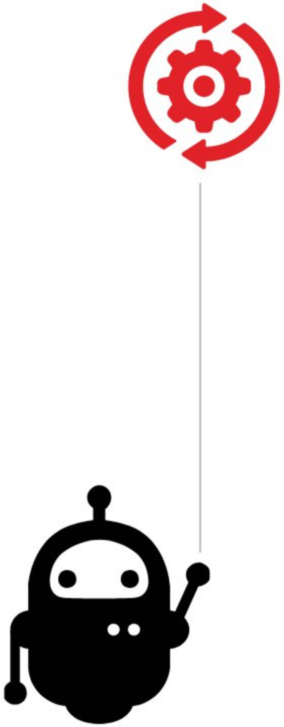
.stride()

b 0x: 0 0 0 1 1 1 2 2 2 3 3 3 4 4 4 5 5 5 6 6 6 7 7 7 8 8 8 9 9 9 a a a b b b c c c d d d e e e f f f

a 0x: 0 0 0 1 1 1 2 2 2 3 3 3 4 4 4 5 5 5 6 6 6 7 7 7 8 8 8 9 9 9 a a a b b b c c c d d d e e e f f f

3x3 Convolution filter

TP2&3 : Automatic Mixed Precision



- Activer l'Automatic Mixed Precision
- Test Mémoire
- Activer le channel last memory format