



Deep Learning Optimisé - Jean Zay

Introduction – Jean Zay – GPU



DLO-JZ

1st part of the IDRIS course.

Commented slides.

Authors : Bertrand Cabot, Myriam Peyrounette

Juin 2023

Chapters:

- Introduction to DLO-JZ
- Jean Zay
- The challenges of scalability
- GPU Computing
- Tensor Cores

Présentation de DLO-JZ

Plan ◀

Imagenet / Resnet-50 ◀

Présentation des participants ◀

2

The purpose of this introductory section is to describe the topics to be covered during the course, the progression of the practical exercises, and to initiate a short discussion between the participants.

The goal of this course is to present good practices in optimizing a Deep Learning loop on a supercomputer, particularly on Jean Zay. We will look at the system aspects and algorithmic techniques for accelerating a training.

The subjects addressed revolve around acceleration of training and the resulting memory footprint.

This training will not cover the state-of-the-art in Deep Learning and its different applications. We consider that you already have the necessary knowledge of the different Deep Learning techniques. Nevertheless, we will present certain interesting sources of information and you may engage in discussions during the free times.

Any questions about the content of this course may be asked throughout its duration. However, if the subject is raised late in the presentation, we may need to postpone our response.

Présentation - Sujets traités

Jour 1

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

Jour 2

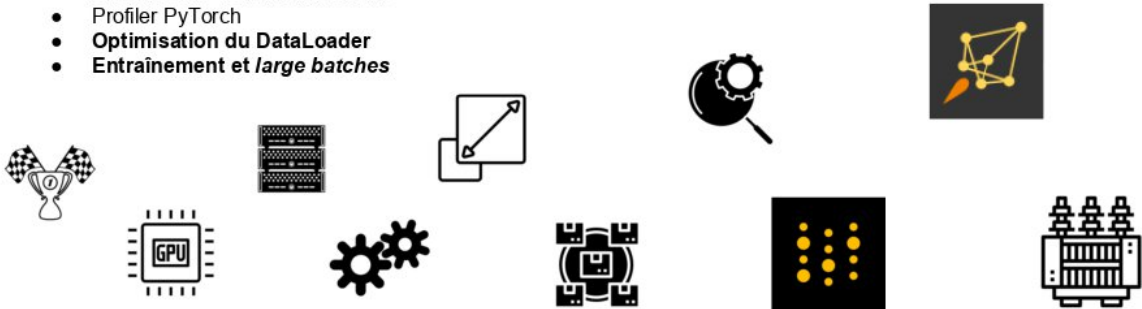
- **Distribution - Data Parallelism**
- Profiler PyTorch
- **Optimisation du DataLoader**
- **Entraînement et large batches**

Jour 3

- Résultats sur Weight & Biases
- Data Augmentation
- Stockage et format de données
- *HyperParameter Optimization*

Jour 4

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



The 5 principal subjects covered during this presentation are:

- GPU computing
- Distribution on multiple GPUs in Data Parallelism
- Optimizing data preprocessing (DataLoader)
- Optimizers and their parameterization with large batches
- Model parallelisms for very large models

The first day, we will focus on problems related to the system.

The second and third days, we will look at the algorithmic optimization techniques linked to Data Parallelism and to the induced usage of large batches. The fourth day will center on parallelism solutions for very large models (more than 1 billion parameters).

We will also look at visualization tools:

- The PyTorch Profiler for the system aspects
- Weights & Biases for the training results

Imagenet Race - Déroulé des TP



- Les TP des jours 1, 2 et 3 :
 - Optimisations système : GPU, Mixed Precision, Data Parallelism
 - DataLoader
 - Profiler
 - Data Augmentation
 - *Optimizers* et hyperparamètres avec des larges *batches*
 - Course de *job* sur 32 GPU pendant les nuits



- Les TP du Jour 4 :
 - Résultats de la course : Meilleur *Top-1 validation accuracy*
 - *Model parallelisms* avec un modèle CoAtNet-7 (gros Vision Transformer SOTA)



- Mini Jean Zay réservé : 32 GPU V100 sur 8 noeuds



The Practice Exercises will be done exclusively in PyTorch. This choice was made because PyTorch is perfectly optimized for the Jean Zay NVIDIA GPUs.

The prerequisites for the Practice Exercises are as follows:

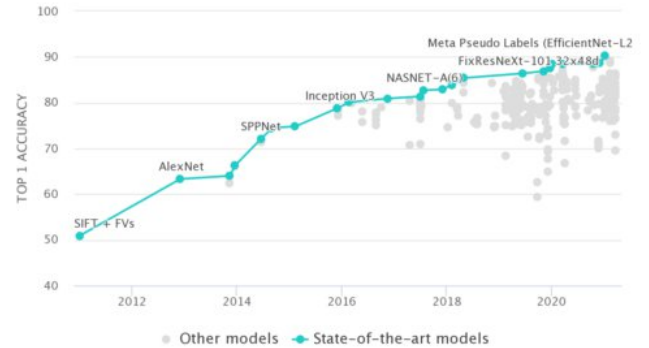
- Mastery of a Python code
- Knowledge of the principles of a training in Deep Learning
- The usage of PyTorch.

Données - Imagenet

But:
Classification (1000 classes)

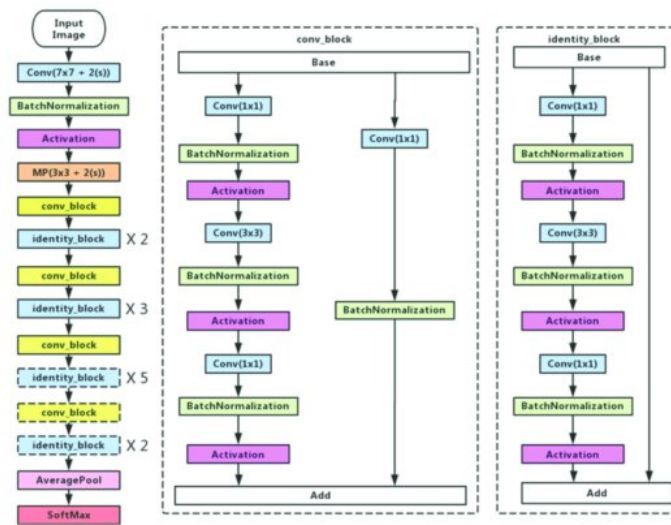


Dataset:
Train dataset: 1,2 Millions d'images labellisées
Validation dataset: 50 000 images labellisées
<http://www.image-net.org/>



To illustrate the themes which interest us, we decided to stay in a simple image classification application using the famous Imagenet database which was the cutting edge for all Deep Learning evolution, at least until 2018. This choice was also made because we have a representative example of training on a supercomputer, enabling us to stay within a reasonable training time while monopolizing a reasonable part of the system.

Imagenet - Resnet-50



Resnet :

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

6

We will use the ResNet model on ImageNet for most of this course (except for the last part on very large models and model parallelism).

ResNet-50 is a ResNet model with a depth of 50 layers.

Resnet is a CNN model of reference. Nearly all the most recent and most efficient CNNs are based directly on ResNet architecture.

Resnet has notably introduced: Residual learning with the shortcut principle to improve and accelerate learning (solution to the vanishing gradient problem, ...), and the BatchNorm layers which improve regularization compared to the dropout.

Recent versions of ResNet add Average Pooling which makes the model independent of the input image sizes.

Imagenet - Resnet-50



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



14 days

NVIDIA M40 GPU

7

We must remember that before 2017, on a computer with a GPU, 14 days were required to train ResNet-50 on ImageNet.

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

8

Since 2017 on supercomputers, ImageNet on Resnet-50 trains in less than an hour but it is necessary to involve a large part of the machine.

Présentation des participant·e·s



9

Once the objectives of this course have been clarified, we will invite the participants to introduce themselves and express their expectations for the course.

Jean Zay

Supercalculateur ◀

Jean Zay ◀

Soumission de jobs ◀

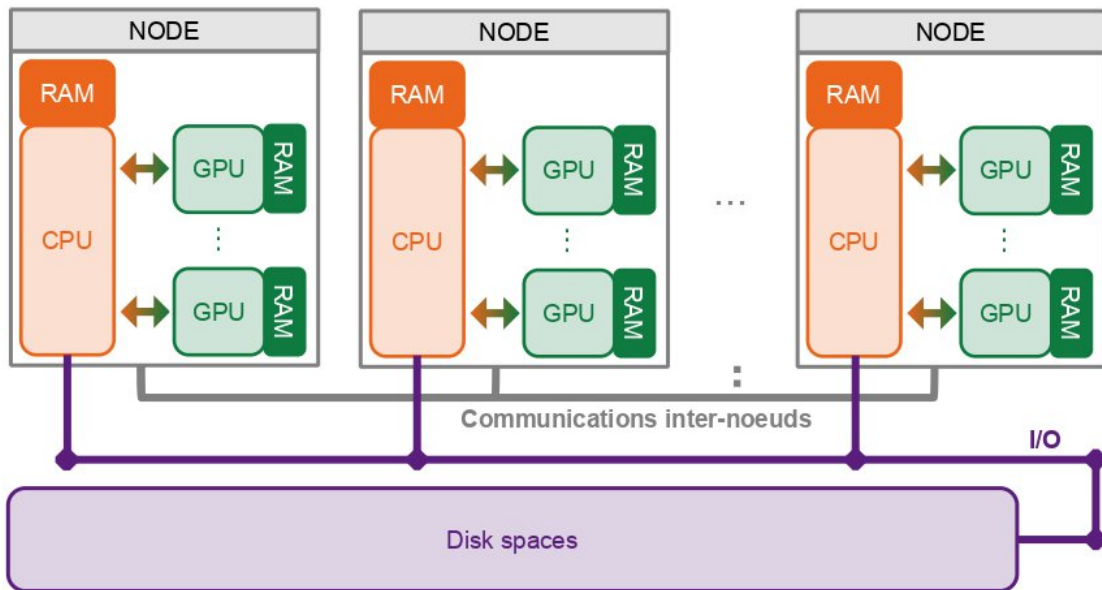
JupyterHub sur Jean Zay ◀

Outils Slurm pour notebook python ◀

10

This section is dedicated to the Jean Zay supercomputer and will describe the computing environment of the hands-on exercises.

C'est quoi un supercalculateur ?



11

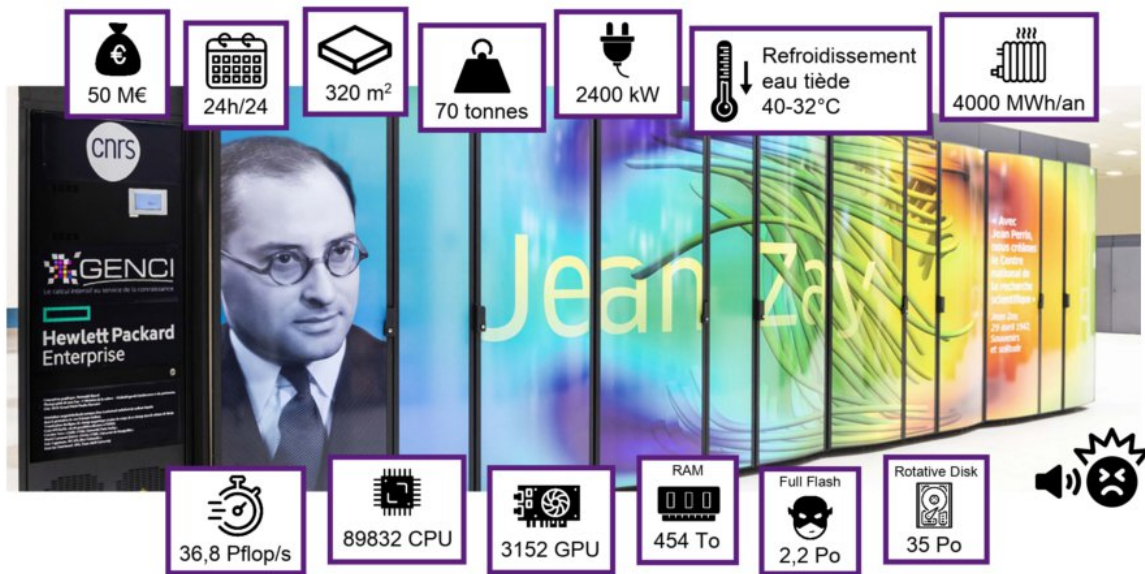
A supercomputer is an aggregation of compute nodes. Each node contains CPU cores accelerated by GPU cards, and associated RAMs.

All the supercomputer nodes together can be considered as a unique machine with aggregated performance on condition that a high performance communication network connects all the nodes (Omni-Path on Jean Zay).

The storage spaces are external to the compute nodes. The files are handled by a high performance parallel file system (Spectrum Scale , ex-GPFS on Jean Zay).

Jean Zay

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



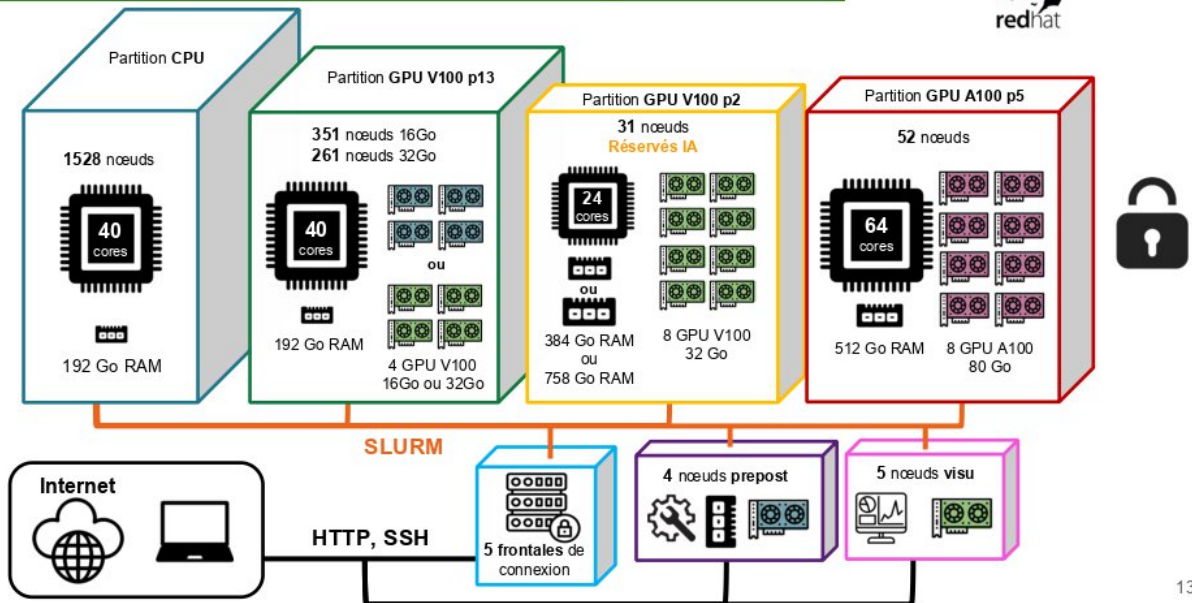
12

Jean Zay is the first converged supercomputer for Artificial Intelligence (AI) and High Performance Computing (HPC) in France.

A grand scale infrastructure is necessary to host this equipment dedicated to the needs of modern research.

The entire infrastructure requires management by a considerable number of human resources: administrative personnel and engineers (~40 employees at IDRIS).

Jean Zay : Ressources disponibles



13

Jean Zay users can reserve compute resources on 4 different partitions having different configurations:

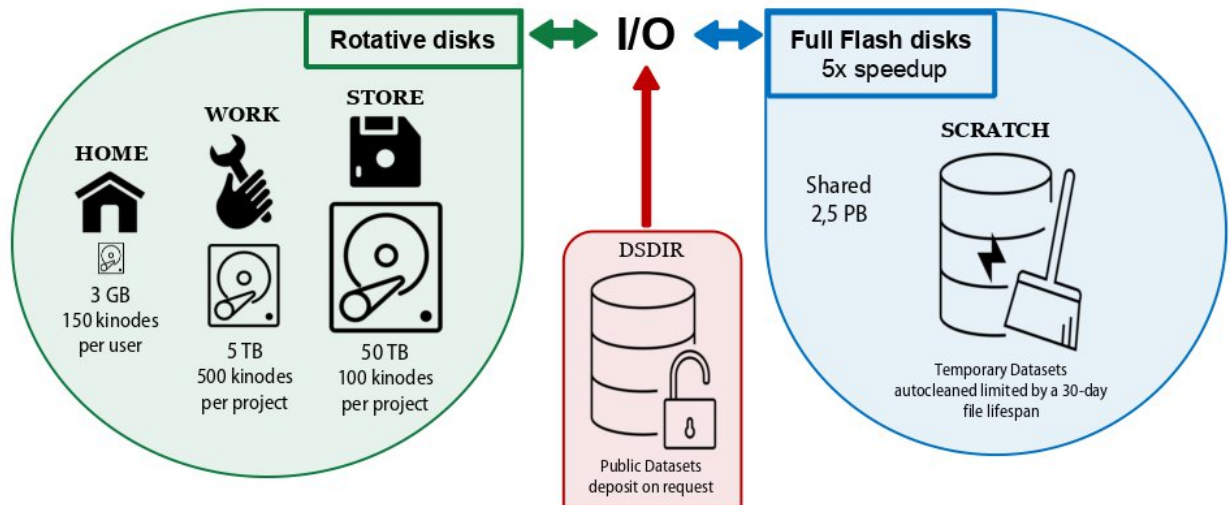
- A CPU partition
- An accelerated V100 quadri-GPU partition
- An accelerated V100 octo-GPU partition exclusively dedicated to AI
- An accelerated A100 octo-GPU partition

Access to the machine is through 5 front ends which are open to the exterior. Compute nodes are accessed from these front ends through the Slurm job scheduler. The compute nodes are isolated from the exterior.

Two dedicated partitions which are free of hourly computing charges and open to the exterior are also in the system:

- A “prepost” partition for all pre/postprocessing work
- A partition dedicated to visualization

Jean Zay : Espaces de stockage



14

Various storage disk spaces are available.

- Three spaces on rotative disks:
 - The HOME to store configuration files of the computing environment
 - The WORK to store code, data, logs, etc
 - The STORE to archive data
- Public datasets and models are available on the DSDIR space and accessible by all users
- The SCRATCH equipped with a rapid Full Flash technology for I/O-intensive jobs




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:~$
```

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



- Personnalisables

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

 **Conflits entre les versions**
Saturation de vos espaces disques

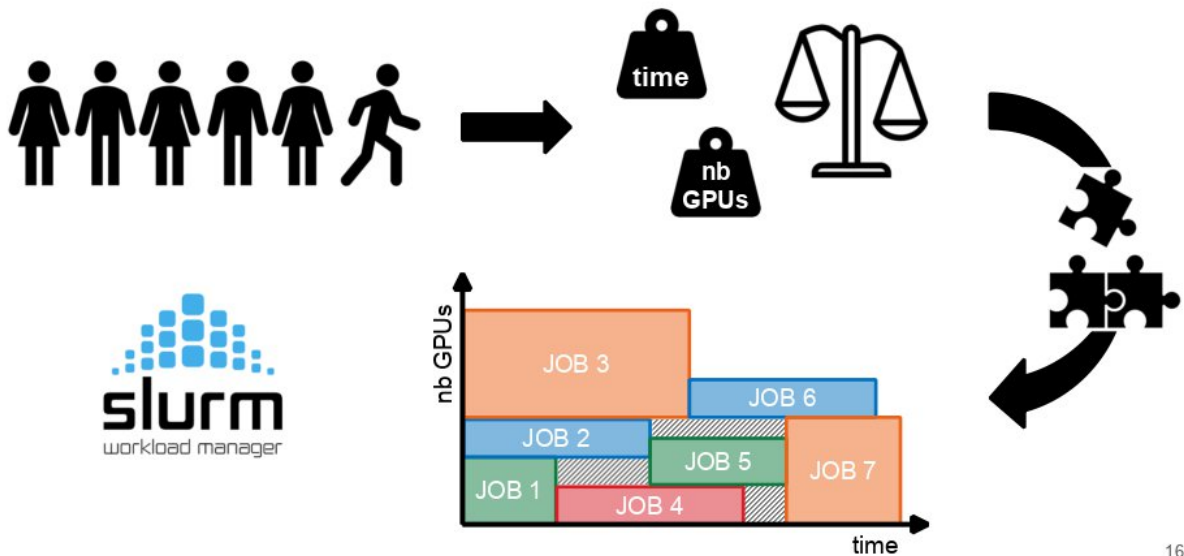
Users have a catalogue of predefined environments at their disposal. These environments are installed on Jean Zay by IDRIS. For AI users, they correspond to conda environments which are built around major libraries such as PyTorch, TensorFlow or MXNet.

Computing environments are handled by the *Environment Modules v4* package.

Users just need to load the associated module to activate the desired conda environment. They are free to build or enrich their own environments. However, we recommend that you request IDRIS to directly enrich the catalogue environments to pool the resources and avoid saturating your project disk spaces.

Users can also import an environment on Jean Zay via Singularity containers.

Soumission de jobs - Slurm



16

The distribution of the computing resources is managed by the job scheduler Slurm.

Jobs submitted by users are sent to a wait queue and Slurm plan their executions according to the requested resources (number of GPUs and execution time).

Slurm takes into account other factors such as project under/over-consumption and requested QoS.

Slurm's goal is to minimize the idle time of the computing resources, while distributing resources among users as fairly as possible.

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/py3/1.11.0 # environment

srun python script.py           # run script
```

17

To submit a job via Slurm, users need to create a batch script defining all the necessary parameters to reserve a computing slot, and the commands they want to be executed on these resources.

To launch a parallel execution on multiple tasks (or processes), the execution command of the python script must be preceded by a call to the parallel Slurm launcher `srun`.

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/py3/1.11.0

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 login PD 0:00 2 (Priority)
```

↓ Lancement du job

```
srun python script.py
```



18

Once the batch script is written, you can submit it thanks to the `sbatch` command.

Slurm will send the job in the wait queue. The jobs currently handled by Slurm can be listed with the `squeue` command.

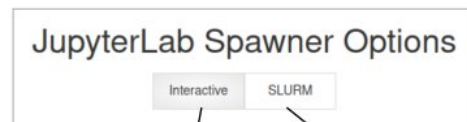
Once the computing resources are available, the execution is automatically launched by Slurm on all the requested tasks (thanks to the parallel launcher `srun` called in the batch script).

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)



19

You can access Jean Zay computing resources through a JupyterHub platform. This platform has been developed by IDRIS engineers to comply with the security constraints of the center.

<https://jupyterhub.idris.fr/>

You can run a Jupyter instance:

- Either on a front end, to use it like an augmented shell enabling job submission on the compute nodes, data preprocessing, visualization of results,... This front end has limited resources and is not equipped with GPUs.
- Or directly on a compute node, in order to run tests in interactive mode. In this case, access to the internet is lost. This configuration is restricted to one node.

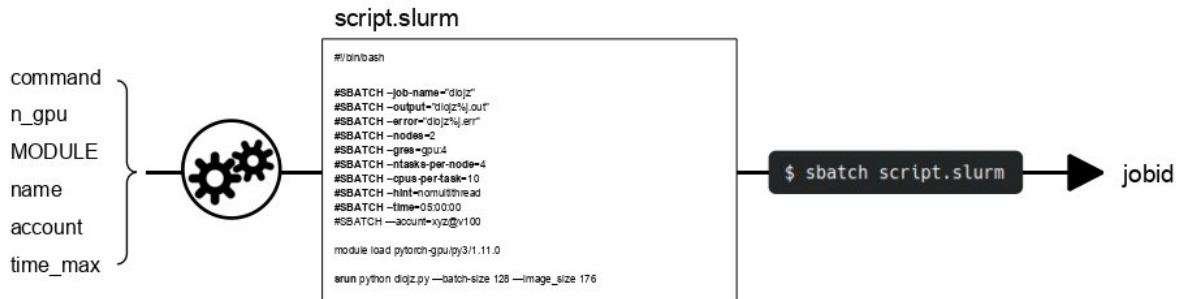
We will launch a Jupyter instance from a front end during the practice exercises of this training.

Jean Zay : Outils Slurm pour notebook python

```
from idr_pytools import gpu_jobs_submitter
```

```
command = 'dlojz.py --batch-size 128 --image_size 176'  
n_gpu = 8  
MODULE = 'pytorch-gpu/py3/1.11.0'  
name = 'dlojz'
```

```
jobid = gpu_jobs_submitter(command, n_gpu, MODULE, name=name, account='xyz@v100', time_max='05:00:00')
```



20

IDRIS developed a python library called `idr_pytools` to ease the interaction with Slurm from a python notebook.

The `gpu_jobs_submitter` function allows you to easily submit jobs.

From a reduced number of instructions (the command to execute, the number of GPUs, the module to load, the name you want to give to your job,...), `gpu_jobs_submitter` will write the corresponding batch script for you and submit it.

This function will be used during the practice exercises of this training.

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]
```

nom du fichier *output*

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

nom du fichier *error*

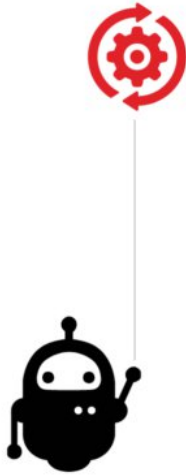
21

Moreover, the `idr_pytools` library allows you to visualize the Slurm wait queue from a python notebook thanks to the `display_slurm_queue` function.

The log files generated during the execution of a job can be displayed thanks to the `search_log` function.

This function will be used during the practice excercises of this training.

TP0 : Préparation de l'environnement



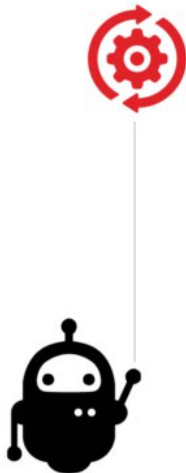
- Lancer un terminal et faire les copies nécessaires

```
local:~$ ssh jean-zay  
jz:~$ cd $WORK  
jz:~$ cp -r $ALL_CCFRWORK/DLO-JZ .
```

- Lancer firefox
- Accéder à jupyterhub.idris.fr

20

TP0 : Accès et prise en main du notebook



- Ouvrir le notebook DLO-JZ_Jour1.ipynb
- Choisir le kernel pytorch-gpu/py3/1.11.0 (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

22

Les enjeux de la montée à l'échelle

Temps d'apprentissage ◀

Empreinte Mémoire ◀

Solutions ◀

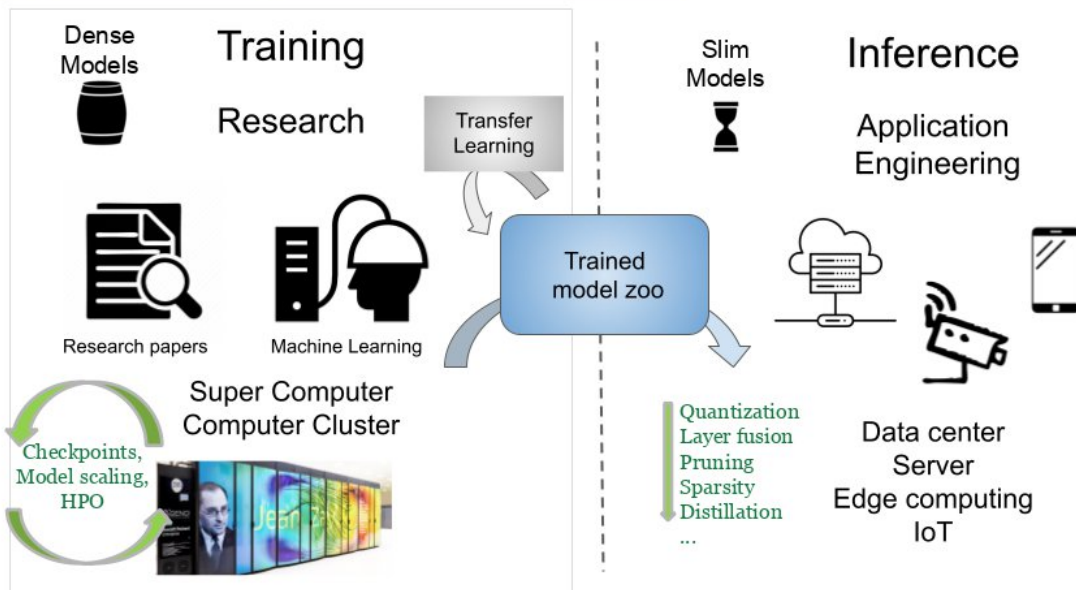
Economie énergétique ◀

23

The objective of this section is to identify the problems of Deep Learning on a supercomputer and to rapidly list the possible available solutions .

When we speak of memory footprint, this refers to the RAM occupation induced by the process.

Apprentissage / Inférence



24

There are two distinct fields in Deep Learning engineering: training and inference. The challenges of the two fields are not the same.

Training is tending towards larger and larger models trained on large computing resources. This represents the core of AI research. It is a long iterative process, with the search for optimal solutions through replicated trials.

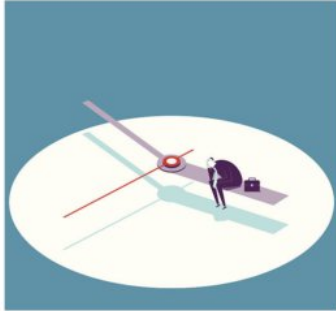
Inference seeks to minimize the model sizes to be able to deploy them on various equipment with techniques such as Quantization, Layer fusion, Pruning, Distillation, ... Inference is a direct process which is meant to be as rapid as possible.

This DLO-JZ course is only interested in problems in the training field because this is what is principally done on Jean Zay.

Contraintes du Deep Learning

2 problèmes à traiter:

Temps d'apprentissage

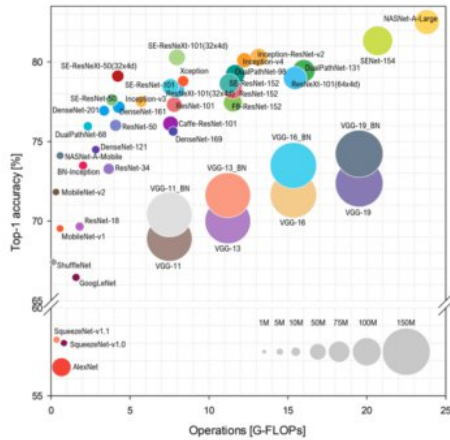


Surconsommation mémoire (OOM)



Les Gros Modèles

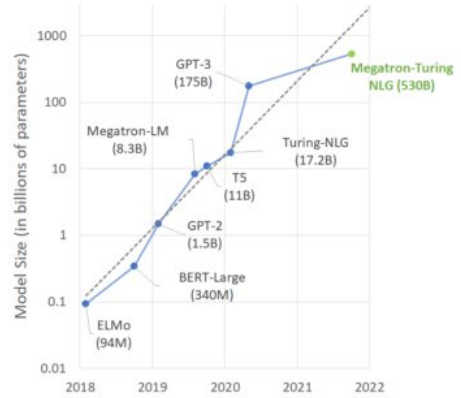
Convolutional Neural Network



Les modèles gros, et profonds permettent d'obtenir de meilleures 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







Larger and larger models in terms of parameters, number of layers and mathematical operations are invading the world of Deep Learning. They enable obtaining higher and higher accuracy and training functions which are more and more complex.

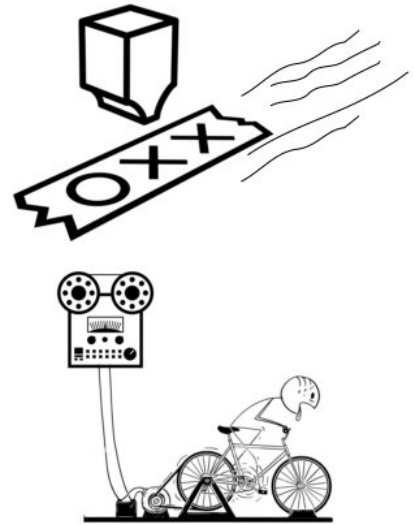
Huge models such as the Transformers pushing the limits of neural networks provoke costly computing times and large memory footprints.

A model with 1 billion parameters represents a variable of more than 4GB, knowing that a parameter saved in float32 is represented by 4 bytes.

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 

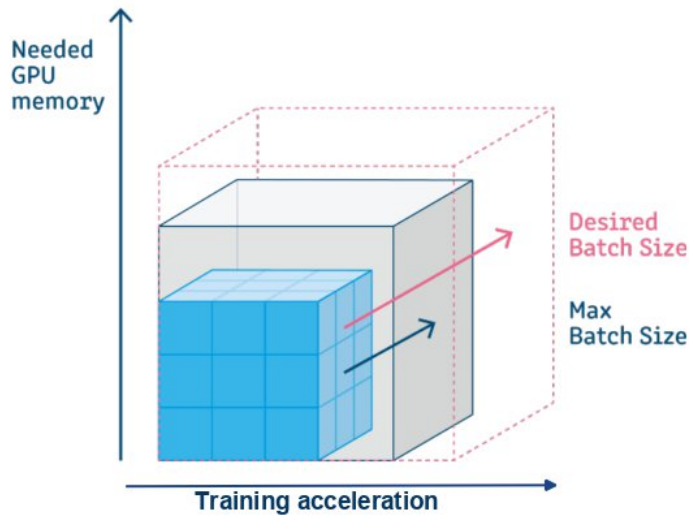


27

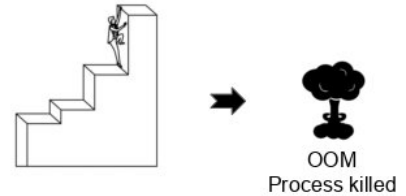
The number of FLOPs representing the number of necessary calculations depends on the size of the model, the depth of the model, and the size of the input data.

The larger the model, the more it is complex, the larger the necessary size of the dataset and the more consequential the number of epochs to reach the end of the training: This increases the duration of the computing time even more.

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.

28

A simple solution to accelerate the training, considering that a fixed number of epochs is sufficient to reach the end of the training, would be to increase the batch size and thereby, to augment the iteration step.

However, we will see that this is not so simple.

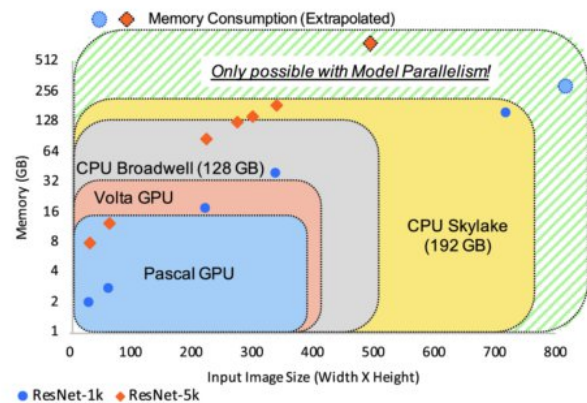
The first constraint is that this equally increases the memory footprint which will rapidly reach the limit of the available RAM.

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](#)

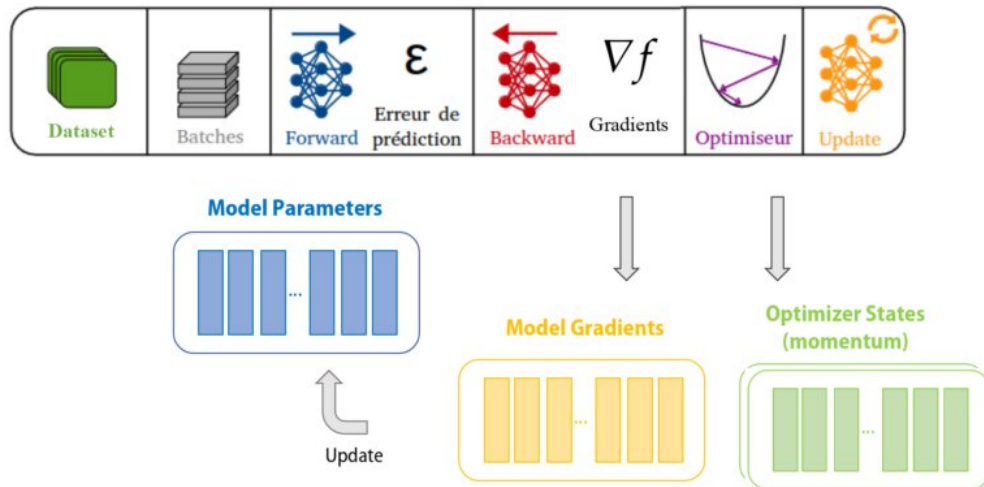
29

The format of the data processed by the model plays an important role in the memory occupation during the training and is greatly accentuated by the model depth.

With 3D images or with very deep models, it is very easy to saturate the memory space available on the equipment.

We will find the explanation for this phenomenon in the following slides.

Forward / Backward – mémoire du modèle



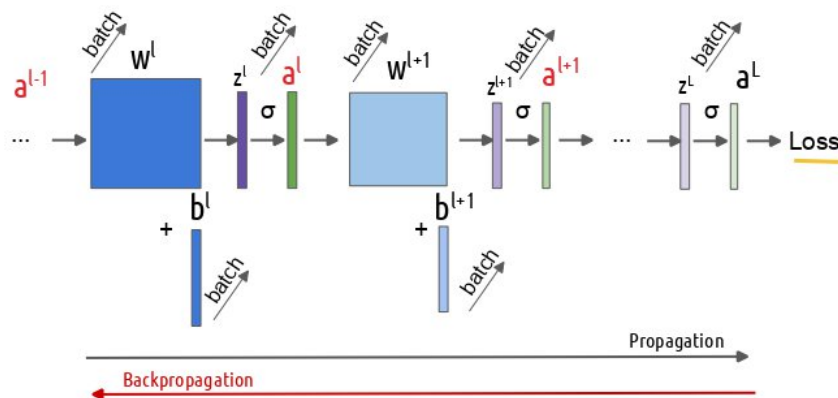
44

A training loop is composed of: loading and preprocessing the data per batch, forwarding, computing the loss, backwarding, computing the gradients, calling the optimizer and updating the weights.

During this training loop, the algorithm needs to keep in memory all the weights of the model, a replica of the gradient of each weight, a replica of the optimizer for each momentum.

The memory footprint required to contain a given model is twice to four times the model size. However, for a model such as Resnet-50, this represents only hundreds of megabytes. So what is the memory constraint mentioned above?

Forward / Backward - problème des activations



Propagation

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

Backpropagation

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

$$\delta^L = \nabla_a C \odot \sigma'(z^L) \quad b^l \rightarrow b^l - \frac{\eta}{m} \cdot \frac{\partial C}{\partial b^l}$$

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

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

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

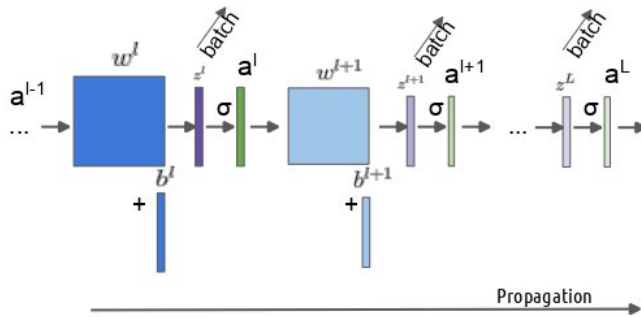
Note: Pour la *backpropagation*, il est nécessaire de garder en mémoire les **activations intermédiaires**.

30

In the training loop, during the gradient backpropagation calculation, the system needs the outputs of the activation function of the preceding layer.

Unless the same operations are recalculated numerous times which would produce a solution much too slowly, it is necessary to keep in memory each activation function output for each layer.

Inférence et évaluation



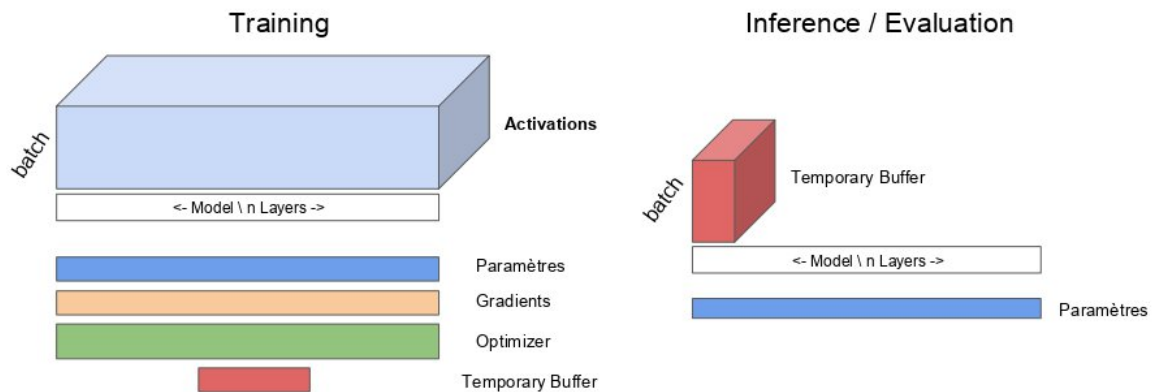
```
...  
with torch.no_grad():  
    val_outputs = model(val_images)  
    loss = criterion(val_outputs, val_labels)  
...
```

Propagation

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

For the inference and validation steps, there aren't any gradient calculations so it is not necessary to keep the intermediate activations. It is important, therefore, to indicate to the system to not allocate this needed memory for the backpropagation which then does not use its autograd mechanism.

Empreinte Mémoire

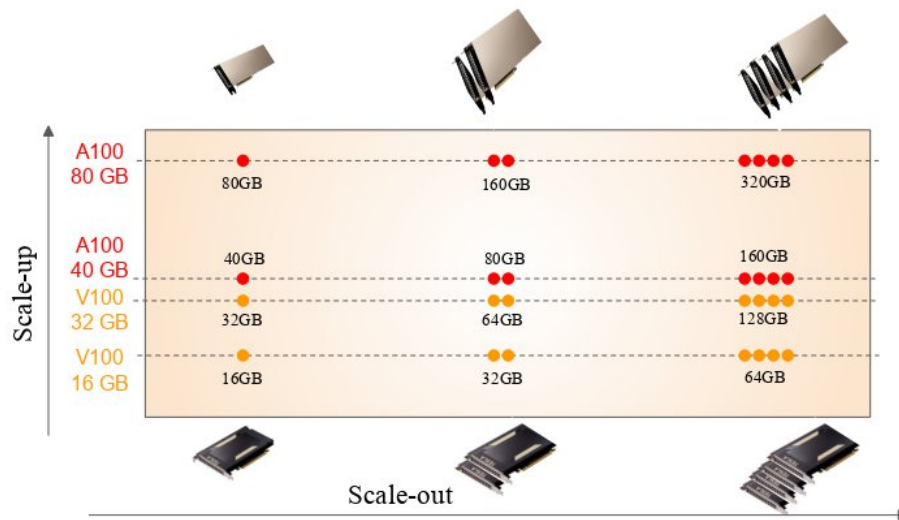


32

During the training step, the memory footprint divides into two parts:

- The part with intermediate activations dependent on the batch size, data format and neural network depth. This part can occupy a space which becomes rapidly too large if we augment one of these characteristics.
- The part directly linked to the storage of model weights and gradients which together represent several times the model size. This size, however, is often insignificant when compared to the system memory (except for very large models).

Solutions système

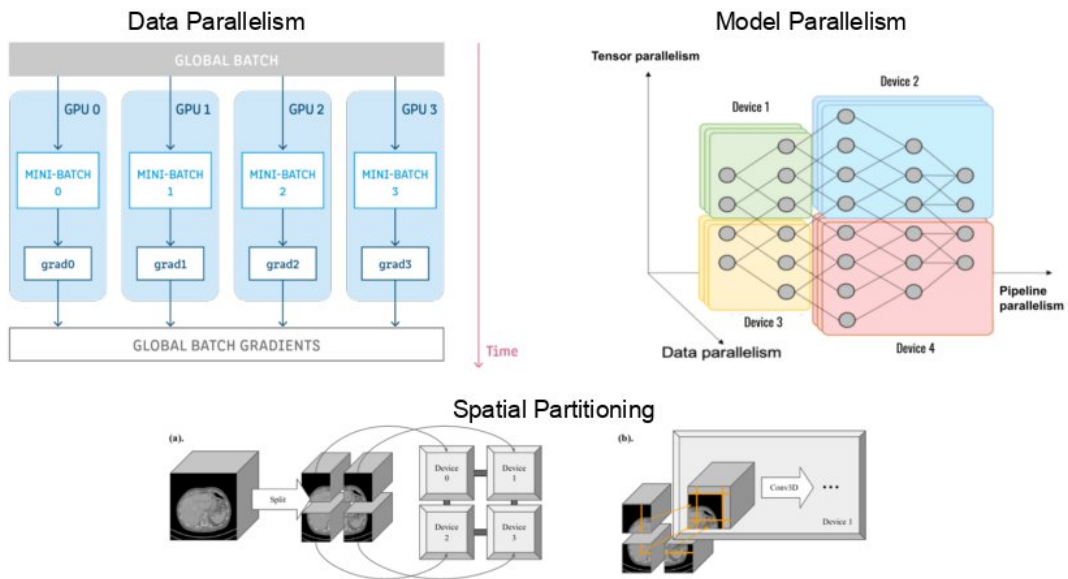


33

There are two possible scalability axes, from a hardware system point of view, to resolve the problems of acceleration and memory occupation:

- The Scale-up, or replacing equipment with upgraded versions; for example, going from V100s to A100s.
- The Scale-out, or the distribution of calculations on multiple equipment.

Solutions: Distribution – Scale-out



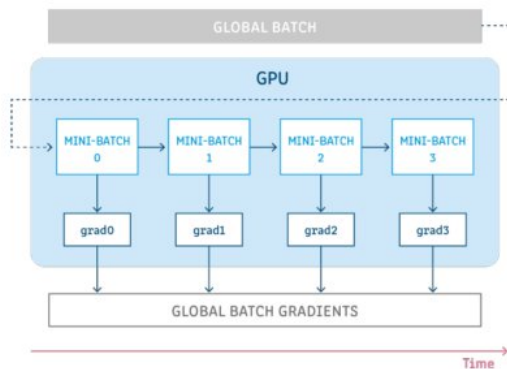
34

The distribution solutions in Deep Learning are as follows:

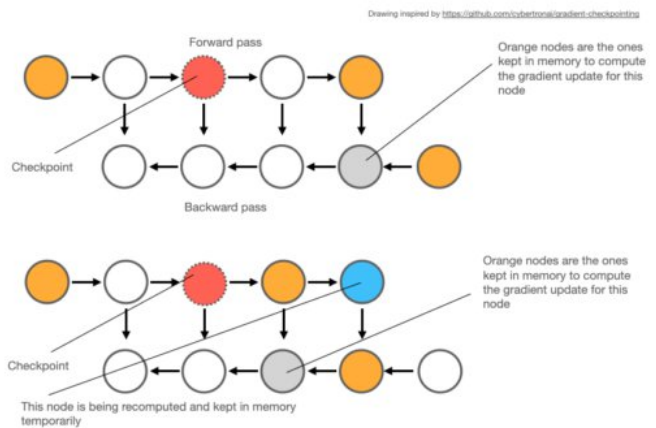
- Data Parallelism, we will discuss at length during this course.
- Model Parallelism, we will discuss at the end of the course.
- Spatial Partitioning, we will not address during this course (as very rarely used). It consists of directly dividing the input data and can be useful for very large data formats but would be costly to implement.

Solutions de contournement

Gradient aggregation



Gradient/activation checkpointing



35

There are also alternative solutions which are less efficient but enable saving memory space at the price of slower calculations:

- Gradient aggregation, which sequentially computes mini-batch gradients but waits for several iterations before updating the model. This is similar to Data Parallelism except that it is completely sequential and non-distributed.
- Gradient checkpointing, which does not save all the activations but only some of them (the checkpoints!!) and recomputes the missing activations from the last checkpoints.

Un 3e problème à traiter ...

La consommation électrique !!

2 problèmes à traiter:

Temps d'apprentissage



Surconsommation mémoire (OOM)

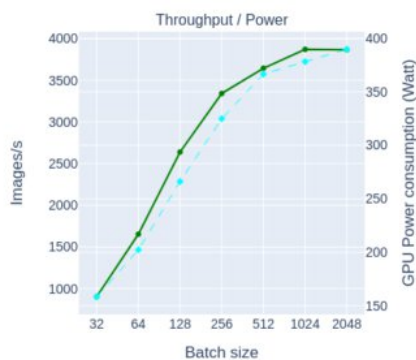


36

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.

37

Économie énergétique / Heures GPU

Économie énergétique
 \cong
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

38

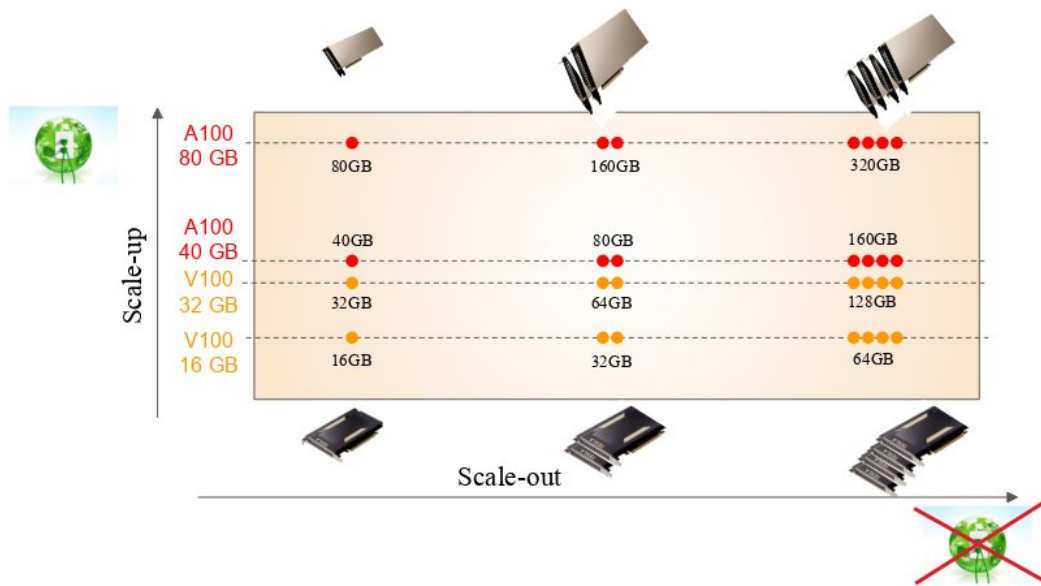
The manufacturers and hosts of systems such as Jean Zay develop complex techniques to reduce electricity consumption while keeping high performance.

As a user, the best way to save energy is to use GPU hours sparingly.

To do this, it is first necessary to apply the optimizations described in this course.

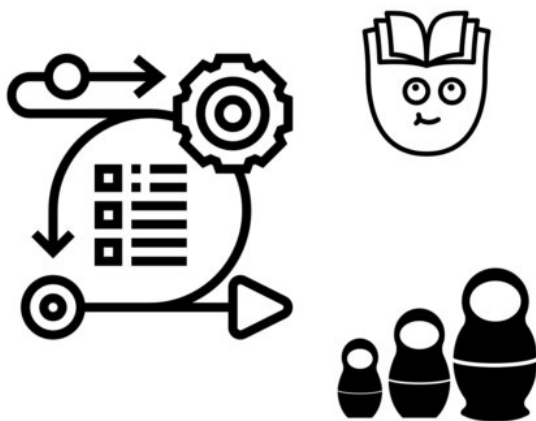
It is worth noticing that scaling up enables saving energy but scaling out does not since we increase the number of GPUs.

Économie énergétique / Heures GPU



39

Économie énergétique / Heures GPU



Méthodologie (économiser la recherche, ne répéter pas 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)

40

Also, to save GPU hours or reduce electrical consumption, we need to be methodical in our search for good hyperparameters for training our model, and not repeat unnecessary trials which are energy-intensive.

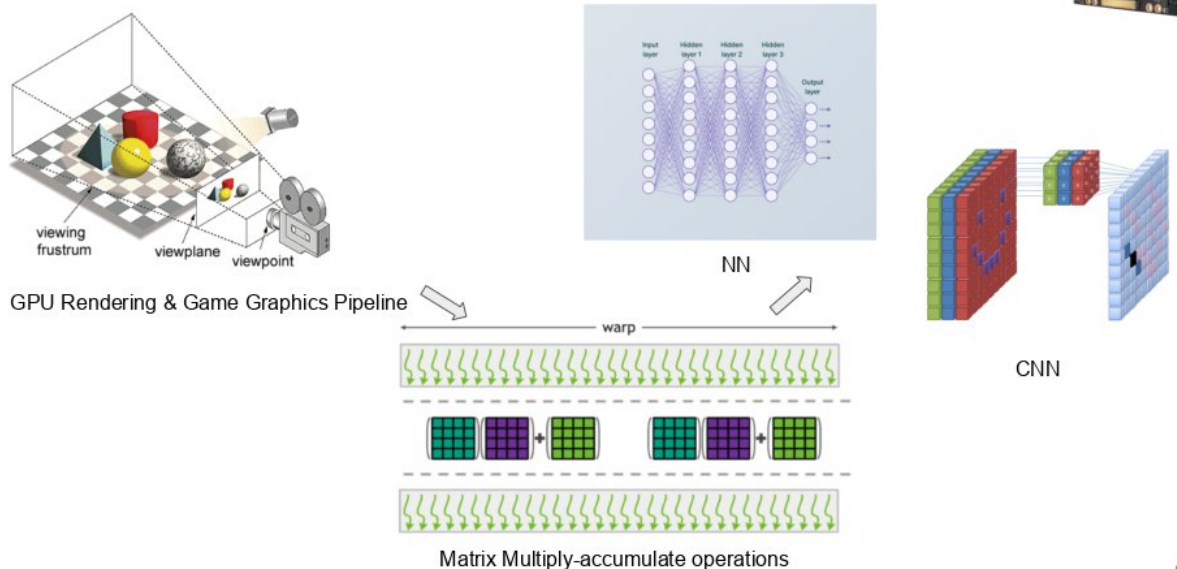
GPU computing

- V100, A100 ◀
- CUDA ◀
- CuDNN ◀
- AMP ◀

41

The objective of this section is to present GPU hardware accelerators which often have a much higher computing speed than CPUs but with less memory.

GPU computing

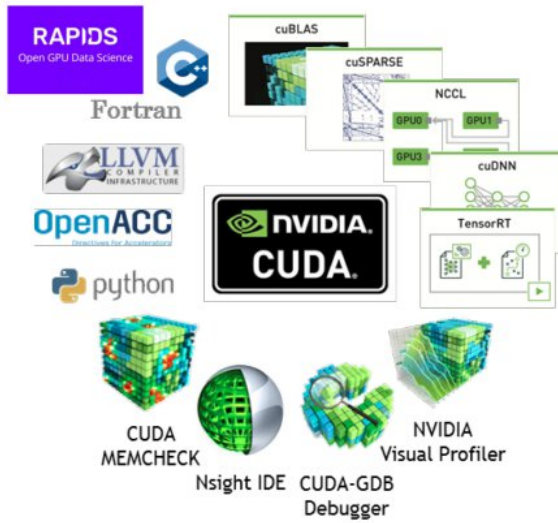


42

GPUs were initially used for the 3D rendering of video games, permitting multiplication computation and matrix accumulation organized by warp (multi-threads).

They are especially adapted for linear algebra and, therefore, for conventional neural networks and convolutional neural networks.

Galaxie NVIDIA



	DESKTOP		DATACENTER AND CLOUD		
INFERENCING AT THE EDGE					
	DGX Station	Titan V	DGX-2	DGX-1	Tesla V100
	AUTONOMOUS MACHINES		AI SELF-DRIVING PLATFORM		
	Jetson TX2	Jetson TX1	DRIVE Pegasus		
	NVIDIA DEEP LEARNING SDK and CUDA				

Source : [NVidia](https://www.nvidia.com)
43

NVIDIA is a manufacturer of Graphics Processing Units and produces accelerated systems for personal computers, data centers, supercomputers, on-board equipment and autonomous cars.

The success of NVIDIA also stems from the whole suite of available AI and HPV applications surrounding CUDA technology.

CuDNN

PyTorch



NVIDIA DEEP LEARNING SDK and CUDA



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

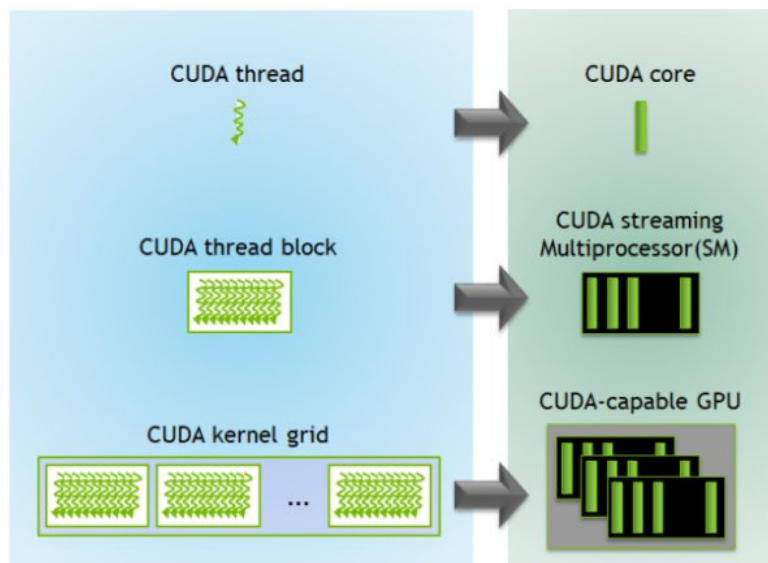
Recommandation: pour optimiser l'utilisation des *Tensor Cores* et des *Cuda Cores* : Utiliser des tenseurs aux dimensions (*batch size*, *sample size*, *channel*, etc ...) multiples de 8 !!

CuDNN is the CUDA application dedicated to neural networks. CuDNN is integrated into most of the Deep Learning frameworks such as PyTorch, TensorFlow and MXNET.

The complex engineering of CUDA is managed by CuDNN. The following slides explaining CUDA engineering can be ignored at our level as CuDNN enables managing this in a transparent way for the users.

Only the following recommendation should be taken into account: To optimize the computation on GPUs, you must size the tensors in multiples of 8 (batch size, data format, number of filters, etc, ...).

GPU computing : CUDA



Source : [NVIDIA](#)
45

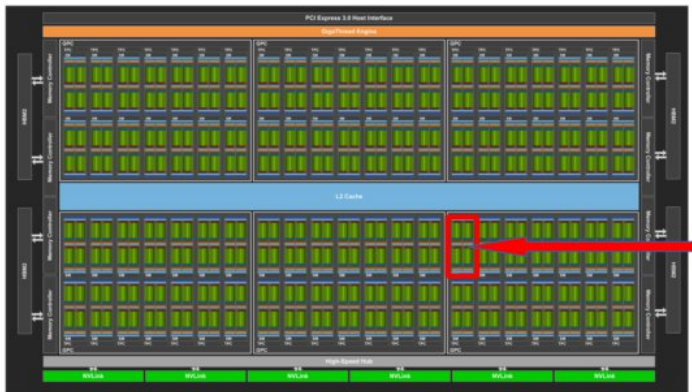
In CUDA language, a function or method, called kernel, is cut into threads. A thread is a simple operation (multiplication/addition of scalars).

Each thread is processed by a CUDA core.

A block of threads is processed by a Streaming Multiprocessor.

A grid block allocated to a kernel is processed by the GPU.

Architecture V100



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



Source : [NVIDIA](#)

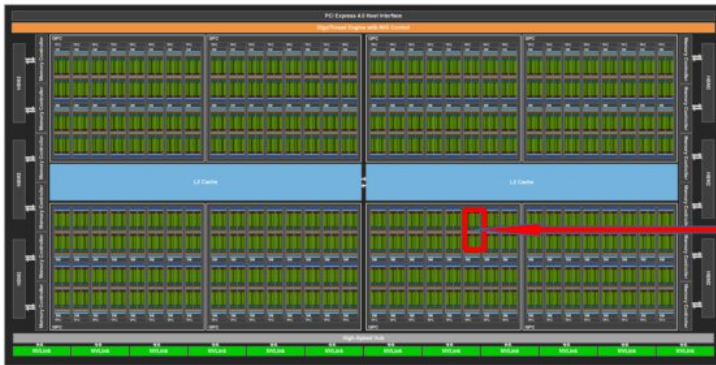
The V100 GPU architecture consists of 6 Graphics Processing Clusters (GPCs).

Each GPC contains a number of Streaming Multiprocessors (SMs) which process the thread blocks.

Each SM has 4 warp schedulers which contain a multiple of 8 CUDA cores for the integers, the float32s and float64s.

It is important to note the presence of Tensor cores which we will address in the following section about mixed precision.

Architecture A100

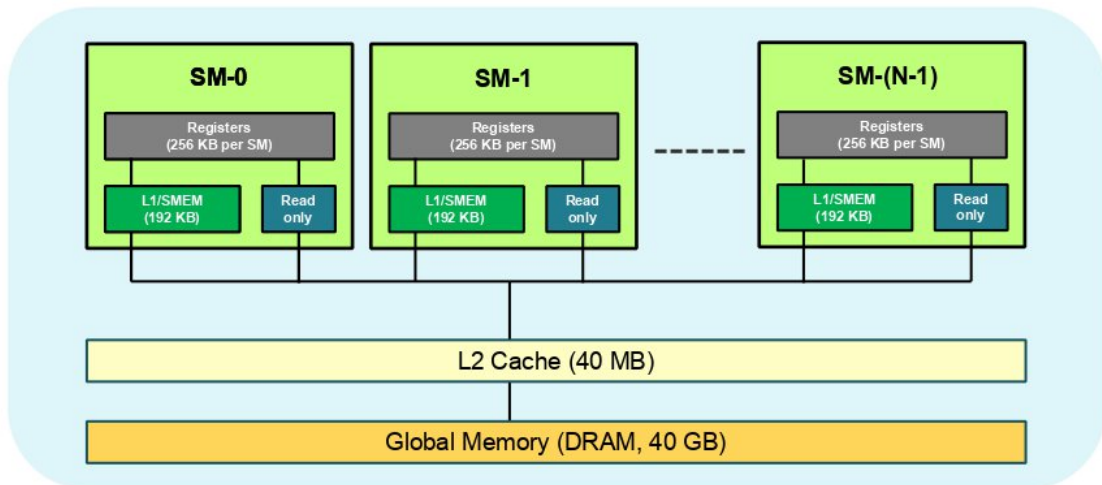


- 8 GPC
- 128 Streaming Multiprocessors (SMs)
- 8192 CUDA Cores
- 512 Tensor Cores per full GPU

Source : [NVIDIA](#)

The A100 GPU architecture is almost the same except for the presence of 8 Graphics Processing Clusters (GPCs) and, therefore, more CUDA cores on the whole of the GPUs.

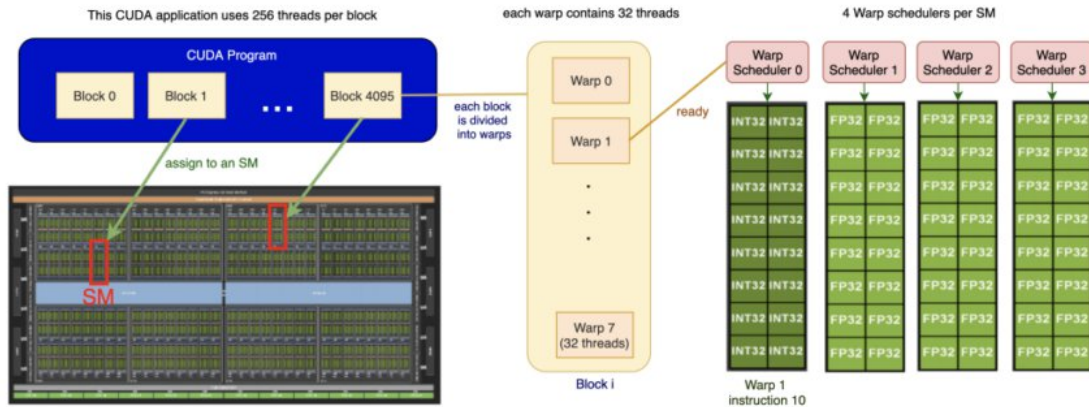
The Tensor cores are the generation which follows that of the V100 GPUs.



The memory to store the variables before or after the computation comprises a funnel-shaped architecture: The closer we get to the thread, the more the memories are small and rapid; inversely, the farther we get from the thread, the more the memories are large and slow.

The RAM is global to all the GPUs: The L2 level corresponds to the GPCs, the L1 level corresponds to the SMs, the L0 level corresponds to the warp schedulers.

From this we understand that the blocks must be regrouped locally on the GPUs as much as possible to economize the residual times of communications between its different memories.



Optimisation :

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

Optimisation avancée :

- Fusion des kernels pour économiser les temps d'initialisation

49

CUDA engineering consists of distributing the threads of a kernel into several blocks which will be divided into warps, and which will be executed in a subset of the GPU cores.

The challenge of CUDA, therefore, is to sufficiently fill each SM and spread all the threads on the whole kernel (as for filling an ice cube tray).

A substantial effort is also brought to the fusion of kernels in order to economize the initialization time of the kernels.

TP1 : Accélération GPU



- Envoyer le calcul sur le GPU
- Test Mémoire



50

Tensor Cores

Tensor Cores ◀

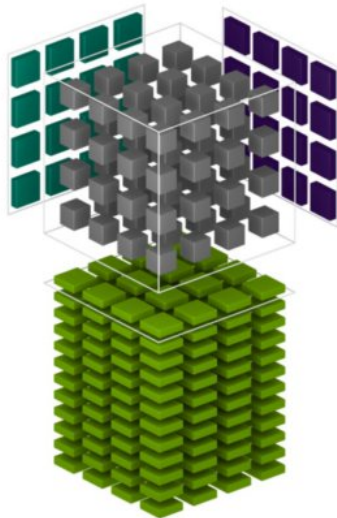
Precisions ◀

AMP ◀

Channel last memory format ◀

51

The objective of this section is to present Mixed Precision and the acceleration brought by using Tensor Cores available on the NVIDIA GPUs calibrated for the supercomputers.



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

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

$$D = \begin{pmatrix} A_{0,0} & A_{0,1} & A_{0,2} & A_{0,3} \\ A_{1,0} & A_{1,1} & A_{1,2} & A_{1,3} \\ A_{2,0} & A_{2,1} & A_{2,2} & A_{2,3} \\ A_{3,0} & A_{3,1} & A_{3,2} & A_{3,3} \end{pmatrix} \begin{pmatrix} B_{0,0} & B_{0,1} & B_{0,2} & B_{0,3} \\ B_{1,0} & B_{1,1} & B_{1,2} & B_{1,3} \\ B_{2,0} & B_{2,1} & B_{2,2} & B_{2,3} \\ B_{3,0} & B_{3,1} & B_{3,2} & B_{3,3} \end{pmatrix} + \begin{pmatrix} C_{0,0} & C_{0,1} & C_{0,2} & C_{0,3} \\ C_{1,0} & C_{1,1} & C_{1,2} & C_{1,3} \\ C_{2,0} & C_{2,1} & C_{2,2} & C_{2,3} \\ C_{3,0} & C_{3,1} & C_{3,2} & C_{3,3} \end{pmatrix}$$

FP16 or FP32
FP16
FP16
FP16 or FP32

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

Source : [NVIDIA](#)
52

NVIDIA has added Tensor Cores to its dedicated GPUs for the computing centers: Turing, Volta, Ampere.

The standard CUDA cores are specialized for vector calculations. Tensor Cores are specialized for tensorial (AKA matricial) calculations.

The Tensor Core version in V100s is capable of multiplying two 4x4 tables and aggregating it in one clock time.

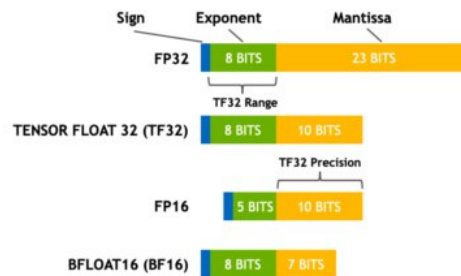
Tensor Cores use lower precision to accelerate the computation.

Tensor Cores, therefore, are more rapid than CUDA cores at equal density.

Précisions & Tensor Cores



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



Source : [NVIDIA](https://www.nvidia.com)
53

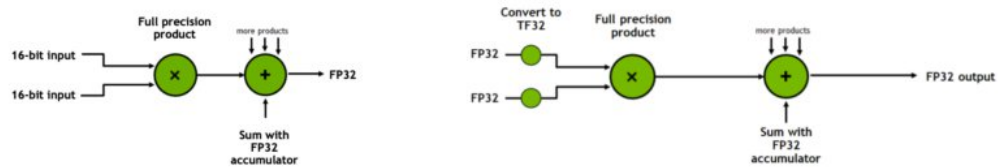
Different versions of Tensor Cores use certain precisions.

Each floating number precision uses a fixed number of bits with a signed bit, an exponent and a mantissa.

For the A100s, most of the precisions can be accelerated by Tensor Cores.

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	-	-



54

For V100s, FP16 precision using Tensor Cores is accelerated 8x in theory, compared to FP32 precision using CUDA Cores.

For A100s, TF32 precision using Tensor Cores is accelerated 8x in theory, and FP16 precision using Tensor Cores is accelerated 16x in theory compared to FP32 precision using CUDA Cores.

With CuDNN, FP32 precision is automatically transformed into TF32 without adding any code.

To transform the FP32s into FP16, it is necessary to implement some code lines by using, for example, the Automatic Mixed Precision (AMP) described later.

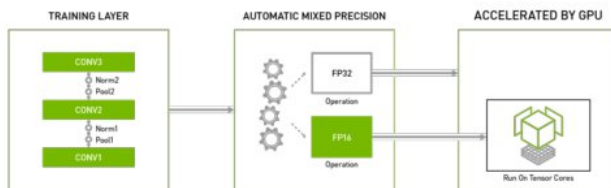
BF16 precision is what made TPUs successful. A100s, therefore, offer a compatibility with this precision.

The other precisions which A100s take into account concern inference. We will, therefore, not address inference during this course.

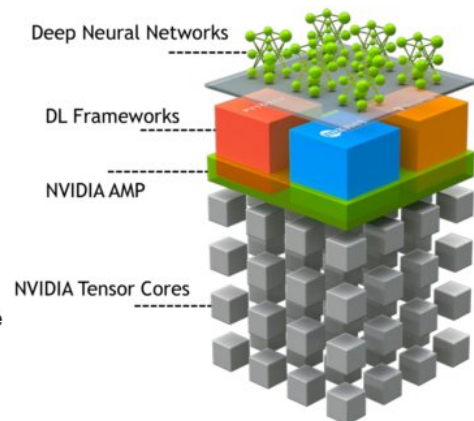
A100s also offer an optimization for sparse tensors. => Structured Sparsity

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



55

Mixed Precision means using some or all of the parameters in half precision, or FP16, to use Tensor Cores. This is necessary for V100 Tensor Cores. However, it requires a minimum of code.

It reduces the memory footprint and accelerates computations, while also causing what is usually an absolutely negligible precision loss in the calculation of gradients, Loss and Accuracy.

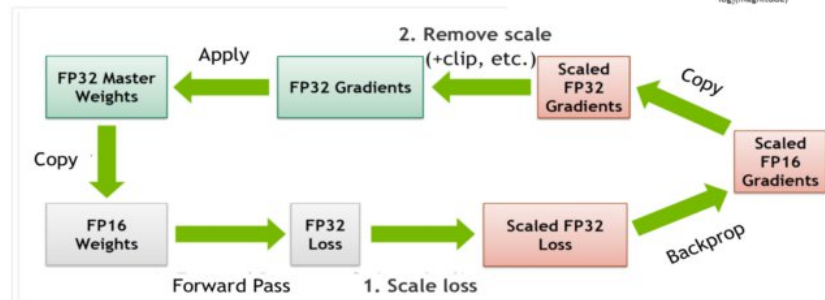
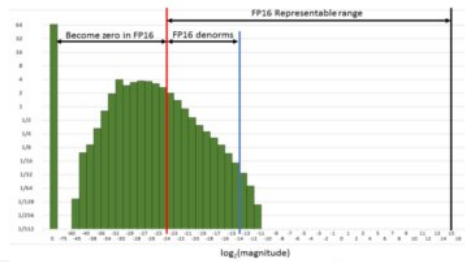
Automatic Mixed Precision dedicated to Deep Learning is an automation of the principle of Mixed Precision.

It transforms layers which are eligible for Mixed Precision (for example, ConvLayers are eligible but Batchnorm layers are not) and uses a scaling system for the calculation of gradients (see the following slide).

AMP Scaler

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

Distribution des gradients



56

The AMP Scaler method is often necessary as the calculated gradients can reach very small values or very close to zero. Indeed, FP16 precision considers every value inferior to $5.96e^{-8}$ as a nul value (value 0). Moreover, very high gradient values do not exist in reality.

Therefore, it is advantageous to transfer the FP16 precision normalized values to values closer to zero for the gradients. This is what is called scaling here.

In practice, the model with its weights expressed in FP32 is transformed into FP16 by the AMP (for the eligible weights), then the forward executes on the Tensor Core.

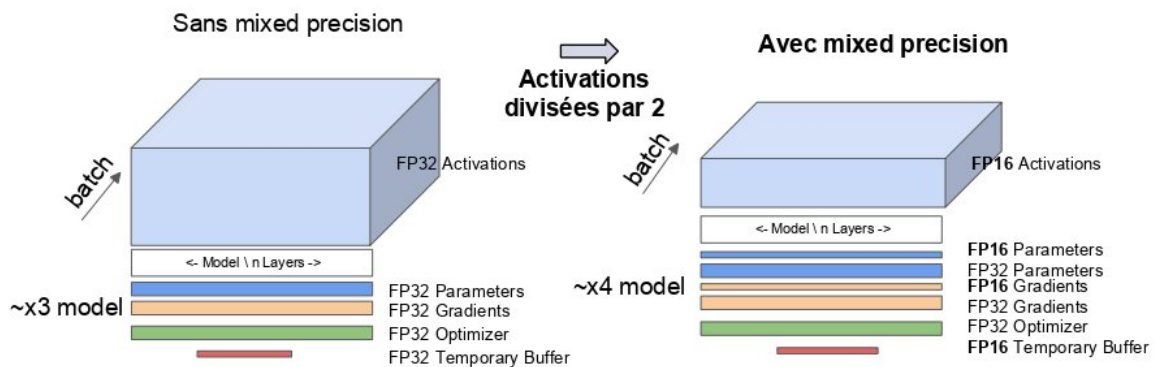
We obtain the FP32 Loss calculation in output.

We apply the scaling on the Loss.

During the backpropagation, the gradients are calculated on the Tensor Cores from the activation values in FP16. Then, the result of the gradients scaled in FP32 is inversely unscaled to update the model weights according to the optimizer method at the end.

This describes a single iteration of training with the AMP.

Empreinte mémoire avec la Mixed Precision

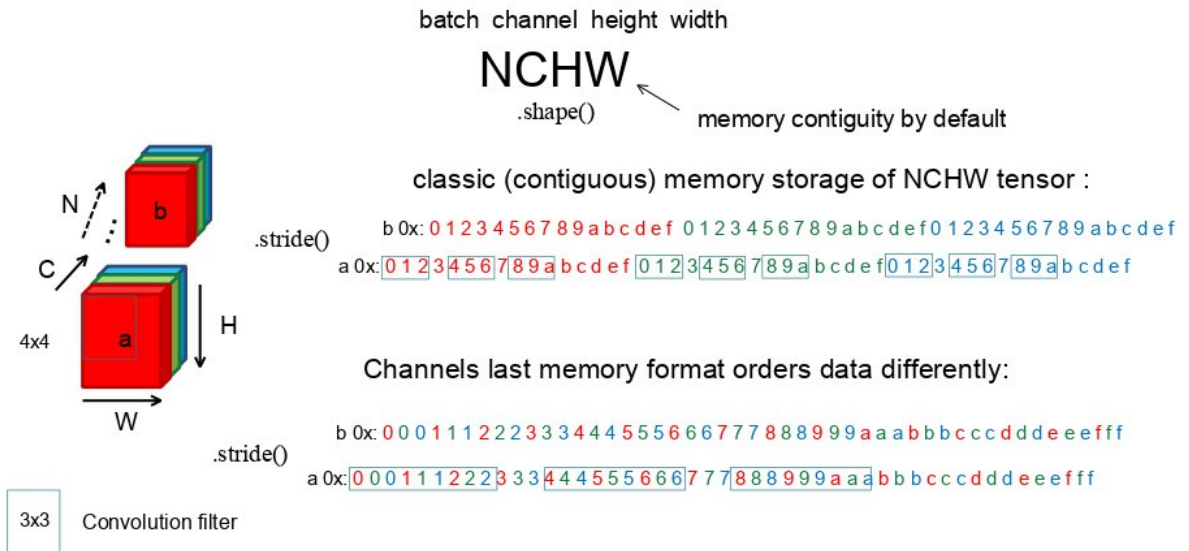


57

The impact of Mixed Precision on the memory footprint is advantageous most of the time because it divides the major activation outputs by 2.

However, it should be noted that the memory footprint linked to the model is increased with Mixed Precision as the gradients and parameters are conserved in FP32 and in FP16 at the same time. As we have seen, this is negligible for most of the models. Nevertheless, this should be taken into consideration for very large models.

Channel last memory format

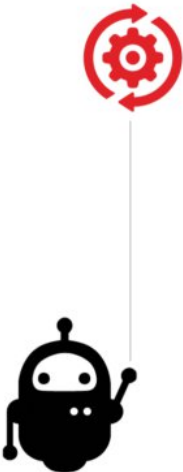


Data movement and memory allocation are essential to benefit from the Tensor Cores performance.

An image input in PyTorch corresponds to an NCHW (Batch Channel Height Width) tensor format. The memory is contiguously allocated according to this order.

Forcing memory contiguity in the Channel direction first enables pooling contiguous data when applying convolutional filters.

TP2&3 : Automatic Mixed Precision



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