IDRIS

DLO-JZ turbo

Une rapide vue de la formation “Deep Learning Optimisée sur Jean Zay”

AI DevTalks #6, 14 décembre 2022

Bertrand Cabot, Idris/Cnrs

DLO-JZ Overview

DL Training Optimization

Day 1

- **Jean Zay**
- **The challenges of scaling**
- **GPU computing**
- **Distribution - Data Parallelism**
- **Dataloader**

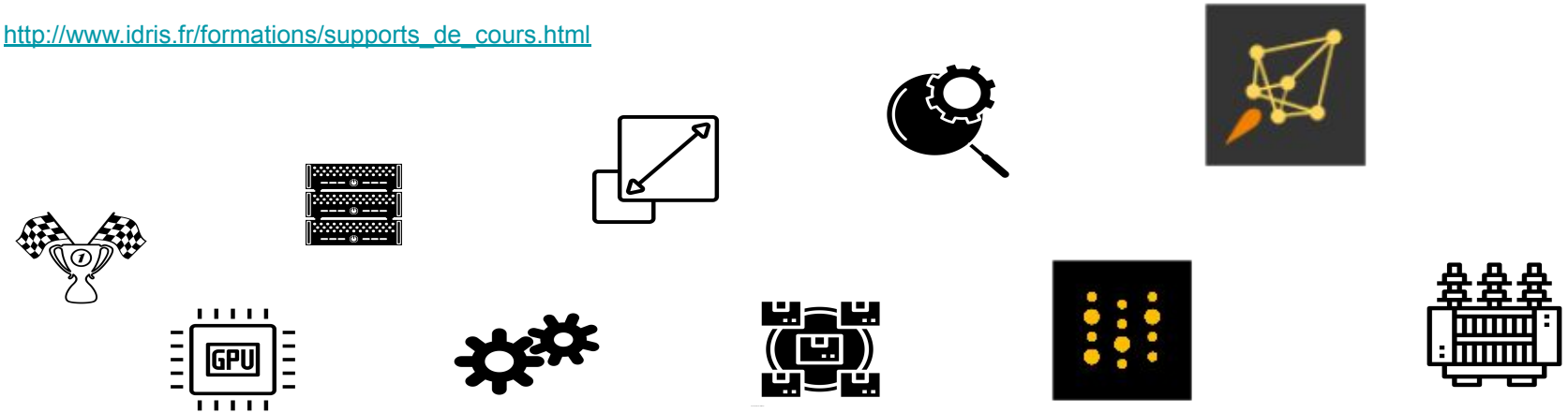
Day 2

- Profiler
- Data Augmentation
- ***Optimizer* & large *batch***
- Visualization Weight & Biases
- SOTA & tricks

Day 3

- **Model Parallelisms**
- Deepspeed
- Transformers / Big Models

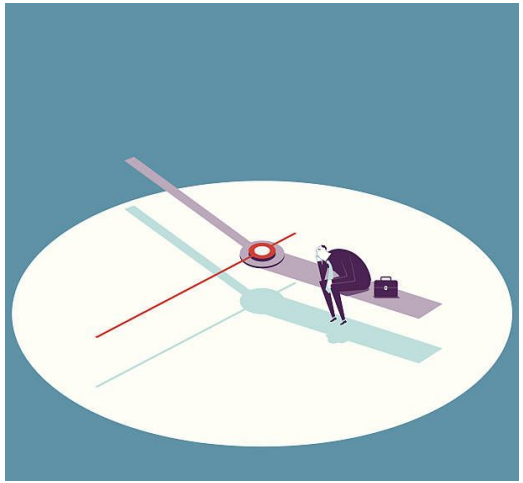
http://www.idris.fr/formations/supports_de_cours.html



Deep Learning Training Cost Items

2 issues to deal:

Training Time

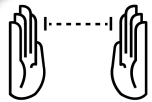


Memory overconsumption (OOM)

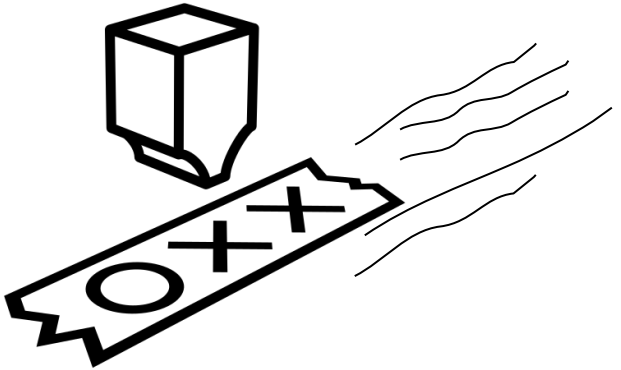
Computation time

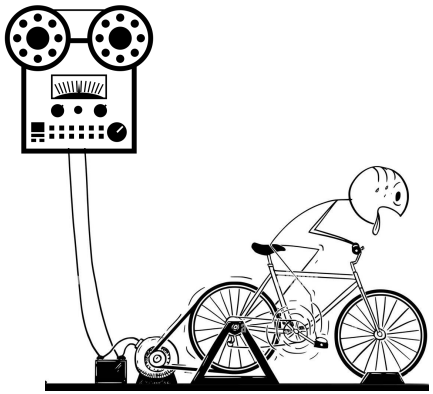
The computation time increases with **the FLOPs needed**, depending on:

- Model size
- The depth of the model
- The size of the input data (Image resolution, length of the sequence, etc.)
- The size of the dataset
- Number of epochs required

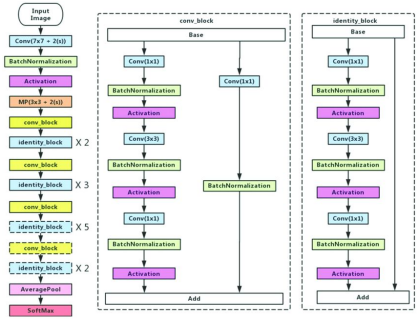


} The complexity of the task





Computation time



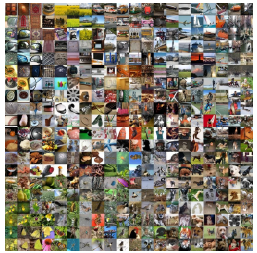
Resnet-50
25.6 M parameters

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

Before 2017

14 days

NVIDIA M40 GPU

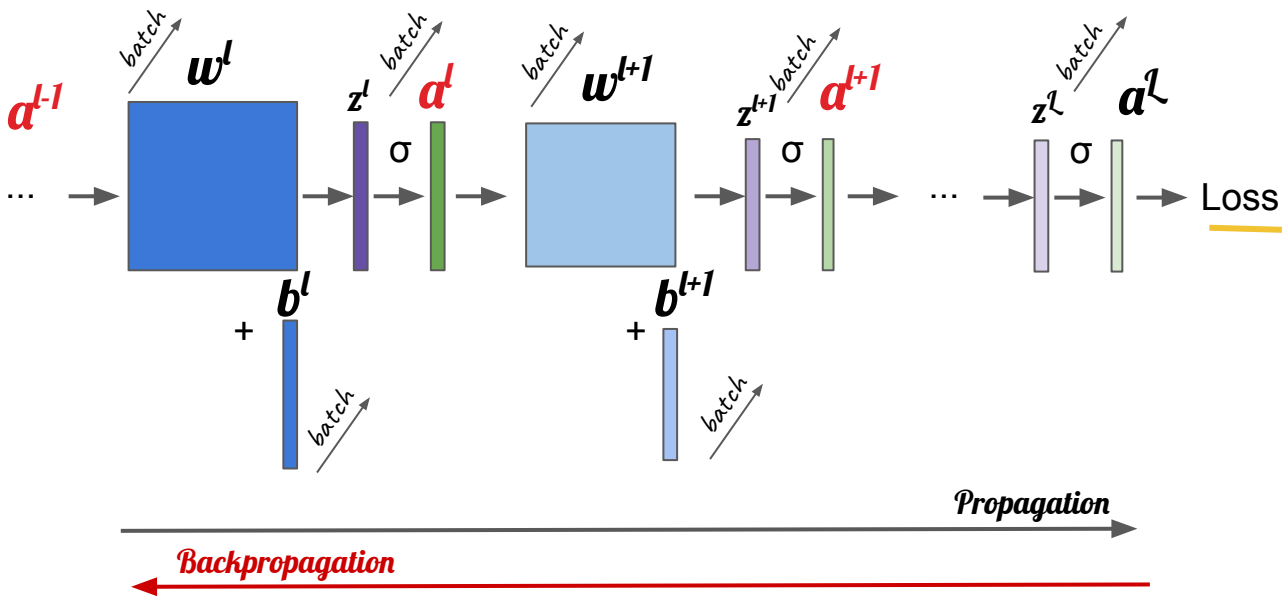


Imagenet
1.2 M images

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

Backpropagation - intermediate activations issue



Propagation

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

Backpropagation

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

$$w^l \to w^l - \frac{\eta}{m} \cdot \frac{\partial C}{\partial w^l}$$

$$b^l \to b^l - \frac{\eta}{m} \cdot \frac{\partial C}{\partial b^l}$$

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

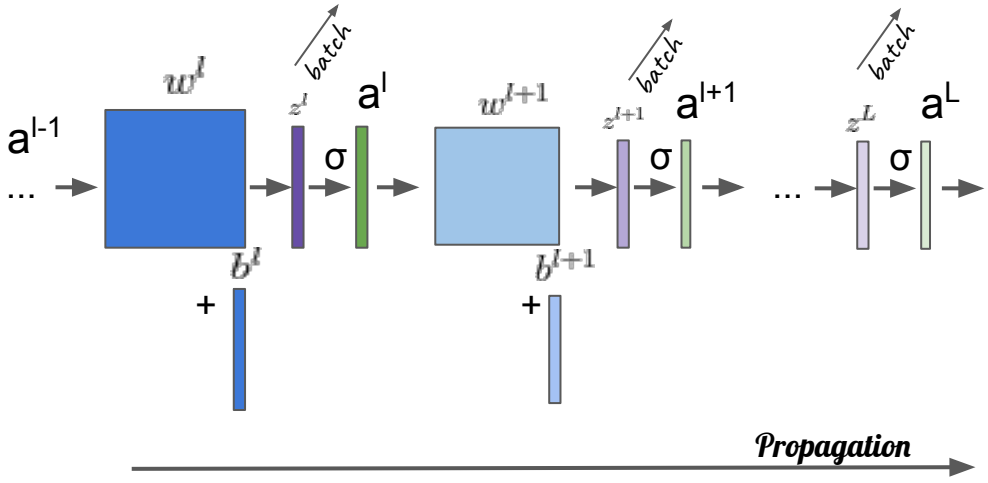
$$\delta^l = ((w^{l+1})^T \delta^{l+1}) \odot \sigma'(z^l)$$

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

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

Note: For backpropagation, it is necessary to keep in memory the intermediate activations.

Inference & Evaluation

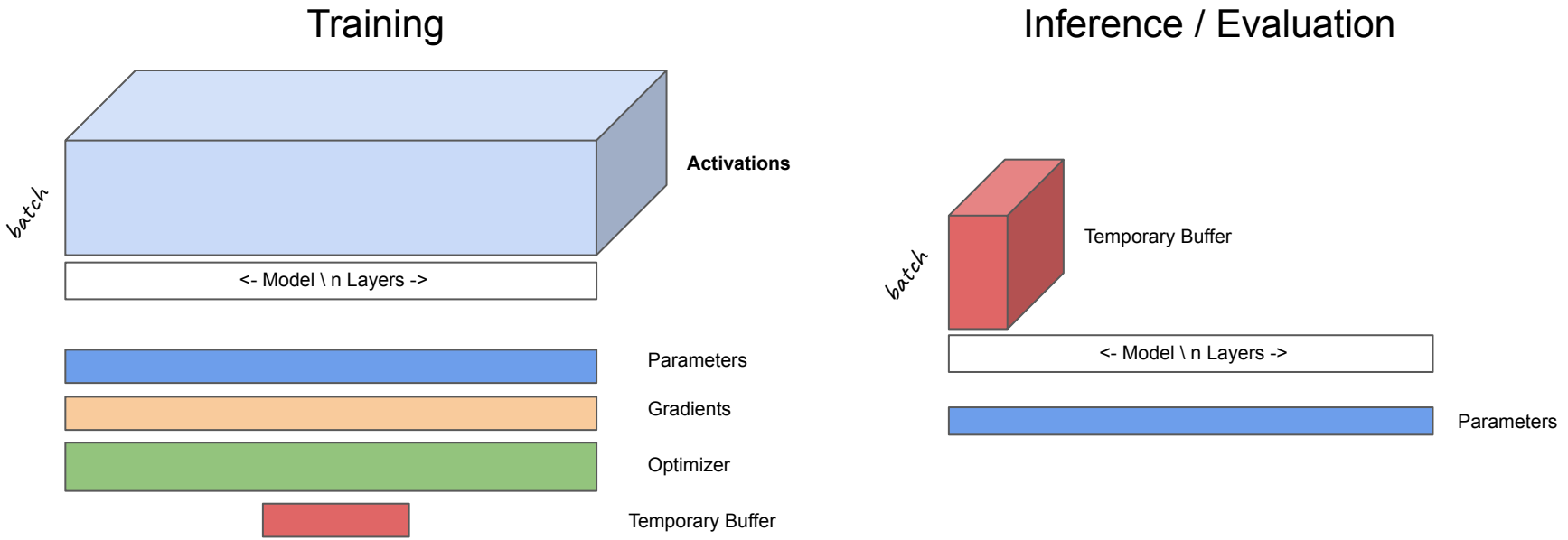


Propagation

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

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

Memory allocation



Power / GPU Hours Saving

Power saving

$\asymp$

GPU hour saving

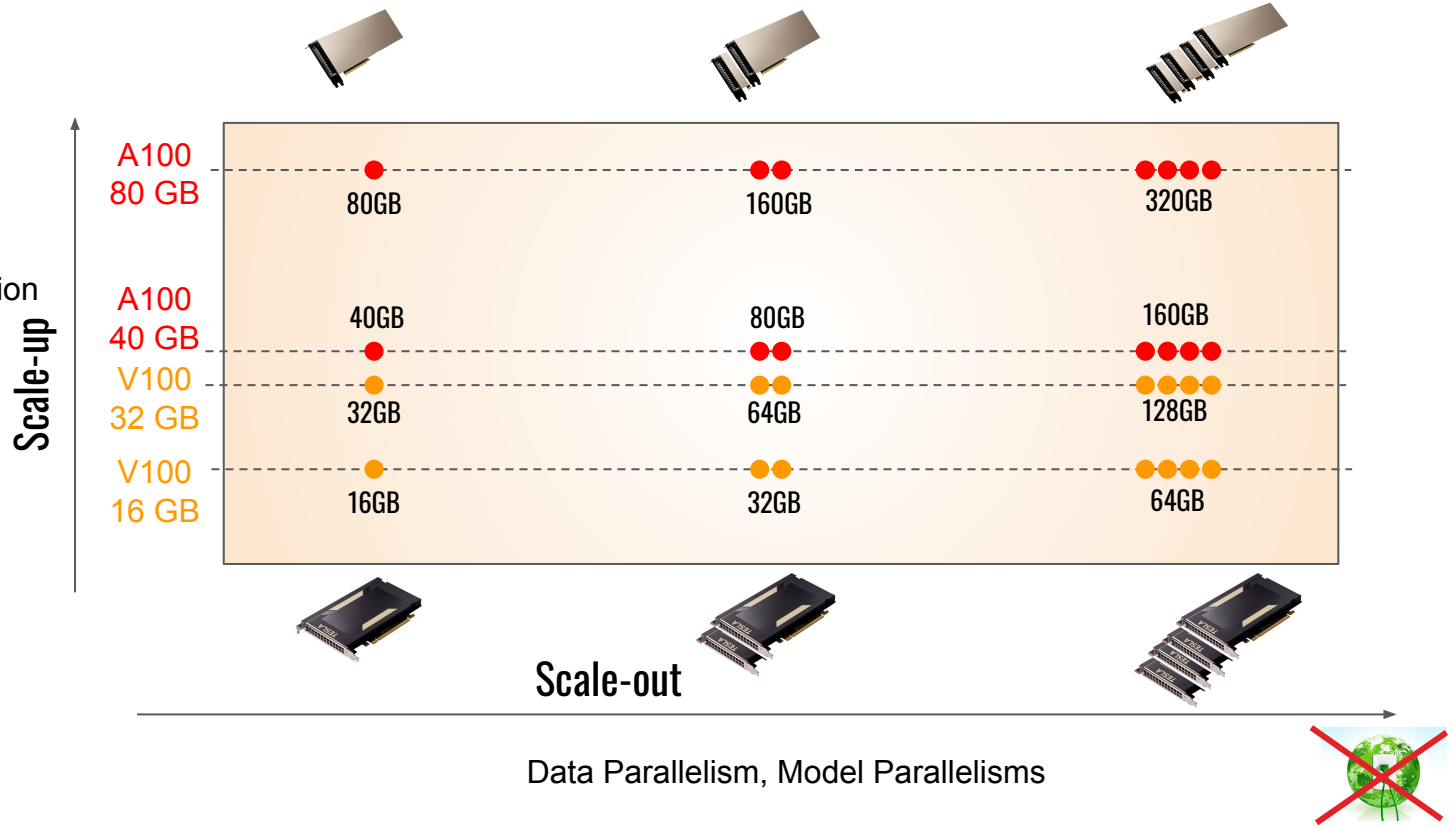
- **System Optimization (DLO-JZ)**
- **Methodology: limit the number of trial**
  - Read posts
  - Search for hyper parameters on smaller models
  - Hyper-Parameter Optimization (HPO) techniques : BO, PBT, SHA, ASHA

Scaling solutions

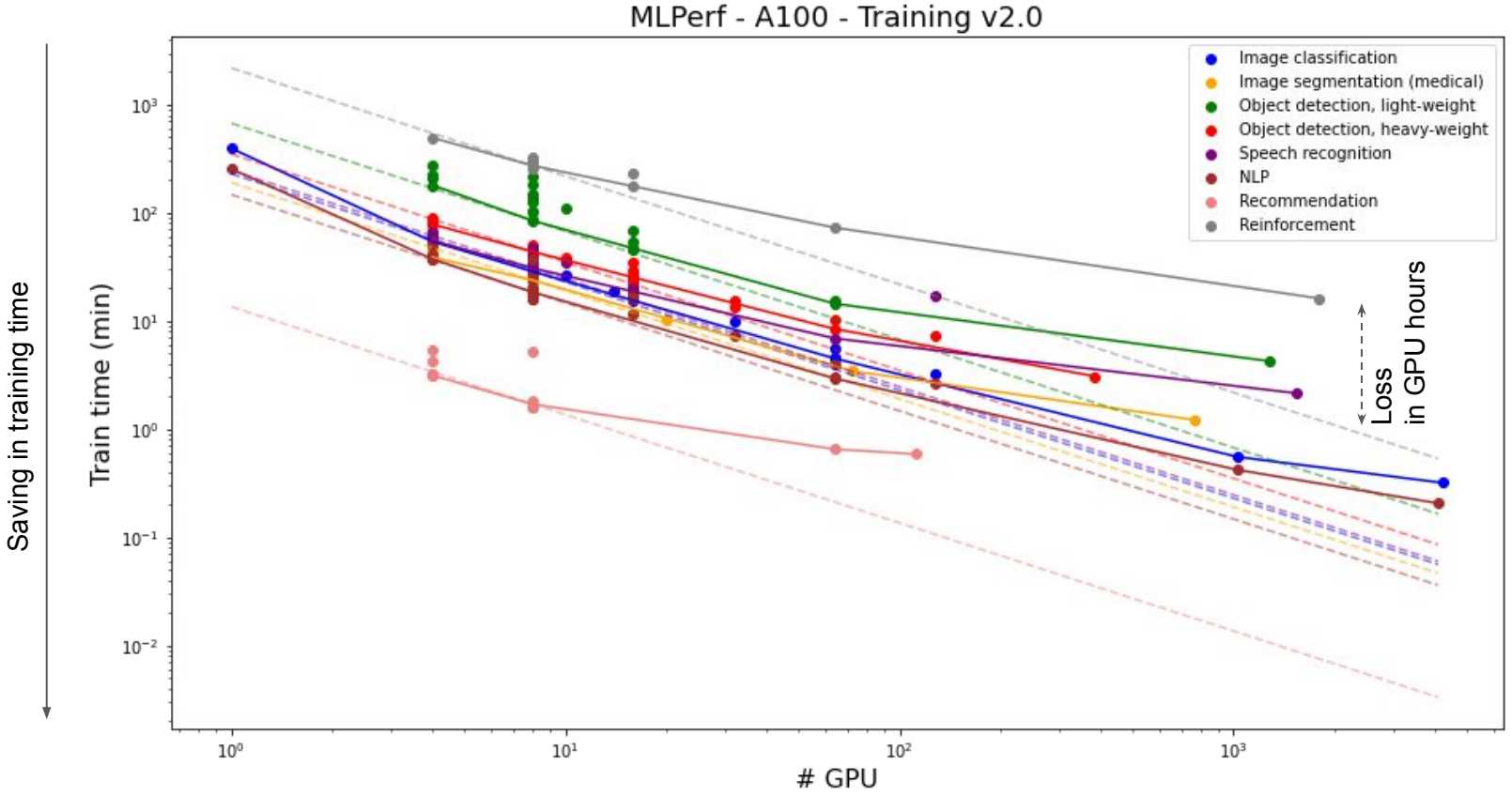
Mixed Precision

Tensor core

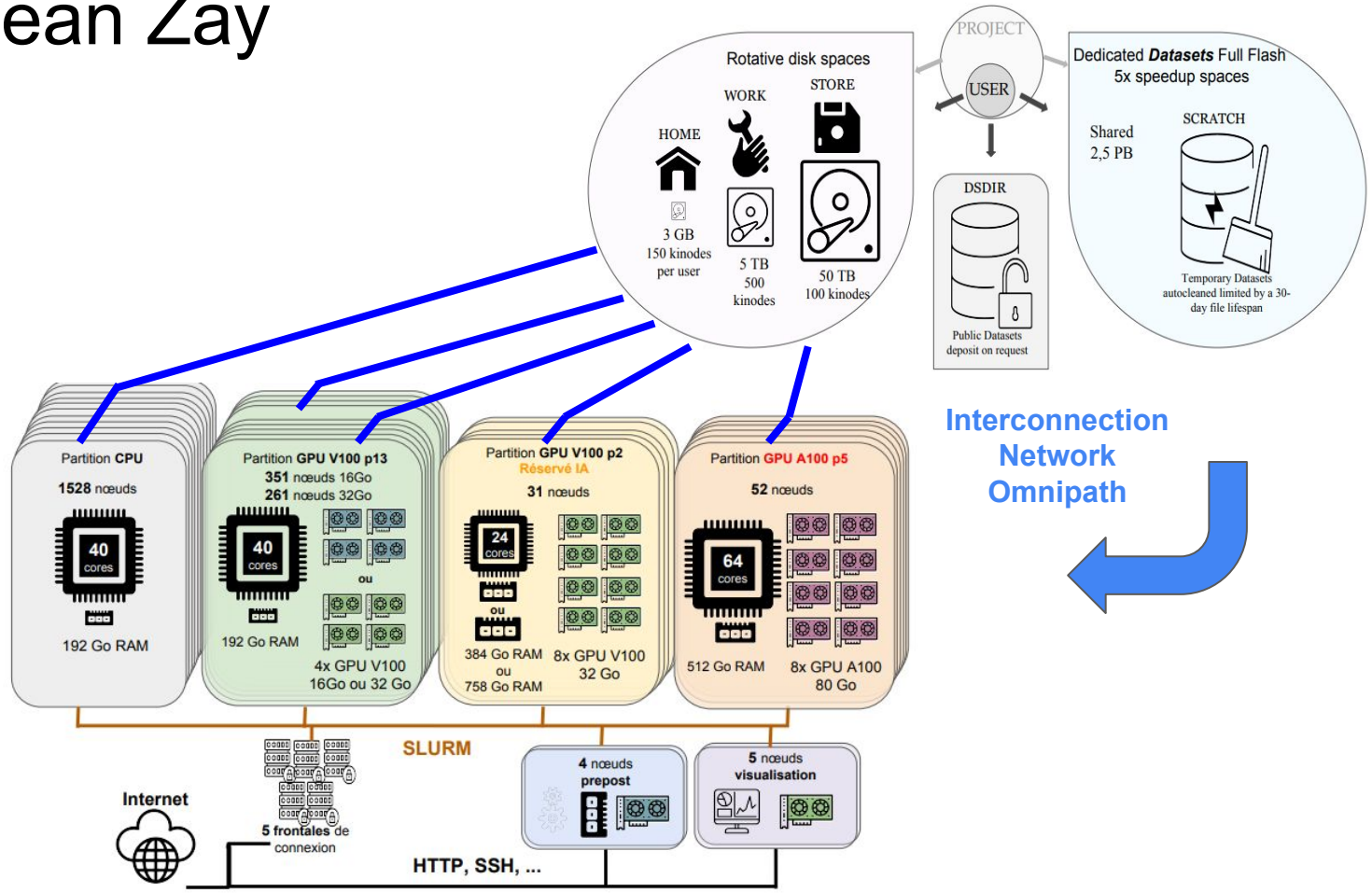
GPU



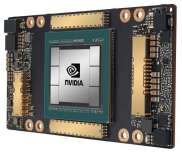
MLPerf - Scaling

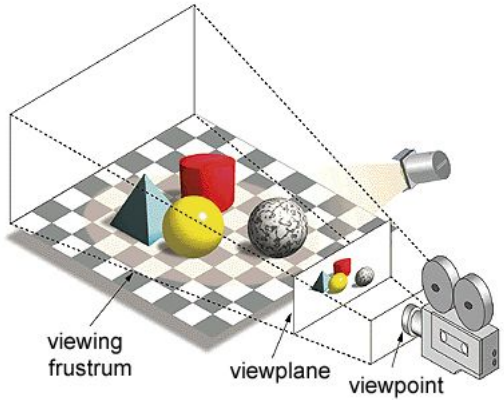


Jean Zay

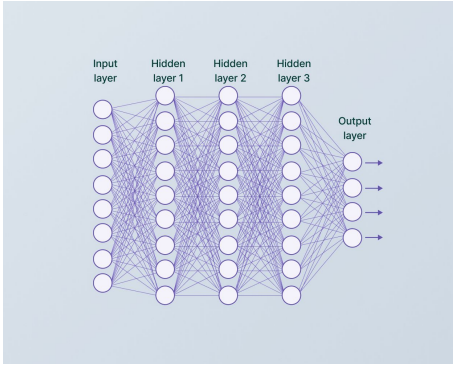

Rotative disk spaces
HOME
WORK
STORE
3 GB
150 kinodes
per user
5 TB
500
kinodes
50 TB
100 kinodes
PROJECT
USER
DSDIR
Public Datasets
deposit on request
Dedicated Datasets Full Flash
5x speedup spaces
Shared
2,5 PB
SCRATCH
Temporary Datasets
autocleaned limited by a 30-
day file lifespan
Interconnection
Network
Omnipath
Partition CPU
1528 nœuds
40
cores
192 Go RAM
Partition GPU V100 p13
351 nœuds 16Go
261 nœuds 32Go
40
cores
192 Go RAM
ou
4x GPU V100
16Go ou 32 Go
Partition GPU V100 p2
Réservé IA
31 nœuds
24
cores
ou
384 Go RAM
ou
758 Go RAM
8x GPU V100
32 Go
Partition GPU A100 p5
52 nœuds
64
cores
512 Go RAM
8x GPU A100
80 Go
SLURM
Internet
5 frontales de
connexion
4 nœuds
prepost
5 nœuds
visualisation
HTTP, SSH, ...

GPU Computing

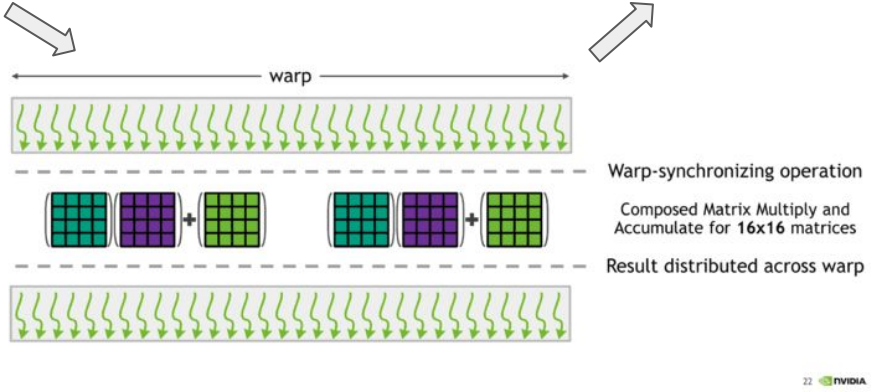




GPU Rendering & Game Graphics Pipeline



DNN

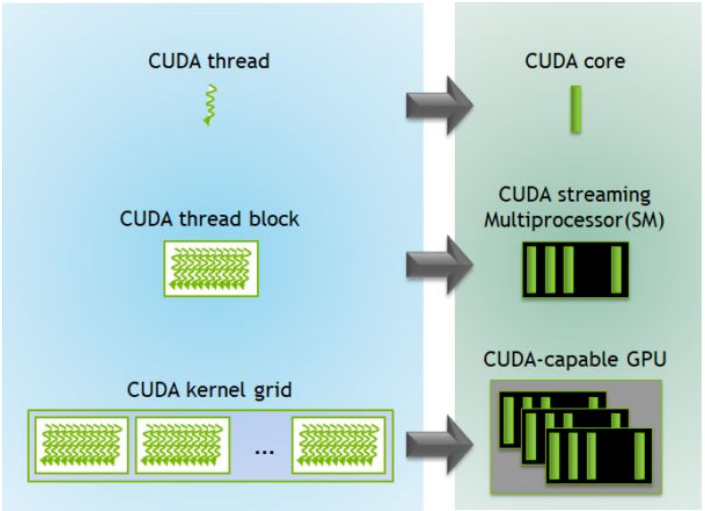


Matrix Multiply-accumulate operations

CuDNN

PyTorch

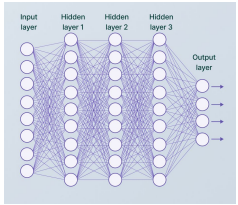
NVIDIA CUDA cuDNN



Caffe Caffe2 Chainer Cognitive Toolkit MATLAB NVIDIA

mxnet PaddlePaddle PYTORCH TensorFlow ...

NVIDIA DEEP LEARNING SDK and CUDA

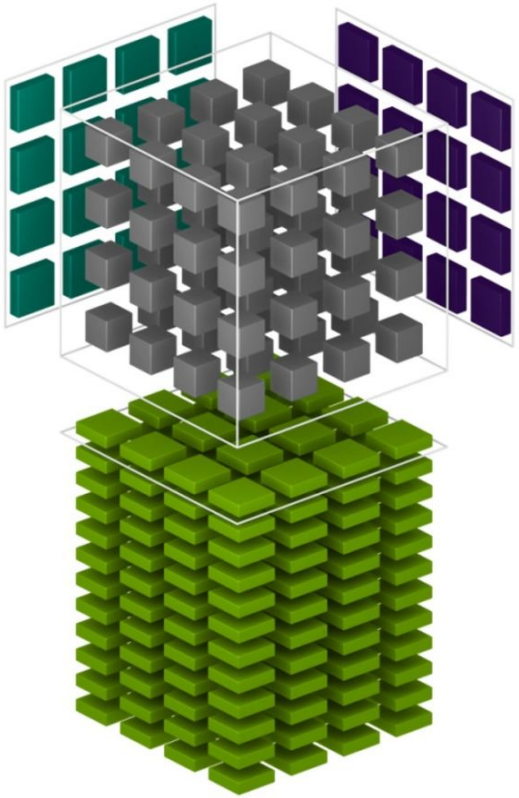


- **Merci cuDNN**
  for everything you do for me

cuDNN

- Please use dimensions (batch size, sample size, channel, etc…) that are **multiples of 8**!!

Tensor core



CPU Core : scalar computing

⇩

CUDA Core : vector computing.

⇩

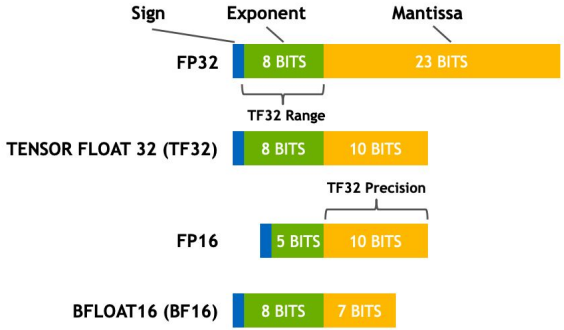
Tensor Core : **matrix computing**.

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

In 1 clock time !!!

Mixed Precision

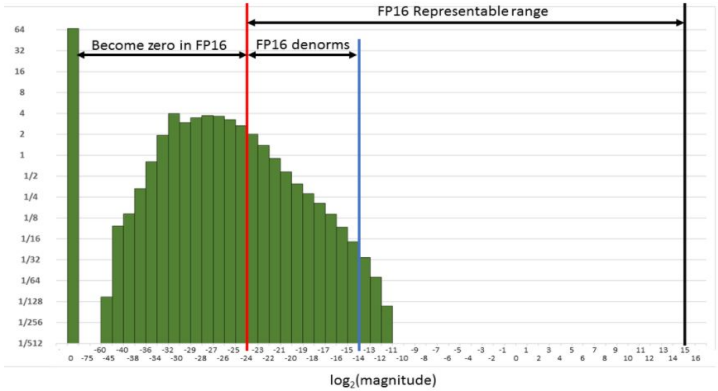
	NVIDIA A100	NVIDIA Volta
Supported Tensor Core Precisions	FP64, TF32, bfloat16, FP16, INT8, INT4, INT1	FP16
Supported CUDA® Core Precisions	FP64, FP32, FP16, bfloat16, INT8	FP64, FP32, FP16, INT8



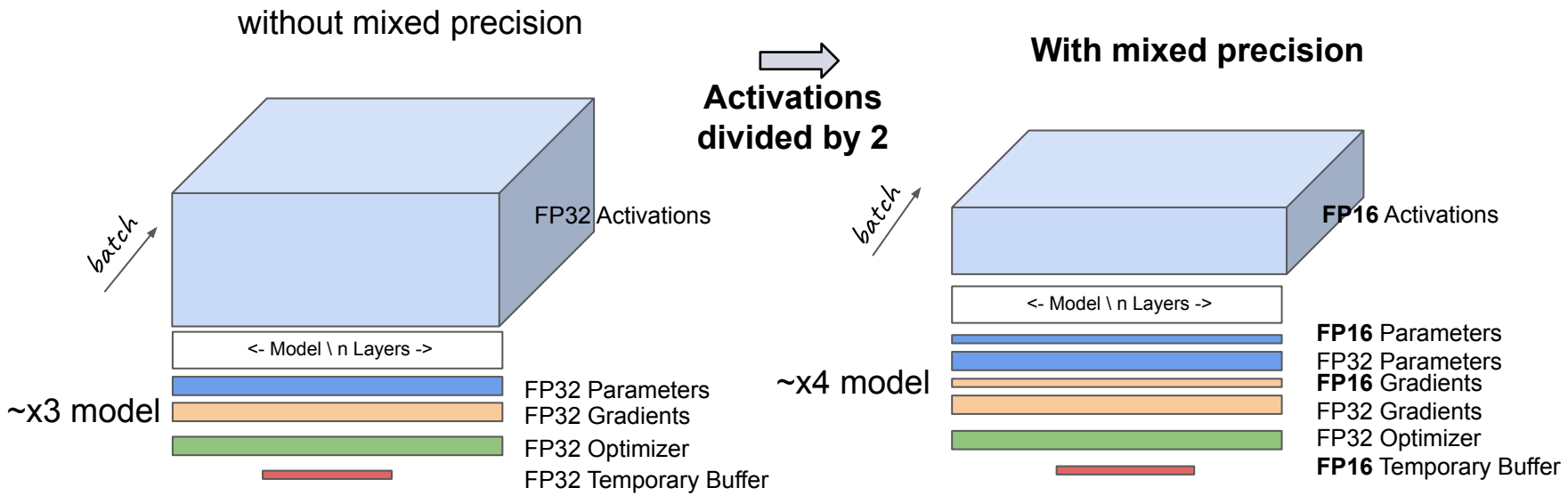
In **FP16** (half precision) values lower than 2^{-24} ($5.96e^{-8}$) are considered as 0.

You need a scaler for backpropagation !!

Gradient Distribution

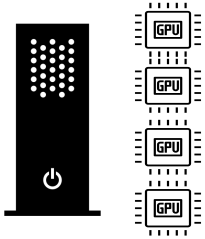


Mixed Precision Memory allocation



Job submission - SLURM

Single device / single user

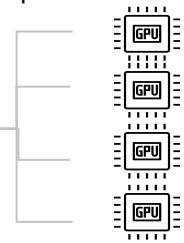


python train.py

1 process

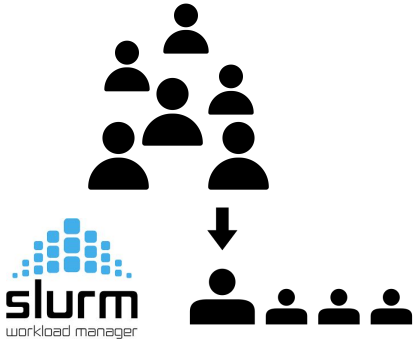
spawn

process 0



process N

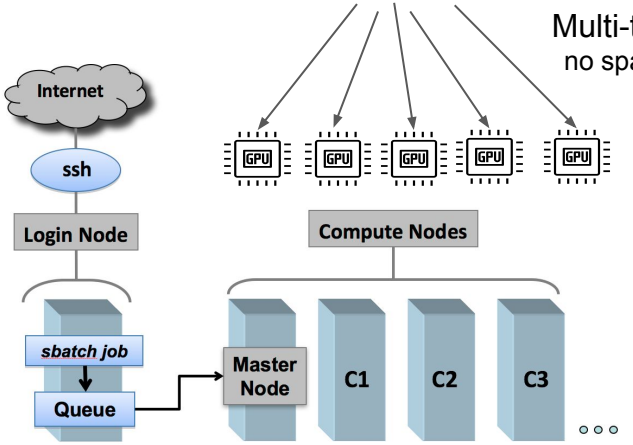
Jean Zay
Multi-nodes system
Super computing



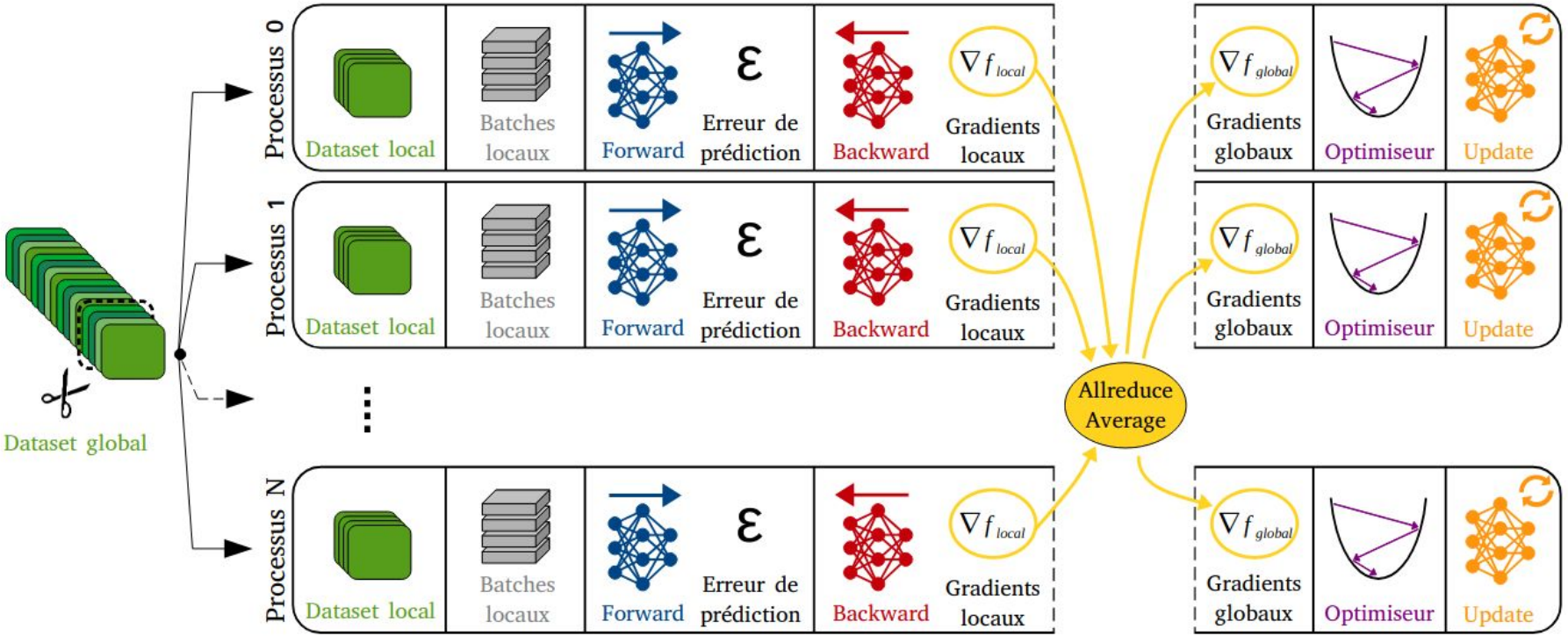
Job scheduler

srun python train.py

Multi-tasks jobs
no spawn to add



Data Parallelism



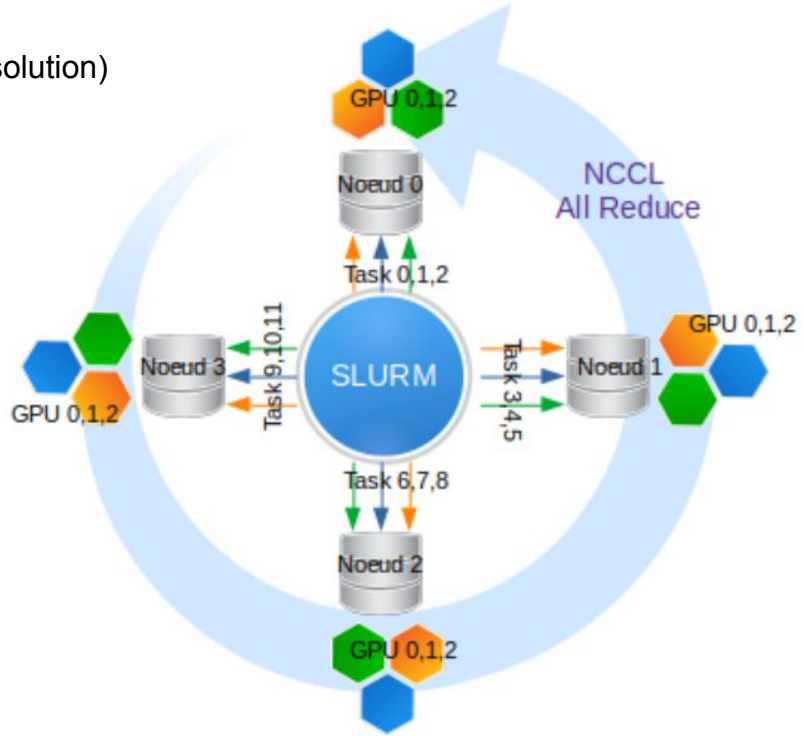
Multi-node Data Parallelism

- **PyTorch** → DistributedDataParallel (Embedded solution)
- **TensorFlow** → MultiWorkerMirroredStrategy (Embedded solution)
- **Horovod** (Dedicated external library)

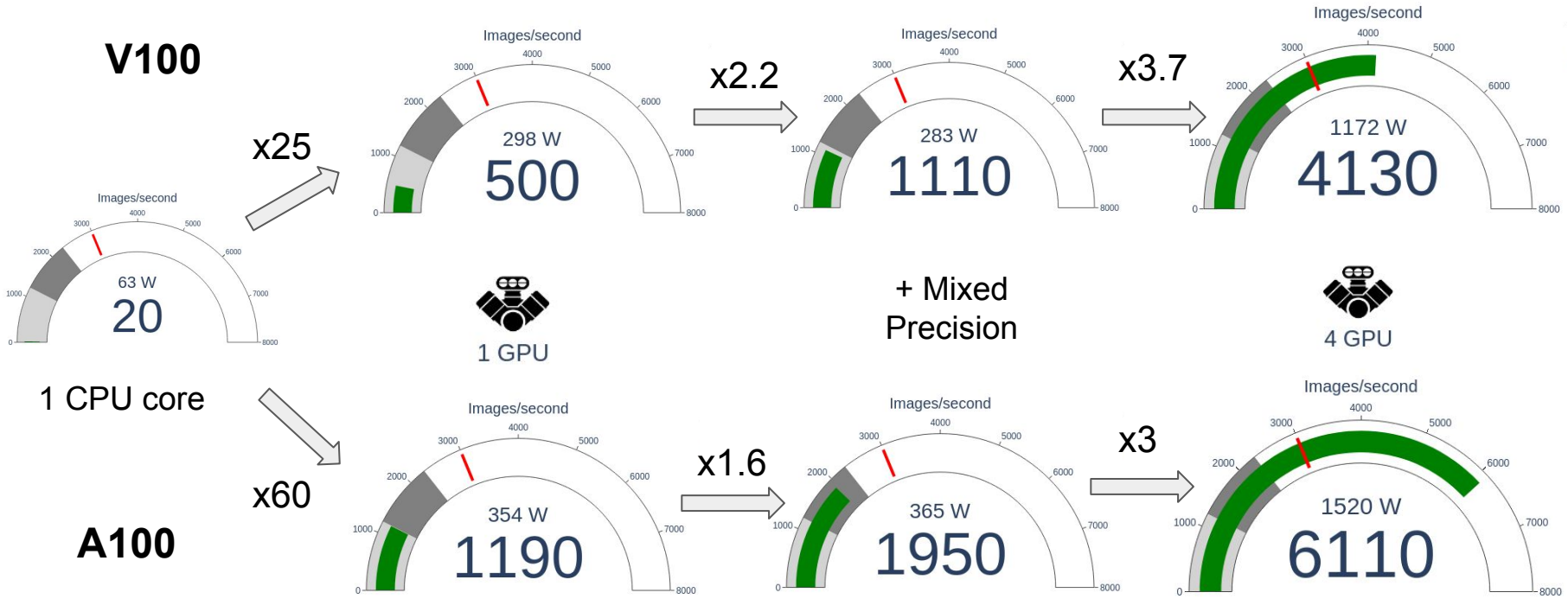
Distribution example : 4 nodes
3 GPU per node

```
## Slurm submission script

#SBATCH --nodes=4                # nb noeuds
#SBATCH --ntasks=12              # nb proc
#SBATCH --ntasks-per-node=3      # nb proc / noeud
#SBATCH --gres=gpu:3             # nb GPU  / noeud

srun python script.py
```

One process must be attached to each GPU.



TP Scaling

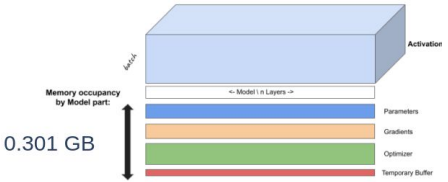

V100
Images/second
63 W
20
x25
Images/second
298 W
500
x2.2
Images/second
283 W
1110
x3.7
Images/second
1172 W
4130
1 GPU
+ Mixed Precision
4 GPU
1 CPU core
x60
A100
Images/second
354 W
1190
x1.6
Images/second
365 W
1950
x3
Images/second
1520 W
6110

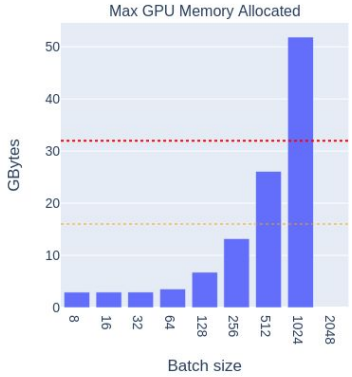
TP Scaling

FP32

V100

A100

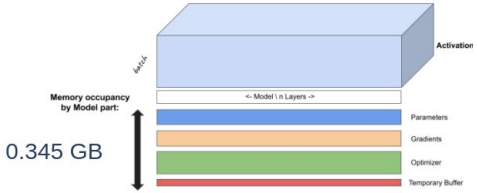


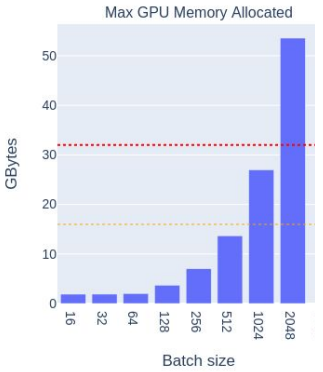


Mixed Precision

V100

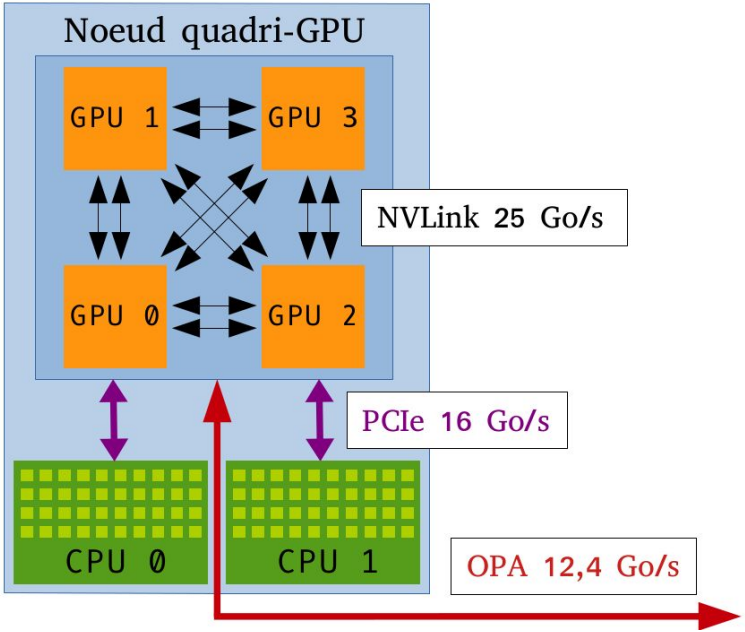
A100



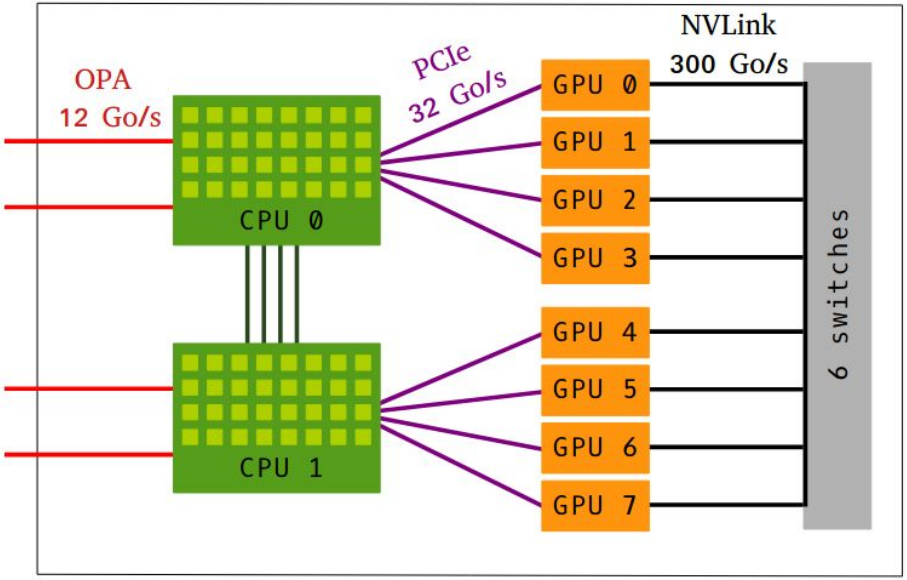


Interconnection bandwidth

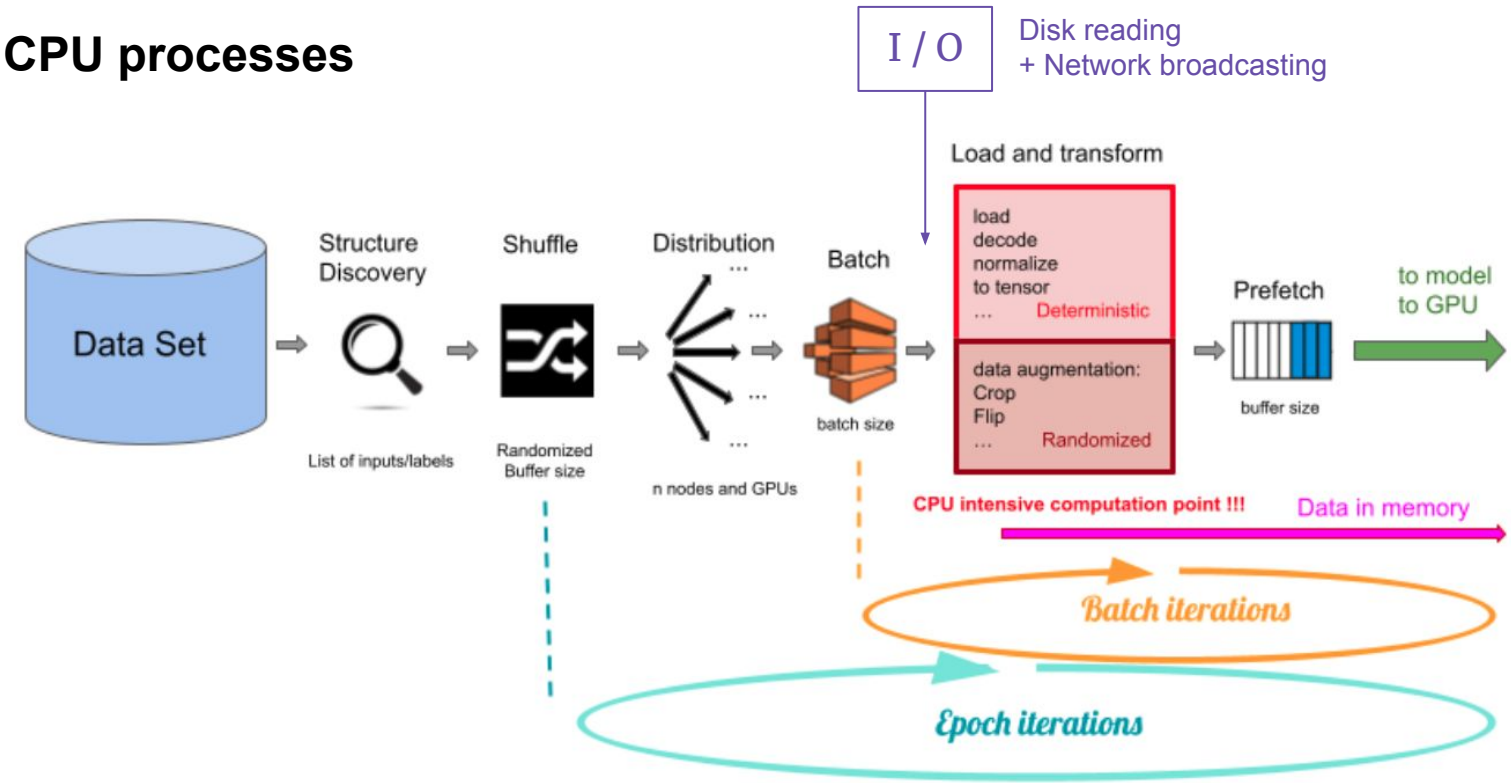
V100 Quadri-GPU node



A100 Octo-GPU node

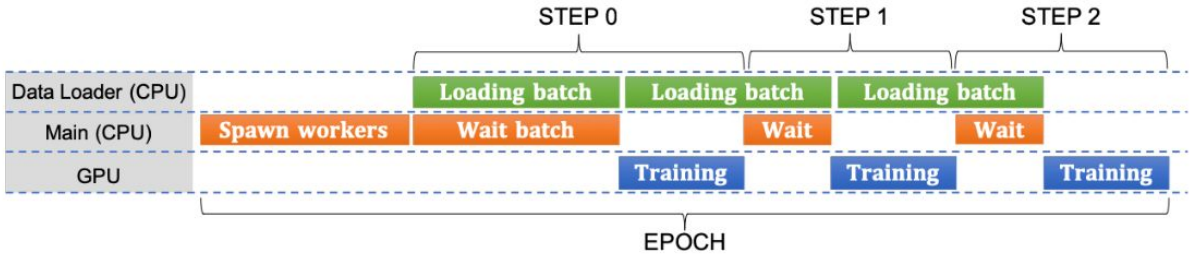


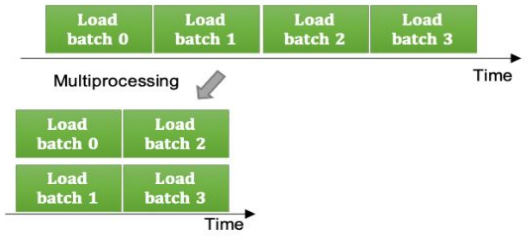
Dataloader


CPU processes
I / O
Disk reading
+ Network broadcasting
Load and transform
Data Set
Structure Discovery
Shuffle
Distribution
Batch
load
decode
normalize
to tensor
… Deterministic
data augmentation:
Crop
Flip
… Randomized
Prefetch
to model
to GPU
List of inputs/labels
Randomized Buffer size
n nodes and GPUs
batch size
buffer size
CPU intensive computation point !!!
Data in memory
Batch iterations
Epoch iterations

Dataloader

make CPU/GPU processes **asynchronous !!**






```
########## DATALOADER #############

train_loader =
torch.utils.data.DataLoader(dataset=train_dataset,
                            batch_size=mini_batch_size,
                            shuffle=False,
                            sampler=train_sampler,
                            num_workers=8,
                            persistent_workers=True,
                            pin_memory=True,
                            prefetch_factor=2)

for i, (images, labels) in enumerate(train_loader):
    # distribution of images and labels to GPUs
    images = images.to(gpu, non_blocking=True)
    labels = labels.to(gpu, non_blocking=True)
```

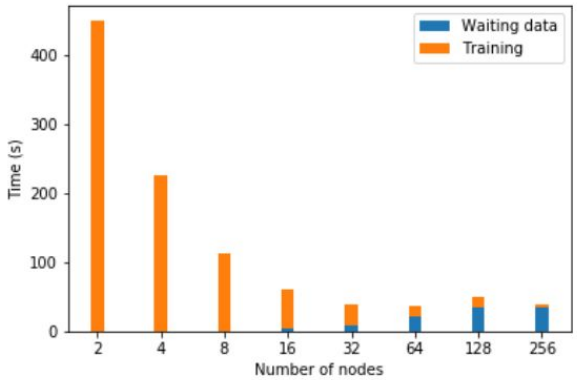
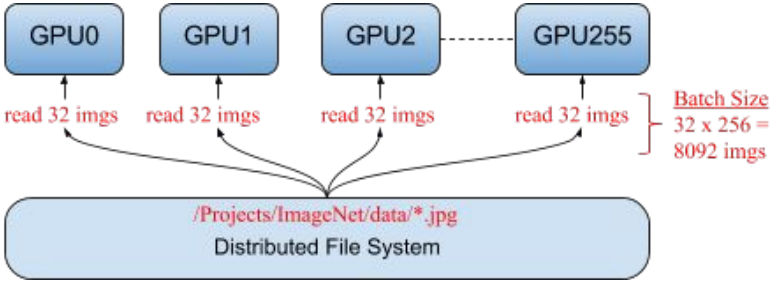
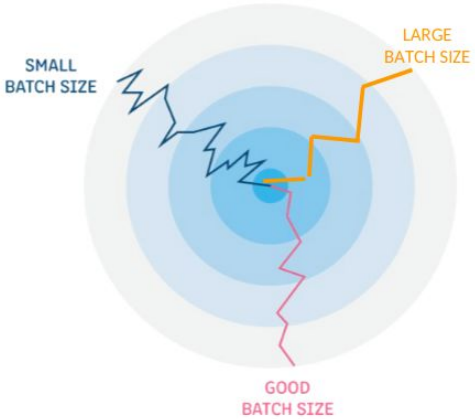


Fig. 1. Average epoch time to train ResNet50 with Imagenet-1K dataset in different scales on LLNL Lassen. The cost stopped decreasing when the data loading overhead stopped scaling.

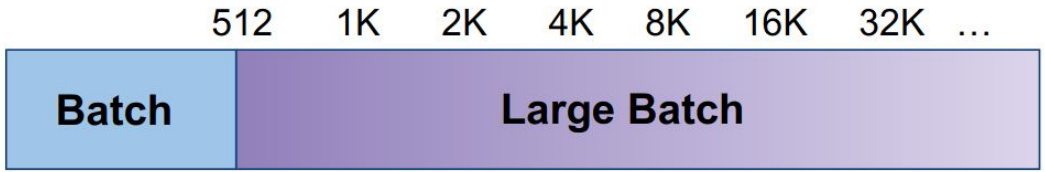
Large Batch with DP



Data Parallelism generates Large Batch !!



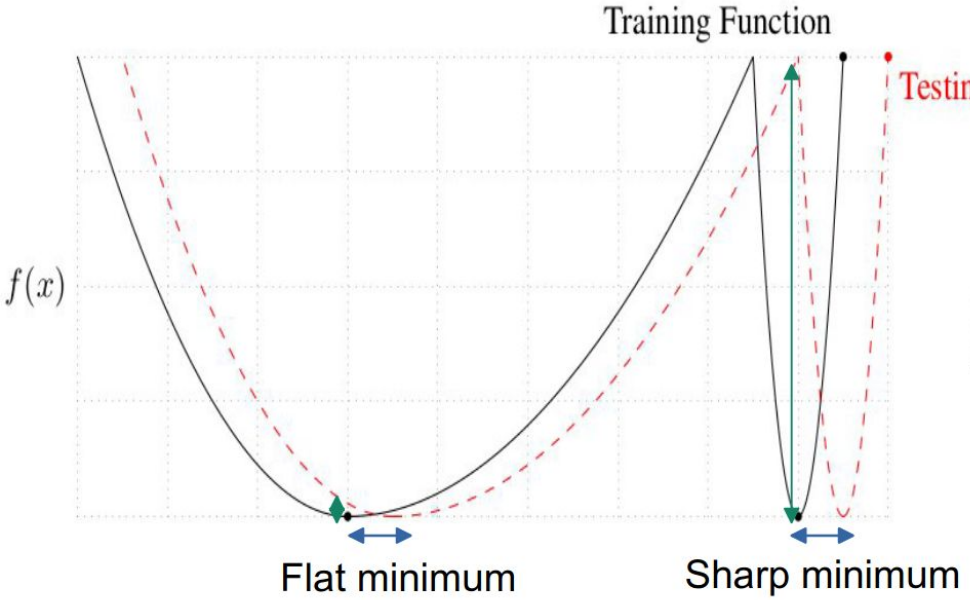
=> larger learning rate

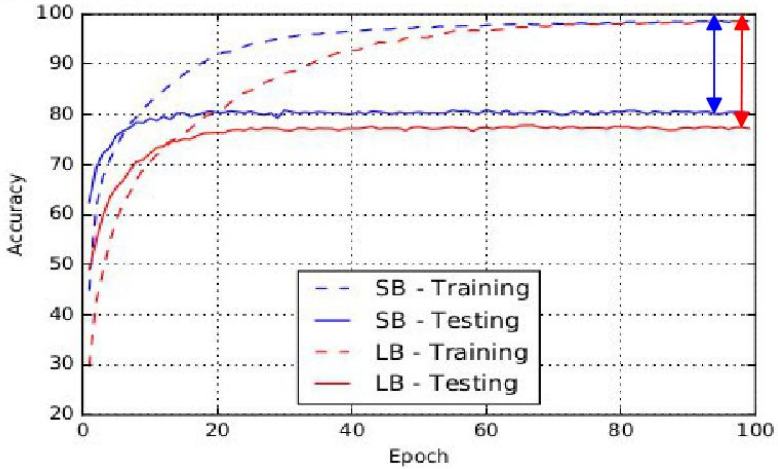


Large Batch GradientDescent

On Large-Batch Training for Deep Learning: Generalization Gap and Sharp Minima

Nitish Shirish Keskar, Dheevatsa Mudigere, Jorge Nocedal, Mikhail Smelyanskiy, Ping Tak Peter Tang





Comparaison d'entraînement d'un réseau convolutionnel avec des petits batch (SB) et large batch (LB) sur CIFAR 10

The larger the batch, the more the model tends to **converge towards sharp minimas**.

Large Batch Rider

SGD AdamW

Weight Decay

LARS

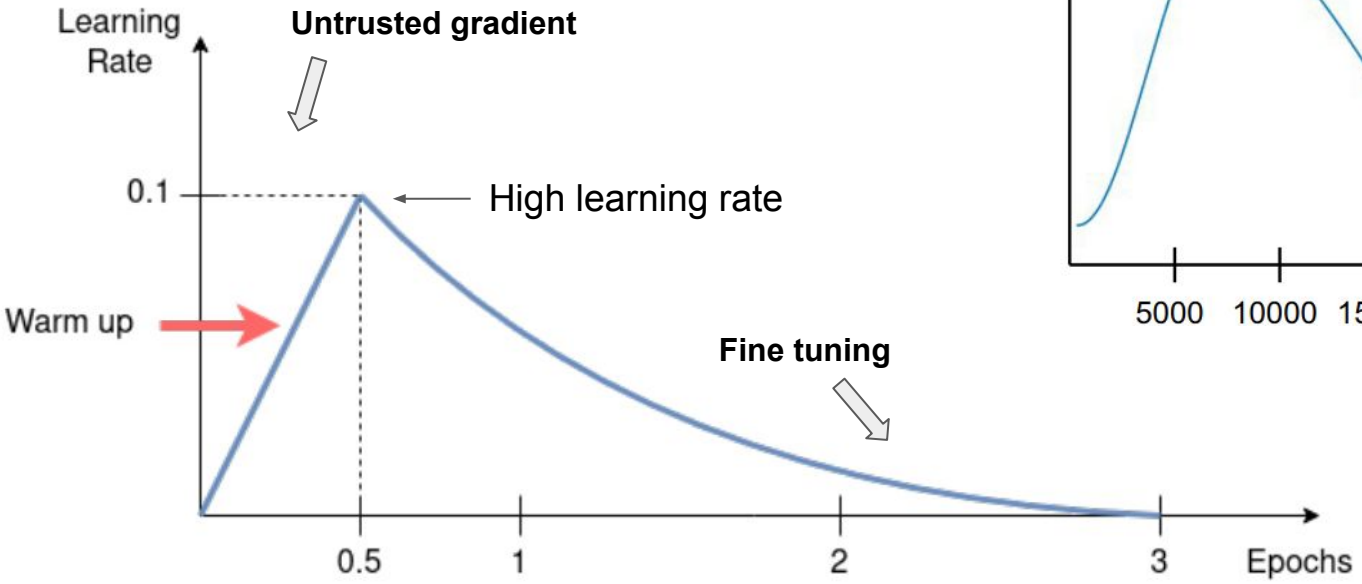
LAMB

LR Scheduler

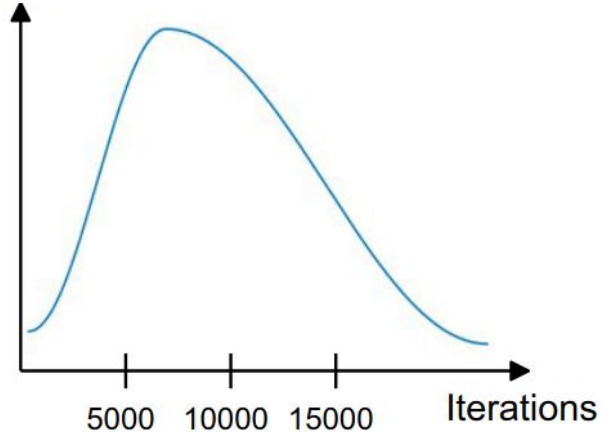
Warmup LR scaling LR Decay

Learning rate scheduler

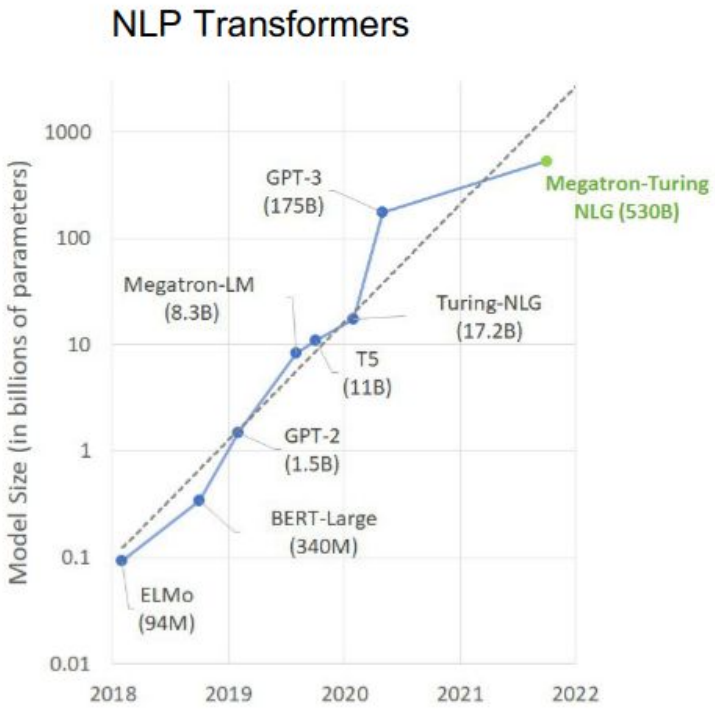
WARMUP for *large batches*



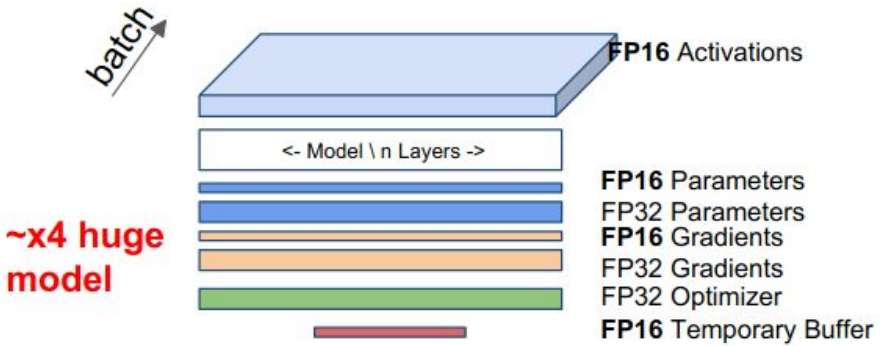
Onecycle LR : Recommended by FastAI for super convergence



Huge Model > 1 billion of parameters

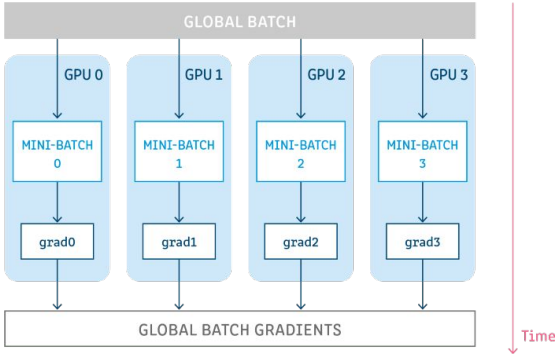


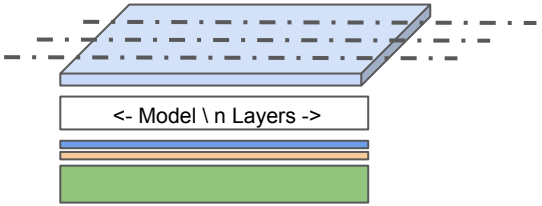
CUDA Out Of Memory



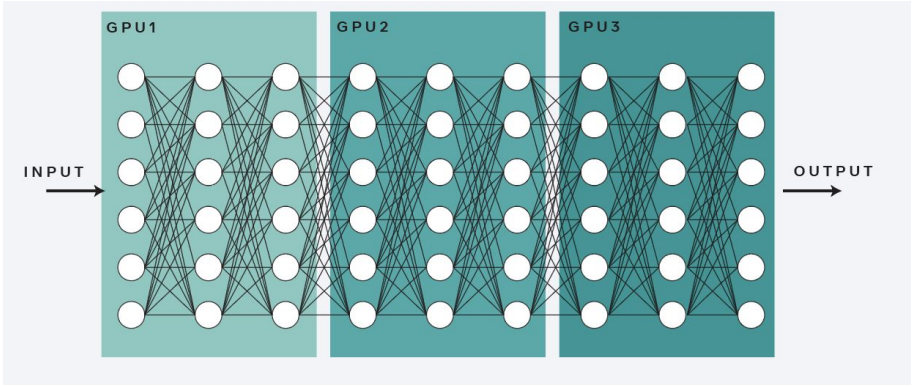
Parallelisms for huge models

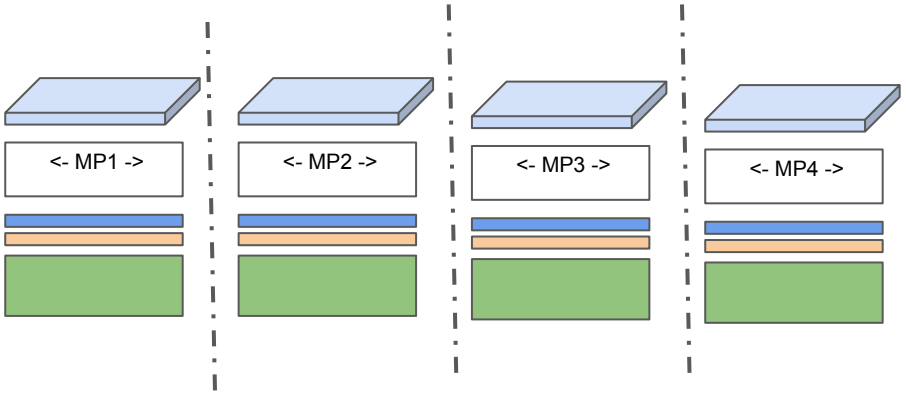
Data Parallelism



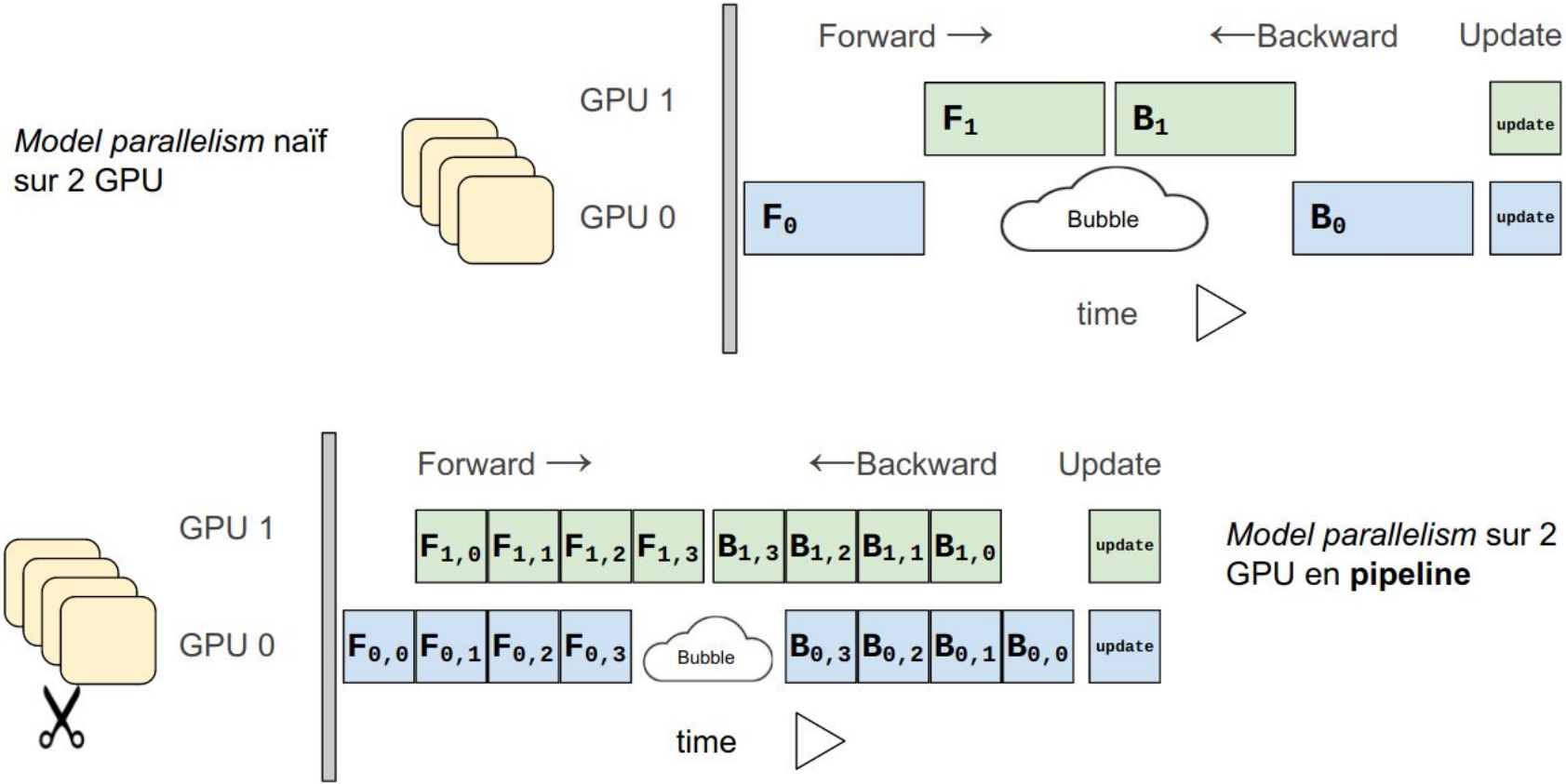


Model Parallelisms

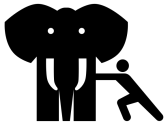




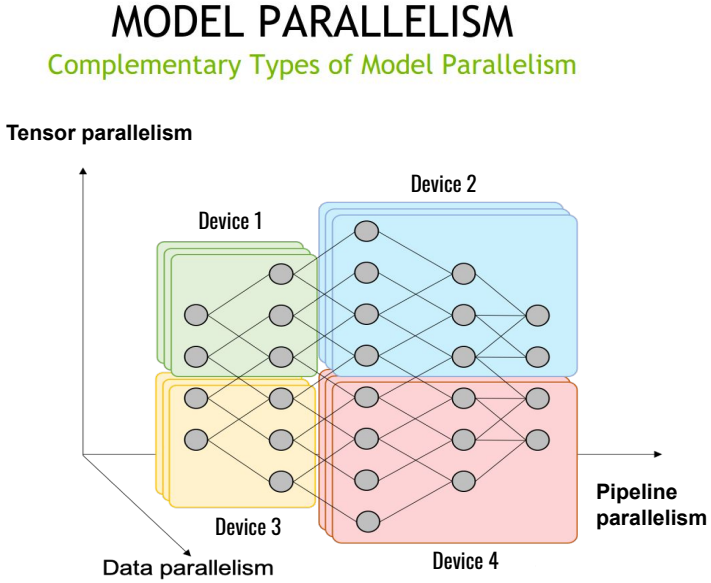
Pipeline Parallelism



Megatron-LM



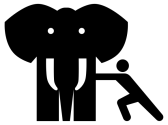
Nvidia optimized architecture of transformer based models such as GPT, BERT, and T5.



Model size	Hidden size	Number of layers	Number of parameters (billion)	Model-parallel size	Number of GPUs	Batch size	Achieved teraFlOPs per GPU	Percentage of theoretical peak FLOPs	Achieved aggregate petaFLOPs
1.7B	2304	24	1.7	1	32	512	137	44%	4.4
3.6B	3072	30	3.6	2	64	512	138	44%	8.8
7.5B	4096	36	7.5	4	128	512	142	46%	18.2
18B	6144	40	18.4	8	256	1024	135	43%	34.6
39B	8192	48	39.1	16	512	1536	138	44%	70.8
76B	10240	60	76.1	32	1024	1792	140	45%	143.8
145B	12288	80	145.6	64	1536	2304	148	47%	227.1
310B	16384	96	310.1	128	1920	2160	155	50%	297.4
530B	20480	105	529.6	280	2520	2520	163	52%	410.2
1T	25600	128	1008.0	512	3072	3072	163	52%	502.0

- TP for intranode parallelism
- PP for internode parallelism

Deepspeed



Model Scale

Support 200B

Toward 100 Trillion

Speed

Up to 10x faster

Scalability

Superlinear speedup

Usability

Few lines of code changes

```
# Include DeepSpeed configuration arguments
parser = deepspeed.add_config_arguments(parser)
```

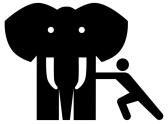
```
# Initialize DeepSpeed to use the following features
# 1) Distributed model
# 2) DeepSpeed optimizer
model_engine, optimizer, _, _ = deepspeed.initialize(
                                    args=args, model=model,
                                    model_parameters=parameters,
                                    optimizer=optimizer)
```

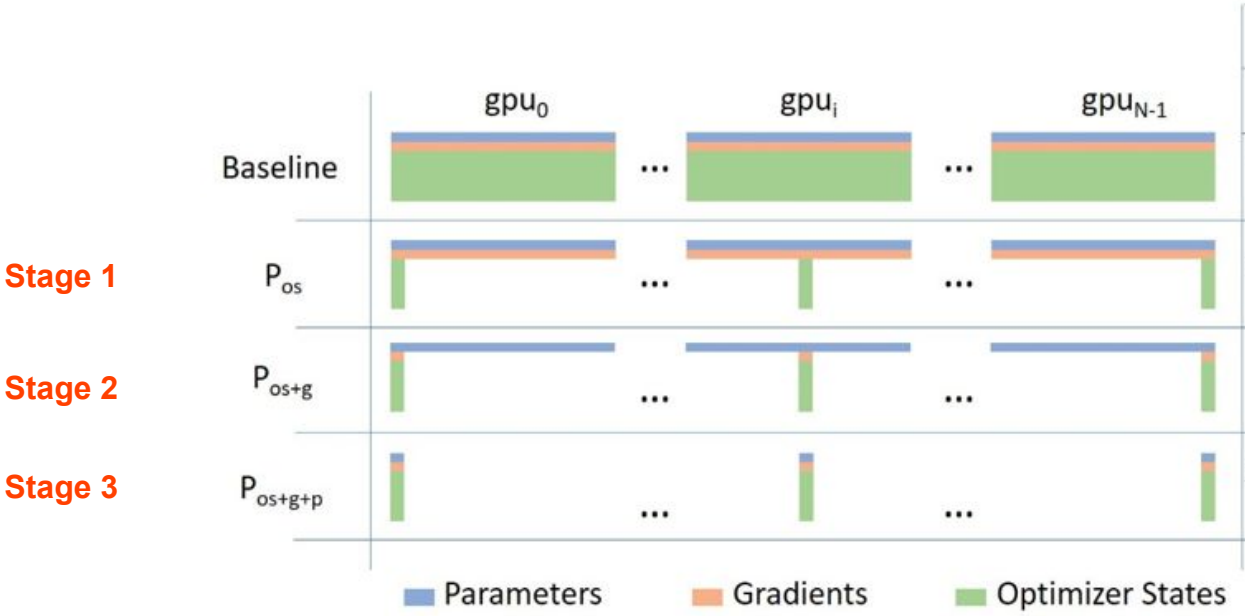
```
for step, batch in enumerate(data_loader):
    #forward() method
    loss = model_engine(batch)

    #runs backpropagation
    model_engine.backward(loss)

    #weight update
    model_engine.step()
```

```
{
   "zero_optimization": {
      "stage":2,
      "contiguous_gradients": true,
      "overlap_comm": true,
      "reduce_scatter": true,
      "reduce_bucket_size": 5e8,
      "allgather_bucket_size": 5e8
   }
}
```

```
# SLURM Job submission
srun train.py -b 28 -s 200 --image-size 288
--deepspeed --deepspeed_config
ds_config_zero2.json
```

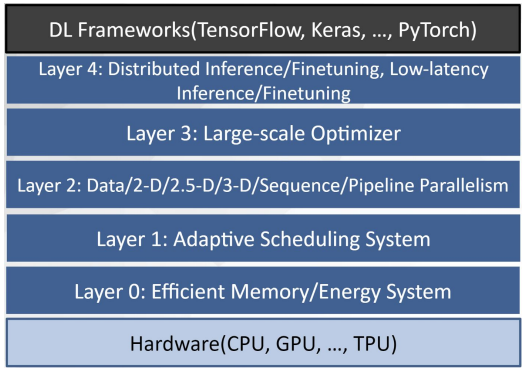
ZeRO - Data Parallelism Optimization

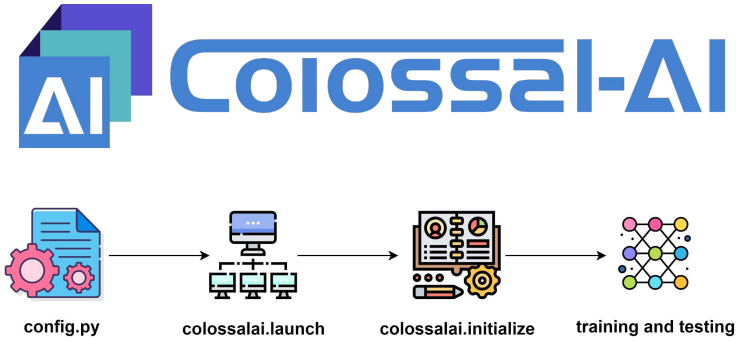




		Memory Consumption		Comm Volume
		Formulation	Specific Example K=12 Ψ=7.5B N_d=64	
	Baseline	$(2+2+K)*\Psi$	120GB	1x
Stage 1	P_{os}	$2\Psi+2\Psi+\frac{K*\Psi}{N_d}$	31.4GB	1x
Stage 2	P_{os+g}	$2\Psi+\frac{(2+K)*\Psi}{N_d}$	16.6GB	1x
Stage 3	P_{os+g+p}	$\frac{(2+2+K)*\Psi}{N_d}$	1.9GB	1.5x

Colossal-AI：A Unified Deep Learning System For Large-Scale Parallel Training





- Tensor Parallelism 1D, 2D, 2.5D, 3D

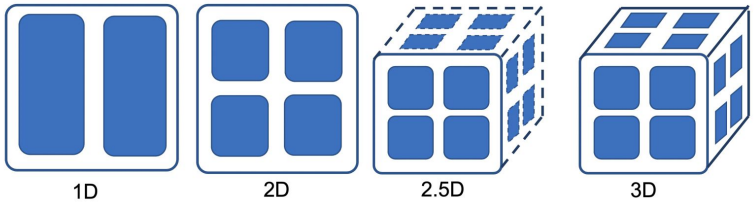
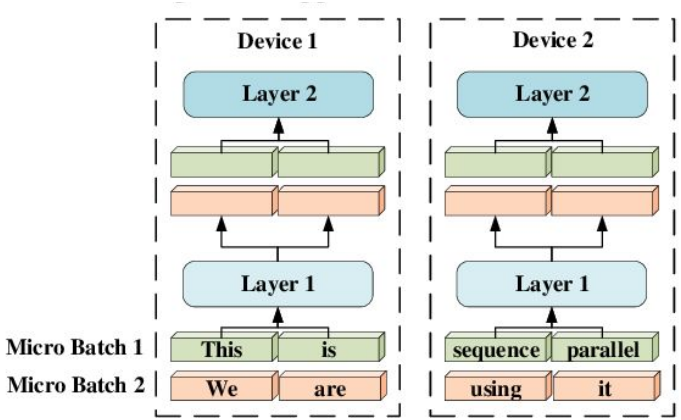


Figure 1: Tensor parallellism including 1D, 2D, 2.5D and 3D tensor splitting

- Sequence Parallelism



Conclusion

- Pay attention to **Dataloader**. It is often the bottleneck. Profile it, Optimize it, Boost I/O !
- **For usual size model**, use Data Parallelism, Mixed Precision, Large Batch optimizations, on 1 or few nodes.
- **For huge model**, use dedicated library like Megatron-LM, Deepspeed, Colossal-AI on several nodes.