



Deep Learning Optimisé - Jean Zay

Les optimisations des gros modèles



INSTITUT DU
DÉVELOPPEMENT ET DES
RESSOURCES EN
INFORMATIQUE
SCIENTIFIQUE



Les très gros modèles

Transformers ◀

Vision Transformers ◀

CoAtNet ◀

Vision Transformers >> Resnet-152: 60M



Rank	Model	Top 1 Accuracy	↑ Top 5 Accuracy	Number of params	Extra Training Data	Paper	Code	Result	Year	Tags
1	CoAtNet-7	90.88%		2440M	✓	CoAtNet: Marrying Convolution and Attention for All Data Sizes			2021	Conv+Transformer JFT-3B
2	ViT-G/14	90.45%		1843M	✓	Scaling Vision Transformers			2021	Transformer JFT-3B
3	CoAtNet-6	90.45%		1470M	✓	CoAtNet: Marrying Convolution and Attention for All Data Sizes			2021	Conv+Transformer JFT-3B
4	V-MoE-15B (Every-2)	90.35%		14700M	✓	Scaling Vision with Sparse Mixture of Experts			2021	Transformer
5	SwinV2-G	90.17%			✓	Swin Transformer V2: Scaling Up Capacity and Resolution			2021	Transformer
6	Florence-CoSwin-H	90.05%	99.02%		✓	Florence: A New Foundation Model for Computer Vision			2021	Transformer
7	TokenLearner L/8 (24+11)	88.87%		460M	✓	TokenLearner: What Can 8 Learned Tokens Do for Images and Videos?			2021	Transformer JFT-300M
8	MViT-H, 512*2 (IN22K-pretrain)	88.8%		667M	✓	Improved Multiscale Vision Transformers for Classification and Detection			2021	Transformer ImageNet-22k MViT

Simple couche de réseaux de neurones



I ← I am trying to understand

am ← I am trying to understand

trying ← I am trying to understand

to	I	am	trying	to	understand
----	---	----	--------	----	------------

understand → I am trying to understand

Ajout des vecteurs positions à la séquences

Séquence de vecteurs (word embeddings)

Inputs

The big red dog

<start> Le gros chien rouge

<stop>

Output Probabilities

Simple classification sur tout le vocabulaire



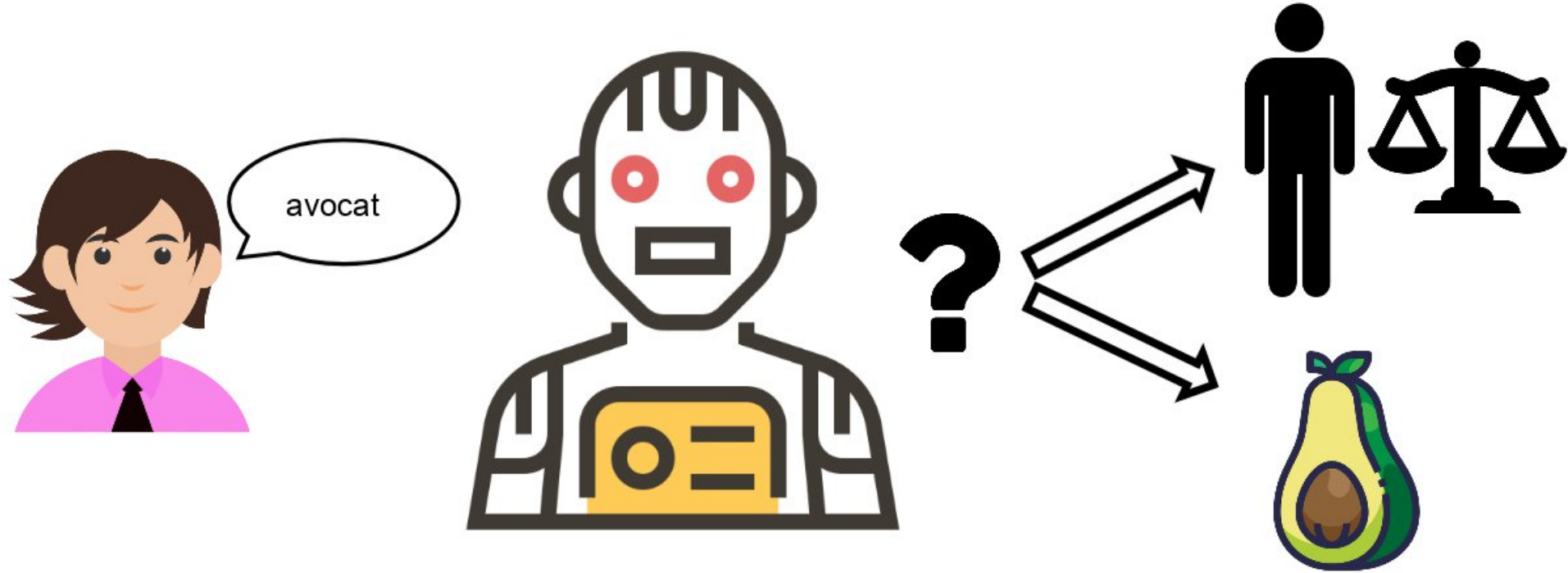
I ← I am trying to understand

am ← I am trying to understand

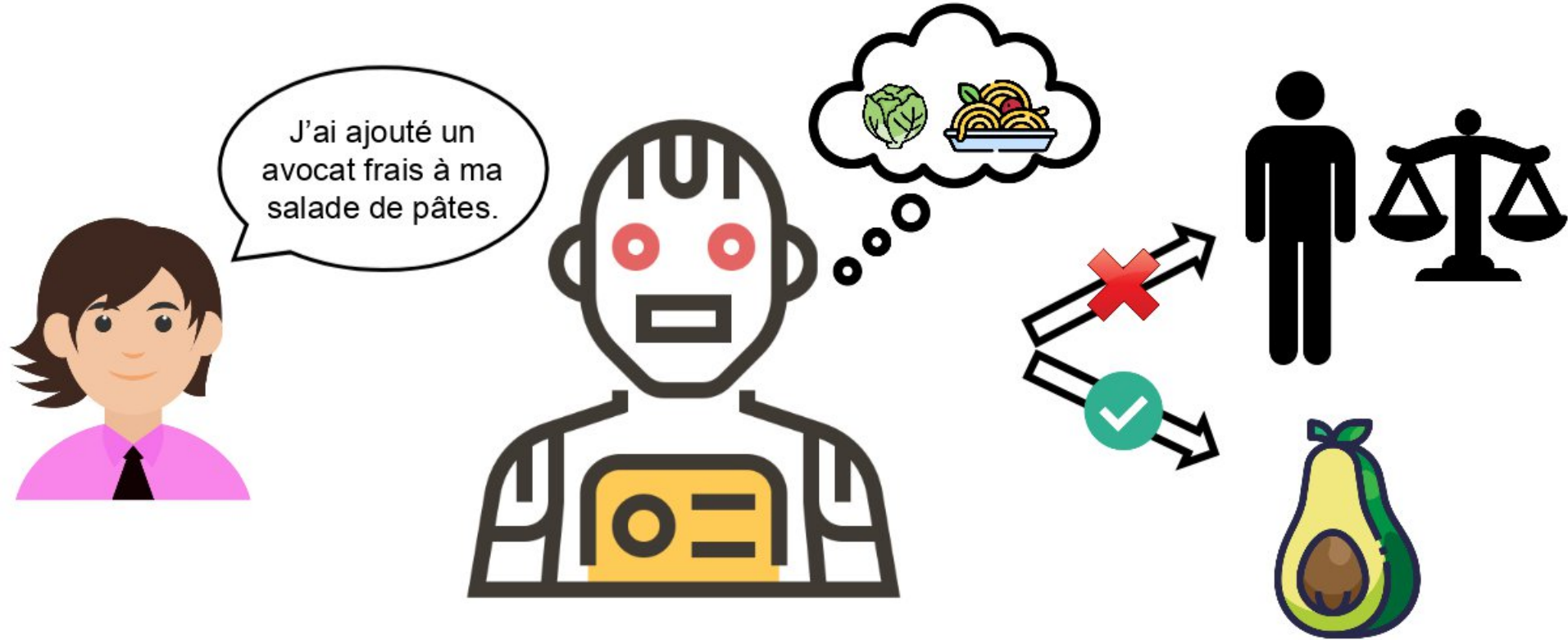
trying	←	I	am	trying	to	understand
--------	---	---	----	--------	----	------------

to I am trying to understand

understand ← I am trying to understand

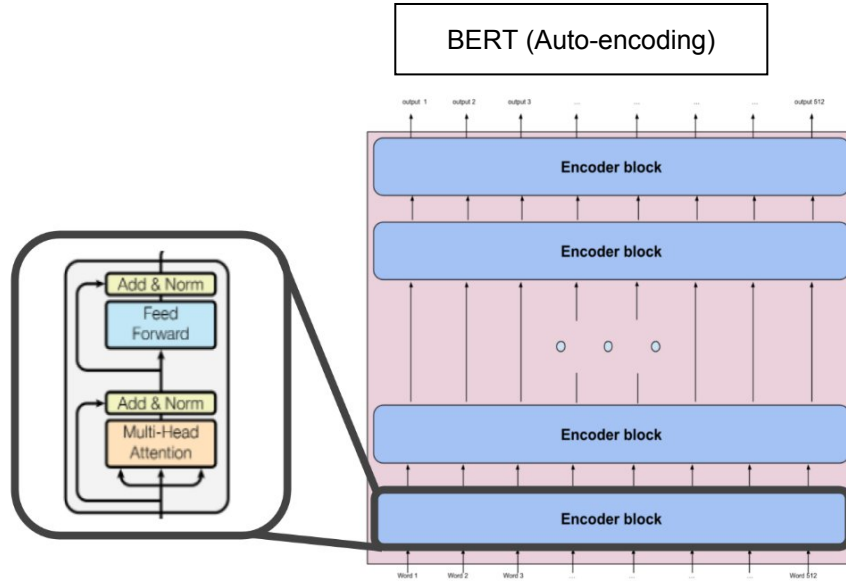


Mécanisme d'attention (intuition)

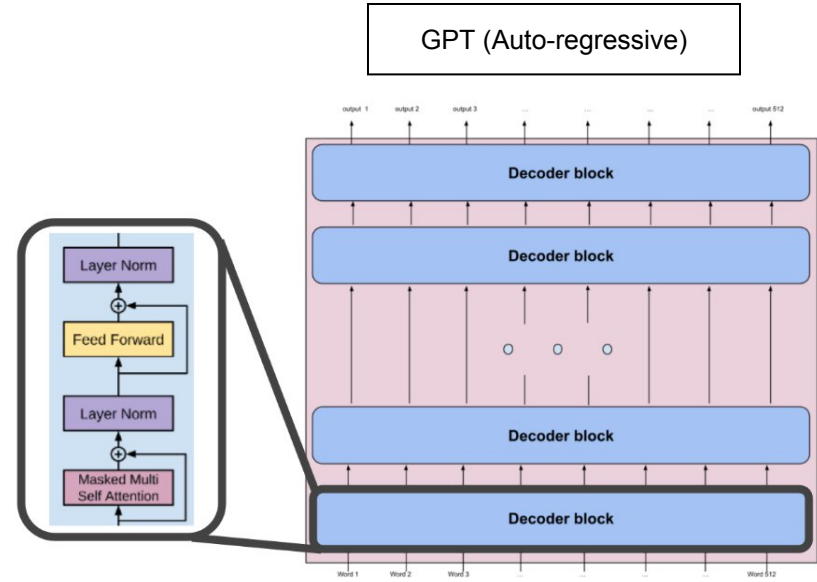


BERT vs GPT

BERT (Auto-encoding)



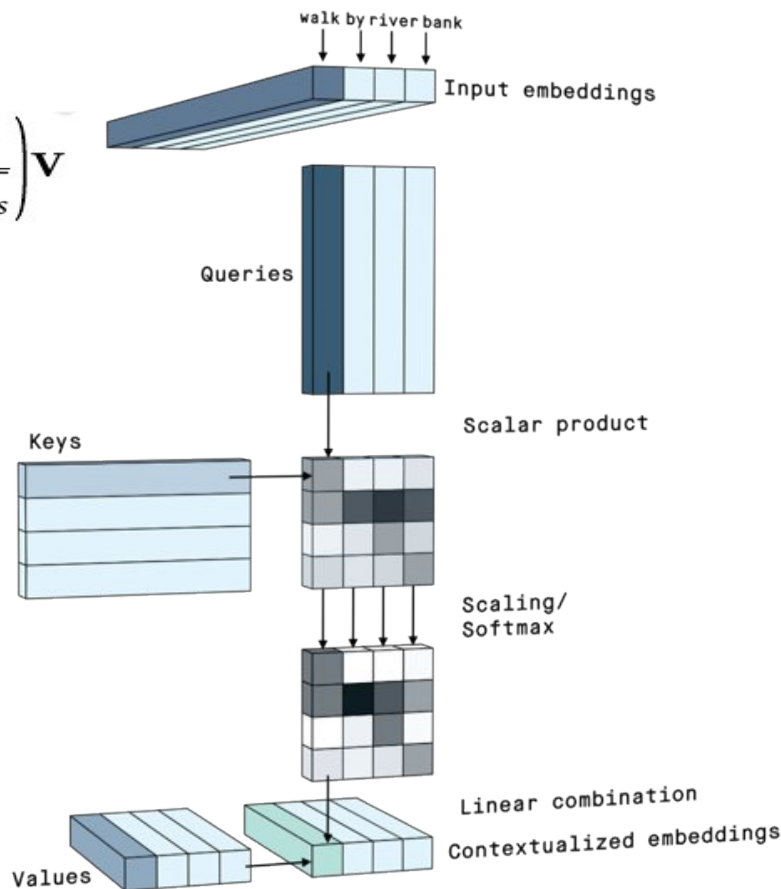
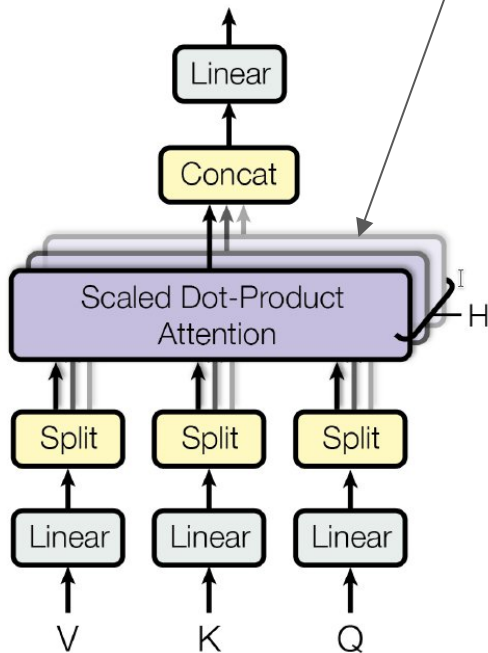
GPT (Auto-regressive)



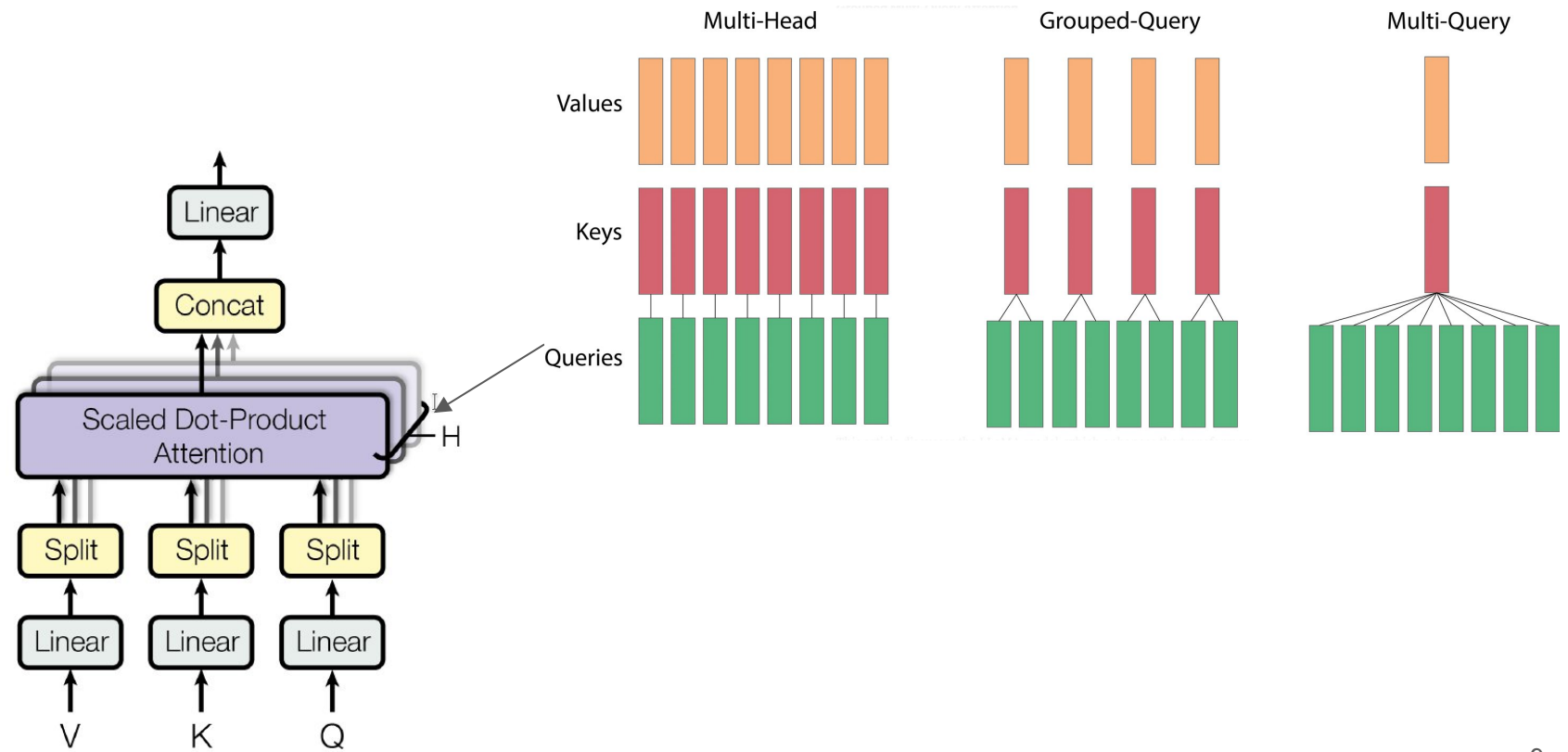
Le mécanisme de Self-Attention

Scaled Dot-Product Attention

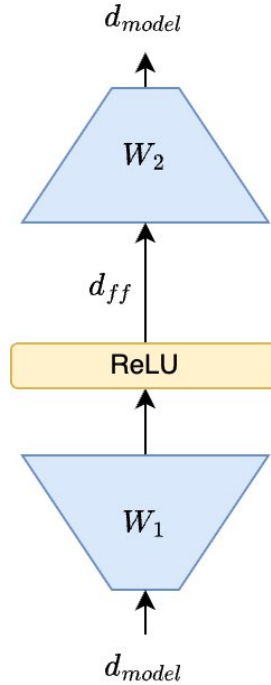
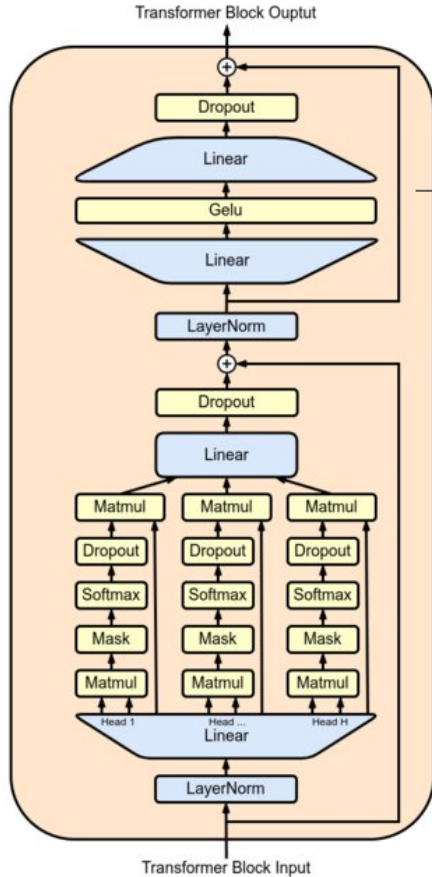
$$\text{Attention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{softmax}\left(\frac{\mathbf{Q}\mathbf{K}^T}{\sqrt{d_{\text{keys}}}}\right)\mathbf{V}$$



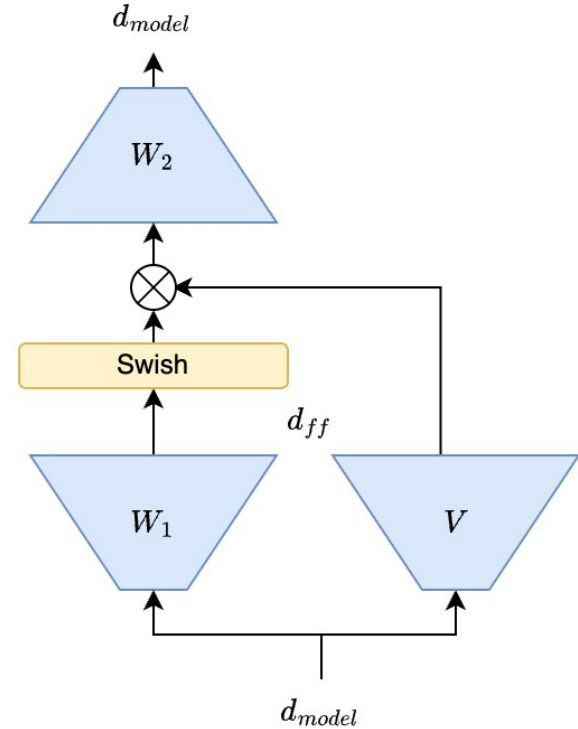
Le mécanisme de Multi-Head Attention



Le mécanisme de Feed Forward

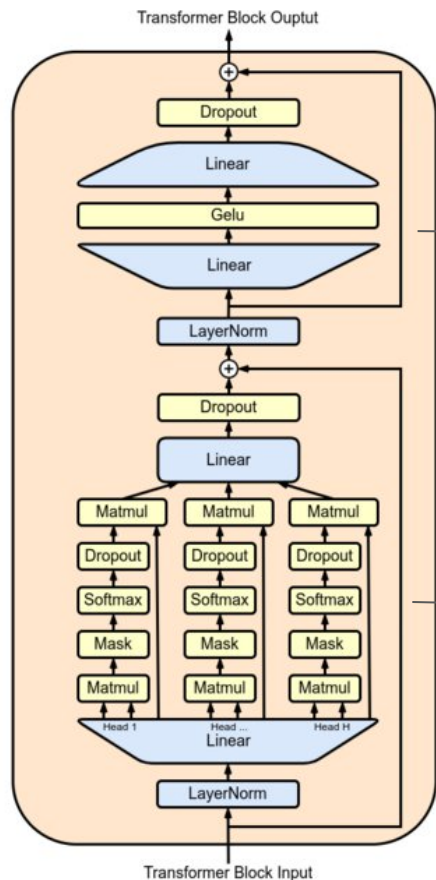


Original Feed Forward Layer



Feed Forward with SwiGLU

Exemple de gros modèles : Llama 3.3-70B



$$\begin{aligned}\text{output_proj} &= \text{vocab_size} \times \text{hidden_size} \\ &= 128256 \times 8192 \approx \mathbf{1.05B}\end{aligned}$$

$$\begin{aligned}\text{proj1} &: \text{hidden_size} \times \text{intermediate_size} = 8192 \times 28672 \\ \text{gate1} &: \text{hidden_size} \times \text{intermediate_size} = 8192 \times 28672 \\ \text{proj2} &: \text{intermediate_size} \times \text{hidden_size} = 28672 \times 8192 \\ 3 \times (8192 \times 28672) &\approx 704M \times 80 \text{ layers} \approx \mathbf{56B}\end{aligned}$$

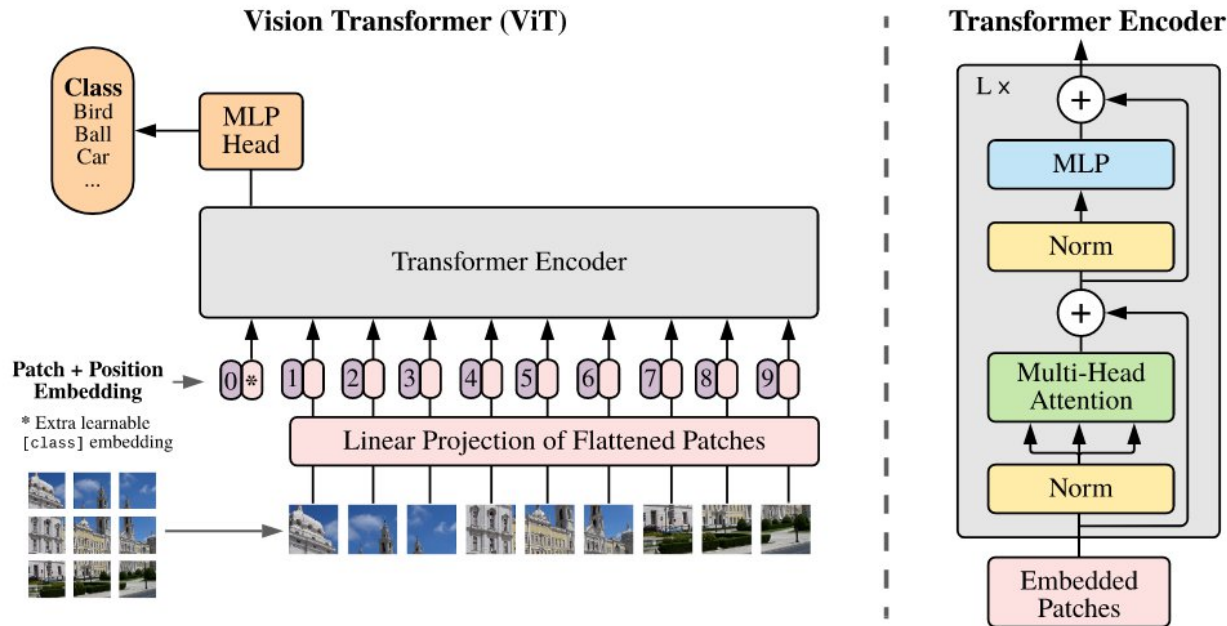
$$\begin{aligned}Q &: \text{hidden_size} \times \text{hidden_size} = 8192 \times 8192 \\ K, V &: \text{hidden_size} \times (\text{num_key_value_heads} \times \text{head_dim}) \\ &= 8192 \times (8 \times 128) = 8192 \times 1024 \\ \text{Output} &: \text{hidden_size} \times \text{hidden_size} = 8192 \times 8192 \\ Q &: 8192 \times 8192 = 67.1M \quad K: 8192 \times 1024 = 8.4M \\ V &: 8192 \times 1024 = 8.4M \quad \text{Output}: 8192 \times 8192 = 67.1M \\ \text{TotalAttention} &: \approx 151M \times 80 \text{ layers} \approx \mathbf{12B}\end{aligned}$$

$$\begin{aligned}\text{token_embedding} &= \text{vocab_size} \times \text{hidden_size} \\ &= 128256 \times 8192 \approx \mathbf{1.05B}\end{aligned}$$

meta-llama/Llama-3.3-70B-Instruct

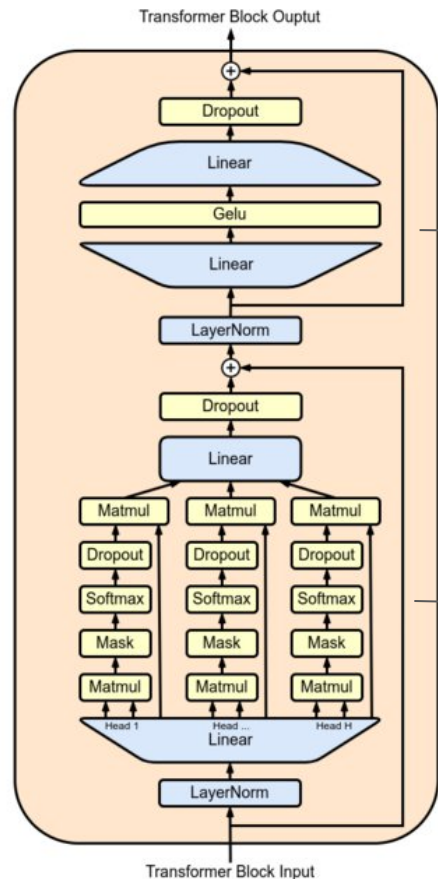
```
{
  "architectures": [
    "LlamaForCausalLM"
  ],
  "attention_bias": false,
  "attention_dropout": 0.0,
  "bos_token_id": 128000,
  "eos_token_id": [
    128001,
    128008,
    128009
  ],
  "head_dim": 128,
  "hidden_act": "silu",
  "hidden_size": 8192,
  "initializer_range": 0.02,
  "intermediate_size": 28672,
  "max_position_embeddings": 131072,
  "mlp_bias": false,
  "model_type": "llama",
  "num_attention_heads": 64,
  "num_hidden_layers": 80,
  "num_key_value_heads": 8,
  "pretraining_tp": 1,
  "rms_norm_eps": 1e-05,
  "rope_scaling": {
    "factor": 8.0,
    "high_freq_factor": 4.0,
    "low_freq_factor": 1.0,
    "original_max_position_embeddings": 8192,
    "rope_type": "llama3"
  },
  "rope_theta": 500000.0,
  "tie_word_embeddings": false,
  "torch_dtype": "bfloat16",
  "transformers_version": "4.47.0.dev0",
  "use_cache": true,
  "vocab_size": 128256
}
```

Vision Transformer (ViT)



- Images découpées en *patch*
- *Patches* séquencés avec un *Position embedding*
- Ajout d'un "classification token" pour réaliser la classification finale

Exemple de gros modèles : ViT-G-1.84B



proj1: $\text{hidden_size} \times \text{intermediate_size} = 1664 \times 8192$
proj2: $\text{intermediate_size} \times \text{hidden_size} = 8192 \times 1664$
 $2 \times (1664 \times 8192) \approx 27\text{M} \times 48 \text{ layers} \approx \mathbf{1.3B}$

Q, K, V: $\text{hidden_size} \times \text{hidden_size} = 1664 \times 1664$
Output: $\text{hidden_size} \times \text{hidden_size} = 1664 \times 1664$
Q, K, V: $\approx 2.7\text{M}$ Output: $\approx 2.7\text{M}$
TotalAttention: $\approx 11\text{M} \times 48 \text{ layers} \approx \mathbf{531M}$

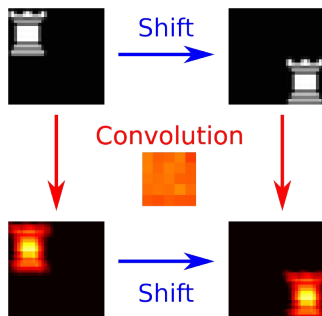
patch_embedding = $(\text{patch_size} \times \text{patch_size}) \times \text{hidden_size}$
= $(14 \times 14) \times 1664 \approx \mathbf{326K}$

```
"vision_config_dict": {  
  "hidden_act": "gelu",  
  "hidden_size": 1664,  
  "intermediate_size": 8192,  
  "num_attention_heads": 16,  
  "num_hidden_layers": 48,  
  "patch_size": 14,  
  "projection_dim": 1280  
}
```

“Marrying Convolution and Attention for All Data Sizes”

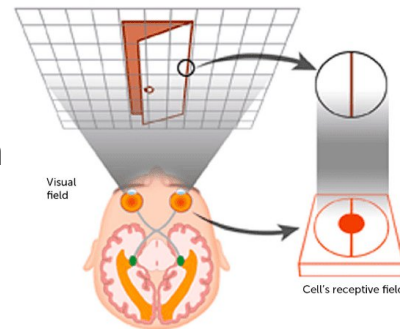
ConvNet

Translation Equivariance



Self-Attention Net

Global Receptive Field
Context comprehension



No patching !!

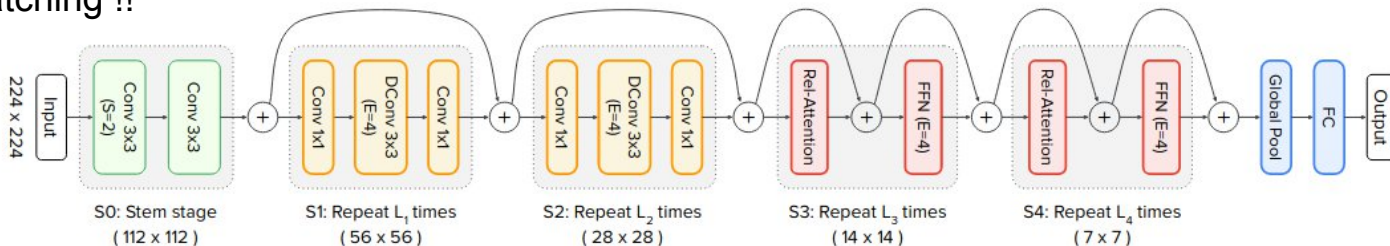


Figure 4: Overview of the proposed CoAtNet.

Models	Eval Size	#Params	#FLOPs	TPUv3-core-days	Top-1 Accuracy
ResNet + ViT-L/16	384 ²	330M	-	-	87.12
ViT-L/16	512 ²	307M	364B	0.68K	87.76
ViT-H/14	518 ²	632M	1021B	2.5K	88.55
NFNet-F4+	512 ²	527M	367B	1.86K	89.2
CoAtNet-3 [†]	384 ²	168M	114B	0.58K	88.52
CoAtNet-3 [†]	512 ²	168M	214B	0.58K	88.81
CoAtNet-4	512 ²	275M	361B	0.95K	89.11
CoAtNet-5	512 ²	688M	812B	1.82K	89.77
ViT-G/14	518 ²	1.84B	5160B	>30K [◇]	90.45
CoAtNet-6	512 ²	1.47B	1521B	6.6K	90.45
CoAtNet-7	512 ²	2.44B	2586B	20.1K	90.88



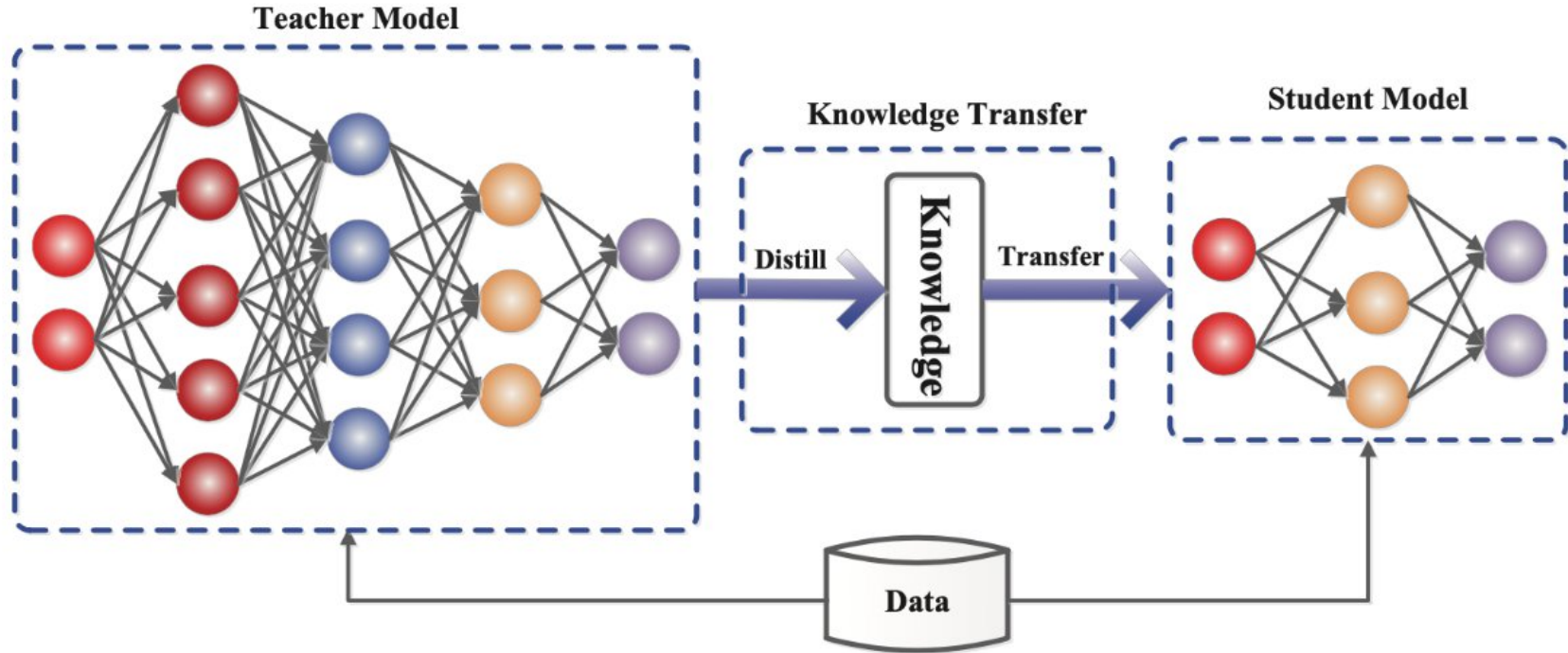
- Transforment la séquence entière (contrairement aux CNN et aux RNN)
- Possèdent un nombre conséquent de poids
- Nécessitent de très gros *datasets*

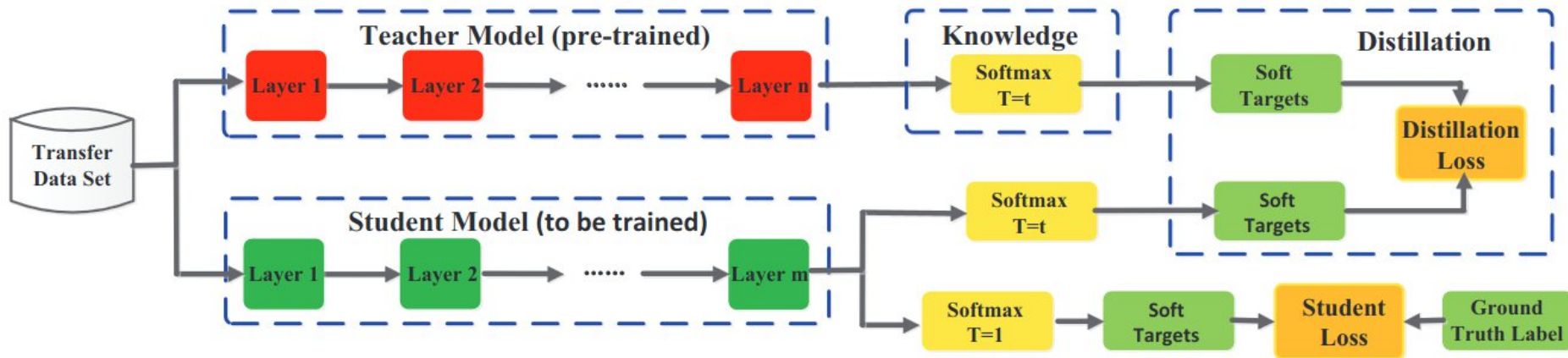
Optimisation de l'inférence des très gros modèles

Distillation ◀

Quantification ◀

Pruning ◀



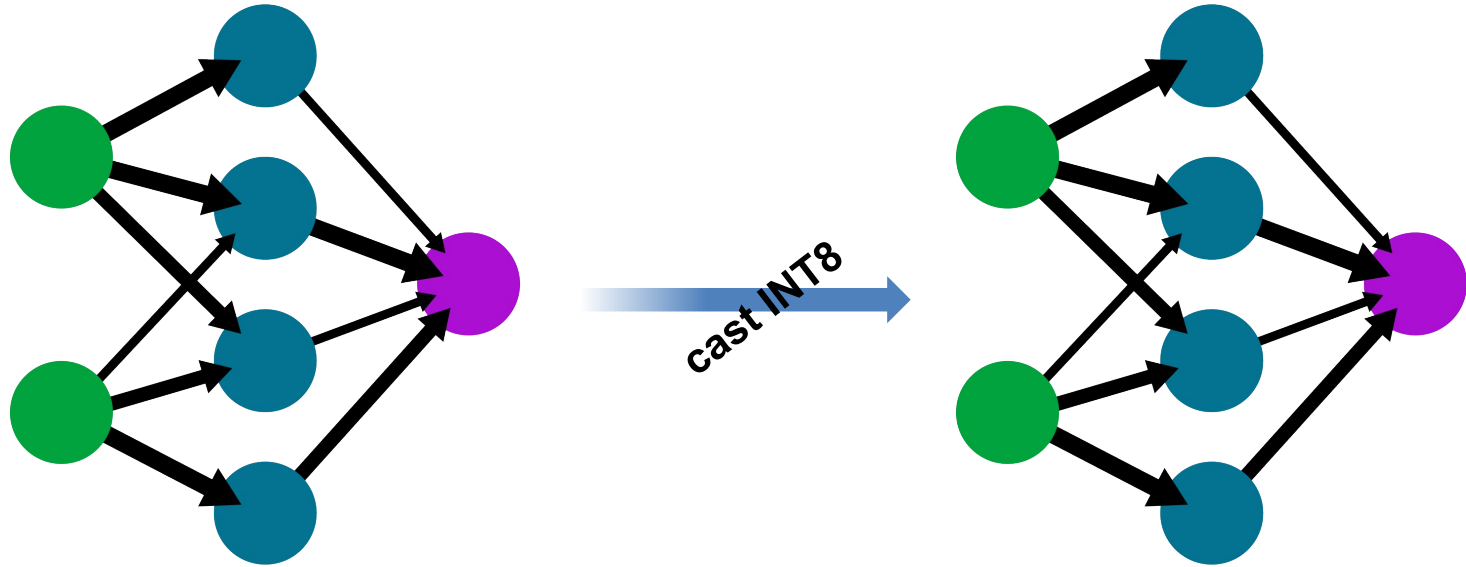


$$\mathcal{L}_{\text{tot}} = \mathcal{L}_{\text{distil}}(y_{\text{teacher}}, y_{\text{student}}) + \lambda \mathcal{L}_{\text{CE}}(y_{\text{target}}, y_{\text{student}})$$

Cross-entropy, Divergence KL, Wasserstein, ...

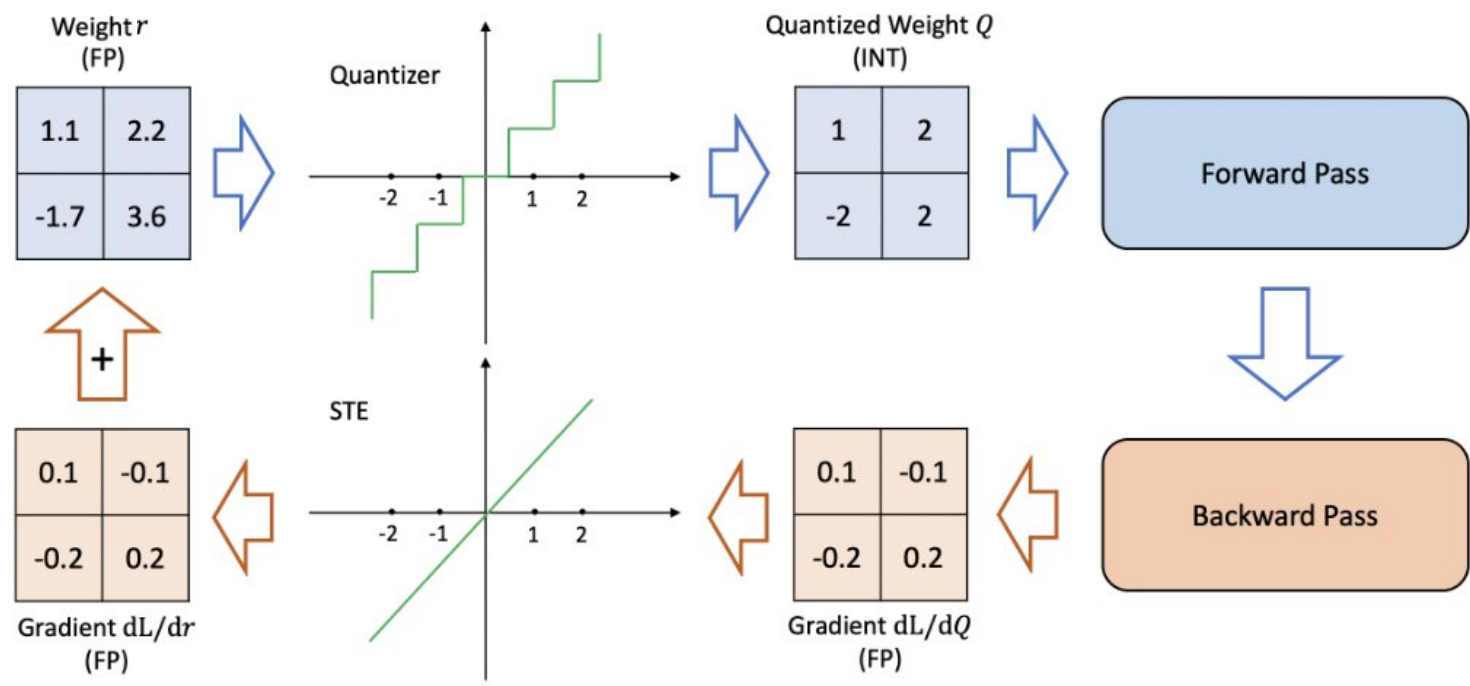
	A100 80 Go PCIe	A100 80 Go SXM
FP64	9,7 TFlops	
FP64 Tensor Core	19,5 TFlops	
FP32	19,5 TFlops	
Tensor Float 32 (TF32)	156 TFlops 312 TFlops*	
BFLOAT16 Tensor Core	312 TFlops 624 TFlops*	
FP16 Tensor Core	312 TFlops 624 TFlops*	
INT8 Tensor Core	624 TOPs 1248 TOPs*	

x2

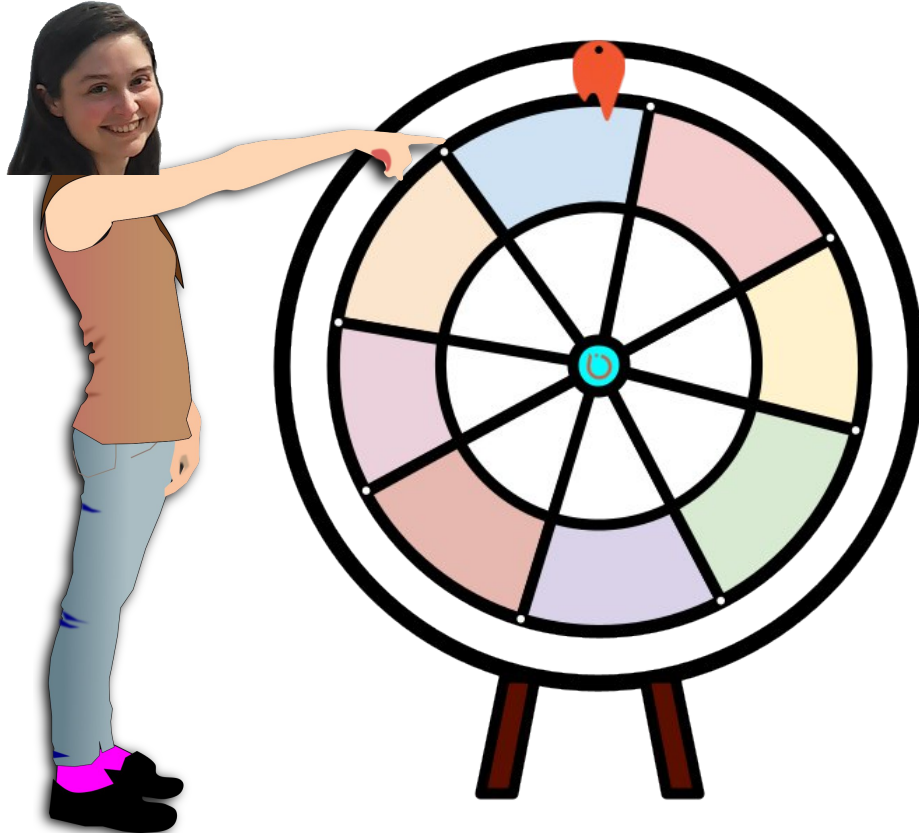


Il est possible de rencontrer une perte en performance

Quantification

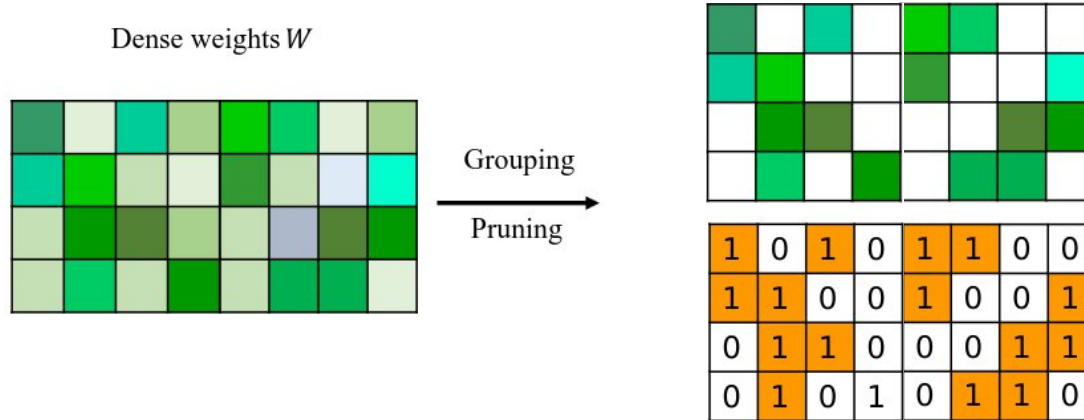


Exemple d'entraînement post-quantisation



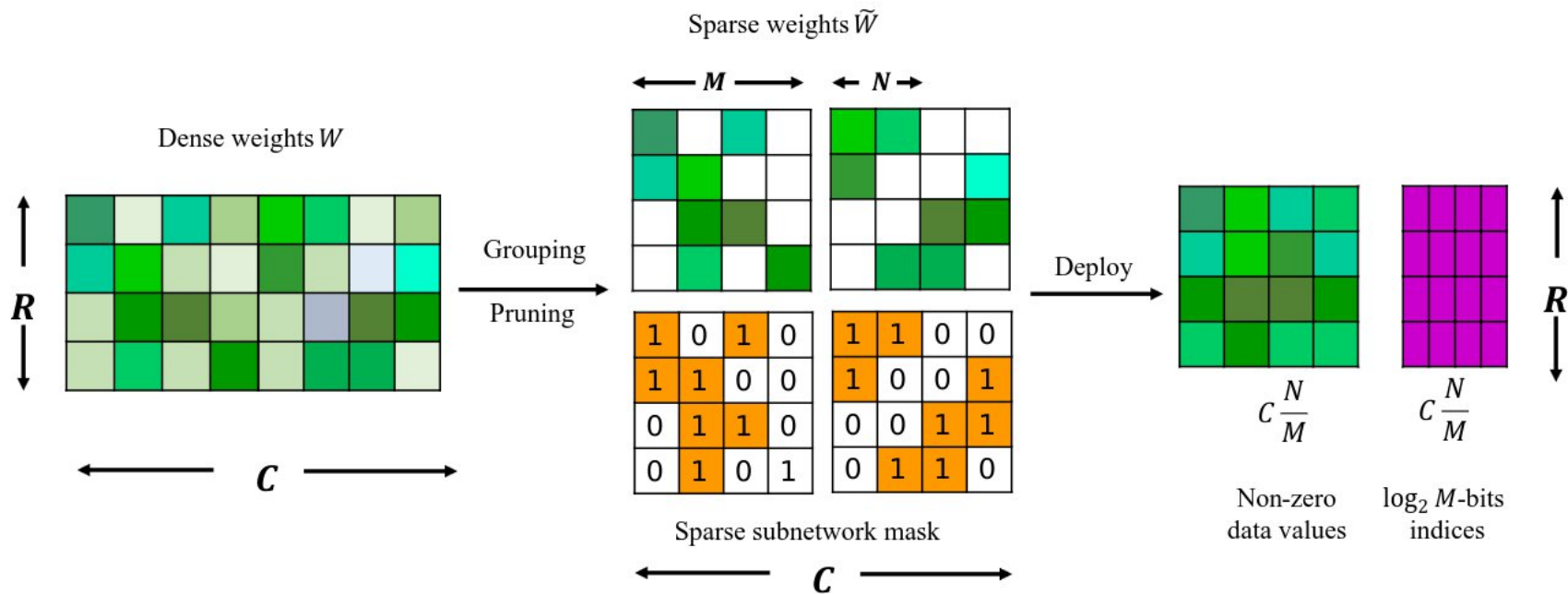
The Lottery Ticket Hypothesis.

A randomly-initialized, dense neural network contains a subnetwork that is initialized such that—when trained in isolation—it can match the test accuracy of the original network after training for at most the same number of iterations.



Les poids les plus petits sont mis à 0. Mais combien ?
Quel impact sur le temps de calcul ?

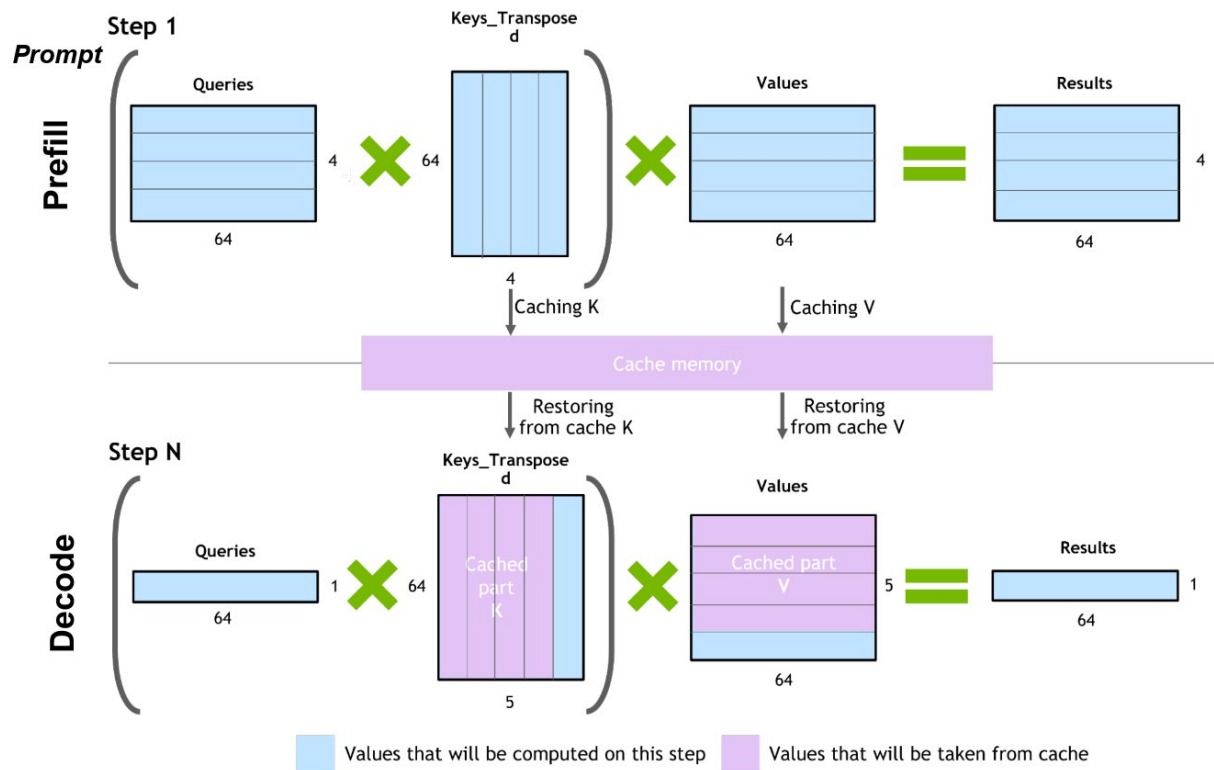
Pruning



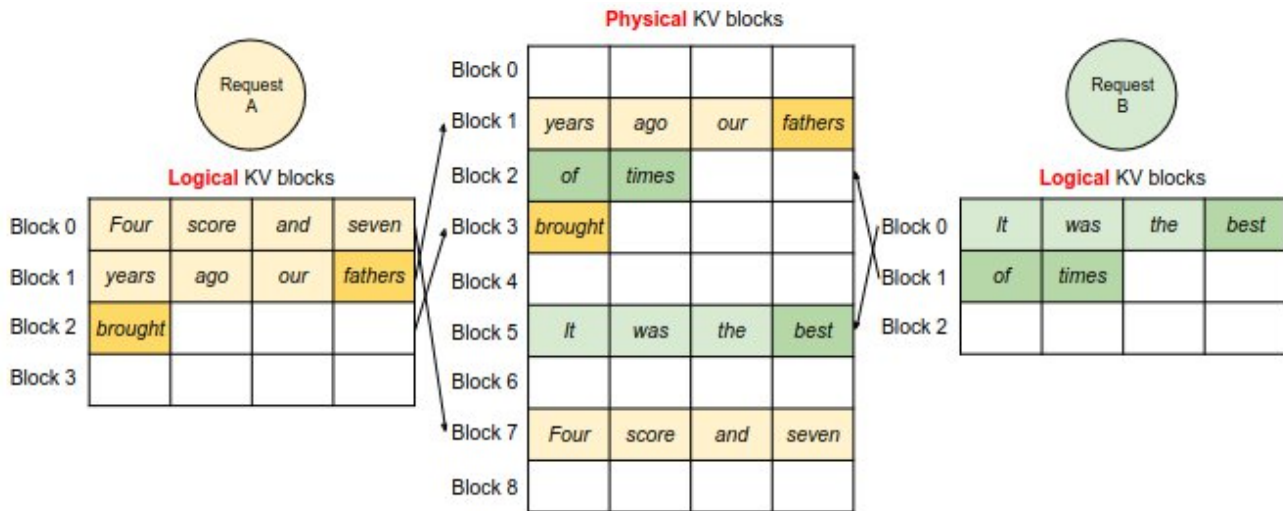
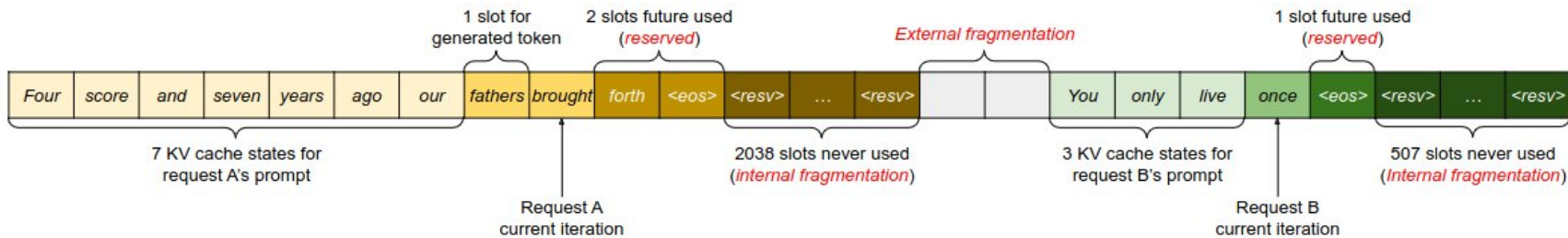
Les Tensor Cores des NVIDIA A100 supportent une dispersion 2:4.

KV Caching

$(Q * K^T) * V$ computation process with caching



Paged Attention



```
llm = LLM(model="facebook/opt-125m")
outputs = llm.generate(prompts, sampling_params)
```

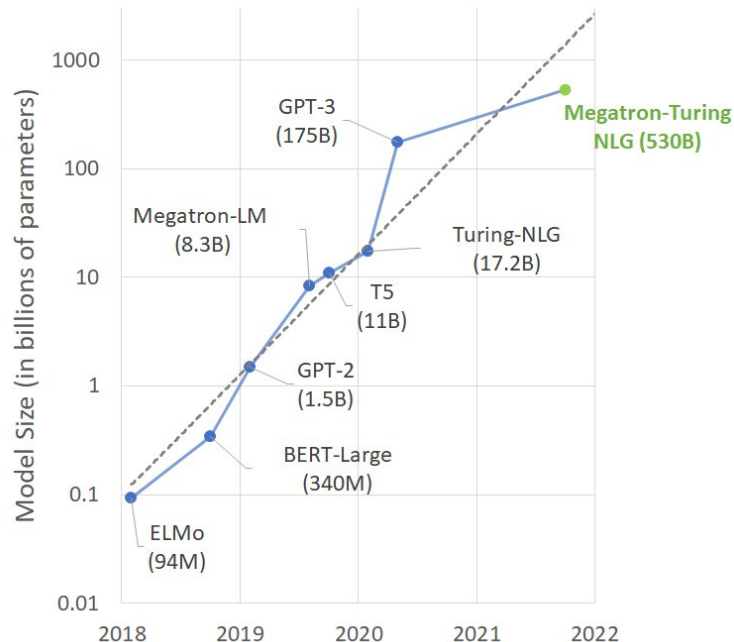
Optimisation du Data Parallelism pour les très gros modèles

Problématiques des gros modèles ◀

ZeRO & Fully Sharded Data Parallelism ◀

Énormes modèles > 1 Milliard de paramètres

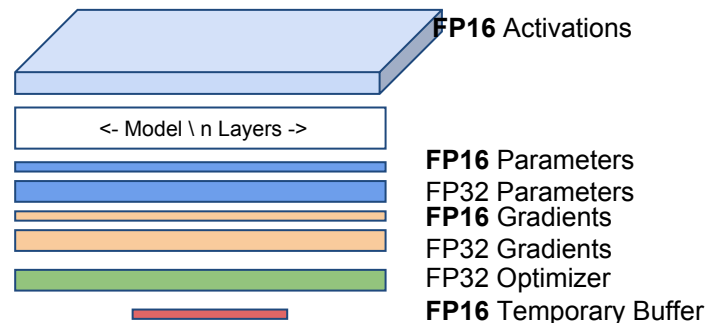
NLP Transformers

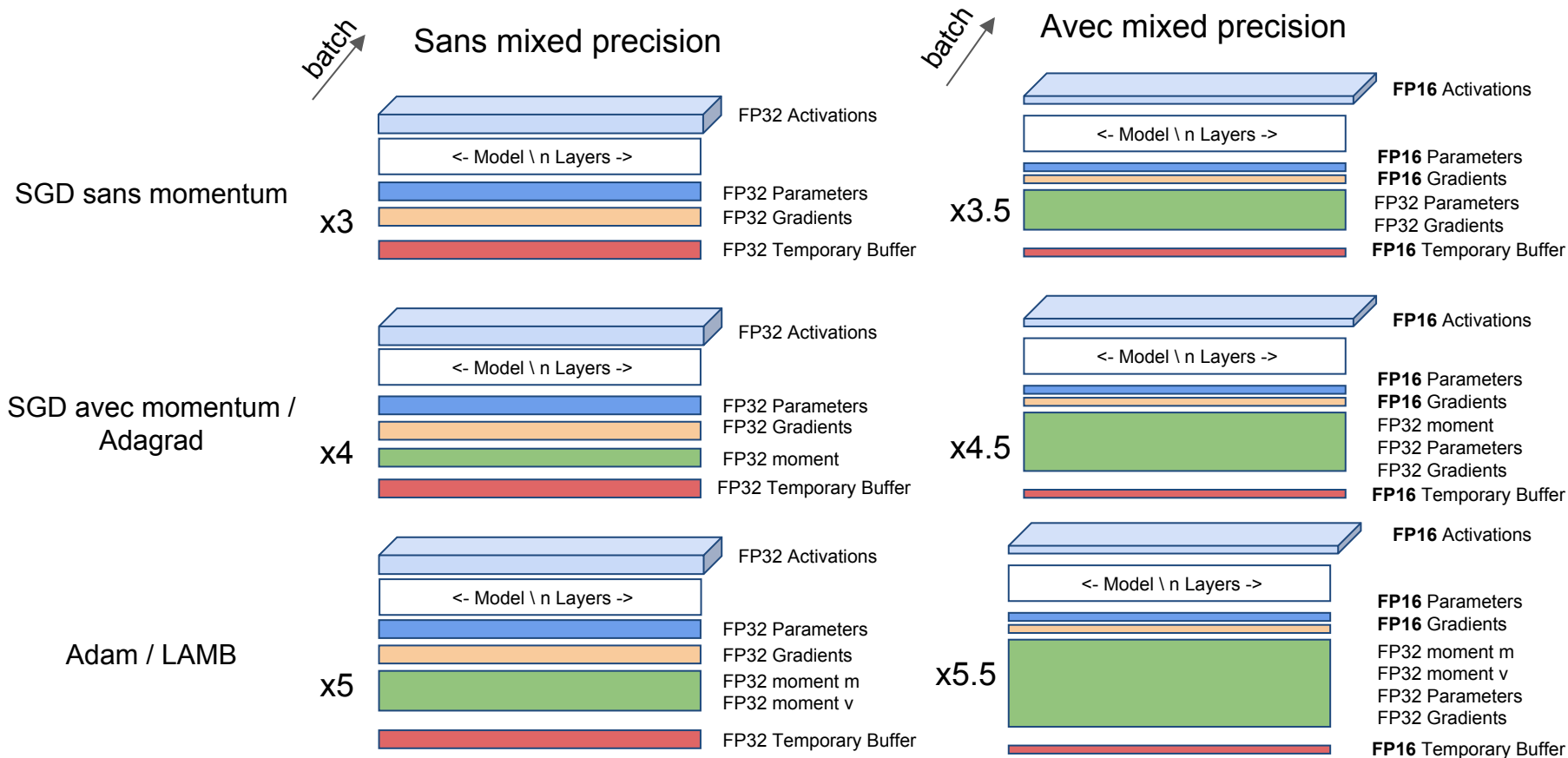


CUDA Out Of Memory

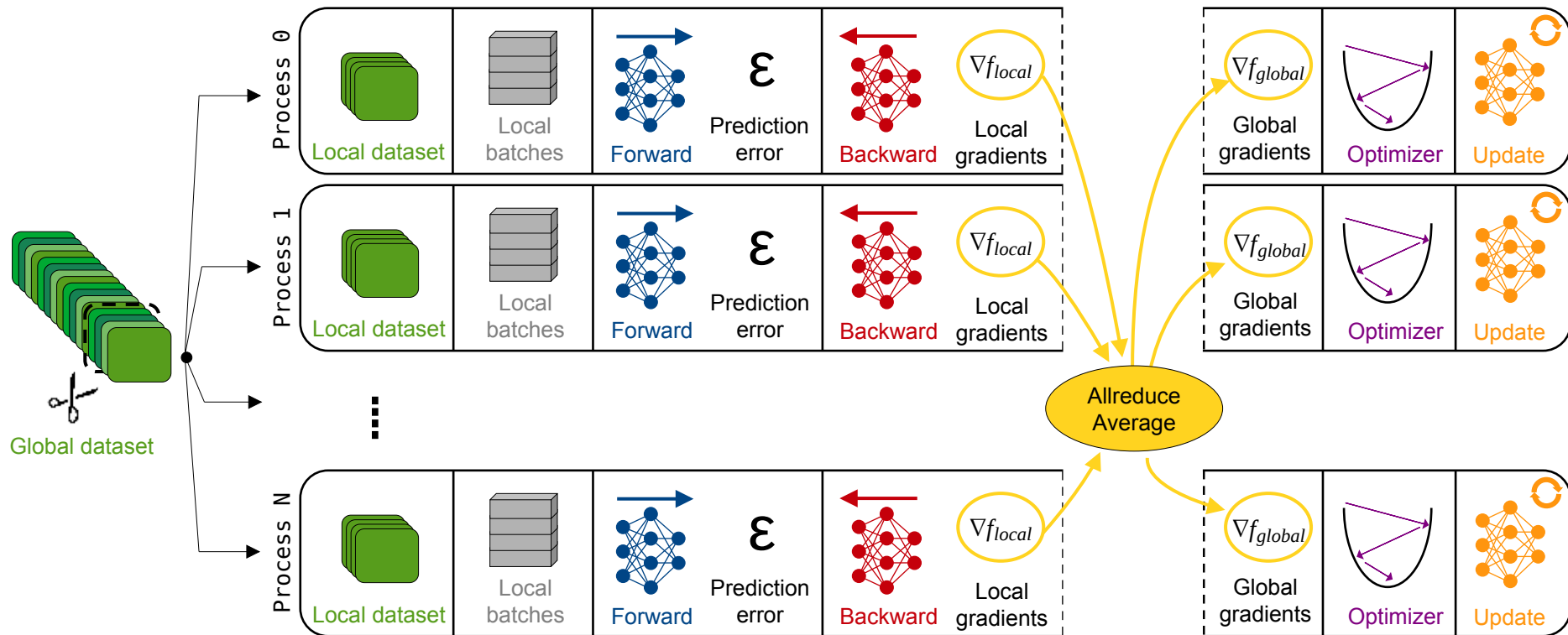
batch

~x4 huge
model



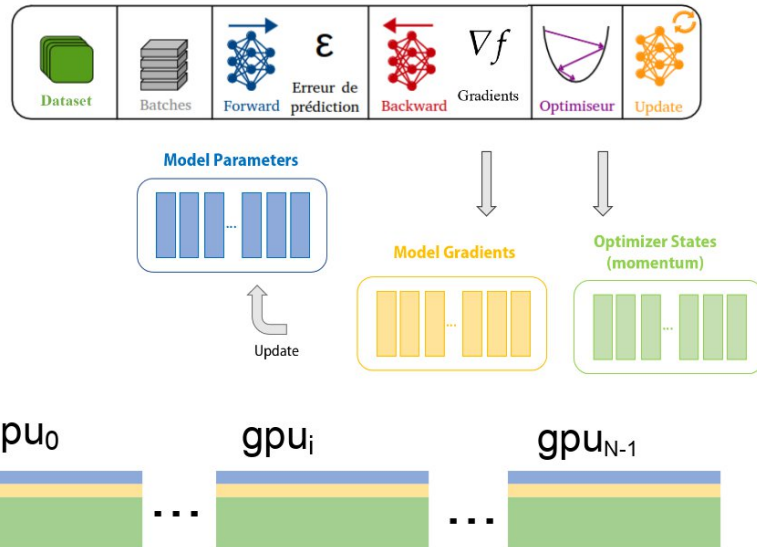
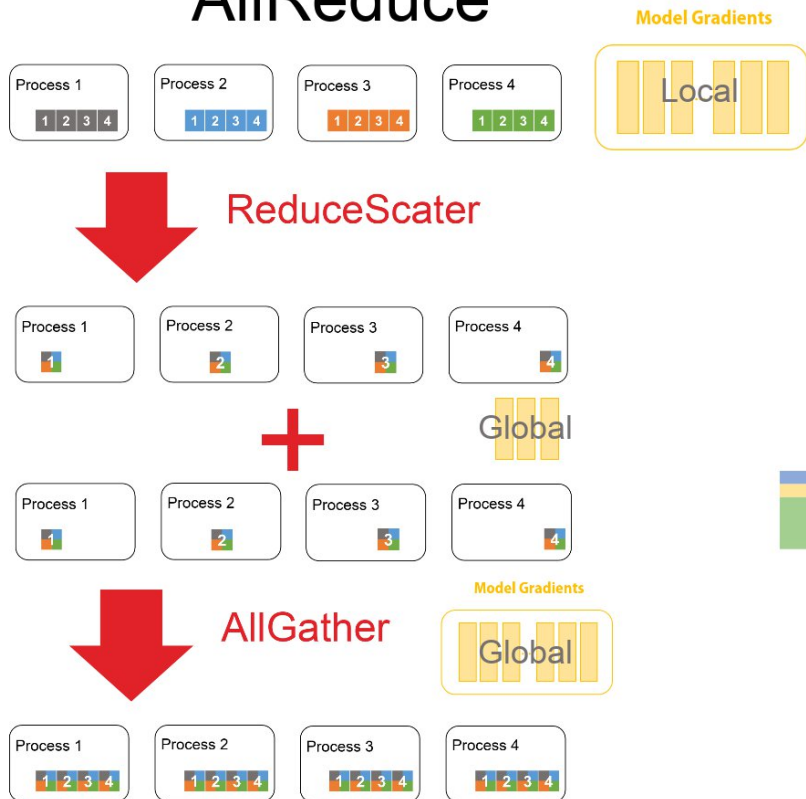


Rappel du Distributed Data Parallelism

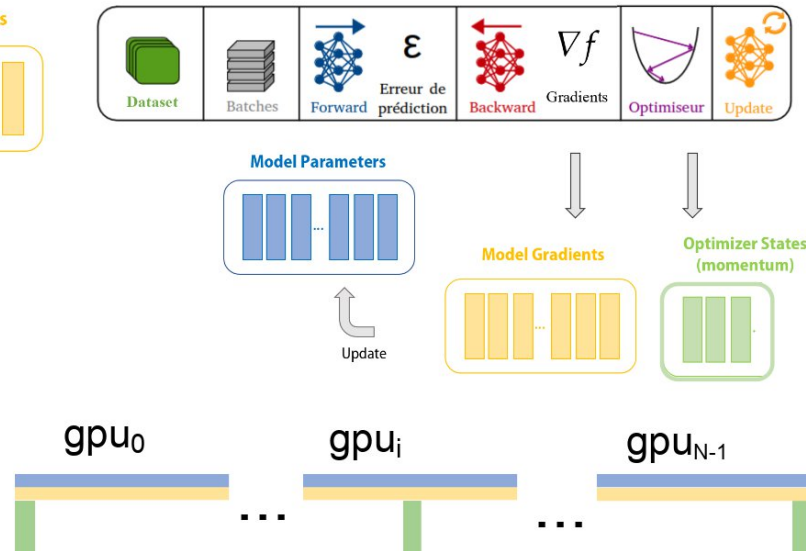
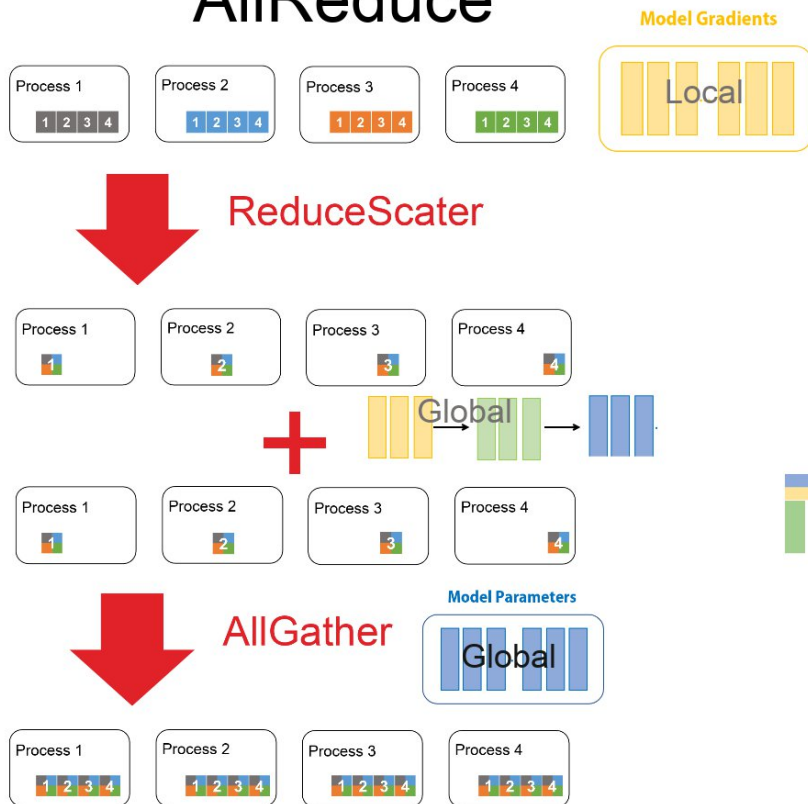


ZeRO-0 / DDP Baseline

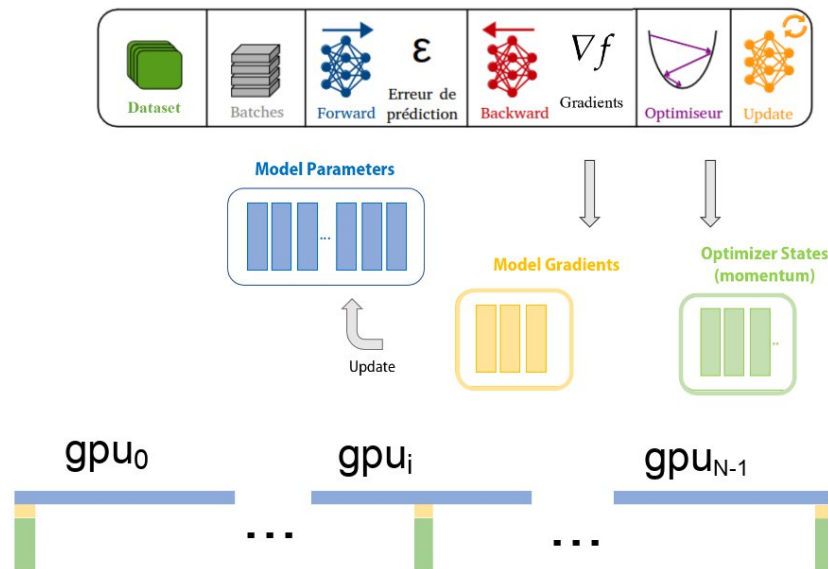
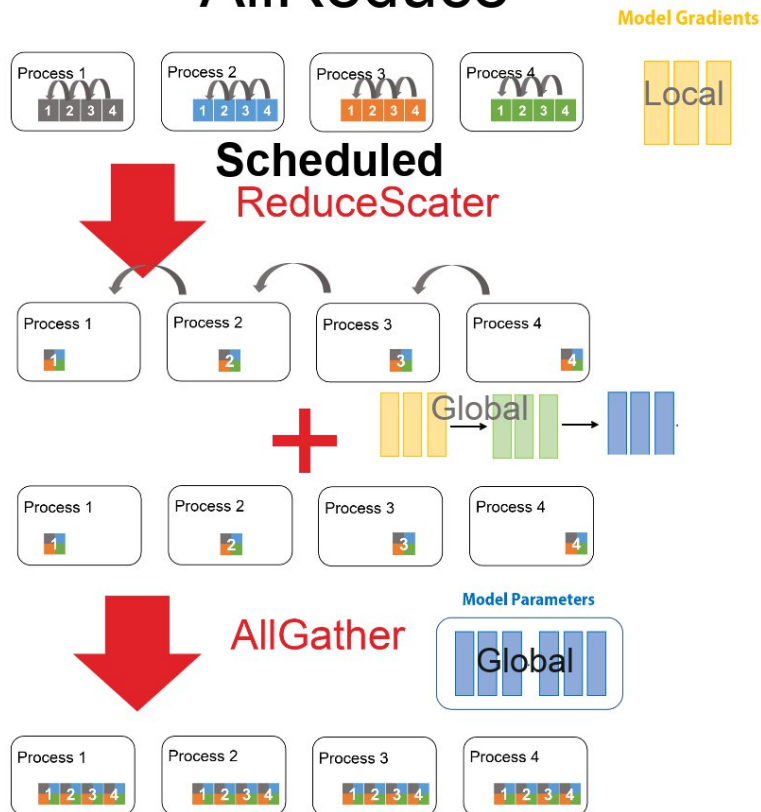
AllReduce



AllReduce



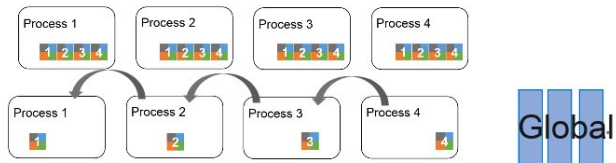
AllReduce



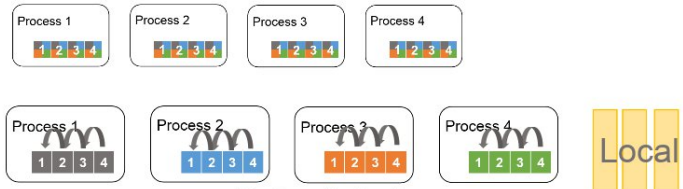
ZeRO-3 / FSDP



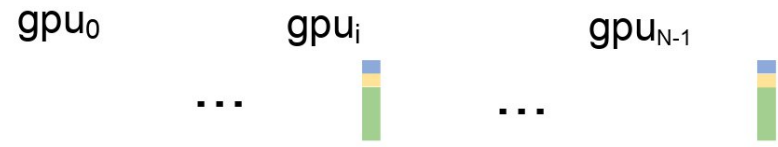
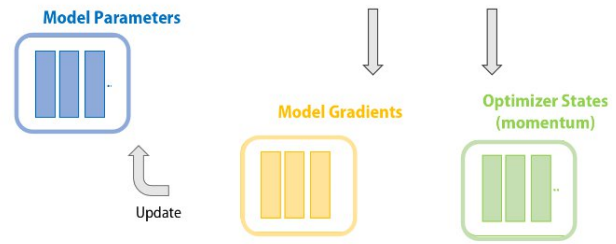
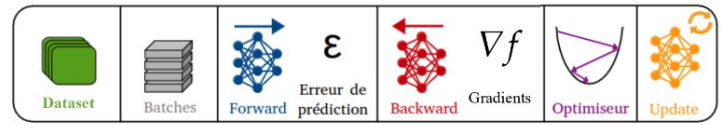
Scheduled AllGather
1. Forward



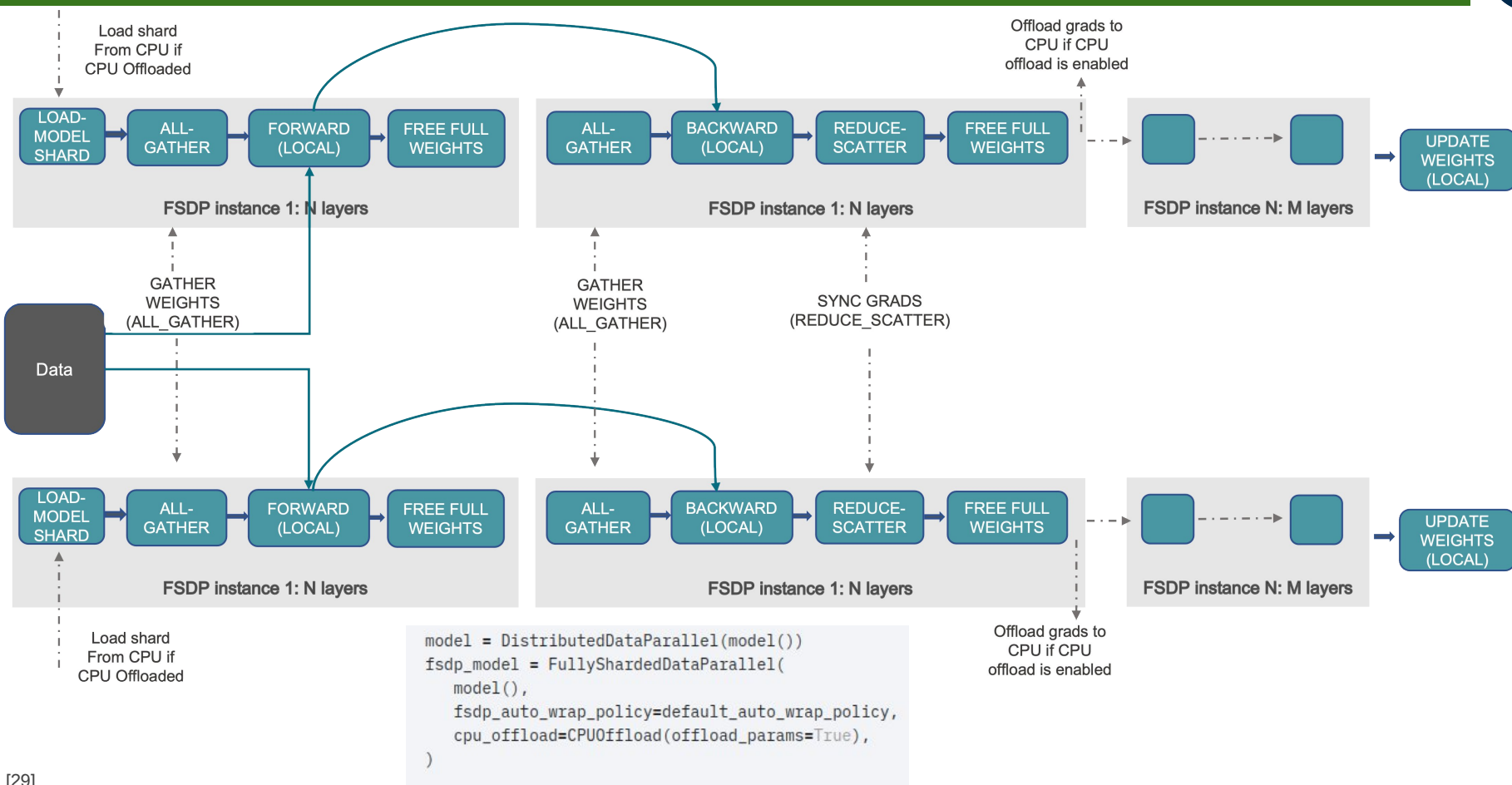
Scheduled AllGather
2. Backprop



Scheduled ReduceScater



Fully Sharded Data Parallel (Pytorch)





Allez dans le répertoire ***tp_fsdp*** dans le
répertoire ***Jour4*** du dépôt du cours et suivez le notebook.

- Limite du *Data Parallelism* avec Llama 3.2 3B
- Implémenter le Fully Sharded Data Parallelism
- Optimisation de la FSDP
- Utilisation de torch.compile sur un module FSDP

Optimisation de l'apprentissage des très gros modèles

Pipeline parallelism ◀

Tensor parallelism ◀

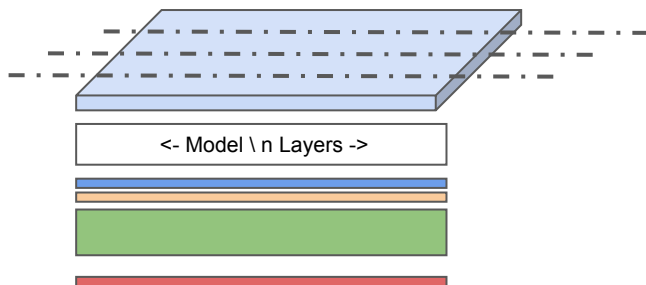
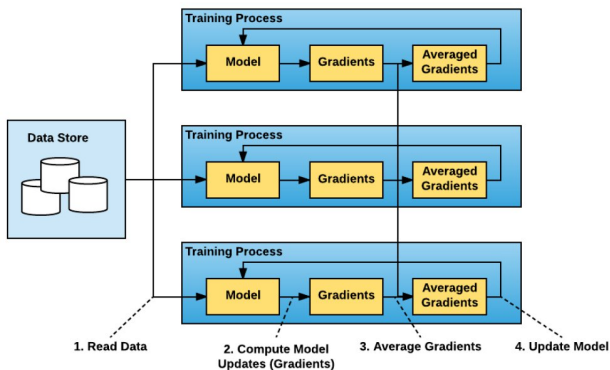
Hybrid parallelism ◀

3D parallelism ◀

Les différents parallélismes

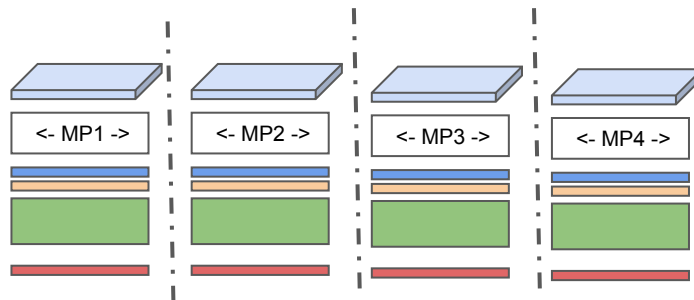
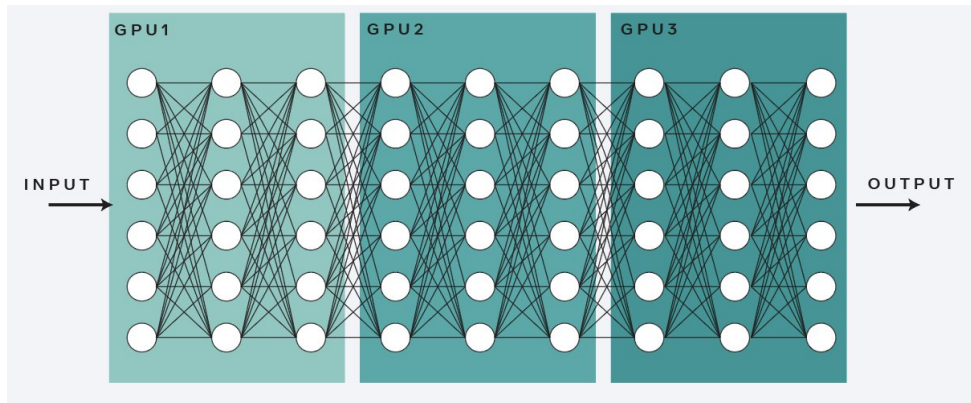
Data Parallelism

- Meilleur Throughput
- Seule l'empreinte mémoire des activations est distribuée
- Multi Processing



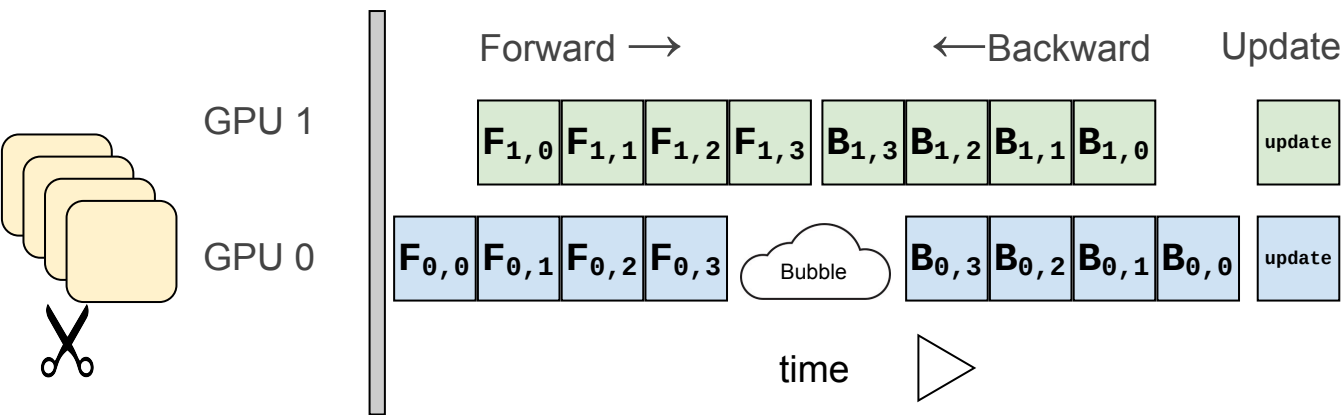
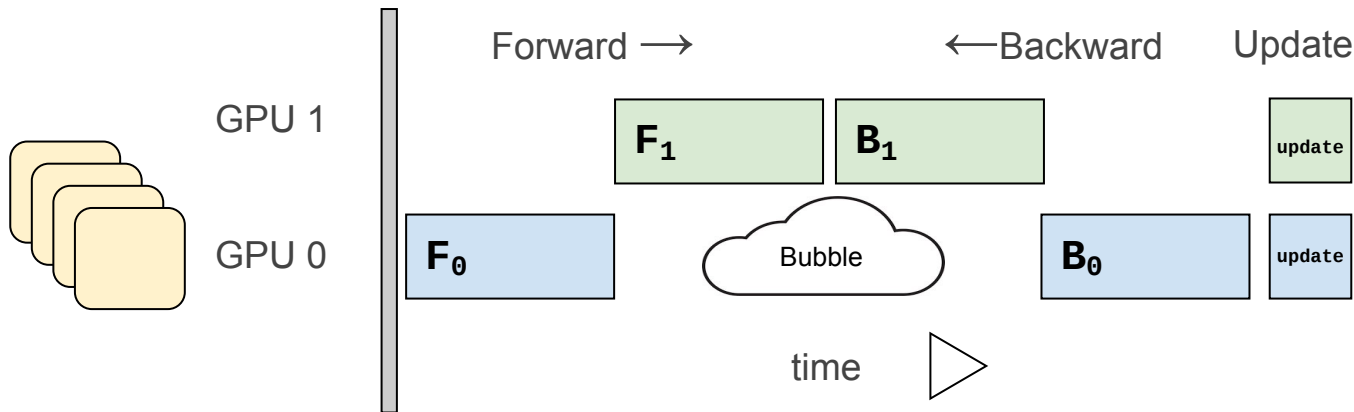
Pipeline Model Parallelism

- Empreinte mémoire distribuée
- Mono ou multi-processing



Pipeline Parallelism

Model parallelism naïf
sur 2 GPU



Model parallelism sur 2
GPU en **pipeline**

Pipeline Parallelism

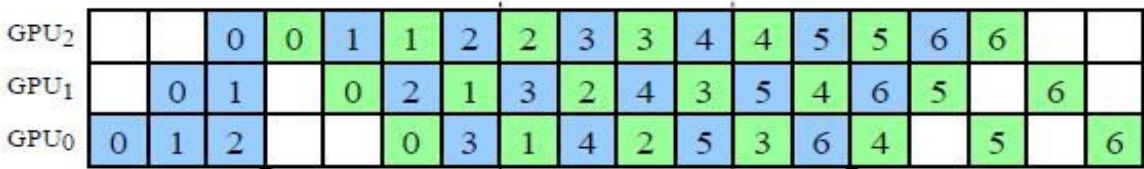
AFAB



(a) GPipe

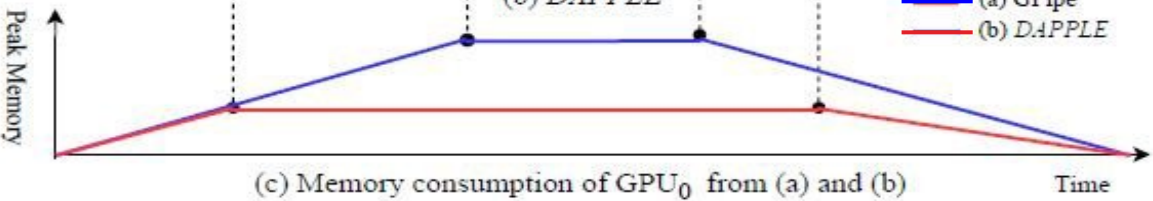
Forward Backward

1F1B



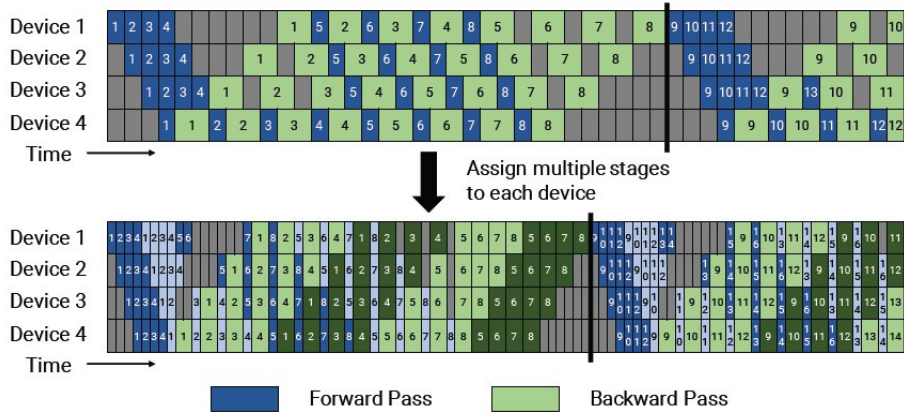
(b) DAPPLE

(a) GPipe
(b) DAPPLE



Pipeline Parallelism

Interleaved 1F1B



Zero Bubble

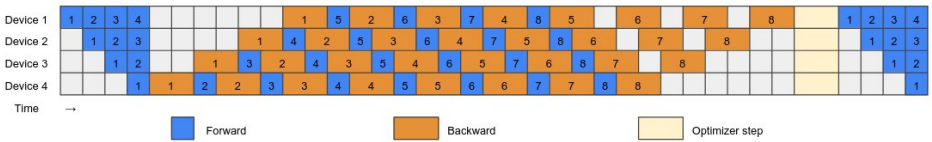


Figure 2: 1F1B pipeline schedule.

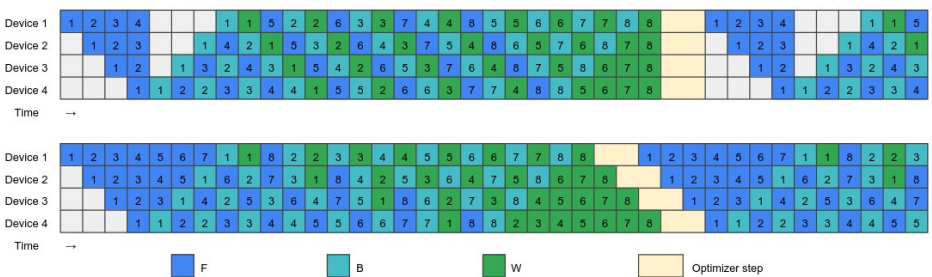
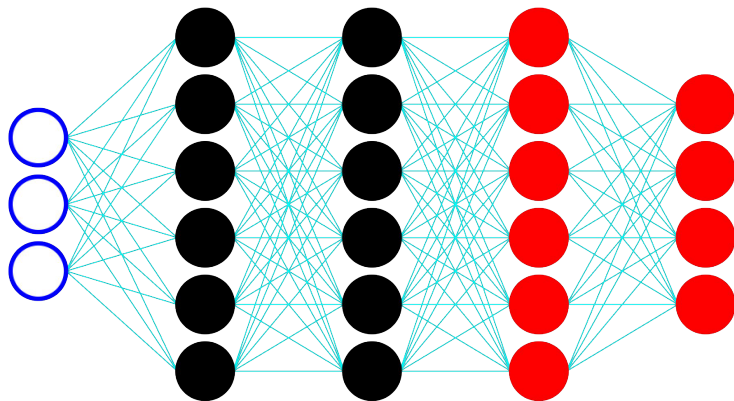


Figure 3: Handcrafted pipeline schedules, top: ZB-H1; bottom: ZB-H2

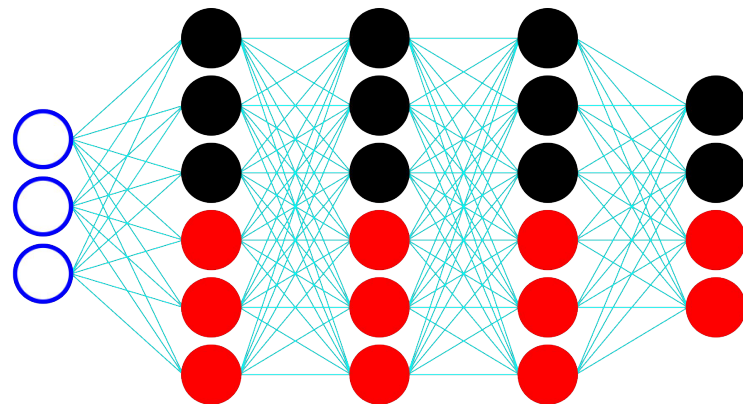
Two paradigms for model parallelism

Pipeline Parallelism



GPU 0

Tensor Parallelism



GPU 1

$$\text{Linear}(\mathbf{X}) = \mathbf{X}\mathbf{W}$$

Découpage par colonne

$$\mathbf{W} = (\mathbf{W}_1 \quad \mathbf{W}_2)$$

$$\text{Linear}(\mathbf{X}) = (\mathbf{X}\mathbf{W}_1 \quad \mathbf{X}\mathbf{W}_2)$$

Découpage par ligne

$$\mathbf{W} = \begin{pmatrix} \mathbf{W}_1 \\ \mathbf{W}_2 \end{pmatrix}$$

$$\text{Linear}((\mathbf{X}_1 \quad \mathbf{X}_2)) = \mathbf{X}_1\mathbf{W}_1 + \mathbf{X}_2\mathbf{W}_2$$

$$\text{Linear}(\mathbf{X}) = \mathbf{X}\mathbf{W}$$

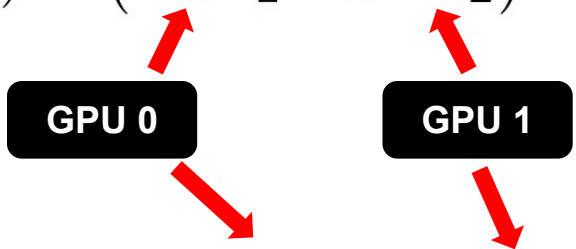
Découpage par colonne

$$\mathbf{W} = (\mathbf{W}_1 \quad \mathbf{W}_2)$$

$$\text{Linear}(\mathbf{X}) = (\mathbf{X}\mathbf{W}_1 \quad \mathbf{X}\mathbf{W}_2)$$

Découpage par ligne

$$\mathbf{W} = \begin{pmatrix} \mathbf{W}_1 \\ \mathbf{W}_2 \end{pmatrix}$$


$$\text{Linear}((\mathbf{X}_1 \quad \mathbf{X}_2)) = \mathbf{X}_1\mathbf{W}_1 + \mathbf{X}_2\mathbf{W}_2$$

$$\text{Linear}(\mathbf{X}) = \mathbf{X}\mathbf{W}$$

Découpage par colonne

$$\mathbf{W} = (\mathbf{W}_1 \quad \mathbf{W}_2)$$

$$\text{Linear}(\mathbf{X}) = (\mathbf{X}\mathbf{W}_1 \quad \mathbf{X}\mathbf{W}_2)$$

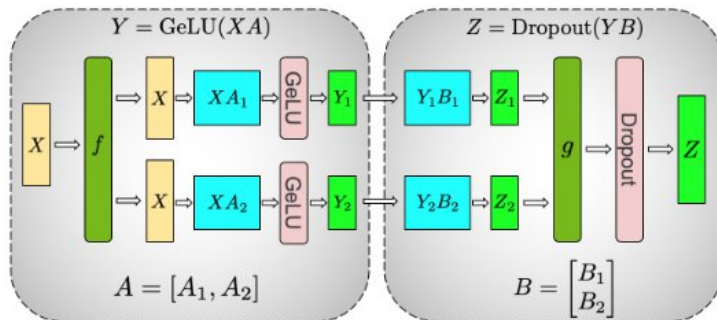
AllGather

Découpage par ligne

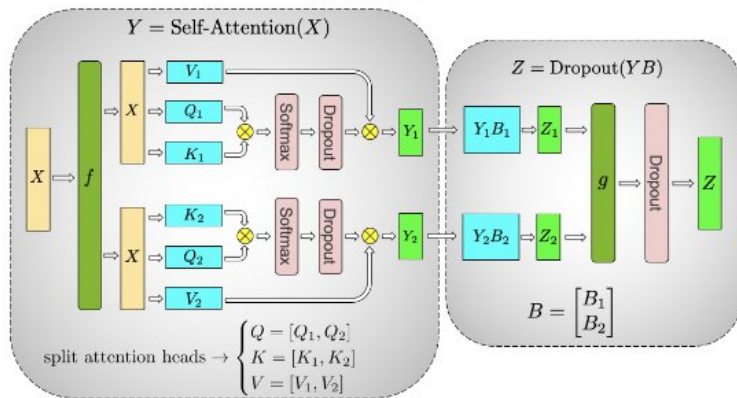
$$\mathbf{W} = \begin{pmatrix} \mathbf{W}_1 \\ \mathbf{W}_2 \end{pmatrix}$$

$$\text{Linear}((\mathbf{X}_1 \quad \mathbf{X}_2)) = \mathbf{X}_1\mathbf{W}_1 + \mathbf{X}_2\mathbf{W}_2$$

AllReduce



(a) MLP

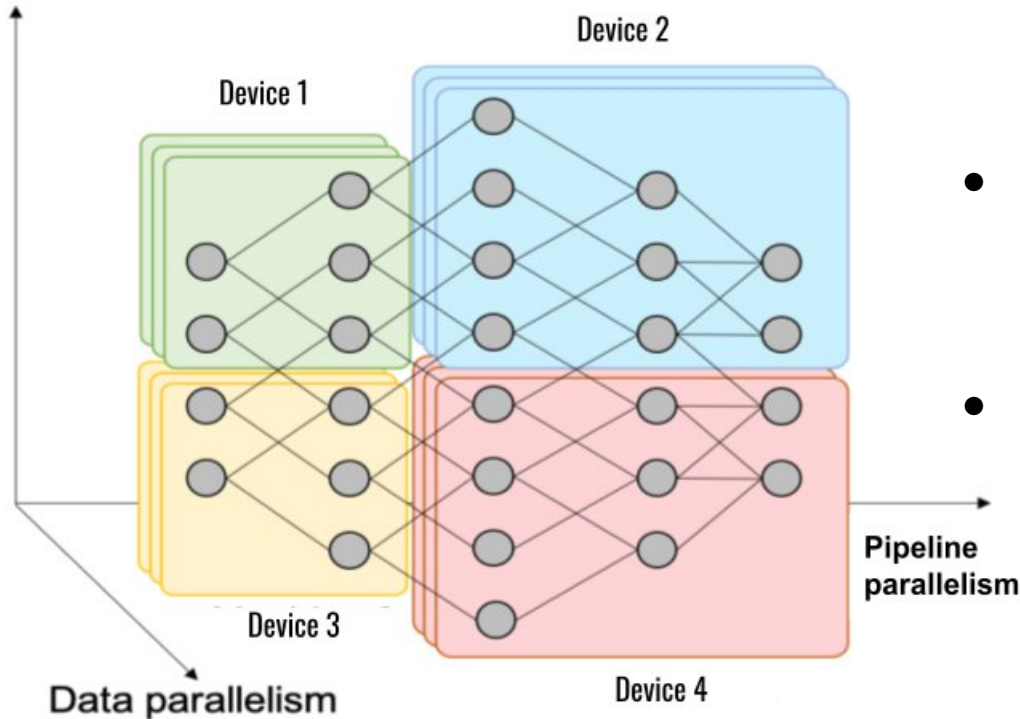


(b) Self-Attention

Par défaut, le tensor parallelism exige des synchronisations à **chaque** couche.

En alternant coupure en lignes et coupure en colonnes, on peut se permettre de ne communiquer qu'une fois toutes les **deux** couches denses.

Tensor parallelism



- **Data Parallelism**

- Simple à implémenter
- Meilleure performance
- Augmente la taille du batch (problème de convergence)

- **Pipeline Parallelism**

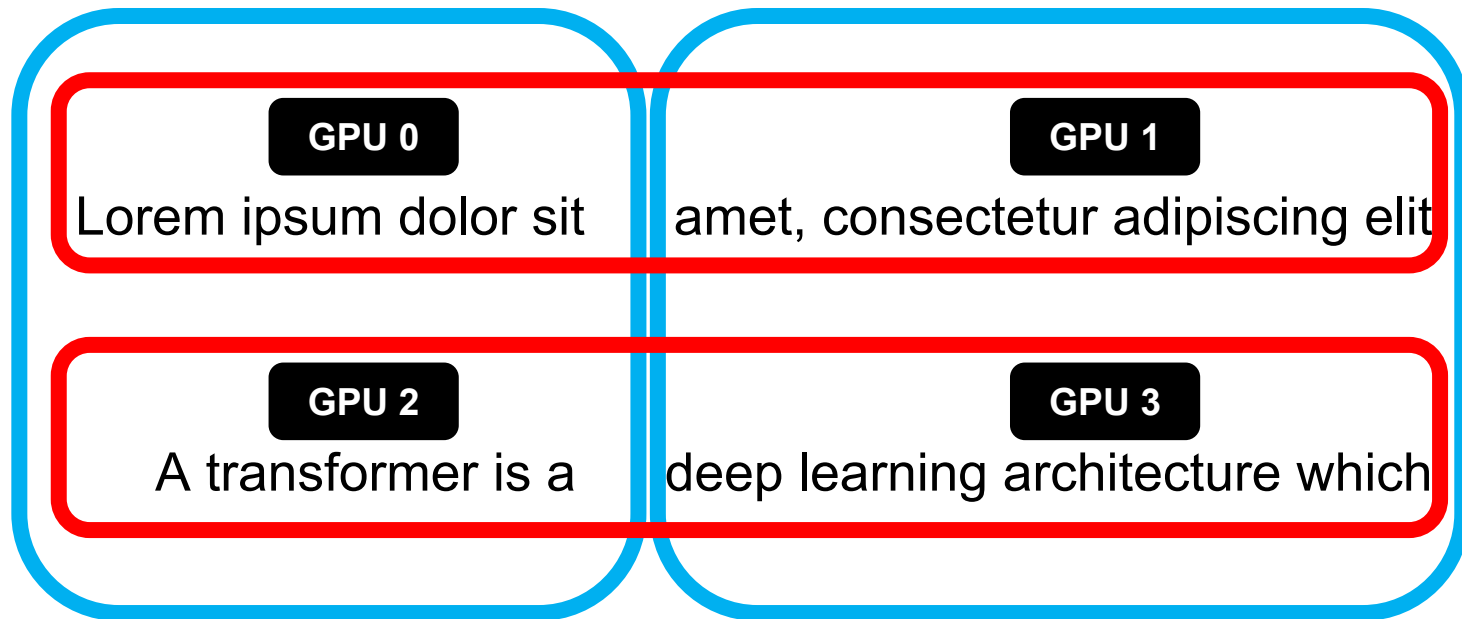
- Effort important d'implémentation.
- Équilibre entre mémoire, performance et convergence.

- **Tensor Parallelism**

- Effort d'implémentation
- Bonne accélération des calculs
- **Bande passante très sollicitée** (implémentation Intra-nœud)

Seulement pour les transformers et modèles séquentiels

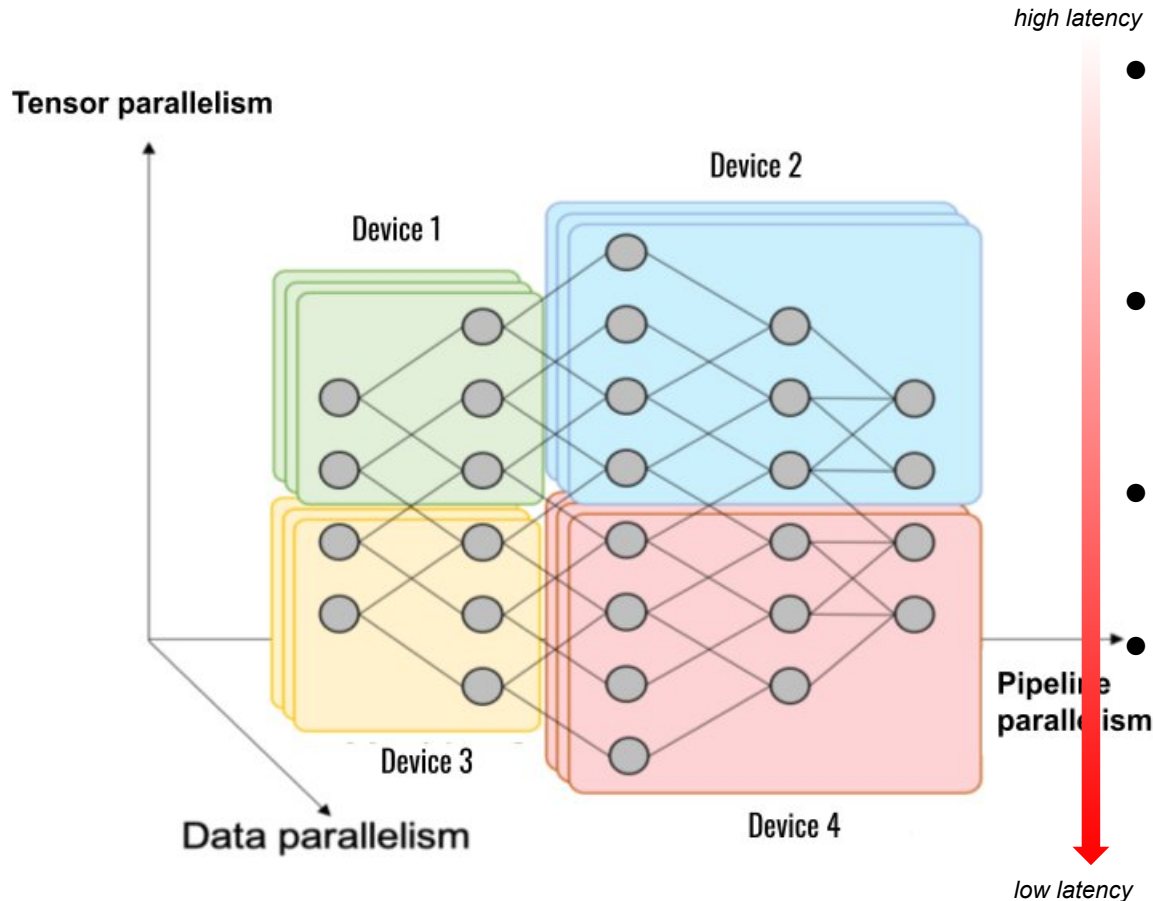
Semblable au Data Parallelism dans la dimension de la séquence



 Data Parallelism

 Context Parallelism

4D Parallelism



- **Data Parallelism**

- Simple à implémenter
- Meilleure performance
- Augmente la taille du batch (problème de convergence)

- **Pipeline Parallelism**

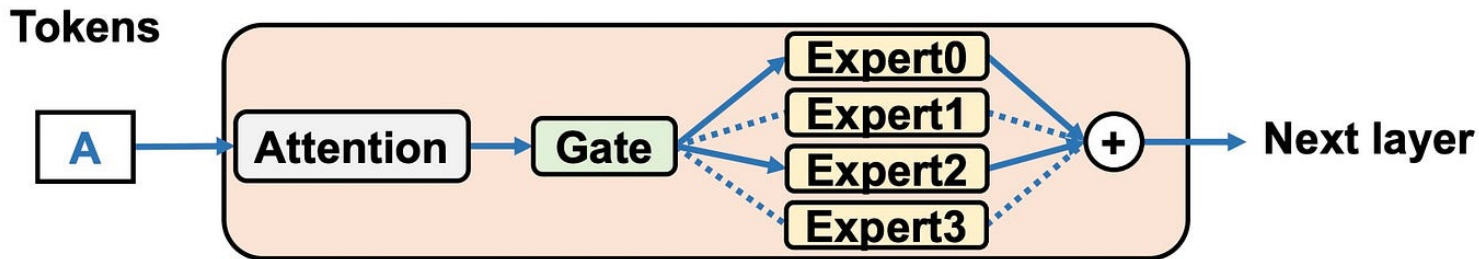
- Effort important d'implémentation.
- Équilibre entre mémoire, performance et convergence.

- **Context parallelism**

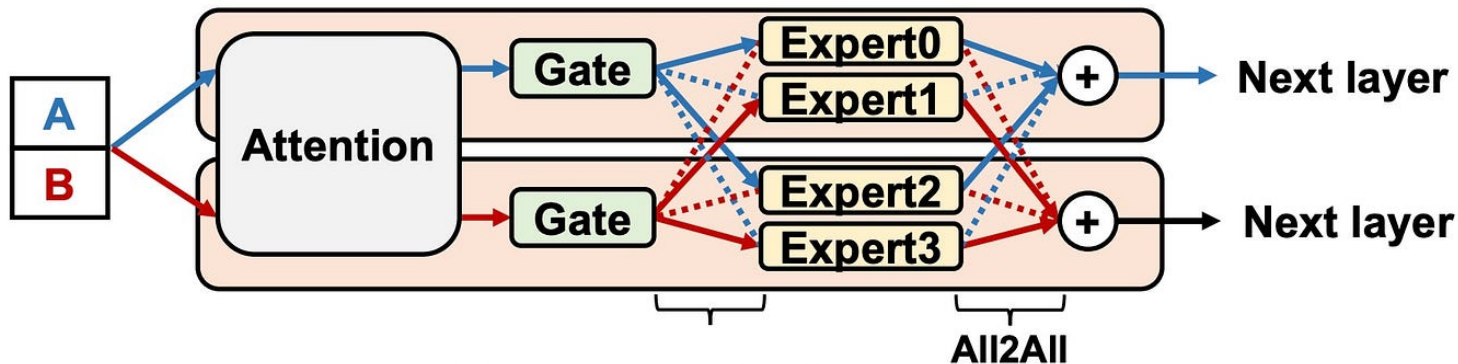
- Uniquement pour transformers
- Seulement pour **très** longues séquences

- **Tensor Parallelism**

- Effort d'implémentation
- Bonne accélération des calculs
- **Bande passante très sollicitée** (implémentation Intra-nœud)



(a) MoE layer.



(b) Expert parallelism.

API pour les gros modèles

Deepspeed ◀

Megatron-LM ◀

Accelerate, Fabric & vLLM ◀

Native Pytorch◀

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)
```

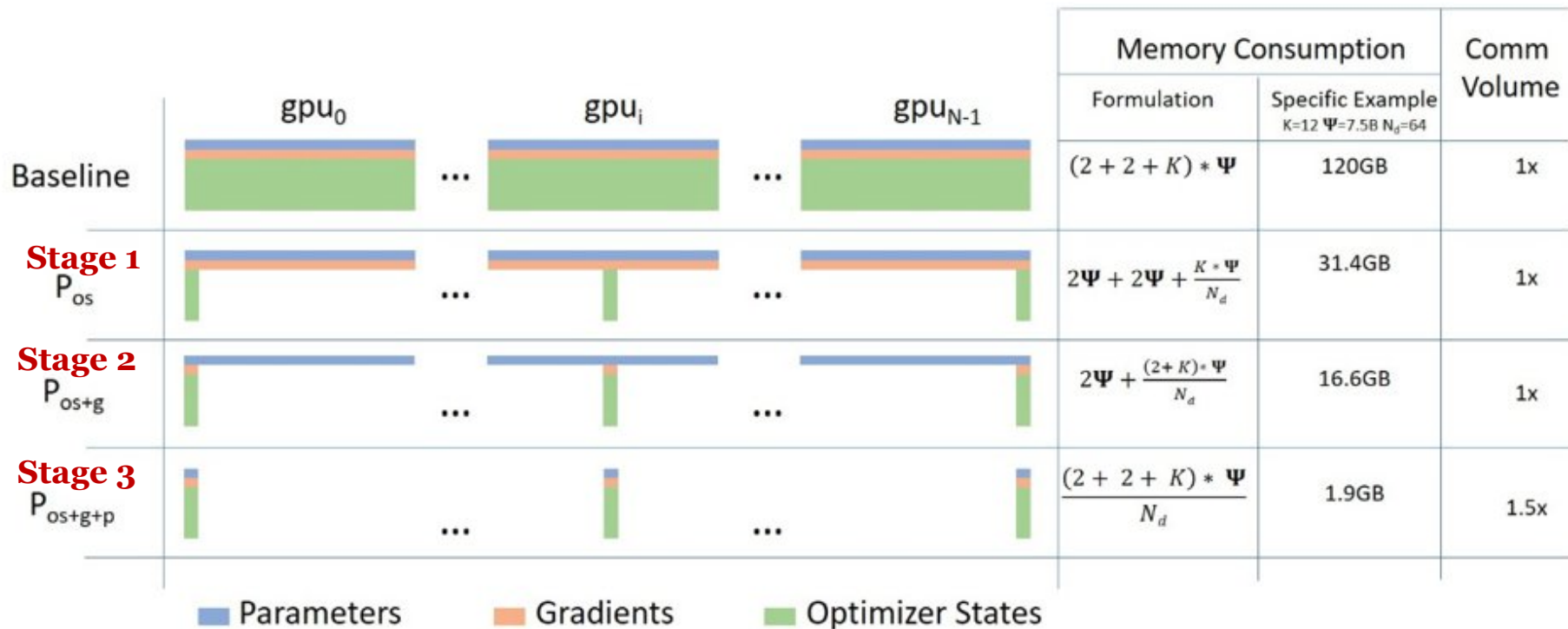
```
#runs backpropagation
model_engine.backward(loss)
```

```
#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 — Optimisation du data parallelism



Implémentations présentées dans *APEX*

Fusionne des **kernels GPU** pour économiser les opérations de lecture / écriture de mémoire

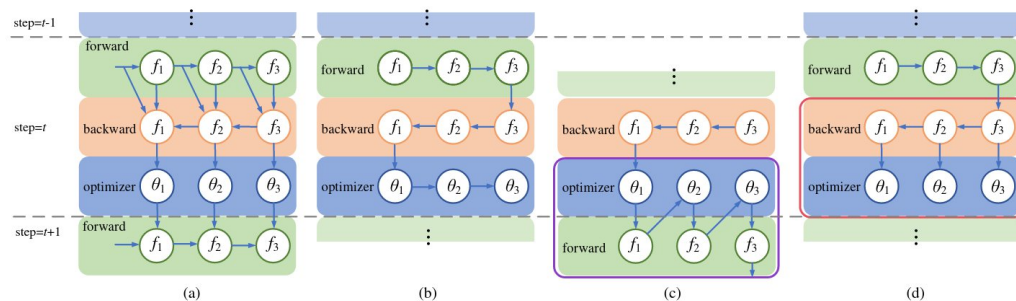
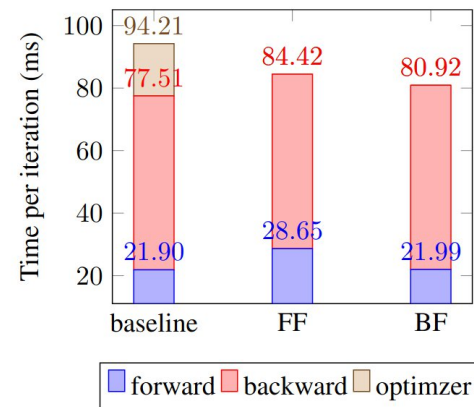
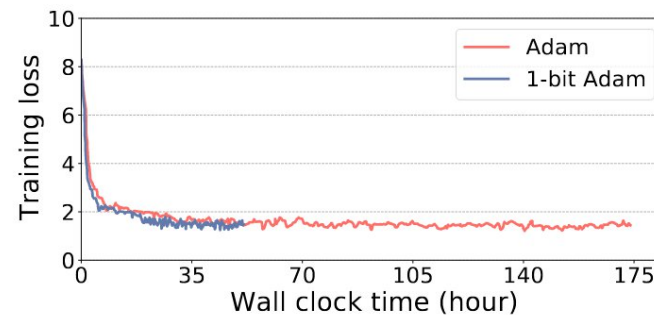
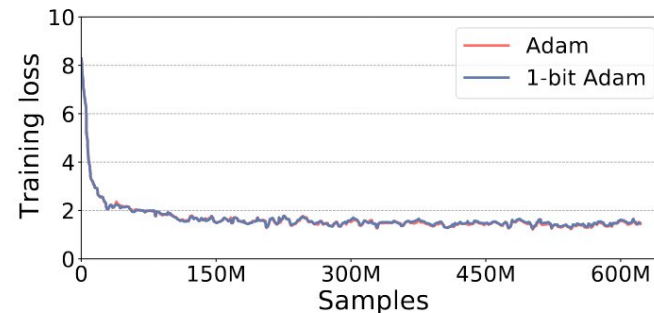
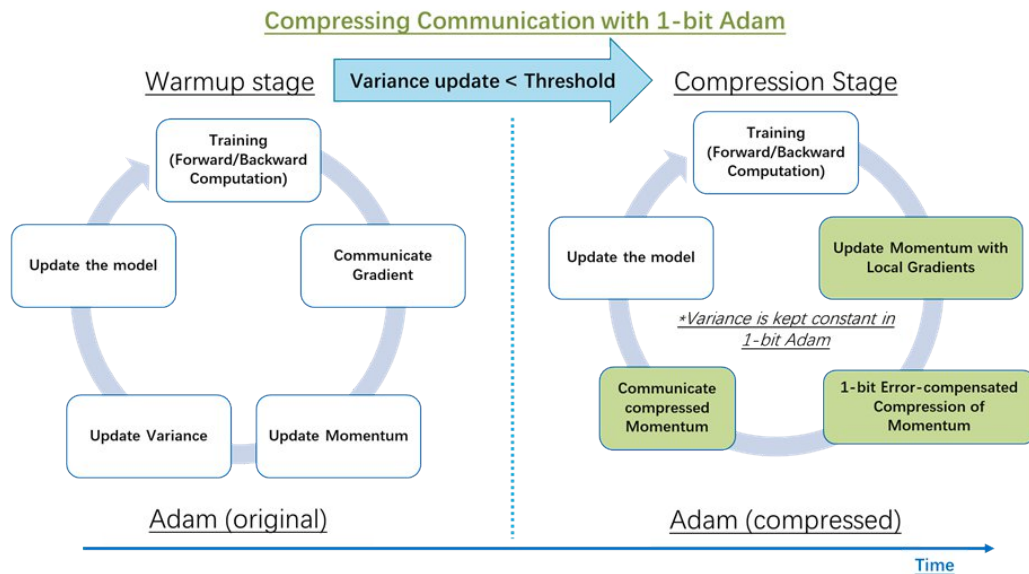


Figure 1: (a) Data dependency graph. (b) Baseline method. (c) Forward-fusion. (d) Backward-fusion. θ_i represents the trainable parameters in the layer f_i .

But : accélère l'étape des optimiseurs sur GPU.



1-bit optimizers

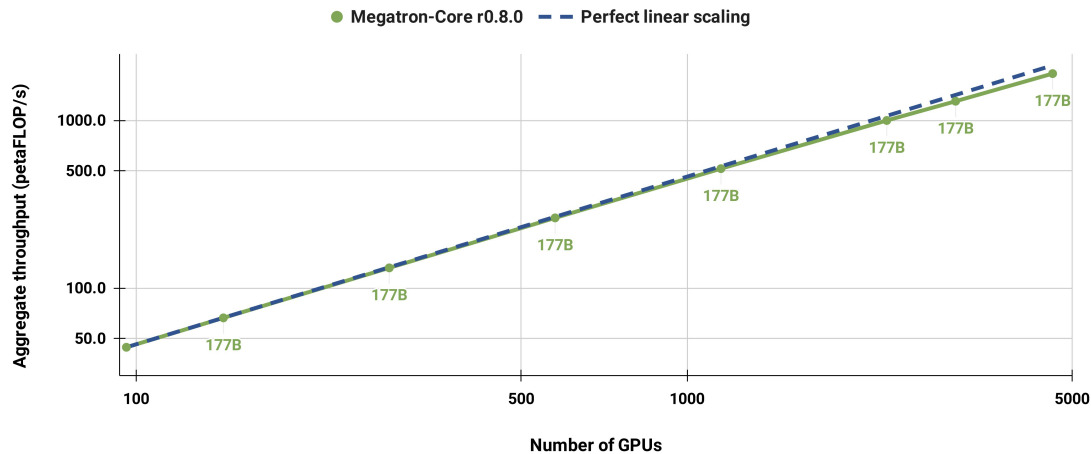
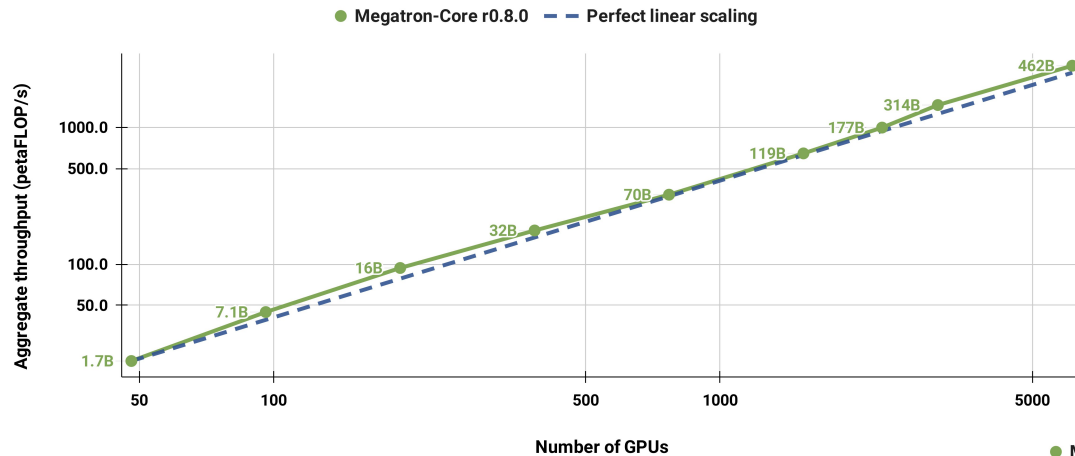


But : diminuent les communications nécessaires et donc accélère l'étape des optimiseurs pour un modèle distribué.

- [Distributed Training with Mixed Precision](#)
 - 16-bit mixed precision
 - Single-GPU/Multi-GPU/Multi-Node
- [Model Parallelism](#)
 - Support for Custom Model Parallelism
 - **Integration with Megatron-LM**
- [Pipeline Parallelism](#)
 - 3D Parallelism
- [The Zero Redundancy Optimizer \(ZeRO\)](#)
 - Optimizer State and Gradient Partitioning
 - Activation Partitioning
 - Constant Buffer Optimization
 - Contiguous Memory Optimization
- [ZeRO-Offload](#)
 - Leverage both CPU/GPU memory for model training
 - Support 10B model training on a single GPU
- [Ultra-fast dense transformer kernels](#)
- [Sparse attention](#)
 - Memory- and compute-efficient sparse kernels
 - Support 10x longer sequences than dense
 - Flexible support to different sparse structures
- [1-bit Adam and 1-bit LAMB](#)
 - Custom communication collective
 - Up to 5x communication volume saving
- [Additional Memory and Bandwidth Optimizations](#)
 - Smart Gradient Accumulation
 - Communication/Computation Overlap
- [Training Features](#)
 - Simplified training API
 - Gradient Clipping
 - Automatic loss scaling with mixed precision
- [Training Optimizers](#)
 - Fused Adam optimizer and arbitrary torch.optim.Optimizer
 - Memory bandwidth optimized FP16 Optimizer
 - Large Batch Training with LAMB Optimizer
 - Memory efficient Training with ZeRO Optimizer
 - CPU-Adam
- [Training Agnostic Checkpointing](#)
- [Advanced Parameter Search](#)
 - Learning Rate Range Test
 - 1Cycle Learning Rate Schedule
- [Simplified Data Loader](#)
- [Performance Analysis and Debugging](#)

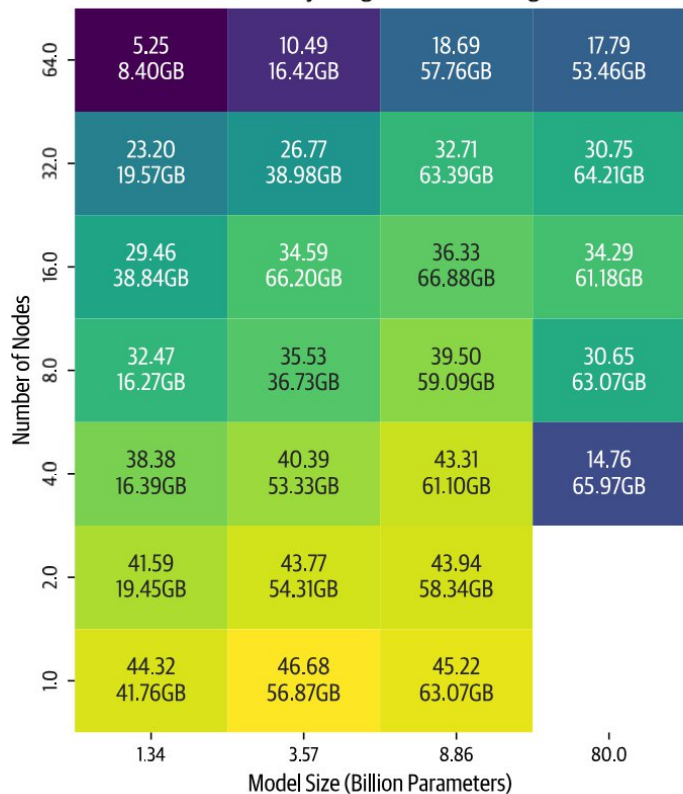
Model size	Attention heads	Hidden size	Number of layers	Tensor MP size	Pipeline MP size	Data-parallel size	Number of GPUs	Batch size	Per-GPU teraFLOP/s	MFU	Aggregate petaFLOP/s
1.7B	16	2048	24	1	1	48	48	192	408.8	41%	19.6
7.1B	32	4096	30	2	1	48	96	192	465.9	47%	44.7
16B	48	6144	32	4	1	48	192	192	489.1	49%	93.9
32B	56	7168	48	8	1	48	384	192	459.6	46%	176.5
70B	64	8192	84	8	2	48	768	384	419.7	42%	322.3
119B	80	10240	92	8	4	48	1536	768	420.5	43%	645.9
177B	96	12288	96	8	6	48	2304	1152	432.8	44%	997.2
314B	128	16384	96	8	8	48	3072	1536	474.4	48%	1457.4
462B	144	18432	112	8	16	48	6144	3072	459.9	47%	2825.6

$$MFU = \frac{\text{Model FLOPs per token} \times \text{Observed tokens per second}}{\text{Theoretical peak FLOPs of the hardware}}$$

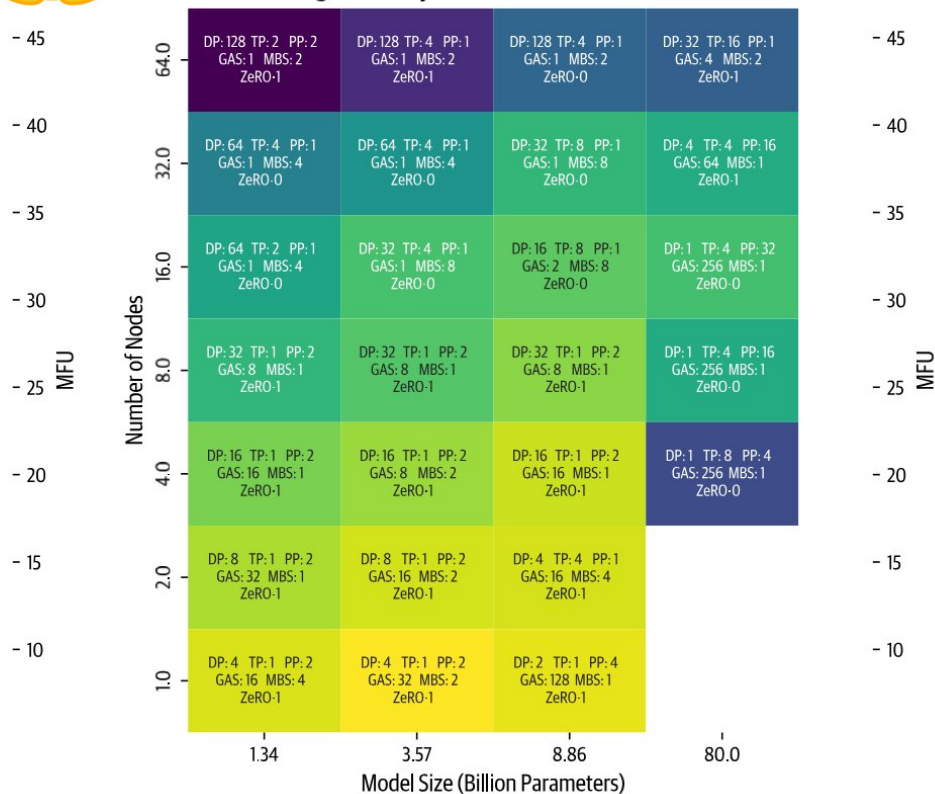




MFU and Memory Usage for Best Configurations



Best Configuration by Model Size and Number of Nodes



huggingface/ **accelerate**



🚀 A simple way to train and use PyTorch models
with multi-GPU, TPU, mixed-precision

```
srn idr_accelerate --config_file myconfig.json --zero_stage 3 train.py --lr 0.5
```



Lightning Fabric



deepspeed

**PyTorch Foundation
Welcomes DeepSpeed**

Références des images utilisées et articles

1. HuggingFace 2021, <https://huggingface.co/blog/large-language-models>
2. Nicolae, Bogdan, et al. "Deepfreeze: Towards scalable asynchronous checkpointing of deep learning models." *2020 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGRID)*. IEEE, 2020.
3. FairScale authors. (2021). FairScale: A general purpose modular PyTorch library for high performance and large scale training. https://fairscale.readthedocs.io/en/latest/deep_dive/pipeline_parallelism.html
4. Fan, Shiqing, et al. "DAPPLE: A pipelined data parallel approach for training large models." *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 2021.
5. Narayanan, Deepak, et al. "PipeDream: Generalized pipeline parallelism for DNN training." *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. 2019.
6. Rajbhandari, Samyam, et al. "Zero: Memory optimizations toward training trillion parameter models." *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2020.
7. Jiang, Zixuan, et al. "Optimizer Fusion: Efficient Training with Better Locality and Parallelism." *arXiv preprint arXiv:2104.00237* (2021).
8. Deepspeed 2020, <https://www.deepspeed.ai/2020/09/08/onebit-adam-blog-post.html>
9. Tang, Hanlin, et al. "1-bit adam: Communication efficient large-scale training with adam's convergence speed." *International Conference on Machine Learning*. PMLR, 2021.
10. Vaswani, Ashish, et al. "Attention is all you need." *Advances in neural information processing systems* 30 (2017).
11. Peltarion, <https://peltarion.com/blog/data-science/self-attention-video>
12. Dosovitskiy, Alexey, et al. "An image is worth 16x16 words: Transformers for image recognition at scale." *arXiv preprint arXiv:2010.11929* (2020).
13. Dai, Zihang, et al. "Coatnet: Marrying convolution and attention for all data sizes." *Advances in Neural Information Processing Systems* 34 (2021): 3965-3977.
14. Medium, https://medium.com/@oskyhn_77789/current-convolutional-neural-networks-are-not-translation-equivariant-2f04bb9062e3
15. AI Summer, <https://theaisummer.com/receptive-field/>
16. <https://vllm.ai/>
17. <https://huggingface.co/docs/accelerate/index>
18. Gou, Jianping, et al. "Knowledge distillation: A survey." *International Journal of Computer Vision* 129 (2021): 1789-1819.
19. Gholami, Amir, et al. "A survey of quantization methods for efficient neural network inference." *arXiv preprint arXiv:2103.13630* (2021).
20. Zhou, Aojun, et al. "Learning N: M fine-grained structured sparse neural networks from scratch." *arXiv preprint arXiv:2102.04010* (2021).
21. <https://developer.nvidia.com/blog/accelerating-inference-with-sparsity-using-ampere-and-tensorrt/>
22. Shoyebi, Mohammad, et al. "Megatron-Lm: Training multi-billion parameter language models using model parallelism." *arXiv preprint arXiv:1909.08053* (2019).
23. Bengio, Yoshua, Nicholas Léonard, and Aaron Courville. "Estimating or propagating gradients through stochastic neurons for conditional computation." *arXiv preprint arXiv:1308.3432* (2013).
24. Frankle, Jonathan, and Michael Carbin. "The lottery ticket hypothesis: Finding sparse, trainable neural networks." *arXiv preprint arXiv:1803.03635* (2018).
25. Gale, Trevor, Erich Elsen, and Sara Hooker. "The state of sparsity in deep neural networks." *arXiv preprint arXiv:1902.09574* (2019).
26. Zhu, Michael, and Suyog Gupta. "To prune, or not to prune: exploring the efficacy of pruning for model compression." *arXiv preprint arXiv:1710.01878* (2017).
27. Sanh, Victor, et al. "DistilBERT, a distilled version of BERT: smaller, faster, cheaper and lighter." *arXiv preprint arXiv:1910.01108* (2019).
28. <https://lightning.ai/docs/fabric/stable/>
29. <https://pytorch.org/blog/introducing-pytorch-fully-sharded-data-parallel-api/>