Deep Learning Optimisé - Jean Zay

Les parallélismes des gros modèles

INSTITUT DU DÉVELOPPEMENT ET DES RESSOURCES EN INFORMATIQUE SCIENTIFIQUE

DLO-JZ

6ème partie de la formation de l'IDRIS.

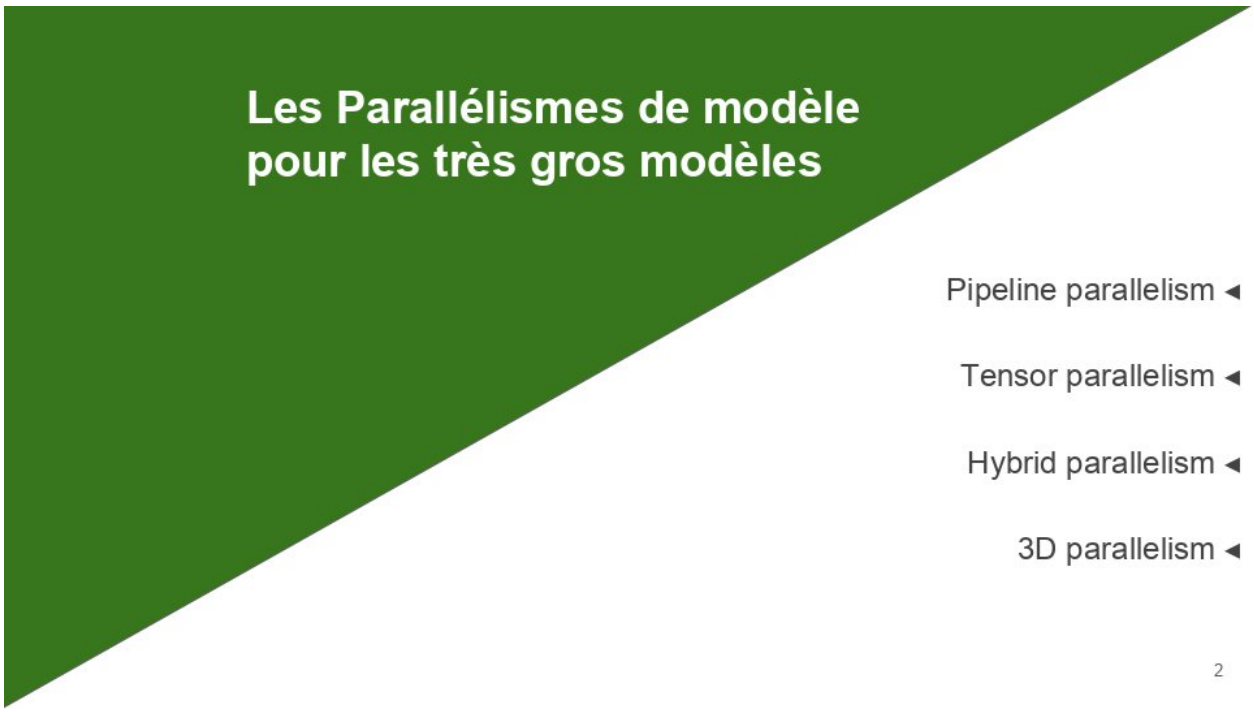
Diapositives commentées.

Auteur : Bertrand Cabot

Mai 2022. Mise à jour Février 2023.

Chapitres :

- Les Parallélismes de modèle pour les très gros modèles
- API parallélismes de modèle
- Les *Vision Transformer*

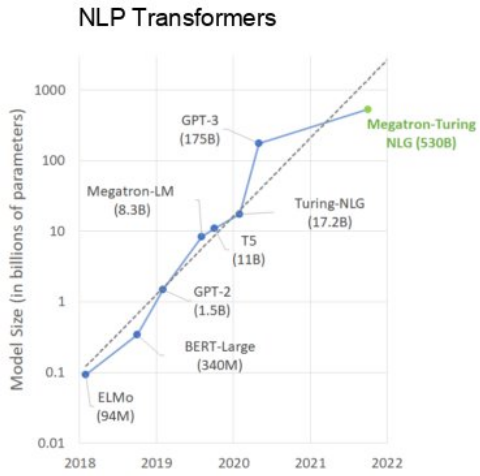


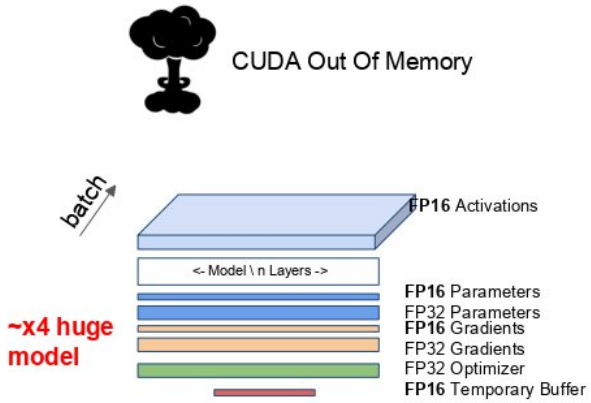
L’objectif de cette partie est de décrire les différentes formes de *Model Parallelism* possibles et l’intérêt de celles-ci.

Du fait de sa complexité, le *Model Parallelism* ne s’utilise seulement que pour les très gros modèles.

Chaque parallélisme doit se penser selon un axe qui lui est spécifique. Nous décrirons alors : le *Pipeline Parallelism*, l’*Hybrid Parallelism*, le *Tensor Parallelism*, le *3D Parallelism*.

Énormes modèles > 1 Milliard de paramètres

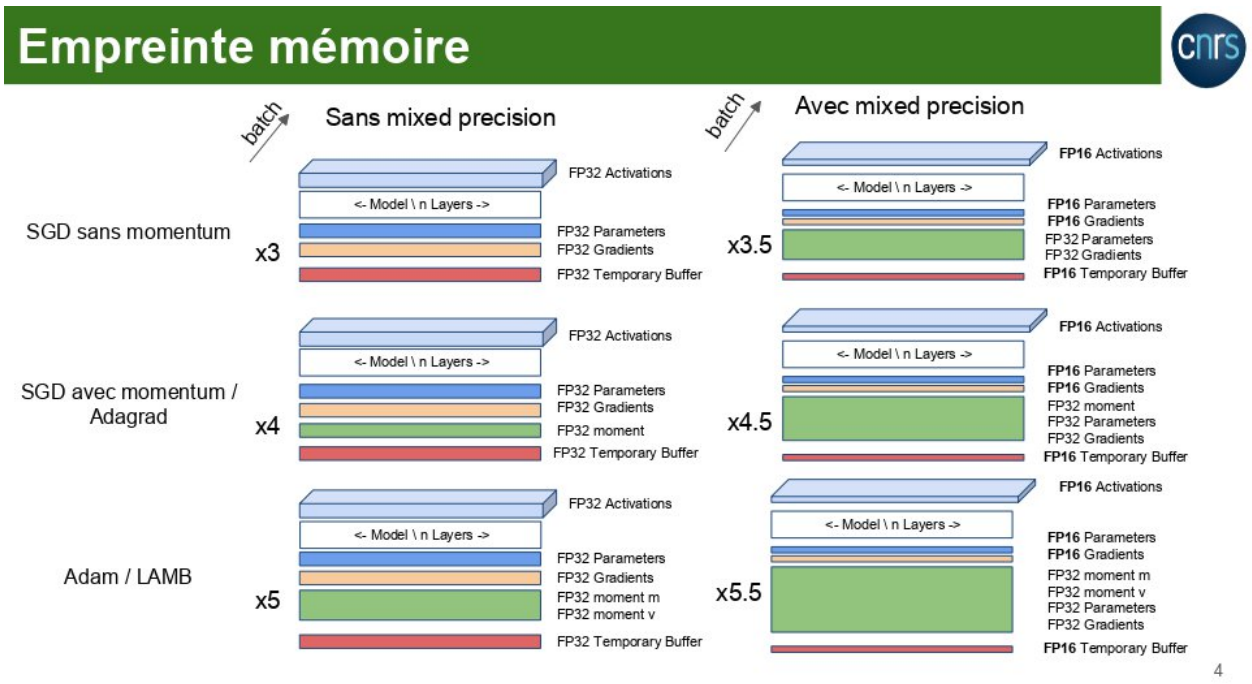




[1]

Depuis l'arrivée des *Transformer* en NLP et des *Vision Transformer* en *Computer Vision,* les modèles de réseaux de neurones sont entrés dans une autre dimension. On parlera de gros modèles de plus d'un milliard de paramètres qui apportent de nouvelles problématiques même à l'échelle d'un supercalculateur et qui sont complètement hors de portée d'un ordinateur classique.

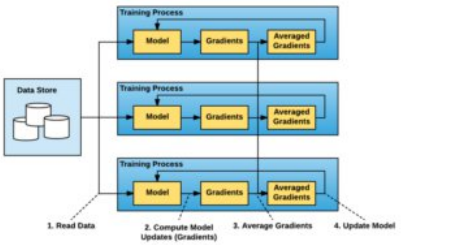
D'après ce que l'on a vu précédemment, l'empreinte mémoire directement liée au modèle était négligeable pour des modèles classiques en terme de taille. Pour les gros modèles, cela devient critique.

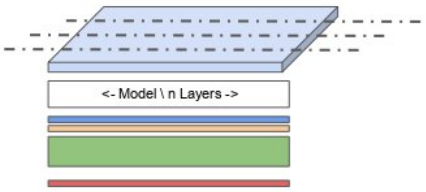


L’empreinte liée au modèle augmente bien sûr en fonction de la taille du modèle, mais aussi lorsque l’on utilise la *Mixed Precision* et selon l’*optimizer* utilisé et le nombre de *momentum* associés.

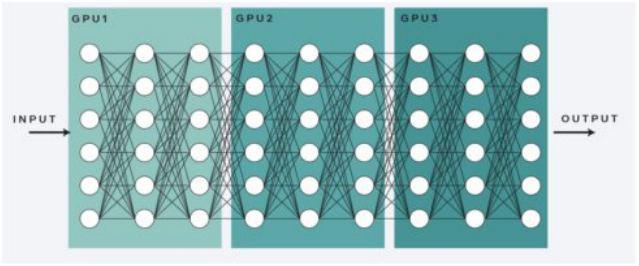
Les différents parallélismes

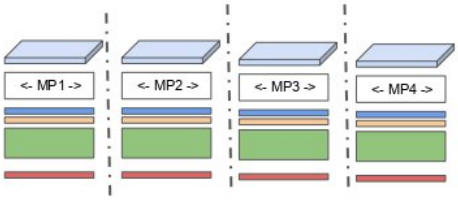
Data Parallelism
- Meilleur Throughput
- Seule l'empreinte mémoire des activations est distribuée
- Multi Processing





Pipeline Model Parallelism
- Empreinte mémoire distribuée
- Mono ou multi-processing



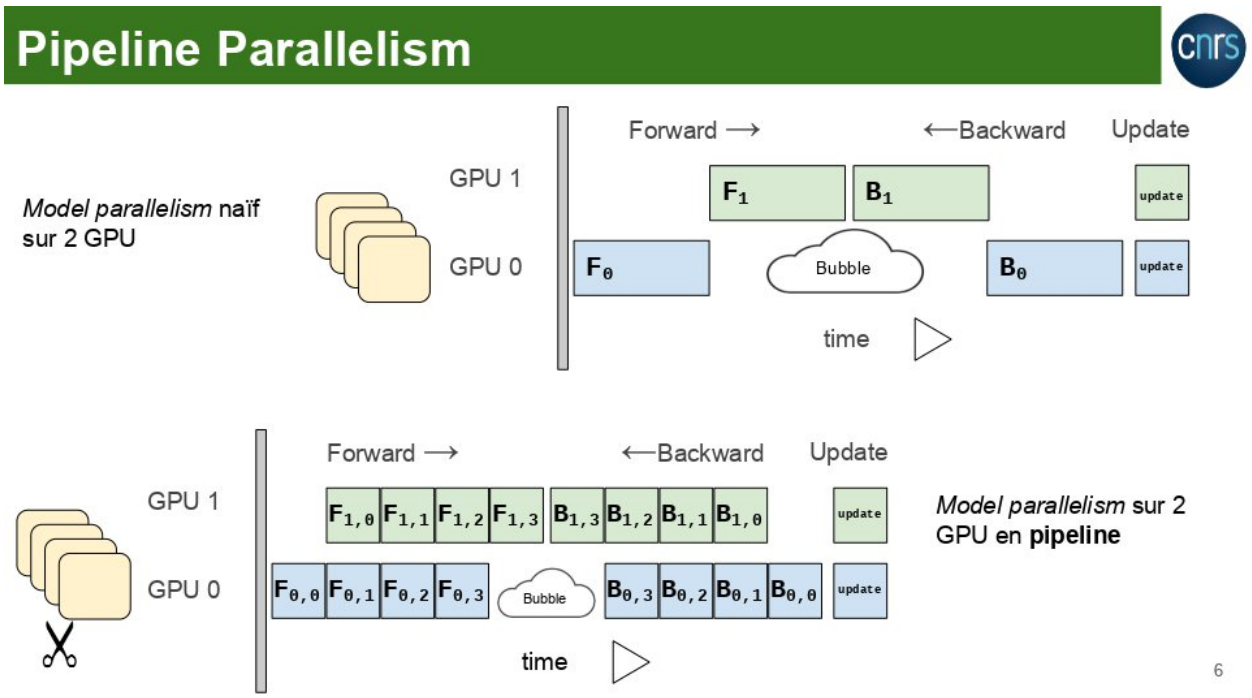


[2, 3]

5

Le *Data Parallelism* que nous avons vu précédemment est la meilleure solution pour le *throughput* et pour sa facilité d'implémentation. Cependant seule l'empreinte mémoire des activations est distribuée. Cela est un problème pour les gros modèles. Le *Data Parallelism* distribue les *batches* à travers les GPU.

Nous ajoutons donc pour les gros modèles le principe de *Model Parallelism* (notamment le *Pipeline Parallelism*). Il s'agit de distribuer le modèle à travers les GPU. Ainsi l'empreinte mémoire complète est distribuée. De plus la taille du *batch* n'est pas augmentée ce qui peut être avantageux lorsque l'on utilise un nombre de GPU très important pour la distribution par rapport à la *DDP*.



La manière naïve de faire du *Model Parallelism* serait de couper le modèle entre 2 couches et d'utiliser les GPU de manière séquentielle pendant la boucle d'apprentissage. Cependant il n'y aurait aucune accélération puisqu'il n'y aurait aucune parallélisation. Seule la distribution de l'empreinte mémoire serait réalisée.

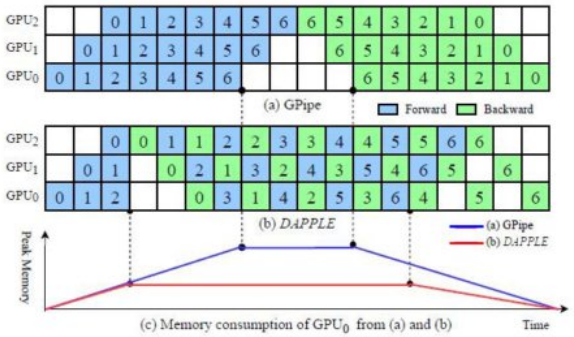
Le *Pipeline Parallelism* en découpant le modèle de la même manière entre 2 couches permet en plus d'accélérer le processus. Pour cela, il subdivise le *batch* en *micro batch*. L'itération d'apprentissage est complète lorsque tous les *micro batches* sont passés en *forward* et *backward*.

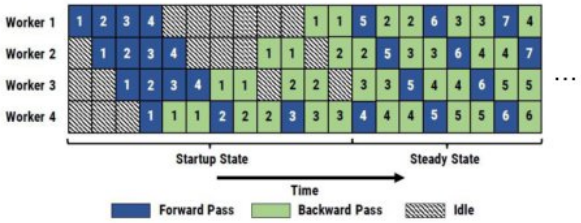
Ainsi, la *bubble* correspondant au temps d'inactivité du GPU est largement réduite. Donc seul, le *Pipeline Parallelism* est utilisé en pratique lorsque l'on coupe le modèle dans le sens des couches.

Pipeline Parallelism

Synchronous pipeline :
GPipe, DAPPLE
- \- Throughput
- \+ Memory consumption
- \+ Convergence

Asynchronous pipeline :
PipeDream, PipeMare
- \+ Throughput
- \- Memory consumption
- \- Convergence

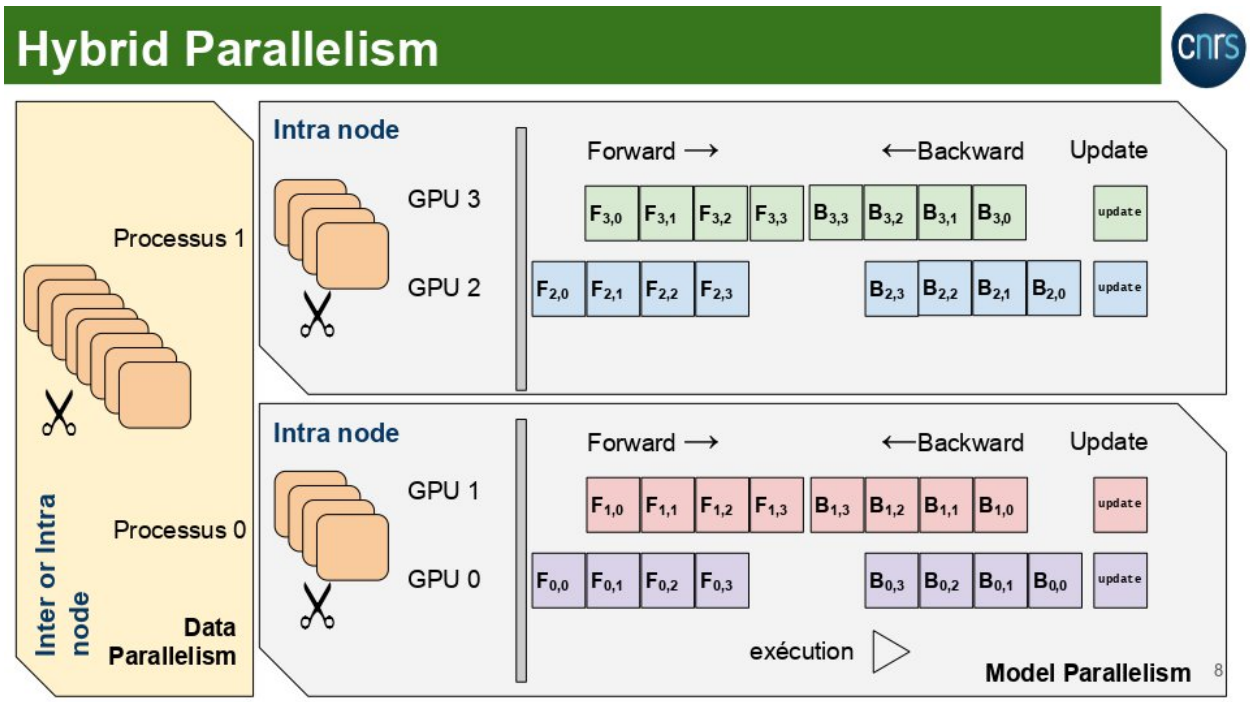




[4, 5]

Les optimisations possibles du *Pipeline Parallelism* s'organisent selon les 2 méthodes suivantes :

- Le *Pipeline Parallelism* **synchrone** correspond au *Pipeline Parallelism* décrit dans la diapositive précédente. Une optimisation possible est de changer l'ordre d'exécution des *micro batches* afin de gagner en empreinte mémoire. En effet lorsque le *backward* est exécuté l'empreinte mémoire des sorties d'activation du *micro batch* peut être libérée.
- Le *Pipeline Parallelism* **asynchrone** se permet de ne pas attendre la fin d'une itération d'apprentissage pour en commencer une nouvelle. Cela entraîne un calcul de la *Loss* parfois non avec le modèle dans sa version *N-1* mais avec le modèle dans sa version *N-2*. Ainsi, le processus de descente de gradient perd en qualité d'apprentissage, mais le parallélisme est accéléré au maximum puis qu'il n'y a quasiment plus de *bubble*. Alors, l'accélération est comparable au *Data Parallelism*. Cependant le *Pipeline* asynchrone est rarement utilisé.



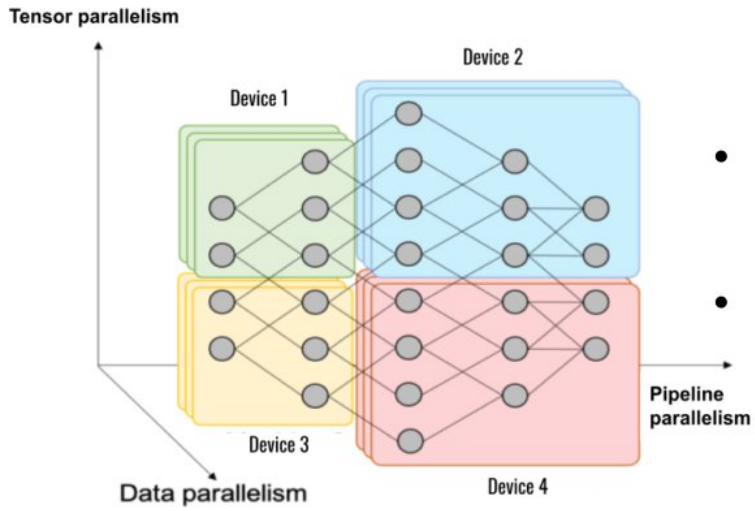
L'*Hybrid Parallelism* correspond à l'utilisation du *Data Parallelism* et du *Pipeline Parallelism* en même temps.

En effet, le *Data Parallelism* est le plus efficace pour accélérer l'apprentissage. Cependant un certain nombre de *Pipeline Parallelism* est nécessaire pour les gros modèles.

Un subtil dosage entre les 2 permettra d'optimiser au mieux le processus.

De plus si on utilise un grand nombre de GPU, le *Data Parallelism* risque de mener à des tailles de *batch* trop importantes. Ainsi, le *Pipeline Parallelism* permet d'équilibrer la taille du batch.

3D Parallelism



- **Data Parallelism**
  - Simple à implémenter
  - Meilleure performance
  - Augmente la taille du batch (problème de convergence)
- **Pipeline Parallelism**
  - Effort d'implémentation.
  - Équilibre entre mémoire, performance et convergence.
- **Tensor Parallelism**
  - Effort important d'implémentation
  - Bonne accélération des calculs
  - **Bande passante très sollicitée** (implémentation Intra-nœud)

9

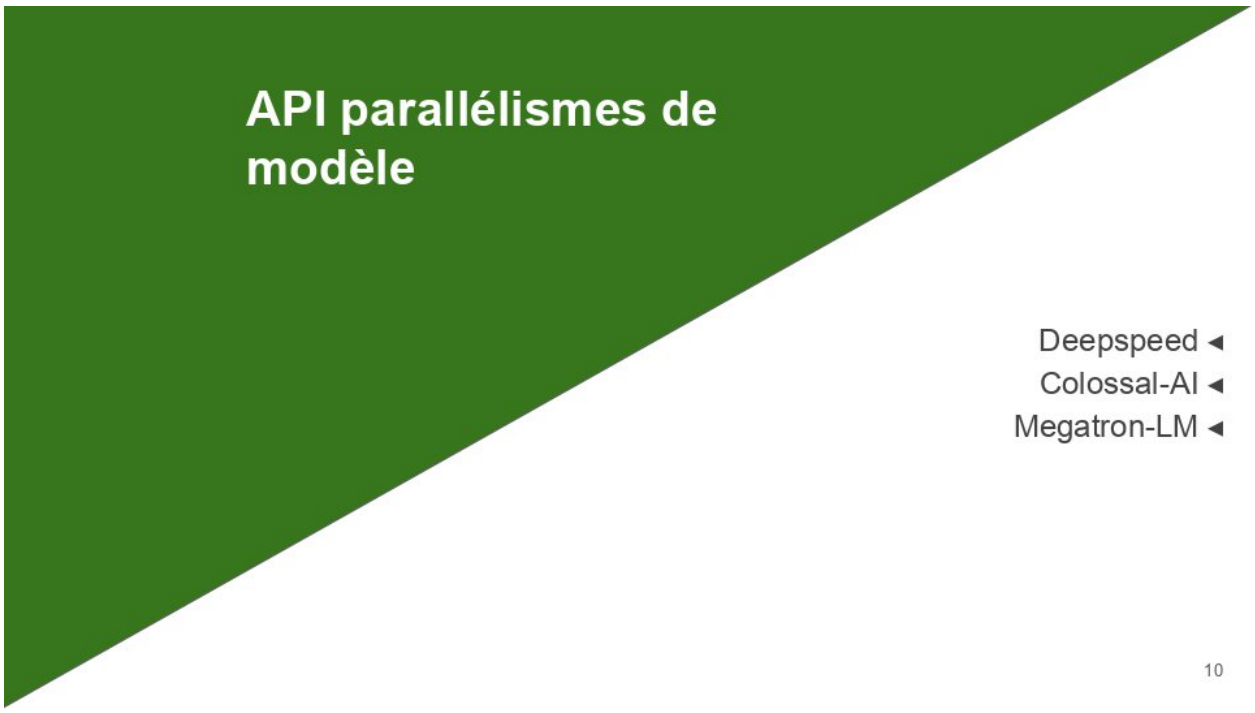
Le *3D Parallelism* distribue selon 3 axes :

- Le *Data Parallelism* selon l'axe des *batch*
- Le *Pipeline Parallelism* selon l'axe des couches
- Et, en plus, le *Tensor Parallelism* selon l'axe des nœuds des couches ou à l'intérieur des calculs des tenseurs propres à chaque couche.

Nous avons vu que le *Data Parallelism* est simple à implémenter car aujourd'hui tous les *framework* de Deep Learning intègre une solution de *Data Parallelism*, et est la meilleure solution pour accélérer le code. Cependant il tend vers des *batch* de plus en plus larges.

Nous avons vu que le *Pipeline Parallelism* permet tout en accélérant le code de distribuer l'empreinte mémoire liée au modèle. Il demande un certain effort d'implémentation ou est accessible via un ensemble de librairie dédié au *Model Parallelism*.

Le *Tensor Parallelism* accessible seulement grâce à un effort conséquent d'implémentation ou grâce à des librairies très spécialisées (Megatron-LM) permet une nette accélération du calcul et un partage de l'empreinte mémoire, mais génère un flux de communication plus important entre les GPU. Il sera utilisé seulement entre les GPU d'un même nœud de calcul.



La complexité de la mise en œuvre du *Model Parallelism* et de l'apprentissage de gros modèles ou de l'accélération à très grande échelle que l'on vient de décrire est accessible grâce à certaines librairies dédiées.

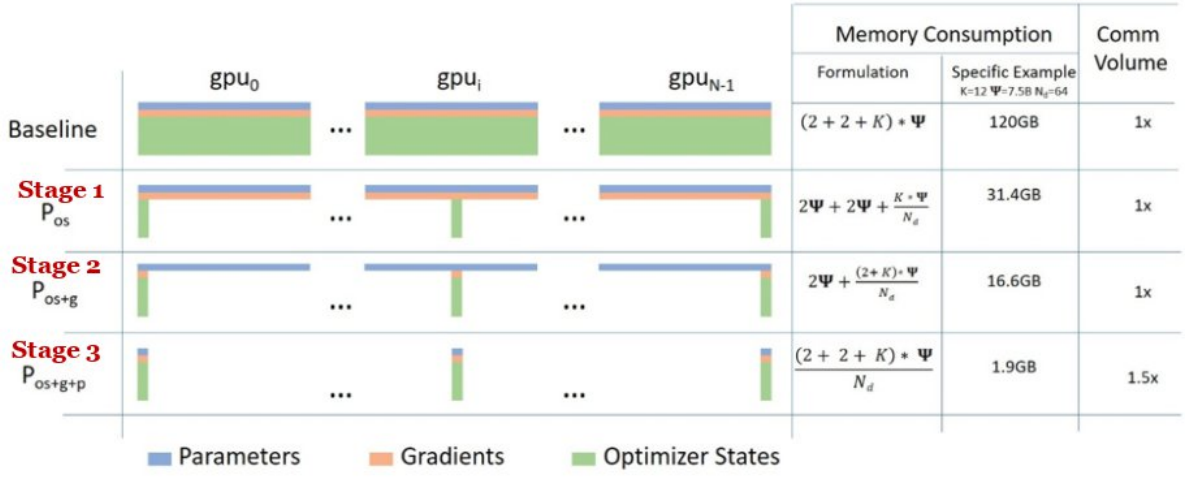
Cette partie décrit les librairies dédiées suivantes :

- *Deepspeed* de Microsoft
- *Colossal-AI* de HPC-AI Tech
- *Megatron-LM* de NVIDIA

La librairie *Deepspeed* de Microsoft dédiée à l'accélération des gros modèles et très gros modèles, propose un ensemble de techniques en PyTorch.

Deepspeed est assez facile à intégrer à un code PyTorch et possède une multitude de fonctions d'accélération et de parallélisme.

ZeRO — Optimisation du data parallelism



	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

[6]

La fonctionnalité la plus intéressante de *Deepspeed* est ZeRO, soit *Zero Redundancy Optimizer*. ZeRO est une optimisation du *Data Parallelism* pour les gros modèles. Il permet d'économiser l'empreinte mémoire liée au modèle en utilisant le *Data Parallelism*. Le *Data Parallelism* recopie dans chaque GPU entièrement les variables liées au modèle (poids, gradients, l'historique de l'*optimizer*).
ZeRO permet de partager l'empreinte mémoire liée au modèle. Chaque GPU conserve une portion différente de l'empreinte du modèle. Lorsqu'un GPU a besoin d'une partie du modèle qu'elle ne possède pas, le GPU détenteur de cette portion la communique au GPU en question. Ainsi, ZeRO fonctionne exactement comme le *Data Parallelism* en distribuant selon l'axe des *batches* tout en distribuant le stockage des informations liées au modèle.

Il existe 3 "stage" de ZeRO:

- Le stage 1 qui distribue la partie *Optimizer* équivalent au *Data Parallelism* en terme de communication inter GPU.
- Le stage 2 qui distribue les parties *Optimizer* et gradient équivalent au *Data Parallelism* en terme de communication inter GPU.
- Le stage 3 qui distribue les parties *Optimizer*, gradient et poids du modèle plus coûteux de 50% en terme de communication.

Il existe aussi des optimisations de ZeRO : *ZeRO Offload* et *ZeRO Infinity* qui utilisent la mémoire CPU pour distribuer des modèles encore plus gros de plus de 1000 milliards de paramètres correspondant aux modèles du futur.

Liens vers les vidéos :

- **MPI_Alltoall**
- **MPI_Allgather**
- **MPI_Allreduce (naïf)**
- **MPI_Allreduce (ReduceScatter + Allgather)**

Pour comprendre comment ZeRO arrive à partitionner les gradients et les états des optimiseurs, il nous faut comprendre comment les communications collectives se déroulent et comment elles sont appliquées au Deep Learning.

MPI (Message Passing Interface) est un protocole de communication entre différents processus. On ne s'intéresse qu'à un sous-ensemble des routines MPI. En pratique, les communications entre processus peuvent être faites avec d'autres librairies que MPI (par exemple NCCL de NVIDIA pour les communications inter-GPU). Quelque soit la librairie utilisée, la dénomination MPI sera utilisée pour ne pas créer de confusion, puisque le même concept peut porter des noms différents selon les implémentations.

Dans les vidéos présentées, on fait une simplification en ajoutant un espace Communicateur. En réalité, les données ne sont pas envoyées dans un espace mémoire neutre (le communicateur) mais les

communications sont réalisées directement avec chacun des autres GPU ; le communicateur permet simplement de se mettre d'accord sur les rangs et la manière de se contacter. Dans certaines vidéos sur le *Data Parallelism*, on fait en plus la simplification de faire la réduction dans le communicateur, alors qu'elle est faite en réalité comme le montre le *All-reduce*, c'est-à-dire partitionnée sur tous les processus contenus dans le communicateur.

Le *All-to-all* est une opération qui "transpose" les données. Le *All-gather* permet à tous les processus de connaître toutes les données des autres processus.

Dans le *Data Parallelism*, on s'intéresse particulièrement au *All-reduce*. C'est une opération de réduction (une somme typiquement) sur les données de tous les processus.

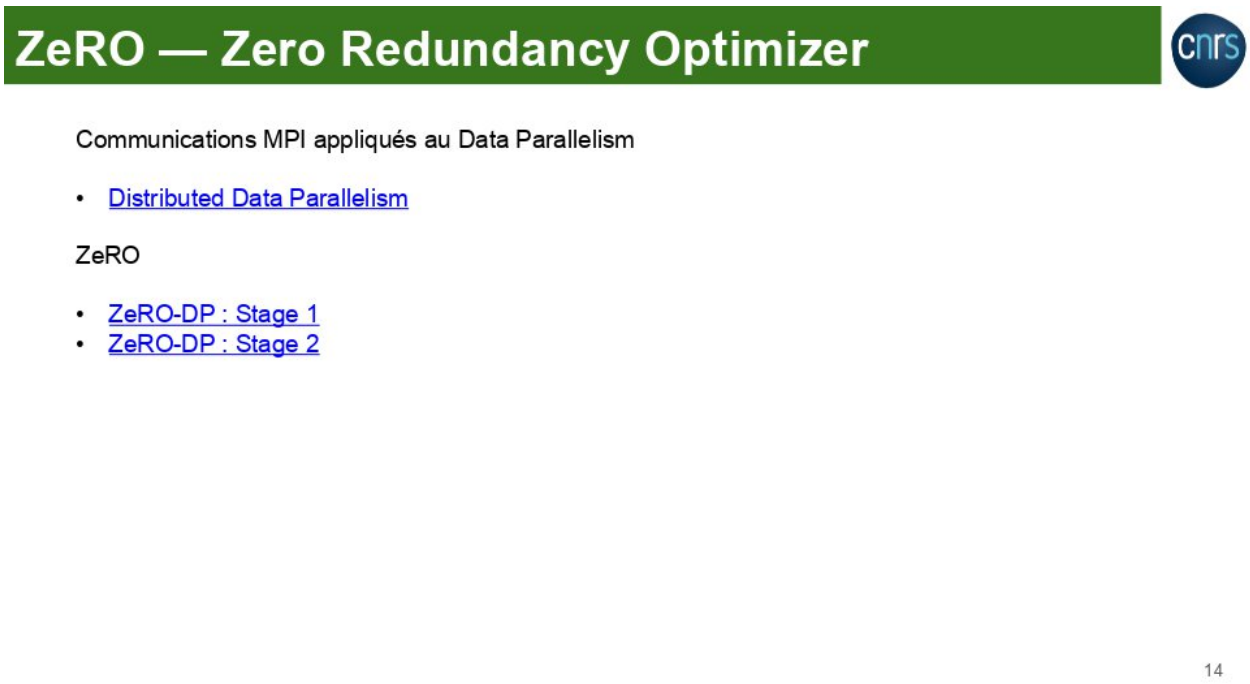
Le *All-reduce* peut être implémenté de plusieurs manières différentes. On pourrait imaginer faire en sorte que chaque processus ait toutes les données, puis qu'ils réalisent tous la même réduction séparément (ce qu'on a appelé le *All-reduce* naïf). Cela fonctionne mais la même opération est répliquée ce qui est inefficace.

Une manière intuitive, qui serait certainement adoptée par des humains voulant réaliser cette opération, est de donner un rôle particulier à l'un des processus (qui devient alors une sorte de contremaître) qui reçoit les données de tous les autres processus, applique la réduction et diffuse le résultat (*broadcast*). Cela permet de pas réaliser plusieurs fois la même réduction mais tous les processus sauf le maître sont en attente du résultat ce qui est également inefficace.

Une manière plus proche de l'implémentation NCCL est de faire en sorte que la réduction soit partitionnée. Grâce à un *All-to-all*, chaque processus participe à une seule et même réduction. Les résultats sont ensuite rassemblés grâce à un *All-gather*.

Cette dernière manière est ce que NCCL appelle un *ReduceScatter* + *Allgather*. En réalité NCCL applique un algorithme plus sophistiqué (le *ring all-reduce*) qui vient prendre la place du *All-to-all*. Mais on arrive également à un point où le vecteur réduit est partitionné (donc suivi d'un *all-gather*).

L'optimisation ZeRO intervient sur cette seconde partie, après avoir réalisé la réduction de manière partitionnée sur tous les processus, donc ce détail d'implémentation ne change pas le principe de ZeRO.



Liens vers les vidéos :

- **Distributed Data Parallelism**
- **ZeRO-DP : Stage 1**
- **ZeRO-DP : Stage 2**

Pour exploiter le parallélisme de données, on procède de la manière suivante.

- On duplique le modèle, les gradients et les états des optimiseurs sur tous les processus
- Chaque processus prend un *mini-batch* de données différents
- Chaque processus réalise le *feed-forward* puis le calcul du gradient sur son *mini-batch* avec le modèle dupliqué
- Les gradients sont mis en commun grâce à un *All-reduce*.
- Les processus ayant le même modèle, les mêmes optimiseurs, et maintenant les mêmes gradients, la même mise à jour des poids peut être faite (le modèle et l'optimiseur restent donc identiques sur tous les processus, puisque les mises à jour sont entièrement déterministes).

Avec ZeRO de niveau 1, on partitionne les états des optimiseurs. Plutôt que de mettre en commun les gradients, chaque processus fait une mise à jour des poids qui sont associés aux états de l'optimiseur qui lui sont attribués lors du partitionnement. On peut ensuite rassembler les

paramètres via un *All-gather*. On n'augmente pas le volume de communication pour la raison suivante : un *all-reduce* des gradients correspond à un *All-to-all* suivi d'une réduction puis un *all-gather* des gradients ; avec ZeRO on fait un *All-to-all* suivi d'une réduction des gradients, on intercale la mise à jour des poids associés, puis un *all-gather* des paramètres. Puisque la matrice de paramètres a la même taille que la matrice des gradients, le volume de communication reste inchangé. Cette optimisation permet de diviser la quantité de mémoire nécessaire par GPU par 4.

Avec ZeRO de niveau 2, puisqu'un processus n'a pas besoin de tous les gradients, plutôt que d'attendre la fin de la rétropropagation pour les envoyer aux autres processus, on peut les envoyer au fur et à mesure qu'ils sont calculés. Cela n'augmente pas le volume de communications, mais cela augmente le nombre de communications réalisées. Cette optimisation permet de diviser la quantité de mémoire nécessaire par GPU par 2 en plus de l'optimisation ZeRO de niveau 1.

Avec ZeRO de niveau 3, on réalise également un partitionnement des paramètres, il faut donc lors du *forward* et du *backward* les demander au processus qui détient les paramètres d'une couche donnée. Cela augmente donc les communications (en l'occurrence jusqu'à 50%). En contrepartie, cela permet de diviser la quantité de mémoire nécessaire par GPU par le nombre de répliques du réseau de neurones.

Fused optimizers

Implémentations présentent dans *APEX*

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

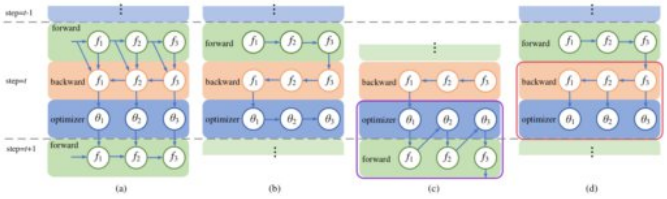
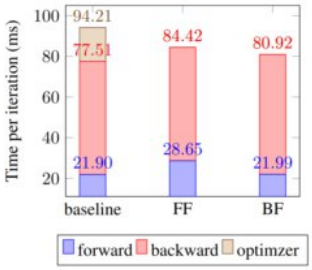


Figure 1: (a) Data dependency graph. (b) Baseline method. (c) `Forward-fusion`. (d) `Backward-fusion`. θ_i represents the trainable parameters in the layer f_i.



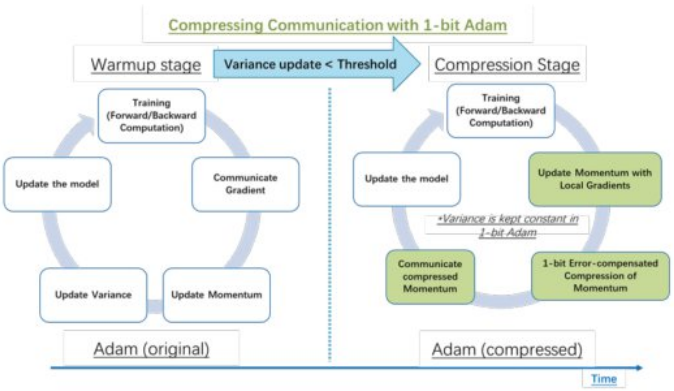
But : accélère l'étape des optimiseurs sur GPU.

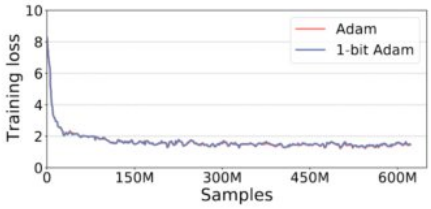
[7]

Deepspeed intègre aussi les *optimizer* dit fusionnés proposés par APEX.

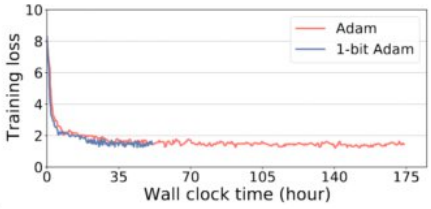
Il fusionne des ***kernels* GPU** pour économiser les opérations d'initialisation et de lecture / écriture de mémoire, ce qui engendre une légère accélération des *optimizer* au sein des GPU.

1-bit optimizers





(a) Sample-wise



(b) Time-wise

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

[8, 9]

Deepspeed propose aussi des *optimizer* dit *One-bit* qui diminuent les communications nécessaires et donc accélèrent l'étape des optimiseurs pour un modèle distribué en *Data Parallelism*.

Le principe, par exemple du 1-*bit Adam* est de calculer le *momentum* seulement localement, puis d'échanger seulement une information codée sur 1 bit de divergence entre les *momentum* locaux. La variance, étant non linéaire, ne peut être traitée de la même façon. Sur le constat empirique que la variance rapidement devient quasiment stable, le 1-bit Adam, après un *warm-up stage* correspondant à un Adam classique pendant quelques *epoch*, bascule dans son mode compressé où la variance est fixée à la dernière valeur calculée.

Cela permet une réduction des communications de l'ordre de 97%. Pour un gros modèle déployé sur un nombre important de GPU en *Data Parallelism*, on observe une réduction de temps d'un facteur 3 pour une *accuracy* quasiment identique.

Toutes les applications de Deepspeed

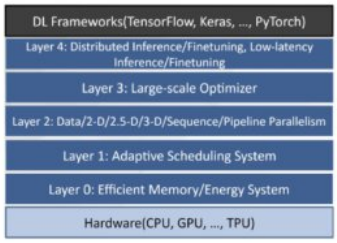
- Distributed Training with Mixed Precision
  - 16-bit mixed precision
  - Single-GPU/Multi-GPU/Multi-Node
- Model Parallelism
  - Support for Custom Model Parallelism
  - **Integration with Megatron-LM**
- Pipeline Parallelism
  - 3D Parallelism
- The Zero Redundancy Optimizer (ZeRO)
  - Optimizer State 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

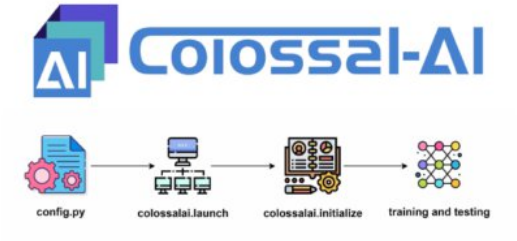
17

Deespeed propose en plus de ZeRO, des *Fused optimizer* et des *1-bit optimize*r, un nombre important d'autres applications pour l'accélération :

- Le *Pipeline Parallelism*
- L'intégration de *Megatron-LM* permettant d'utiliser le *3D Parallelism* couplé avec ZeRO
- Le *0/1 Adam* qui est une optimisation du *1-bit Adam*
- Le paramétrage de la mémoire et des *buffers*
- Le *Sparse Attention*
- etc

Colossal-AI





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

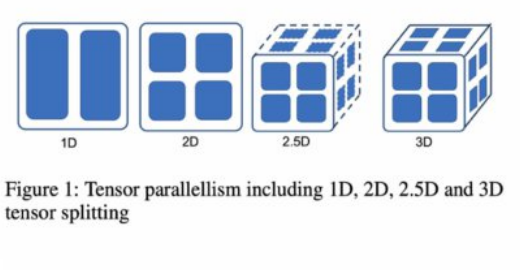
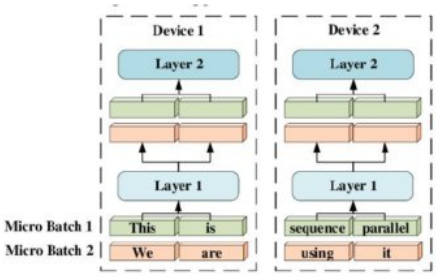


Figure 1: Tensor parallellism including 1D, 2D, 2.5D and 3D tensor splitting

- Sequence Parallelism



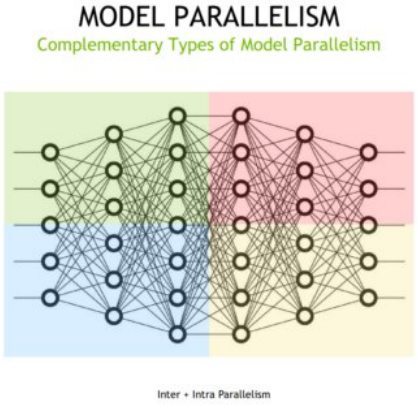
[16, 17]

Colossal-AI cherche à être une référence dans le domaine de la parallélisation pour le Deep Learning. Cette bibliothèque unifie les multiples paradigmes de parallélisation et les met à disposition dans un unique outil, simplifiant leur utilisation et permettant aux chercheurs et aux ingénieurs de ne pas avoir à implémenter des méthodes de parallélisme parfois particulièrement complexes.

L'outil est récent mais a beaucoup de potentiels. Sur certains *benchmarks*, cet outil atteindrait des performances supérieures à *DeepSpeed* ou *Megatron-LM*. Enfin, la bibliothèque contient des implémentations d'autres types de parallélismes non abordés dans ce cours comme le *Sequence Parallelism* (apparu avec *Megatron-LM*), ou des *Tensor Parallelism* multi-dimensionnels.

Megatron-LM

Model Parallelism de GPU NVIDIA (tensor and pipeline) efficace en multi-nœud pour le *pre-training* de *Transformer* comme GPT, BERT, et T5 utilisant la *mixed precision*.



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

La colonne *Model-parallel size* décrit un degré de *Tensor Parallelism* et de *Pipeline Parallelism* combinés

Pour les nombres supérieurs à 8, un *Tensor Parallelism* de taille 8 est typiquement utilisé. Ainsi, par exemple, le modèle de *145B* indique une taille de *Model Parallelism* totale de 64, ce qui signifie que cette configuration a utilisé TP=8 et PP=8.

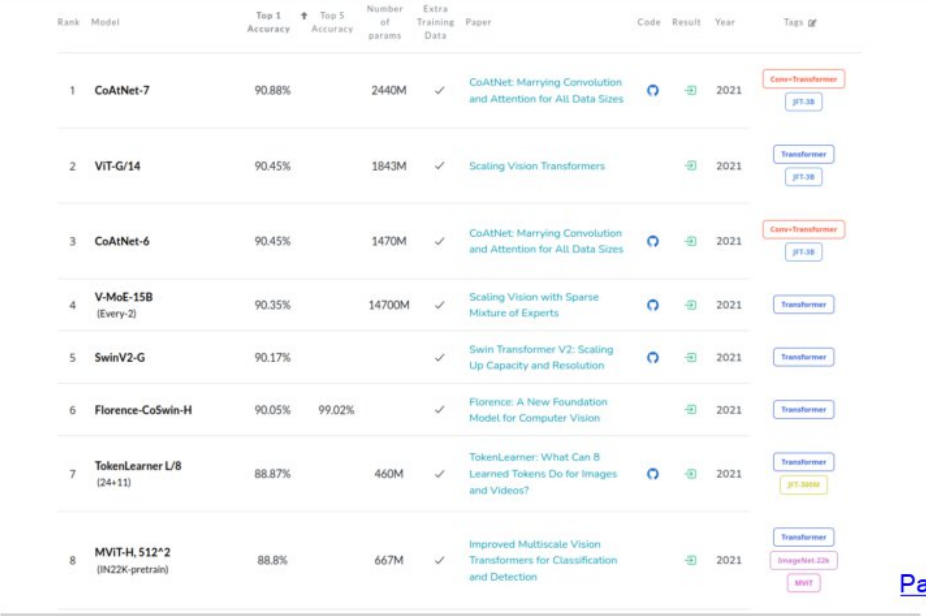
19

NVIDIA développe *Megatron-LM* qui permet de gérer un *3D Parallelism* clef en main utilisant la *Mixed Precision* pour les plus célèbres architectures de *Transformer* comme GPT, BERT, T5 avec des GPU NVIDIA.

Megatron-LM est un apport majeur pour l'apprentissage des plus gros modèles actuels de Transformer. Les versions plus récentes de *Megatron-LM* ont également introduit le *Sequence Parallelism*, une méthode pour partitionner les activations dans le sous-groupe de GPU pour les couches où le *Tensor Parallelism* n'intervient pas (par exemple les *LayerNorm*).

Cette partie a pour objectif de présenter les *Transformer*, les *Vision Transformer,* puis CoAtNet qui servira de gros modèle type pour le TP sur le *Model Parallelism*.

Vision Transformers >> Resnet-50: 25M

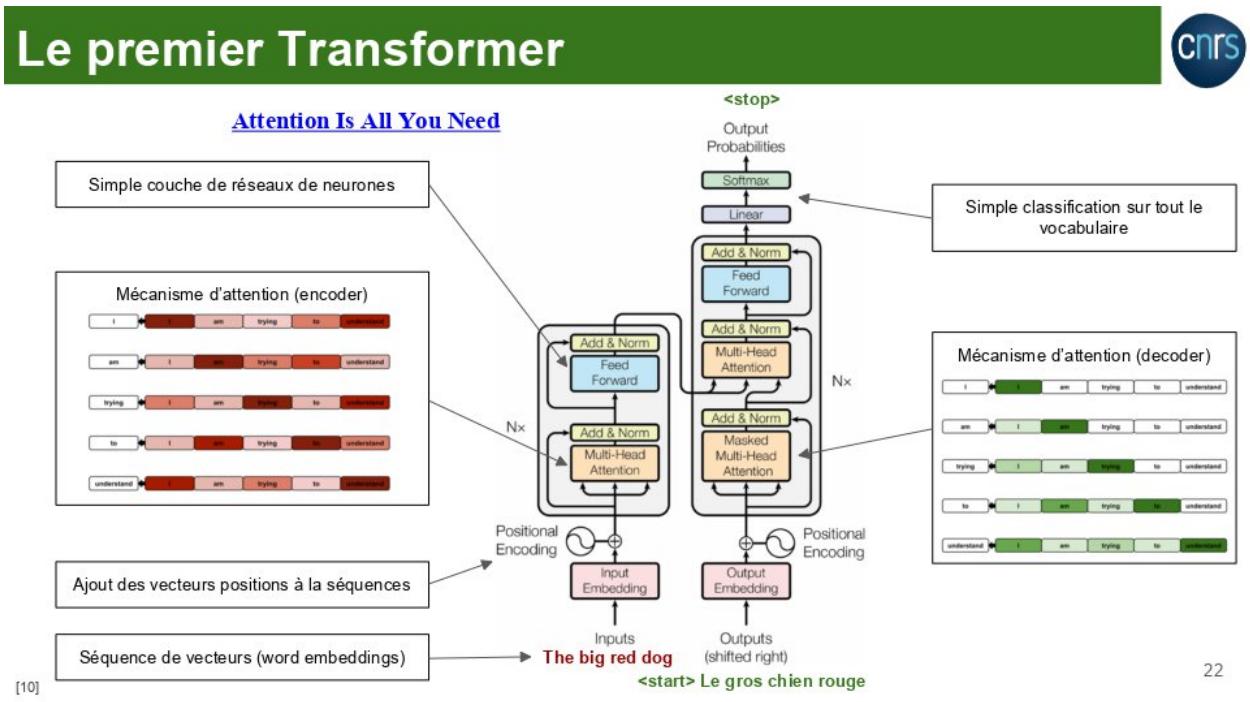


Rank	Model	Top 1 Accuracy	Top 5 Accuracy	Number of params	Extra Training Data	Paper	Code	Result	Year	Tags
1	CoAtNet-7	90.88%		2440M	✓	CoAtNet: Marrying Convolution and Attention for All Data Sizes			2021	Conv+Transformer, JFT-3B
2	ViT-G/14	90.45%		1843M	✓	Scaling Vision Transformers			2021	Transformer, 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

Paperwithcode

Sur *Paperwithcode* nous pouvons voir que l'état de l'art des modèles sur *Imagenet* sont des *Vision Transformer* de plus d'un milliard de paramètres, soit des gros modèles qui rentrent dans les problématiques vues précédemment.

A noter que *CoAtNet* que l'on utilisera pendant le TP, a la première place du classement.

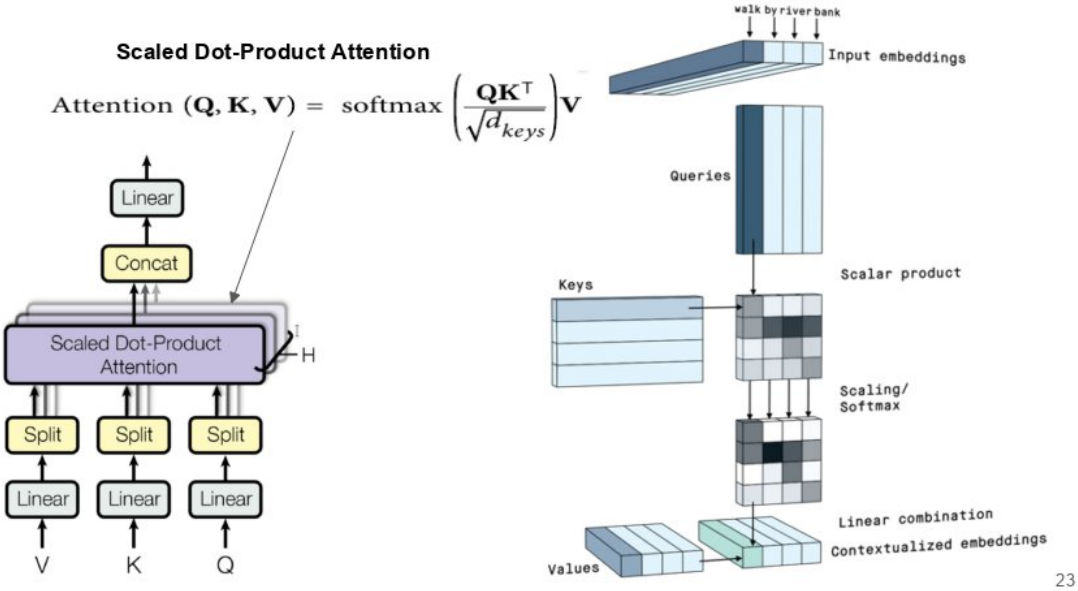


Le premier *Transformer* décrit dans le papier “Attention Is All You Need” correspondait alors à un nouveau type d’architecture permettant la traduction de texte de l’anglais au français. Ce premier *Transformer* s’inspire et améliore les RNN utilisés pour cette même application de traduction. Il est composé d’un encoder et d’un encoder.

Les RNN les plus sophistiqués exploitaient un mécanisme d’*attention* pour résoudre le problème de la représentation d’une phrase entre l’*encoder* et le *decoder.* Le Transformer exploite exclusivement ce mécanisme d’*attention* sans avoir de couches récurrentes.

Au lieu d’encoder la phrase mot à mot comme le fait les RNN, il prend la séquence entière à encoder en ajoutant une information de position du mot dans la phrase et transforme la séquence en une séquence latente en passant par le mécanisme de *self-attention*.

Le mécanisme de Self-Attention



[10, 11]

Le mécanisme de *Self-Attention* consiste à appliquer le *Scaled Dot-Product Attention* sur une séquence d'*Input embeddings* (qui a déjà subit une transformation linéaire).

Les tenseurs *Queries*, *Keys* et *Values* sont identiques dans la *Self-Attention* ce qui permet de transformer la séquence en une séquence latente qui prend en compte la dépendance de chaque mot par rapport à chaque autre mot. QK^T correspond à la matrice de dépendance.

Le *Multi-head Attention* divise la séquence d'*Input Embeddings* pour la passer dans plusieurs *heads* de *Scaled Dot-Product*. Cela permet de prendre en compte plusieurs niveaux de liens de dépendance en même temps et plusieurs représentations d'un même vecteur. Cela devient aussi un processus complètement parallélisable et accéléré. Toutes les transformations linéaires permettent d'avoir en plus une liberté totale sur les dimensions des tenseurs.

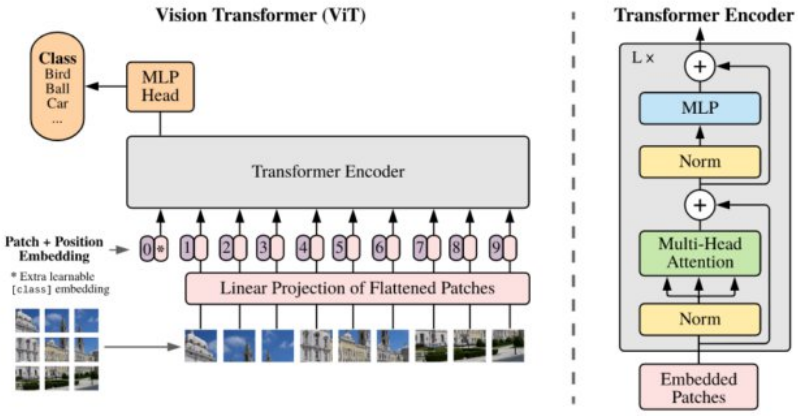
Les Transformers

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

En résumé, les Transformers :

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

Vision Transformer (ViT)



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

[12]

Le premier *Vision Transformer* appliqué à de l'imagerie est décrit dans le papier "An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale".

Au lieu de traiter une séquence de mot, le *ViT* traite une séquence d'image. Pour cela il *patch* l'image en sous-images représentées en vecteurs de pixels. Ces *patch* après une transformation linéaire représentent la séquence d'I*nput Embeddings* auquel on ajoute une information de position : exactement comme pour le premier *Transformer* en NLP.

Cependant comme ce qui est décrit pour *BERT* en NLP, on ajoute un patch 0, un class token à la séquence qui servira après transformation à la sortie de classification.

CoAtNet

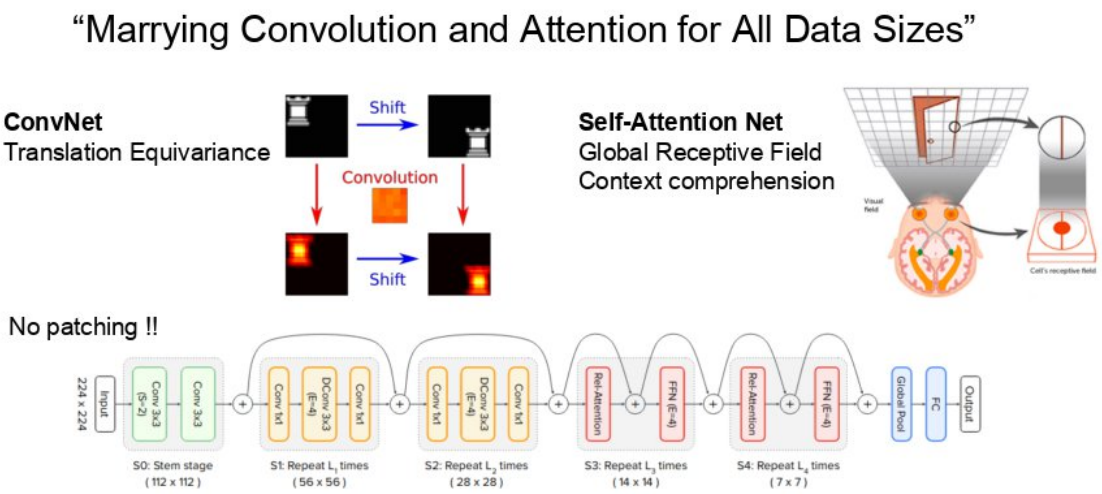


Figure 4: Overview of the proposed CoAtNet.

[13, 14, 15]

Enfin, *CoAtNet* que l'on utilisera pour le TP sur le *Model Parallelism* est décrit dans le papier "CoAtNet: Marrying Convolution and Attention for All Data Sizes".

CoAtNet est l'assemblage d'abord d'un réseau CNN classique puis d'un *Vision Transformer*. Puisque le modèle commence en CNN, il n'y a pas de *patching* d'image à faire en entrée. Nous pouvons donc remplacer facilement un CNN dans un code par un *CoAtNet*. La transition entre CNN et *Vision Transformer* se fait en patchant la sortie de la partie CNN.

Cela a pour avantage de profiter de l'équivariance en translation des CNN et de la compréhension globale du contexte de l'image des *Vision Transformer*, en même temps.

CoAtNet - Résultats

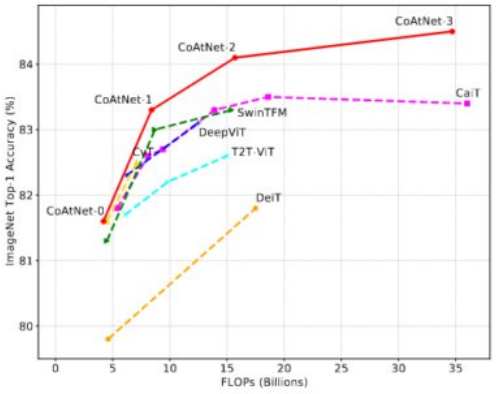


Figure 2: Accuracy-to-FLOPs scaling curve under ImageNet-1K only setting at 224x224.

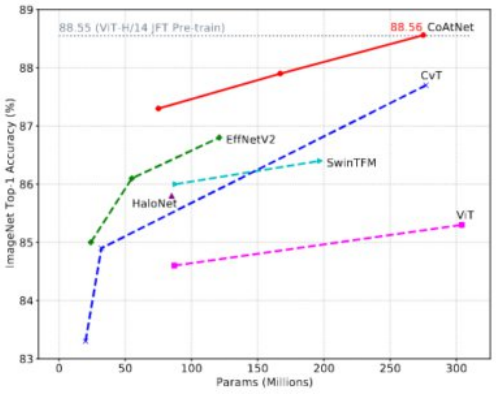


Figure 3: Accuracy-to-Params scaling curve under ImageNet-21K ⇒ ImageNet-1K setting.

[13]

Les résultats des 7 niveaux des modèles *CoAtNet* surpassent tous les *Vision Transformers* actuels et les CNN actuels à nombre de paramètres égal ou à coût de calcul égal.

Il est intéressant de noter que pour augmenter le *score* d'*accuracy* des *Vision Transformer* sur *Imagenet-1k*, il existe des extensions, par exemple *ImageNet-21k* 100 fois plus grosse, pour un apprentissage augmenté.

TP : CoAtNet-7

```
local:~$ ssh jean-zay

jz:~$ module load pytorch-gpu/py3/1.11.0
jz:~$ idrjup $WORK
```

- Limite du *Data Parallelism* avec CoAtNet
- Implémenter ZeRO
- Implementer le Pipeline Parallelism
- Recherche du meilleur compromis

Références des images utilisées

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. Bian, Zhengda, et al. "Colossal-AI: A unified deep learning system for large-scale parallel training." *arXiv preprint arXiv:2110.14883* (2021).
17. https://medium.com/@hpcaitech/colossal-ai-a-unified-deep-learning-system-for-large-scale-parallel-training-2fed5df097c0