

Soporta nativo de entrenamiento distribuido en Pytorch

Diego Andrade Canosa

Roberto López Castro



Índice

- Introducción al entrenamiento distribuido
- Estrategias de entrenamiento distribuido
- *Minibatch Stochastic Gradient Descent*
- Soporte en Pytorch para Distributed Data-Parallel
- Soporten Pytorch para RPC Based Distributed Training



Glosario para tener a mano

<https://ml-cheatsheet.readthedocs.io/en/latest/index.html>



Introducción

- La era del Deep Learning (DL) dentro del Machine Learning (ML) se caracteriza por la utilización masiva de recursos computacionales
 - Varios aceleradores: CPU multinúcleo, GPUs, TPUs
 - Varios nodos de forma coordinada
- El uso masivo de estos recursos es una necesidad
 - Entrenar un modelo tipo GPT en una sola GPU de alta gama llevaría años
 - Si usamos 256 GPUs (por ejemplo) reducimos el tiempo necesario a meses, incluso semanas



Beneficios

- Tolerancia a fallos
 - Si un nodo falla, los restantes pueden continuar operando
- Eficiencia
 - Permiten reducir el tiempo de entrenamiento añadiendo más nodos
- Escalabilidad
 - Los sistemas distribuidos permiten escalar el hardware añadiendo más nodos
- Coste
 - Es más barato tener un sistema con varios nodos, que un gran nodo con todos los recursos
 - A veces no es incluso posible tener un único nodo



Entrenamiento eficiente en una GPU

- Guía de Huggingface:

https://huggingface.co/docs/transformers/perf_train_gpu_on_e



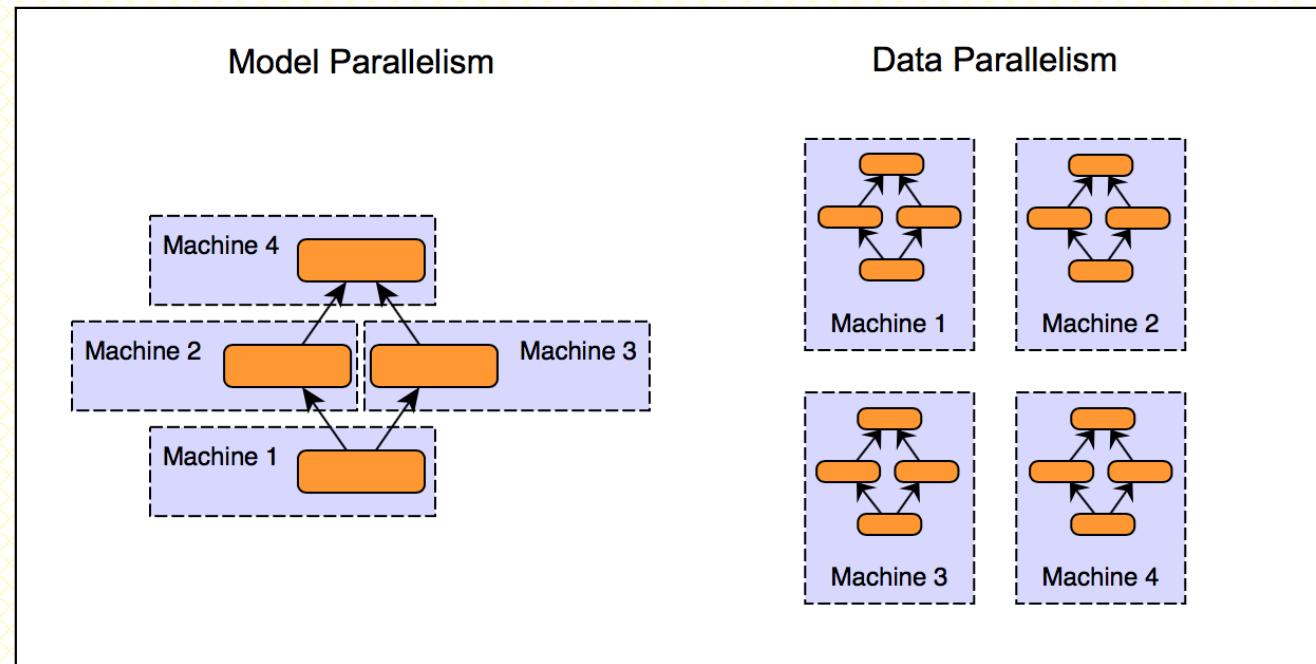
Entrenamiento distribuido

- Existen varios motivos para distribuir el entrenamiento
 - Memoria: Los modelos cada vez tienen más parámetros entrenables lo que implica un mayo consumo de memoria
 - Distribuir el entrenamiento es una estrategia para poder entrenar modelos que desborden la memoria de un solo dispositivo
 - La otra sería hacer un pipeline del modelo dividiéndole en trozos e ir cargándolo en la GPU poco a poco
 - Implica un trasiego de datos en ambas direcciones (CPU-GPU) constante
 - Tiempo de computación: El entrenamiento de modelos modernos a gran escala (Stable Diffusion, GPT-3, Dall-e, etc...) requerirían años de cómputo usando un solo dispositivo (o unos pocos).
 - Incluso considerando un dispositivo de gama alta, por ejemplo una Nvidia A100
 - “Training GPT-3 with 175 billion parameters would require approximately 36 years with 8 V100 GPU”
 - Ese tiempo se puede dividir, idealmente, por el número de dispositivos empleados de forma concurrente



Estrategias generales de entrenamiento distribuido

- A nivel teórico, el entrenamiento de un modelo puede explotar 3 tipos de paralelismo:
 - Paralelismo de datos (data parallelism). Mirrored, parameter server
 - Paralelismo de modelo (model parallelism): Pipeline, tensor
 - Paralelismo híbrido (data+model)



Fuente:
<https://xiandong79.github.io/Intro-Distributed-Deep-Learning>

Estrategias de entrenamiento distribuido

- Paralelismo de datos
- Paralelismo de modelo
- Paralelismo híbrido (modelo+distribuido)
- Centralizado/descentralizado
- Síncrono/asíncrono



Paralelismo de datos

- Si tenemos n trabajadores, dividimos el conjunto de datos en n partes
- Cada trabajador realiza el entrenamiento sobre su porción del conjunto de datos
 - Cada trabajador tiene una copia completa del modelo
- Los parámetros del modelo obtenidos por todos los trabajadores se integran siguiendo alguna técnica (síncrona/asíncronamente)



Paralelismo de datos

- El conjunto de datos de entrenamiento se reparte entre varios trabajadores (*workers*)
 - Cada uno de ellos entrena una epoch del modelo **completo** con ese sub-conjunto de datos
 - Se utiliza algún mecanismo para integrar los pesos calculados por cada trabajador y realizar un cálculo global de los pesos. Dos estrategias alternativas:
 - Espejo (*Mirrored*): Todos los trabajadores reconstruyen una copia espejo de los pesos globales resultantes al final de cada epoch
 - Esta estrategia requiere realizar operaciones de comunicación colectivas entre los trabajadores tipo **all-reduce**
 - Servidor de parámetros (*Parameter server*): Existen uno o varios trabajadores especializados que mantienen una copia centralizada de los valores globales de los parámetros
 - Al final de cada epoch se reconstruyen los pesos globales de forma centralizada en los servidores de parámetros
 - Al principio de la siguiente epoch, cada trabajador pide a los servidores de parámetros una copia actualizada de los pesos



Sincronización de pesos

- Varias aproximaciones posibles para combinar los pesos de varios trabajadores:
 - Calcular la media (la aproximación más simple)
 - Requiere una costosa sincronización al final de cada *epoch*
 - Alternativa, sincronizar cada n *epochs*: Compromiso entre rendimiento y precisión
 - Intercambiar los gradientes de los pesos y aplicarlos a los nuevos pesos
 - Abre la posibilidad de no requerir sincronización como el cálculo de la media
 - Puede provocar el problema del “*average gradient stale*”

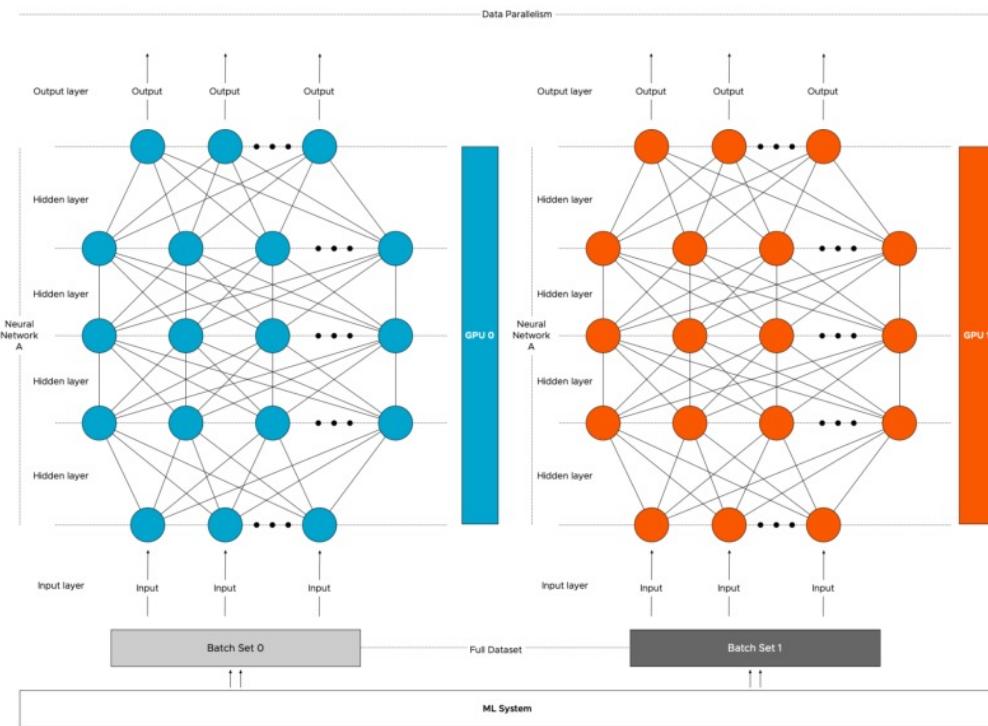


Paralelismo de datos

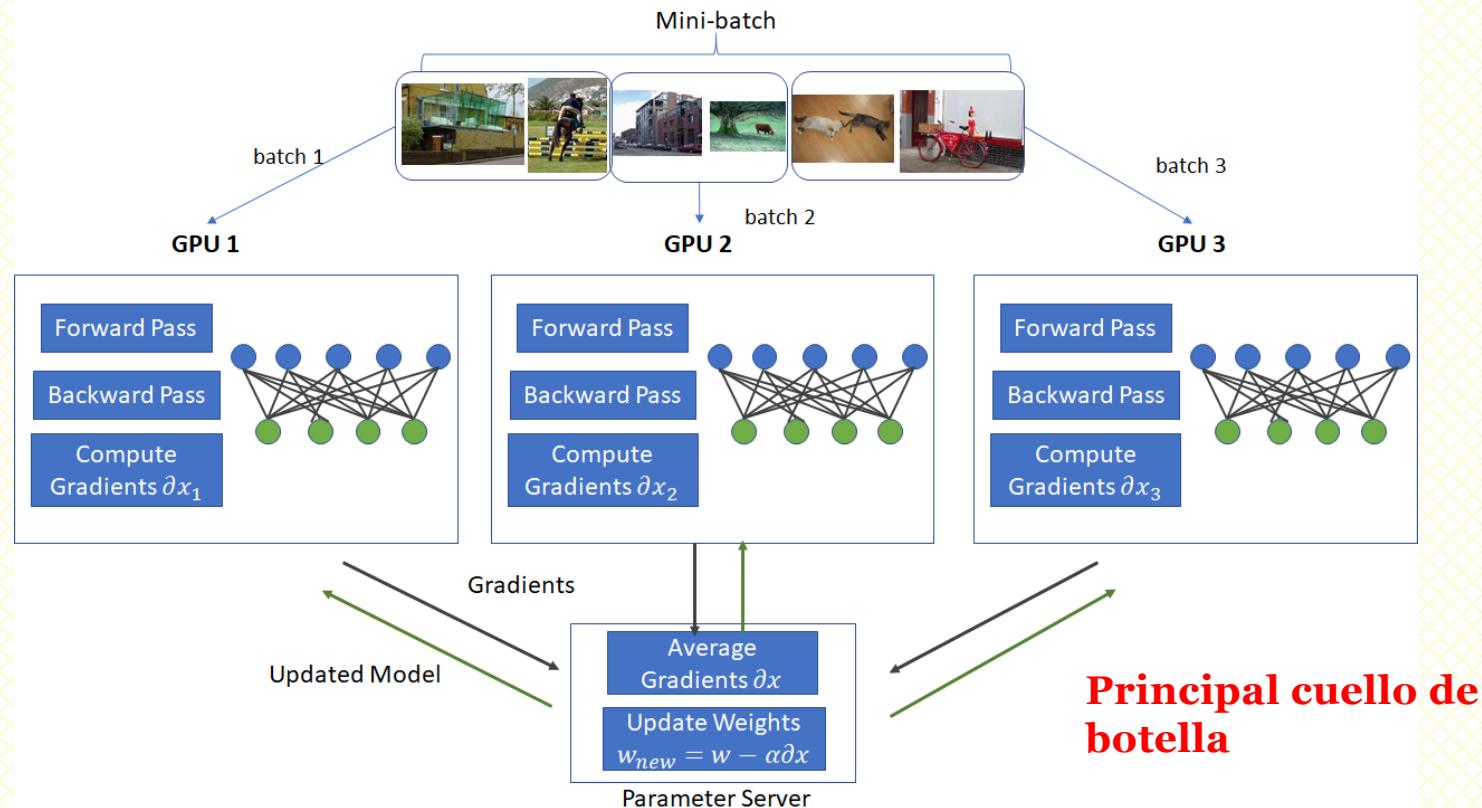
- Ventajas
 - Simplicidad
 - De hecho, es la técnica de entrenamiento distribuido más popular
- Desventajas
 - La sincronización de los trabajadores se puede convertir en un cuello de botella
 - Solo es posible si el modelo cabe en la memoria de cada trabajador
 - Si el trabajador es una GPU, el modelo debe caber en la memoria de la GPU
 - El tamaño del batch también se ve limitado por la cantidad de memoria disponible



Paralelismo de datos



Paralelismo de datos



Fuente: [Understanding Data Parallelism in Machine Learning](#)



Paralelismo de datos

- La forma de integrar los pesos supone un aspecto crítico que determina el rendimiento. Dos aproximaciones:
 - Entrenamiento síncrono
 - Ejemplo: All-Reduce
 - Entrenamiento asíncrono
 - Ejemplo: Parameter-Server



Entrenamiento síncrono

- Todos los trabajadores realizan la pasada *forward* a la vez
 - Cada trabajador computa sus propios pesos y salidas
 - Espera hasta que terminen todos los demás trabajadores
- Una vez todos los trabajadores han terminado su pasada *forward* intercambian su cálculo de los gradientes
- Los gradientes globales calculados cada trabajador ejecuta con ellos su pasada *backward* y actualizar su copia local del modelo
- Una vez todos los trabajadores han terminado la actualización de los pesos, se comienza el siguiente ciclo del proceso



Entrenamiento síncrono: all-reduce

- All-reduce: Es una aproximación popular para realizar reducciones que involucran a un número elevado de trabajadores
- Diferentes implementaciones:
 - Un trabajador centralizado:
 - Recibe los gradientes locales de todos los trabajadores
 - Computa los gradientes globales y se los envía a los otros trabajadores
 - Los trabajadores se organizan en una topología de anillo:
 - Cada uno recibe los gradientes del trabajador anterior (en el anillo), computa unos gradientes integrados y envía el resultado al siguiente trabajador (en el anillo)



Entrenamiento síncrono: all-reduce

- Cuello de botella: actualización de los gradientes por todos los *trabajadores*
 - Operación all-reduce
 - Los valores de un array se promedian a partir de los valores de la copia privada del array de los trabajadores
 - El array global, con sus valores calculados, se transfiere de vuelta a los trabajadores
 - Hay múltiples implementaciones del algoritmo all-reduce
 - Dependiendo de la topología de los trabajadores
 - El patrón de intercambio de valores



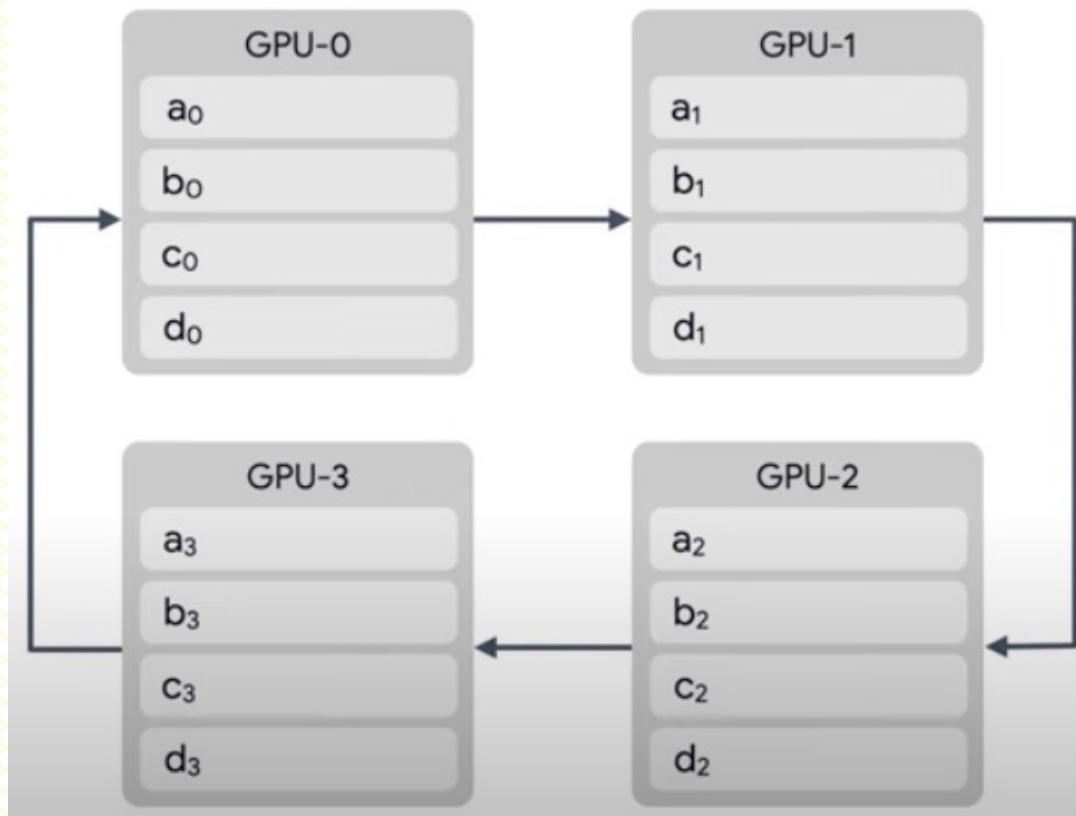
Entrenamiento síncrono: all-reduce ring

- El algoritmo all-reduce ring se compone de dos fases:
 - Reduce-scatter
 - All-gather



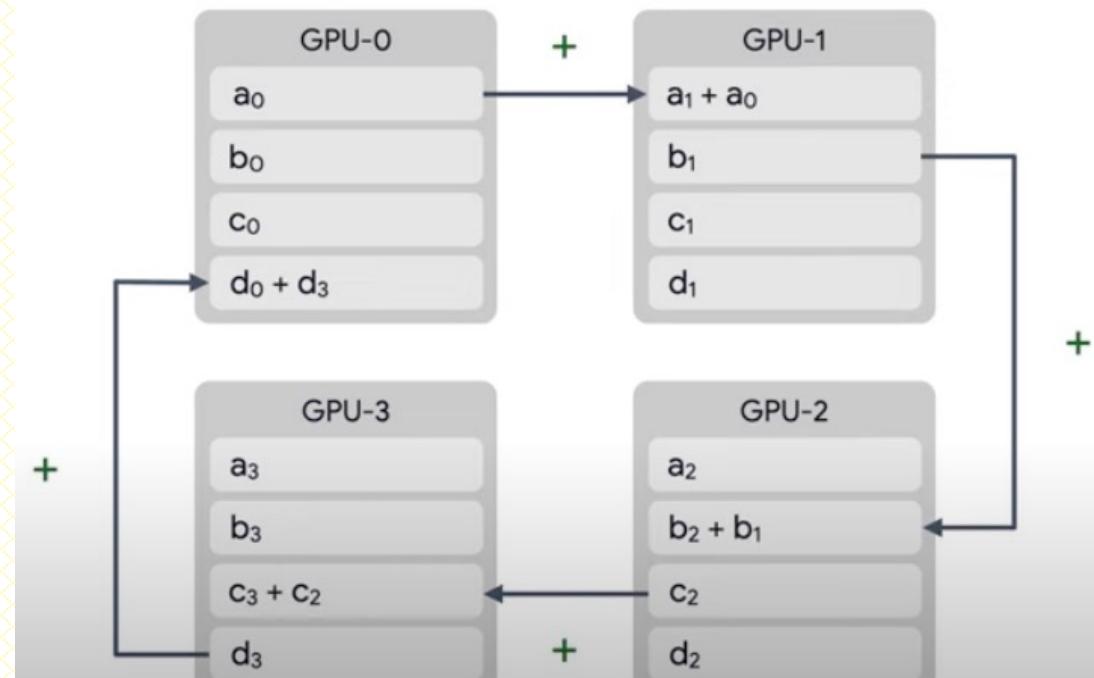
Entrenamiento síncrono: all-reduce ring

- El algoritmo all-reduce ring se compone de dos fases:
 - Reduce-scatter
 - All-gather



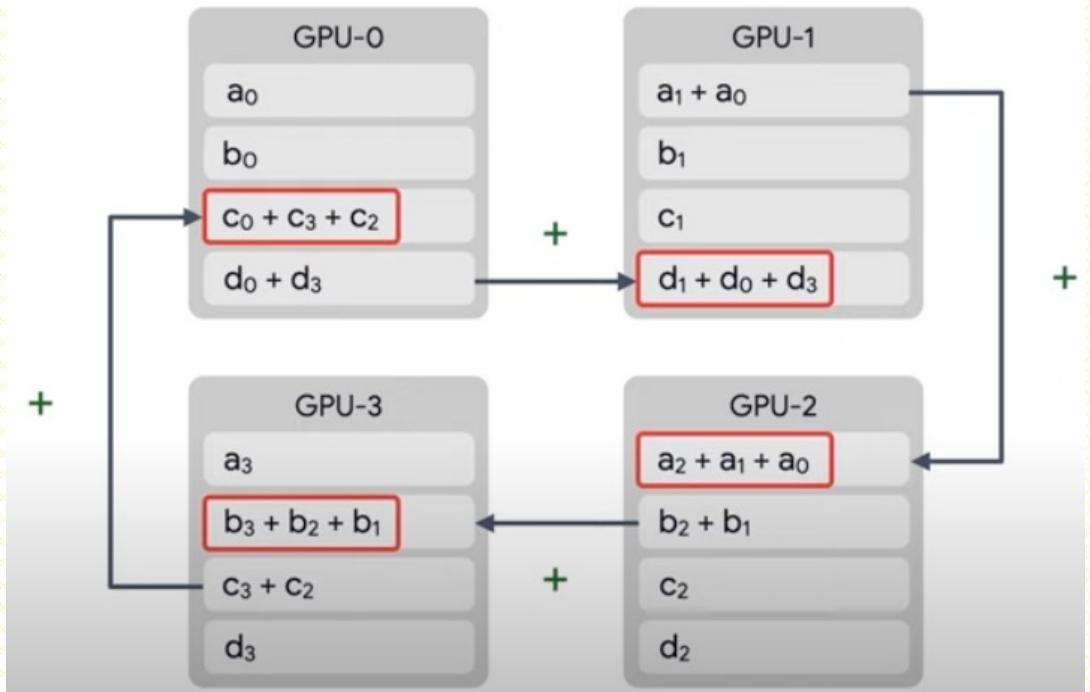
Entrenamiento síncrono: all-reduce ring

- El algoritmo all-reduce ring se compone de dos fases:
 - Reduce-scatter
 - All-gather



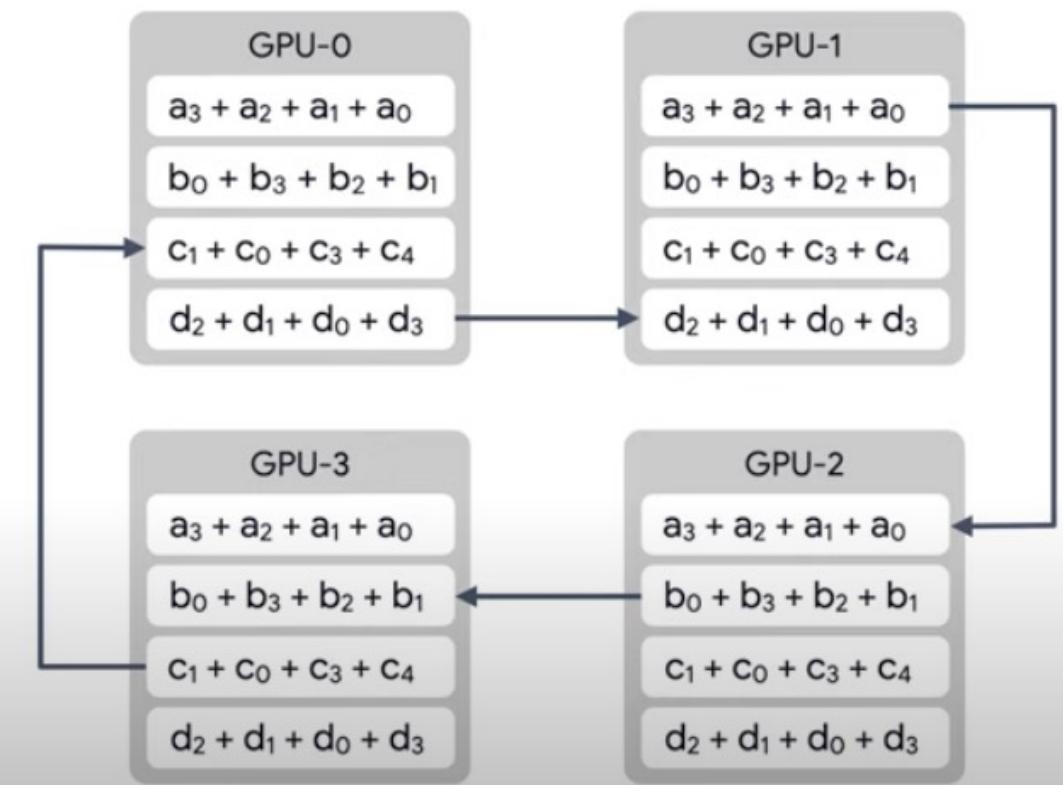
Entrenamiento síncrono: all-reduce ring

- El algoritmo all-reduce ring se compone de dos fases:
 - Reduce-scatter
 - All-gather



Entrenamiento síncrono: all-reduce ring

- El algoritmo all-reduce ring se compone de dos fases:
 - Reduce-scatter
 - All-gather



Entrenamiento asíncrono

- El punto de sincronización entre todos los trabajadores, requerido por la estrategia síncrona supone un cuello de botella
- Aproximación alternativa: entrenamiento asíncrono
 - Asigna roles diferentes a los trabajadores
 - Varios trabajadores
 - Uno o varios servidores de parámetros
 - Trabajadores:
 - Mantienen una copia completa del modelo
 - Piden el valor actualizado de los parámetros al modelo
 - Ejecutan el bucle de entrenamiento
 - Envía los gradientes calculados al servidor de parámetros



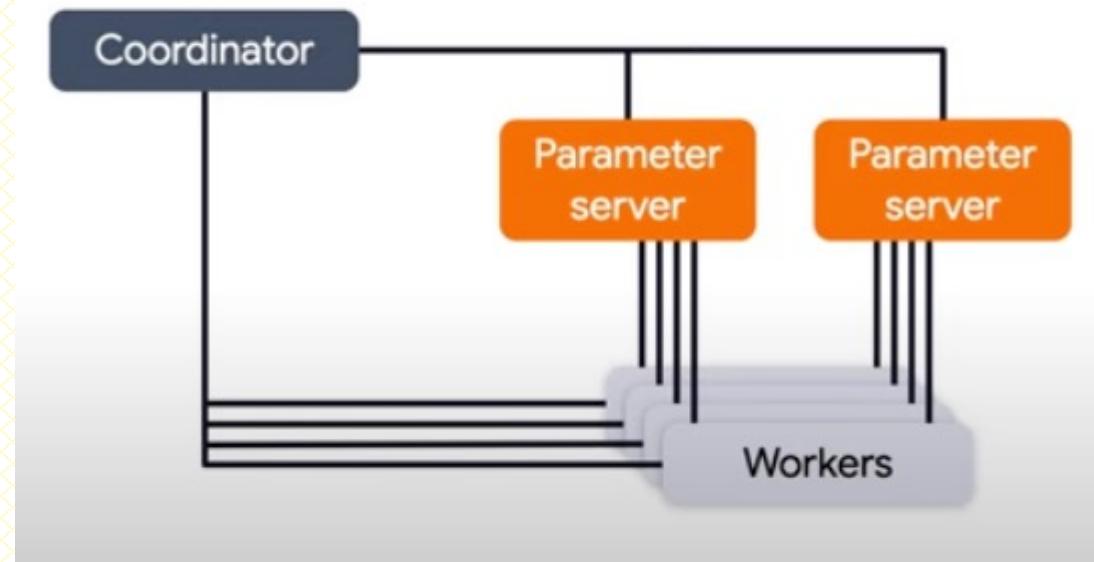
Entrenamiento asíncrono

- Servidor(es) de parámetros:
 - Reciben los gradientes calculados por cada trabajador
 - Actualizan sus parámetros locales en base a ellos
 - Bajo demanda, envía una copia de los parámetros cuando se lo requieren



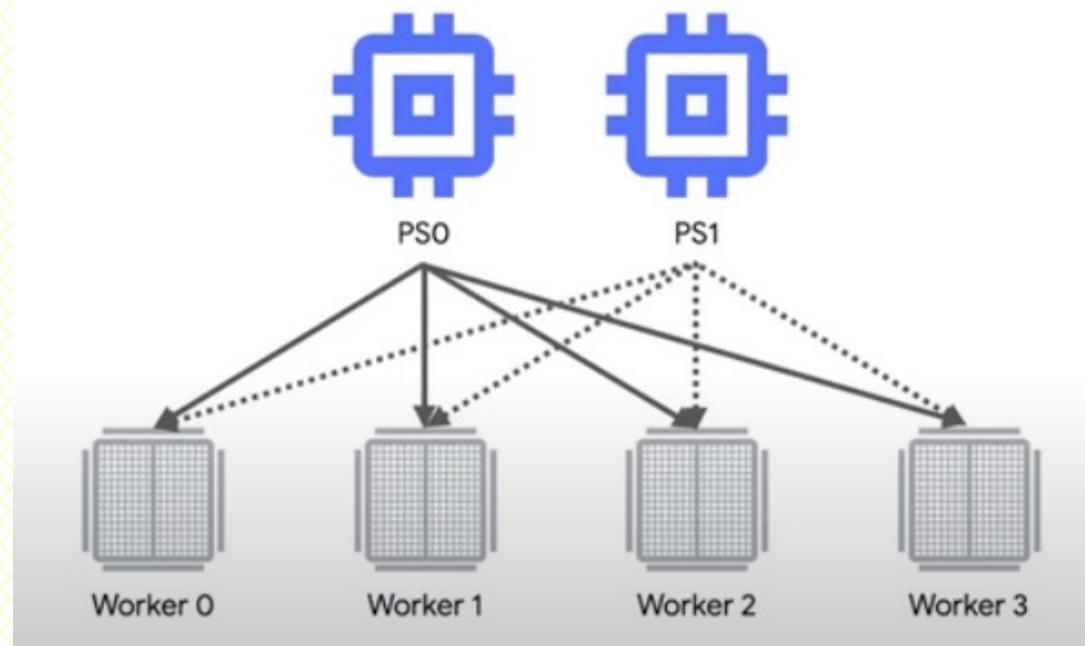
Servidor de parámetros

- Es un tipo de entrenamiento multimodo asíncrono
 - Reduce el cuello de botella del all-reduce en las estrategias síncronas
- Los nodos implicados se dividen en:
 - Trabajadores, *Workers*
 - Servidores de parámetros, *Parameter servers*
 - Un coordinador, *One coordinator*



Servidor de parámetros

- Cada trabajador pide la última copia de los parámetros a cada uno de los *parameter servers*
 - Los parámetros están distribuidos entre varios servidores
- Cada trabajador calcula los gradientes de acuerdo a un subconjunto del *dataset*
- Los trabajadores envían los parámetros de vuelta a los servidores de parámetros donde se integran



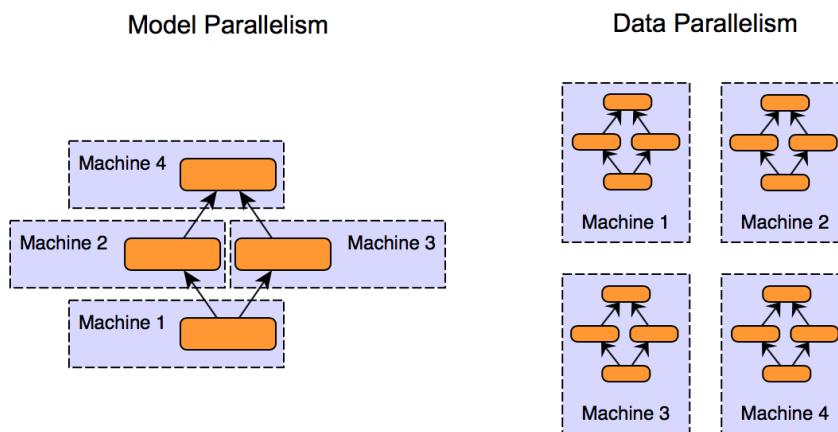
Entrenamiento asíncrono

- Ventajas:
 - Relaja la necesidad de mantener sincronizados a los trabajadores
 - Puede ser crítico con muchos trabajadores
- Desventajas:
 - Si los trabajadores progresan a velocidades diferentes, manejarán copias distintas del modelo
 - El uso de un único servidor de parámetros puede convertirlo en un cuello de botella
 - Solo es un problema en grandes infraestructuras
 - Atenuable mediante el uso de varios servidores de parámetros
 - Cada servidor mantiene un subconjunto de los parámetros del modelo



Paralelismo de modelo

- Paralelismo de modelo (*model parallelism*)
 - Dividimos la arquitectura del modelo entre varios trabajadores
 - Es necesario que las partes se puedan ejecutar de forma independiente
 - Más difícil de implementar conceptualmente, y depende de la arquitectura del modelo



Fuente: [Intro Distributed Deep Learning](#)

Paralelismo de modelo

- El uso del paralelismo de datos está limitado por la memoria disponible en un solo dispositivo acelerador
 - El modelo + los datos de un batch deben caber en la memoria del dispositivo
 - Una A100 tiene 40/80GB de memoria
- El paralelismo de modelos/red consiste en particionar el modelo (horizontal/verticalmente)
 - Cada trabajador entrena con los mismos datos distintas partes del modelo
 - La sincronización ocurre a través de los parámetros compartidos entre las distintas particiones
 - Una vez en la pasada *forward*
 - Una vez en la pasada *backward*



Paralelismo de modelo

- El tamaño del modelo está limitado por el número de parámetros que caben en la memoria de un dispositivo
 - Cada parámetro requiere aproximadamente 20 bytes
 - 10 billones de parámetros ocupan 186GBytes en memoria
 - GPT-3 tiene 175 billones
 - Dall-e tiene 12 billones
 - Stable Diffusion tiene 890 millones
 - Nvidia A100: **40**/80 GB de memoria
- La única forma de superar esta limitación es distribuyendo el modelo entre varios dispositivos

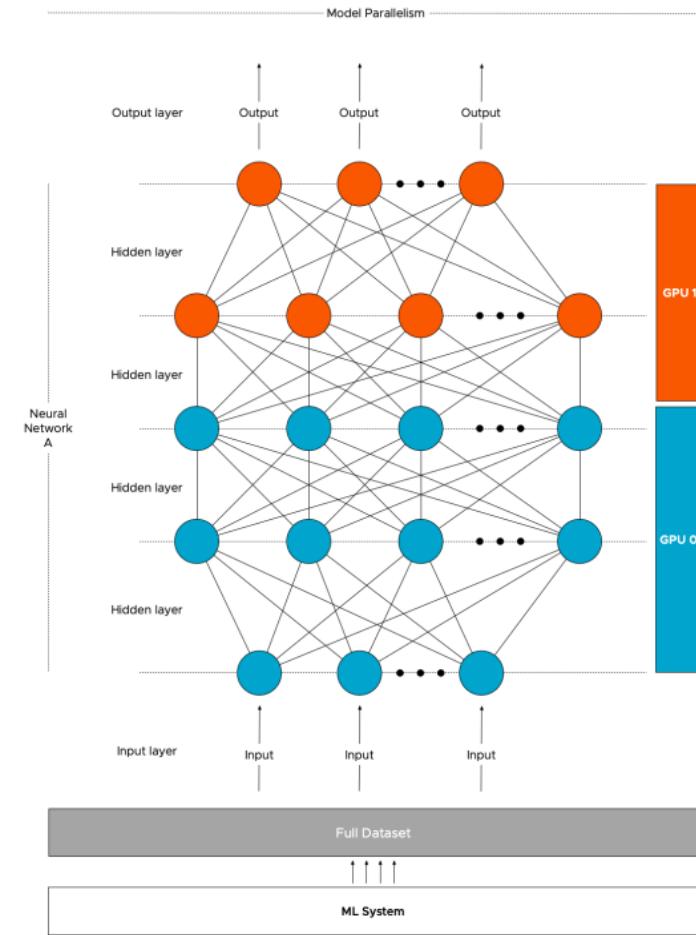
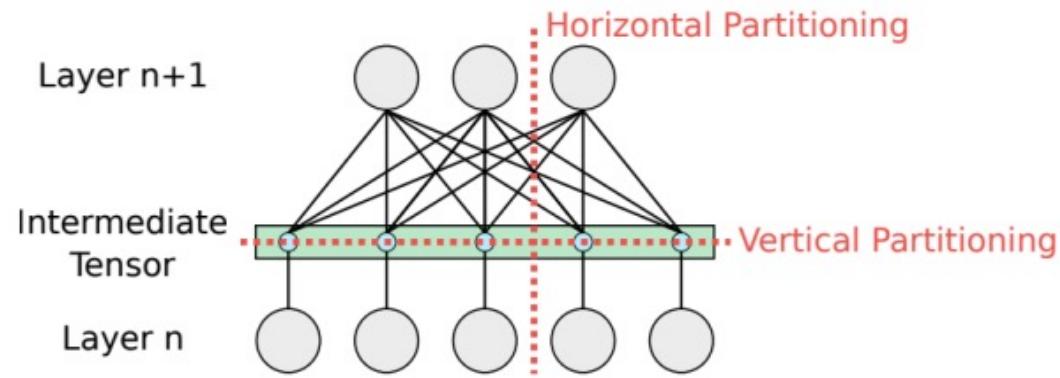


Paralelismo de modelo

- Aproximación *naive*
 - Particionamos las capas de modelos (capas completas) y las distribuimos entre varios dispositivos
 - Los dispositivos se van pasando los datos de unas a otras a medida que el modelo hace los cálculos hacia adelante (*forward-pass*)
 - Luego repiten el proceso análogamente hacia atrás (*backward-pass*)
 - Problemas de esta aproximación
 - Requiere comunicaciones frecuentes entre dispositivos, incluso ubicados en nodos diferentes
 - Solo uno de los dispositivos trabaja en cada momento, los demás están ociosos
 - No aporta paralelismo real
 - Eficaz solo para dar soporte a modelos que desbordan la memoria de un dispositivo



Paralelismo de modelo



Paralelismo de modelo

- Partición vertical
 - La opción más simple
 - No afecta a la configuración de las capas
 - Se puede aplicar a cada modelo
- Partición horizontal
 - Más difícil de implementar técnicamente
 - No aplicable a todos los modelos
- Otras particiones
 - En una arquitectura encoder/decoder, el encoder y el decoder puede ser entrenado por parte de trabajadores diferentes
 - Modelos de lenguaje

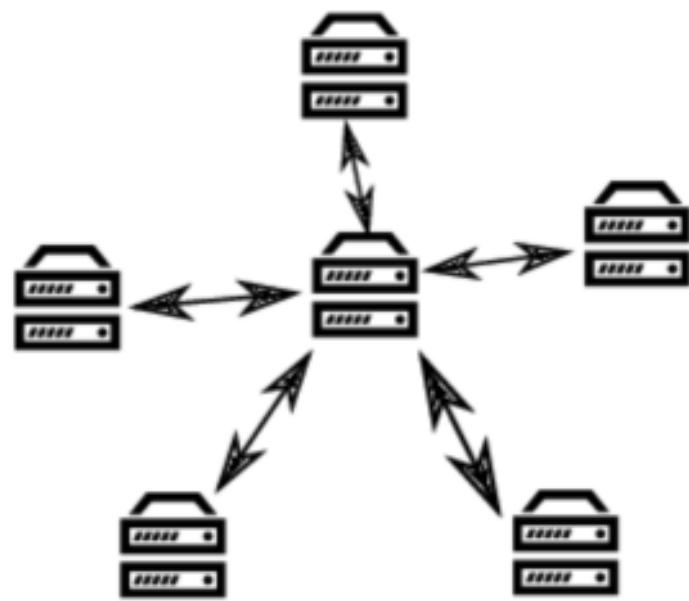


Paralelismo de modelo

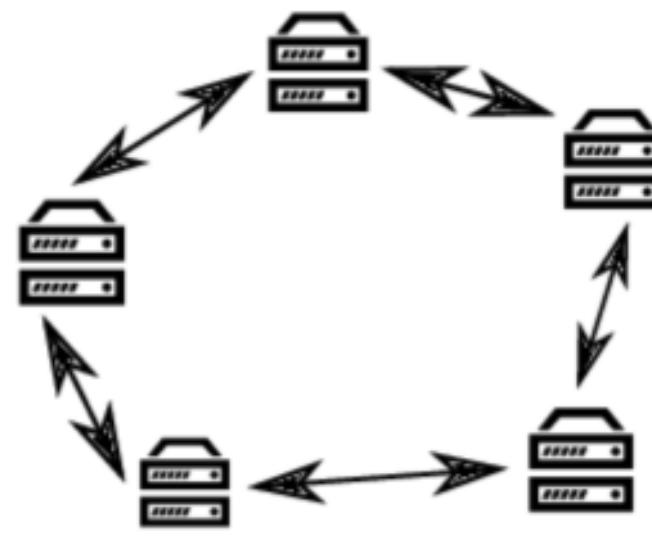
- El paralelismo de modelo también requiere el intercambio de información entre trabajadores. Los parámetros del modelo.
Dos opciones:
 - Centralizado: Existe uno o varios nodos dedicados a la actualización y almacenamiento de los parámetros del modelo (servidor de parámetros)
 - El servidor de parámetro puede actuar de forma
 - Asíncrona: Los parámetros se actualizan a medida que los gradientes van llegando
 - Síncrona:: Los parámetros se actualizan cuando todos los gradientes han llegado de todos los trabajadores
 - Descentralizado: Cada nodo intercambia sus gradientes con el trabajador vecino
 - Es posible hacerlo de forma síncrona o asíncrona



Paralelismo de modelo



(a) Parameter Server



(b) Decentralized Architecture

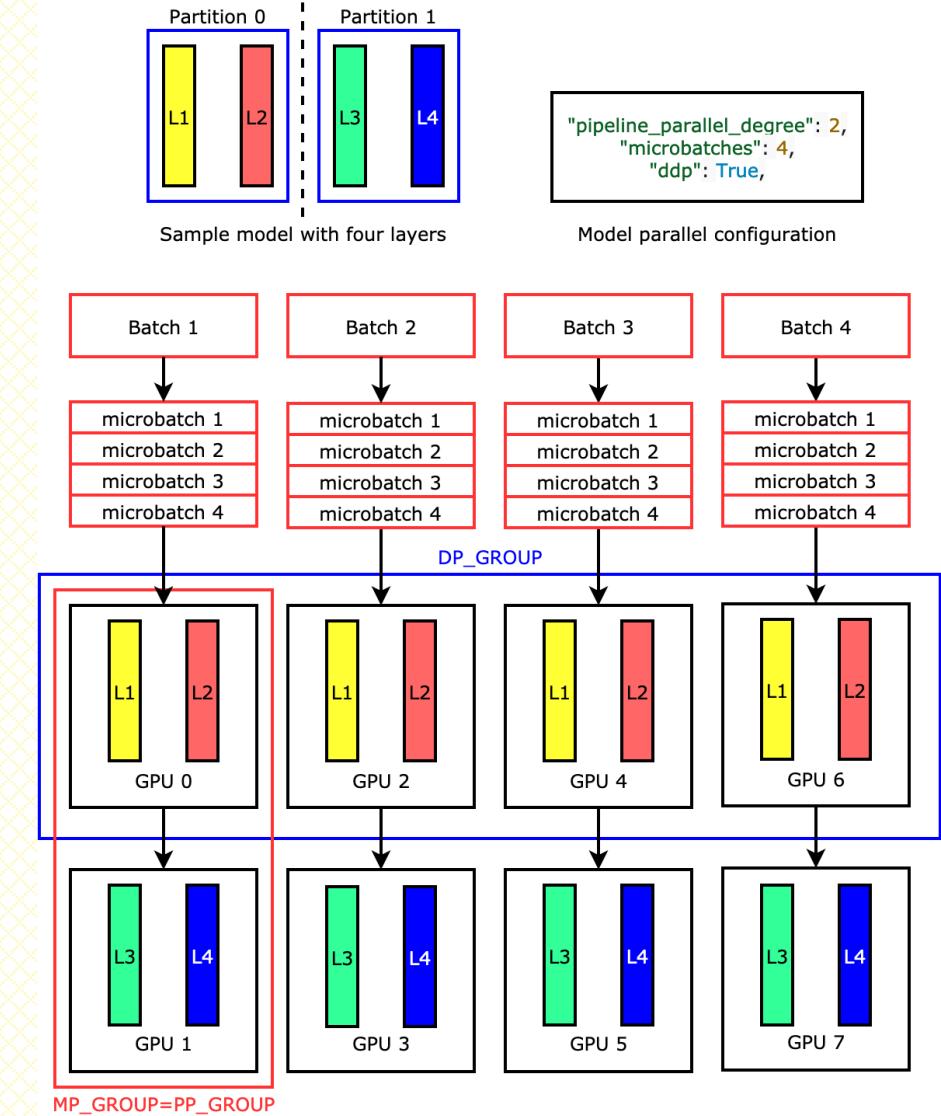
Introducción: paralelismo de modelo + híbrido

- Paralelismo de modelo + híbrido
 - Más difícil de explotar que el paralelismo de datos
 - Difícilmente generalizable para cualquier modelo
 - Desarrollos en fase experimental
 - O ad-hoc para ciertos modelos
 - Imprescindible para seguir escalando el tamaño de los modelos (no el de los *datasets*)
- Herramientas
 - Implementaciones ad-hoc para ciertos modelos
 - Bert, Stable-difusión, Bloom, etc...
 - Herramientas que escalan el entrenamiento en entornos distribuidos



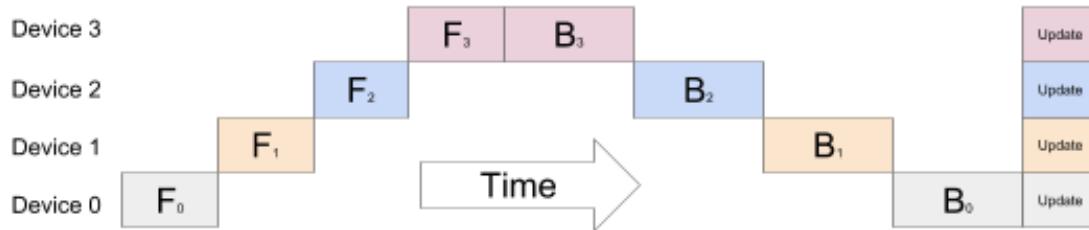
Paralelismo de modelo

- Aproximaciones
 - **Paralelismo de Pipeline:** Se partitiona el pipeline del modelo
 - Cada dispositivo va procesando una parte del modelo (*partition*) y una parte de los datos (*batch*)
 - El procesado de cada batch se subdivide en microbatches que mantiene a todos los dispositivos ocupados en todo momento
 - El número de microbatches es un hiperparámetro más del modelo
 - Epoch, Step, (Batch/Mini-batch) y Microbatch
 - Se crea un pipeline que aporta cierto paralelismo real respecto a la aproximación anterior
 - Cada partición es asignada a uno o varios dispositivos
 - Ejemplo de la figura: 2 particiones, 8 GPUs

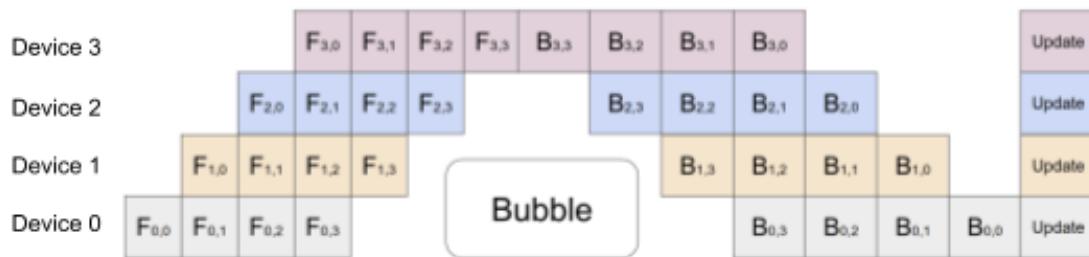


Fuente: <https://docs.aws.amazon.com/sagemaker/latest/dg/model-parallel-intro.html>

Naive vs Pipeline MP



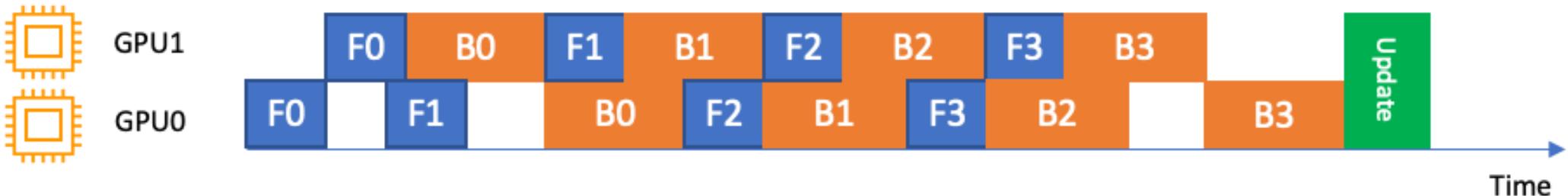
F es para *Forward*
B es para *Backward*



Top: The naive model parallelism strategy leads to severe underutilization due to the sequential nature of the network. Only one accelerator is active at a time. *Bottom:* GPipe divides the input mini-batch into smaller micro-batches, enabling different accelerators to work on separate micro-batches at the same time.

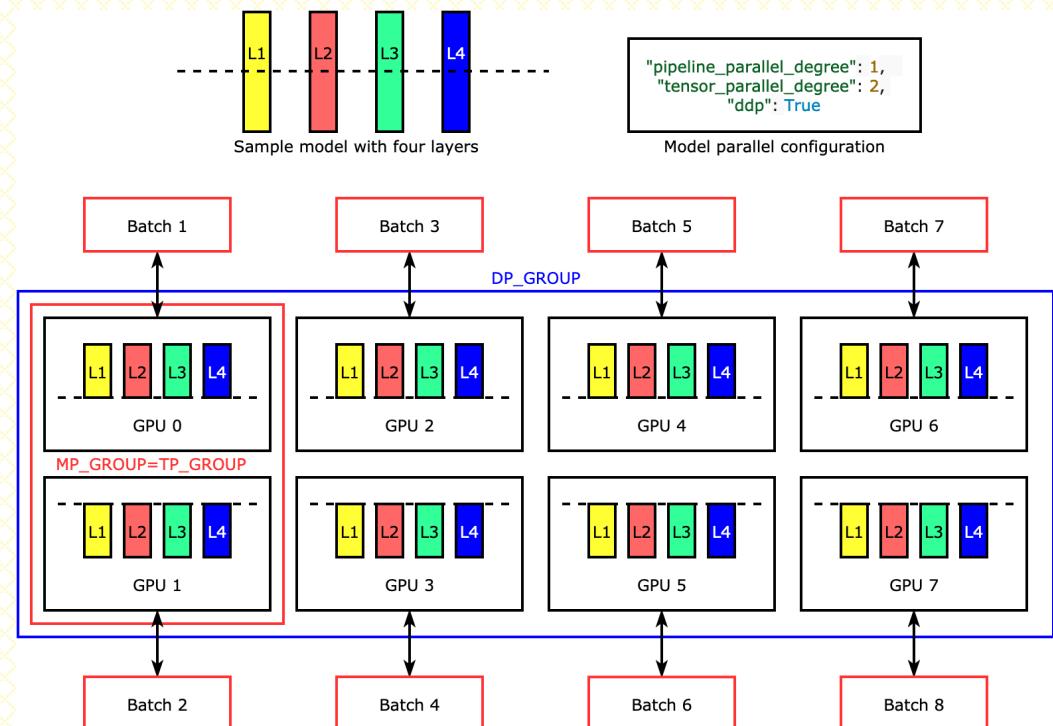
Pipeline Parallelism

- Aproximaciones más modernas como [DeepSpeed](#), [Varuna](#) o [SageMaker](#) (usable en AWS) implementan el concepto de *Interleaved Pipeline*
 - Elimina algunas esperas entrelazando las etapas *forward* y *backward*
 - Varuna hace simulaciones para encontrar la planificación de las tareas más eficiente



Paralelismo de modelo

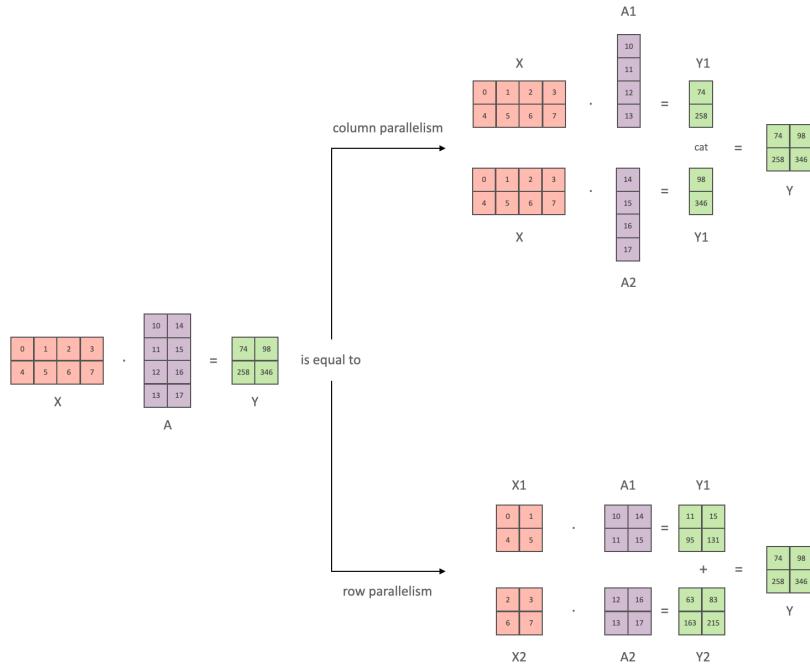
- Aproximaciones
 - **Paralelismo de Tensor:** Partitiona internamente las capas y las distribuye entre varios dispositivos
 - Ejemplo: 2 particiones/vías, 8 GPUs



Fuente: <https://docs.aws.amazon.com/sagemaker/latest/dg/model-parallel-intro.html>

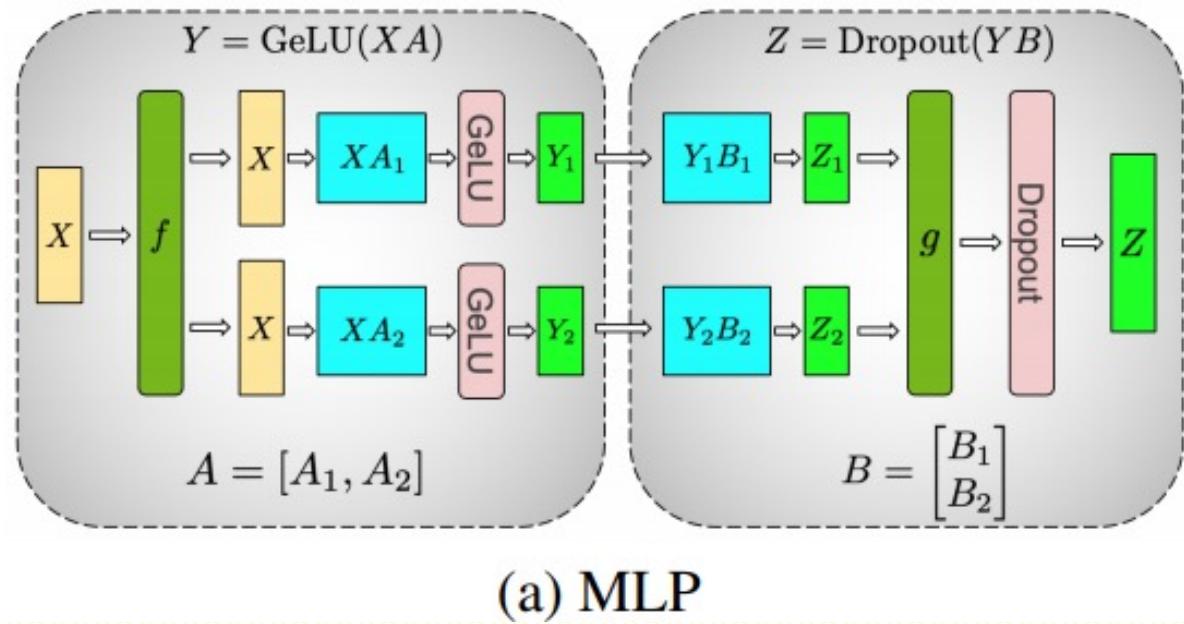


Paralelismo de tensor

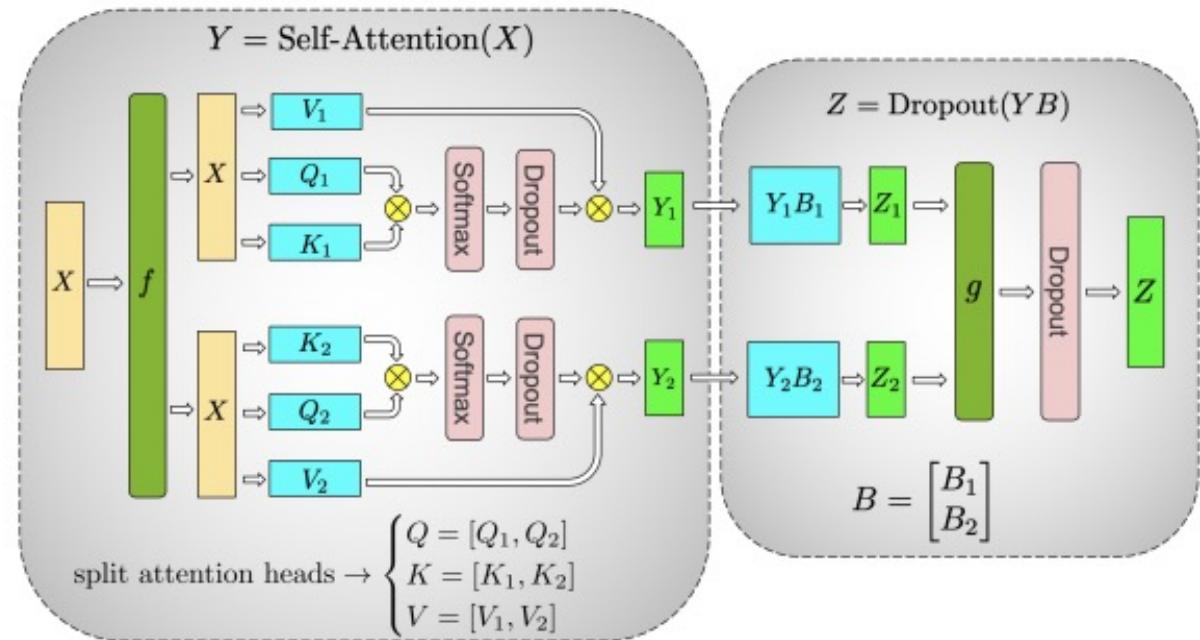


Hacer un GEMM y luego aplicar una activación GELU es un bloque constructor básico. $Y = \text{GeLU}(XA)$, donde X y Y son los vectores de entrada y salida, y A es la matriz de pesos. Se puede paralelizar distribuyendo por columnas, y la activación se puede aplicar independientemente sobre cada resultado parcial. Por filas requiere un punto de sincronización entre de GELU. [Fuente: paper Megatron](#)

- Podemos diseñar una estrategia para el modelo (Multi-layer Perceptron) MLP que no implique comunicación entre dispositivos hasta el final y que pueda encadenar dos GEMM.
 - Primer GEMM particionado por columnas
 - Segundo GEMM particionado por filas



- Podemos diseñar una estrategia análoga para un bloque self-attention completo de un Transformer
 - Los primeros GEMM asociados con Key (K), Query (Q) y Value (V) son particionados por columnas. Ejecutado localmente en una GPU.
 - El siguiente GEMM (de la capa lineal de salida) se paralleliza por filas y toma la salida directamente del paso anterior sin requerir sincronización

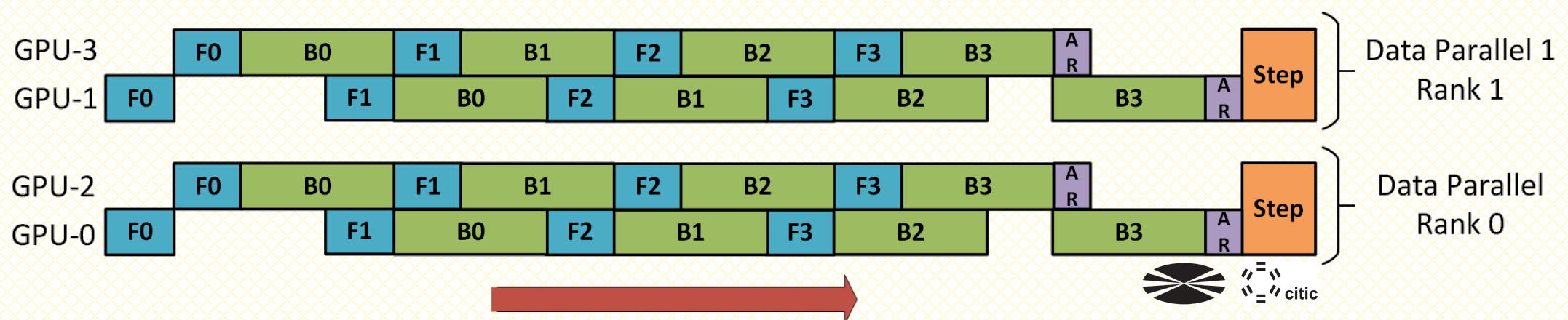


(b) Self-Attention

Paralelismo híbrido

- Paralelismo de datos + de modelo (pipeline)

Fuente: <https://huggingface.co/docs/transformers/v4.15.0/parallelism>

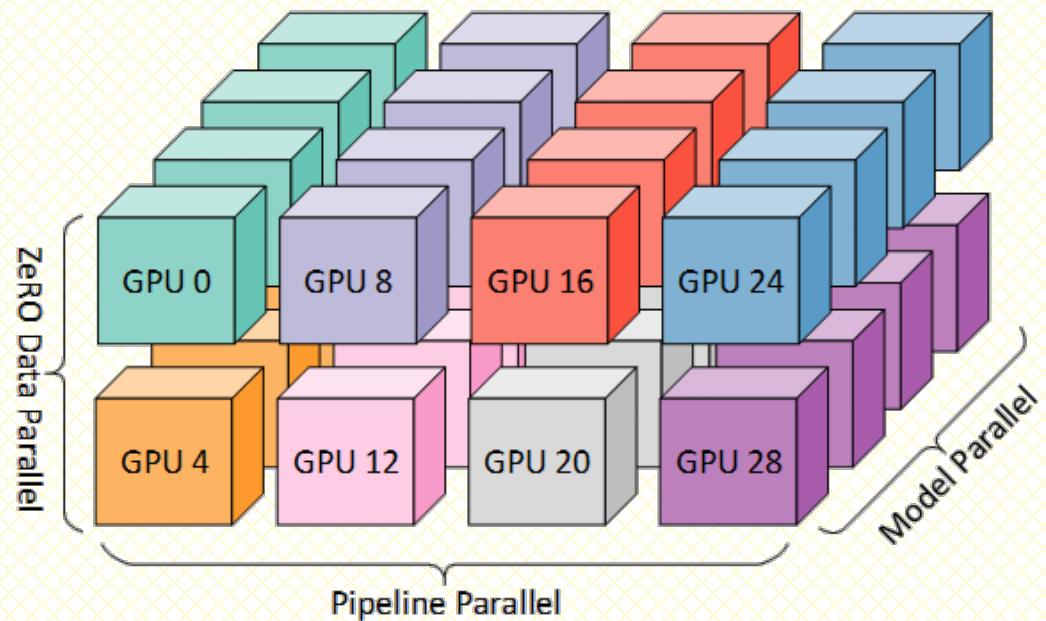


Paralelismo híbrido

- Paralelismo de datos + de modelo (pipeline) + tensorial

Fuente:

<https://huggingface.co/docs/transformers/v4.15.0/parallelism>



Paralelismo avanzado: Caso de estudio Hugginface

- Los modelos reales a gran escala requieren estrategias más complejas que las disponibles por defecto en TF (Mirrored, DistributedMirrored, Parameter Server) o Pytorch (DP, DDP)
 - A menudo estrategias híbridas que combinan paralelismo de datos y de modelo
- ZeRO DataParallel (Zero-DP)
 - Similar a DP pero cada GPU en vez de guardar una copia de todo el modelo, guarda sólo una porción de él
 - Cuando las GPUs necesitan trozos del modelo que no le son locales se produce un proceso de sincronización
 - Se puede entender como un parameter-server (PS) distribuido (donde todos los trabajadores actúan como PS)

Fuente :<https://huggingface.co/docs/transformers/v4.15.0/parallelism>



ZERO-DP: Ejemplo de juguete

- Dado un modelo con 3 capas (L_x), cada una con 3 pesos a_x

L_a	L_b	L_c
---	---	---
a_0	b_0	c_0
a_1	b_1	c_1
a_2	b_2	c_2

ZERO-DP: Ejemplo de juguete

- Dado un modelo con 3 capas (L_x), cada una con 3 pesos a_x
- Podemos distribuir el modelo entre 3 GPUs
 - Tensor-parallelism

GPU0:
La	Lb	Lc
a0 | b0 | c0

GPU1:
La	Lb	Lc
a1 | b1 | c1

GPU2:
La	Lb	Lc
a2 | b2 | c2



ZERO-DP: Ejemplo de juguete

- Dado un modelo con 3 capas (L_x), cada una con 3 pesos a_x
- Podemos distribuir el modelo entre 3 GPUs
 - Tensor-parallelism
- A su vez cada GPU recibe un mini-batch diferente
 - Data-parallelism

$x_0 \Rightarrow$

GPU $_0$

$x_1 \Rightarrow$ GPU $_1$

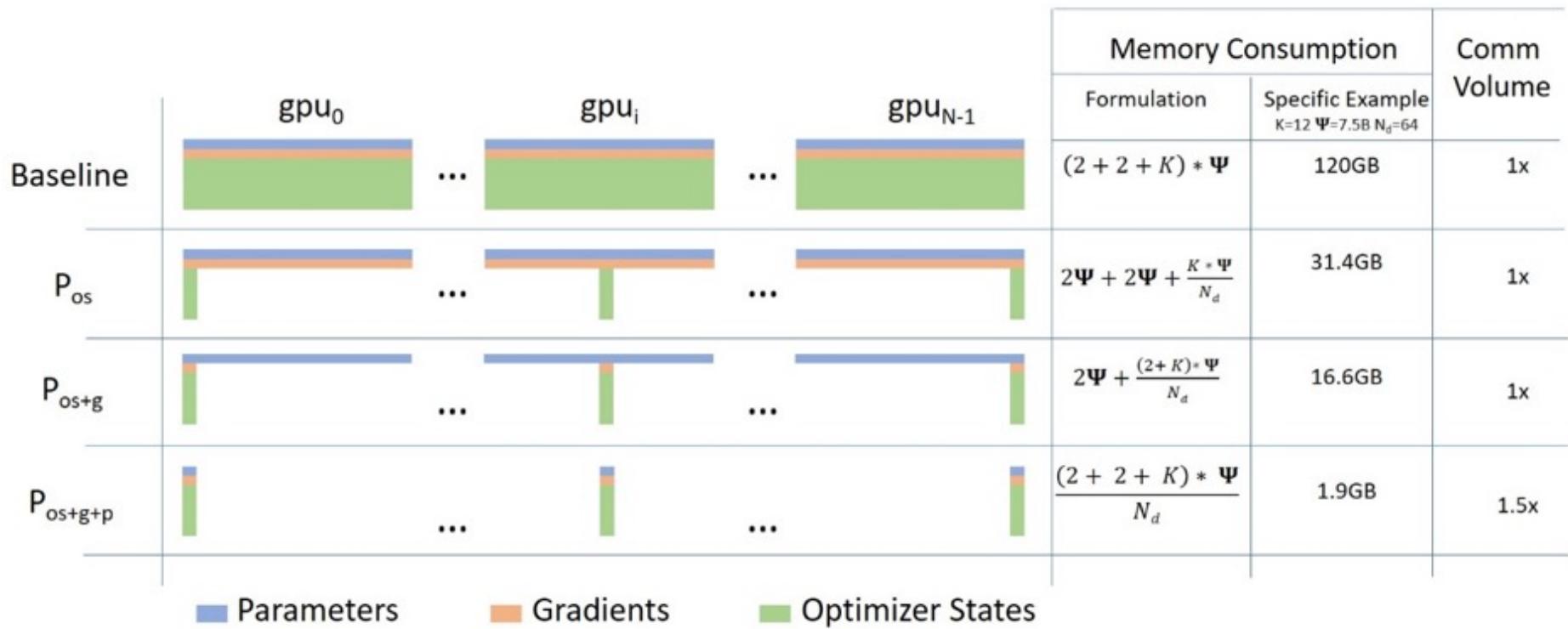
$x_2 \Rightarrow$ GPU $_2$



- Paso 1 *forward-pass*: Las entradas llegan a la primera capa (L_a)
 - En GPU0: para procesar el mini-batch x_0 necesita a_0 , a_1 y a_2 para hacer la *forward-pass*
 - Solo tiene localmente a_0 , con lo cual recibe a_1 de GPU1 y a_2 de GPU2
 - En GPU1 y GPU2 (concurrentemente) se procesan de forma análoga los minibatches x_1 y x_2
 - Cuando se terminan los cálculos de la *forward-pass*, los datos de las otras GPUs ya no se necesitan y se liberan
 - Se repite iterativamente el proceso para las otras 2 capas L_b y L_c , hasta que se termina la *forward-pass*
- Paso 2 *backward-pass*: Se repite el proceso de forma análoga pero hacia atrás



ZERO-DP



¿Qué estrategia utilizar?

Tenemos que tomar la decisión en función de cuántos recursos podemos emplear o cuántos queremos emplear en un entrenamiento

- Una GPU
- Un nodo – varias GPUs
- Varios Nodos – varias GPUs

Fuente:

<https://huggingface.co/docs/transformers/v4.15.0/parallelism#which-strategy-to-use-when>



Estrategia: Una GPU

- El modelo cabe en una GPU -> Entrenamiento convencional
- El modelo **no** cabe en una GPU ->
 - ZeRO + Offload CPU



Estrategia: Un nodo -> varias GPUs

- El modelo cabe en una GPU ->
 - Pytorch DDP
 - ZeRO
- El modelo **no** cabe en una GPU ->
 - Pipeline Paralelism (PP)
 - ZeRO
 - Tensor Parallelism
 - Indicado para nodos con conexiones NVLINK o NVSwitch entre las GPUs
 - Si no es el caso -> Zero o PP



Estrategia: Un nodo -> varias GPUs

- El modelo cabe en una GPU ->
 - Pytorch DDP
 - ZeRO
- El modelo **no** cabe en una GPU ->
 - Pipeline Paralelism (PP)
 - ZeRO
 - Tensor Parallelism
 - Indicado para nodos con conexiones NVLINK o NVSwitch entre las GPUs
 - Si no es el caso -> Zero o PP



Estrategia: Varios nodos -> varias GPUs

- Si tienes redes de interconexión de baja latencia y alto ancho de banda
 - ZeRO
 - PP+TP+DP
- Si no es el caso
 - DP+PP+TP+ZeRO-1



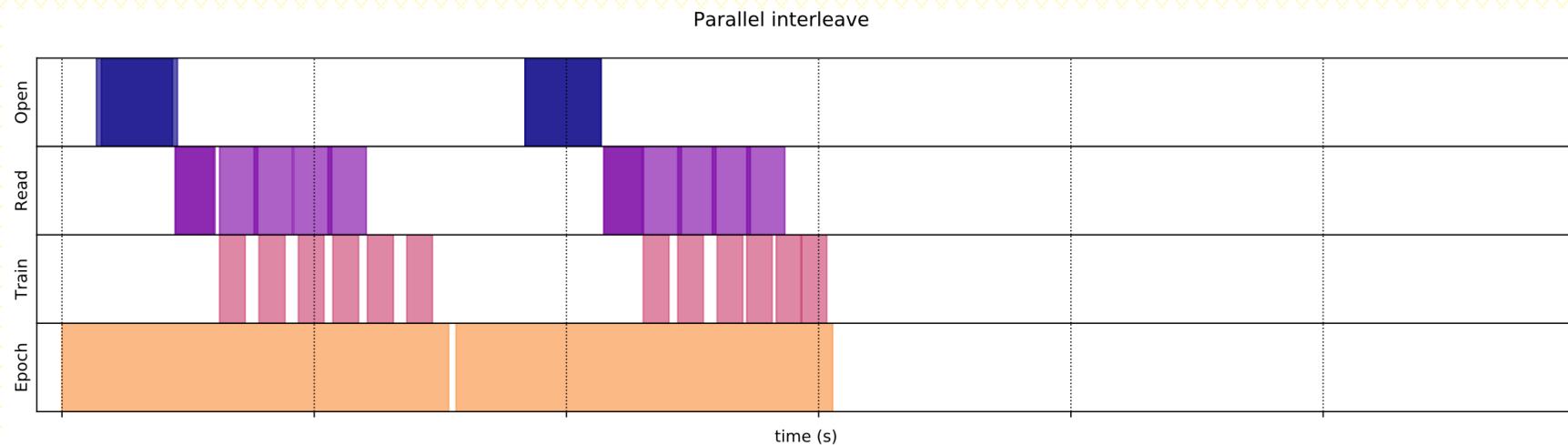
Concepto clave: Carga de datos

- Uno de los factores cruciales en el rendimiento de un pipeline de entrenamiento es que las GPUs estén trabajando la mayor parte del tiempo
- El uso de datos en ML implica
 - Abrir el fichero donde están los datos
 - Cargar o leer un dato (o conjunto de datos de ese fichero)
 - Usarlo en un proceso de entrenamiento
 - Puede implicar su envío a la GPU a través de un bus



Concepto clave: Carga de datos

- Es clave que no haya esperas por la carga de datos
 - Cargas asíncronas
 - Prebúsqueda (*prefetch*). Mientras se ejecuta un step de entrenamiento otro proceso busca concurrentemente los datos para el siguiente step.
 - Ajuste del tamaño de batch (o minibatch)
 - Carga distribuida. Varios trabajadores colaborando, o trabajadores que cargan solo datos que necesita en su nodo
 - Entrelazamiento (interleave) para realizar cargas entre varios trabajos dividiendo las cargas entre varios procesos
- Todos estos conceptos están implementados en Pytorch



Concepto clave: Carga de datos

- Es clave que no haya espera por la carga de los datos
 - Cargas asíncronas
 - Prebúsqueda (*prefetch*). Mientras se ejecuta un step de entrenamiento otro proceso ya busca los datos para el siguiente step.
 - Ajuste del tamaño de batch (o minibatch)
 - Carga distribuida
- Todos estos conceptos están implementados Pytorch

En FT3 es clave utilizar el sistema de ficheros que proporcione el mejor rendimiento: \$LUSTRE/\$STORE

Usar TensorBoard profiler para auditar si la carga de datos es un cuello de botella

Optimizadores para sistemas distribuidos

- El entrenamiento de modelos a gran escala (en entornos distribuidos) tiene en la sincronización de la actualización de los pesos un importante cuello de botella
- El método de optimización elegido es clave para atenuar este cuello de botella
- Los modelos a pequeña escala a menudo usan SGD
- En modelos a gran escala (NLP, Visión) es habitual usar variantes como ADAM (variante de SGD)
- El método de actualización de los pesos (síncrono o asíncrono) también determina el rendimiento



Optimizadores adaptativos

- Son variantes del SGD tradicional
 - Permiten mantener *Learning Rates* (LR) adaptados para cada peso, y que pueden cambiar a medida que el entrenamiento progresá
 - Adagrad, Adadelta y Adam son ejemplos de este tipo de optimizadores
 - Se presenta como una combinación de dos variantes de SGD
 - AdaGraD que introduce el concepto de LRs adaptados a cada parámetro
 - RMSProp que realizar la adaptación de los LR en base a cómo de rápido han cambiado últimamente los pesos



Minibatch SGD

- Minibatch Stochastic Gradient Descent (SGD) es una variante de GD utilizada para el entrenamiento de modelos de ML
 - Evolución: GD -> SGD -> Minibatch SGD
 - GD/Batch GD (Poco eficiente estadísticamente con *dataset* con entradas similares)
 - Se hace la *forward-pass* con todos los datos del dataset antes de ajustar los pesos con la *backpropagation*
 - SGD (Más eficiente estadísticamente, pero computacionalmente poco eficiente, no puede explotar vectorización eficientemente)
 - Se hace la *forward-pass* y la backpropagation para cada punto del *dataset*
 - Minibatch SGD: Aproximación intermedia
 - Se hace la forward-pass para b puntos del data set, y luego la back-propagation
 - b puede denominarse *batch* o *mini-batch*
 - Ganamos rendimiento respecto a SGD
 - Ganamos precisión respecto a Batch GD

Fuente: [11.5. Minibatch Stochastic Gradient Descent](#)



Minibatch SGD

- Procesar las entradas en minibatches de observaciones individuales permite convertir operaciones matriz-vector (o vector-vector) en operaciones matriz-matriz, mucho más
 - Eficientes
 - Paralelizables
- Proporciona lo mejor de GD (eficiencia computacional) y SGD (eficiencia estadística)



Distributed training en Pytorch

- Paquete de referencia: `torch.distributed`
- Tres componentes principales:
 - Distributed Data-Parallel Training (DDP)
 - Similar a MirroredStrategy
 - Se replica el modelo en cada proceso
 - Cada instancia del modelo se alimenta con diferentes trozos del *data set*
 - Se comunican los gradientes entre réplicas para que los pesos de las instancias estén sincronizados
 - RPC-based Distributed Training (RPC)
 - Da soporte a estructuras de entrenamiento generales que no son soportadas por DDP, tales como:
 - Distributed Pipeline Parallelism
 - Parameter Server
 - Híbrido
 - Collect Communication (c10d)
 - Librería que da soporte a operaciones colectivas frecuentes en entrenamiento distribuido.
 - All_reduce, all_gather, send, isend, etc...

Fuente: [PyTorch Distributed Overview — PyTorch Tutorials 1.13.0+cu117 documentation](#)



Launchers

- En el ecosistema de pytorch hay varios lanzadores (*launchers*) aptos para entornos distribuidos
 - Lanzar el código directamente con el intérprete de Python
 - La configuración del lanzamiento distribuido debe ser hecha desde el propio código Python
 - *torch.distributed.launch*: es un módulo que permite el lanzamiento de múltiples procesos de entrenamiento concurrentes (**deprecated**)
 - Ejemplo:

```
python -m torch.distributed.launch --nproc-per-node=NUM_GPUS_YOU_HAVE
    --nnodes=2 --node-rank=0 --master-addr="192.168.1.1"
    --master-port=1234 YOUR_TRAINING_SCRIPT.py (--arg1 --arg2 --arg3
    and all other arguments of your training script)
```



Launchers

- En el ecosistema de pytorch hay varios lanzadores (*launchers*) aptos para entornos distribuidos
 - *torchrun*: Es la opción más actual de lanzador nativo de Pytorch
 - Proporciona un superconjunto de las funcionalidades de *torch.distributed.launch*
 - Además:
 - Proporciona tolerancia a fallos: si un trabajador falla los demás pueden ser reiniciados
 - Los identificadores RANK y WORLD_SIZE se asignan automáticamente
 - Se puede especificar un número flexible de trabajadores dentro de un rango con un límite inferior y otro superior



Launchers

- Torchrun: Asignación automática de identificadores de cada trabajador

```
torch.distributed.launch
```

```
import argparse
parser = argparse.ArgumentParser()
parser.add_argument("--local_rank",
                    type=int)
args = parser.parse_args()

local_rank = args.local_rank
```

```
torchrun
```

```
import os
local_rank =
int(os.environ["LOCAL_RANK"])
```

```
local_rank = int(os.environ["LOCAL_RANK"])
model = torch.nn.parallel.DistributedDataParallel(model,
                                                 device_ids=[local_rank],
                                                 output_device=local_rank)
```



Launchers

- Torchrun: Ejemplo de lanzamiento

```
torchrun
  --nnodes=1:4
  --nproc-per-node=$NUM_TRAINERS
  --max-restarts=3
  --rdzv-id=$JOB_ID
  --rdzv-backend=c10d
  --rdzv-endpoint=$HOST_NODE_ADDR
YOUR_TRAINING_SCRIPT.py (--arg1 ... train script args...)
```



Launchers

- Torchrun: Definiciones

Node

Worker

WorkerGroup

LocalWorkerGroup

RANK

WORLD_SIZE

LOCAL_RANK

LOCAL_WORLD_SIZE

rdzv_id

rdzv_backend

rdzv_endpoint



Launchers

- Torchrun: Variables de entorno

LOCAL_RANK

RANK

GROUP_RANK

ROLE_RANK

LOCAL_WORLD_SIZE

WORLD_SIZE

MASTER_ADDR

MASTER_PORT

...



Launchers

- **Torchx:** Es un launcher universal para aplicaciones Pytorch
 - Desarrollo muy reciente (versión 0.5.0)
 - Basados en dos conceptos clave:
 - Planificadores (*schedulers*) aptos para la ejecución sobre varios entornos:
 - Local, Docker, Kubernetes, Slurm, Ray, AWS, ...
 - Componentes que permiten la ejecución de distintos tipos de aplicaciones:
 - Carga de Datos, Entrenamiento DDP, Paralelismo de Modelo, etc...

Ejemplo:

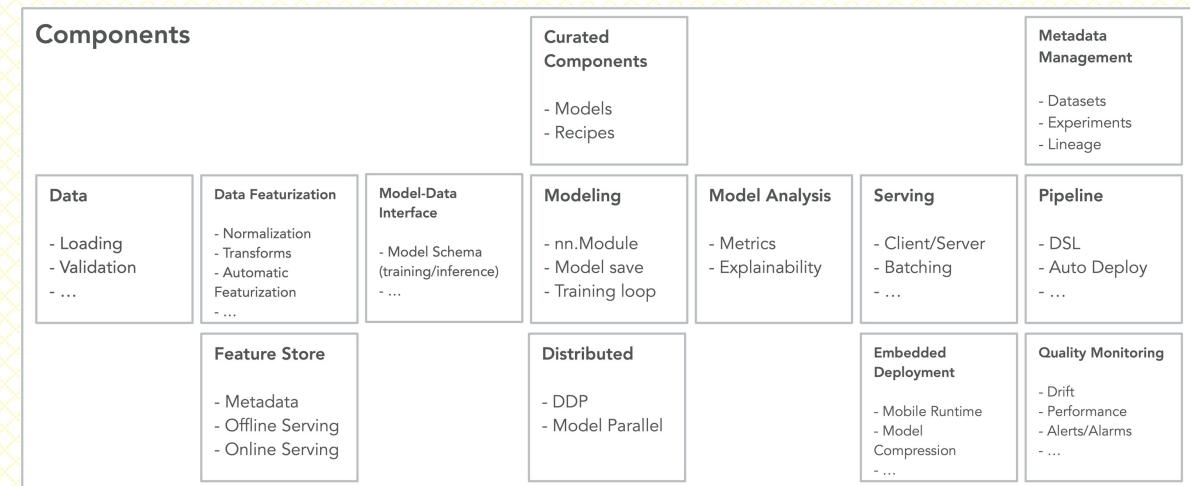
```
torchx run -s slurm dist.ddp --j 2x2 --gpus 2 script.py params
```

Fuente: <https://pytorch.org/torchx/latest/quickstart.html>



Launchers

- Torchx: Componentes
 - Hay muchos componentes disponibles
 - Hay un mecanismo bien documentado para crear componentes propios
 - También basados en los que ya hay



Launchers

- Torchx: Componentes

```
\$ torchx builtins, cancel, configure, describe, list, log, run, runop  
● (mytorchdist) [ulcesdac@login211-1 dac]$ torchx builtins  
Found 11 builtin components:  
 1. dist.ddp  
 2. dist.spmd  
 3. metrics.tensorboard  
 4. serve.torchserve  
 5. utils.binary  
 6. utils.booth  
 7. utils.copy  
 8. utils.echo  
 9. utils.python  
10. utils.sh  
11. utils.touch
```



Distributed training en Pytorch

- Elementos comunes
 - Varias soluciones basadas en replicar el contenedor del modelo (*Module*): DataParallel, DistributedDataParallel



Distributed training en Pytorch: Data parallel training

- Varias opciones disponibles que cubren todo el ciclo de desarrollo de simple a complejos:
 - Single-device
 - 1-nodo multi-GPU: DataParallel
 - Poco cambio en el código
 - 1-nodo multi-GPU: DistributedDataParallel
 - Más cambios en el código
 - N-nodos:
 - Basado en DistributedDataParallel
 - Script específico de lanzamiento
 - N-nodos (variable):
 - torch.distributed.elastic
 - Se usa si los recursos (GPUs/nodos) pueden unirse y desaparecer dinámicamente
 - O si se esperan errores tipo *out-of-memory*



Distributed training en Pytorch: DataParallel

- Paraleliza la aplicación dividiendo y distribuyendo la entrada entre varios dispositivos
 - La división se realiza por la dimensión batch
 - `batch_size > n_gpus` (y mejor si es divisible)
 - `(Tamaño del modelo + tamaño del batch local) < memoria de cada dispositivo`

Uso sencillo:

```
○ ○ ○  
1 net = torch.nn.DataParallel(model, device_ids=[0, 1, 2])  
2 output = net(input_var) # input_var can be on any device, including CPU
```



Distributed training en Pytorch: DataParallel

- En la pasada Forward, se replica el módulo (contenedor) del modelo en cada dispositivo disponible en un nodo
 - Cada réplica recibe una porción del conjunto de datos (*dataset*)
 - Los parámetros del módulo deben estar localizados en el dispositivo o del nodo antes de ejecutar el módulo
 - Por defecto, los tensores de entrada se dividen por la dimensión cero
 - Dimensión *batch*
 - Para las demás tipos de estructuras se hará una copia en profundidad
- Durante la pasada Backward, los gradientes calculados por cada réplica del modelo se suman en el módulo original (el ubicado en el dispositivo o)



Distributed training en Pytorch: DistributedDataParallel

- Basado en el paquete torch.distributed
- Soporta varios nodos y varios dispositivos por nodo
- Requiere un paso más que DataParallel
 - [torch.distributed.init_process_group\(\)](#).

```
○ ○ ○  
1 torch.cuda.set_device(i)  
2 torch.distributed.init_process_group(  
3     backend='nccl', world_size=N, init_method='...'  
4 )  
5 model = DistributedDataParallel(model, device_ids=[i], output_device=i)
```



Torch.distributed backends

- Vemos que al inicializar el grupo de procesos podemos configurar el backend para las comunicaciones. Existen varias opciones:
 - Gloo
 - Mpi (opcional)
 - Nccl (el más rápido con GPUs de Nvidia)
- No todas las operaciones colectivas de comunicación están disponibles para todos



Backend	gloo		mpi		nccl	
Device	CPU	GPU	CPU	GPU	CPU	GPU
send	✓	✗	✓	?	✗	✓
recv	✓	✗	✓	?	✗	✓
broadcast	✓	✓	✓	?	✗	✓
all_reduce	✓	✓	✓	?	✗	✓
reduce	✓	✗	✓	?	✗	✓
all_gather	✓	✗	✓	?	✗	✓
gather	✓	✗	✓	?	✗	✓
scatter	✓	✗	✓	?	✗	✓



Reglas-del-pulgar backend

- Varias GPUs de Nvidia -> NCCL
 - Más si cabe en sistemas con Infiniband
- Varias CPUs (distribuido) -> Gloo



Timeout

- Configurable al ejecutar el método *init_process_group*
 - Si el valor no es suficientemente alto y hay desbalanceo de carga entre los trabajadores -> Error en tiempo de ejecución



Checkpointing

- Ya que el proceso de entrenamiento distribuido tiende a producir errores en tiempo de ejecución, es bueno guardar el estado del modelo cada cierto número de epochs (*save*) para poder restaurarlo si fuese necesario (*load*)

Ejemplo:

[https://pytorch.org/tutorials/intermediate/ddp_tutorial.html#s
ave-and-load-checkpoints](https://pytorch.org/tutorials/intermediate/ddp_tutorial.html#save-and-load-checkpoints)



DDP + Paralelismo de modelo

- Podemos crear un modelo en el que repartimos los parámetros del modelo entre varios dispositivos

```
class ToyMpModel(nn.Module):
    def __init__(self, dev0, dev1):
        super(ToyMpModel, self).__init__()
        self.dev0 = dev0
        self.dev1 = dev1
        self.net1 = torch.nn.Linear(10, 10).to(dev0)
        self.relu = torch.nn.ReLU()
        self.net2 = torch.nn.Linear(10, 5).to(dev1)

    def forward(self, x):
        x = x.to(self.dev0)
        x = self.relu(self.net1(x))
        x = x.to(self.dev1)
        return self.net2(x)
```



```
def demo_model_parallel(rank, world_size):
    print(f"Running DDP with model parallel example on rank {rank}.")
    setup(rank, world_size)

    # setup mp_model and devices for this process
    dev0 = rank * 2
    dev1 = rank * 2 + 1
    mp_model = ToyMpModel(dev0, dev1)
    ddp_mp_model = DDP(mp_model)

    loss_fn = nn.MSELoss()
    optimizer = optim.SGD(ddp_mp_model.parameters(), lr=0.001)

    optimizer.zero_grad()
    # outputs will be on dev1
    outputs = ddp_mp_model(torch.randn(20, 10))
    labels = torch.randn(20, 5).to(dev1)
    loss_fn(outputs, labels).backward()
    optimizer.step()

cleanup()
```



```
if __name__ == "__main__":
    n_gpus = torch.cuda.device_count()
    assert n_gpus >= 2, f"Requires at least 2 GPUs to run, but got {n_gpus}"
    world_size = n_gpus
    run_demo(demo_basic, world_size)
    run_demo(demo_checkpoint, world_size)
    world_size = n_gpus//2
    run_demo(demo_model_parallel, world_size)
```



DataParallel vs DDP

- Es recomendable el uso de DDP también para un nodo con varios dispositivo (en vez de DataParallel)
 - DataParallel usa un único proceso, varios hilos y funciona en un único nodo
 - Problema de contención GIL (global interpreter lock) entre hilos
 - La implementación Cpython no es thread-safe, así que los hilos de un mismo proceso Python se serializarán, incluso en una plataforma multinúcleo
 - DDP funciona con paralelismo de modelo
 - Indispensable cuando el modelo no cabe en un único dispositivo



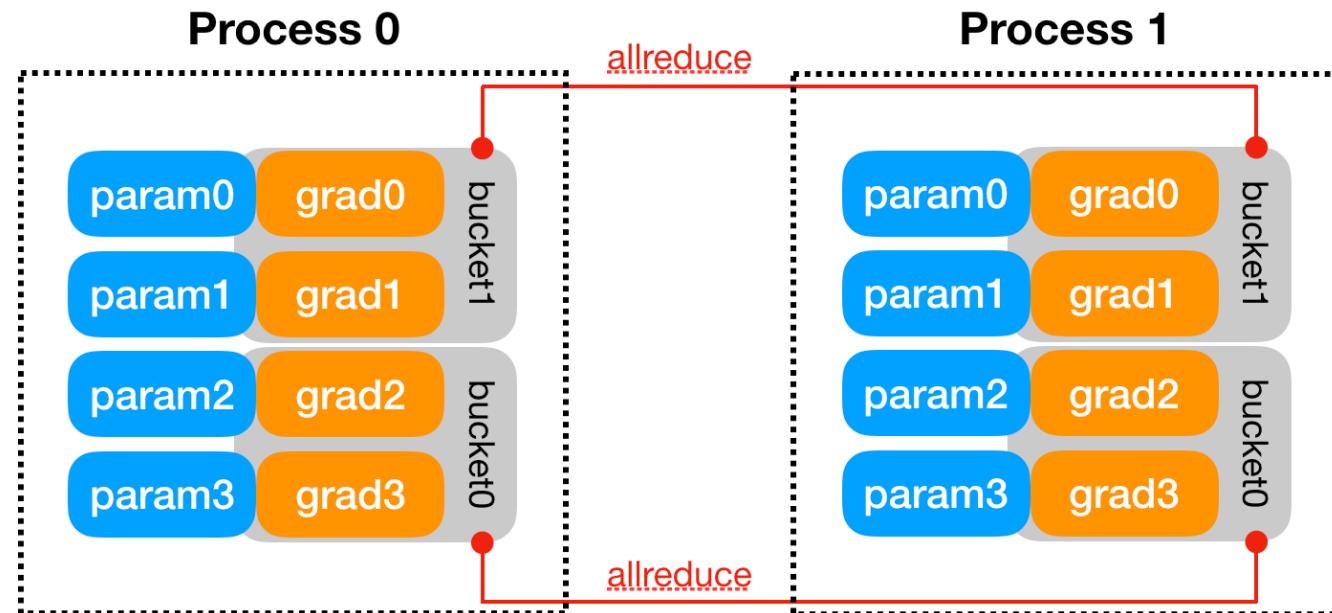
Distributed Data Parallel

- Se basa en la replicación del *Module* del modelo (como DataParallel)
- DDP lanza varios procesos y crea una instancia del modelo por cada proceso
 - Un proceso → un dispositivo
 - Un nodo → Varios procesos (varios dispositivos)
- Los gradientes y buffers se sincronizan a través de operaciones colectivas del paquete torch.distributed
- DDP registra un *hook* de autograd por cada parámetro del modelo
 - Cuando el gradiente asociado al parámetro se calcula en la pasada *backward*, se alcanza el *hook* lo cual desencadena la sincronización de los gradientes entre los procesos



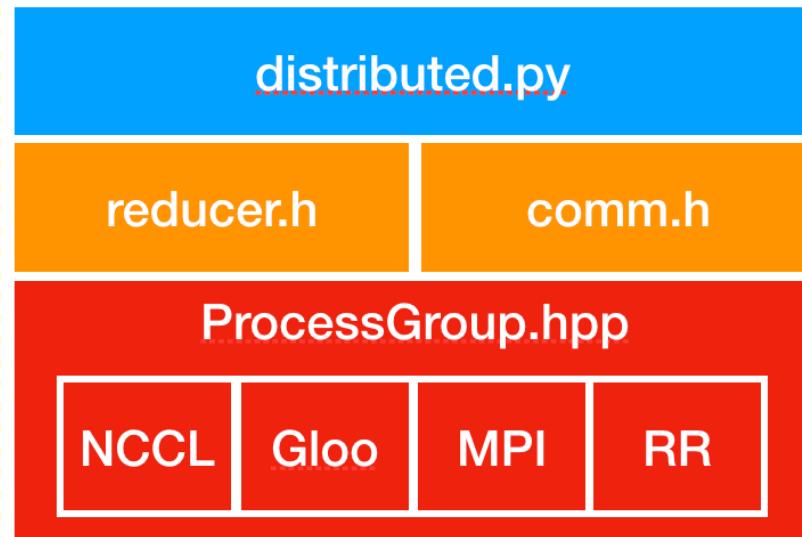
Distributed Data Parallel: Internals

- Detalles aquí: <https://pytorch.org/docs/master/notes/ddp.html>



Distributed Data Parallel: Internals

- Detalles aquí: <https://pytorch.org/docs/master/notes/ddp.html>



Distributed training en Pytorch: DistributedDataParallel

- Para lanzar varios procesos por nodo, normalmente uno por dispositivo debemos usar *torch.distributed.launch*

```
python -m torch.distributed.launch --nproc_per_node=NUM_GPUS_YOU_HAVE train.py
```

- o *torch.multiprocessing.spawn*

```
from torch.multiprocessing import spawn  
spawn(train, args=(world_size,), nprocs=world_size, join=True)
```

- También se podría usar el launcher torchx para ello

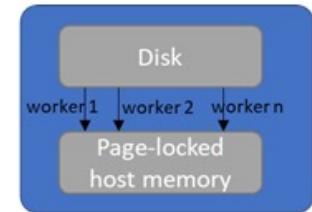
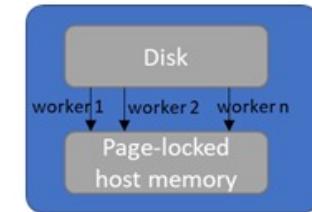
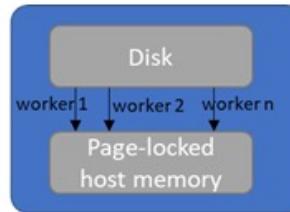


Distributed Data Parallel

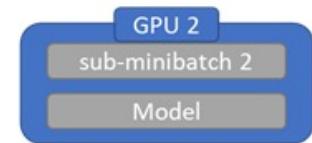
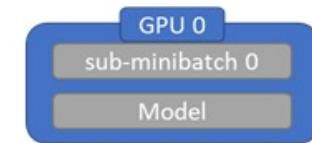
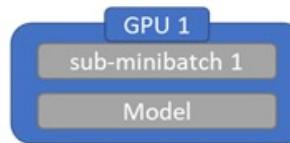
No master GPUs

Implemented in PyTorch
DistributedDataParallel
nodule

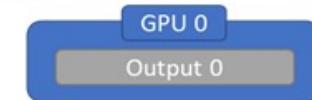
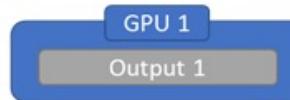
1. Load data from disk into page-locked memory on the host. Use multiple worker processes to parallelize data load.
Distributed minibatch sampler ensures that each process loads non-overlapping data



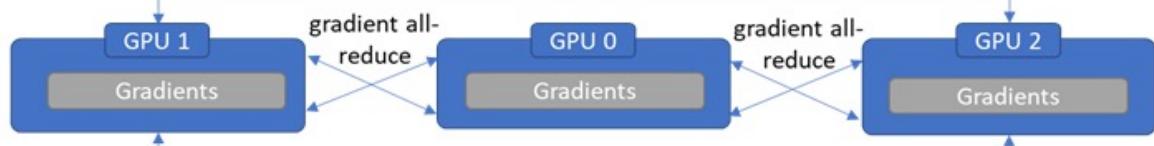
2. Transfer minibatch data from page-locked memory to each GPU concurrently. No data broadcast is needed. Each GPU has an identical copy of the model and no model broadcast is needed either



3. Run forward pass on each GPU, compute output



4. Compute loss, run backward pass to compute gradients. Perform gradient all-reduce in parallel with gradient computation



5. Update Model parameters. Because each GPU started with an identical copy of the model and gradients were all-reduced, weights updates on all GPUs are identical. Thus no model sync is required



Notebook pytorchDDP101

- cd \$STORE/Cesga2023Courses/scripts
- ./interactive_1node_2gpus.sh
- cd \$STORE/Cesga2023Courses/pytorch_dist/ooo
- Activa el entorno conda mytorchdist
source \$STORE/mytorchdist/bin/activate
- Lanza el servidor de jupyter
jupyter notebook –ip `hostname –i`
- Abre el notebook *pytorchDP101.ipynb*



Lanzamiento en SLURM

- Sigue las instrucciones del README:

https://github.com/diegoandradecanosa/Cesga2023Courses/tree/main/pytorch_dist/DDP/oo1



Fully Sharded Data Parallel (FSDP)

- Una de las limitaciones de DP y DDP es que los parámetros del modelo y cada batch de datos de entrenamiento debe caber en la memoria de cada dispositivo
- El mecanismo de Fully Sharded Data Parallel (FSDP) permite superar esta limitación distribuyendo también los parámetros del modelo entre los dispositivos
 - Menor consumo de memoria en cada dispositivo
 - Mayor necesidad de comunicaciones entre dispositivos (incluso entre nodos)
 - Se atenúa con el solapamiento entre comunicaciones y computación



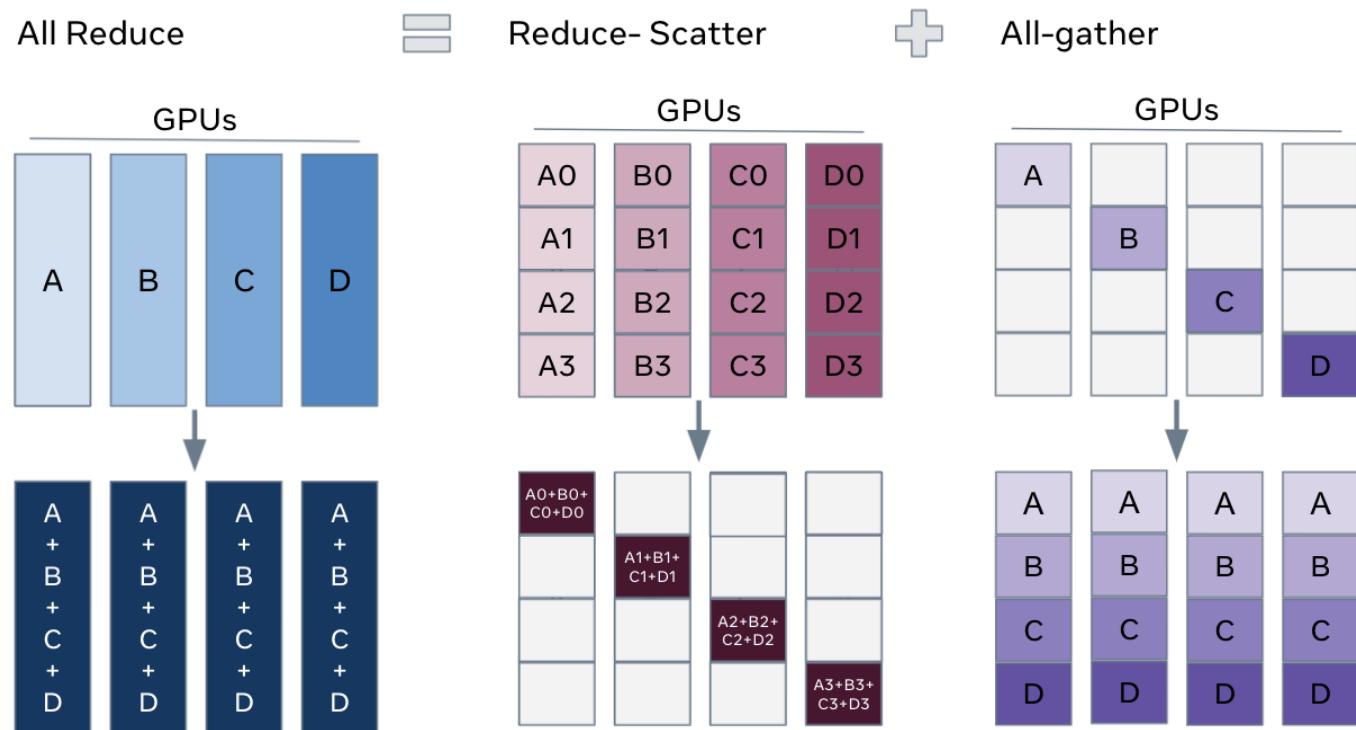
FSDP

La operación all-reduce se divide en:

- Reduce-scatter
- All-gather

Fuente:

<https://engineering.fb.com/2021/07/15/open-source/fsdp/>

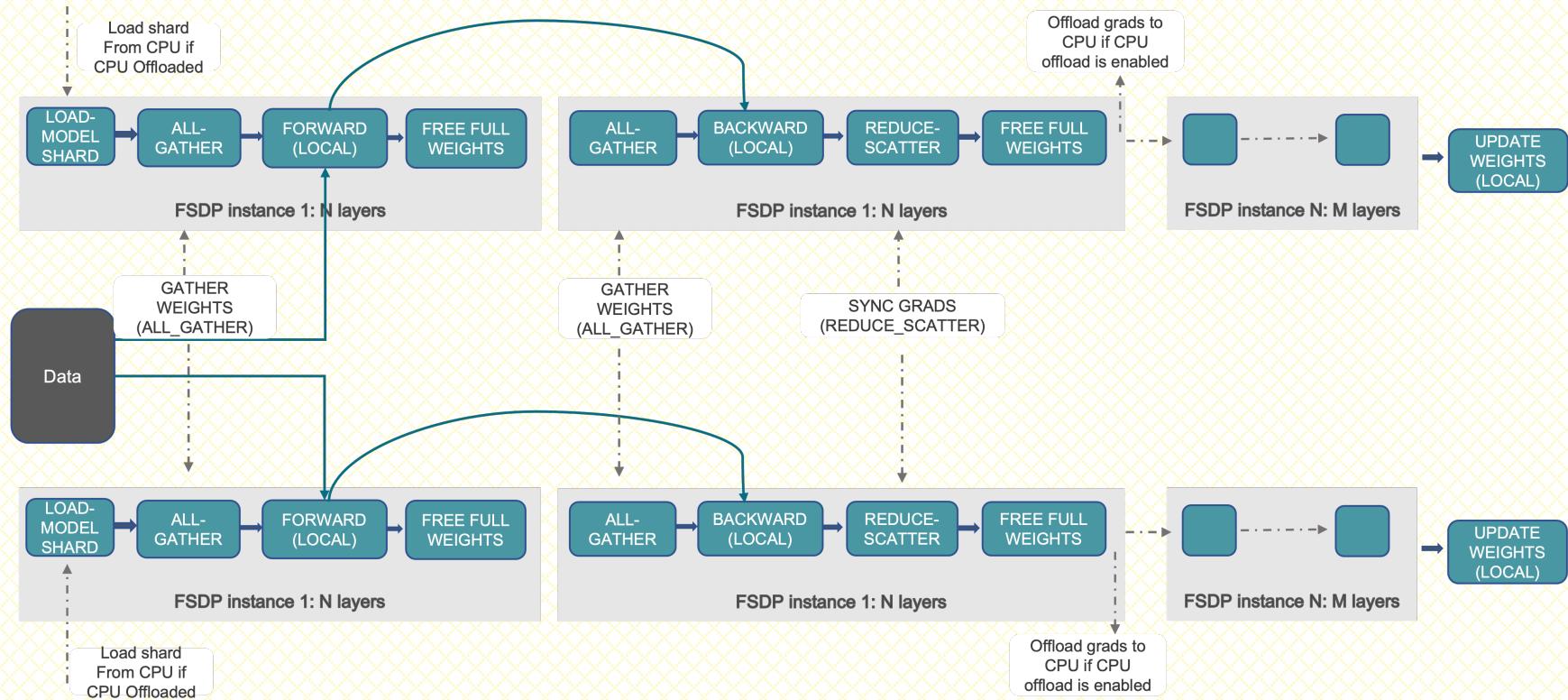


Fully Sharded Data Parallel (FSDP): Operación

- El constructor: Distribuye los parámetros del modelo entre los dispositivos
 - La estrategia usada es similar a la estrategia Zero de DeepSpeed
- En cada trabajador
 - Forward
 - Ejecuta all_gather para recoger todos los trozos (*shards*) de todos los trabajadores y reconstruir una copia de todos los parámetros
 - Ejecutar la pasada forward
 - Descartar los *shards* de parámetros que se han recopilado
 - Backward
 - Ejecutar all_gather para recoger todos los *shards* de todos...
 - Ejecutar la pasada backward
 - Ejecutar *reduce_scatter* para sincronizar los gradientes
 - Descartar los parámetros

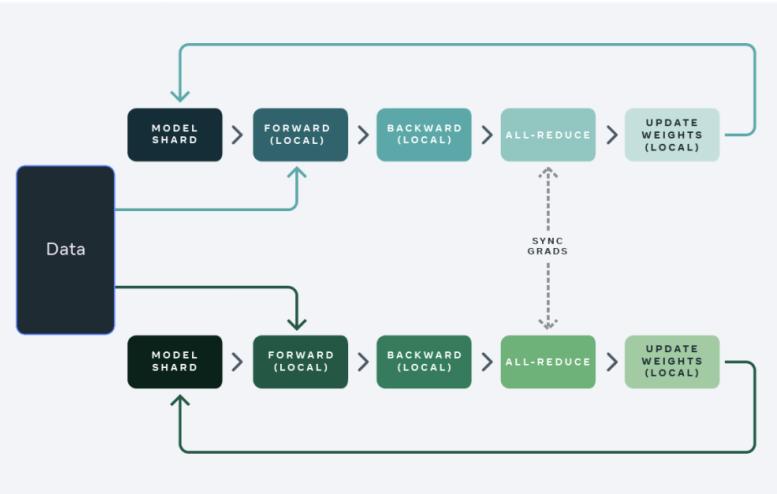


Fully Sharded Data Parallel (FSDP)

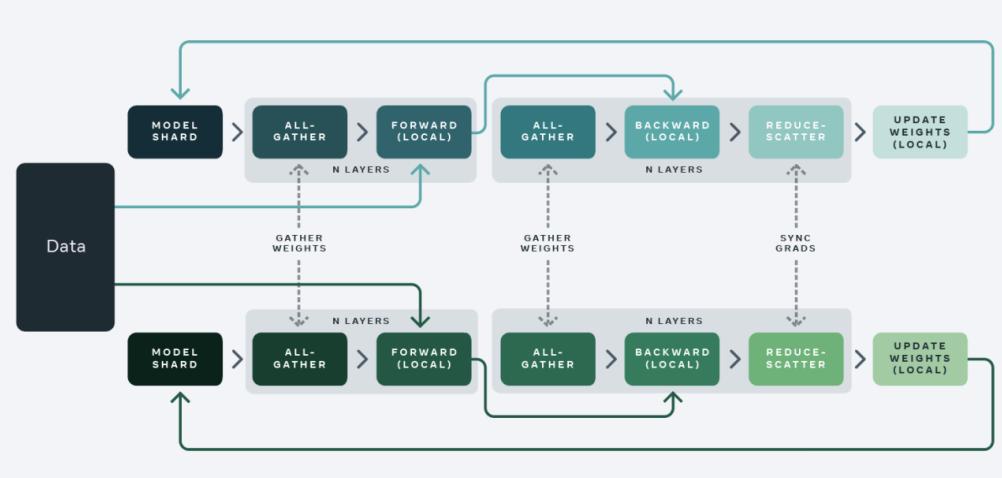


DDP vs FSDP

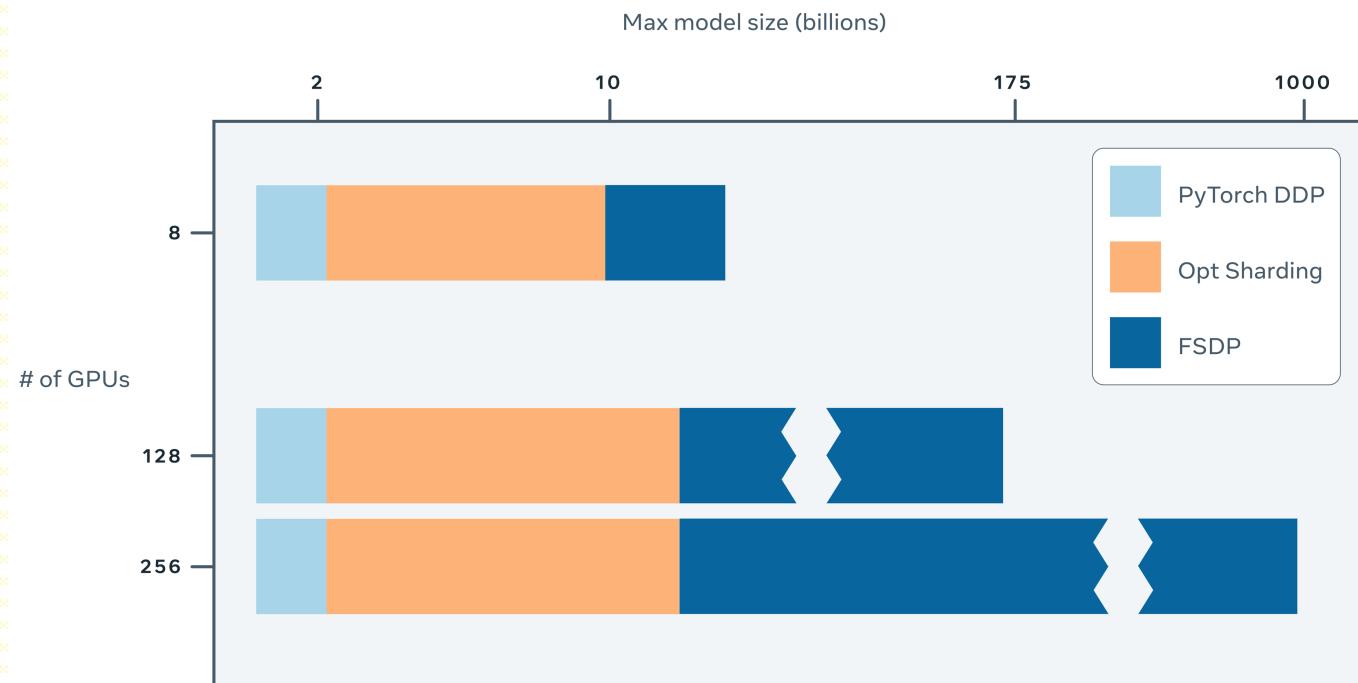
Standard data parallel training



Fully sharded data parallel training



FSDP: Beneficios



Ejemplo FSDP: SLURM

- Sigue las instrucciones del README:

https://github.com/diegoandradecanosa/Cesga2023Courses/tree/main/pytorch_dist/DDP/oo2



Referencias

- <https://pytorch.org/docs/stable/generated/torch.nn.DataParallel.html>
- https://pytorch.org/tutorials/beginner/blitz/data_parallel_tutorial.html
- <https://neptune.ai/blog/distributed-training>
- https://github.com/pytorch/tutorials/blob/main/intermediate_source/FSDP_tutorial.rst

