

TensorFlow distribuido

Diego Andrade Canosa

Roberto López Castro

Índice

- Introducción al curso
- Introducción a TensorFlow
- Repaso de conceptos de entrenamiento distribuido

Contenidos del curso

- Soporte nativo en TF para entrenamiento distribuido
 - MirroredStrategies
 - ParameterServer
 - DTENSORS
- Uso de Tensorflow con Ray
- Uso de Tensorflow con Horovod

Metodología

- Tres sesiones de cuatro horas
- Uso frecuente de los recursos de FT3

Entorno

Módulos en FT3

```
r-keras: r-keras/2.4.0-cuda-system  
  Interface to 'Keras' <https://keras.io>, a high-level neural networks 'API'. -- cesga/2020 r-keras/2.4.0-cuda-system : Core  
  
tensorflow: tensorflow/2.4.1-cuda-system, tensorflow/2.5.0-cuda-system, tensorflow/2.11.0  
  An open-source software library for Machine Intelligence -- cesga/2020 tensorflow/2.5.0-cuda-system: Compiler: Requires gcccore/system  
  
transformers: transformers/4.6.1  
  State-of-the-art Natural Language Processing for PyTorch and TensorFlow 2.0 -- cesga/2020 transformers/4.6.1 : Core
```

- Entornos virtuales venv
- Entornos conda

TensorFlow

- TensorFlow (TF) es una plataforma de aprendizaje automática
 - Junto a Pytorch, su principal competidor, una de las más populares
- Proporciona:
 - Herramientas avanzadas para el procesamiento y carga de datos
 - Definición de modelos utilizando bloques constructores con distinto nivel de complejidad
 - Implementación de servidores de inferencia de modelos (en producción)
 - Técnicas de regularización
 - Herramientas auxiliares como tensorboard o tf profiler

Evolución histórica

- Versión actual: 2.14
- Año 2011: el *Google Brain Team* (Andrew Ng and Jeff Dean) empiezan un proyecto llamado DistBelief
 - Sistema de ML escalable y distribuido
- Año 2015: Google libera el código de DistBelief y lo renombra como **TensorFlow**
 - *El código abierto como acelerador de la innovación*
- Año 2019: Tensorflow 2.0
 - API más simple
 - Mejor rendimiento
 - Mejor integración con Keras

Ecosistema de TF

- TensorFlow.js (entornos web)
- TensorFlow Lite (IoT)
- TFX (eXtended TF)
- Keras (Bloques constructores más sofisticados)
- TensorBoard (Visualización)
- Jupyter, Colab
- Otras herramientas:
 - Kubeflow (Contenedores para ML)
 - Frameworks distribuidos: Ray, Horovod, etc...

TF vs Pytorch

- Static vs Dynamic Computation Graph
 - Static: Menos flexible, mejor rendimiento
 - Dynamic: Más flexible y fácil de usar, peor rendimiento

```
import torch

# Define the neural network
class Net(torch.nn.Module):
    def forward(self, x, y):
        return x * y

# Create an instance of the neural network
net = Net()

# Define the input values
x = torch.tensor([2.0, 3.0])
y = torch.tensor([4.0, 5.0])

# Compute the output
output = net(x, y)
print(output) # Output: tensor([ 8., 15.] )
```

```
import tensorflow as tf

# Define the input values
x = tf.constant([2.0, 3.0])
y = tf.constant([4.0, 5.0])

# Define the computation
output = x * y

# Create a session and run the computation
with tf.Session() as sess:
    result = sess.run(output)
    print(result) # Output: [ 8. 15.]
```

Dynamic Computation Graphs en TF

- Los DCGs están disponibles en TF a través del modo *eager*
 - Habilitado por defecto
 - Recomendable deshabilitarlo para modelos en producción (inferencia)

Tensorflow (TF): Conceptos básicos

- Características básicas:
 - Soporte para tensores (arrays multidimensionales)
 - Procesamiento en GPU y distribuido
 - Diferenciación automática
 - Definición de modelos, entrenamiento y exportación

TF: Tensores

```
import tensorflow as tf

x = tf.constant([[1., 2., 3.],
                 [4., 5., 6.]])

print(x)
print(x.shape)
print(x.dtype)
```

Tensores: operadores

- $x+x$
- $5*x$
- transpose
- concat
- reduce_sum
- softmax
- ...
- **Variables:** Son la versión mutable de los tensores (usados para almacenar, por ejemplo, los parámetros entrenables del modelo)

Diferenciación automática

- El mecanismo de **autodiff** es similar al disponible en Pytorch
 - Construye un grafo con los nodos de la computación (durante la pasada *forward*) para calcular los gradientes de los pesos aplicables durante la pasada *backward*
- Se activa poniendo el código dentro del entorno

with tf.GradientTape() as tape:

(...)

@tf.function

- Se trata de un decorador que aplicado a una función habilita varias características
 - Optimización del rendimiento
 - Acelera inferencia y entrenamiento
 - Exportación del modelo al final del entrenamiento
 - La primera vez que se ejecuta se genera un grafo de la computación que se utiliza para acelerar ejecuciones posteriores

Modules, layers, tensor, variables & models

- Como en Pytorch, existen varias abstracciones que actúan como contenedores o bloques constructores de modelos de ML
 - *Module*: Similar al concepto homónimo de Pytorch
 - Los modules sirven de contenedores para los modelos
 - *Layers* predefinidas: Evitan tener que definir tipos de capas comunes desde cero mediante tensores

Keras

- Es un API de nivel superior de TF
- Proporciona bloques constructores de alto nivel para aplicaciones de ML
 - Interfaz sencilla y consistente
 - Minimizar el código necesario para casos de uso comunes
 - Mejora la legibilidad del código

Keras: Layers y Models

- **Layers:** Encapsulan una capa de un modelo de ML: un estado (pesos) y alguna computación (call)
 - Los pesos pueden ser entrenables o no
 - Las capas se componen recursivamente
 - También se pueden usar para tareas de preprocesado
- **Models:** Son agrupaciones de capas
 - El modelo *Sequential* es el más común, se usa para agrupar una secuencia de capas
 - Arquitecturas más comunes se componen usando la *Keras functional API*
 - Proporciona:
 - Método fit: para entrenar el modelo
 - Método predict: para generar predicciones (inferencia) en base a *samples* de entrada
 - Método evaluate: para devolver la función de pérdida y otras métricas generadas en el momento de la compilación (*compile*) del modelo

Otros componentes de Keras

- Optimizers
- Metrics
- Losses
- Utilidades de carga de datos

Bucles de entrenamiento: Contexto

- Estructura de un script de entrenamiento
 1. Importar y procesar un conjunto de datos (*dataset*). Separar en:
 1. Entrenamiento
 2. Validación
 2. Definir la arquitectura del modelo e instanciarlo
 3. Bucle de entrenamiento
 1. Inferencia
 2. Cálculo de la función de pérdida
 3. Cálculo de los gradientes
 4. Aplicación de los gradientes a los parámetros del modelo
 4. Validación de la precisión del modelo

Bucle de entrenamiento en TF

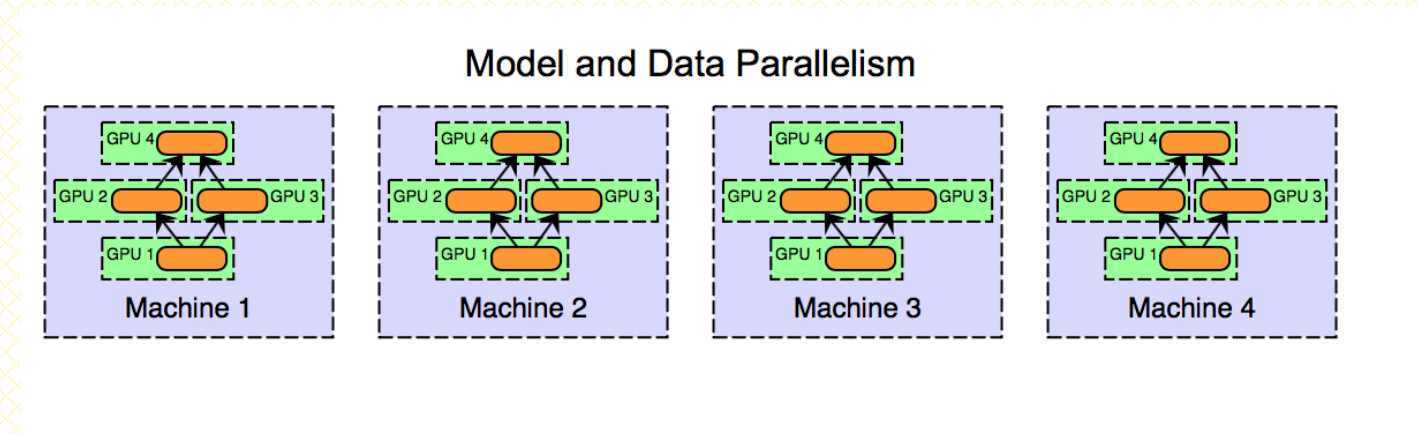
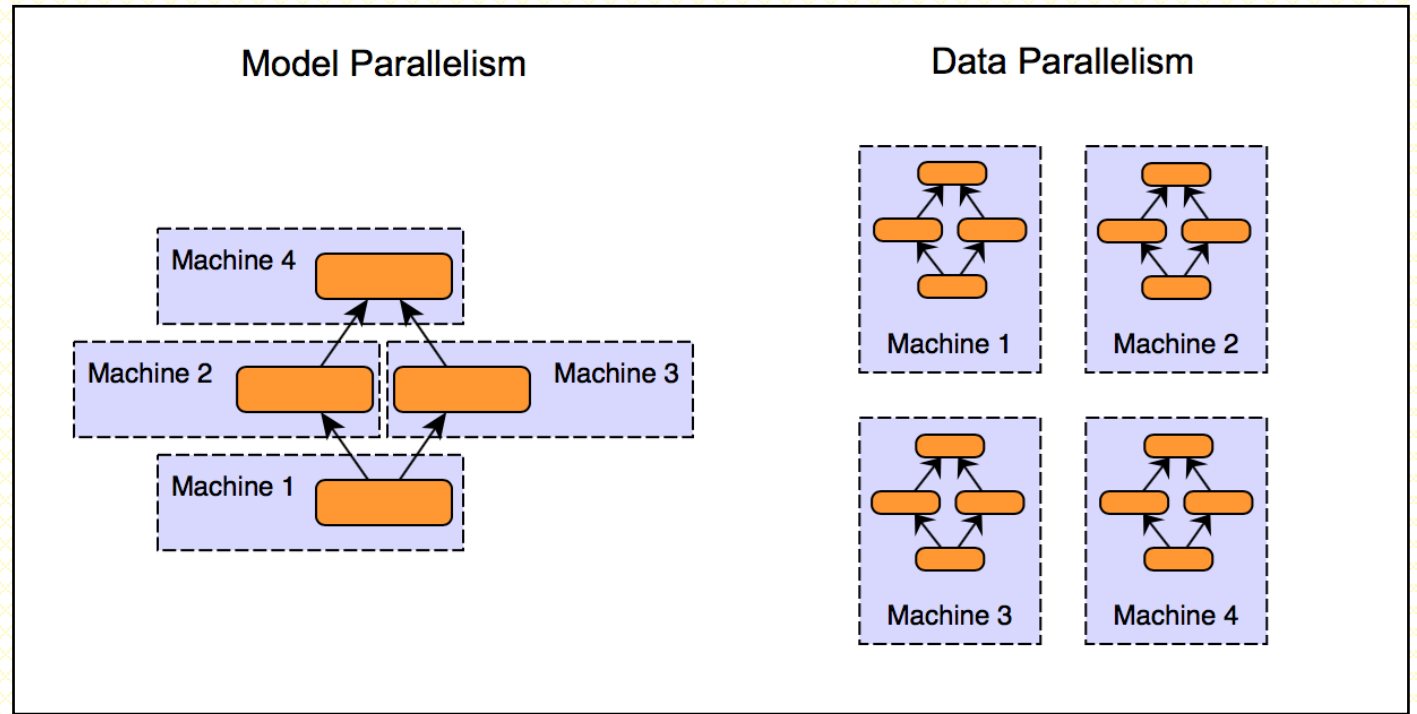
```
model=keras.Sequential([layer1, layer2, layer2 ...])
model.compile(optimizer="optname", loss="lossfuncname", metrics=['metric1', metric2, ...])

for epoch in range(num_epochs):
    for i in range(0,len(train_data, batch_size):
        with tf.GradientTape as tape:
            predictions= model(batch_data)
            loss=tf.keras.losses.somelossfunction(batch_labels,predictions
            gradients = tape.gradient(loss, model.trainable_variables)
            optimizer.apply_gradients(zip(gradients, model.trainable_variables))

model.save('filename')
```

Entrenamiento distribuido

- Paralelismo de datos
- Paralelismo de modelo
- Paralelismo híbrido



Centralizado vs descentralizado

- Copias espejo (*Mirror*)
 - allreduce
- Parameter Server
 - 1 PS – n trabajadores
 - n PS – n trabajadores

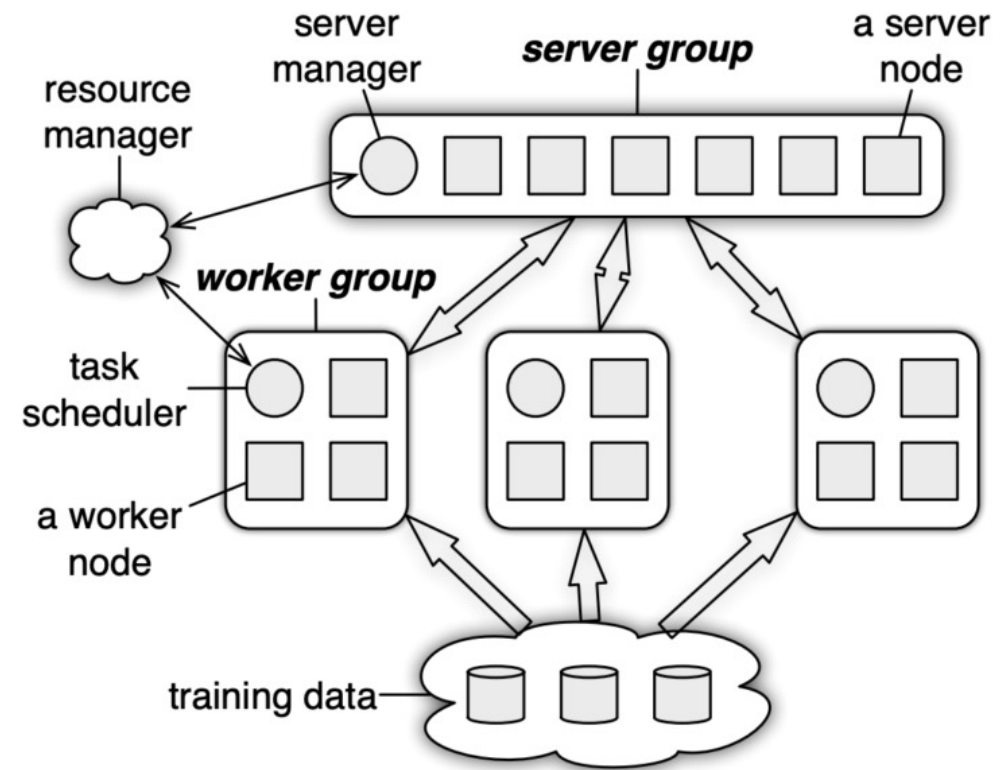
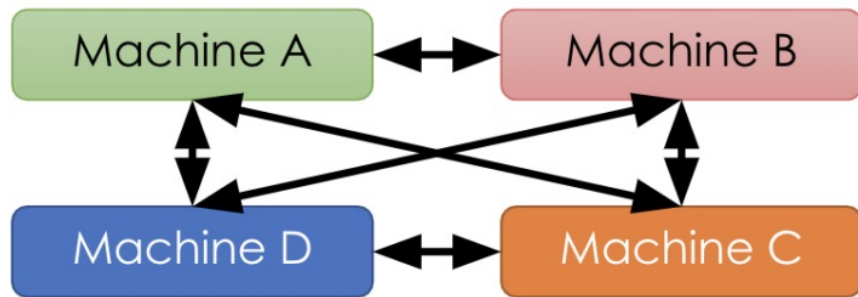
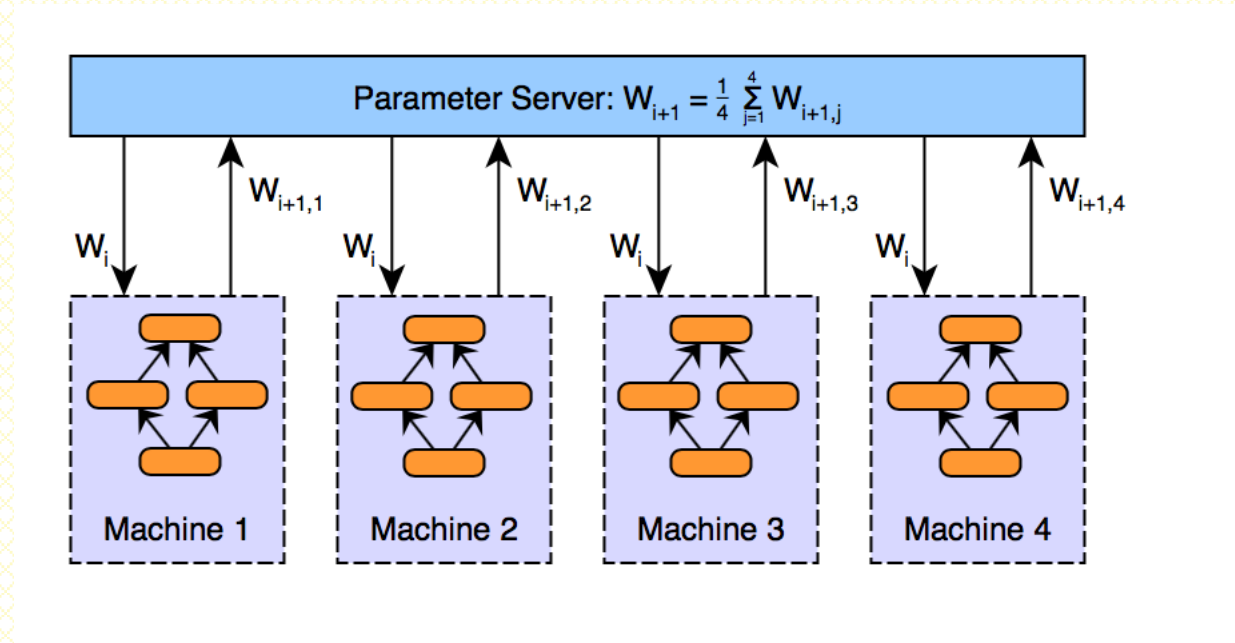


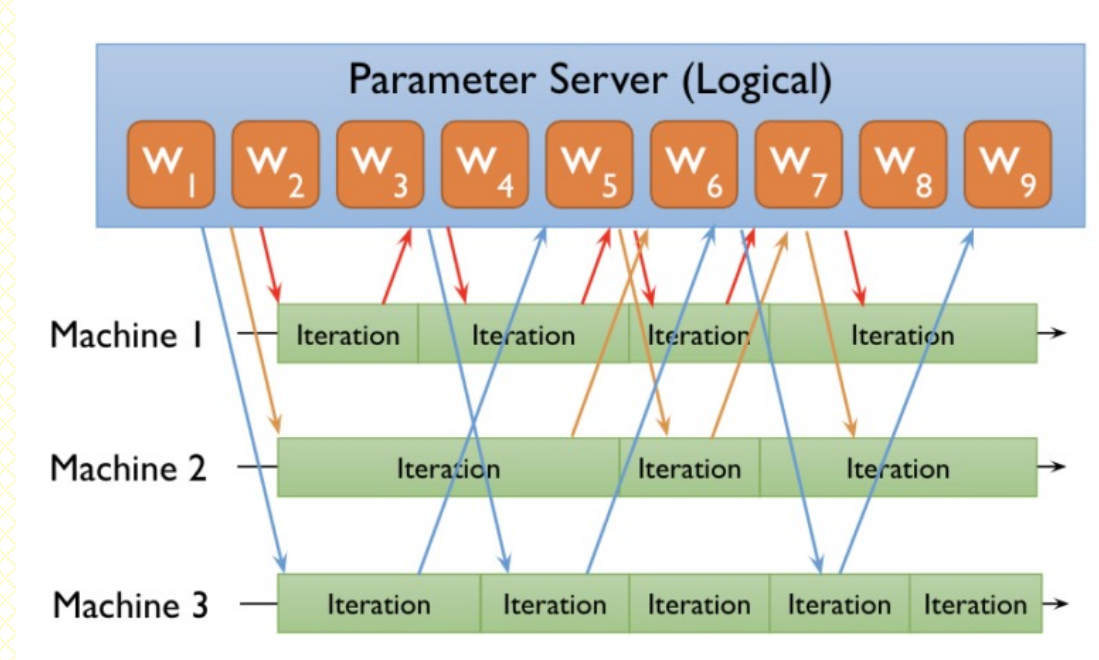
Figure 4: Architecture of a parameter server communicating with several groups of workers.

Promediado de pesos



Promediado síncrono vs asíncrono

- Síncrono: Hay que esperar por todos los trabajadores para promediar los pesos
 - La sincronización supone un cuello de botella
- Asíncrono: El promediado se produce sin asegurar la sincronización de los trabajadores
 - Convergencia más lenta del modelo



Estrategias avanzadas

- Paralelismo de tensores
- Paralelismo multinivel
 - Data + Model + Tensor
- Estrategias ad-hoc para ciertas arquitecturas de modelo
- Estrategias avanzadas como Zero-DeepSpeed

Actividad: Conf. y prueba del entorno

- Creación y configuración del entorno
 - Conectarse a FT3
 - `compute --gpu`
 - `cd Cesga2023Courses/tf_dist/scripts`
 - `source createVENVTF.sh`
 - `source $STORE/mytf/bin/activate`
- Comprobar la instalación
 - `python`
 - `>> import tensorflow as tf`
 - `>> print("Num GPUs Available: ", len(tf.config.list_physical_devices('GPU')))`

Actividad: Conf. y prueba del entorno

- Clona el repositorio en \$STORE

`cd $STORE`

`git clone https://github.com/diegoandradecanosa/Cesga2023Courses.git`

- Si ya lo tenías basta con hacer un pull

`cd $STORE/Cesga2023Courses`

`git pull`

Soporte nativo para TF distribuido

- Evaluación del rendimiento (tf.profile)
- Coexistencia con SLURM y envío de trabajos
- Carga de datos en entornos distribuido
- Entrenamiento en un nodo (CPU)
- Estrategias de entrenamiento distribuido
 - Mirrored y MultiworkerMirrored
- Estrategias de tipo Parameter-Server
- Uso de DTENSORS

Herramientas de profiling del uso de recursos: TensorBoard

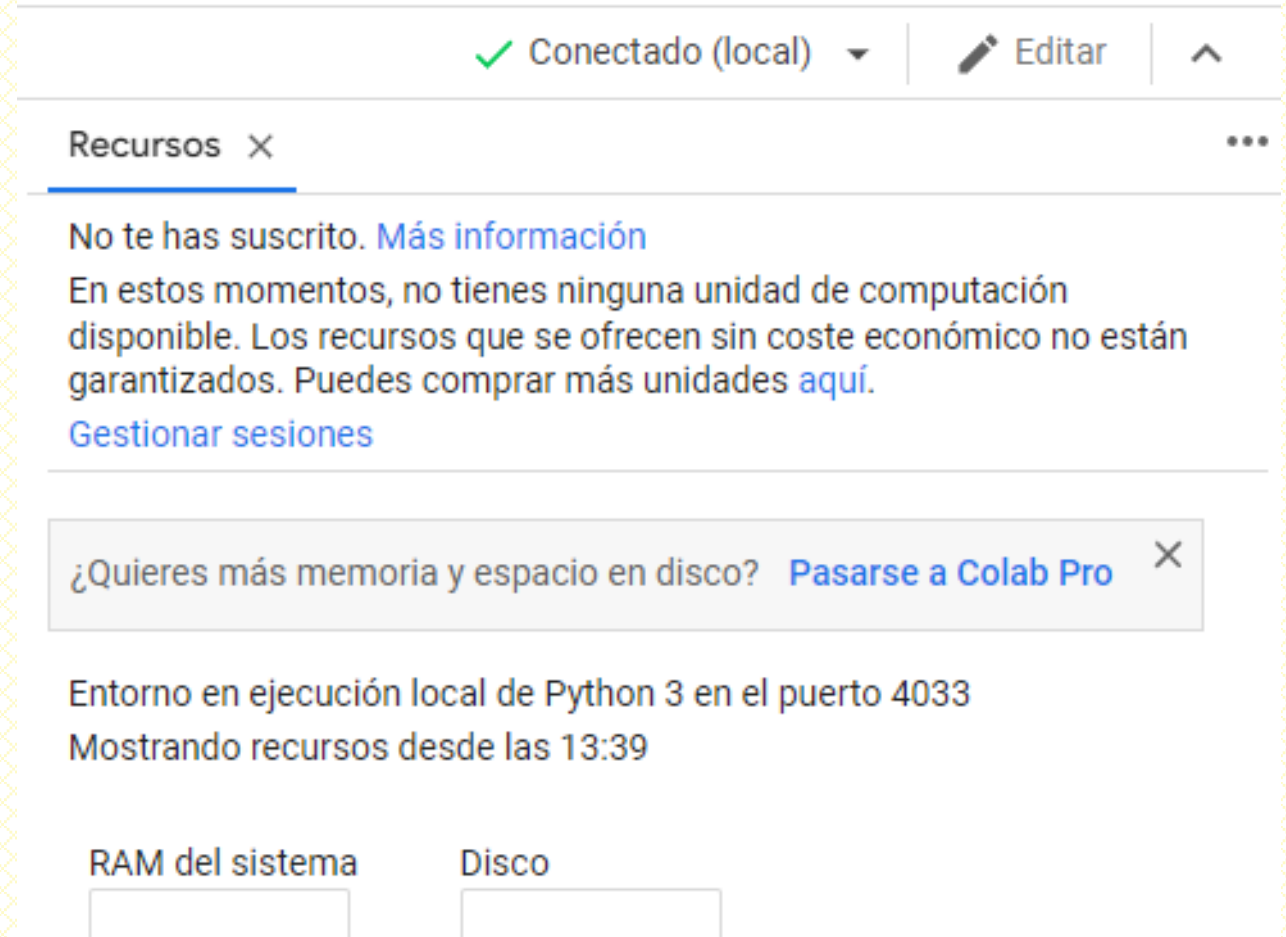
- TensorBoard es un conjunto de herramientas de visualización para ML
 - Soporte para TF y Pytorch
 - Visualización de la evolución de métricas como: loss y accuracy
 - Visualización del grafo del modelo
 - Visualización de histogramas de pesos, bias y otros tensores mientras evolucionan en el tiempo
 - ...
 - **Profiling del rendimiento del proceso de entrenamiento**

Fuente: [Get started with TensorBoard](#) | [TensorFlow](#)



Herramientas de profiling del uso de recursos: TensorBoard

- Probar la conexión “local” de un notebook alojado en Google Colab
- Elegir Conectarse a un entorno de ejecución local
- Seguir las instrucciones para conectarse al Jupyter en ejecución en el FT3

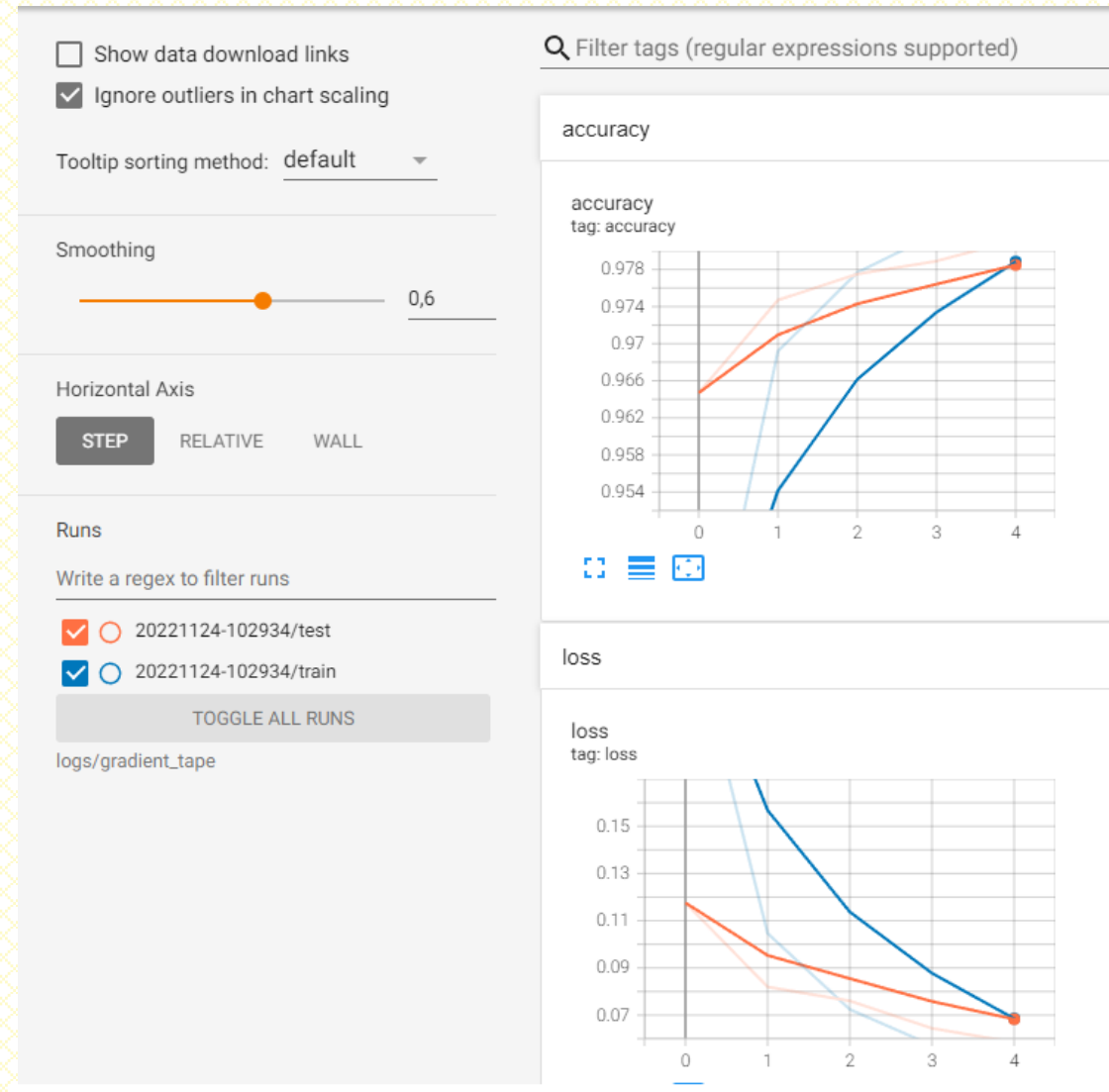


The screenshot shows the TensorBoard web interface. At the top, there's a status bar with a green checkmark and the text "Conectado (local)", followed by an "Editar" button and an upward arrow. Below this is a tab labeled "Recursos" with a close button (X) and a menu icon (three dots). The main content area displays a message: "No te has suscrito. Más información". Below this, it states: "En estos momentos, no tienes ninguna unidad de computación disponible. Los recursos que se ofrecen sin coste económico no están garantizados. Puedes comprar más unidades aquí." and a link "Gestionar sesiones". A grey banner with a close button (X) asks: "¿Quieres más memoria y espacio en disco? Pasarse a Colab Pro". Below the banner, it says: "Entorno en ejecución local de Python 3 en el puerto 4033" and "Mostrando recursos desde las 13:39". At the bottom, there are two labels: "RAM del sistema" and "Disco", each followed by a small rectangular box representing a progress bar.

Herramientas de profiling del uso de recursos: TensorBoard

- TensorBoard es un conjunto de herramientas de visualización para ML
 - Visualización de la evolución de métricas como: loss y accuracy

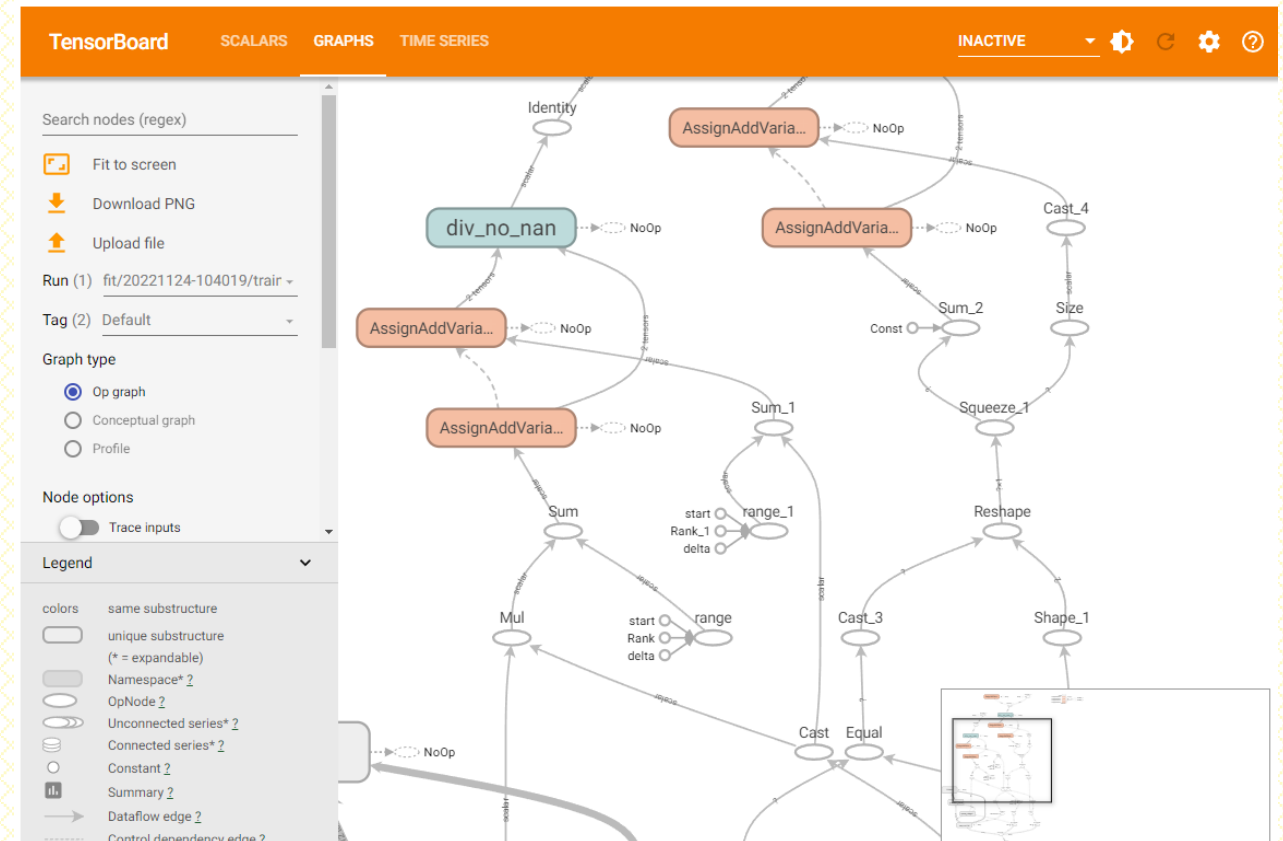
Fuente: [TensorBoard Scalars: Logging training metrics in Keras | TensorFlow](#)



Herramientas de profiling del uso de recursos: TensorBoard

- TensorBoard es un conjunto de herramientas de visualización para ML
 - Visualización del grafo del modelo

Fuente: [Examining the TensorFlow Graph | TensorBoard](#)

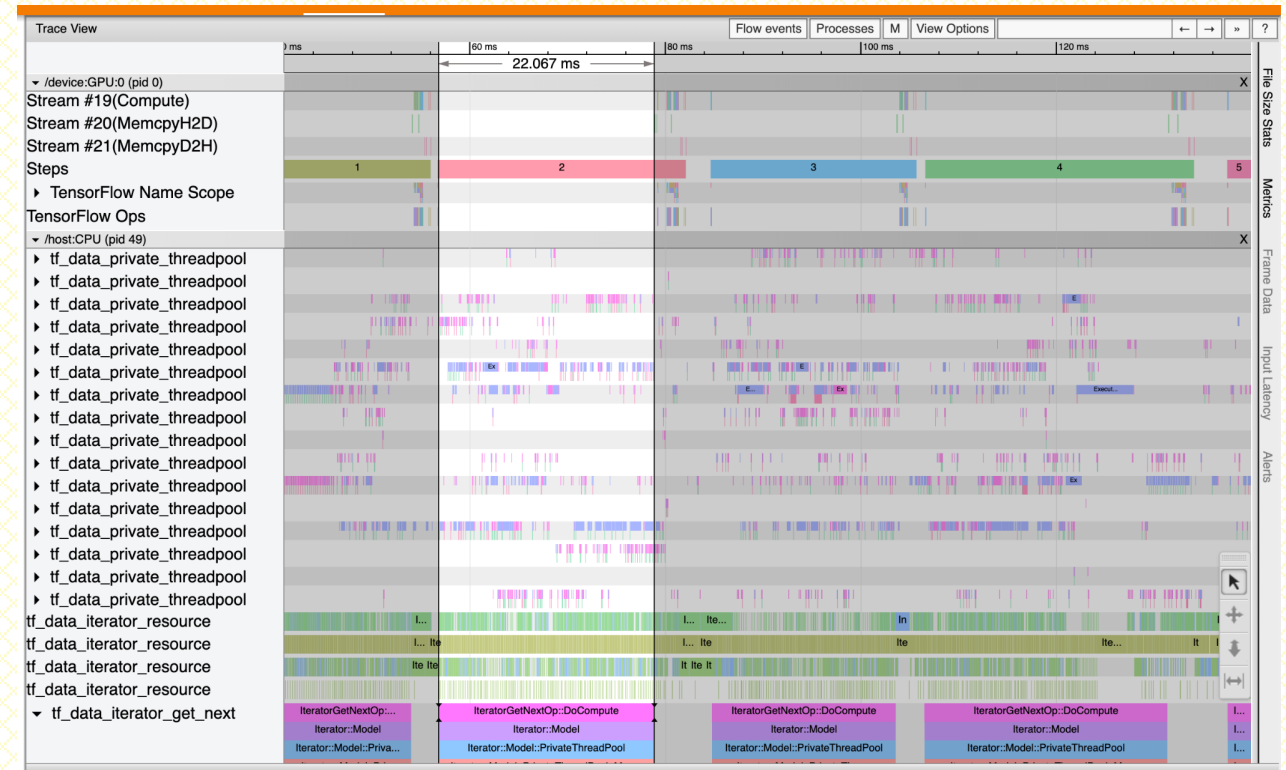


Herramientas de profiling del uso de recursos: TensorBoard

- TensorBoard es un conjunto de herramientas de visualización para ML
 - **Profiling del rendimiento del proceso de entrenamiento**

Fuente:

[tensorboard profiling keras.ipynb](https://colab.research.google.com/github/tensorflow/tensorboard/blob/master/tensorboard/plugins/profile/README.md) - Colaboratory (google.com)



Actividad: Tensorboard

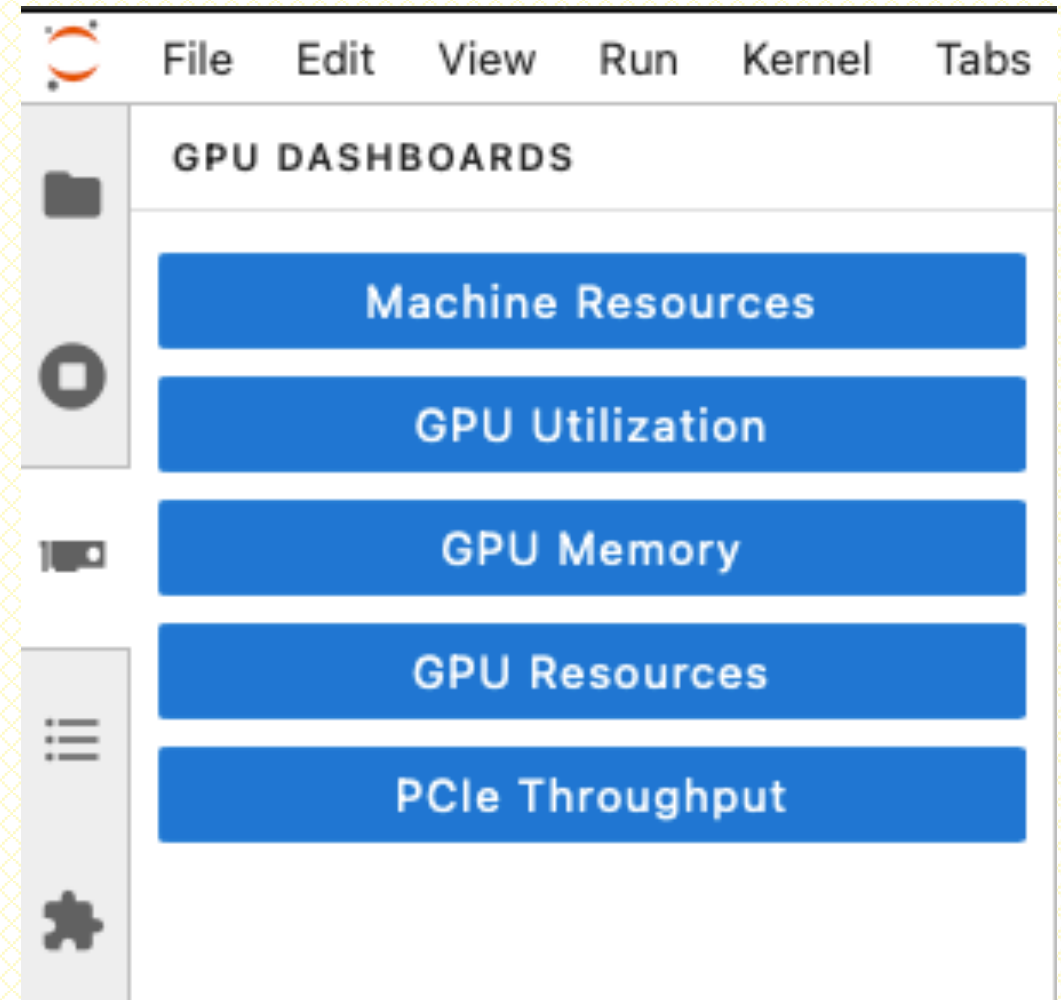
- https://github.com/diegoandradecanosa/Cesga2023Courses/tree/main/tf_dist/TF/001#actividad-tensorboard

Información complementaria

- https://www.tensorflow.org/guide/gpu_performance_analysis
- <https://www.tensorflow.org/guide/profiler>
- https://www.tensorflow.org/guide/mixed_precision
- https://www.tensorflow.org/guide/graph_optimization

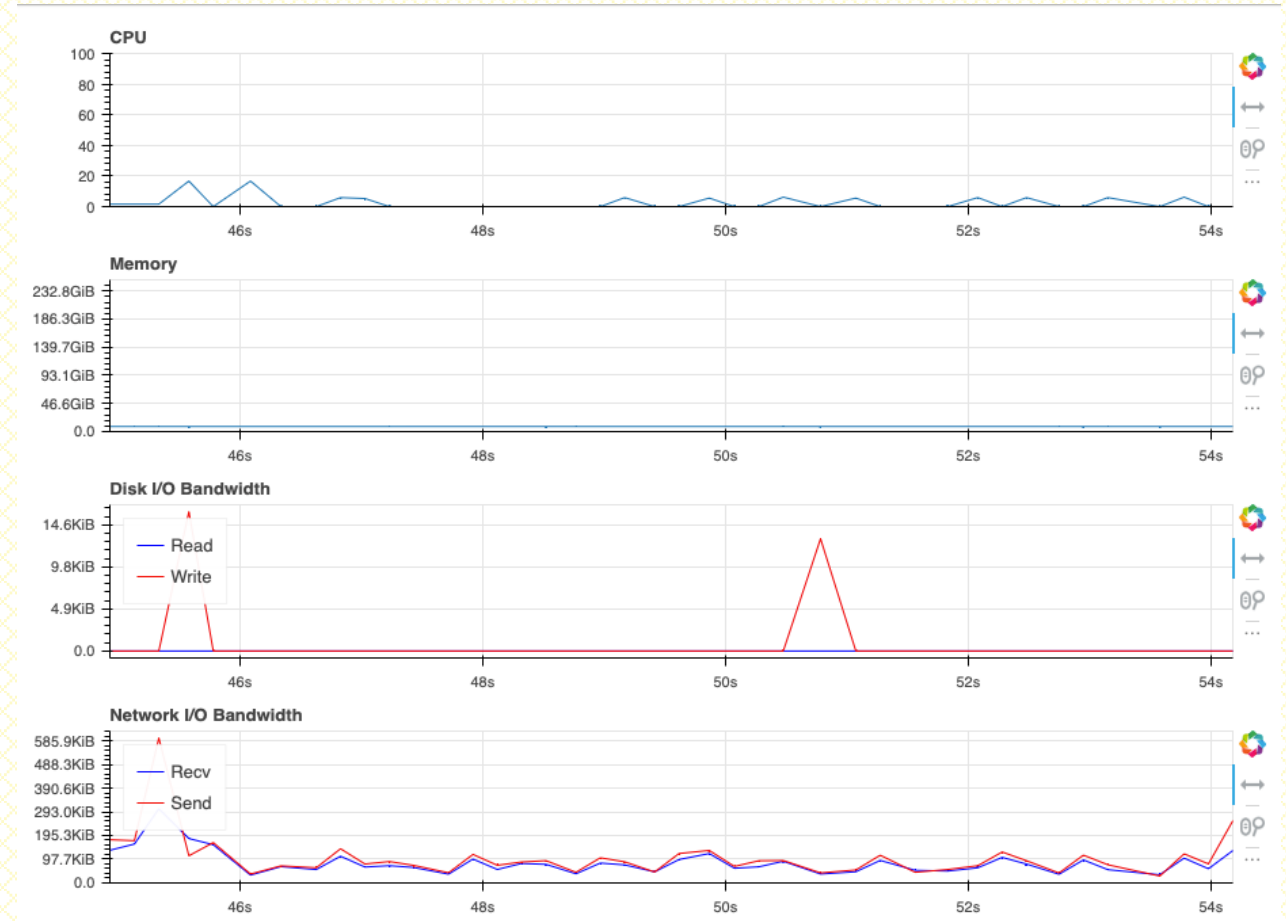
Herramientas de profiling del uso de recursos

- NVBoard: Es una extensión de jupyterlab que nos permite observar en tiempo real la ocupación de los recursos de la máquina durante la ejecución de un código
 - CPU
 - Memoria
 - I/O
 - Memoria
 - Red
 - GPU
 - Utilización
 - Memoria
 - PCIe throughput



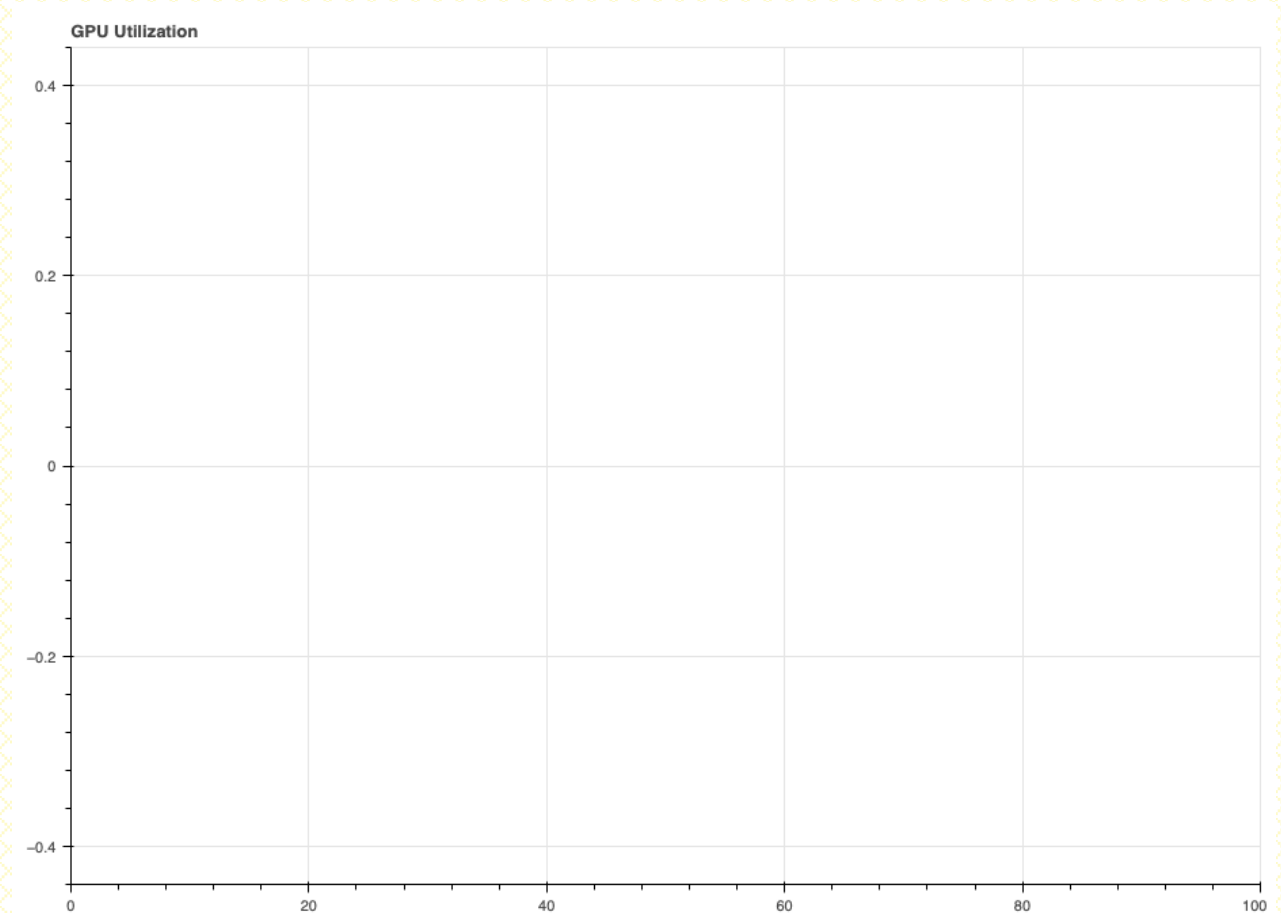
Herramientas de profiling del uso de recursos: nvboard

- Vista “machine resources”



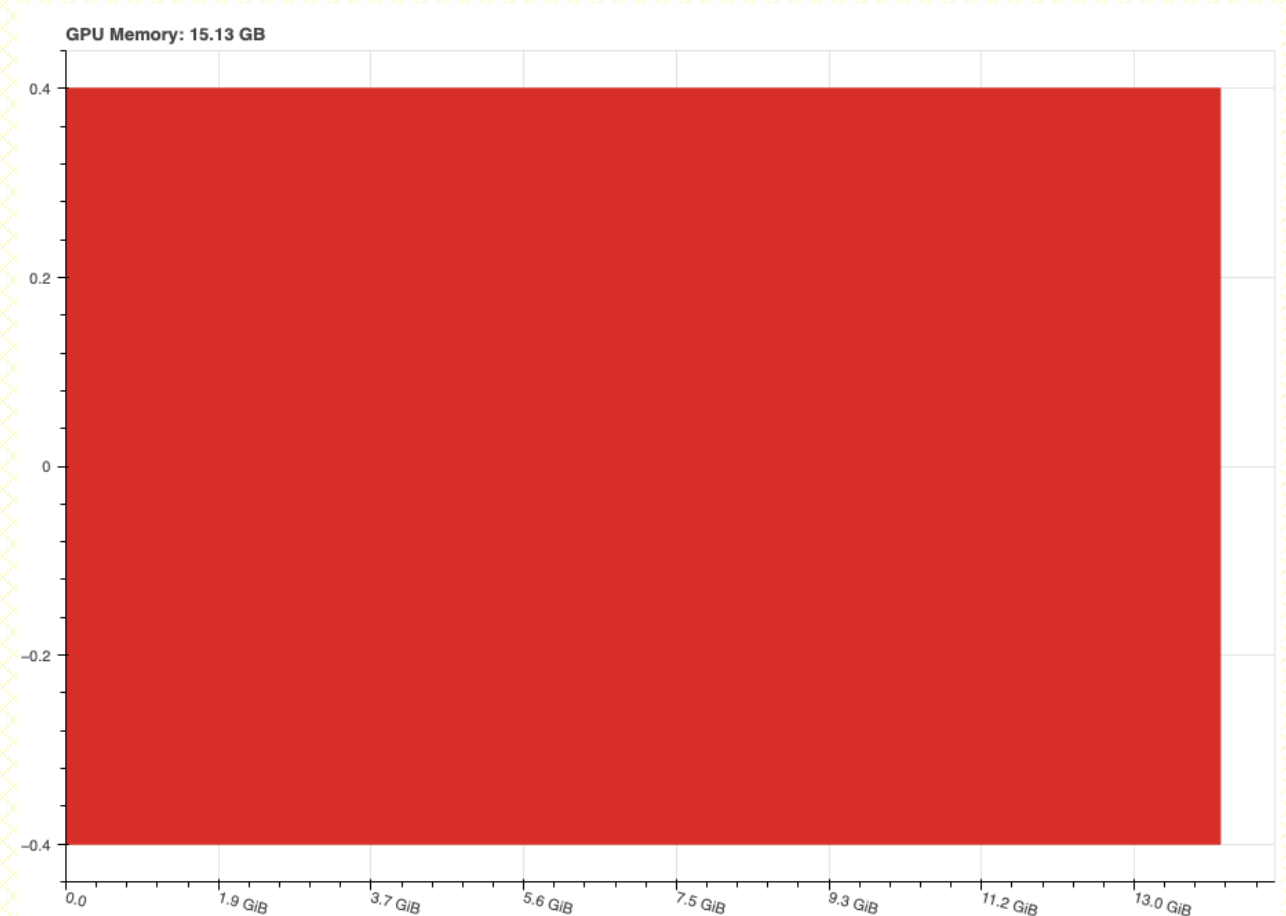
Herramientas de profiling del uso de recursos: nvboard

- Vista “GPU utilization”



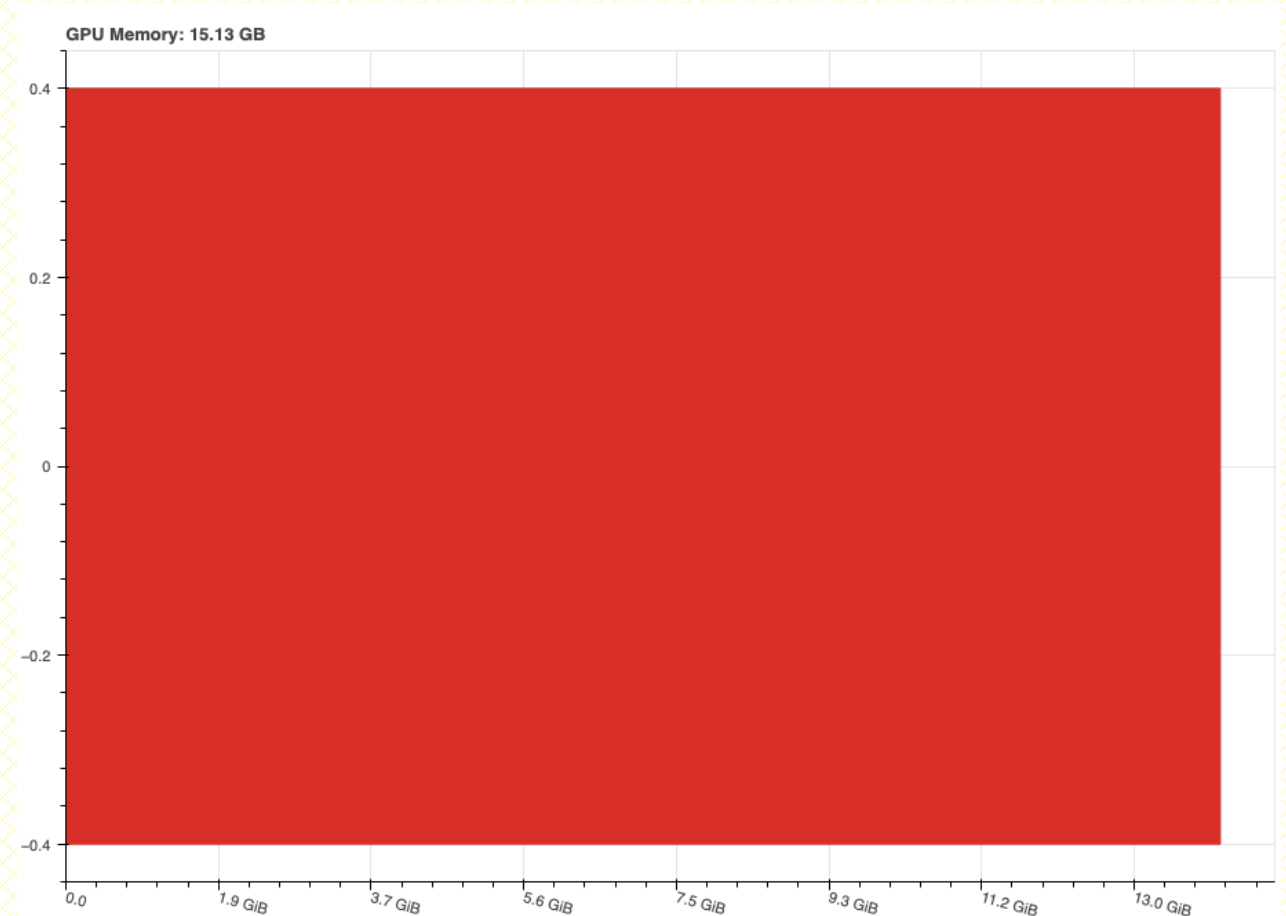
Herramientas de profiling del uso de recursos: nvboard

- Vista “GPU memory”



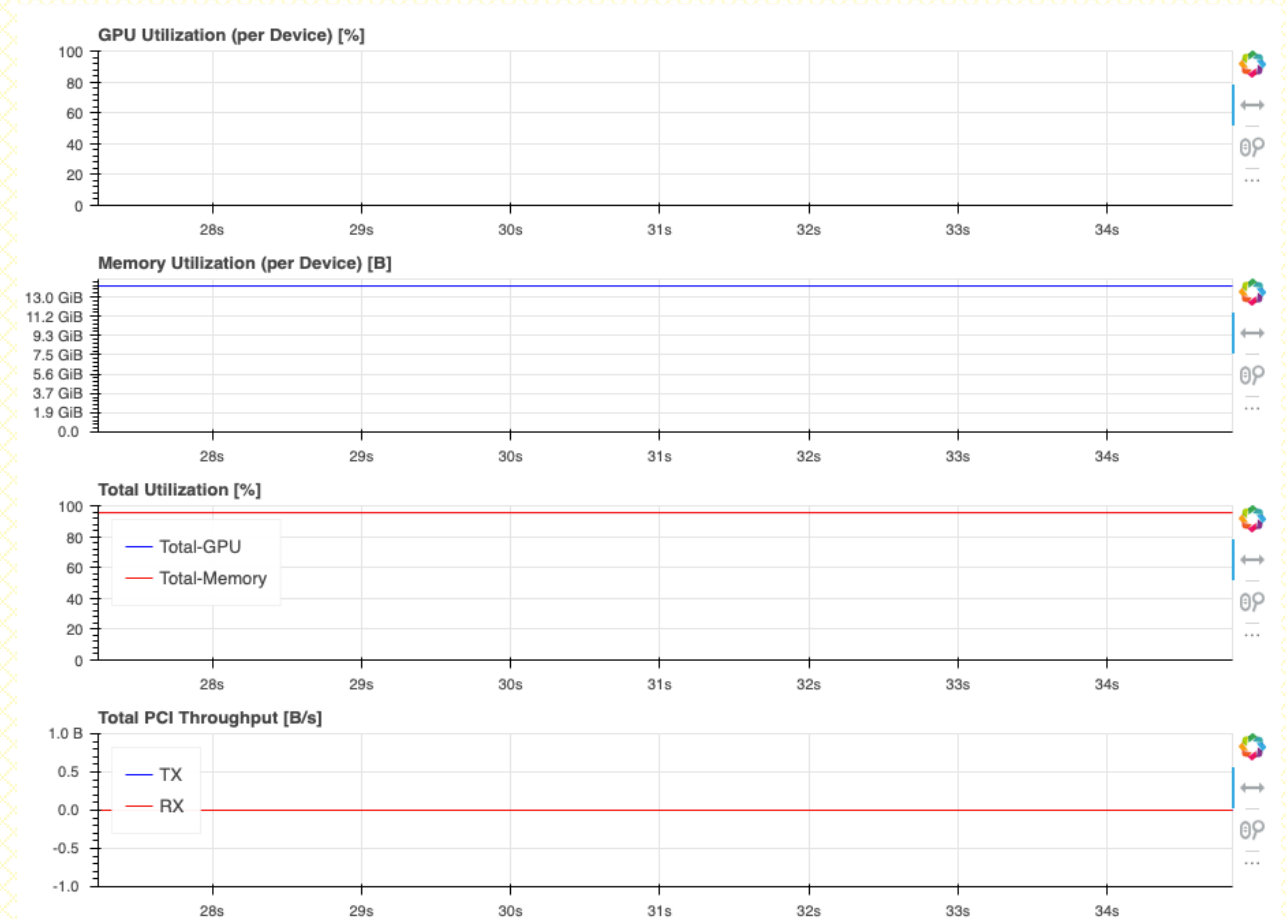
Herramientas de profiling del uso de recursos: nvboard

- Vista “GPU memory”



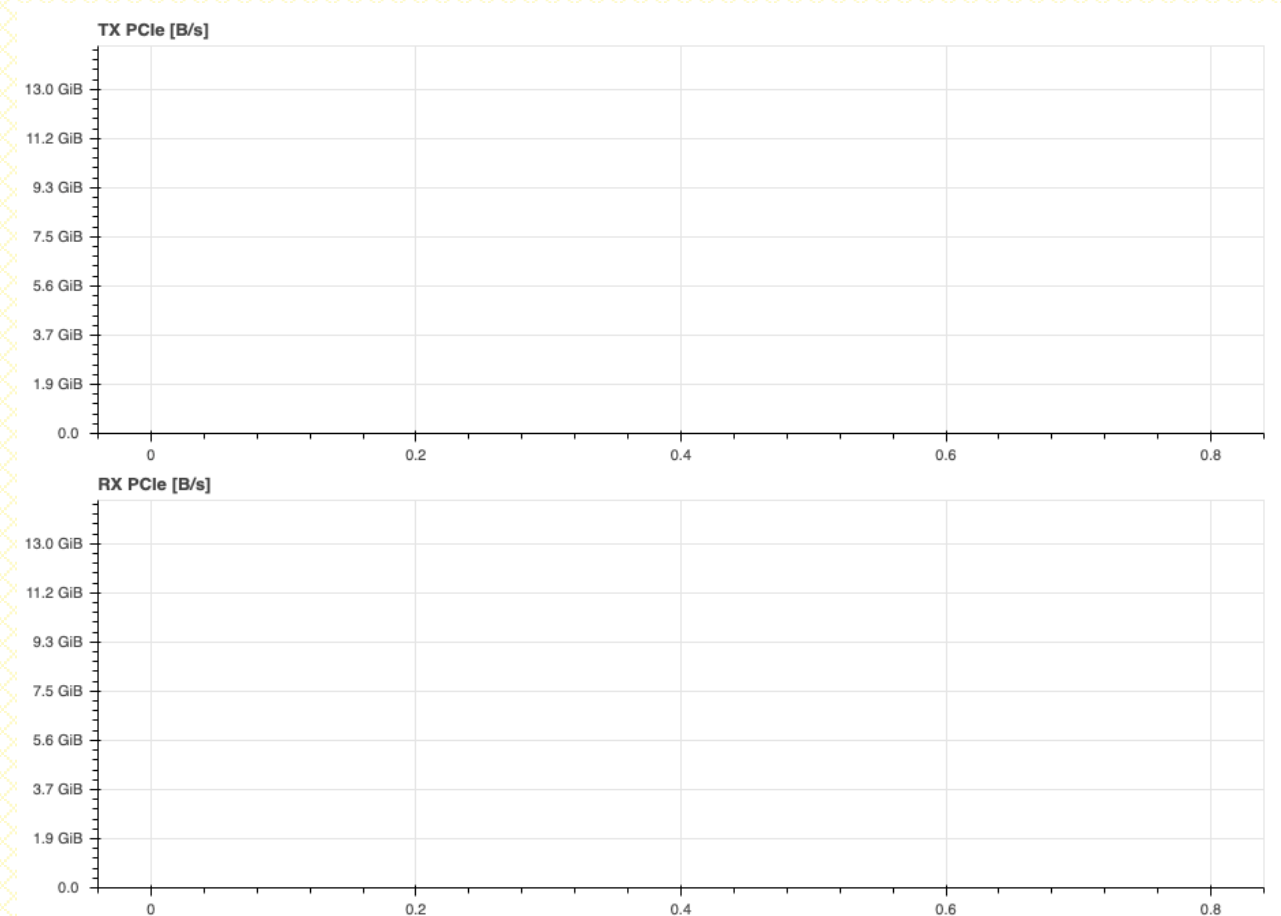
Herramientas de profiling del uso de recursos: nvboard

- Vista “GPU resources”



Herramientas de profiling del uso de recursos: nvboard

- Vista “PCIe resources”



Actividad: NVDashBoard

- https://github.com/diegoandradecanosa/Cesga2023Courses/tree/main/tf_dist/TF/001#actividad-nvboard

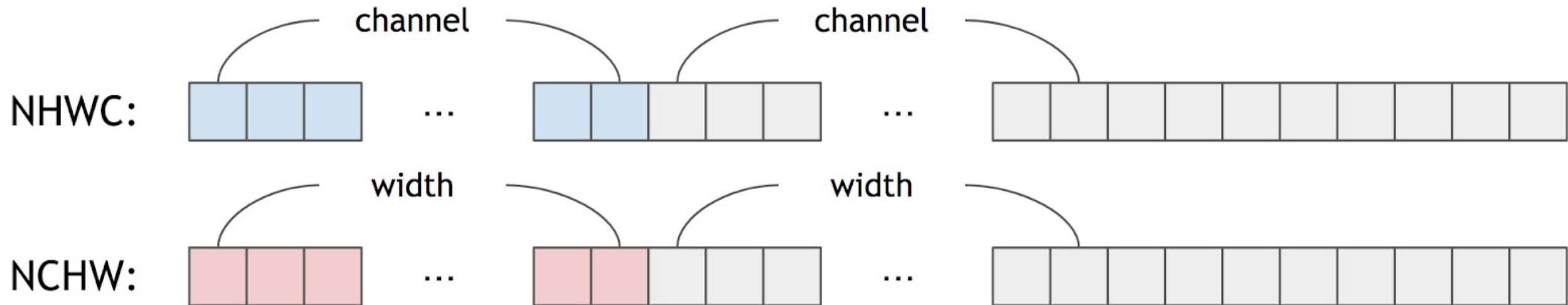
Optimización para TensorFlow

- A partir de la versión 2.5 de TensorFlow, el uso de las implementaciones Intel MKL (o oneDNN) de los kernels está habilitado por defecto
- Mismos controles que Pytorch para:
 - NUMA
 - Variables de entorno de OpenMP
- Intel también proporciona versiones optimizadas de otros frameworks de IA [AI Frameworks \(intel.com\)](https://www.intel.com/ai-frameworks):
 - Mxnet
 - PaddlePaddle
 - ScikitLearn
 - XGBoost

Optimización para TensorFlow

Opciones de tiempo de ejecución que afectan al rendimiento:

- `--intra_op_parallelism_threads= n` N° de cores físicos por socket
- `--inter_op_parallelism_threads= n` N° de sockets
- `--data_format=[NHWC|NCHW]` Tipo de layout de los datos



Optimización para Tensorflow

- Formato de datos:
 - N: batch
 - C: canal
 - WxH (ancho x alto)
- Recomendación para oneDNN: NCHW
- Valor por defecto de TensorFlow: NHWC

Explotación de una GPU

- TF puede utilizar una GPU de forma totalmente transparente
 - No necesita cambios en el código
- En FT3 es necesario estar en un *compute node* con una GPU disponible: *compute -gpu*

```
import tensorflow as tf
print("Num GPUs Available:
", len(tf.config.list_physical_devices('GPU'))
)
```

Fuente: <https://www.tensorflow.org/guide/>



Explotación de una GPU: modo de explotación

- Cuando haya una CPU y una GPU disponibles
 - TF prioriza la GPU si la operación a ejecutar tiene una implementación específica para GPU
 - En caso contrario, se ejecuta en la CPU
- El siguiente código permite saber dónde se ejecuta una función (ej. matmul)

```
tf.debugging.set_log_device_placement(True)

# Create some tensors
a = tf.constant([[1.0, 2.0, 3.0], [4.0, 5.0, 6.0]])
b = tf.constant([[1.0, 2.0], [3.0, 4.0], [5.0, 6.0]])
c = tf.matmul(a, b)

print(c)
```

Exploración de una GPU: modo de explotación

- Existe una forma de forzar una ubicación para unos cálculos

```
tf.debugging.set_log_device_placement(True)

# Place tensors on the CPU
with tf.device('/CPU:0'):
    a = tf.constant([[1.0, 2.0, 3.0], [4.0, 5.0, 6.0]])
    b = tf.constant([[1.0, 2.0], [3.0, 4.0], [5.0, 6.0]])

# Run on the GPU
c = tf.matmul(a, b)
print(c)
```

Explotación de una GPU: Límite de memoria

- Por defecto, TF se asigna a toda la memoria de todas la GPUs visibles
 - Podemos usar el método `set_visible_devices` para limitarlo

```
gpus = tf.config.list_physical_devices('GPU')
if gpus:
    # Restrict TensorFlow to only use the first GPU
    try:
        tf.config.set_visible_devices(gpus[0], 'GPU')
        logical_gpus = tf.config.list_logical_devices('GPU')
        print(len(gpus), "Physical GPUs,", len(logical_gpus), "Logical GPU")
    except RuntimeError as e:
        # Visible devices must be set before GPUs have been initialized
        print(e)
```

Explotación de una GPU: Límite de memoria

- También podemos usar el mecanismo experimental *set_memory_growth*
 - Hace un aumento paulatino de la reserva de memoria bajo demanda

```
(...)  
for gpu in gpus:  
    tf.config.experimental.set_memory_growth(gpu, True)  
    logical_gpus = tf.config.list_logical_devices('GPU')  
    print(len(gpus), "Physical GPUs,", len(logical_gpus), "Logical GPUs")  
(...)
```

Explotación de una GPU: Límite de memoria

- También se puede establecer un límite fijo a través del mecanismo *set_logical_device_configuration*

```
# Place tensors on the CPU
with tf.device('/CPU:0'):
    a = tf.constant([[1.0, 2.0, 3.0], [4.0, 5.0, 6.0]])
    b = tf.constant([[1.0, 2.0], [3.0, 4.0], [5.0, 6.0]])

gpus = tf.config.list_physical_devices('GPU')
if gpus:
    # Restrict TensorFlow to only allocate 1GB of memory on the first GPU
    try:
        tf.config.set_logical_device_configuration(
            gpus[0],
            [tf.config.LogicalDeviceConfiguration(memory_limit=1024)])
        logical_gpus = tf.config.list_logical_devices('GPU')
        print(len(gpus), "Physical GPUs,", len(logical_gpus), "Logical GPUs")
    except RuntimeError as e:
        # Virtual devices must be set before GPUs have been initialized
        print(e)
```

Explotación de una GPU única en un sistema multi-GPU

```
tf.debugging.set_log_device_placement(True)

try:
    # Specify an invalid GPU device
    with tf.device('/device:GPU:0'):
        ...
except RuntimeError as e:
    print(e)
```

Tensorflow ClusterResolver

- Es una librería que permite tener acceso a los recursos computacionales reservados en entornos de supercomputación
- Se asocia con el framework ClusterSpec
- Soporta varios sistemas:
 - GCE
 - Kubernetes
 - **Slurm**
 - ...
- Fuente: [Module: tf.distribute.cluster_resolver | TensorFlow v2.11.0](#)

Tensorflow ClusterResolver

- SlurmClusterResolver es el que corresponde con Slurm el sistema de colas de FT3
- Devuelve un objeto ClusterResolver que puede ser usado directamente en TF
- El método cluster_spec devuelve un objeto ClusterSpec para ser usado en Distributed TF

```
tf.distribute.cluster_resolver.SlurmClusterResolver(  
    jobs=None,  
    port_base=8888,  
    gpus_per_node=None,  
    gpus_per_task=None,  
    tasks_per_node=None,  
    auto_set_gpu=True,  
    rpc_layer='grpc'  
)
```


Actividad: ClusterResolver

- https://github.com/diegoandradecanosa/Cesga2023Courses/blob/main/tf_dist/TF/002/README.md

Distribute.strategy de TF: CrossDeviceOps

- Clase para seleccionar la implementación a usar para los algoritmos de
 - Reducción
 - Broadcasting
- Es uno de los parámetros que podemos pasar a la MirroredStrategy
- Implementaciones:
 - `tf.distribute.ReductionToOneDevice`
 - Copia todos los valores a un dispositivo donde se hará la reducción de forma centralizada
 - `tf.distribute.NcclAllReduce`
 - Usa la implementación de Nvidia NCCL para el all reduce
 - `tf.distribute.HierarchicalCopyAllReduce`
 - Utiliza un algoritmo de reducción jerárquica
 - Pensado para Nvidia-DGX1
 - Asume que las GPUs están interconectadas como en ese tipo de nodo

Fuente: https://www.tensorflow.org/api_docs/python/tf/distribute/CrossDeviceOps



Distribute.strategy de TF: DistributedDataSet

- Clase que permite definir un *dataset* distribuido entre varios nodos
 - Apropiado para su uso con el módulo `tf.distribute.strategy`
- Dos APIs diferentes:
 - [`tf.distribute.Strategy.experimental_distribute_dataset\(dataset\)`](#)
 - Más sencillo de utilizar si tenemos un dataset convencional
 - [`tf.distribute.Strategy.distribute_datasets_from_function\(dataset_fn\)`](#)
 - Más difícil de utilizar pero más flexible

Fuente:

https://www.tensorflow.org/api_docs/python/tf/distribute/DistributedDataset

Distribute.strategy de TF: DistributedDataSet

- Concepto más amplio: Dataset sharding
 - Distribución del conjunto de datos entre varios nodos

Distributed training en TF

- El uso de hardware *en paralelo* puede reducir el tiempo de entrenamiento
- La paralelización del entrenamiento requiere esfuerzo por parte del programador
 - El uso de una GPU o una CPU sí que no requiere ese esfuerzo
- La paralelización requiere que TF sepa cómo coordinar el trabajo de varios trabajadores (*workers*)

Fuente: <https://www.youtube.com/watch?v=S1tN9a4Proc>

Distribute.Strategy de TF

- Tf.distribute.Strategy es una API de TF para distribuir el entrenamiento entre múltiples GPUs, máquinas o TPUs
- Permite ejecutar los entrenamientos en paralelo ávidamente (*eagerly*) o siguiendo una estrategia de grafo (usando tf.function)
 - Ávidamente -> depuración
 - Tf.function -> Recomendado
- [Fuente: Distributed training with TensorFlow | TensorFlow Core](#)

Distributed training en TF

- Categorías de algoritmos paralelos de TF
 - Paralelismo de datos (data parallism)
 - Paralelismo de modelo (model parallism)

Distributed training en TF: Data parallelism

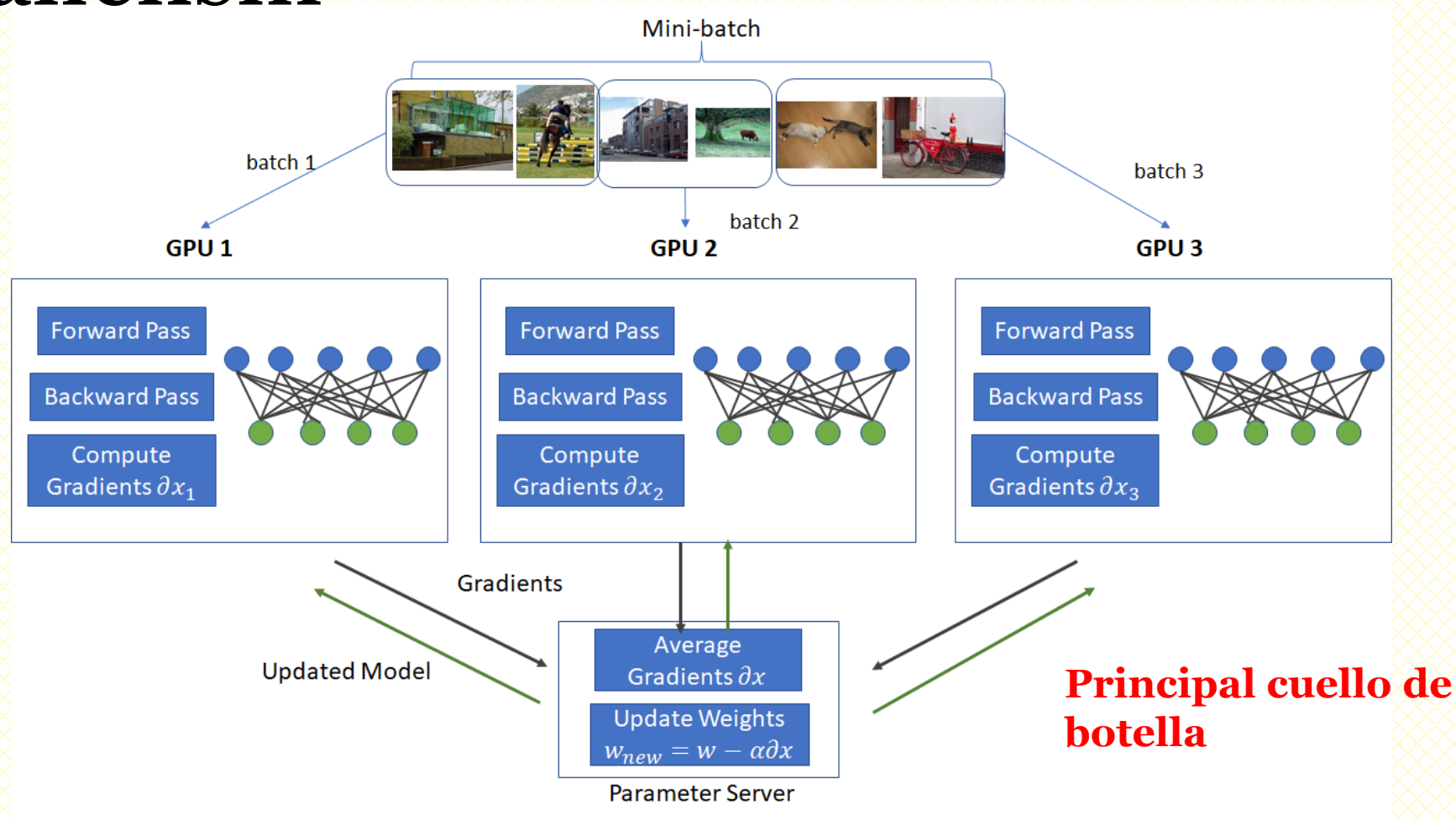
- Paralelismo de datos (data parallelism)

`model.fit(x,y,batch_size=32)`

Dimensiones del entrenamiento:

- Epoch (procesado de todo el *dataset*). Una pasada completa del *dataset*
- En cada *step* procesamos *batch_size* elementos del *data_set* a la vez
 - Incrementar el *batch_size* está limitado por la memoria de una GPU
 - Incrementarlo mejora el rendimiento -> Podemos hacer más cosas en paralelo
 - Usando varios *workers*
 - Podemos seguir aumentando el *batch_size*
 - Se divide efectivamente entre varias GPUs
 - Y acortar el entrenamiento
- Usando varios *workers*
 - Cada uno procesa un *step* del entrenamiento de forma independiente calculando sus propios gradientes
 - Estos gradientes son *reducidos* (promediados) entre todos los trabajadores y usados en la actualización de los pesos

Distributed training en TF: Data parallelism



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

Distributed training en TF: Data parallelism

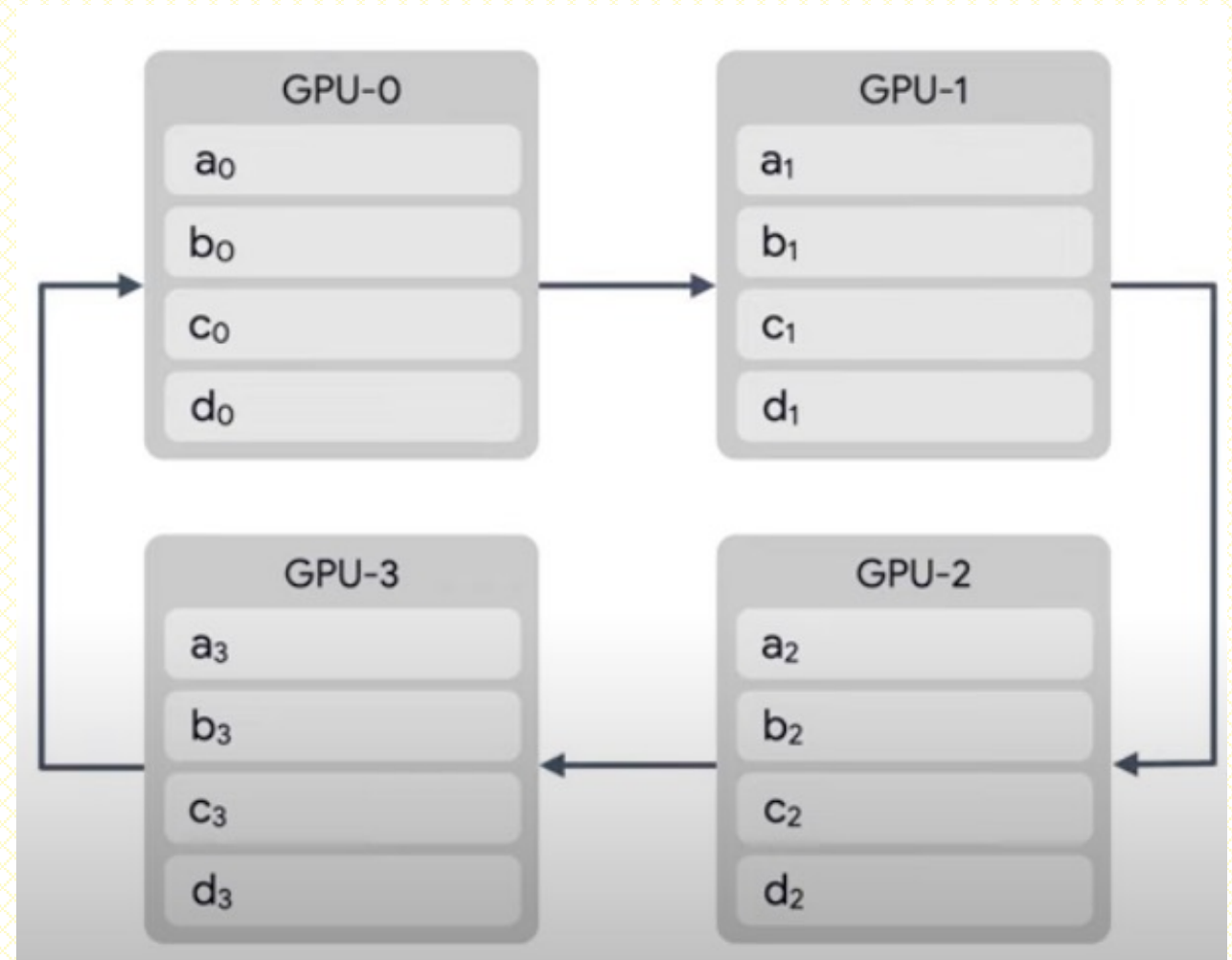
- Cuello de botella: actualización de los gradientes por todos los *workers*
 - Basada en la operación all-reduce
 - Los valores de un array se promedian a partir de los valores de la copia privada del array de los trabajadores
 - El array global, con sus valores calculados, se transfiere de vuelta a los trabajadores
 - Hay múltiples implementaciones del algoritmo all-reduce
 - Dependiendo de la topología de los trabajadores
 - El patrón de intercambio de valores
 - TF se encarga de seleccionar el algoritmo que realizará la operación de la forma más eficiente en cada caso

Distributed training en TF: All-reduce ring

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

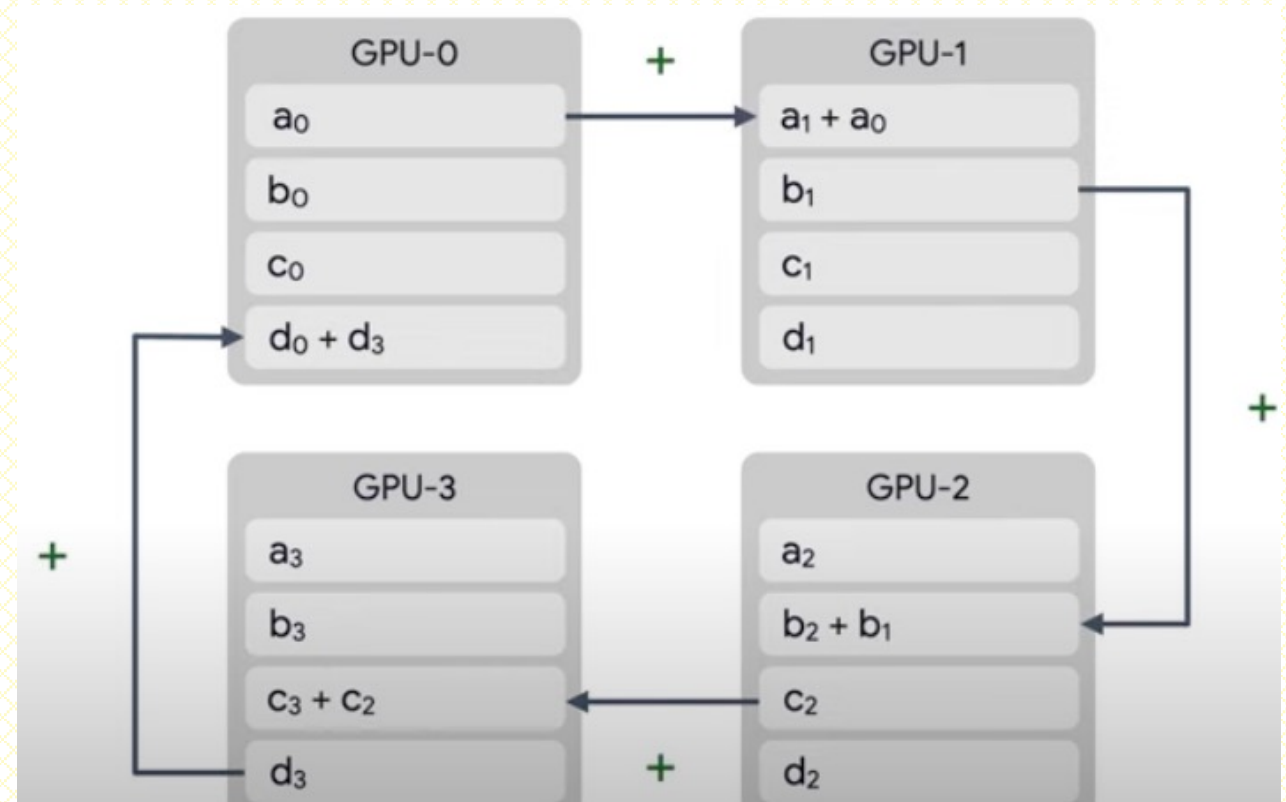
Distributed training en TF: All-reduce ring

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



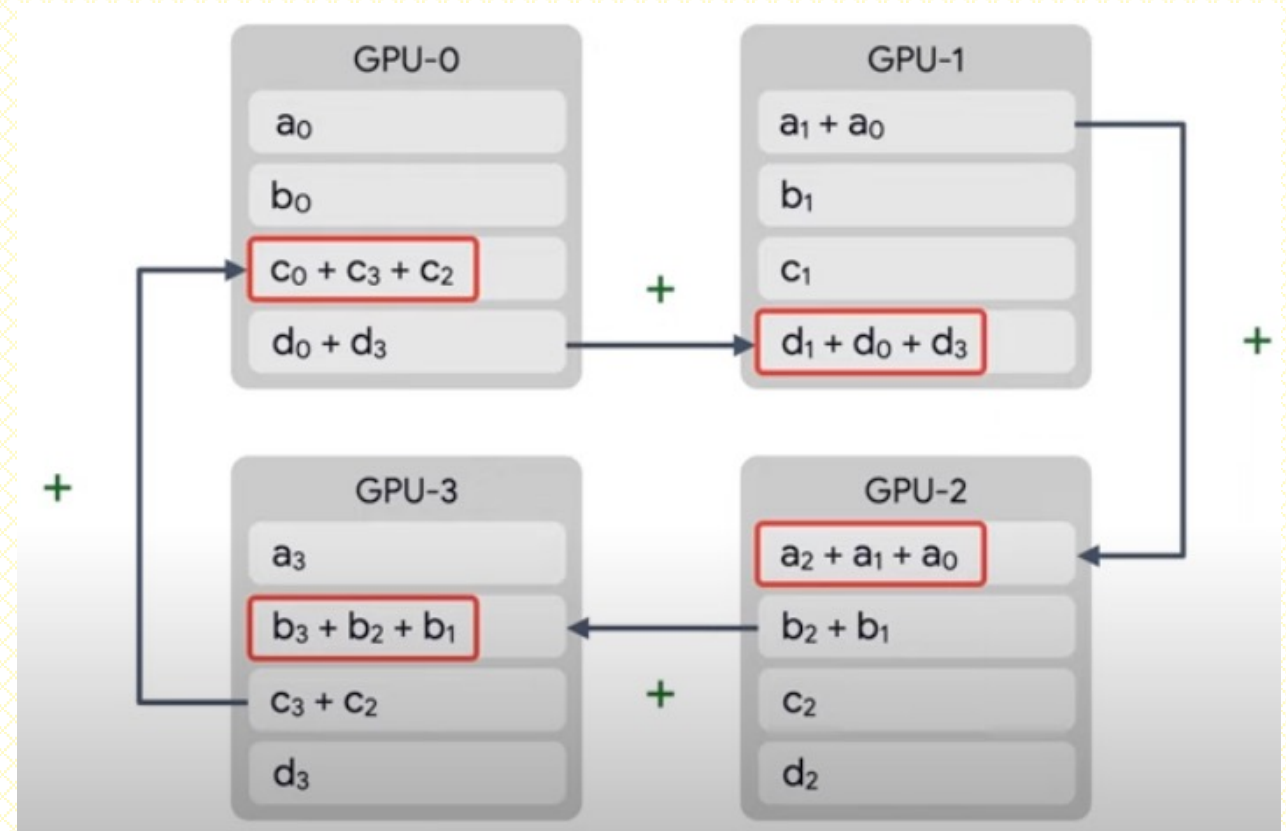
Distributed training en TF: All-reduce ring

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



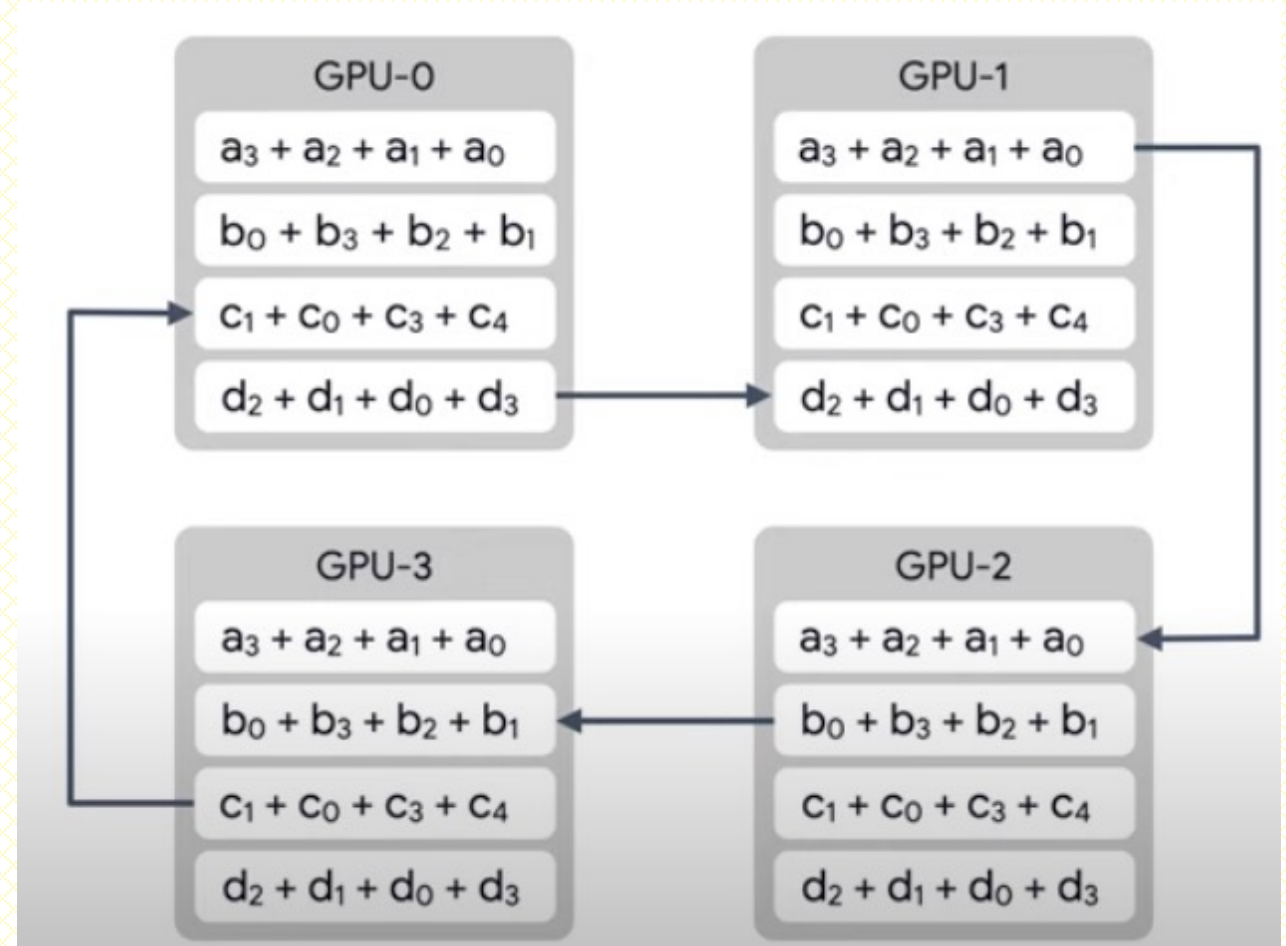
Distributed training en TF: All-reduce ring

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



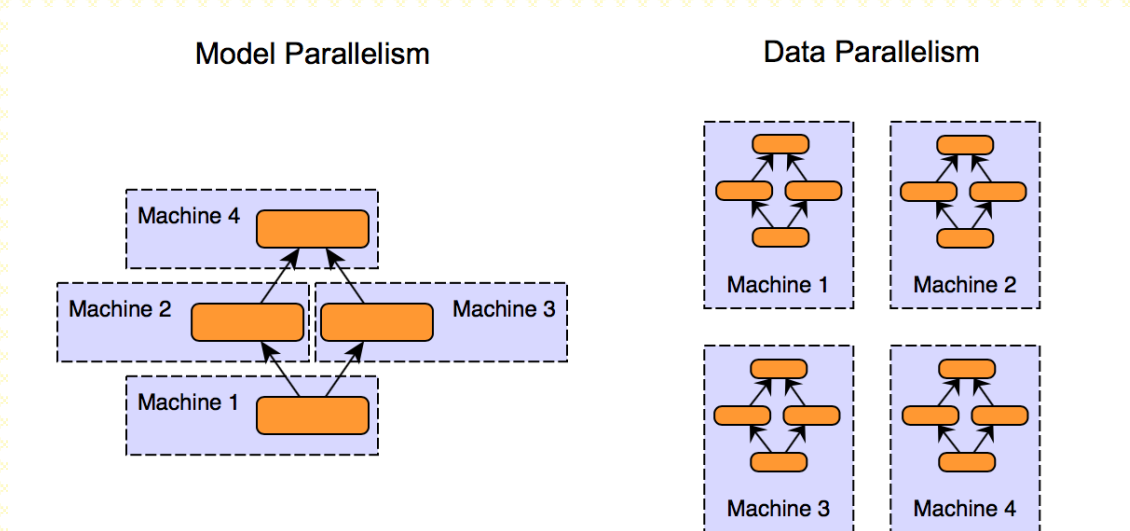
Distributed training en TF: All-reduce ring

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



Distributed training en TF: Model parallelism

- Paralelismo de modelo (*model parallelism*)
 - Dividimos la arquitectura del modelo entre varios *workers*
 - 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](#)

Distribute.Strategy de TF

- Tipos de estrategia, se cubren varias posibilidades en un abanico amplio dependiendo de varios factores:
 - Entrenamiento síncrono o asíncrono
 - Son dos estrategias distintas para aplicar paralelismo de datos
 - Condicionadas por el cuello de botella de la actualización de los gradientes
 - Operación all-reduce
 - Síncrono: Se divide el dataset de entrenamiento entre los diferentes trabajadores que realizan el entrenamiento de forma independiente, y los gradientes se agregan al final de cada step
 - Asíncrono: No se divide el dataset, todos los trabajadores lo usan por completo, y actualizan las variables de forma asíncrona
 - Plataformas hardware utilizada
 - Tipo de acelerador utilizado: Multicore CPU, GPU, TPU, etc...
 - Un nodo con múltiples aceleradores o varios nodos

Distribute.Strategy de TF

- Estrategias disponibles en TF:
 - Síncrono
 - OneDeviceStrategy -> https://www.tensorflow.org/api_docs/python/tf/distribute/OneDeviceStrategy
 - MirroredStrategy
 - TPUStrategy
 - MultiWorkerMirroredStrategy
 - Asíncrono
 - ParameterServerStrategy
 - CentralStorageStrategy
 - https://www.tensorflow.org/guide/distributed_training
 - https://www.tensorflow.org/api_docs/python/tf/distribute/Strategy

Distribute.StrategyExtended

- API adicional para algoritmos que necesitan ser *distribution-aware*
- https://www.tensorflow.org/api_docs/python/tf/distribute/StrategyExtended

Distribute.Strategy de TF

Grado de soporte de estrategias en TF en diversos escenarios

Training API	MirroredStrategy	TPUStrategy	MultiWorkerMirroredStrategy	CentralStorageStrategy	ParameterServerStrategy
Keras Model.fit	Supported	Supported	Supported	Experimental support	Experimental support
Custom training loop	Supported	Supported	Supported	Experimental support	Experimental support
Estimator API	Limited Support	Not supported	Limited Support	Limited Support	Limited Support

Distribute.Strategy de TF

Grado de soporte de estrategias en TF en diversos escenarios

Training API	MirroredStrategy	TPUStrategy	MultiWorkerMirroredStrategy	CentralStorageStrategy	ParameterServerStrategy
Keras Model.fit	Supported	Supported	Supported	Experimental support	Experimental support
Custom training loop	Supported	Supported	Supported	Experimental support	Experimental support
Estimator API	Limited Support	Not supported	Limited Support	Limited Support	Limited Support

Distribute.Strategy de TF: MirroredStrategy

- La MirroredStrategy soporta entrenamiento distribuido síncrono en múltiples GPUs en un nodo
- Se crea una réplica por cada GPU
 - Juntas forman una única variable conceptual llamada MirroredVariable
 - Se mantiene la coherencia aplicando actualizaciones similares en todas
 - Como si fuese un espejo (mirror)
 - Implementaciones eficientes de algoritmos all-reduce

```
mirrored_strategy = tf.distribute.MirroredStrategy(devices=["/gpu:0", "/gpu:1"])
```

Distribute.Strategy de TF: MirroredStrategy

- Cada GPU realiza la *forward pass* en un subconjunto diferente de los datos de entrada para calcular la *loss function*
- Cada GPU calcula sus propios gradientes basándose en la *loss function* calculada localmente
- Se realiza la agregación global (promedio) de estos gradientes a través de un algoritmo *all-reduce*
- Se actualizan los pesos usando los gradientes resultantes
 - Todos los dispositivos tendrán una copia sincronizada (espejo) del modelo entrenado

Actividad: Entrenamiento 1 nodo – 2 GPUs

- https://github.com/diegoandradecanosa/Cesga2023Courses/tree/main/tf_dist/TF/003#actividad-un-nodo-dos-gpus

Actividad: Entrenamiento 1 nodo – 2 GPUs

- Solución de problemas comunes
 - Si se cuelga el kernel hay que reiniciarlo
 - Kernel -> Restart kernel
 - O la combinación de teclas “o+o”
 - Si falla la ejecución por problemas de uso de memoria, entonces podemos matar los procesos que hayan quedado ejecutándose en la GPU
 - nvidia-smi Al final del comando habrá una lista de procesos
 - Eliminarlos con `kill -9 pid`

Distribute.Strategy de TF: TPUStrategy

- Específica para Google TPUs
- Similar a MirroredStrategy
- Usa una implementación específica de las operaciones all-reduce optimizada para TPUs

```
cluster_resolver =  
tf.distribute.cluster_resolver.TPUClusterResolver(  
    tpu=tpu_address)  
tf.config.experimental_connect_to_cluster(cluster_resolver)  
tf.tpu.experimental.initialize_tpu_system(cluster_resolver)  
tpu_strategy = tf.distribute.TPUStrategy(cluster_resolver)
```

Distribute.Strategy de TF: MultiWorkerMirroredStrategy

- MultiWorkerMirroredStrategy es similar a MirroredStrategy pero con soporte para varios nodos
 - Crea copias de todas las variables en todos los trabajadores y en todos los dispositivos

```
communication_options = tf.distribute.experimental.CommunicationOptions(  
    implementation=tf.distribute.experimental.CommunicationImplementation.NCCL)  
strategy =  
tf.distribute.MultiWorkerMirroredStrategy(communication_options=communication_options)
```

Hay 2 opciones para las Comunicaciones entre dispositivos:

- .RING: Basado en RPC, válido para CPU y GPU
- .NCCL: Específico para GPU, mejor rendimiento cuando se puede utilizar
- .AUTO: Deja a TF elegir el mejor método disponible

Distribute.Strategy de TF: MultiWorkerMirroredStrategy

- El uso de múltiples nodos requiere configurar la variable de entorno: TF_CONFIG.
 - Tiene estructura de diccionario
 - Dos componentes:
 - La definición de un **cluster**
 - **Diccionario con listas de nodos (host:puerto) de distintos tipos:**
 - **Ps: servidores**
 - **Workers: trabajadores**
 - La definición de cada tarea (**task**)
 - Type: worker/ps
 - Index

```
os.environ["TF_CONFIG"] = json.dumps({  
    "cluster": {  
        "worker": ["host1:port", "host2:port", "host3:port"],  
        "ps": ["host4:port", "host5:port"]  
    },  
    "task": {"type": "worker", "index": 1}  
})
```

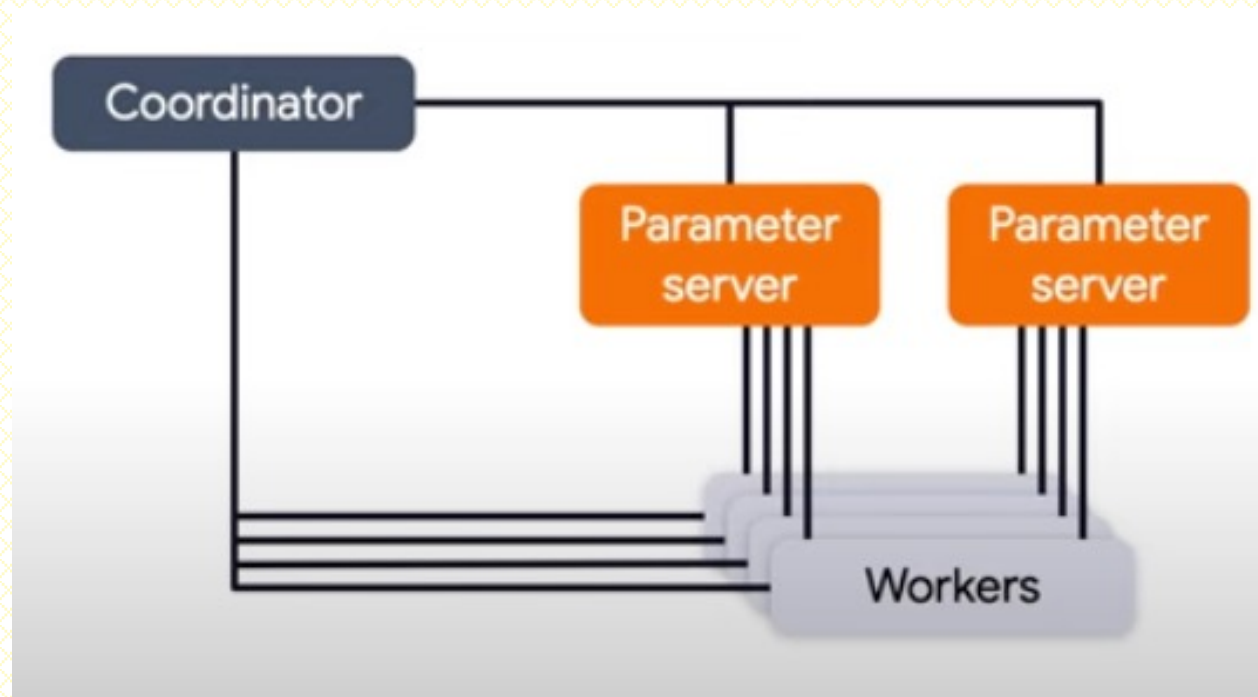
Fuente: [Distributed training with TensorFlow | TensorFlow Core](#)

Actividad: Entrenamiento 2 nodos – 2 GPUs

- https://github.com/diegoandradecanosa/Cesga2023Courses/blob/main/tf_dist/TF/004/README.md

Distribute.Strategy de TF: ParameterServerStrategy

- Es un tipo de entrenamiento multimodo asíncrono
 - Reduce el cuello de botella del all-reduce en las estrategias síncronas
 - Recomendable para usar un nodo alto de trabajadores
- Los nodos implicados se dividen en:
 - *Workers* (`tf.distribute.Server`)
 - *Parameter servers* (`tf.distribute.Server`)
 - Un *coordinator* (`tf.distribute.experimental.coordinator.ClusterCoordinator`)
 - Usa la *ParameterServerStrategy* para definir el paso de entrenamiento y usar un *ClusterCoordinator* que envía pasos de entrenamiento a los trabajadores

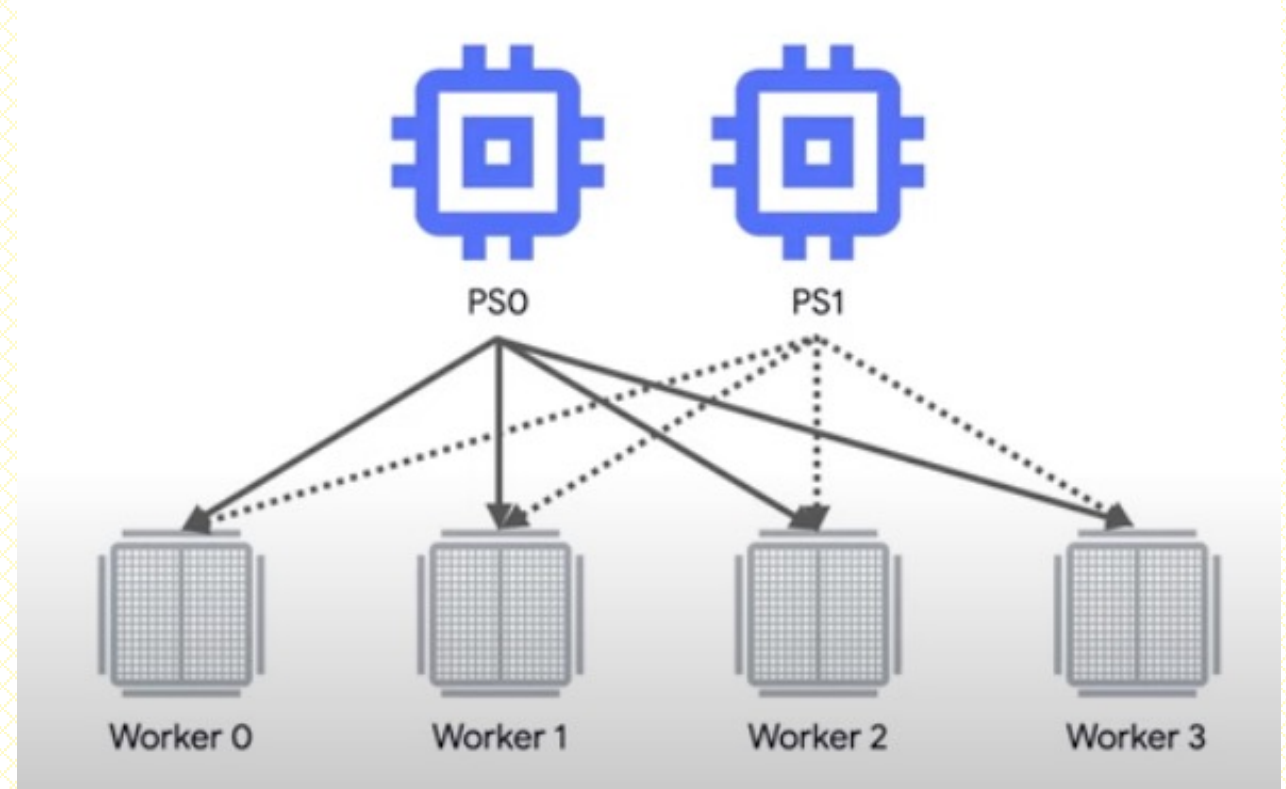


Distribute.Strategy de TF: ParameterServerStrategy

- Soporta dos modos de entrenamiento
 - Keras Model.fit
 - Bucles de entrenamiento definidos por el usuario
- Abstracciones de Model.fit
 - Cluster, Jobs y Tasks
- Con PS, también tenemos:
 - Un Coordinator job (called *chief*)
 - Varios Worker jobs (llamados *workers*)
 - Varios PS Jobs (llamados *ps*)

Distribute.Strategy de TF: ParameterServerStrategy

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



Preparación del Clúster

- Components: 1 Coordinator (type *chief*), N PS (*ps*), N Workers (*worker*), y puede que una tarea Evaluator
- La tarea de coordinación necesita conocer las direcciones y puertos de todas las tareas Server (PS y Workers), pero no del Evaluator
- Las tareas Server deben saber en qué puerto escuchar
- La tarea Evaluator no tiene por qué conocer la configuración del Clúster
- La estrategia PS usará todas las GPUs disponibles en cada nodo
 - Todos deben tener el mismo número de GPUs

PS con Keras model.fit(): Esqueleto

```
variable_partitioner = (  
    tf.distribute.experimental.partitioners.MinSizePartitioner(  
        min_shard_bytes=(256 << 10),  
        max_shards=NUM_PS))  
  
strategy = tf.distribute.experimental.ParameterServerStrategy(  
    tf.distribute.cluster_resolver.TFConfigClusterResolver(),  
    variable_partitioner=variable_partitioner)  
coordinator = tf.distribute.experimental.coordinator.ClusterCoordinator(  
    strategy)  
  
with strategy.scope():  
    //model definition  
    model.compile(...)  
  
model.fit(...)
```

Concepto relacionado: Variable sharding

- Consiste en dividir una variable en variables más pequeñas llamadas *shards*
- Útil para:
 - Reducir consumo de red
 - Distribuir la carga de computación y almacenamiento de una variable
 - Útil, por ejemplo, para *embeddings* muy grandes que no caben en la memoria de un dispositivo
- Cómo hacerlo: Pasando un *variable_partitioner* al construir un objeto *ParameterServerStrategy*
- El particionador entonces se llamará cada vez que se cree una variable, y devuelve un nuevo de shards particionando en cada dimensión de la variable
- Varios particionadores disponibles: [Min/Max/Fixed]SizePartitioner

PS con bucle de usuario

- Creación de una instancia de un ClusterCoordinator para enviar trabajos (normalmente *steps* de entrenamiento) para su ejecución en otros *workers*
 - Opcional trabajando con Keras Model.fit
 - Necesario con bucles de entrenamiento de usuario

Definición del *step* de entrenamiento

Wrapper `tf.function`

`@tf.function`

`def step_fn(iterator):`

`def replica_fn(batch_data, labels):`

`with tf.GradientTape() as tape:`

`pred = model(batch_data, training=True)`

`per_example_loss = tf.keras.losses.BinaryCrossentropy(`

`reduction=tf.keras.losses.Reduction.NONE)(labels, pred)`

`loss = tf.nn.compute_average_loss(per_example_loss)`

`model_losses = model.losses`

`if model_losses:`

`loss += tf.nn.scale_regularization_loss(tf.add_n(model_losses))`

`gradients = tape.gradient(loss, model.trainable_variables)`

`optimizer.apply_gradients(zip(gradients, model.trainable_variables))`

`actual_pred = tf.cast(tf.greater(pred, 0.5), tf.int64)`

`accuracy.update_state(labels, actual_pred)`

`return loss`

`batch_data, labels = next(iterator)`

`losses = strategy.run(replica_fn, args=(batch_data, labels))`

`return strategy.reduce(tf.distribute.ReduceOp.SUM, losses, axis=None)`

Inferencia para un batch

1. Siguiendo batch
2. Ejecutar step de entrenamiento para cada batch
3. Reducción de los resultados en cada trabajador

Definición del ClusterCoordinator (I)

Instancia del coordinador

```
coordinator = tf.distribute.coordinator.ClusterCoordinator(strategy)

...

@tf.function
def per_worker_dataset_fn():
    return strategy.distribute_datasets_from_function(dataset_fn)

per_worker_dataset = coordinator.create_per_worker_dataset(per_worker_dataset_fn)
per_worker_iterator = iter(per_worker_dataset)
```

Replica el conjunto de datos entre los trabajadores

Definición del ClusterCoordinator (II)

```
num_epochs = 4
steps_per_epoch = 5
for i in range(num_epochs):
    accuracy.reset_states()
    for _ in range(steps_per_epoch):
        coordinator.schedule(step_fn, args=(per_worker_iterator,))
    # Wait at epoch boundaries.
    coordinator.join()
    print("Finished epoch %d, accuracy is %f." % (i, accuracy.result().numpy()))
    ...

loss = coordinator.schedule(step_fn, args=(per_worker_iterator,))
print("Final loss is %f" % loss.fetch())
```

Para cada step

Envío de steps a los trabajadores

Punto de sincronización

Central Storage Strategy

- Es una estrategia de tipo servidor de parámetros que pone todas las variables en el mismo dispositivo

```
strategy = tf.distribute.experimental.CentralStorageStrategy()
# Create a dataset
ds = tf.data.Dataset.range(5).batch(2)
# Distribute that dataset
dist_dataset = strategy.experimental_distribute_dataset(ds)

with strategy.scope():
    @tf.function
    def train_step(val):
        return val + 1

# Iterate over the distributed dataset
for x in dist_dataset:
    # process dataset elements
    strategy.run(train_step, args=(x,))
```

https://www.tensorflow.org/api_docs/python/tf/distribute/experimental/CentralStorageStrategy

Actividad: ParameterServer Ejemplo simple

- https://github.com/diegoandradecanosa/Cesga2023Courses/tree/main/tf_dist/TF/005#parameter-server-ejemplo-simple

Entrenamiento distribuido con DTENSORS

- DTensor es una extensión de Tensorflow para computación distribuida síncrona (desde TF 2.9)
- Proporciona un modelo de programación que opera globalmente sobre tensores manejando de forma transparente la distribución en dispositivos
- La distribución se realiza en base a directivas de *sharding* (fragmentación)
- Si un código de TF utiliza DTensors, ese mismo código se puede ejecutar en un número variable de dispositivos

https://www.tensorflow.org/guide/dtensor_overview

https://www.tensorflow.org/tutorials/distribute/dtensor_ml_tutorial

Entrenamiento Distribuido con DTensors

- El mecanismo de Dtensors es apropiado para distribuir el entrenamiento de modelos entre varios dispositivos
- Adecuado para
 - Paralelismo de datos
 - Paralelismo de modelo
 - Particionado Spatial (equivalente a paralelismo tensorial)

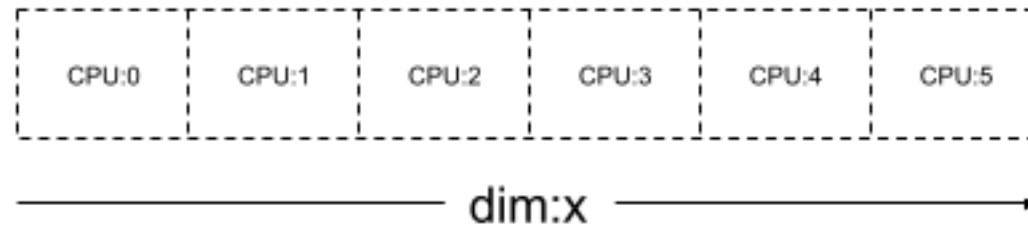
Fuente:

https://www.tensorflow.org/tutorials/distribute/dtensor_ml_tutorial

DTENSORS: Conceptos

- Se basa en dos conceptos básicos:
 - Mesh: define la lista de dispositivos disponibles
 - Podemos tener un grid con varias dimensiones
 - Layout: define cómo distribuir la dimensión Tensor sobre una Mesh

```
dtensor.create_mesh([('x', 6)], devices=DEVICES)
```



```
mesh_1d = dtensor.create_mesh([('x', 6)], devices=DEVICES)  
print(mesh_1d)
```

Anatomía de un DTENSOR

- Se tratar de un Tensor pero enriquecido con la anotación Layout que define su política de distribución. Consta de:
 - Shape y dtype como cualquier tensor
 - Un Layout que define la Mesh a la que pertenece el Tensor, y como este es distribuido sobre la Mesh
 - Una lista de tensores-componentes con un ítem por dispositivo local de la Mesh
- Las operaciones unpack y pack de un Dtensor, permiten extraer los vectores componentes, y devolver el tensor a su versión compacta, respectivamente

Ejemplo de un Dtensor 2D

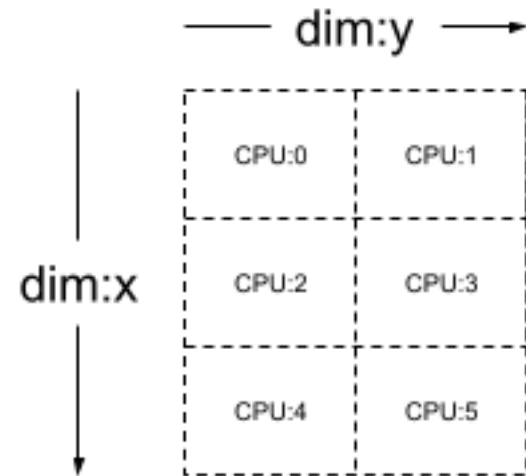
- Consideremos una *mesh* 2D formada por 3x2 dispositivos:

```
mesh = dtensor.create_mesh([("x", 3), ("y", 2)], devices=DEVICES)
```

- Creemos un tensor 3x2 con rank-2
 - La primera dimensión se distribuye por la dimensión x
 - La segunda por la dimensión y
 - Con esta distribución cada dispositivo recibe un elemento del tensor

DTENSORS: Meshes

```
dtensor.create_mesh(  
    [('x', 3), ('y', 2)],  
    devices=DEVICES)
```

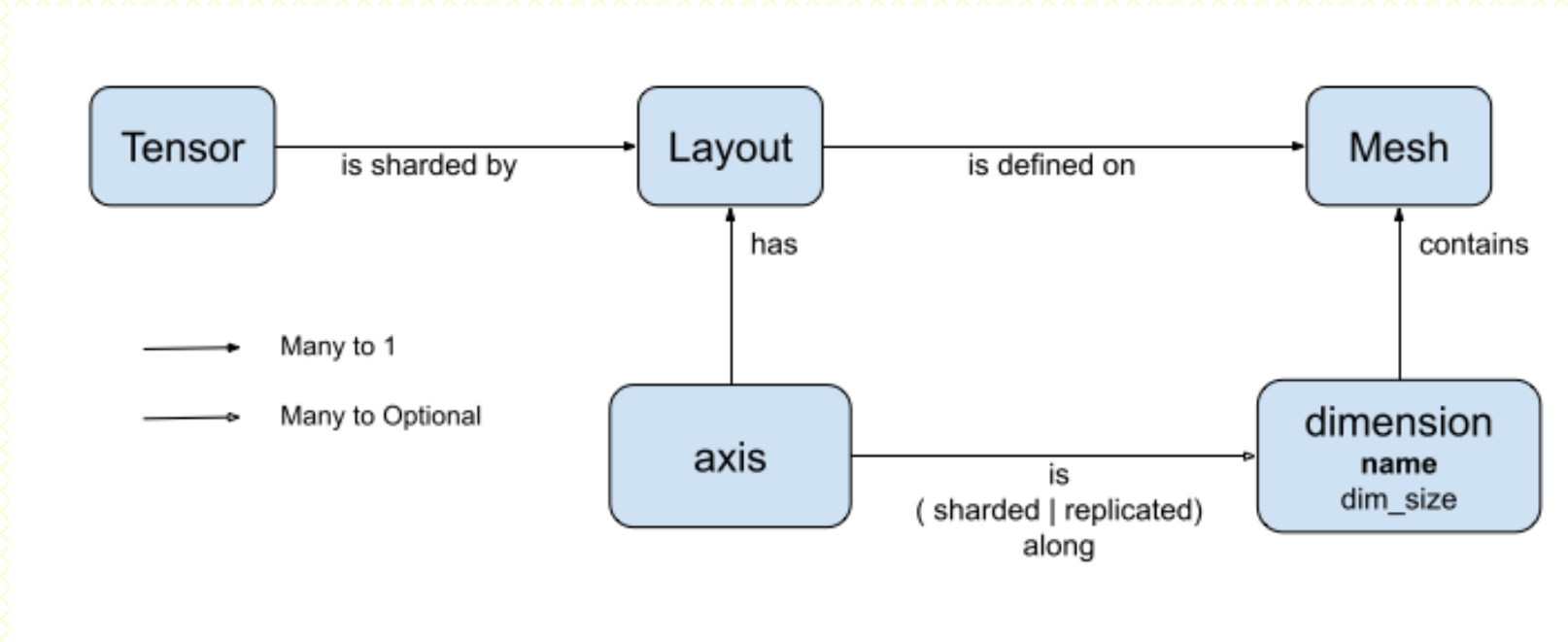


```
mesh_2d = dtensor.create_mesh([('x', 3), ('y', 2)], devices=DEVICES)  
print(mesh_2d)
```

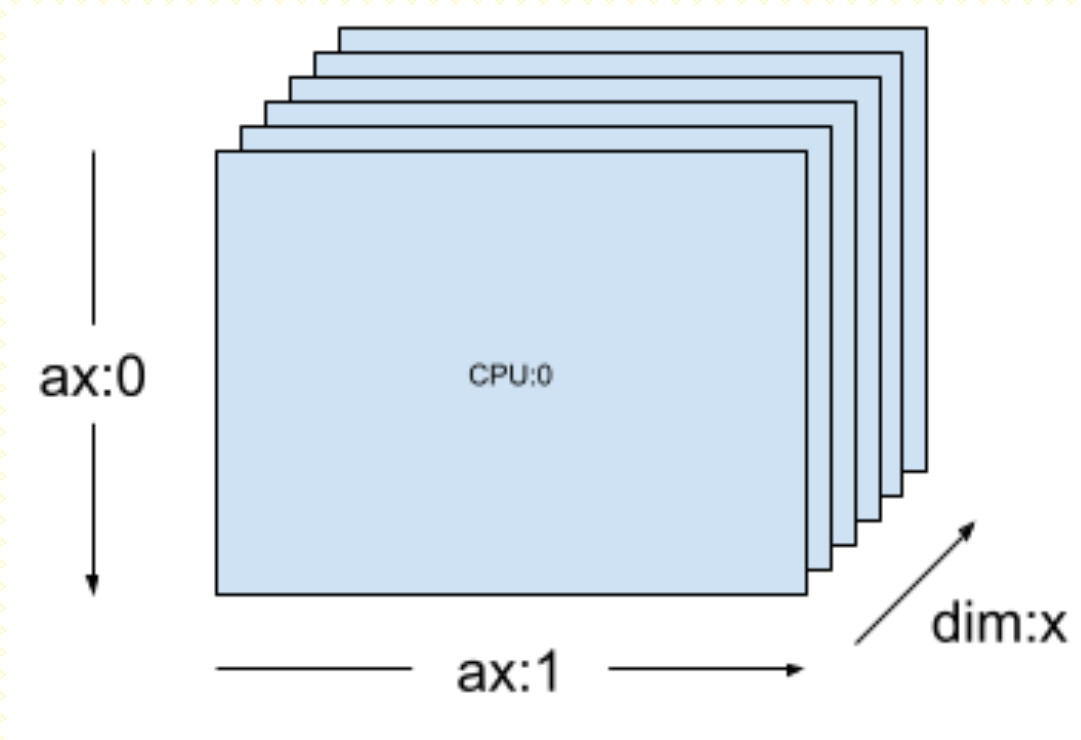
DTENSORS: Layouts

- Un Layout indica cómo distribuir un tensor sobre una Mesh
- El tamaño de un Layout debe ser el mismo que el del Tensor sobre el que se aplica
- Cada dimensión del Layout/Tensor hay que especificar
 - La dimensión del Mesh a través de la que se distribuye
 - Si la dimensión es unsharded: entonces se replica a través de esa dimensión de la Mesh

DTENSORS: Layout, Tensor, Mesh

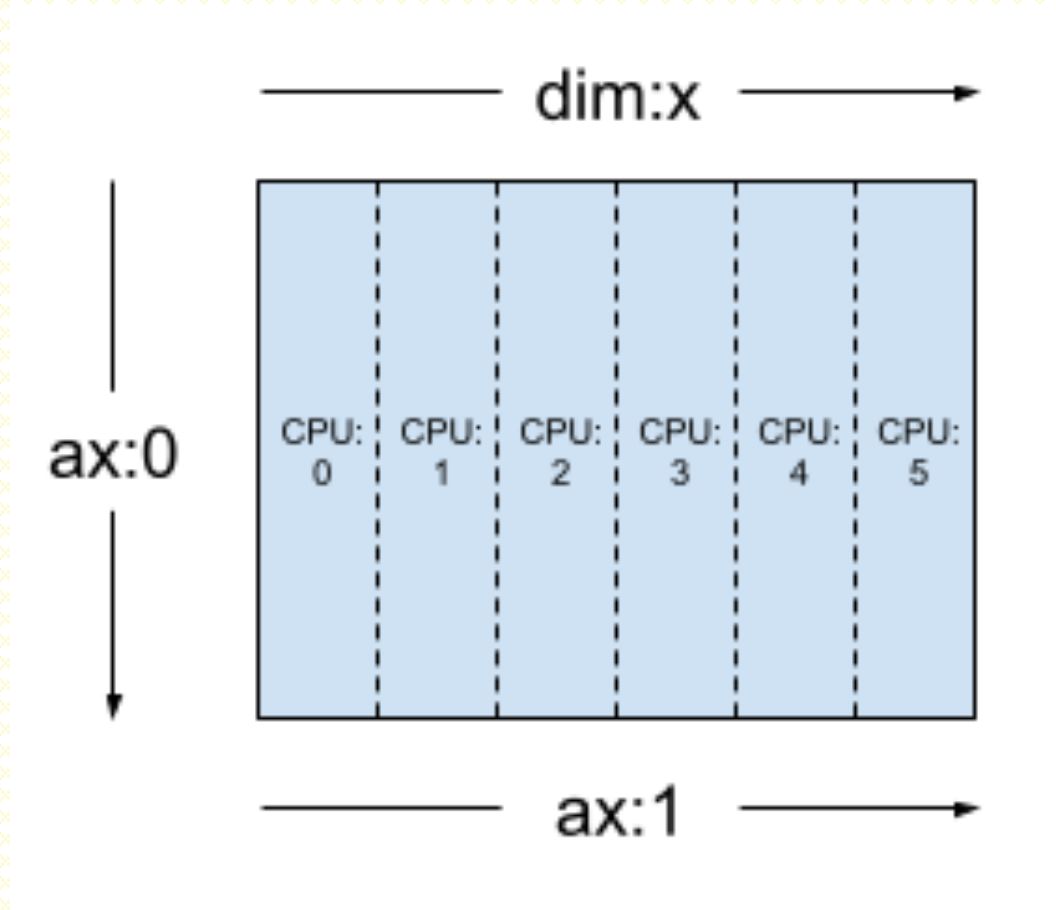


DTENSORS: Ejemplos de meshes vs layouts



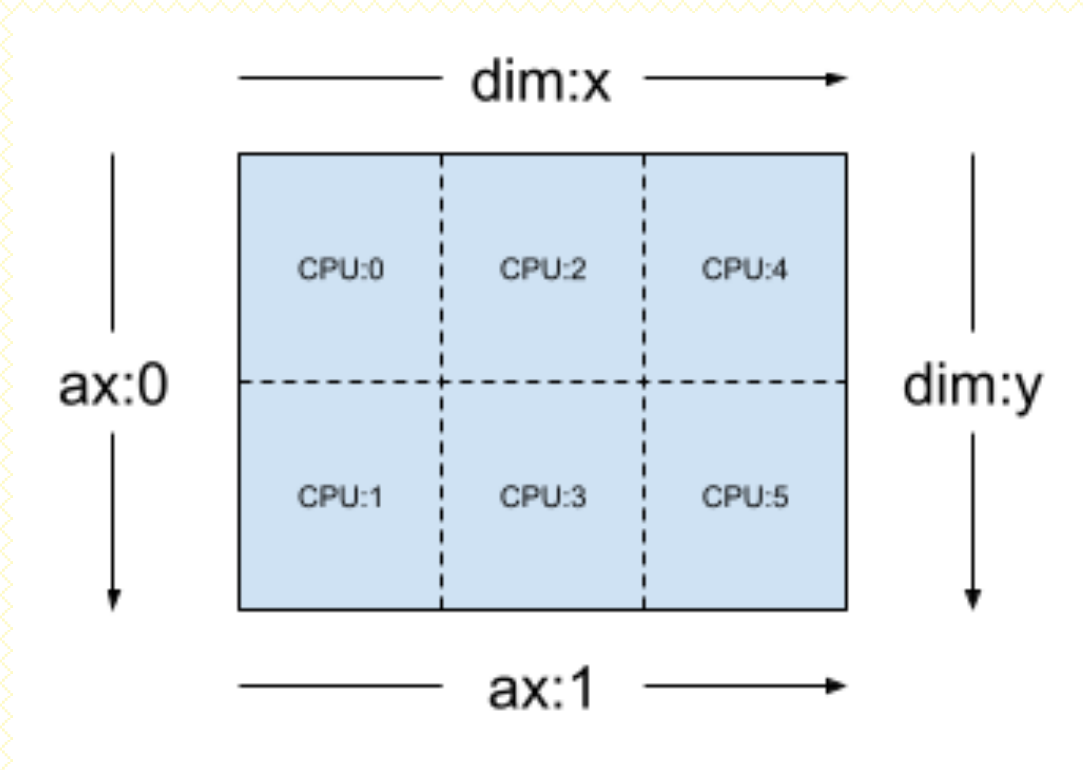
- Mesh: `[("x", 6)]`
- Layout: `Layout(["unsharded", "unsharded"], mesh_1d)`

DTENSORS: Ejemplos de meshes vs layouts



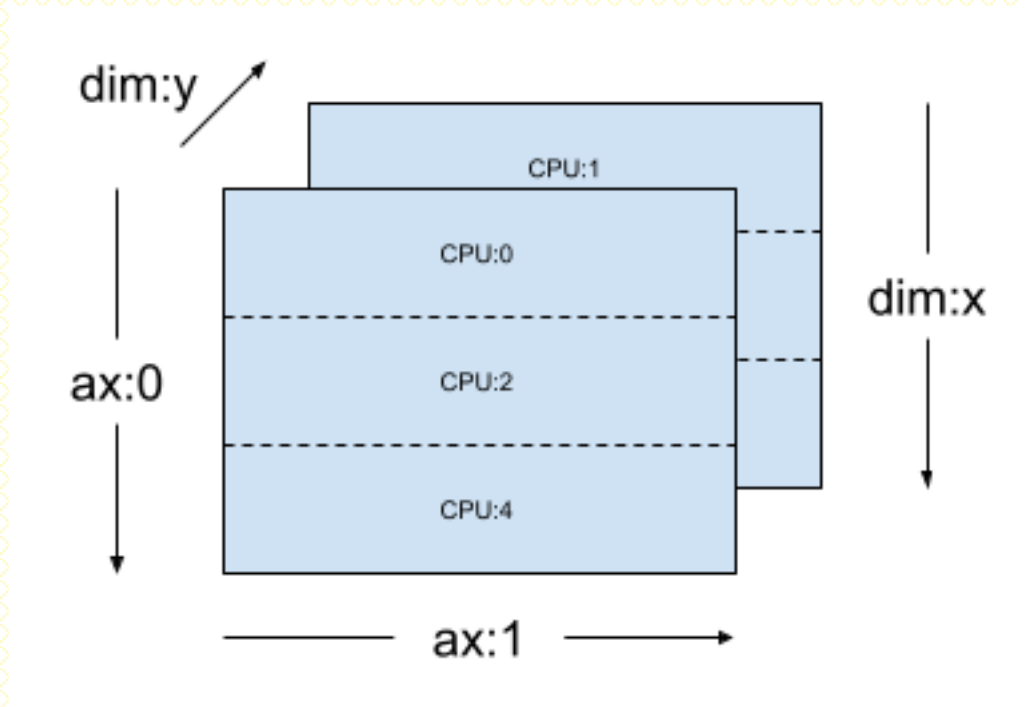
- Mesh: `[("x", 6)]`
- Layout:
`Layout(['unsharded', 'x'])`

DTENSORS: Ejemplos de meshes vs layouts



- Mesh: `[("x", 3), ("y", 2)]`
- Layout: `Layout(["y", "x"], mesh_2d)`

DTENSORS: Ejemplos de meshes vs layouts



- Mesh: `[("x", 3), ("y", 2)]`
- Layout: `Layout(["x", dtensor.UNSHARDED], mesh_2d)`

DTENSORS: Cliente

- Dos escenarios posibles:
 - Un cliente ejecutando un único proceso Python
 - Varios clientes ejecutando varios procesos Python actuando de forma colectiva como una aplicación coherente
 - Todos los clientes ven la misma Mesh, pero cada dispositivo es local o global según el cliente
 - [dtensor.create_distributed_mesh](#)

DTENSORS: Creación

- La función `dtensor_from_array` permite crear un Dtensor a partir de algo como un `tf.Tensor`, en dos pasos:
 - El Tensor se replica en cada dispositivo de la Mesh
 - Distribuye (*shards*) la copia al Layout solicitado a través de sus argumentos

DTENSORS: Creación

```
def dtensor_from_array(arr, layout, shape=None, dtype=None):  
    if shape is not None or dtype is not None:  
        arr = tf.constant(arr, shape=shape, dtype=dtype)  
  
    # replicate the input to the mesh  
    a = dtensor.copy_to_mesh(arr,  
                             layout=dtensor.Layout.replicated(layout.mesh, rank=layout.rank))  
    # shard the copy to the desirable layout  
    return dtensor.relayout(a, layout=layout)
```


DTENSORS: Creación

```
mesh = dtensor.create_mesh([("x", 6)], devices=DEVICES)
layout = dtensor.Layout([dtensor.UNSHARDED], mesh)

my_first_dtensor = dtensor_from_array([0, 1], layout)

# Examine the DTensor content
print(my_first_dtensor)
print("global shape:", my_first_dtensor.shape)
print("dtype:", my_first_dtensor.dtype)
```

DTENSOR: Ejemplo básico

```
import tensorflow as tf
from tensorflow.experimental import dtensor

print('TensorFlow version:', tf.__version__)

def configure_virtual_gpus(ngpu):
    phy_devices = tf.config.list_physical_devices('GPU')
    tf.config.set_logical_device_configuration(phy_devices[0], [
        tf.config.LogicalDeviceConfiguration(),
    ] * ngpu)

configure_virtual_gpus(2)
DEVICES = [f'GPU:{i}' for i in range(2)]

tf.config.list_logical_devices('GPU')
```

DTENSOR: Compatibilidad con TF

- Al ser un reemplazo para el tipo Tensor, funciona también con los mecanismos compatibles con estos:
 - `tf.function`
 - `tf.GradientTape`
- Para ello, el TF Graph se convierte en un SPMD Graph a través de un proceso llamado expansión SPMD

Dtensors y el API de TF

- Los Dtensors coexisten con el API de TF como un reemplazo de los tensores convencionales
 - `tf.function`
 - `tf.GradientTape`
- Para ello, para cada TF Graph, Dtensor genera y ejecuta un grafo SPMD equivalente
 - *SPMD expansión*
 - Propagación del Layout de Dtensor a través del TF Graph
 - Reescritura las TF Ops sobre el tensor global usando TF Ops equivalente sobre los tensores componentes
 - Puede implicar a inserción de directivas de comunicación donde sea necesario
 - Algunas TF Ops se puede reemplazar por versiones propias de un determinado backend

Ejecución sobre DTensors

- La ejecución de Dtensors se desencadena cuando
 - Se usa un Dtensor como un operando de una función de Python
 - Ej: `tf.matmul(a,b)`
 - Solicitar que el resultado de una función de Python se proporcione en forma de Dtensor
 - Ej: `dtensor.call_with_layout(tf.ones,layout,shape=(3,2))`
*Solicita que la salida de la función `tf.ones` se distribuya de acuerdo a un **layout***

Dtensor como operando (caso I)

```
mesh = dtensor.create_mesh(["x", 6]), devices=DEVICES)
layout = dtensor.Layout([dtensor.UNSHARDED, dtensor.UNSHARDED], mesh)
a = dtensor_from_array([[1, 2, 3], [4, 5, 6]], layout=layout)
b = dtensor_from_array([[6, 5], [4, 3], [2, 1]], layout=layout)

c = tf.matmul(a, b) # runs 6 identical matmuls in parallel on 6 devices

# `c` is a DTensor replicated on all devices (same as `a` and `b`)
print('Sharding spec:', dtensor.fetch_layout(c).sharding_specs)
print("components:")
for component_tensor in dtensor.unpack(c):
    print(component_tensor.device, component_tensor.numpy())
```

Dtensor como operando

```
Sharding spec: ['unsharded', 'unsharded']  
components:  
/job:localhost/replica:0/task:0/device:CPU:0 [[20 14]  
[56 41]]  
/job:localhost/replica:0/task:0/device:CPU:1 [[20 14]  
[56 41]]  
/job:localhost/replica:0/task:0/device:CPU:2 [[20 14]  
[56 41]]  
/job:localhost/replica:0/task:0/device:CPU:3 [[20 14]  
[56 41]]  
/job:localhost/replica:0/task:0/device:CPU:4 [[20 14]  
[56 41]]  
/job:localhost/replica:0/task:0/device:CPU:5 [[20 14]  
[56 41]]
```

Dtensor como operando (caso II)

```
mesh = dtensor.create_mesh([("x", 3), ("y", 2)], devices=DEVICES)
a_layout = dtensor.Layout([dtensor.UNSHARDED, 'x'], mesh)
a = dtensor_from_array([[1, 2, 3], [4, 5, 6]], layout=a_layout)
b_layout = dtensor.Layout(['x', dtensor.UNSHARDED], mesh)
b = dtensor_from_array([[6, 5], [4, 3], [2, 1]], layout=b_layout)

c = tf.matmul(a, b)
# `c` is a DTensor replicated on all devices (same as `a` and `b`)
print('Sharding spec:', dtensor.fetch_layout(c).sharding_specs)
```

```
Sharding spec: ['unsharded', 'unsharded']
```


Dtensor como operando (caso III)

```
mesh = dtensor.create_mesh([("x", 3), ("y", 2)], devices=DEVICES)

a_layout = dtensor.Layout(['y', 'x'], mesh)
a = dtensor_from_array([[1, 2, 3], [4, 5, 6]], layout=a_layout)
b_layout = dtensor.Layout(['x', dtensor.UNSHARDED], mesh)
b = dtensor_from_array([[6, 5], [4, 3], [2, 1]], layout=b_layout)

c = tf.matmul(a, b)
# The sharding of `a` on the first axis is carried to `c`
print('Sharding spec:', dtensor.fetch_layout(c).sharding_specs)
print("components:")
for component_tensor in dtensor.unpack(c):
    print(component_tensor.device, component_tensor.numpy())
```

```
Sharding spec: ['y', 'unsharded']
components:
/job:localhost/replica:0/task:0/device:CPU:0 [[20 14]]
/job:localhost/replica:0/task:0/device:CPU:1 [[56 41]]
/job:localhost/replica:0/task:0/device:CPU:2 [[20 14]]
/job:localhost/replica:0/task:0/device:CPU:3 [[56 41]]
/job:localhost/replica:0/task:0/device:CPU:4 [[20 14]]
/job:localhost/replica:0/task:0/device:CPU:5 [[56 41]]
```

Dtensor como salida

- Hay funciones que no reciben operandos como tensores pero devuelven un Tensor que puede ser distribuido (ejemplos: `tf.ones`, `tf.zeros`, `tf.random.stateless_normal`)
- Para ellas, existe una función llamada `dtensor.call_with_layout` que ejecuta una función Python generando un Dtensor que sigue un Layout especificado

`call_with_layout(función, layout)`

```
mesh = dtensor.create_mesh([("x", 3), ("y", 2)], devices=DEVICES)
ones = dtensor.call_with_layout(tf.ones, dtensor.Layout(['x',
'y'], mesh), shape=(6, 4))
print(ones)
```

Dtensor como salida

- Si la función emite múltiples TF Ops, se debe convertir primero a una única operación usando `tf.function`
 - Ejemplo `tf.random.stateless_normal`

```
ones=tensor.call_with_layout(tf.function(tf.random.stateless_normal),  
dtensor.Layout(['x', 'y'], mesh),  
shape=(6, 4),  
seed=(1, 1))  
print(ones)
```

Ejemplo: DTENSOR

https://github.com/diegoandradecanosa/Cesga2023Courses/tree/main/tf_dist/TF/006DTENSORS

Elementos comunes frameworks distribuidos ML

- Normalmente deben partir de una configuración de los dispositivos y/o nodos disponibles
 - En formato JSON, XML o algún tipo de estructura de Python similar (diccionarios/listas)
 - Puede definir roles, tipo ps (parameter server) o worker
 - Normalmente se da la IP de cada trabajador o un identificador válido en la LAN (tipo compute206-1 en FT3)
 - También es necesario conocer el puerto de cada nodo/dispositivo
 - A menudo se puede especificar una tecnología para las comunicaciones entre nodos nccl, mpi, gloo, etc...

Elementos comunes frameworks distribuidos ML

- La interacción con SLURM en el caso de FT3 es fundamental
 - Los nodos/dispositivos entre los que se distribuirá una computación son reservados a través del sistema de colas de SLURM
 - En cada nodo la información que necesita el framework de ML distribuido va a estar disponible a través de variables de entornos
 - O en slurm con llamadas al comando scontrol
 - A veces vamos a tener wrappers en el propio API del framework que nos van a facilitar el trabajo (tipo ClusterResolver de TF y sus variantes específicas para un gestor de colas, ej. SlurmClusterResolver)